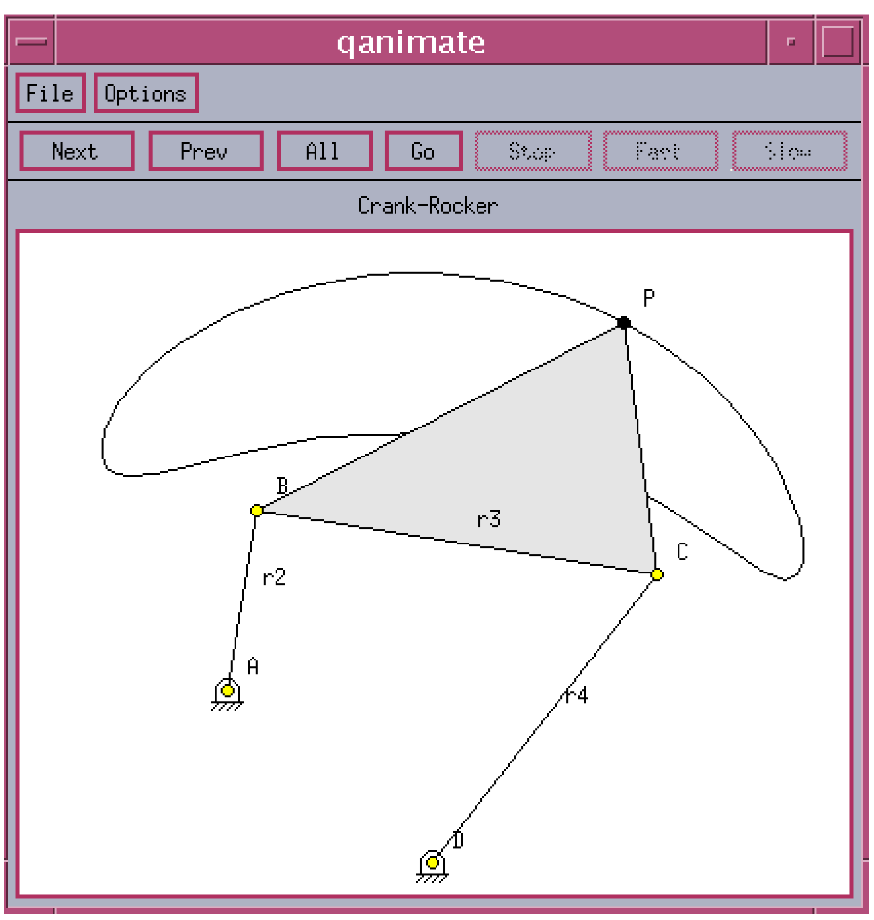
Ch Mechanism Toolkit

Version 2.1

User's Guide



How to Contact SoftIntegration

Mail	SoftIntegration, Inc. 216 F Street, #68 Davis, CA 95616
Phone	+ 1 530 297 7398
Fax	+ 1 530 297 7392
Web	http://www.softintegration.com
Email	info@softintegration.com

Copyright ©2004-2007 by SoftIntegration, Inc. All rights reserved.
November, 2007

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the copyright holder.

SoftIntegration, Inc. is the holder of the copyright to the Ch Mechanism Toolkit described in this document. **SoftIntegration, Inc. makes no representations, expressed or implied, with respect to this documentation, or the software it describes, including without limitations, any implied warranty merchantability or fitness for a particular purpose, all of which are expressly disclaimed. Users should be aware that included in the terms and conditions under which SoftIntegration is willing to license the Ch Mechanism Toolkit as a provision that SoftIntegration, and their distribution licensees, distributors and dealers shall in no event be liable for any indirect, incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of the Ch Mechanism Toolkit, and that liability for direct damages shall be limited to the amount of purchase price paid for the Ch Mechanism Toolkit.**

Ch, SoftIntegration, One Language for All, and Quick Animation are either registered trademarks or trademarks of SoftIntegration, Inc. in the United States and/or other countries. Microsoft, MS-DOS, Windows, Windows 95, Windows 98, Windows Me, Windows NT, Windows 2000, and Windows XP are trademarks of Microsoft Corporation. Solaris and Sun are trademarks of Sun Microsystems, Inc. Unix is a trademark of the Open Group. HP-UX is either a registered trademark or a trademark of Hewlett-Packard Co. Linux is a trademark of Linus Torvalds. All other trademarks belong to their respective holders.

Table of Contents

1 Getting Started with Ch Mechanism Toolkit **1**
1.1 Introduction 1
1.2 Features 1
1.3 Getting Started 2
1.4 Solving Complex Equations 5

2 Fourbar Linkage **7**
2.1 Position Analysis 7
2.2 Transmission Angle Analysis 15
2.3 Velocity Analysis 17
2.4 Acceleration Analysis 19
2.5 Dynamics 20
2.6 Kinematics and Dynamics with Constant Angular Velocity for Link 2 24
2.7 Three-Position Synthesis 29
2.8 Animation 32
2.9 Web-Based Fourbar Linkage Analysis 34
2.9.1 Position Analysis 36
2.9.2 Coupler Curve Plotting 36
2.9.3 Animation 36

3 Crank-Slider Mechanism **44**
3.1 Position Analysis 45
3.2 Transmission Angles 49
3.3 Velocity Analysis 49
3.4 Acceleration Analysis 51
3.5 Position, Velocity and Acceleration of Coupler Point 52
3.6 Dynamic Force Analysis 53
3.7 Animation 59
3.8 Web-Based Crank-Slider Linkage Analysis 62

4 Geared Five-Bar Linkage **77**
4.1 Position Analysis 77
4.2 Velocity Analysis 78
4.3 Acceleration Analysis 80
4.4 Coupler Point Analysis 81
4.5 Animation 81
4.6 Web-Based Geared Fivebar Linkage Analysis 84

5 Multi-Loop Six-Bar Linkages **90**
5.1 Fourbar/Slider-Crank Linkage . . . 90
5.1.1 Position Analysis . . . 90
5.1.2 Velocity Analysis . . . 93
5.1.3 Acceleration Analysis . . . 94
5.1.4 Animation . . . 97
5.1.5 Web-Based Fourbar-Slider Linkage Analysis . . . 98
5.2 Watt Six-bar (I) Linkage . . . 105
5.2.1 Position Analysis . . . 106
5.2.2 Velocity Analysis . . . 106
5.2.3 Acceleration Analysis . . . 107
5.2.4 Coupler Position, Velocity, and Acceleration . . . 109
5.2.5 Animation . . . 111
5.2.6 Web-Based Analysis . . . 114
5.3 Watt Six-bar (II) Linkage . . . 122
5.3.1 Position Analysis . . . 122
5.3.2 Velocity Analysis . . . 123
5.3.3 Acceleration Analysis . . . 124
5.3.4 Coupler Point Position, Velocity, and Acceleration . . . 126
5.3.5 Input/Output Ranges . . . 130
5.3.6 Animation . . . 131
5.3.7 Web-Based Analysis . . . 132
5.4 Stephenson Six-bar (I) Linkage . . . 139
5.4.1 Position Analysis . . . 139
5.4.2 Velocity Analysis . . . 140
5.4.3 Acceleration Analysis . . . 141
5.4.4 Coupler Point Position, Velocity, and Acceleration . . . 146
5.4.5 Animation . . . 146
5.4.6 Web-Based Analysis . . . 148
5.5 Stephenson Six-bar (III) Linkage . . . 149
5.5.1 Position Analysis . . . 156
5.5.2 Velocity Analysis . . . 156
5.5.3 Acceleration Analysis . . . 157
5.5.4 Coupler Point Position, Velocity, and Acceleration . . . 160
5.5.5 Animation . . . 160
5.5.6 Web-Based Analysis . . . 162

6 Cam Design **171**
6.1 Introduction to Cam Design . . . 171
6.2 Cam Design with Class CCam . . . 175
6.3 Web-Based Cam Design . . . 187

7 Quick Animation **194**
7.1 Input Data Format . . . 195
7.1.1 General Drawing Primitives . . . 196
7.1.2 Mechanical Drawing Primitives . . . 199
7.2 Quick Animation Examples . . . 201

8 Implementations of Interactive Web Pages for Mechanism Design and Analysis **214**
8.1 Introduction to CGI Programming . . . 214
8.1.1 Writing HTML Files . . . 214
8.1.2 Writing CGI Script Files . . . 217
8.2 Web-Based Animation Example . . . 219
8.2.1 Configuration and Setup of Web Servers . . . 220

9 References **230**

A Header File linkage.h **231**
linkage.h . . . 231

B Class CFourbar **233**
CFourbar . . . 233
angularAccel . . . 235
angularAccels . . . 237
angularPos . . . 238
angularPoss . . . 239
angularVel . . . 241
angularVels . . . 242
animation . . . 244
couplerCurve . . . 252
couplerPointAccel . . . 255
couplerPointPos . . . 256
couplerPointVel . . . 257
displayPosition . . . 259
displayPositions . . . 261
forceTorque . . . 261
forceTorques . . . 263
getAngle . . . 266
getJointLimits . . . 266
grashof . . . 268
plotAngularAccels . . . 269
plotAngularPoss . . . 270
plotAngularVels . . . 272
plotCouplerCurve . . . 273
plotForceTorques . . . 274
plotTransAngles . . . 275
printJointLimits . . . 276
setAngularVel . . . 277
setCouplerPoint . . . 278
setGravityCenter . . . 279
setInertia . . . 279
setLinks . . . 280
setMass . . . 280
setNumPoints . . . 281
synthesis . . . 281
transAngle . . . 283

transAngles 284
uscUnit 285

C Class CCrankSlider **287**
CCrankSlider 287
angularAccel 288
angularPos 290
angularVel 291
animation 292
couplerCurve 293
couplerPointAccel 295
couplerPointPos 296
couplerPointVel 297
displayPosition 298
forceTorque 299
forceTorques 301
getJointLimits 304
plotCouplerCurve 305
plotForceTorques 306
setCouplerPoint 308
setGravityCenter 308
setInertia 309
setAngularVel 309
setLinks 310
setMass 310
setNumPoints 311
sliderAccel 311
sliderPos 313
sliderVel 314
transAngle 315
uscUnit 316

D Class CGearedFivebar **317**
CGearedFivebar 317
angularAccel 318
angularPos 320
angularVel 322
animation 323
couplerCurve 325
couplerPointAccel 327
couplerPointPos 329
couplerPointVel 330
displayPosition 331
plotCouplerCurve 333
setCouplerPoint 335
setAngularVel 336
setLinks 336
setNumPoints 337

uscUnit 337

E Class CFourbarSlider **338**
CFourbarSlider 338
angularAccel 339
angularPos 341
angularVel 343
animation 345
couplerPointAccel 347
couplerPointPos 349
couplerPointVel 351
displayPosition 353
setCouplerPoint 354
setAngularVel 355
setLinks 356
setNumPoints 356
sliderAccel 357
sliderPos 358
sliderVel 360
uscUnit 362

F Class CWattSixbarI **363**
CWattSixbarI 363
angularAccel 364
angularPos 366
angularVel 367
animation 369
couplerPointAccel 371
couplerPointPos 373
couplerPointVel 375
displayPosition 377
setCouplerPoint 379
setAngularVel 380
setLinks 380
setNumPoints 381
uscUnit 381

G Class CWattSixbarII **382**
CWattSixbarII 382
angularAccel 383
angularPos 385
angularVel 387
animation 390
couplerPointAccel 392
couplerPointPos 394
couplerPointVel 395
displayPosition 395
getIORanges 397

setCouplerPoint 399
setAngularVel 399
setLinks 400
setNumPoints 400
uscUnit 401

H Class CStevSixbarI **402**
CStevSixbarI 402
angularAccel 403
angularPos 405
angularVel 407
animation 409
couplerPointAccel 411
couplerPointPos 413
couplerPointVel 415
displayPosition 417
setCouplerPoint 419
setAngularVel 420
setLinks 420
setNumPoints 421
uscUnit 421

I Class CStevSixbarIII **423**
CStevSixbarIII 423
angularAccel 424
angularPos 426
angularVel 427
animation 429
couplerPointAccel 431
couplerPointPos 433
couplerPointVel 434
displayPosition 436
setCouplerPoint 438
setAngularVel 439
setLinks 439
setNumPoints 440
uscUnit 440

J Class CCam **442**
CCam 442
addSection 444
angularVel 444
animation 446
baseRadius 447
CNCCode 448
cutDepth 449
cutter 449
cutterOffset 450

deleteCam . . . 450
feedrate . . . 451
followerType . . . 451
getCamAngle . . . 456
getCamProfile . . . 457
getFollowerAccel . . . 458
getFollowerPos . . . 459
getFollowerVel . . . 461
makeCam . . . 462
plotCamProfile . . . 467
plotFollowerAccel . . . 468
plotFollowerPos . . . 469
plotFollowerVel . . . 470
plotTransAngle . . . 471
spindleSpeed . . . 473
cam.ch . . . 474

K Solving Complex Equations **478**

Index **481**

Chapter 1

Getting Started with Ch Mechanism Toolkit

Note: The source code for all examples described in this document are available in CHHOME/toolkit/demos/mechanism. CHHOME is the directory where Ch is installed. It is recommended that you try these examples while reading this document.

1.1 Introduction

Mechanism design is an intriguing subject, through which one can gain some experience and physical appreciation of mechanical design. Using this Ch Mechanism Toolkit, one can use its high-level building blocks to conveniently build their own software programs to solve complicated practical engineering analysis and design problems. It can also be used to develop software for Web-based design and analysis of complicated mechanisms.

This Mechanism Toolkit is developed in Ch. Ch is an interpreter which provides a superset of C. Ch is object-based with classes in C++. Unlike other mathematical software packages, Ch conforms to the open C/C++ standards. Programming features such as complex numbers, pass-by-reference, and computational arrays are very useful for engineering applications. These features are simple and easy to comprehend by users who have only a first course in computer programming. Ch is the simplest possible solution for 2D/3D graphical plotting and numerical computing in the domain of C/C++. Therefore, Ch is ideal for development of mechanism toolkit which uses graphical plotting and numerical computing features extensively.

Ch is a very high-level language environment. Ch programs are created not by writing large programs starting from scratch. Instead, they are combined by relatively small components. These components are small and concentrate on simple tasks so that they are easy to build, understand, describe and maintain. In this documentation, we will describe how the mechanism toolkit can be used as building blocks for analysis and design of closed-loop planar mechanisms including four-bar, five-bar, and six-bar liankges as well as cam mechanism. Although the presentation is focused on these commonly used planar mechanisms, the ideas presented, however, are applicable to other complicated planar mechanisms as well. A user can either write a computer program to solve problems in analysis and design of mechanisms, or use a web browser to solve the problem interactively through the internet.

1.2 Features

Ch Mechanism Toolkit has the following salient features.

1. **Variety of Mechanisms**
   Contain classes for design and analysis of four-bar, five-bar, six-bar linkages including fourbar/crank-

slider, Watt six-bar, Stephenson six-bar, and cam-follower mechanism. Follow the examples of the source code for these mechanisms, users can develop their own software for analysis and design of other mechanisms.

2. **Kinematic Analysis**
   Perform position, velocity, acceleration analysis for joint angles and coupler points.

3. **Synthesis**
   Perform synthsis of mechanisms.

4. **Dynamic Analysis**
   Perform dynamic analysis based on equations of motion.

5. **Animation**
   Perform animation of mechanisms either in a local machine or through the internet. Easily build animation for other mechanisms.

6. **Web-Based**
   Design and analysis of commonly used mechanisms such as fourbar, fivebar, sixbar, and cam mechanisms can be performed through the internet using a Web brower, without any programming. The user can develop other web-based applications easily using this mechanism toolkit.

7. **Plotting Utilities**
   Provide many plotting functions to allow output visually displayed or exported as external files with a variety of different file formats including postscript file, PNG, LaTeX, etc. They can readily be copied and pasted in other applications such as Word.

8. **C/C++ Compatible**
   Different from other software packages, programs written in Ch Mechansim Toolkit can work with existing C/C++ programs and libraries seamlessly.

9. **Object-Oriented**
   Implemented as classes for commonly used mechanisms, Ch Mechanism Toolkit is object-oriented.

10. **Embeddable**
    With Embedded Ch, Ch programs using Mechanism Toolkit can be embedded in other C/C++ application programs.

1.3 Getting Started

To help users to get familiar with Ch Mechanism Toolkit, a sample program will be used to illustrate basic features and applications of Ch Mechanism Toolkit. In this example, a four-bar linkage is shown in Figure 1.1. Link lengths of the linkage are given as $r_1 = 12cm, r_2 = 4cm, r_3 = 10cm, r_4 = 7cm$. The phase angle for the ground link is $\theta_1 = 0$, the coupler point P is defined by the distance $r_p = 5cm$ and constant angle $\beta = 20°$. This is a crank-rocker four-bar linkage. A branch of coupler curves for the coupler point will be plotted and animation of the linkage will be created.

A Ch program for solving this problem is shown in Program 1. The first line of program

```
#include <fourbar.h>
```

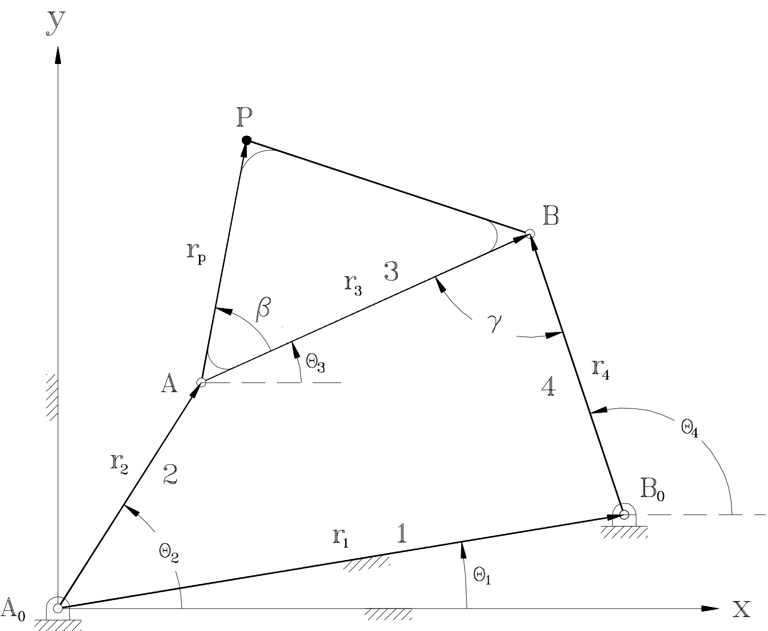


Figure 1.1: The four-bar linkage.

```
#include <fourbar.h>

int main() {
    /* specify a crank-rocker four-bar linkage */
    double r1 = 0.12, r2 = 0.04, r3 = 0.10, r4= 0.07;
    double theta1 = 0;
    double rp = 0.05, beta = 20*M_PI/180;
    int branchnum = 1;
    class CPlot plot;
    class CFourbar fourbar;

    fourbar.setLinks(r1, r2, r3, r4, theta1);
    fourbar.setCouplerPoint(rp, beta);
    fourbar.plotCouplerCurve(&plot,branchnum);
    fourbar.animation(branchnum);
}
```

Program 1: A sample program for plotting a coupler curve and animation of a crank-rocker four-bar linkage.

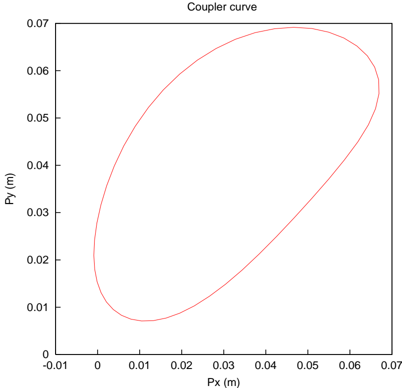


Figure 1.2: A coupler curve for a crank-rocker four-bar linkage.

includes the header file **fourbar.h** which defines the class **CFourbar**, macros, and prototypes of member functions. Like a C/C++ program, a Ch program will start to execute at the **main**() function after the program is parsed. The next three lines

```
double r1 = 0.12, r2 = 0.04, r3 = 0.10, r4= 0.07;
double theta1 = 0;
double rp = 0.05, beta = 20*M_PI/180;
```

define the four-linkage and coupler point. Note that the link lengths are specified in meters. The macro `M_PI` for π is defined in the header file **math.h** which is included in the header file **fourbar.h**. For a crank-rocker four-bar linkage, there are two circuits or branches. The branch number is selected in the program by integer variable `branchnum`. Line

```
class CPlot plot;
```

defines a class **CPlot** for creating and manipulating two and three dimensional plots. The **CPlot** class is defined in **chplot.h** which is included in **fourbar.h** header file. Line

```
class CFourbar fourbar;
```

constructs an object of four-bar linkage. The keyword **class** is optional. Lines

```
fourbar.setLinks(r1, r2, r3, r4, theta1);
fourbar.setCouplerPoint(rp, beta);
```

specify the demensions of the four-bar linkage. The member function `setLinks()` has five arguments. The first four arguments specify the link lengths and the fifth one is the phase angle for link 1. The member function `setCouplerPoint()` specifies a coupler point with two arguments, the first one for distance and the second one for the phase angle as shown in Figure 1.1. Line

```
fourbar.plotCouplerCurve(&plot,branchnum);
```

computes and plots the coupler curve for the branch specified in the second argument. Member function **plotCouplerCurve**() has two arguments. The first argument is a pointer to an existing object of class **CPlot**. The second argument is the branch number of the linkage. The coupler curve is displayed in Figure 1.2 when Program 1 is executed. Line

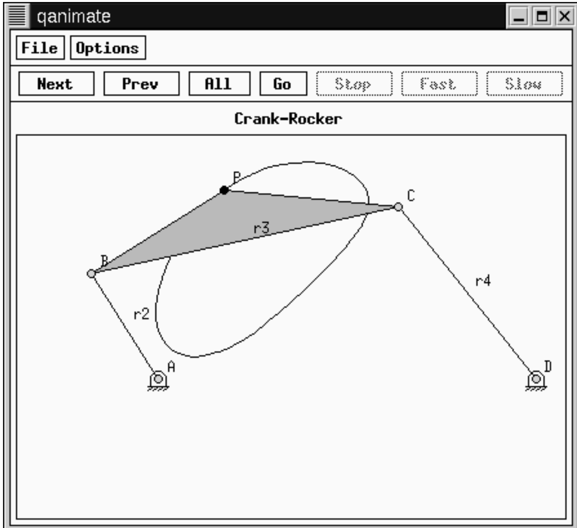


Figure 1.3: The animation for a crank-rocker four-bar linkage.

```
fourbar.animation(branchnum);
```

creates an animation of the four-bar linkage for the branch specified in its argument. The animation is shown in Figure 1.3 when Program 1 is executed. The animation is performed by the QuickAnimation™ program `qanimate` for quick animation. The menu bar in the qanimate window, as shown in Figure 1.3, contains two menus, `File` and `Options`, and a series of buttons which manipulate the mechanism. The `File` menu allows one to quit the program and the `Options` menu allows one to change various display settings. The `Next` and `Prev` buttons control the mechanism's position, and the `All` button displays all mechanism positions at once. The `Fast` and `Slow` buttons change the speed of animation. The `Go` and `Stop` buttons start and stop animation, respectively. The mechanism can move in either direction by pressing the `Prev` button for one direction and the `Next` button for the opposite direction. When the `Go` button is pressed, the mechanism will move in the direction previously assigned by the `Prev` or `Next` button.

1.4 Solving Complex Equations

Complex numbers are used for analysis and design of planar linkages in Ch Mechanism Toolkit. Many analysis and design problems for planar linkages need to solve a complex equation. A complex equation can be expressed in a general polar form of

$$R_1 e^{i\phi_1} + R_2 e^{i\phi_2} = z_3 \tag{1.1}$$

where z_3 can be expressed in either Cartesian coordinates $x_3 + iy_3$ as `complex(x3, y3)`, or polar coordinates $R_3 e^{i\phi_3}$ as `polar(R3, phi3)`. Many analysis and design problems for planar mechanisms can be formulated in this form. Because a complex equation can be partitioned into real and imaginary parts, two unknowns out of four parameters R_1, ϕ_1, R_2, and ϕ_2 can be solved in this equation. Details of derivation for solutions of a complex equation are given in Appendix K.

Function `complexsolve()` in Ch can be conveniently used to solve the complex equation (1.1). The function prototype for the function `complexsolve()` is given below,

```
int complexsolve(int n1, int n2, double firstknown,
                 double secondknown,  double complex z,
```

```
#include <numeric.h>
#include <complex.h>

int main () {
  double x1, x2, x3, x4;
  int n1, n2, num;
  double firstknown, secondknown;
  double complex z3;

  firstknown = 3;
  secondknown = 4;
  z3 = complex(1,2);
  n1 = 2;
  n2 = 4;
  num = complexsolve(n1,n2,firstknown,secondknown,z3,x1,x2,x3,x4);
  printf("For n1=2, n2=4, the number of solution is = %d\n", num);
  printf("phi1 = %f phi2 = %f\n", x1, x2);
  printf("phi1 = %f phi2 = %f\n", x3, x4);
}
```

Program 2: Solve for ϕ_1 and ϕ_2 for a general complex equation.

```
double &x1, double &x2, double &x3, double &x4)
```

where parameter `n1` is the position of the first of two unknowns to be obtained on the left-hand side of equation (1.1) , `n2` is the position of the second of two unknowns, `firstknown` is the value of the first known on the left-hand side of equation (1.1), `secondknown` is the value of the second known, and `z` is the complex number on the right-hand side of equation (1.1). The arugment x1 of double data passed back from the function by reference gives the result of the the first unknown value. whereas x2 gives the result of the second unknown in equation (1.1). If either ϕ_1 or ϕ_2 is to be found, there are two sets of solutions for equation (1.1). In this case, the x3 and x4 give the second set of solutions for the first and second unknowns, respectively. The function returns the number of set of solutions with a value of 0, 1, or 2.

For example, two unknowns ϕ_1 and ϕ_2 in equation $3.5e^{i\phi_1} + 4.5e^{i\phi_2} = e^i + 3e^{i4}$ can be solved by Program 2. The output from Program 2 is given below.

```
For n1=2, n2=4, the number of solution is = 2
phi1 = -0.613277 phi2 = 1.942631
phi1 = 2.827574 phi2 = 0.271667
```

Chapter 2

Fourbar Linkage

The four-bar linkage, as shown in Figure 2.1, is the simplest closed-loop linkage. This section will describe how to perform kinematic and dynamic analysis of four-bar linkages using the Ch Mechanism Toolkit.

2.1 Position Analysis

For the four-bar linkage shown in Figure 2.1, the displacement analysis can be formulated by the following loop-closure equation

$$\mathbf{r}_2 + \mathbf{r}_3 = \mathbf{r}_1 + \mathbf{r}_4. \tag{2.1}$$

Using complex numbers, equation (2.1) becomes

$$r_2 e^{i\theta_2} + r_3 e^{i\theta_3} = r_1 e^{i\theta_1} + r_4 e^{i\theta_4}, \tag{2.2}$$

where link lengths r_1, r_2, r_3, and r_4 and angular position θ_1 for the ground link are constants. Angle θ_2 for the input link is an independent variable; angles θ_3 and θ_4 for the coupler and output links, respectively, are dependent variables. Equation (2.2) can be rearranged as

$$r_3 e^{i\theta_3} - r_4 e^{i\theta_4} = r_1 e^{i\theta_1} - r_2 e^{i\theta_2}. \tag{2.3}$$

Let $R_1 = r_3, \phi_1 = \theta_3, R_2 = -r_4, \phi_2 = \theta_4, z = (x_3, y_3) = r_1 e^{i\theta_1} - r_2 e^{i\theta_2}$, equation (2.3) becomes the following general complex equation

$$R_1 e^{i\phi_1} + R_2 e^{i\phi_2} = z. \tag{2.4}$$

Given link lengths of a four-bar linkage and angles θ_1 and θ_2, the angular positions θ_3 and θ_4 can be solved using function `complexsolve()` described in section 1.4. In general, there are two sets of solutions for θ_3 and θ_4 for a given θ_2, which correspond to two different circuits or two geometric inversions of a circuit of the four-bar linkage [3]. A Non-Grashof linkage has one circuit with two geometric inversions. A Grashof Crank-Rocker or Crank-Crank linkage has two circuits, each having only one geometric inversion. However, a Grashof Rocker-Crank or Rocker-Rocker linkage has two circuits, each with two geometric inversions.

Once the joint angle for θ_3 is obtained, the position of coupler point P shown in Figure 2.1 can be obtained. The position vector **P** for coupler point P can be expressed in vector form using complex numbers as:

$$P = r_2 e^{i\theta_2} + r_p e^{i(\theta_3 + \beta)} \tag{2.5}$$

A complex number $z = (x, y) = re^{i\theta}$ in Ch can be constructed either by `complex(x,y)` or `polar(r,theta)`. Equation (2.5) can be translated into a Ch programming statement

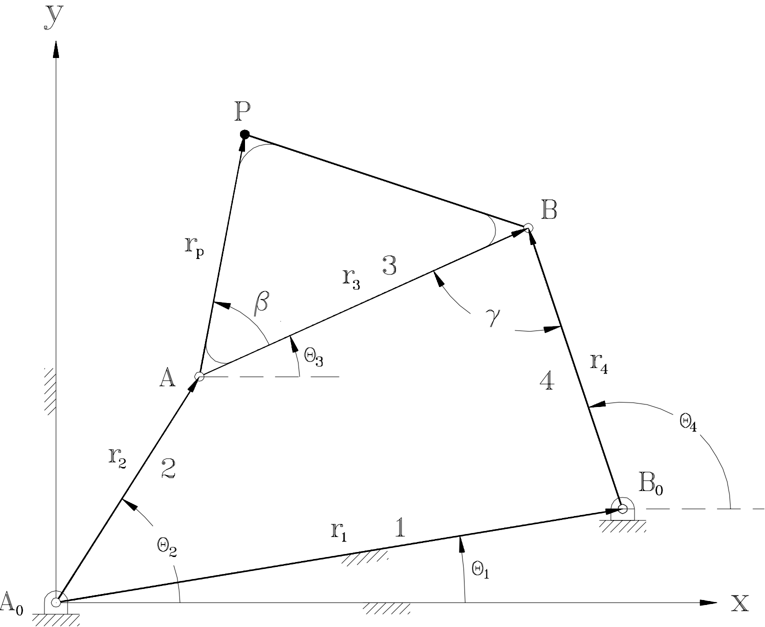


Figure 2.1: The four-bar linkage.

```
P = polar(r2,theta2) + polar(rp,theta3+beta).
```

Using class `CFourbar`, a four-bar linkage analysis problem can be solved conveniently, which can be illustrated by the following analysis problem.

> **Problem 1:** Link lengths of a four-bar linkage, as shown in Figure 2.1, are given as follows: $r_1 = 12cm, r_2 = 4cm, r_3 = 12cm, r_4 = 7cm$. The phase angle for the ground link is $\theta_1 = 10°$, the coupler point P is defined by the distance $r_p = 5cm$ and constant angle $\beta = 20°$. Find the angular positions θ_3 and θ_4 and the position for coupler point P when the input angle θ_2 is 70°. Display the current position of the fourbar mechanism.

The four-bar linkage given in Problem 1 is a crank-rocker. There are two distinct circuits for each input link position. The class `CFourbar` can be used to solve the analysis and design problems related to the four-bar linkage as shown in Program 3. Two sets of solutions for angles θ_3 and θ_4 as well as the position vector for coupler point P are calculated by the member functions `angularPos()` and `couplerPointPos()`, respectively. Arrays in Ch are fully ISO C compatible, they are pointers. For consistency with text description, we use arrays with index starting with 1, instead of 0, in the mechanism toolkit. The output of Program 3 is shown below:

```
theta3 =  0.459, theta4 =  1.527, P = complex( 4.822, 7.374) cm
theta3 = -0.777, theta4 = -1.845, P = complex( 5.917, 1.684) cm
```

Member function `displayPosition()` is called to graphically display the current position of the fourbar linkage. It is prototyped as follows,

```
int CFourbar::displayPosition(double theta2, double theta3,
                    double theta4, ...
                   /*[int outputtype [, char *filename]]*/);
```

```
#include <math.h>
#include <fourbar.h>
int main() {
  CFourbar fourbar;
  double r1 = 0.12, r2 = 0.04, r3 =0.12, r4 = 0.07,
         theta1 = 10*M_PI/180;
  double rp = 0.05, beta = 20*M_PI/180;
  double theta_1[1:4], theta_2[1:4];
  double complex p1, p2;  // two solution of coupler point P
  double theta2 = 70*M_PI/180;

  theta_1[1] = theta1;
  theta_1[2] = theta2; // theta2
  theta_2[1] = theta1;
  theta_2[2] = theta2; // theta2
  fourbar.uscUnit(false);
  fourbar.setLinks(r1, r2, r3, r4, theta1);
  fourbar.setCouplerPoint(rp, beta);
  fourbar.angularPos(theta_1, theta_2, FOURBAR_LINK2);
  fourbar.couplerPointPos(theta2, p1, p2);
  fourbar.displayPosition(theta2, theta_1[3], theta_1[4]);
  fourbar.displayPosition(theta2, theta_2[3], theta_2[4]);

  /**** the first set of solutions ****/
  printf("theta3 = %6.3f, theta4 = %6.3f, P = %6.3f cm\n",
         theta_1[3], theta_1[4], p1*100);
  /**** the second set of solutions ****/
  printf("theta3 = %6.3f, theta4 = %6.3f, P = %6.3f cm\n",
         theta_2[3], theta_2[4], p2*100);
}
```

Program 3: Program for computing θ_3, θ_4 and position of the coupler point P of a four-bar linkage using class `CFourbar`.

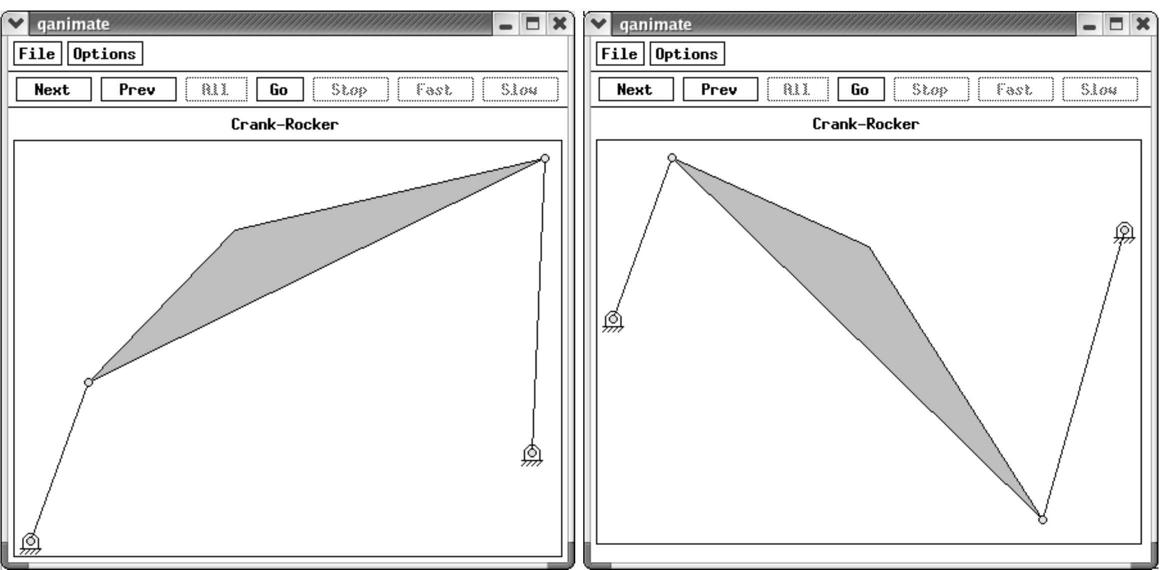


Figure 2.2: Current positions of the fourbar linkage.

where `theta2`, `theta3`, and `theta4` are the current angular positions. It utilizes the QuickAnimation™ program `qanimate` to display the fourbar mechanism. QuickAnimation™ will be discussed in further detail in Chapter 7. Two optional arguments may be inputted into member function `displayPosition()` after argument `theta4`. The value of the first optional argument may be either macro `QANIMATE_OUTPUTTYPE_DISPLAY`, `QANIMATE_OUTPUTTYPE_FILE`, and `QANIMATE_OUTPUTTYPE_STREAM`. Macro `QANIMATE_OUTPUTTYPE_DISPLAY` displays the fourbar figure on the computer terminal, whereas macro `QANIMATE_OUTPUTTYPE_STREAM` sends the `qanimate` data to the standard output stream. With macro `QANIMATE_OUTPUTTYPE_FILE`, the `qanimate` data can be saved to a file with file name specified by the second optional input argument. The default output value is `QANIMATE_OUTPUTTYPE_DISPLAY`. Figure 2.2 shows the current position for both kinematic inversions of the fourbar linkage described in Problem 1. Note that fourbar linkage analysis can be performed in either SI of US Customary units with class `CFourbar`. Member function `uscUnit()` is called prior to any of the other member functions to specify whether US Customary units are desired. The function prototype for member function `uscUnit()` is as follows,

```
void CFourbar::uscUnit(bool unit);
```

where the value of argument `unit` is either false or true. If `unit` is `true`, then the input and output values are assumed to be in US Customary units. In this case, the length, time, force, and mass shall be specified in foot, second, pound, and slug ($lb - sec^2/ft$), respectively. If the value of `unit` is `false`, then the default SI units are assumed. In this case, the length, time, force, and mass shall be specified in meter, second, Newton, and kilogram, respectively. For example, the lengths, r_1 to r_4 for the linkage in Program 3 are specified in meter.

Problem 2: Link lengths of a four-bar linkage, as shown in Figure 2.1, are given as follows: $r_1 = 4.72in$, $r_2 = 1.57in$, $r_3 = 4.72in$, $r_4 = 2.76in$. The phase angle for the ground link is $\theta_1 = 10^o$, the coupler point P is defined by the distance $r_p = 1.97in$ and constant angle $\beta = 20^o$. Find the angular positions θ_3 and θ_4 and the position for coupler point P when the input angle θ_2 is 70°.

```
#include <math.h>
#include <fourbar.h>
int main() {
  CFourbar fourbar;
  double r1 = 4.72/12.0, r2 = 1.57/12.0, r3 = 4.72/12.0, r4 = 2.76/12.0,
         theta1 = 10*M_PI/180;
  double rp = 1.97/12.0, beta = 20*M_PI/180;
  double theta_1[1:4], theta_2[1:4];
  double complex p1, p2;  // two solution of coupler point P
  double theta2 = 70*M_PI/180;

  theta_1[1] = theta1;
  theta_1[2] = theta2; // theta2
  theta_2[1] = theta1;
  theta_2[2] = theta2; // theta2
  fourbar.uscUnit(true);
  fourbar.setLinks(r1, r2, r3, r4, theta1);
  fourbar.setCouplerPoint(rp, beta);
  fourbar.angularPos(theta_1, theta_2, FOURBAR_LINK2);
  fourbar.couplerPointPos(theta2, p1, p2);

  /**** the first set of solutions ****/
  printf("theta3 = %6.3f, theta4 = %6.3f, P = %6.3f in\n",
         theta_1[3], theta_1[4], p1*12.0);
  /**** the second set of solutions ****/
  printf("theta3 = %6.3f, theta4 = %6.3f, P = %6.3f in\n",
         theta_2[3], theta_2[4], p2*12.0);
}
```

Program 4: Program for computing θ_3, θ_4 and position of the coupler point P of a four-bar linkage using US Customary units.

As another example, consider Problem 2 above, which requires the use of US Customary units for analysis. Problem 2 is similar to Problem 1, except that the link lengths are specified in inches rather than centimeters. The solution to this problem is Program 4. In this program, the link lengths are converted from inches to the desired unit of feet by multiplying a coefficient of $\frac{1}{12}$. In contrast to Program 3, the argument of member function `uscUnit()` is `true` to indicate that US Customary units are desired for the results of the fourbar analysis. The output of Program 4 is as follows:

```
theta3 =  0.462, theta4 =  1.529, P = complex( 1.894, 2.903) in
theta3 = -0.778, theta4 = -1.845, P = complex( 2.329, 0.656) in
```

Alternatively, fuunction `complexsolve()` can be used to solve the analysis and design problems related to the four-bar linkage as shown in Program 3. Problem 1 can be solved by using Program 5. Two sets of solutions for angles θ_3 and θ_4 as well as the position vector for coupler point P are calculated by Program 5. The numerical output from Programs 3 and 5 are the same. Note that since Program 5 does not use class `CFourbar` to solve Problem 1, the link dimensions may be specified in centimeters rather than meters. However, the value for the coupler point position is in centimeters as well.

According to the IEEE 754 standard for binary floating-point arithmetic, any invalid solution in Ch is symbolically represented as NaN. This can be very useful for analysis of mechanisms. For example, if the link dimensions for the four-bar linkage in Problem 1 are changed to $r_1 = 12cm, r_2 = 12cm, r_3 = 4cm, r_4 = 7cm$. The linkage then becomes a double-rocker. There are two circuits, each with two geometric

```
#include <numeric.h>

int main() {
  double r[1:4], theta[1:4], rp, beta;
  int n1 = 2,  n2 = 4;               /* positions of two unknowns */
  double complex z, P;
  double x1, x2, x3, x4;

  /* specification of the four-bar linkage */
  r[1] = 12; r[2] = 4; r[3] = 12; r[4] = 7; rp = 5; beta = 20*M_PI/180;
  theta[1] = 10*M_PI/180; theta[2]=70*M_PI/180;
  z = polar(r[1], theta[1]) - polar(r[2], theta[2]);      /* z = r1-r2 */
  complexsolve(n1, n2, r[3], -r[4], z, x1, x2, x3, x4);
  /**** the first set of solutions ****/
  theta[3] = x1;  theta[4] = x2;
  P = polar(r[2], theta[2]) + polar(rp, theta[3]+beta); /* P=r2+rp */
  printf("theta3 = %6.3f, theta4 = %6.3f, P = %6.3f \n", theta[3], theta[4], P);
  /**** the second set of solutions ****/
  theta[3] = x3; theta[4] = x4;
  P = polar(r[2], theta[2]) + polar(rp, theta[3]+beta); /* P=r2+rp */
  printf("theta3 = %6.3f, theta4 = %6.3f, P = %6.3f \n", theta[3], theta[4], P);
}
```

Program 5: Program for computing θ_3, θ_4 and position of the coupler point P of a four-bar linkage.

inversions, for this linkage. The input ranges for two separate circuits are $24.36° < \theta_2 < 64.56°$ and $315.44° < \theta_2 < 355.64°$. When the input angle θ_2 is set to $70°$, there exist no solutions for θ_3 and θ_4 . This can be gracefully handled in a Ch program. If the following programming statement

```
r[1] = 12; r[2] = 4; r[3] = 12; r[4] = 7;
```

in Program 5 is changed to

```
r[1] = 12; r[2] = 12; r[3] = 4; r[4] = 7;
```

the output from the program becomes
theta3 = NaN, theta4 = NaN, P = complex(NaN,NaN)
theta3 = NaN, theta4 = NaN, P = complex(NaN, NaN)

For motion analysis of a crank-rocker mechanism using Program 6, the output range of the rocker is within $0 \sim 2\pi$. For this mechanism, the output may be as shown on the top part in Figure 2.3. There is a jump when θ_4 is π, because $\theta_4 = \pi$ and $\theta_4 = -\pi$ are the same point for the crank-rocker mechanism. If the **unwrap**() function is used, a smooth curve for output angle θ_4 can be obtained as shown on the lower part in Figure 2.3.

Function **unwrap**() with the prototype of

```
int unwrap(array double &y, array double &x, ...
           /* [double cutoff] */);
```

in the header file `numeric.h` unwraps the radian phase of each element of input array *x* by changing its absolute jump greater than π to its 2π complement. The input array *x* can be of a vector or a two-dimensional array. If it is a two-dimensional array, the function unwraps it through every row of the array. Array argument *y* is the same dimension and size as *x*. It contains the unwrapped data. Optional argument *cutoff* specifies the jump value. If the user does not specify this input, *cutoff* has a value of π by default. Function **unwrap** returns 0 on success and -1 on failure. The details about this function can be found in the *Ch Reference Guide*.

```
#include <numeric.h>
#include <chplot.h>

int main(){
    double r[1:4],theta1,theta31;
    int n1=2,n2=4, i;
    double complex z,p,rb;
    double  x1,x2,x3,x4;
    array double theta2[36],theta4[36],theta41[36];
    class CPlot subplot, *plot;

    /* four-bar linkage*/
    r[1]=5; r[2]=1.5; r[3]=3.5; r[4]=4;
    theta1=30*M_PI/180;
    linspace(theta2,0,2*M_PI);
    for (i=0;i<36;i++) {
       z=polar(r[1],theta1)-polar(r[2],theta2[i]);
       complexsolve(n1,n2,r[3],-r[4],z,x1,x2,x3,x4);
       theta4[i] = x2;
    }
    unwrap(theta41, theta4);
    subplot.subplot(2,1);
    plot = subplot.getSubplot(0,0);
    plot->data2D(theta2, theta4);
    plot->title("Wrapped");
    plot->label(PLOT_AXIS_X,"Crank input: radians");
    plot->label(PLOT_AXIS_Y,"Rocker output: radians");

    plot = subplot.getSubplot(1,0);
    plot->data2D(theta2, theta41);
    plot->title("Unwrapped");
    plot->label(PLOT_AXIS_X,"Crank input: radians");
    plot->label(PLOT_AXIS_Y,"Rocker output: radians");
    subplot.plotting();
}
```

Program 6: A program using **unwrap**().

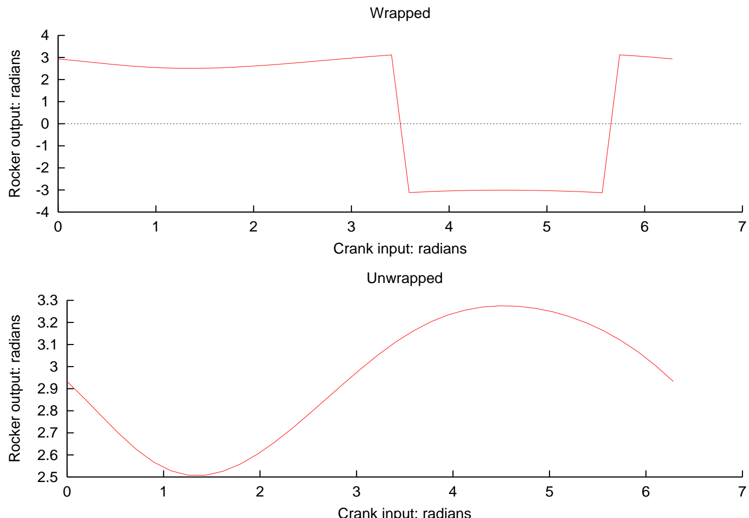


Figure 2.3: Comparison of results with and without using **unwrap**() function.

A four-bar linkage may take form of a crank-rocker, double-crank (drag-link), double-rocker, or triple-rocker [3]. Given the link dimensions and ground link, the type of the four-bar linkage can be determined by Grashof criteria. The number of circuits and number of geometric inversions as well as the input and output ranges for a given four-bar linkage can be determined. All these information can be determined by member functions `grashof()` and `getJointLimits()` in the fourbar class. The function prototype for member function `grashof()` is as follow:

```
int CFourbar::grashof(string_t &name)
```

where `name` of string type indicates the grashof type. The function returns a number that corresponds to the distinct grashof type for the given link dimensions. If the fourbar links cannot form a valid linkage, the return value is -1. Otherwise, it return a macro number which distinct the grashof type. The function prototype for `getJointLimits()` is:

```
int CFourbar::getJointLimits(double inputmin[2], inputmax[2],
                             double outputmin[2], outputmax[2]);
```

where `inputmin` and `inputmax` are the minimum and maximum values for the ranges of motion for input link 2, and
`outputmin` and `outputmax` are the minimum and maximum values for the ranges of motion for output link 4. How these functions in the linkage toolbox is used for mechanism design can be demonstrated by the following mechanism design problem.

> **Problem 3**: The link lengths of a four-bar linkage, as shown in Figure 2.1, are given as follows: $r_1 = 12cm, r_2 = 4cm, r_3 = 12cm, r_4 = 7cm$. The phase angle for link 1 is $\theta_1 = 10°$, the coupler point P is defined by distance $r_p = 5cm$ and constant angle $\beta = 20°$, Determine the type, and input and output ranges of the four-bar linkage. Plot the coupler curve for coupler point $P = (x_p, y_p)$ when input link 2 is rotated from θ_{2min} to θ_{2max}.

You can solve this problem by Program 7. The output of Program 7 is shown in Figure 2.4. Note that member function `grashof()` internally calls `getJointLimits()` to determine the input/output ranges of the fourbar, so that member function `printJointLimits()` can display these values. Coupler curve plots

```
#include <fourbar.h>

int main() {
    CFourbar fourbar;
    double r1 = 0.12, r2 = 0.04, r3 = 0.10,
           r4= 0.07;//crank-rocker
    double theta1 = 10*M_PI/180;
    double rp = 0.05, beta = 20*M_PI/180;
    string_t fourbartype;

    fourbar.uscUnit(false);
    fourbar.setLinks(r1, r2, r3, r4, theta1);
    fourbar.setCouplerPoint(rp, beta);
    fourbar.setNumPoints(50);
    fourbar.grashof(fourbartype);
    printf("Linkage type: %s\n", fourbartype);
    fourbar.printJointLimits();

    int branchnum = 1;
    class CPlot plot1, plot2;
    fourbar.plotCouplerCurve(&plot1, branchnum);
    branchnum++;
    plot2.outputType(PLOT_OUTPUTTYPE_FILE, "postscript eps", "couplerCurve.eps");
    fourbar.plotCouplerCurve(&plot2, branchnum);
}
```

Program 7: Program `couplerCurve()` for generating coupler curves of a four-bar linkage.

may also be saved to a file. For Program 7, the coupler curve plot for the second branch of the fourbar mechanism is saved into an encapsulted postscript file by the following statement.

```
plot2.outputType(PLOT_OUTPUTTYPE_FILE, "postscript eps",
                 "couplerCurve.eps");
```

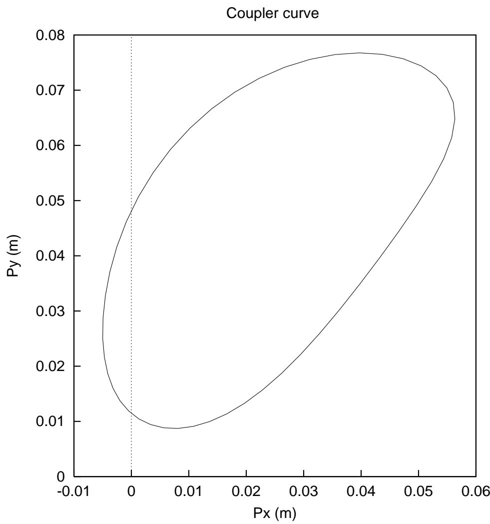
Function `outputType()` is a member of class `CPlot`. The first argument is a macro specifying that the output plot should be saved to a file, with file type specified by the second input argument. The third input argument is the file name. Note that member function `CPlot::outputType()` should be called prior to member function `CFourbar::plotCouplerCurve()` as well as similar plotting functions for class `CFourbar` and other mechanism classes.

2.2 Transmission Angle Analysis

The transmission angle for the fourbar mechanism is shown in Figure 2.1 as the angle γ. It is defined as the acute angle between the velocity difference vector V_{BA} (velocity of point B relative to point A) and the absolute velocity vector V_{out} of the output link (link 4). Since vector V_{BA} will always be perpendicular to link 3 and V_{out} will always be perpendicular to link 4 at the 3-4 connection point (point B), the transmission angle γ can be determined by using the following formula.

$$\gamma = \theta_4 - \theta_3 \tag{2.6}$$

```
Linkage type: Crank-Rocker
Input Characteristics: Input 360 degree rotation
Output Range:
    Circuit:         1        2
                   (deg)    (deg)
    Lower limit:   98.98  -149.15
    Upper limit:  169.15   -78.98
```



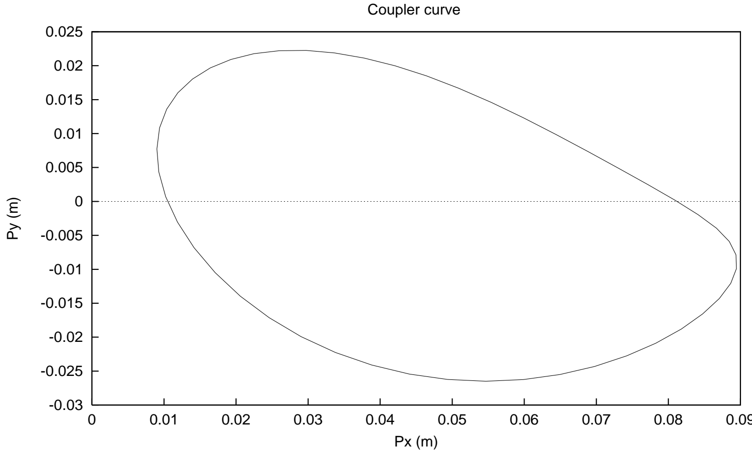


Figure 2.4: The output of Program 7.

The `CFourbar` class contains three member functions for transmission angle analysis: `transAngle()`, `transAngles()` and `plotTransAngles()`. Member function `transAngle()` can be used to calculate the transmission angle of a fourbar linkage given the angular position of either link 2, 3, or 4. Its function prototype is shown below.

```
void CFourbar::transAngle(double &gamma1, &gamma2, double theta,
                          int theta_id);
```

Output arguments `gamma1` and `gamma2` stores the two possible solutions of the transmission angle. Argument `theta` is the angular position value of the link specified by `theta_id`.

The other two member functions, `transAngles()` and `plotTransAngles()`, can be used for analysis of the transmission angle over the entire range of motion of the fourbar mechanism. The function prototypes for `transAngles()` and `plotTransAngles()` are as follows.

```
void CFourbar::transAngles(int branchnum, double theta2[:],
                           double gamma[:]);
void CFourbar::plotTransAngles(class CPlot *pl, int branchnum);
```

For member function `transAngle()`, `branchnum` indicates the branch of the fourbar, and `theta2` and `gamma` are arrays for storing values of θ_2 and γ, respectively. The values of `theta2` contains equally incremented values ranging from $\theta_{2,min}$ to $\theta_{2,max}$. The corresponding transmission angle values are stored in array `gamma`. Note that the array size of `theta2` and `gamma` must be the same.

Since the transmission angle γ is dependent on the input angle, θ_2, it is convenient to be able to generate a plot of the transmission angle for the entire range of motion of the input link. Although the two sets of data values for θ_2 and γ obtained by calling member function `transAngle()` can be used for plotting, member function `plotTransAngles()` can be easily called to accomplish the same goal. Its first argument `pl` is an object of class `CPlot` used for plotting.

> **Problem 4:** Link lengths of a four-bar linkage, as shown in Figure 2.1, are given as follows: $r_1 = 12cm$, $r_2 = 4cm$, $r_3 = 12cm$, $r_4 = 7cm$. The phase angle for the ground link is $\theta_1 = 10^o$. Generate a plot of the transmission angle for the valid range of motion for the fourbar linkage.

The simplest solution to Problem 4 is use class `CFourbar` to first specify the fourbar linkage parameters, and then call member function `plotTransAngles()` to generate the desired plot. Figure 2.5 shows the two possible transmission angle plots for the fourbar linkage described in the above problem statement. The source code for generating these two plots is listed as Program 8. Note that member function `setNumPoints()` is used to specify the number of data points to generate for the plots.

2.3 Velocity Analysis

The velocity analysis for a closed-loop linkage can be carried out from its loop-closure equation. For example, taking the derivative of the loop-closure equation (2.3), we get the following velocity relation

$$\omega_3 r_3 e^{i\theta_3} - \omega_4 r_4 e^{i\theta_4} = -\omega_2 r_2 e^{i\theta_2} \tag{2.7}$$

for the four-bar linkage shown in Figure 2.1. Given values of $r_2, r_3, r_4, \theta_2, \theta_3, \theta_4$ and ω_2, we can readily use the function `complexsolve()` to compute angular velocities ω_3 and ω_4 for coupler and output links,

```
#include <fourbar.h>

int main() {
    CFourbar fourbar;
    double r1 = 0.12, r2 = 0.04, r3 = 0.12, r4 = 0.07, theta1 = 10*M_PI/180;
    int numpoints = 50;
    CPlot plota, plotb;

    fourbar.setLinks(r1, r2, r3, r4, theta1);
    fourbar.setNumPoints(numpoints);
    fourbar.plotTransAngles(&plota, 1);
    fourbar.plotTransAngles(&plotb, 2);
}
```

Program 8: Program for plotting the transmission angle for the valid range of motion of the fourbar mechanism.

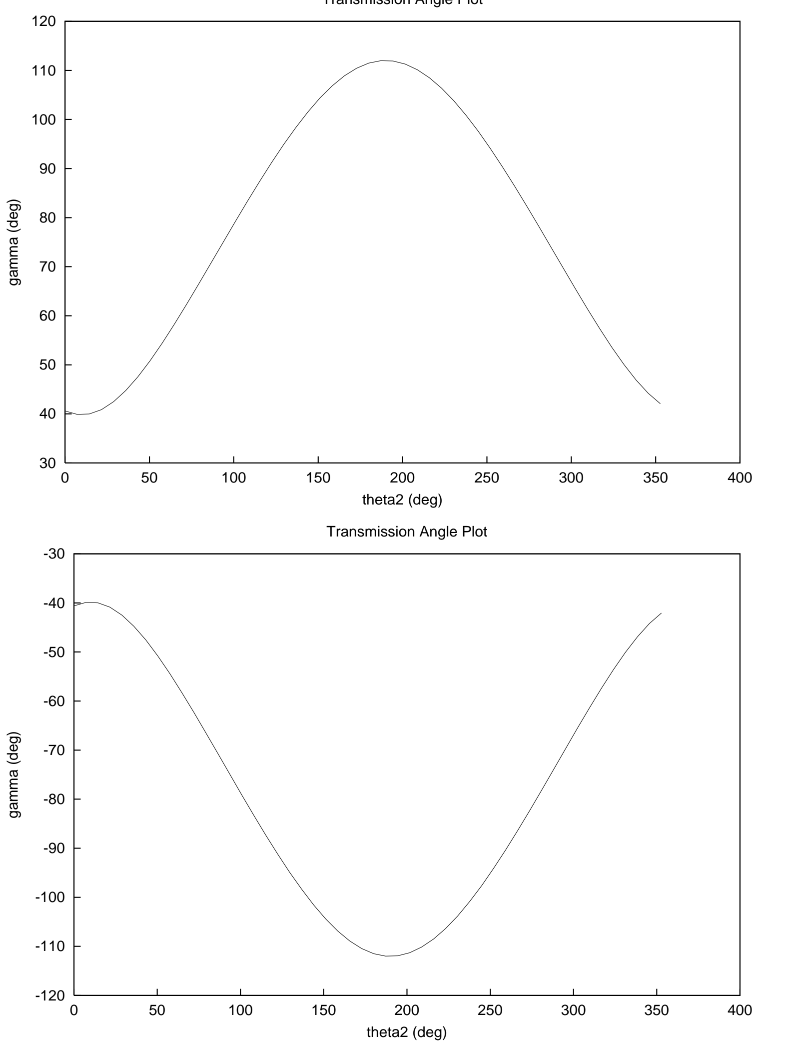


Figure 2.5: Transmission angle plots.

respectively. We can also derive analytical solutions for ω_3 and ω_4. Multiplying equation (2.7) with $e^{-i\theta_4}$, equation (2.7) becomes

$$\omega_3 r_3 e^{i(\theta_3-\theta_4)} - \omega_4 r_4 = -\omega_2 r_2 e^{i(\theta_2-\theta_4)} \tag{2.8}$$

The imaginary part of equation (2.8) gives

$$\omega_3 r_3 \sin(\theta_3-\theta_4) = -\omega_2 r_2 \sin(\theta_2-\theta_4) \tag{2.9}$$

Then

$$\omega_3 = -\frac{\omega_2 r_2 \sin(\theta_4-\theta_2)}{r_3 \sin(\theta_4-\theta_3)} \tag{2.10}$$

Computation of the angular velocity ω_3 can be programmed in Ch as function files or they can use a single line of code. For example, ω_3 can be calculated in foubar class using a member function named `angularVel()`.

Similarly, the following analytical expression for ω_4 can be derived by multiplying equation (2.7) with $e^{-i\theta_3}$,

$$\omega_4 = \frac{\omega_2 r_2 \sin(\theta_3-\theta_2)}{r_4 \sin(\theta_3-\theta_4)} \tag{2.11}$$

ω_4 also can be calculated in Ch using the fourbar class member function `angularVel()`.

The derivative of equation (2.5) gives the following analytical expression for the velocity of coupler point P.

$$V_p = i\omega_2 r_2 e^{i\theta_2} + i\omega_3 r_p e^{i(\theta_3+\beta)} \tag{2.12}$$

which can be translated into a Ch code fragment as

```
double r2, r3, theta2, theta3, rp, beta, omega2, omega3;
double complex I=complex(0,1), Vp;
Vp = I*omega2*polar(r2, theta2) + I*polar(omega3*rp, theta3+beta);
```

2.4 Acceleration Analysis

For a closed-loop planar linkage, the acceleration relation can be obtained by taking the second derivative of the loop-closure equation. For example, by taking the second derivative of the loop-closure equation (2.3), we get the following acceleration relation for the four-bar linkage shown in Figure 2.1.

$$i\alpha_3 r_3 e^{i\theta_3} - \omega_3^2 r_3 e^{i\theta_3} - i\alpha_4 r_4 e^{i\theta_4} + \omega_4^2 r_4 e^{i\theta_4} = i\alpha_2 r_2 e^{i\theta_2} + \omega_2^2 r_2 e^{i\theta_2} \tag{2.13}$$

where α_2, α_3, and α_4 are angular accelerations for input, coupler, and output links, respectively, Similar to the derivation for ω_3, the following analytical formulas for α_3 and α_4, respectively, can be derived:

$$\alpha_3 = \frac{-r_2\alpha_2\sin(\theta_4-\theta_2) + r_2\omega_2^2\cos(\theta_4-\theta_2) + r_3\omega_3^2\cos(\theta_4-\theta_3) - r_4\omega_4^2}{r_3\sin(\theta_4-\theta_3)} \tag{2.14}$$

$$\alpha_4 = \frac{r_2\alpha_2\sin(\theta_3-\theta_2) - r_2\omega_2^2\cos(\theta_3-\theta_2) + r_4\omega_4^2\cos(\theta_3-\theta_4) - r_3\omega_3^2}{r_4\sin(\theta_3-\theta_4)} \tag{2.15}$$

A fourbar class member function `angularAccel()` has been written for calculating α_3 and α_4. It is included in the Ch Mechanism Toolkit.

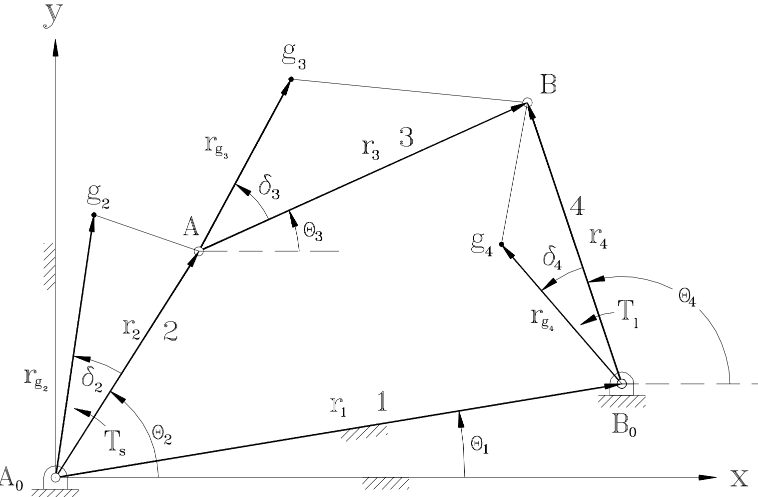


Figure 2.6: The four-bar linkage with offset gravity centers for moving links.

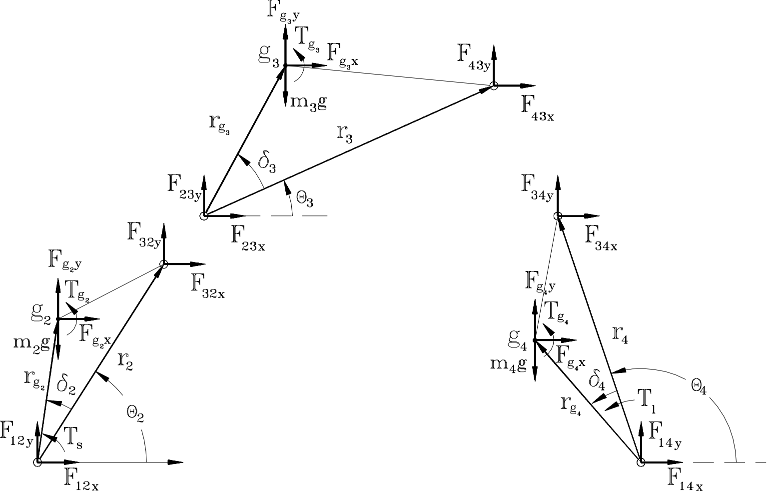


Figure 2.7: Free body diagrams for the moving links of the four-bar linkage.

2.5 Dynamics

The purpose of acceleration analysis is for inertia-force analysis. Given position, velocity, acceleration, and inertia properties such as mass and mass moment of inertia for each moving link of a four-bar linkage, we are able to perform force analysis for the linkage. Various formulations are available for dynamics. The matrix method has been used in the formulation of the Ch Mechanism Toolkit [3]. To simplify the programming burden, we have implemented computational arrays in the Ch programming language. Computational arrays can be treated as single objects.

For the four-bar linkage shown in Figure 2.6, dynamic formulations can be derived to calculate the required input torque T_s and joint reaction forces. Three free-body diagrams for links 2, 3, and 4 are given in Figure 2.7. Three static equilibrium equations, in terms of forces in X and Y directions and moment about the center of gravity of the link, can be written for each link.

For link 2, we get

$$F_{12x} + F_{32x} + F_{g2x} = 0 \tag{2.16}$$

$$-m_2 g + F_{12y} + F_{32y} + F_{g2y} = 0 \quad (2.17)$$
$$T_s + (-\mathbf{r}_{g_2}) \times \mathbf{F}_{12} + (\mathbf{r}_2 - \mathbf{r}_{g_2}) \times \mathbf{F}_{32} + T_{g_2} = 0 \quad (2.18)$$

where $\mathbf{r}_{g_2} = r_{g_2} e^{i(\theta_2+\delta_2)}$ is the position vector from joint A_0 to the center of gravity of link 2. $\mathbf{F}_{12}$ and $\mathbf{F}_{32}$ are the joint forces acting on link 2 from the ground and link 3, respectively. F_{g_2} and T_{g_2} are the inertia force and inertia moment, respectively, of link 2. m_2 is the mass of link 2. T_s is the driving torque.

For link 3, we get

$$F_{23x} + F_{43x} + F_{g3x} = 0 \quad (2.19)$$
$$-m_3 g + F_{23y} + F_{43y} + F_{g3y} = 0 \quad (2.20)$$
$$(-\mathbf{r}_{g_3}) \times \mathbf{F}_{23} + (\mathbf{r}_3 - \mathbf{r}_{g_3}) \times \mathbf{F}_{43} + T_{g_3} = 0 \quad (2.21)$$

where $\mathbf{r}_{g_3} = r_{g_3} e^{i(\theta_3+\delta_3)}$ is the position vector from joint A to the center of gravity of link 3. $\mathbf{F}_{23}$ and $\mathbf{F}_{43}$ are the joint forces acting on link 3 from links 2 and 4, respectively. F_{g_3} and T_{g_3} are the inertia force and inertia moment, respectively, of link 3. m_3 is the mass of link 3.

For link 4, we get

$$F_{34x} + F_{14x} + F_{g4x} = 0 \quad (2.22)$$
$$-m_4 g + F_{34y} + F_{14y} + F_{g4y} = 0 \quad (2.23)$$
$$(-\mathbf{r}_{g_4}) \times \mathbf{F}_{14} + (\mathbf{r}_4 - \mathbf{r}_{g_4}) \times \mathbf{F}_{34} + T_{g_4} + T_l = 0 \quad (2.24)$$

where $\mathbf{r}_{g_4} = r_{g_4} e^{i(\theta_4+\delta_4)}$ is the position vector from joint B_0 to the center of gravity of link 4. $\mathbf{F}_{14}$ and $\mathbf{F}_{34}$ are the joint forces acting on link 4 from the ground and link 3, respectively. F_{g_4} and T_{g_4} are the inertia force and inertia moment, respectively, of link 4. m_4 is the mass of link 4. T_l is the torque of external load.

Equations (2.18), (2.21), and (2.24) can be expressed explicitly as

$$T_s - r_{g_2}\cos(\theta_2+\delta_2)F_{12y} + r_{g_2}\sin(\theta_2+\delta_2)F_{12x}$$
$$+[r_2\cos\theta_2 - r_{g_2}\cos(\theta_2+\delta_2)]F_{32y} - [r_2\sin\theta_2 - r_{g_2}\sin(\theta_2+\delta_2)]F_{32x} + T_{g_2} = 0 \quad (2.25)$$
$$-r_{g_3}\cos(\theta_3+\delta_3)F_{23y} + r_{g_3}\sin(\theta_3+\delta_3)F_{23x}$$
$$+[r_3\cos\theta_3 - r_{g_3}\cos(\theta_3+\delta_3)]F_{43y} - [r_3\sin\theta_3 - r_{g_3}\sin(\theta_3+\delta_3)]F_{43x} + T_{g_3} = 0 \quad (2.26)$$
$$-r_{g_4}\cos(\theta_4+\delta_4)F_{14y} + r_{g_4}\sin(\theta_4+\delta_4)F_{14x}$$
$$+[r_4\cos\theta_4 - r_{g_4}\cos(\theta_4+\delta_4)]F_{34y} - [r_4\sin\theta_4 - r_{g_4}\sin(\theta_4+\delta_4)]F_{34x} + T_{g_4} + T_l = 0 \quad (2.27)$$

Note that $F_{ijx} = -F_{jix}$ and $F_{ijy} = -F_{jiy}$, equations (2.16-2.24) can be rewritten as nine linear equations in terms of nine unknowns $F_{12x}, F_{12y}, F_{23x}, F_{23y}, F_{34x}, F_{34y}, F_{14x}, F_{14y}$, and T_s (8 joint reaction forces plus one input torque). They can be expressed in a symbolic form

$$\mathbf{A}\mathbf{x} = \mathbf{b} \quad (2.28)$$

where $\mathbf{x} = (F_{12x}, F_{12y}, F_{23x}, F_{23y}, F_{34x}, F_{34y}, F_{14x}, F_{14y}, T_s)^T$ is a vector consisting of the unknown forces and input torque, $\mathbf{b} = (F_{g2x}, F_{g2y} - m_2 g, T_{g_2}, F_{g3x}, F_{g3y} - m_3 g, T_{g_3}, F_{g4x}, F_{g4y} - m_4 g, T_{g_4} + T_l)^T$ is a vector that contains external load plus inertia forces and inertia torques, and **A** is a 9x9 square matrix

$$A = \begin{bmatrix} -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ -r_{g2}\sin(\theta_2+\delta_2) & r_{g2}\cos(\theta_2+\delta_2) & r_{g2}\sin(\theta_2+\delta_2) - r_2\sin\theta_2 & r_2\cos\theta_2 - r_{g2}\cos(\theta_2+\delta_2) & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & -r_{g3}\sin(\theta_3+\delta_3) & r_{g3}\cos(\theta_3+\delta_3) & r_{g3}\sin(\theta_3+\delta_3) - r_3\sin\theta_3 & r_3\cos\theta_3 - r_{g3}\cos(\theta_3+\delta_3) & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & r_4\sin\theta_4 - r_{g4}\sin(\theta_4+\delta_4) & r_{g4}\cos(\theta_4+\delta_4) - r_4\cos\theta_4 & -r_{g4}\sin(\theta_4+\delta_4) & r_{g4}\cos(\theta_4+\delta_4) & 0 \end{bmatrix} \tag{2.29}$$

formed using the angular position of each link and link parameters. What distinguish the above-derived equations (2.16)-(2.24) from those in Erdman and Sandor [3] are that the center of gravity of each link is not at the center line between two joints and the gravitation force for each link is included in formulations explicitly. Because equation (2.28) has 9 unknowns, it should be solved numerically. This can be easily implemented in Ch by only a single line of code shown below,

```
X = inverse(A)*B
```

A fourbar class member function `force()` has been written for the Ch Mechanism Toolkit. Function `force()` can calculate the joint forces and required input torque to achieve the desired motion of the four-bar linkage. The first three input arguments of the function `force()` are arrays, `theta[i]` for joint angle θ_i, `omega[i]` for angular velocity ω_i, `alpha[i]` for angular acceleration α_i, `tl` is the external load T_l. The output `X` from the function `force()` contains the joint forces and required input torque, which is passed to the calling function as an argument of assumed-shape computational array. How to use this function in the toolkit can be demonstrated by the following mechanism design problem given in [3].

Problem 5: Link parameters and inertia properties of a four-bar linkage, as shown in Figure 2.6, are given in the chart below.

Link	Length r (in)	Weight (lb)	I_g (lbf ft sec^2)	C. G. r_g (in)	δ
1	12	—	—	—	—
2	4	0.8	0.0010	2	0
3	12	2.4	0.0099	6	0
4	7	1.4	0.0032	3.5	0

The phase angle for link 1 is $\theta_1 = 0$. There is no external load. At one point the input angular position $\theta_2 = 150°$, angular velocity $\omega_2 = 5$ rad/sec ccw and angular acceleration $\alpha_2 = 5$ rad/sec^2 cw, find the joint reaction forces and required input torque at this moment.

```
#include <math.h>
#include <fourbar.h>

int main() {
    CFourbar fourbar;
    double r1 = 12/12.0, r2 = 4/12.0, r3 =12/12.0, r4 = 7/12.0, theta1 = 0;
    double rp = 5/12.0, beta = 20*M_PI/180;
    double theta_1[1:4], theta_2[1:4], omega[1:4], alpha[1:4];
    array double X[9];
    double g=32.2;
    double rg2 = 2/12.0, rg3 = 6/12.0, rg4 = 3.5/12.0;
    double delta2 = 0.0, delta3 = 0, delta4 = 0.0;
    double m2 = 0.8/g, m3 = 2.4/g, m4 = 1.4/g;
    double ig2 = 0.0010, ig3 = 0.0099, ig4 = 0.0032, tl=0;

    /* initialization of link parameters and
    inertia properties */

    theta_1[1] = 0; theta_1[2]=150*M_PI/180;
    theta_2[1] = 0; theta_2[2]=150*M_PI/180;
    omega[2] = 5;  alpha[2] = -5;
    fourbar.uscUnit(true);
    fourbar.setLinks(r1, r2, r3, r4, theta1);
    fourbar.setCouplerPoint(rp, beta);
    fourbar.setGravityCenter(rg2, rg3, rg4, delta2, delta3, delta4);
    fourbar.setInertia(ig2, ig3, ig4);
    fourbar.setMass(m2, m3, m4);

    // find theta3, theta4
    fourbar.angularPos(theta_1, theta_2, FOURBAR_LINK2);
    // find omega3, omega4, first set
    fourbar.angularVel(theta_1, omega, FOURBAR_LINK2);
    // find alpha3, alpha4
    fourbar.angularAccel(theta_1, omega, alpha, FOURBAR_LINK2);
    // find forces, torque
    fourbar.forceTorque(theta_1, omega, alpha, tl, X);
    printf("first solution X = %.4f \n", X);

    // find omega3, omega4, second set
    fourbar.angularVel(theta_2, omega, FOURBAR_LINK2);
    // find alpha3, alpha4
    fourbar.angularAccel(theta_2, omega, alpha, FOURBAR_LINK2);
    // find forces, torque
    fourbar.forceTorque(theta_2, omega, alpha, tl, X);
    printf("second solution X = %.4f \n", X);
}
```

Program 9: Program `force()` for computing joint reaction forces and required input torque.

You can solve this problem by Program 9. Note that the various lengths, masses, and moments of inertia are represented in $feet$, $slugs$, and $lb-ft-sec^2$, respectively. The output of Program 9 is given as follows:

```
first solution X = 1.7993 2.3553 1.6993 1.5895 1.1643 -0.7428 -1.0273 2.1624 -0.8659

second solution X = -0.6161 2.4778 -0.7161 1.7120 -1.1555 -0.4193 1.2368 1.7218 -0.4987
```

Problem 6: Link parameters and inertia properties of a four-bar linkage are given in the chart below.

Link	Length r (cm)	Mass (kg)	I_g (kg m^2)	C. G. r_g (cm)	δ
1	30.48	—	—	—	—
2	10.16	0.3628	0.001356	5.08	0
3	30.48	1.0883	0.013445	15.24	0
4	17.78	0.6348	0.004293	8.89	0

The phase angle for link 1 is $\theta_1 = 0$. There is no external load. At one point the input angular position $\theta_2 = 150°$, angular velocity $\omega_2 = 5 \text{ rad/sec}$ ccw and angular acceleration $\alpha_2 = 5 \text{ rad/sec}^2$ cw, find the joint reaction forces and required input torque at this moment.

Program 10 is the equivalent of Program 9, except that SI units are used rather than US Customary units. The output of Program 10 is shown below. Note that this output is the SI equivalent to the output for Program 9.

```
first solution X = 8.0075 10.4824 7.5624 7.0739 5.1815 -3.3056 -4.5716 9.6234 -1.1746

second solution X = -2.7421 11.0279 -3.1873 7.6194 -5.1427 -1.8663 5.5045 7.6627 -0.6765
```

2.6 Kinematics and Dynamics with Constant Angular Velocity for Link 2

Analysis of the fourbar mechanism described in the previous sections were performed for only one specific position. Assuming constant angular velocity for the input link, link 2, this section will discuss analysis of the fourbar linkage over the entire range of motion of this link. With constant angular velocity, the relationships between the angular position, velocity, and acceleration for link 2 is as follows,

$$
\begin{aligned}
\theta_2 &= \omega_0 t + \theta_{2,min} \\
\omega_2 &= \omega_0 \\
\alpha_2 &= 0
\end{aligned}
$$

where $\theta_{2,min}$ is the minimum angular position value of the link, and ω_0 is a constant value. Note that the total time for one rotation of the input link, $\theta_2 = \theta_{2,min}$ to $\theta_2 = \theta_{2,max}$, can be determined by the following equation,

$$
t_{max} = \frac{\theta_{2,max} - \theta_{2,min}}{\omega_0}
$$

```
#include <math.h>
#include <fourbar.h>

int main() {
    CFourbar fourbar;
    double r1 = 0.3048, r2 = 0.1016, r3 = 0.3048, r4 = 0.1778,
           theta1 = 0;
    double rp = 0.1270, beta = 20*M_PI/180;
    double theta_1[1:4], theta_2[1:4], omega[1:4], alpha[1:4];
    array double X[9];
    double g = 9.81;
    double rg2 = 0.0508, rg3 = 0.1524, rg4 = 0.0889;
    double delta2 = 0.0, delta3 = 0, delta4 = 0.0;
    double m2 = 0.3628, m3 = 1.0883, m4 = 0.6348;
    double ig2 = 0.001356, ig3 = 0.013445, ig4 = 0.004293, tl=0;

    /* initialization of link parameters and
    inertia properties */

    theta_1[1] = 0; theta_1[2]=150*M_PI/180;
    theta_2[1] = 0; theta_2[2]=150*M_PI/180;
    omega[2] = 5;  alpha[2] = -5;
    fourbar.uscUnit(false);
    fourbar.setLinks(r1, r2, r3, r4, theta1);
    fourbar.setCouplerPoint(rp, beta);
    fourbar.setGravityCenter(rg2, rg3, rg4, delta2, delta3, delta4);
    fourbar.setInertia(ig2, ig3, ig4);
    fourbar.setMass(m2, m3, m4);

    // find theta3, theta4
    fourbar.angularPos(theta_1, theta_2, FOURBAR_LINK2);
    // find omega3, omega4, first set
    fourbar.angularVel(theta_1, omega, FOURBAR_LINK2);
    // find alpha3, alpha4
    fourbar.angularAccel(theta_1, omega, alpha, FOURBAR_LINK2);
    // find forces, torque
    fourbar.forceTorque(theta_1, omega, alpha, tl, X);
    printf("first solution X = %.4f \n", X);

    // find omega3, omega4, second set
    fourbar.angularVel(theta_2, omega, FOURBAR_LINK2);
    // find alpha3, alpha4
    fourbar.angularAccel(theta_2, omega, alpha, FOURBAR_LINK2);
    // find forces, torque
    fourbar.forceTorque(theta_2, omega, alpha, tl, X);
    printf("second solution X = %.4f \n", X);
}
```

Program 10: Program `force()` for computing joint reaction forces and required input torque in SI units.

```
#include <math.h>
#include <stdio.h>
#include <fourbar.h>

int main() {
    double r[1:4], theta1;
    double omega2;
    int numpoints = 50;
    CFourbar fourbar;
    CPlot plota, plotb, plotc;

    /* default specification of the four-bar linkage */
    r[1] = 12; r[2] = 4; r[3] = 12; r[4] = 7;
    theta1 = 10*M_PI/180;
    omega2 = 5;  /*  rad/sec  */

    fourbar.setLinks(r[1], r[2], r[3], r[4], theta1);
    fourbar.setAngularVel(omega2);
    fourbar.setNumPoints(numpoints);
    fourbar.plotAngularPoss(&plota, 1);
    fourbar.plotAngularVels(&plotb, 1);
    fourbar.plotAngularAccels(&plotc, 1);

    return 0;
}
```

Program 11: Program for plotting θ_3, θ_4, ω_3, ω_4, α_3, and α_4 with respect to time.

where $\theta_{2,max}$ is the maximum angular position value for link 2.

Using the above relationships, member functions `plotAngularPos()`, `plotAngularVel()`, and `plotAngularAccel()` of class `CFourbar` were developed for time-based kinematic analysis of links 3 and 4 of a fourbar mechanism . Given a constant value for ω_2, these member functions can be used to plot the angular positions, velocities, and accelerations of links 3 and 4 with respect to time. The function prototypes for these member functions are as follows.

```
void CFourbar::plotAngularPoss(class CPlot *pl, int branchnum);
void CFourbar::plotAngularVels(class CPlot *pl, int branchnum);
void CFourbar::plotAngularAccels(class CPlot *pl, int branchnum);
```

Argument `pl` is an object of class **CPlot**used for plotting, and `branchnum` is the branch number of the fourbar linkage to analyze.

> **Problem 7:** Link lengths of a four-bar linkage, as shown in Figure 2.1, are given as follows: $r_1 = 12cm$, $r_2 = 4cm$, $r_3 = 12cm$, $r_4 = 7cm$. The phase angle for the ground link is $\theta_1 = 10^o$, and the constant angular velocity of the input link is $\omega_2 = 5rad/sec$. Plot the angular positions, velocities, and accelerations of links 3 and 4 with respect to time for the 1st branch.

The solution to Problem 7 is Program 11. After specifying the required parameters for the fourbar class, member functions `plotAngularPoss()`, `plotAngularVels()`, and `plotAngularAccels()` are called to generate the desired plots. The outputs of Program 11 are Figures 2.8 - 2.10.

In addition to member functions `plotAngularPoss()`, `plotAngularVels()`, and `plotAngularAccels()`, memeber functions `angularPoss()`, `angularVels()`, and

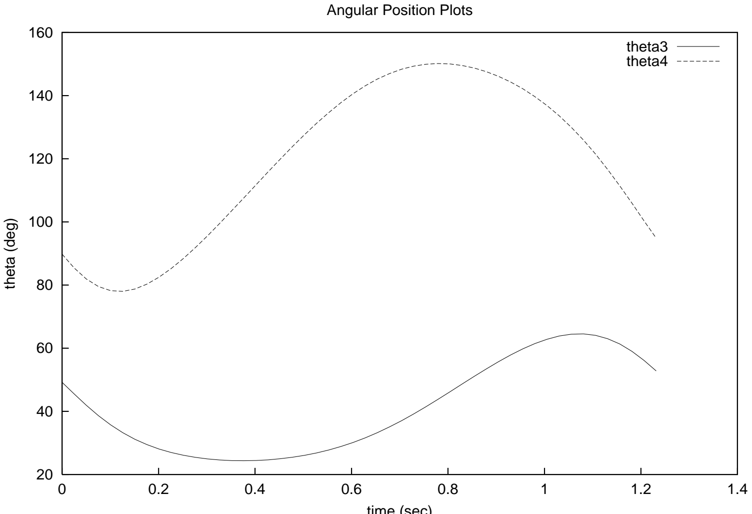


Figure 2.8: Angular position plot.

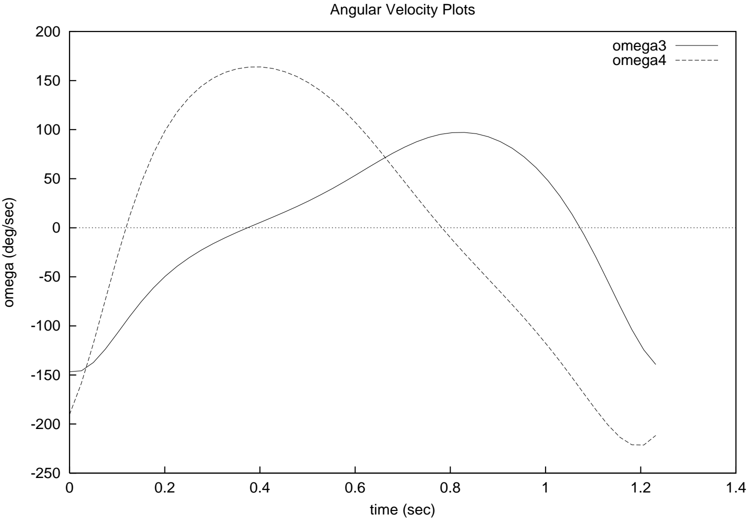


Figure 2.9: Angular velocity plot.

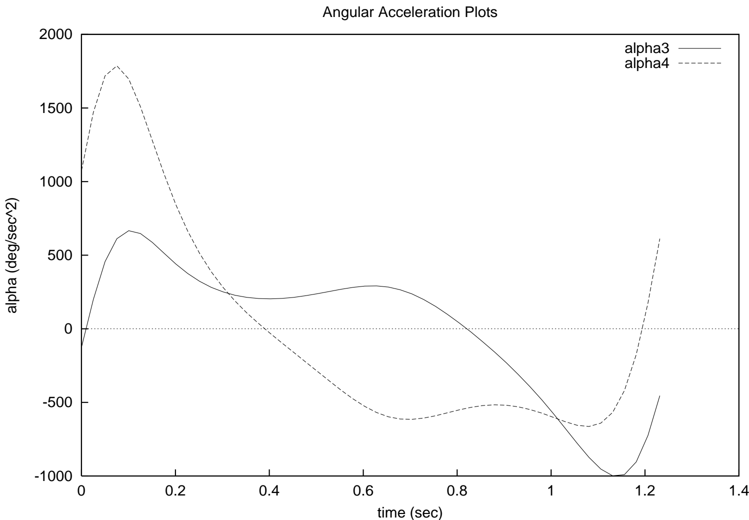


Figure 2.10: Angular acceleration plot.

`angularAccels()` are also available for kinematic analysis of the fourbar linkage with constant angular velocity for link 2. Instead of generating plots of the angular position, velocity, and acceleration of links 3 and 4 as with the plotting functions described above, these other functions generates the data used for plotting and store them into arrays for later use. For, example the generated data can be saved to a data file or used to generate plots similar to `plotAngularPoss()`, `plotAngularVels()`, and `plotAngularAccels()`. These functions have the following function prototypes.

```
int CFourbar::angularPoss(int branchnum, double time[:],
                          double theta3[:], double theta4[:]);
int CFourbar::angularVels(int branchnum, double time[:],
                          double omega3[:], double omega4[:]);
int CFourbar::angularAccels(int branchnum, double time[:],
                            double alpha3[:], double alpha4[:]);
```

Argument `branchnum` is an integer value specifying the branch of the fourbar. Array `time` is a set of time values equally incremented from time $t = 0$ to $t = t_{max}$, where t_{max} is the total time required for one complete motion of the fourbar. Arrays `theta3`, `theta4`, `omega3`, `omega4`, `alpha3`, and `alpha4` contains corresponding values for the angular positions, velocities, and accelerations of links 3 and 4, respectively. Note that the array arguments for the above member functions should all have the same size.

For dynamic analysis of the fourbar linkage with a constant ω_2 value, member functions `forceTorques()` and `plotForceTorques()` are available. Member function `forceTorques()` determines the joint forces and input torque for the entire valid range of motion of the fourbar mechanism, whereas `plotForceTorques()` generates a graphical representation of these values. Their function prototypes are as follows.

```
int CFourbar::forceTorques(int branchnum, double tl, array double time[:],
                           f12x[:], f12y[:], f23x[:], f23y[:],
                           f34x[:], f34y[:], f14x[:], f14y[:], ts[:]);
int CFourbar::plotForceTorques(class CPlot *pl, int branchnum,
                               double tl);
```

```
/************************************************************
*  This example plots the joint forces and output torque curves.
************************************************************/
#include <math.h>
#include <fourbar.h>

int main()
{
    CFourbar fourbar;
    double r1 = 0.12, r2 = 0.04, r3 = 0.10, r4= 0.07;
    double theta1 = 10;
    double rg2 = 0.0508, rg3 = 0.1524, rg4 = 0.0889;
    double delta2 = 0.0, delta3 = 0, delta4 = 0.0;
    double m2 = 0.3628, m3 = 1.0833, m4 = 0.6348;
    double ig2 = 0.001356, ig3 = 0.013445, ig4 = 0.004293, tl=0;
    int numpoint = 50;
    double  omega2 = 5;  /* constant omega2 */
    class CPlot plot;

    /* initialization of link parameters and
    inertia properties */

    fourbar.uscUnit(true);
    fourbar.setLinks(r1, r2, r3, r4, theta1);
    fourbar.setGravityCenter(rg2, rg3, rg4, delta2, delta3, delta4);
    fourbar.setInertia(ig2, ig3, ig4);
    fourbar.setMass(m2, m3, m4);
    fourbar.setNumPoints(numpoint);
    fourbar.setAngularVel(omega2);
    fourbar.plotForceTorques(&plot,1,tl); //first branch
}
```

Program 12: Program for plotting joint forces and output torque with respect to time.

For these functions, `tl` is the load torque, `time` is an array to record time, and `f12x`, `f12y`, `f23x`, `f23y`, `f34x`, `f34y`, `f14x`, `f14y`, and `ts` are arrays for the joint forces and input torque.

Problem 8: Link lengths of a four-bar linkage, as shown in Figure 2.1, are given as follows: $r_1 = 12cm$, $r_2 = 4cm$, $r_3 = 10cm$, $r_4 = 7cm$. The phase angle for the ground link is $\theta_1 = 0$, the constant angular velocity of the input link is $\omega_2 = 5rad/sec$, and load torque $t_l = 0$. Using the inertia properties provided in the prevous problems for dynamic analysis, plot the joint forces and input torque for the valid range of motion of the fourbar mechanism for the 1st branch.

Program 12 is the solution to the above problem statement. After indicating the link parameters and inertia properties, member function `plotForceTorques()` is called to generate the desired plot. The plot of the joint forces and input torque is listed as Figure 2.11.

2.7 Three-Position Synthesis

Recall the loop-closure equation for the fourbar linkage rewritten below.

$$\mathbf{r}_1 + \mathbf{r}_4 = \mathbf{r}_2 + \mathbf{r}_3 \tag{2.30}$$

$$r_1 e^{i\theta_1} + r_4 e^{i\theta_4} = r_2 e^{i\theta_2} + r_3 e^{i\theta_3} \tag{2.31}$$

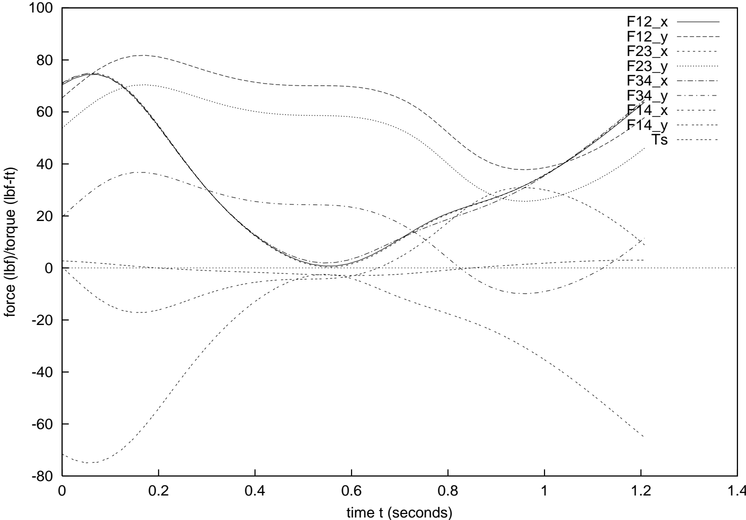


Figure 2.11: Joint forces and input torque plot.

Previously, equation (2.31) was used to derive equations for calculating the angular positions, velocities, and accelerations of the remaining links given the dynamic properties of one link. Similarly, the loop-closure equation provides a basis for the synthesis of fourbar mechanisms given three sets of input/output angles. Assuming $\theta_1 = 0$, equation (2.31) is rewritten as equations (2.32) and (2.33). The next step is to eliminate θ_3 in the equations, which can be done by first isolating the terms containing θ_3 on the right-hand side.

$$r_1 + r_4 \cos\theta_4 = r_2 \cos\theta_2 + r_3 \cos\theta_3 \qquad (2.32)$$

$$r_4 \sin\theta_4 = r_2 \cos\theta_2 + r_3 \cos\theta_3 \qquad (2.33)$$

$$r_1 + r_4 \cos\theta_4 - r_2 \cos\theta_2 = r_3 \cos\theta_3 \qquad (2.34)$$

$$r_4 \sin\theta_4 - r_2 \sin\theta_2 = r_3 \sin\theta_3 \qquad (2.35)$$

Squaring and adding equations (2.34) and (2.35) together will then cancel out the θ_3 term, which results in equation (2.37),

$$\frac{r_3^2 - r_1^2 - r_2^2 - r_4^2}{2r_2r_4} + \frac{r_1}{r_4}\cos\theta_2 - \frac{r_1}{r_2}\cos\theta_4 = -\cos\theta_2 - \theta_4 \qquad (2.36)$$

$$k_1 \cos\theta_2 + k_2 \cos\theta_4 + k_3 = -\cos\theta_2 - \theta_4 \qquad (2.37)$$

where

$$k_1 = \frac{r_1}{r_4}$$

$$k_2 = \frac{-r_1}{r_2}$$

$$k_3 = \frac{r_3^2 - r_1^2 - r_2^2 - r_4^2}{2r_2r_4}.$$

Equation 2.37 is called Freudenstein's equation.

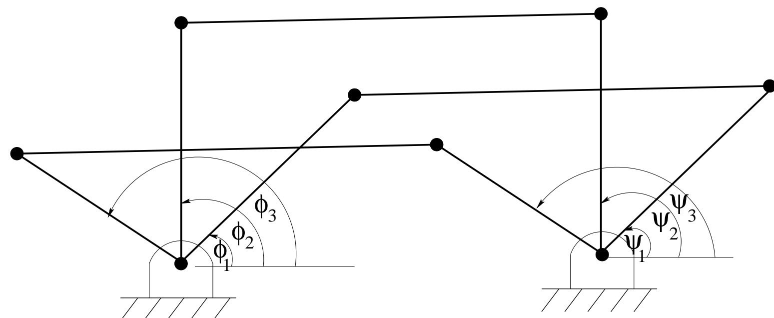


Figure 2.12: Freudenstein analysis.

The three unknowns in equation (2.37), k_1, k_2, and k_3, can be solved for three-point function generation. Thus, for three prescribed positions as shown in Figure 2.12, Freudenstein's equation becomes,

$$k_1 \cos\phi_1 + k_2 \cos\phi_1 + k_3 = -\cos\phi_1 - \psi_1 \tag{2.38}$$
$$k_1 \cos\phi_2 + k_2 \cos\phi_2 + k_3 = -\cos\phi_2 - \psi_2 \tag{2.39}$$
$$k_1 \cos\phi_3 + k_2 \cos\phi_3 + k_3 = -\cos\phi_3 - \psi_3 \tag{2.40}$$

which can be simplified into $\mathbf{Ax} = \mathbf{b}$ or equation (2.41).

$$\begin{bmatrix} \cos(\phi_1) & \cos(\psi_1) & 1 \\ \cos(\phi_2) & \cos(\psi_2) & 1 \\ \cos(\phi_3) & \cos(\psi_3) & 1 \end{bmatrix} \begin{bmatrix} k_1 \\ k_2 \\ k_3 \end{bmatrix} = \begin{bmatrix} -\cos(\phi_1 - \psi_1) \\ -\cos(\phi_2 - \psi_2) \\ -\cos(\phi_3 - \psi_3) \end{bmatrix} \tag{2.41}$$

Once the values for k_1, k_2, and k_3 are obtained from equation (2.41, link dimensions r_2, r_3, and r_4 can be solved in terms of input link length r_1 with equations (2.42) - (2.44). Typically, the length r_1 is chosen to be 1.

$$r_4 = \frac{r_1}{k_1} \tag{2.42}$$
$$r_2 = \frac{-r_1}{k_2} \tag{2.43}$$
$$r_3 = \sqrt{2k_3 r_2 r_4 + r_1^2 + r_2^2 + r_4^2} \tag{2.44}$$

Class `CFourbar` also contains member function `synthesis()` to calculate the various link lenths using Freudenstein's equation. The function prototype for `synthesis()` is shown below.

```
int CFourbar::synthesis(double r[1:4], double phi[:], double psi[:]);
```

Argument `r` is a 4-element array to store the values of the link lengths. Note that element `r[1]` shall be a given value, and elements `r[2]`, `r[3]`, and `r[4]` shall contain the outputs of the function. Arguments `phi` and `psi` are used to specify the sets of input and output angles, respectively.

Problem 9: Given $r_1 = 1m$, $\phi = [105^o, 157^o, 209^o]$, and $\psi = [66.27^o, 102.42^o, 119.67^o]$, calculate the link lengths r_2, r_3, and r_4. Also display the fourbar linkage at the three specified positions.

Problem 9 can be solved by using member function `synthesis()` described above and another member function `displayPositions()`, which is similar to member function `displayPosition()` described in Section 2.1. Its function prototype is as follows,

```
int CFourbar::displayPositions(double theta2[:], double theta3[:],
                               double theta4[:], ...
                               /*[int outputtype [, char *filename]]*/);
```

It is similar to member function `displayPosition()`, except that multiple positions may be displayed in one figure. Arguments `theta2`, `theta3`, and `theta4` shall have the same number of elements, and the arrays' argument size shall specify the number of positions to display.

The solution to Problem 9 is Program 13. After specifying the length of the ground link r_1 and the three sets of input/output angles, member function `synthesis()` is called to determine link dimensions r_2, r_3, and r_4. Next, the three θ_3 values are calculated by using member function `getAngle()`. It has the following function prototype,

```
int CFourbar::getAngle(double theta[1:], int theta_id);
```

where `theta` are the angular position values and `theta_id` indicates the unknown link angle. Member function `setLinks()` is then called to define the fourbar linkage in order to use member function `displayPositions()` to display the fourbar mechanism in the three specified positions. The output of Program 13 is shown in Figure 2.13.

2.8 Animation

The concepts discussed in Section 2.1 concerning position analysis of a fourbar linkage can be applied to simulate the motion of a fourbar mechanism. By applying the Grashof criteria or calling member function `grashof()`, the type of fourbar as well as the number of geometric inversions can be determined. Furthermore, the range of the input link, link 2, can also be calculated by hand or by using member function `getJointLimits()`, which is invoked internally when calling member function `grashof()`. With the range of the input link known, the angular positions of the other links can be determined by applying the equations developed in Section 2.1 for any instance of link 2. Thus, a **for**-loop can be utilized to obtain the positions of links 3 and 4 for the entire range of the input link.

Utilizing the above observations, member function `animation()` was designed to simulate the motion of a fourbar linkage. Its function prototype is as follows,

```
int CFourbar::animation(int branchnum, ...);
```

where `branchnum` indicates the branch number of the fourbar to simulate. For typical fourbar linkages, `branchnum` can be either 1 or 2 for the first and second geometric inversions, respectively. However, for a rocker-rocker mechanism, `branchnum` can have a maximum value of 4 for the four possible branches of the rocker-rocker mechanism.

Additionally, member function `animation()` supports the option to store the data used for the generating the fourbar animation into a file. This file can be used by the Quick Animation program to simulate the movement of the fourbar linkage at another time. The format for saving data to a file is shown below.

```
fourbar.animation(branchnum, QANIMATE_OUTPUTTYPE_FILE, "data.qnm");
```

```
#include <stdio.h>
#include <fourbar.h>

int main()
{
    double r[1:4];
    double psi[1:3], phi[1:3];
    double theta[1:4], theta3[1:3];
    CFourbar fourbar;

    /* specify input/output relation for a four-bar linkage */
    r[1] = 1;
    psi[1]=66.27*M_PI/180; psi[2]=102.42*M_PI/180; psi[3]=119.67*M_PI/180;
    phi[1]=105*M_PI/180; phi[2]=157*M_PI/180; phi[3]=209*M_PI/180;
    fourbar.synthesis(r,phi,psi);

    /* display link lengths */
    printf("r2 = %.3f, r3 = %.3f, r4 = %.3f\n", r[2], r[3], r[4]);

    /* obtain theta3 in three positions and display these positions */
    theta[1] = 0;
    fourbar.setLinks(r[1], r[2], r[3], r[4], theta[1]);
    theta[2]=phi[1];  theta[4] = psi[1];
    fourbar.getAngle(theta, FOURBAR_LINK3);
    theta3[1] = theta[3];
    theta[2] = phi[2]; theta[4] = psi[2];
    fourbar.getAngle(theta, FOURBAR_LINK3); theta3[2] = theta[3];
    theta[2] = phi[3]; theta[4] = psi[3];
    fourbar.getAngle(theta, FOURBAR_LINK3); theta3[3] = theta[3];
    fourbar.displayPositions(phi, theta3, psi);

    return 0;
}
```

Program 13: Program for fourbar synthesis and displaying positions.

```
r2 = 0.555, r3 = 1.441, r4 = 0.725
```

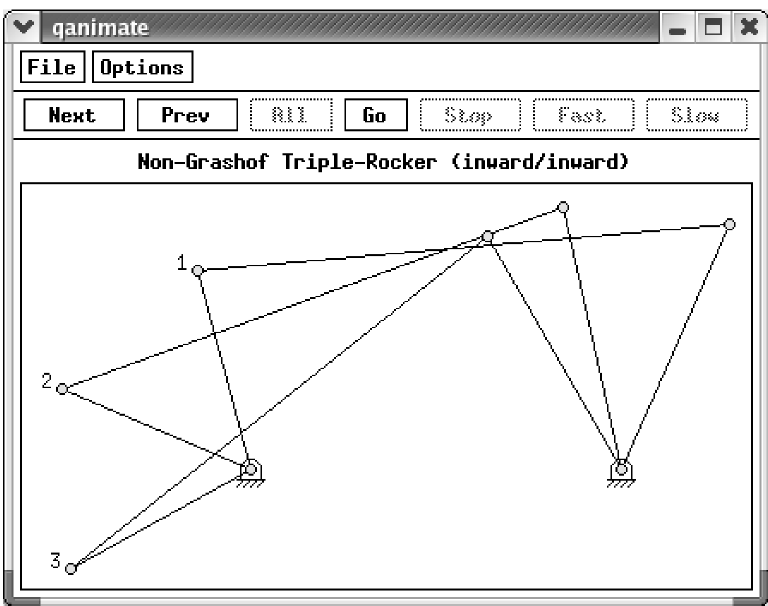


Figure 2.13: The output of Program 13.

Macro QANIMATE_OUTPUTTYPE_FILE specifies that the data to be generated by the animation() function is to be stored into file "data.qnm", where .qnm is the file extension for the Quick Animation program. A more in depth discussion of Quick Animation will be given in Chapter 7.

In order to simulate movement of the fourbar linkage, the number of frames to generate needs to be specified. This can be done by member function setNumPoints(), which is also used to specify the number of data points to plot the coupler and force/torque curves with member functions plotCouplerCurve() and plotForceTorques(). As an example, consider the crank-rocker mechanism described in Problem 3. Program 14 can be used to simulate the motion for each branch of this linkage. Figure 2.14 represents the output of Program 14, which consists of two frames from the two geometric inversions of the fourbar linkage in Problem 3. The third argument of member function setCouplerPoint() is an optional argument to indicate whether or not the coupler curve should be traced in the animation. This argument may be either macro TRACE_ON or TRACE_OFF. For the case of Program 14, tracing of the coupler curve in the animation is desired. by default, the tracing is turned off. Note that if the fourbar is a Grashof Rocker-Rocker, there would exist a total of four branches, since a Rocker-Rocker mechanism has four possible input ranges. If this is true, then the following code fragment would simulate the motion of the third and fourth branch of the fourbar linkage. However, if a third or fourth branch is specified, and the fourbar is not a Grashof Rocker-Rocker, then an error would occur.

```
fourbar.animation(3);
fourbar.animation(4);
```

2.9 Web-Based Fourbar Linkage Analysis

Fourbar linkage analysis can also be done through the World Wide Web. Web pages have been specifically designed to perform the same types of analysis discussed in previous sections. The main web page for mechanism design and analysis through the internet is shown as Figures 2.15. This web page contain links

```
#include <fourbar.h>

int main() {
    CFourbar fourbar;
    double r1 = 0.12, r2 = 0.04, r3 = 0.10,
           r4= 0.07;  //crank-rocker
    double theta1 = 10*M_PI/180;
    double rp = 0.05, beta = 20*M_PI/180;
    string_t fourbartype;

    fourbar.uscUnit(false);
    fourbar.setLinks(r1, r2, r3, r4, theta1);
    fourbar.setCouplerPoint(rp, beta, TRACE_ON);
    fourbar.setNumPoints(50);

    fourbar.animation(1);
    fourbar.animation(2);
}
```

Program 14: Program for simulating the motion of a fourbar linkage.

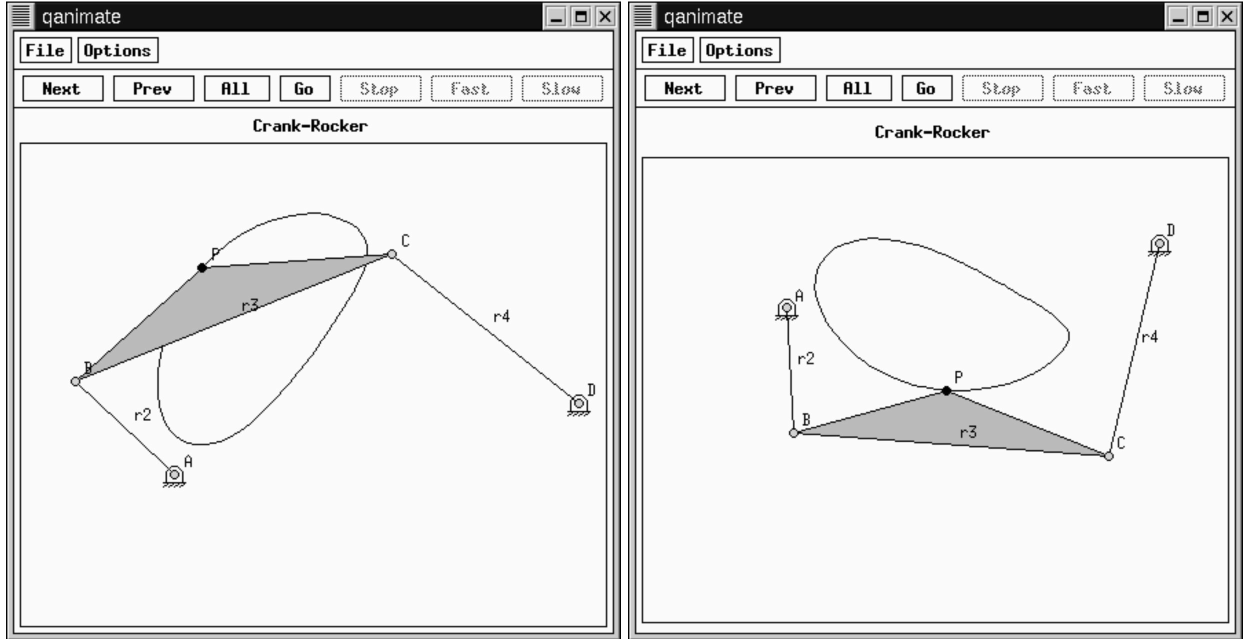


Figure 2.14: The animation from Program 14.

to other web pages that include descriptions and examples of special fourbar linkages as well as links to web pages to perform fourbar linkage analysis and synthesis.

2.9.1 Position Analysis

As an example of using the fourbar linkage analysis web pages, recall the problem statement of Problem 1 in Section 2.1. Given all the necessary link lenghts, r_1 to r_4, and angles θ_1 and θ_2, the problem asked for the values of θ_3 and θ_4. This problem can easily be solved with the "Interactive Four-Bar Linkage Position Analysis" web page and corresponding Ch script, which can be directed by the "Position Analysis" link under the "Four-Bar Linkage Analysis" section shown in Figure 2.15. The web page designed for calculating θ_3 and θ_4 as well as the vector representing the coupler point position is shown in Figure 2.16. The derivation of the analytical solution of θ_3 and θ_4 are provided in the first section of the page. The second section allows the user to input data required for performing the position analysis. In this section, the user can input the link lengths, θ_1, and coupler point parameters r_p and β. Note that angles θ_1 and β can be specified in either degrees or radians with the `Degree Mode/Radian Mode` tab. Furthermore, given any known angle (θ_2, θ_3, θ_4), this web page can be used to calculate the value of the two remaining angles. For this example, $r_1 = 12cm$, $r_2 = 4cm$, $r_3 = 12cm$, $r_4 = 7cm$, $r_p = 5cm$, $\theta_1 = 10^o$, $\theta_2 = 70^o$, and $\beta = 20^o$. After all the parameters have been entered, the user can click on the "Run" button to execute the Ch script for calculating θ_3 and θ_4 as well as the coupler point position. Output of the analysis are displayed directly on the web page, which is shown Figure 2.17. Note that the "Reset" button resets all the inputted values to their default values.

2.9.2 Coupler Curve Plotting

Plotting the coupler curve can be done by the internet web page shown in Figure 2.18. All the user need to do is input the values for the link lengths, θ_1, and coupler point parameters, r_p and β. Once all the values have been entered, the user can click on the "Run" button to plot the coupler curve for the specified fourbar linkage. For example, using the parameters specified in Problem 2, the coupler curve plot for the first geometric inversion is shown as Figure 2.19.

2.9.3 Animation

Figure 2.20 shows the web page that allows users to simulate the motion of a fourbar mechanism via the internet. Like all the other web pages previously mentioned, the user can click on the "Run" button after inputting on the required parameters to generate the animation. Note that the number frames and branch number may be indicated in their specified locations. Using the fourbar specifications of Problem 2, a frame of the animation obtained from the web page is shown in Figure 2.21.

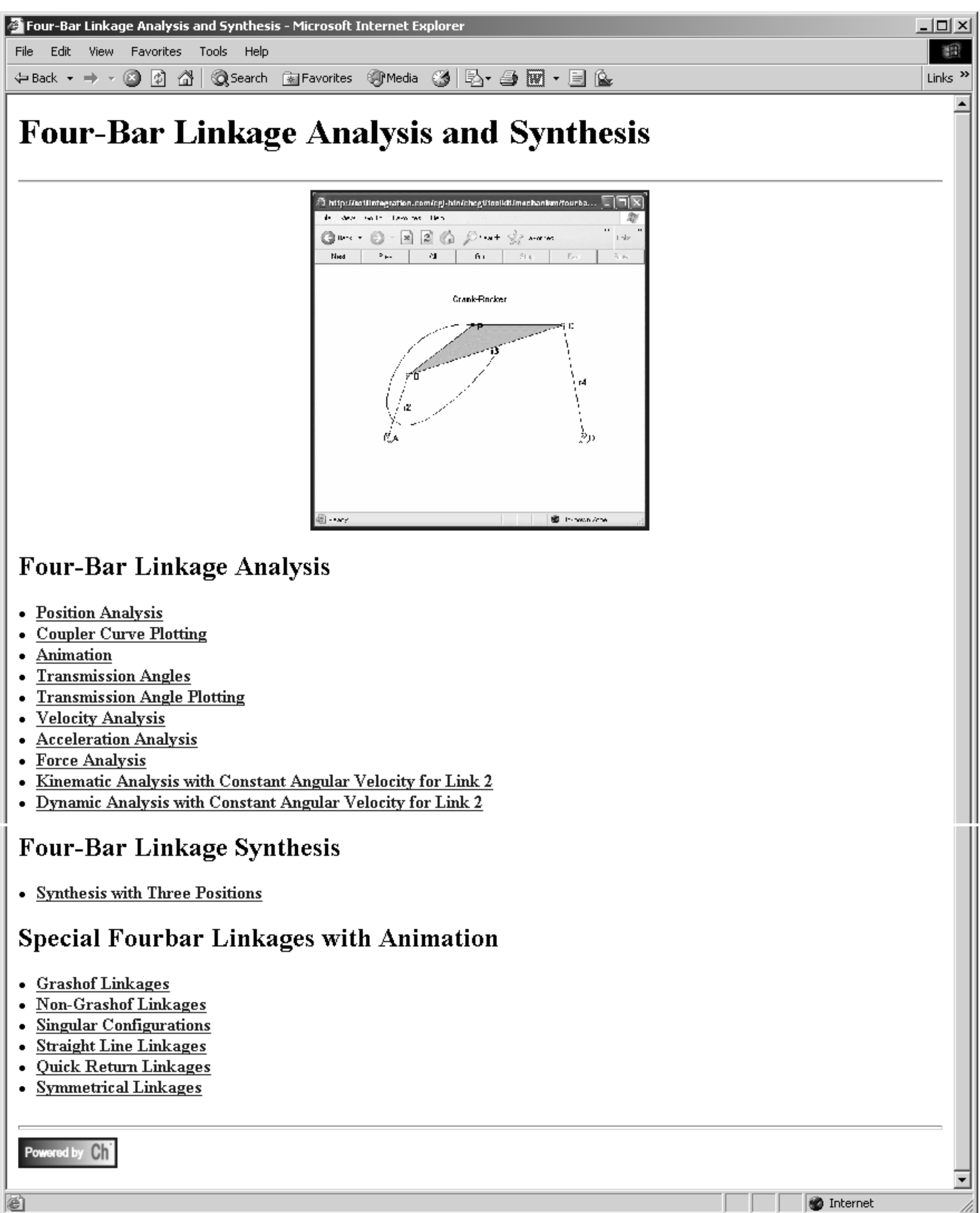


Figure 2.15: Main web page for fourbar linkage analysis and synthesis.

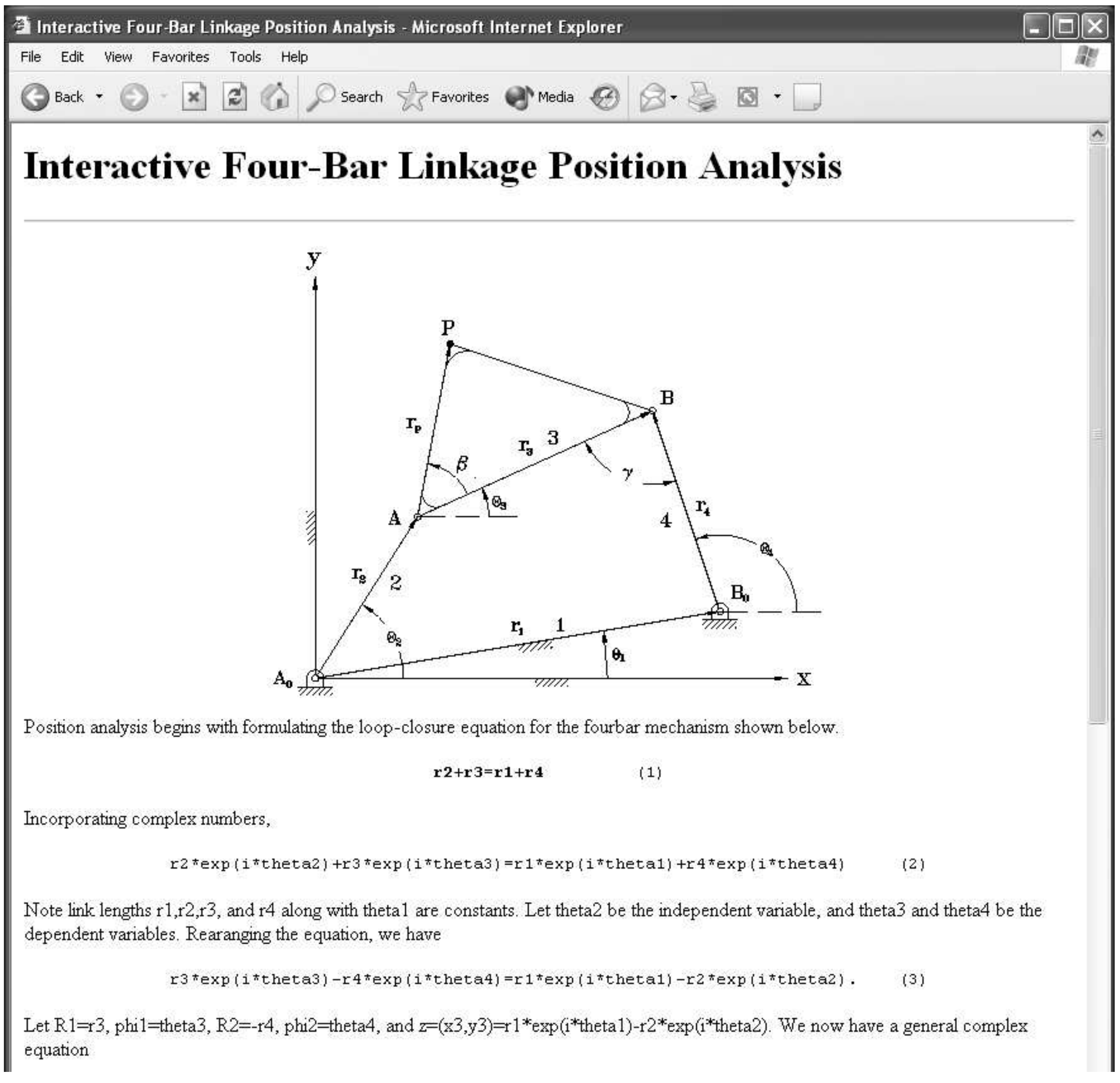


Figure 2.16: Web page for fourbar linkage position analysis.

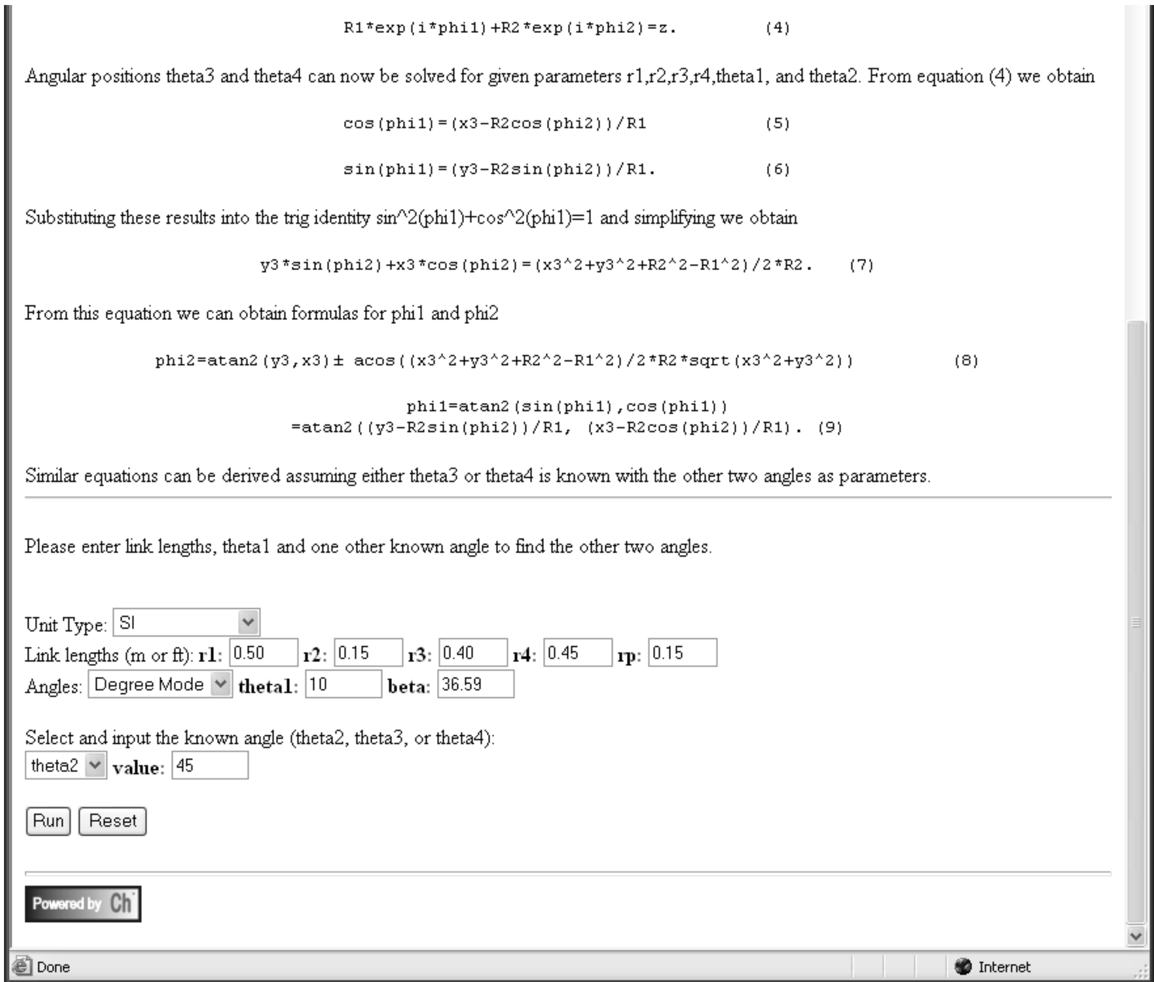


Figure 2.16: Web page for fourbar linkage position analysis (Contd.).

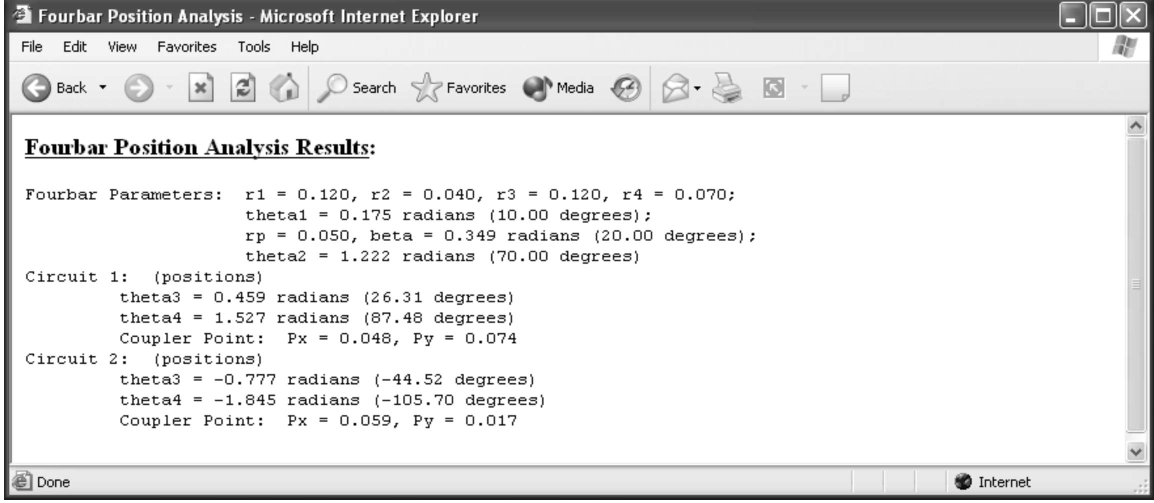


Figure 2.17: Output of fourbar linkage position analysis.

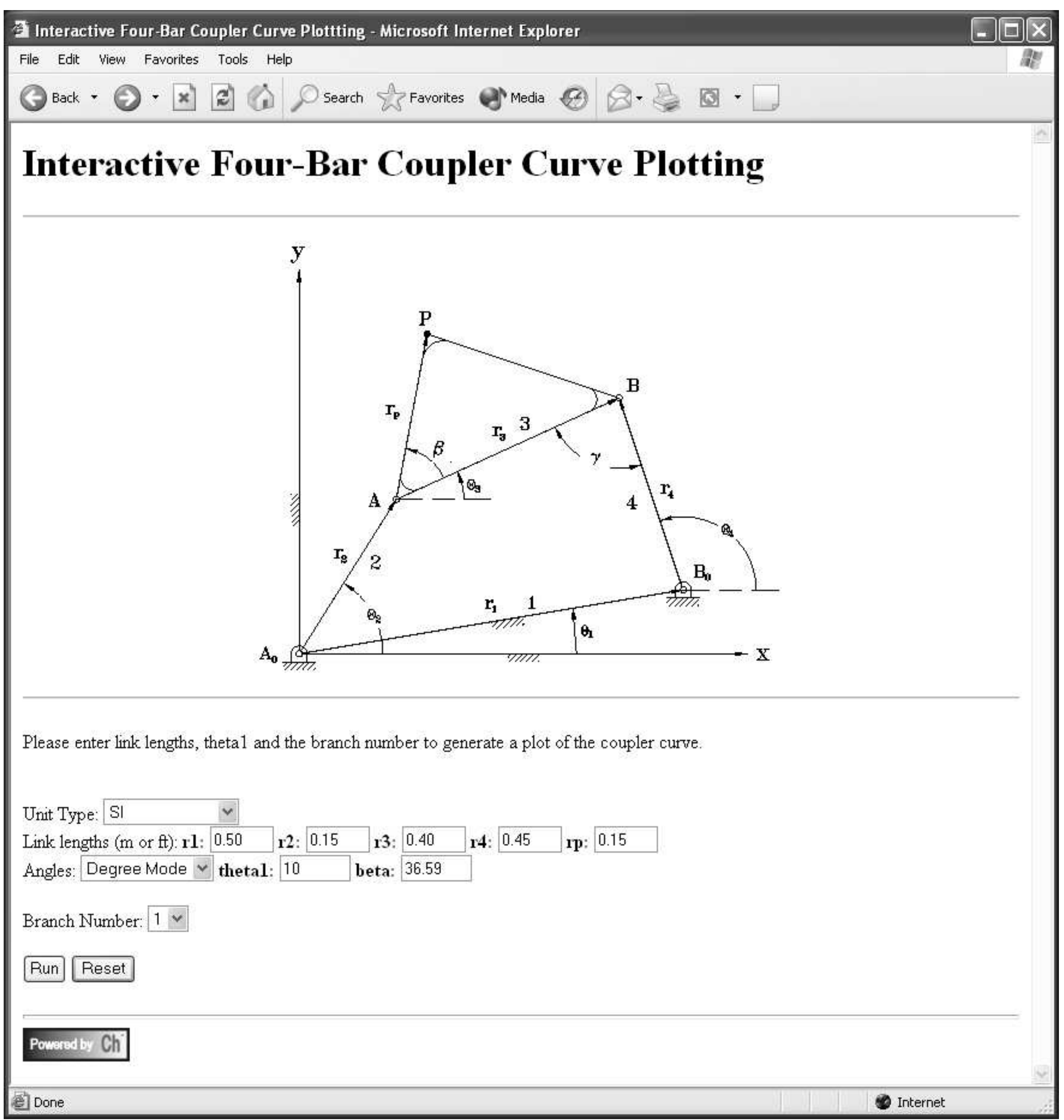


Figure 2.18: Web page for coupler curve plotting.

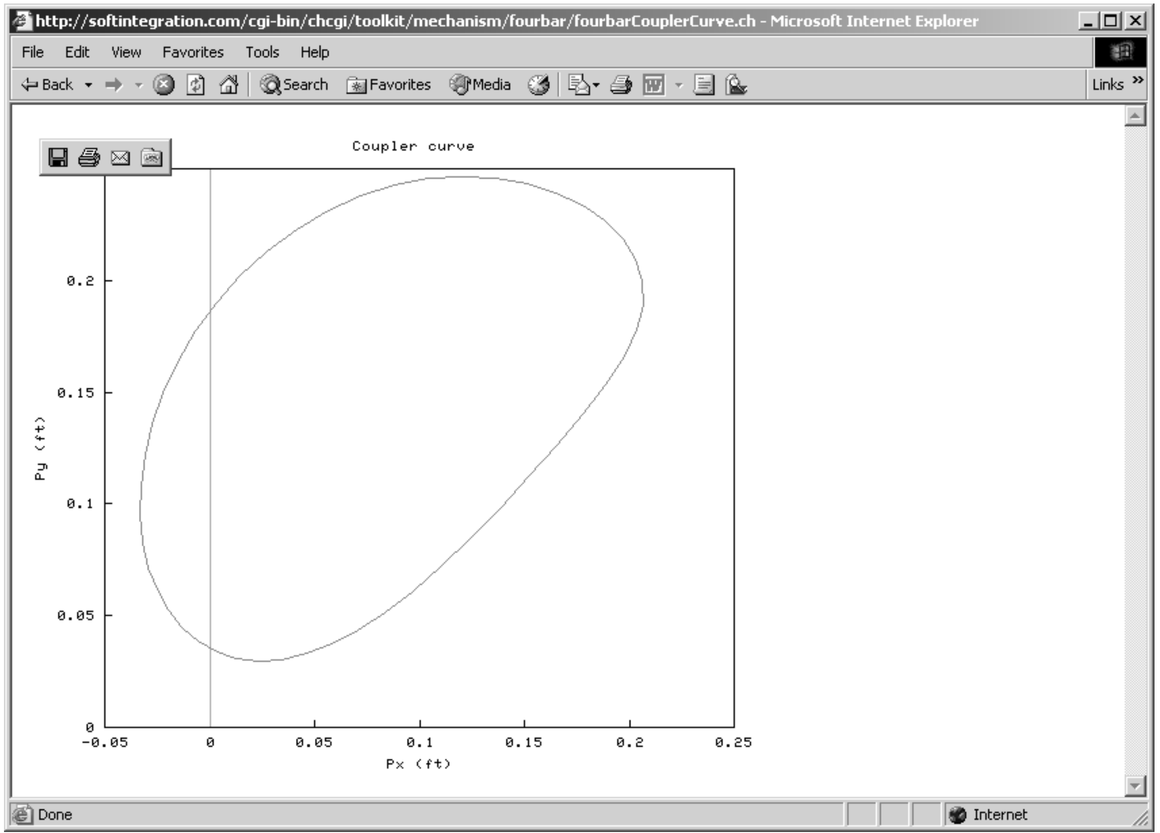


Figure 2.19: Coupler curve plot.

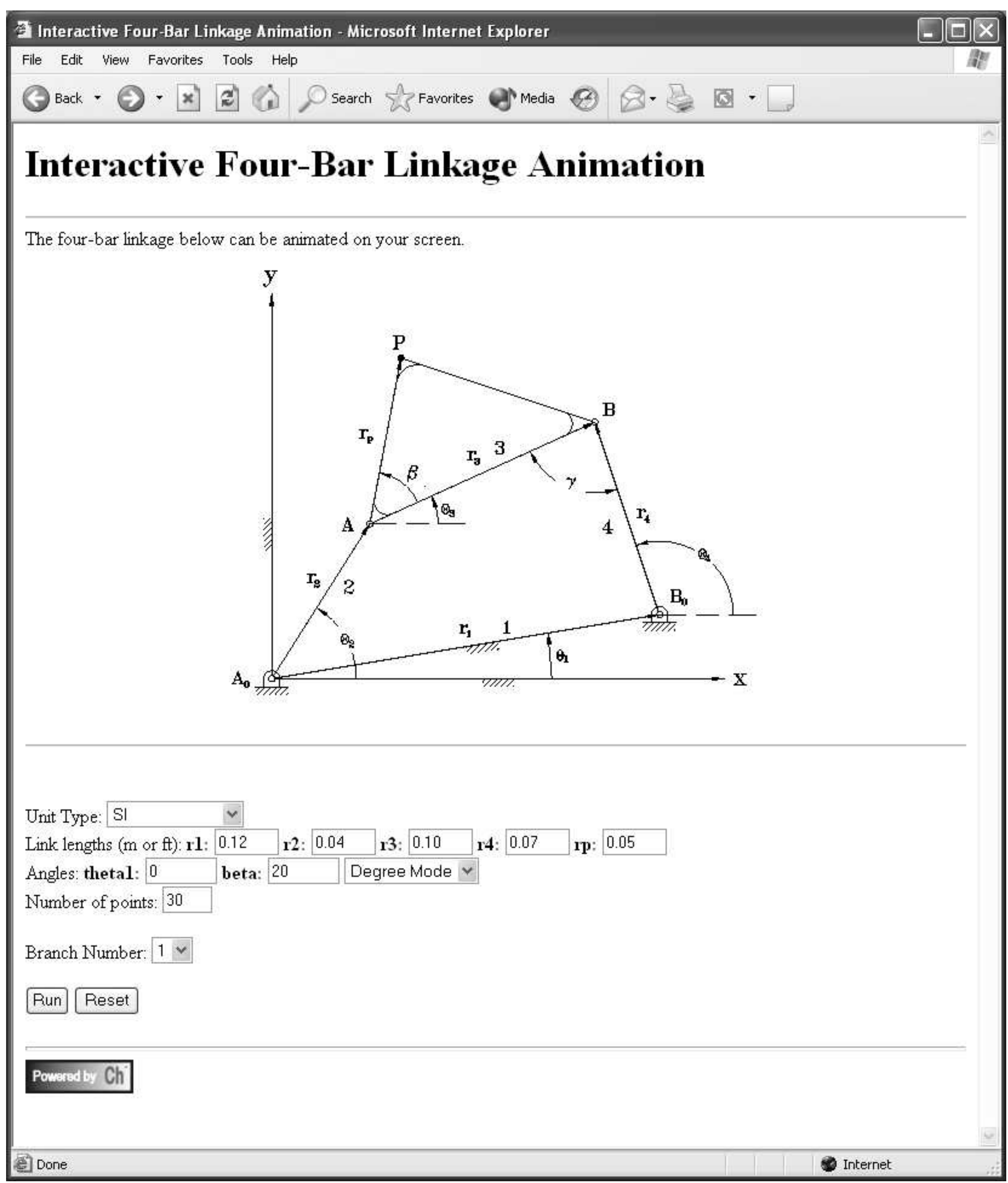


Figure 2.20: Web page for fourbar animation.

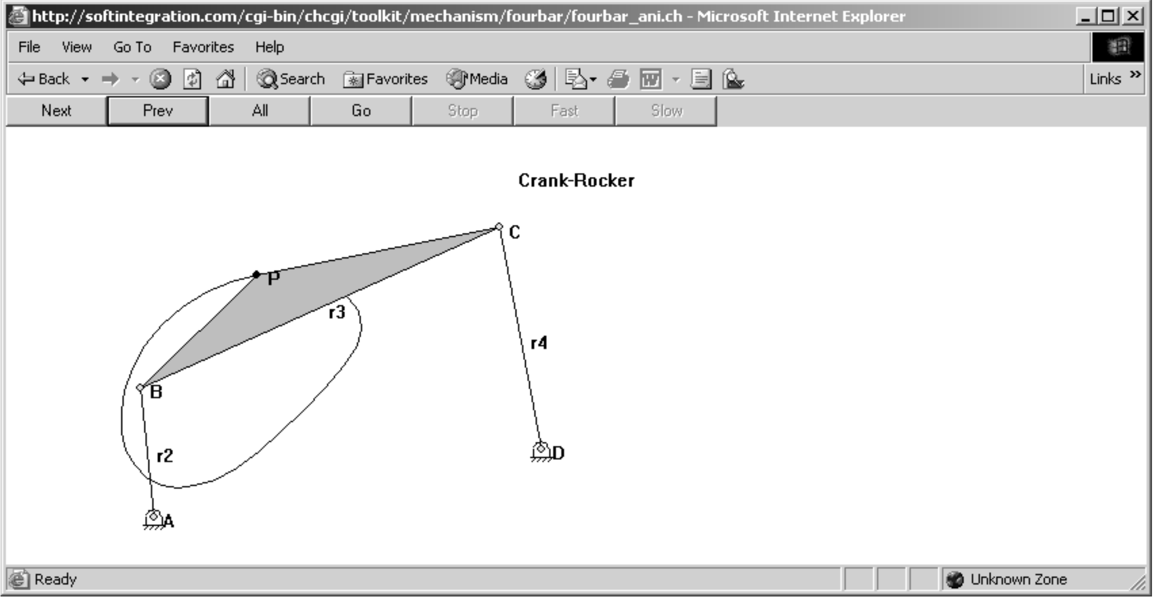


Figure 2.21: Fourbar animation.

Chapter 3

Crank-Slider Mechanism

For a crank-slider mechanism shown in Figure 3.1, the displacement R_1 for the slider has constant angle and variable length. the *crank* R_2, has constant length and variable angle as does R_3, often called the *connecting rod*. The offset R_4 for slider has constant length and angle. It is the variation in the length of R_1 that causes the translational motion. The offset R_4 can be positive or negative. In most applications, the offset is zero. The angle θ_4 is related to θ_1 as $\theta_4 = \theta_1 + 90°$.

Often an intermediate type of motion between the pure rotation of the crank and the pure translation of the slider is desired. This can be obtained at P, the *coupler point*, which is offset from link 3 by a constant length and angle.

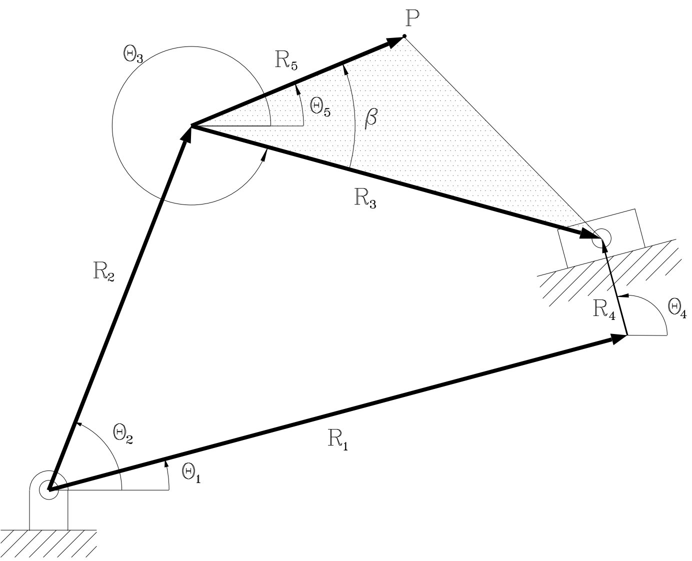


Figure 3.1: A crank-slider mechanism.

In order to have 360° rotation for the crank link, the length of link 3 shall be greater than link 2 and the difference of the link lengths for links 3 and 2 shall be greater than the offset. That is $r_3 > r_2$ and $r_3 - r_2 > r_4$. If r_2 is greater than r_3, the limiting configuration shown in Figure 3.2 will be reached. The value of θ_2 at this limiting configuratoin is

$$\theta_{2_{lim}} = \left[\theta_1 + \sin^{-1}\left(\frac{r_3 + r_4}{r_2}\right)\right] \tag{3.1}$$

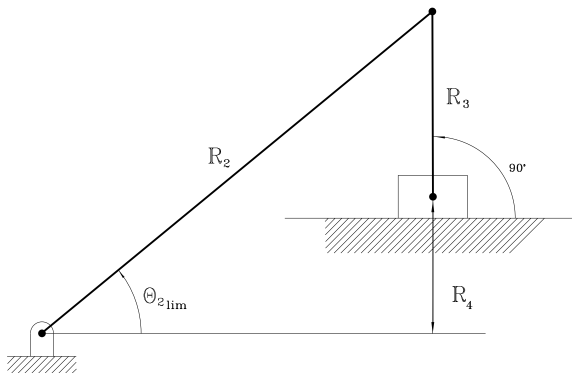


Figure 3.2: A limiting position for a crank-slider mechanism for $r_2 > r_3$.

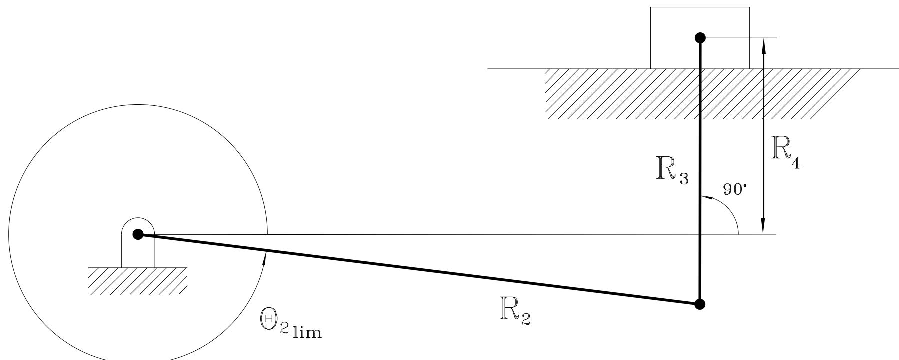


Figure 3.3: A limiting position for a crank-slider mechanism for $r_3 - r_2 < r_4$.

If $r_3 - r_2 < r_4$, a limiting configuration shown in Figure 3.3 will be reached. The value of θ_2 for this limiting configuration will be

$$\theta_{2_{lim}} = \left[\theta_1 + \sin^{-1}\left(\frac{r_3 - r_4}{r_2}\right)\right] \tag{3.2}$$

If $r_2 + r_3 = r_4$, no rotation is possible. If $r_2 + r_3 > r_4$, the links cannot form a valid crank-slider mechanism.

3.1 Position Analysis

Since the vectors describing the four links of the crank-slider mechanism form a complete loop, the describing equation is known as the *loop equation*. The loop equation is the foundation for the analysis of the four bar linkage and its variations. The loop equation is

$$R_2 + R_3 = R_1 + R_4 \tag{3.3}$$

A general vector R can be represented in complex polar with two parameters, a length r and an angle θ. Thus each vector can be represented as

$$R = re^{i\theta} \tag{3.4}$$

In complex polar form the loop equation becomes

$$r_2e^{i\theta_2} + r_3e^{i\theta_3} = r_1e^{i\theta_1} + r_4e^{i\theta_4} \tag{3.5}$$

The first step in describing the motion of the crank slider is deriving the two unknown parameters as functions of the known parameters. At this point θ_1, θ_2, θ_4, r_2, r_3 and r_4 are known. The unknown parameters are r_1 and θ_3. Rearranging equation (3.5) so that the terms containing the unknown parameters are on the left and the terms containing only known parameters are on the right gives

$$r_1 e^{i\theta_1} - r_3 e^{i\theta_3} = r_2 e^{i\theta_2} - r_4 e^{i\theta_4} \tag{3.6}$$

Converting the right side into Cartesian form where

$$a = r_2 \cos\theta_2 - r_4 \cos\theta_4 \tag{3.7}$$

and

$$b = r_2 \sin\theta_2 - r_4 \sin\theta_4 \tag{3.8}$$

gives

$$r_1 e^{i\theta_1} - r_3 e^{i\theta_3} = a + ib \tag{3.9}$$

Multiplying by $e^{-i\theta_1}$ gives

$$r_1 - r_3 e^{i(\theta_3 - \theta_1)} = e^{-i\theta_1}(a + ib) \tag{3.10}$$

and equating the imaginary parts of both sides eliminates r_1 and produces

$$-r_3 \sin(\theta_3 - \theta_1) = a \sin(-\theta_1) + b \cos(-\theta_1) \tag{3.11}$$

Solving for θ_3 yields two solutions below.

$$\theta_3 = \theta_1 + \sin^{-1}\left(\frac{a \sin\theta_1 - b \cos\theta_1}{r_3}\right) \tag{3.12}$$

$$\theta_3 = \theta_1 + \pi - \sin^{-1}\left(\frac{a \sin\theta_1 - b \cos\theta_1}{r_3}\right) \tag{3.13}$$

Equating the real parts of equation (3.9) gives

$$r_1 \cos\theta_1 - r_3 \cos\theta_3 = a \tag{3.14}$$

and solving for r_1 gives

$$r_1 = \frac{a + r_3 \cos\theta_3}{\cos\theta_1} \tag{3.15}$$

The slider position is determined by

$$R_1 + R_4 \tag{3.16}$$

$$r_1 e^{i\theta_1} + r_4 e^{i\theta_1 + 90^\circ} \tag{3.17}$$

The x and y components of the slider position are the real and imaginary parts of equation(3.17).

The joint angle θ_3 and slider position r_1 can be determined by using member functions `angularPos()` and `sliderPos()` of class `CCrankSlider`, respectively. The following problem illustrates how class `CCrankSlider` can be used to calculate these two values.

Problem 1: For the crank-slider mechanism shown in Figure 3.1, if $r_2 = 1cm, r_3 = 5cm, r_4 = 0.5cm$, and $\theta_1 = 10^o$, find the displacement r_1 of the slider and the joint angle θ_3 when $\theta_2 = 45^o$. Display the current position of the crank-slider mechanism.

```
#include <math.h>
#include <crankslider.h>
int main()
{
  CCrankSlider crankslider;
  double r2 = 0.01, r3 = 0.05, r4 = 0.005, theta1 = 10*M_PI/180;
  double theta2 = 45*M_PI/180;
  double first_theta3, sec_theta3;
  double complex s1, s2;  //two solution for slider position

  crankslider.uscUnit(false);
  crankslider.setLinks(r2, r3, r4, theta1);
  crankslider.angularPos(theta2, first_theta3, sec_theta3);
  crankslider.sliderPos(theta2, s1, s2);
  crankslider.displayPosition(theta2, first_theta3);
  crankslider.displayPosition(theta2, sec_theta3);

  /**** the first set of solutions ****/
  printf("theta3 = %6.3f, S = %6.3f cm\n", first_theta3, s1*100);
  /**** the second set of solutions ****/
  printf("theta3 = %6.3f, S = %6.3f cm\n", sec_theta3, s2*100);

  return 0;
}
```

Program 15: Program for computing r_1 and θ_3 of a crank-slider mechanism.

The solution to Problem 1 is Program 15. Since the parameters for the crank-slider mechanism are specified in SI units (centimeters), the input argument for member function `CCrankSlider::uscUnit()` is set to false to indicate the usage of SI units. However, if the crank-slider parameters were specified in feet or inches (US Customary units), then the input value of member function `uscUnit()` would be set to true. By default, input parameters are assumed to be in SI standards, and member function `uscUnit()` does not need to be called unless US Customary standards are desired. Note that the link lenghts for the crank-slider mechanism in Program 15 is converted from centimeter to meter, since meter is the true unit for length in SI terms. After the crank-slider parameters have been set, member functions `angularPos()` and `sliderPos()` are called to calculate angular position, θ_3, and slider position, respectively. Member function `displayPosition()` is called to generate images of the current position of the crank-slider mechanism for both geometric inversions. These images are shown in Figure 3.4. The output of Program 15 is shown below.

```
theta3 =  0.160, S = complex( 5.643, 1.503) cm
theta3 = -2.952, S = complex(-4.204,-0.233) cm
```

The position vector for the slider is represented as a complex number with its real part for the x-component and its imaginary part for the y-component.

> **Problem 2**: For the crank-slider mechanism shown in Figure 3.1, if $r_2 = 0.3937in, r_3 = 1.9685in, r_4 = 0.1969in$, and $\theta_1 = 10^o$, find the displacement r_1 of the slider and the joint angle θ_3 when $\theta_2 = 45^o$.

Now, if the parameters for the crank-slider mechanism defined in Problem 1 were converted to US Customary standard units, as shown in Problem 2, Program 16 can be used to perform the crank-slider analysis. Note that the input value for member function `uscUnit()` is `true` for this program since US Customary units are desired. The output for Program 16 is as follows.

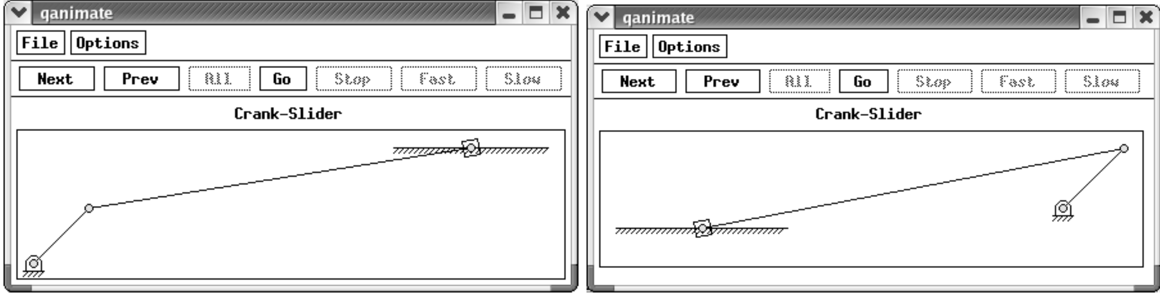


Figure 3.4: Current positions of the crank-slider mechanism.

```
#include <math.h>
#include <crankslider.h>
int main()
{
  CCrankSlider crankslider;
  double r2 = 0.3937/12.0, r3 = 1.9685/12.0, r4 = 0.1969/12.0,
         theta1 = 10*M_PI/180;
  double theta2 = 45*M_PI/180;
  double first_theta3, sec_theta3;
  double complex s1, s2;  //two solution for slider position

  crankslider.uscUnit(true);
  crankslider.setLinks(r2, r3, r4, theta1);
  crankslider.angularPos(theta2, first_theta3, sec_theta3);
  crankslider.sliderPos(theta2, s1, s2);

  /**** the first set of solutions ****/
  printf("theta3 = %6.3f, S = %6.3f in\n", first_theta3, s1*12);
  /**** the second set of solutions ****/
  printf("theta3 = %6.3f, S = %6.3f in\n", sec_theta3, s2*12);

  return 0;
}
```

Program 16: Program for computing r_1 and θ_3 of a crank-slider mechanism in US Customary standard.

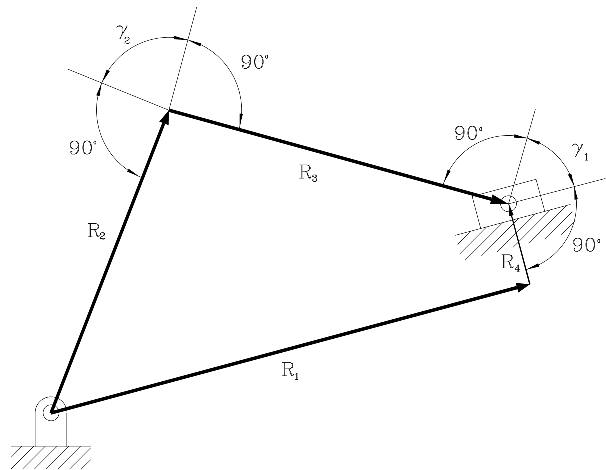


Figure 3.5: Transmission angle for a crank-slider mechanism.

```
theta3 =  0.160, S = complex( 2.222, 0.592) in
theta3 = -2.952, S = complex(-1.655,-0.092) in
```

3.2 Transmission Angles

The transmission angle, γ, is a measure of how effectively force is transmitted to the output link. When γ is 90° all of the force is transmitted to the output link and when γ is 0 no force is transmitted to the output link. In the crank-slider there are two transmission angles; one where the slider is the output link and one where the crank is the output link as shown in Figure 3.5.

If the slider is the output, the transmission angle γ_1 can be obtained by

$$\gamma_1 = 90^\circ - (\theta_3 - \theta_1) \tag{3.18}$$

$$\gamma_1 = 90^\circ + \theta_1 - \theta_3 \tag{3.19}$$

If the crank is the output, the transmission angle γ_2 can be obtained by

$$\gamma_2 = (\theta_2 - \theta_1) + (\theta_3 - \theta_1) \tag{3.20}$$

$$\gamma_2 = \theta_2 + \theta_3 - 2\theta_1 \tag{3.21}$$

The transmission angles can be computed by the member function `transAngle()`, whose prototype is as follow:

```
void CCrankSlider::transAngle(double &gamma1, &gamma2, double theta2);
```

where the first two arguments are the first and second solution to the transmission angle and argument `theta2` is the angular position of link 2.

3.3 Velocity Analysis

The angular and translational velocities are the rates at which the positions of the links change with respect to time.

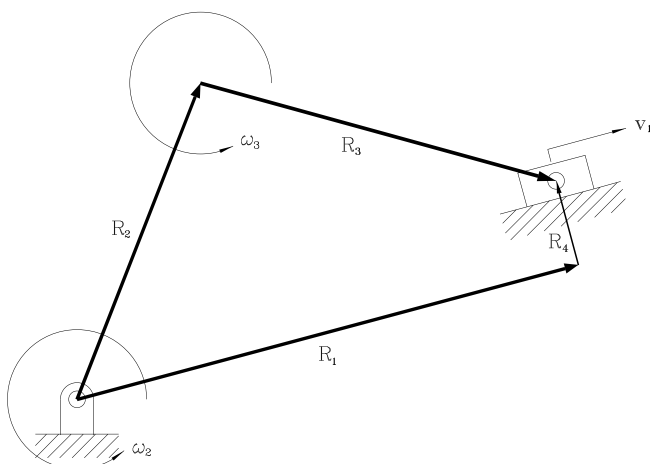


Figure 3.6: Velocities for a crank-slider mechanism.

Beginning with the complex loop equation (3.5), the velocities can be derived by first taking the derivative with respect to time.

$$\frac{d}{dt}(r_2e^{i\theta_2} + r_3e^{i\theta_3}) = \frac{d}{dt}(r_1e^{i\theta_1} + r_4e^{i\theta_4}) \tag{3.22}$$

$$\dot{r}_2e^{i\theta_2} + i\dot{\theta}_2r_2e^{i\theta_2} + \dot{r}_3e^{i\theta_3} + i\dot{\theta}_3r_3e^{i\theta_3} = \dot{r}_1e^{i\theta_1} + i\dot{\theta}_1r_1e^{i\theta_1} + \dot{r}_4e^{i\theta_4} + i\dot{\theta}_4r_4e^{i\theta_4} \tag{3.23}$$

For the crank-slider the parameters r_2, r_3, r_4, θ_1 and θ_4 are constant, thus

$$\dot{r}_2 = \dot{r}_3 = \dot{r}_4 = 0 \text{ and } \dot{\theta}_1 = \dot{\theta}_4 = 0$$

By definition

$$\dot{\theta}_2 \equiv \omega_2 \text{ , } \dot{\theta}_3 \equiv \omega_3 \text{ and } \dot{r}_1 \equiv v_1$$

Thus upon substitution into equation (3.23), we obtain

$$V_1 = v_1e^{i\theta_1} = i\omega_2r_2e^{i\theta_2} + i\omega_3r_3e^{i\theta_3} \tag{3.24}$$

This is the vector describing the velocity of link 1, the slider. In order to find the scalar values ω_3 and v_1 first multiply both sides by $e^{-i\theta_1}$ to produce

$$v_1 = i\omega_2r_2e^{i(\theta_2-\theta_1)} + i\omega_3r_3e^{i(\theta_3-\theta_1)} \tag{3.25}$$

Equating the imaginary parts of both sides in equation (3.25) and solving for ω_3 gives

$$\omega_3 = \frac{-\omega_2r_2\cos(\theta_2 - \theta_1)}{r_3\cos(\theta_3 - \theta_1)} \tag{3.26}$$

The x component of velocity of the slider is the real part of equation (3.24).

$$v_x = \omega_2\sin\theta_2 + \omega_3\sin\theta_3 \tag{3.27}$$

The y component of velocity of the slider is the imaginary part of equation (3.24).

$$v_y = \omega_2\cos\theta_2 + \omega_3\cos\theta_3 \tag{3.28}$$

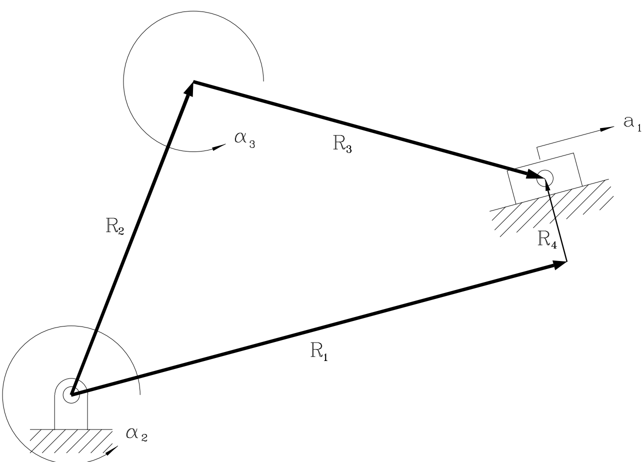


Figure 3.7: Accelerations for a crank-slider mechanism.

The absolute magnitude of V_1 is the real part of equation (3.25).

$$|v_1| = \omega_2 r_2 \sin(\theta_2 - \theta_1) + \omega_3 r_3 \sin(\theta_3 - \theta_1) \tag{3.29}$$

The angular and linear velocities can also be computed by member functions `angularVel()` and `sliderVel()`, respectively.

3.4 Acceleration Analysis

The accelerations of the links is the rate at which their velocities change with respect with time.
The accelerations are derived by taking the derivative with respect to time of the velocity vector V_1, equation (3.24).

$$\frac{d}{dt}V_1 = \frac{d}{dt}v_1 e^{i\theta_1} = \frac{d}{dt}(i\omega_2 r_2 e^{i\theta_2} + i\omega_3 e^{i\theta_3}) \tag{3.30}$$

$$\dot{v}_1 e^{i\theta_1} + i\dot{\theta}_1 v_1 e^{i\theta_1} = i\dot{\omega}_2 r_2 e^{i\theta_2} + i\omega_2 \dot{r}_2 e^{i\theta_2} + i^2\dot{\theta}_2 \omega_2 r_2 e^{i\theta_2} + i\dot{\omega}_3 r_3 e^{i\theta_3} + i\omega_3 \dot{r}_3 e^{i\theta_3} + i^2 \dot{\omega}_3 r_3 e^{i\theta_3} \tag{3.31}$$

By definition

$$\dot{v}_1 \equiv a_1, \dot{\omega}_2 \equiv \alpha_2, \dot{\omega}_3 \equiv \alpha_3 \text{ and } i^2 \equiv -1$$

Substituting these and the previously defined values gives

$$A_1 = a_1 e^{i\theta_1} = i\alpha_2 r_2 e^{i\theta_2} - \omega_2^2 r_2 e^{i\theta} + i\alpha_3 r_3 e^{i\theta_3} - \omega_3^2 r_3 e^{i\theta_3} \tag{3.32}$$

Multiplying equation (3.32) by $e^{-i\theta_1}$ yields

$$a_1 = i\alpha_2 r_2 e^{i(\theta_2-\theta_1)} - \omega_2^2 r_2 e^{i(\theta_2-\theta_1)} + i\alpha_3 r_3 e^{i(\theta_3-\theta_1)} - \omega_3^2 r_3 e^{i(\theta_3-\theta_1)} \tag{3.33}$$

Equating the imaginary parts of equation (3.33) and solving for α_3 gives

$$\alpha_3 = \frac{\omega_2^2 r_2 \sin(\theta_2 - \theta_1) + \omega_3^2 r_3 \sin(\theta_3 - \theta_1) - \alpha_2 r_2 \cos(\theta_2 - \theta_1)}{r_3 \cos(\theta_3 - \theta_1)} \tag{3.34}$$

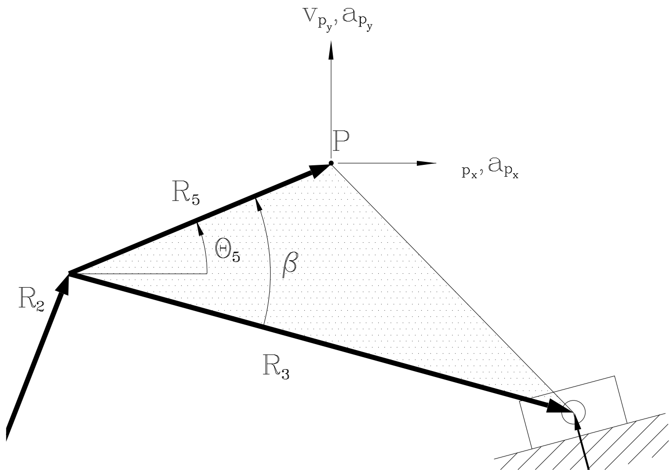


Figure 3.8: Coupler Point of a crank-slider mechanism.

The real part of equation (3.32) gives the x component of the slider acceleration.

$$a_x = \alpha_2 r_2 \sin\theta_2 - \omega_2^2 r_2 \cos\theta_2 + \alpha_3 r_3 \sin\theta_3 - \omega_3^2 r_3 \cos\theta_3 \tag{3.35}$$

The imaginary part of equation (3.32) gives the y component of the slider acceleration.

$$a_y = \alpha_2 r_2 \cos\theta_2 - \omega_2^2 r_2 \sin\theta_2 + \alpha_3 r_3 \cos\theta_3 - \omega_3^2 r_3 \sin\theta_3 \tag{3.36}$$

The real part of equation (3.33) gives the absolute magnitude of the slider acceleration.

$$|a_1| = \alpha_2 r_2 \sin(\theta_2 - \theta_1) - \omega_2^2 r_2 \cos(\theta_2 - \theta_1) + \alpha_3 r_3 \sin(\theta_3 - \theta_1) - \omega_3^2 r_3 \cos(\theta_3 - \theta_1) \tag{3.37}$$

Similar to velocity analysis, the angular and linear accelerations are computed by member functions `angularAccel()` and `sliderAccel()`, respectively.

3.5 Position, Velocity and Acceleration of Coupler Point

The motion of the coupler point is related to that of link 3. Its angle with respect to ground is that of link 3 plus a constant β.

$$\theta_5 = \theta_3 + \beta \tag{3.38}$$

The vector P represents the location of the coupler point.

$$P = R_2 + R_5 = r_2 e^{i\theta_2} + r_5 e^{i(\theta_3+\beta)} \tag{3.39}$$

The x coordinate of P is

$$p_x = r_2 \cos\theta_2 + r_5 \cos(\theta_3 + \beta) \tag{3.40}$$

and the y coordinate is

$$p_y = r_2 \sin\theta_2 + r_5 \sin(\theta_3 + \beta) \tag{3.41}$$

The time derivative of P gives the velocity vector V_p.

$$V_p = \frac{d}{dt}P = i\omega_2 r_2 e^{i\theta_2} + i\omega_3 r_5 e^{i(\theta_3+\beta)} \tag{3.42}$$

The x coordinate of the coupler velocity is

$$v_{p_x} = \omega_2 r_2 \sin\theta_2 + \omega_3 r_5 \sin(\theta_3 + \beta) \tag{3.43}$$

and the y coordinate is

$$v_{p_y} = \omega_2 r_2 \cos\theta_2 + \omega_3 r_5 \cos(\theta_3 + \beta) \tag{3.44}$$

The time derivative of V_p gives the acceleration vector A_p.

$$A_p = \frac{d}{dt} V_p = i\alpha_2 r_2 e^{i\theta_2} - \omega_2^2 r_2 e^{i\theta_2} + i\alpha_3 r_5 e^{i(\theta_3+\beta)} - \omega_3^2 r_5 e^{i(\theta_3+\beta)} \tag{3.45}$$

The x coordinate of the coupler acceleration is

$$a_{p_x} = \alpha_2 r_2 \sin\theta_2 - \omega_2^2 r_2 \cos\theta_2 + \alpha_3 r_5 \sin(\theta_3 + \beta) - \omega_3^2 r_5 \cos(\theta_3 + \beta) \tag{3.46}$$

and the y coordinate is

$$a_{p_y} = \alpha_2 r_2 \cos\theta_2 - \omega_2^2 r_2 \sin\theta_2 + \alpha_3 r_5 \cos(\theta_3 + \beta) - \omega_3^2 r_5 \sin(\theta_3 + \beta) \tag{3.47}$$

The position, velocity, and acceleration of a coupler point can be computed by the following member functions: `couplerPointPos()`, `couplerPointVel()`, `couplerPointAccel()`, respectively. For example, the following problem may be solved with Program 17.

> **Problem 3**: For a crank-slider mechanism with properties $r_2 = 1cm, r_3 = 5cm, r_4 = 0.5cm$, $\theta_1 = 10^o$, $r_p = 2.5cm$, and $\beta = 20^o$, find the position, velocity, and acceleration of the coupler point, P, when $\theta_2 = 45^o$, $\omega_2 = 5rad/sec$, and $\alpha_2 = 0$.

The output of Program 17 is as follows:

```
Circuit 1:
  Coupler Position: complex(2.669,2.257) cm
  Coupler Velocity: complex(-2.266,1.929) cm/s
  Coupler Acceleration: complex(-23.424,-13.111) cm/s^2
Circuit 2:
  Coupler Position: complex(2.574,2.370) cm
  Coupler Velocity: complex(-4.898,5.065) cm/s
  Coupler Acceleration: complex(-0.142,-0.241) cm/s^2
```

3.6 Dynamic Force Analysis

Dynamic force analysis can be performed based on *D'Alembert's Principle – the sum of the inertial or body forces and torques and the external forces and torques together produce equilibrium.* This is analogous to the criteria for static equilibrium as

$$\sum F = 0 \text{ and } \sum M = 0$$

except that now the inertial or dynamic forces must be included in the sum.

The summations can be expressed as

$$\sum F + (-ma_g) = 0 \tag{3.48}$$

and

$$\sum M + (-I\alpha_g) = 0 \tag{3.49}$$

```
#include <math.h>
#include <crankslider.h>

int main() {
    CCrankSlider crankslider;
    double r2 = 0.01, r3 = 0.05, r4 = 0.005, theta1 = 10*M_PI/180;
    double rp = 0.025, beta = 20*M_PI/180;
    double theta2 = 45*M_PI/180;
    double complex P[1:2], Vp[1:2], Ap[1:2];
    double omega2 = 5;  /*  rad/sec  */
    double alpha2 = 0;  /*  rad/sec*sec  */
    double first_theta3, sec_theta3;
    double first_omega3, sec_omega3;
    double first_alpha3, sec_alpha3;


    crankslider.setLinks(r2, r3, r4, theta1);
    crankslider.setCouplerPoint(rp, beta);
    crankslider.angularPos(theta2, first_theta3, sec_theta3);
    crankslider.couplerPointPos(theta2, P[1], P[2]);
    first_omega3 = crankslider.angularVel(theta2, first_theta3, omega2);
    sec_omega3 = crankslider.angularVel(theta2, sec_theta3, omega2);
    Vp[1] = crankslider.couplerPointVel(theta2, first_theta3,
                                        omega2, first_omega3);
    Vp[2] = crankslider.couplerPointVel(theta2, sec_theta3,
                                        omega2, sec_omega3);
    first_alpha3 = crankslider.angularAccel(theta2, omega2,
                                            first_theta3, first_omega3, alpha2);
    sec_alpha3 = crankslider.angularAccel(theta2, omega2,
                                          sec_theta3, sec_omega3, alpha2);
    Ap[1] = crankslider.couplerPointAccel(theta2, first_theta3, omega2,
                                          first_omega3, alpha2,
                                          first_alpha3);
    Ap[2] = crankslider.couplerPointAccel(theta2, sec_theta3, omega2,
                                          sec_omega3, alpha2,
                                          sec_alpha3);

    printf("Circuit 1:\n");
    printf("  Coupler Position: %.3f cm\n", P[1]*100);
    printf("  Coupler Velocity: %.3f cm/s\n", Vp[1]*100);
    printf("  Coupler Acceleration: %.3f cm/s^2\n", Ap[1]*100);
    printf("Circuit 2:\n");
    printf("  Coupler Position: %.3f cm\n", P[2]*100);
    printf("  Coupler Velocity: %.3f cm/s\n", Vp[2]*100);
    printf("  Coupler Acceleration: %.3f cm/s^2\n", Ap[2])*100;

    return 0;
}
```

Program 17: Program for computing the coupler point properties of a crank-slider mechanism.

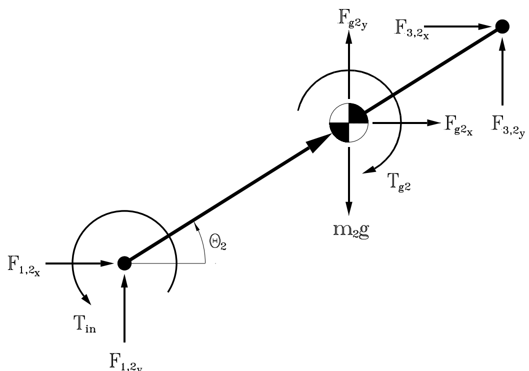


Figure 3.9: The free body diagram for link 2 of a crank-slider mechanism.

In dynamic analysis, we assume that the position, velocity and acceleration of each moving body in the system are already known. The free body diagram for link 2 with all forces acting on it is shown in Figure 3.9.

The real part of the complex acceleration vector at the mass center, A_g, is the x component of acceleration and the imaginary part is the y component.

Acceleration at the mass center can be obtained by

$$\mathbf{A}_{g2} = \frac{d^2}{dt^2}(R_{g2}) = \frac{d^2}{dt^2}(r_{g2}e^{i\theta_2}) \tag{3.50}$$

$$\mathbf{A}_{g2} = i\alpha_2 r_{g2}e^{i\theta_2} - \omega_2^2 r_{g2}e^{i\theta_2} \tag{3.51}$$

Based on the Newton's second law, we can obtain the dynamic equation

$$F_{g2_x} = m_2\text{Re}(A_{g2}) \tag{3.52}$$

$$F_{g2_y} = m_2\text{Im}(A_{g2}) \tag{3.53}$$

$$T_{g2} = I_2\alpha_2 \tag{3.54}$$

where m is the mass of the body and I is the mass moment of inertia about the mass center. The above equation can be expanded as

$$\sum F_x = F_{1,2_x} + F_{3,2_x} + F_{g2_x} = 0 \tag{3.55}$$

$$\sum F_y = F_{1,2_y} + F_{3,2_y} + F_{g2_y} = -m_2 g = 0 \tag{3.56}$$

$$\begin{aligned}\sum M_g \quad = \quad & F_{1,2_x} rg_2 \sin\theta_2 - F_{1,2_y} rg_2 \cos\theta_2 - F_{3,2_x}(r_2 - r_{g2})\sin\theta_2 \\ & + F_{3,2_y}(r_2 - r_{g2})\cos\theta_2 + T_{in} - T_{g2} = 0\end{aligned} \tag{3.57}$$

The free body diagram for link 3 with all forces acting on it is shown in Figure 3.10. Acceleration at the center of the mass for link 3 is

$$\mathbf{A}_{g3} = \frac{d^2}{dt^2}(R_2 + R_{g3}) = \frac{d^2}{dt^2}(r_2 e^{i\theta_2} + r_{g3}e^{i(\theta_3+\phi)}) \tag{3.58}$$

$$\mathbf{A}_{g3} = i\alpha_2 r_2 e^{i\theta_2} - \omega_2^2 r_2 e^{i\theta_2} + i\alpha_3 r_{g3} e^{i(\theta_3+\phi)} - \omega_3^2 r_{g3} e^{i(\theta_3+\phi)} \tag{3.59}$$

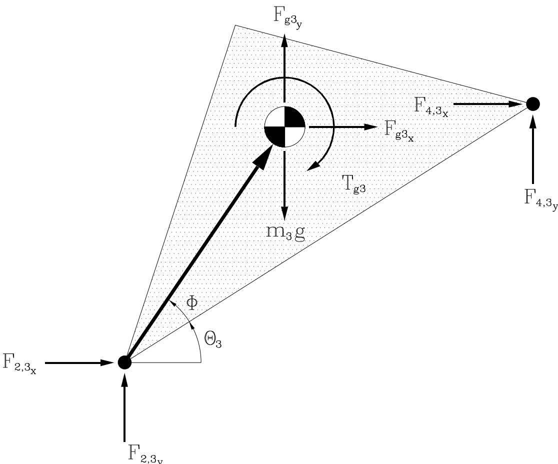


Figure 3.10: The free body diagram for link 3 of a crank-slider mechanism.

The equations of motion

$$F_{g3_x} = m_3\text{Re}(A_{g3}) \tag{3.60}$$

$$F_{g3_y} = m_3\text{Im}(A_{g3}) \tag{3.61}$$

$$T_{g3} = I_3\alpha_3 \tag{3.62}$$

The alternative equations of motion

$$\sum F_x = F_{2,3_x} + F_{4,3_x} + F_{g3_x} = 0 \tag{3.63}$$

$$\sum F_y = F_{2,3_y} + F_{4,3_y} + F_{g3_y} - m_3 g = 0 \tag{3.64}$$

$$\begin{aligned}\sum M_g \quad = \quad & F_{2,3_x} r_{g3}\sin(\theta_3+\phi) - F_{2,3_y} r_{g3}\cos(\theta_3+\phi) - F_{4,3_x}[r_3\sin\theta_3 - r_{g3}\sin(\theta_3+\phi)] \\ & + F_{3,4_y}[r_3\cos\theta_3 - r_{g3}\cos(\theta_3+\phi)] - T_{g3} = 0\end{aligned} \tag{3.65}$$

The free body diagram for link 4 with all forces acting on it is shown in Figure 3.11.
The acceleration at the center of the mass for the slider of link 4 is

$$\mathbf{A}_{g4} = A_1 = i\alpha_2 r_2 e^{i\theta_2} - \omega_2^2 r_2 e^{i\theta_2} + i\alpha_3 r_3 e^{i\theta_3} - \omega_3^2 r_3 e^{i\theta_3} \tag{3.66}$$

The equations of motion

$$F_{g4_x} + F_l\cos\theta_1 = m_4\text{Re}(A_{g4}) \tag{3.67}$$

$$F_{g4_y} + F_l\sin\theta_1 = m_4\text{Im}(A_{g4}) \tag{3.68}$$

$$T_{g2} = 0 \tag{3.69}$$

The alternative equations of motion

$$\sum F_x = F_{3,4_x} + F_{1,4_x} + F_{g4_x} + F_l\cos\theta_1 = 0 \tag{3.70}$$

$$\sum F_y = F_{3,4_y} + F_{1,4_y} - m_4 g + F_{g4_y} + F_l\sin\theta_1 = 0 \tag{3.71}$$

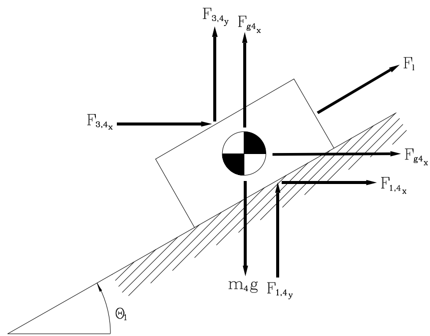


Figure 3.11: The free body diagram for link 4 of a crank-slider mechanism.

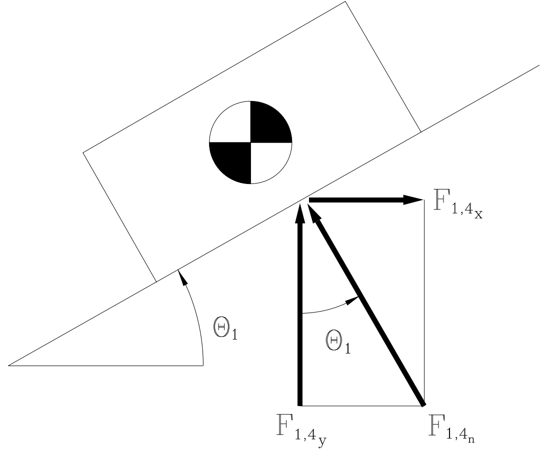


Figure 3.12: The reaction forces on slider.

There is no moment acting on link 4 as its motion is completely translational. The third equation necessary to describe the forces acting on link 4 comes from the fact that $F_{1,4_x}$ and $F_{1,4_y}$ are components of a force $F_{1,4_N}$ that always acts normal to the ground as shown in Figure 3.12.

$$F_{1,4_n}\cos\theta_1 = F_{1,4_x} \tag{3.72}$$

$$-F_{1,4_n}\sin\theta_1 = F_{1,4_y} \tag{3.73}$$

$$F_{1,4_x}\cos\theta_1 = F_{1,4_y}\sin\theta_1 \tag{3.74}$$

$$F_{1,4_x}\cos\theta_1 + F_{1,4_y}\sin\theta_1 = 0 \tag{3.75}$$

Let $F_{3,2} = -F_{2,3}$ and $F_{4,3} = -F_{3,4}$, we can get the following system of equations.

$$\begin{aligned}
F_{g2_x} &= -F_{1,2_x} + F_{2,3_x}\\
F_{g2_y} - m_2 g &= -F_{1,2_y} + F_{2,3_y}\\
-T_{g2} &= -[(r_2 - r_{g2})\sin\theta_2]F_{2,3_x} + [(r_2 - r_{g2})\cos\theta_2]F_{2,3_y}\\
&\quad -(r_{g2}\sin\theta_2)F_{1,2_x} + (r_{g2}\cos\theta_2) - T_{in}\\
F_{g3_x} &= -F_{2,3_x} + F_{3,4_x}\\
F_{g3_y} - m_3 g &= -F_{2,3_y} + F_{3,4_y}\\
-T_{g3} &= -[r_{g3}\sin(\theta_3+\phi)]F_{2,3_x} + [r_{g3}\cos(\theta_3+\phi)]F_{2,3_y}\\
&\quad -[r_3\sin\theta_3 - r_{g3}\sin(\theta_3+\phi)]F_{3,4_x}\\
&\quad +[r_3\cos\theta_3 - r_{g3}\cos(\theta_3+\phi)]F_{3,4_y}\\
F_{g4_x} + F_l\cos\theta_1 &= -F_{3,4_x} - F_{1,4_x}\\
F_{g4_y} - m_4 g + F_l\sin\theta_1 &= -F_{3,4_y} - F_{1,4_y}\\
0 &= \cos\theta_1 F_{1,4_x} + \sin\theta_1 F_{1,4_y}
\end{aligned}$$

For the convenience of equation writing, let

$$\begin{array}{lclclcl}
a &=& -r_{g2}\sin\theta_2 & & f &=& r_{g3}\cos(\theta_3+\phi)\\
b &=& r_{g2}\cos\theta_2 & & g &=& -(r_3\sin\theta_3 - r_{g3}\sin(\theta_3+\phi))\\
c &=& -(r_2 - r_{g2})\sin\theta_2 & & h &=& r_3\cos\theta_3 - r_{g3}\cos(\theta_3+\phi)\\
d &=& (r_2 - r_{g2})\cos\theta_2 & & i &=& \cos\theta_1\\
e &=& -r_{g3}\sin(\theta_3+\phi) & & j &=& \sin\theta_1
\end{array}$$

the system of equations in a matrix form can be obtained as follows.

$$\begin{bmatrix} F_{g2_x}\\ F_{g2_y} - m_2 g\\ -T_{g2}\\ F_{g3_x}\\ F_{g3_y} - m_3 g\\ -T_{g3}\\ F_{g4_x} + F_l\cos\theta_1\\ F_{g4_y} - m_4 g + F_l\sin\theta_1\\ 0 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0\\ 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0\\ a & b & c & d & 0 & 0 & 0 & 0 & -1\\ 0 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & -1 & 0 & 1 & 0 & 0 & 0\\ 0 & 0 & e & f & g & h & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & -1 & 0 & -1 & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 0\\ 0 & 0 & 0 & 0 & 0 & 0 & i & j & 0 \end{bmatrix} \begin{bmatrix} F_{1,2_x}\\ F_{1,2_y}\\ F_{2,3_x}\\ F_{2,3_y}\\ F_{3,4_x}\\ F_{3,4_y}\\ F_{1,4_x}\\ F_{1,4_y}\\ T_{in} \end{bmatrix}$$

The matrix equation is of the form

$$\mathbf{b} = \mathbf{C}\mathbf{x} \tag{3.76}$$

where $\mathbf{b}$ and $\mathbf{C}$ are known and $\mathbf{x}$ ontains the unknown values to be solved. The solution is of the form

$$\mathbf{x} = \mathbf{C}^{-1}\mathbf{B} \tag{3.77}$$

which can be conveniently solved by function `inverse()` in Ch. Member function `forceTorque()` with prototype

```
void CCrankSlider::forceTorque(double theta2, theta3, omega2, omega3,
                               alpha2, alpha3, Fl, double complex as,
                               array double x[9]);
```

can be used to perform the above analysis, where the first six arguments correspond to the angular properties of link 2 and 3, `Fl` is the load force, `as` is the acceleration of the slider, and `x` is an array that contains the joint forces and output torque.

Problem 4: Link parameters and inertia properties of a crank-slider mechanism are given in the chart below.

Link	Length r (cm)	Mass (kg)	I_g $(\mathrm{kg}-\mathrm{m}^2)$	C. G. r_g (cm)	δ
1	2.54	—	—	—	—
2	5.08	0.3626	0.0014	0.0508	0
3	1.27	1.0877	0.0134	0.1524	0
4	—	0.6345	—	—	0

The phase angle for link 1 is $\theta_1 = 10^o$. There is no external load. At one point the input angular position $\theta_2 = 45°$, angular velocity $\omega_2 = 5$ rad/sec ccw and angular acceleration $\alpha_2 = -5$ rad/sec^2 cw, find the joint reaction forces and required input torque at this moment.

The solution to Problem 4 is listed as Program 18. It utilizes the crank-slider class and various member functions to calculate the joint reaction forces and required input torque for the crank-slider mechanism defined in the above problem statement when $\theta_2 = 45^o$. After the specifying the required parameters for the crank-slider, member functions `angularPos()`, `angularVel()`, `angularAccel()`, and `sliderAccel()` are called to determine the values needed for member function `forceTorque()`, which calculates the joint forces and required input torque. The output of Program 18 is as follows.

```
first solution X = 8.4397 -16.7026 8.1792 -20.6504 6.8925 -30.5133 -6.4904 36.8086 -0.3923

second solution X = 6.8260 -16.3604 6.5655 -20.3082 6.5401 -29.9487 -6.3832 36.2008 -0.3572
```

3.7 Animation

Similar to the fourbar mechanism discussed in the previous chapter, motion of the crank-slider mechanism can also be simulated with the `animation()` function of class `CCrankSlider`. The prototype for member function `animation()` is as follows.

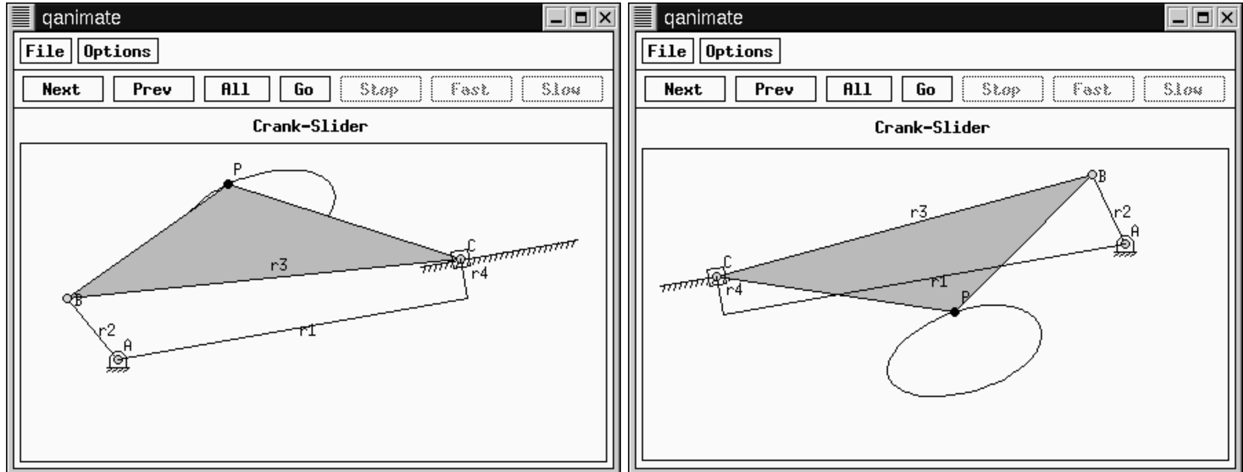
```
int CCrankSlider::animation(int branchnum, ...);
```

```
#include <crankslider.h>

int main() {
    CCrankSlider crankslider;
    double r2 = 0.0254, r3 = 0.0508, r4 = 0.0127, theta1 = 10*M_PI/180;
    array double X[9];
    double rg2 = 0.0508, rg3 = 0.1524;
    double m2 = 0.3626, m3 = 1.0877, m4 = 0.6345;
    double ig2 = 0.0014, ig3 = 0.0134, Fl=0;
    double theta2 = 45*M_PI/180;
    double omega2 = 5;  /*  rad/sec  */
    double alpha2 =-5;  /*  rad/sec*sec  */
    double first_theta3, sec_theta3;
    double first_omega3, sec_omega3;
    double first_alpha3, sec_alpha3;
    double complex first_as, sec_as;

    /* initialization of link parameters and
    inertia properties */
    crankslider.setLinks(r2, r3, r4, theta1);
    crankslider.setGravityCenter(rg2, rg3);
    crankslider.setInertia(ig2, ig3);
    crankslider.setMass(m2, m3, m4);

    crankslider.angularPos(theta2, first_theta3, sec_theta3);
    first_omega3 = crankslider.angularVel(theta2, first_theta3, omega2);
    sec_omega3 = crankslider.angularVel(theta2, sec_theta3, omega2);
    first_alpha3 = crankslider.angularAccel(theta2, omega2, first_theta3, first_omega3,
                                            alpha2);
    sec_alpha3 = crankslider.angularAccel(theta2, omega2, sec_theta3, sec_omega3,
                                          alpha2);
    first_as = crankslider.sliderAccel(theta2, first_theta3, omega2, first_omega3,
                                       alpha2, first_alpha3);
    sec_as = crankslider.sliderAccel(theta2, sec_theta3, omega2, sec_omega3, alpha2,
                                     sec_alpha3);

    crankslider.force(theta2, first_theta3, omega2, first_omega3, alpha2,
                      first_alpha3, Fl, first_as, X);
    printf("first solution X = %.4f \n", X);

    crankslider.force(theta2, sec_theta3, omega2, sec_omega3, alpha2, sec_alpha3,
                      Fl, sec_as, X);
    printf("second solution X = %.4f \n", X);
}
```

Program 18: Program for computing the joint reaction forces and required input torque of a crank-slider mechanism.

```
#include <stdio.h>
#include <crankslider.h>

int main() {
    /* default specification of the four-bar linkage */
    double  r2 = 0.1, r3 = 0.5, r4= 0.05;//cranker-rocker
    double theta1 = 10*M_PI/180;
    double rp = 0.25, beta = 30*M_PI/180;
    int numpoints = 50;
    CCrankSlider crankslider;

    crankslider.setLinks(r2, r3, r4, theta1);
    crankslider.setCouplerPoint(rp, beta, TRACE_ON);
    crankslider.setNumPoints(numpoints);
    crankslider.animation(1);
    crankslider.animation(2);
}
```

Program 19: Program for simulating the motion of a crank-slider mechanism.



It has the same features and functionality as that of function `animation()` of class `CFourbar`. Argument `branchnum` indicates which geometric inversion of the crank-slider to animate. The animation data can also be saved into a file similar to the method discussed in Section 2.8. For example, the following problem will illustrate how to animate a crank-slider linkage with the given parameters.

> **Problem 5**: For the crank-slider mechanism shown in Figure 3.1, given $r_2 = 1cm, r_3 = 5cm, r_4 = 0.5cm$, $\theta_1 = 10^o$, $r_p = 2.5cm$, and $\beta = 30^o$, simulate the motion of this mechanism. Do this for both geometric inversions.

The solution to the above problem is Program 19. After setting the parameters for the crank-slider mechanism with member functions `setLinks()`, `setCouplerPoint()`, and `setNumPoints()`, member function `animation()` is called twice to simulate the motion of the crank-slider mechanism. Each function call corresponds to each geometric inversion. Frames from each animation are displayed in Figure 3.7.

3.8 Web-Based Crank-Slider Linkage Analysis

The main web page for the analysis and synthesis of a crank-slider mechanism is shown in Figure 3.13. It contain links to web pages for analysis of the slider, coupler point, and angular properties. Furthermore, there are web pages to calculate the transmission angle of a crank-slider and animate it as well. For example, the web page shown in Figure 3.14 can be used to calculate the angular position of a crank-slider mechanism. This page also contains the derivation for calculating θ_3, so that the user may understand the steps taken by the web-based solver.

As an example, recall the crank-slider mechanism defined in the previous problems, where $r_2 = 1cm$, $r_3 = 5cm$, $r_4 = 0.5cm$, $\theta_1 = 10^o$, $r_p = 2.5cm$, and $\beta = 30^o$. Given $\theta_2 = 45^o$, the angular position of link 3, θ_3, can be solved by using the web page shown in Figure 3.14, which can be accessed by clicking on the link after the derivation for θ_3. After inputting the appropriate data, the output is shown in Figure 3.15.

Furthermore, analysis of the slider component can also be performed on the crank-slider mechanism mentioned above. Figure 3.16 shows the slider position analysis web page, which again includes the derivation for determining r_1. Note that the input parameters are the same as that of the page for solving for θ_3. After executing the script, the output is shown in Figure 3.17.

For analysis of the coupler point, the web pages shown in Figures 3.18 - 3.22 can be used. Similar to the other web pages, these pages also contain derivations for calculation the coupler point position, velocity, and acceleration. Entering the crank-slider parameters given by prior problem statements into Figure 3.18 results in the output shown in Figure 3.19. Now, for $\omega_2 = 5rad/sec$, the velocity of the coupler point can be calculated by web the page of Figure 3.20 with the output shown in Figure 3.21. Similarly, if $\alpha_2 = 0$, the coupler point acceleration can be determined by Figure 3.22. The output is shown as Figure 3.23.

Simulating the motion of the crank-slider mechanism may also be accomplished through web-based analysis. Figure 3.24 is the HTML document for inputting data to animate the crank-slider linkage. Asides from the link lengths and required angles, the user may also specify the number of animation frames to generate, and the geometric inversion of the crank-slider mechanism. Using parameters specified in problem statements of previous sections, Figure 3.25 is a snapshot of animation for the first geometric inversion of the crank-slider mechanism.

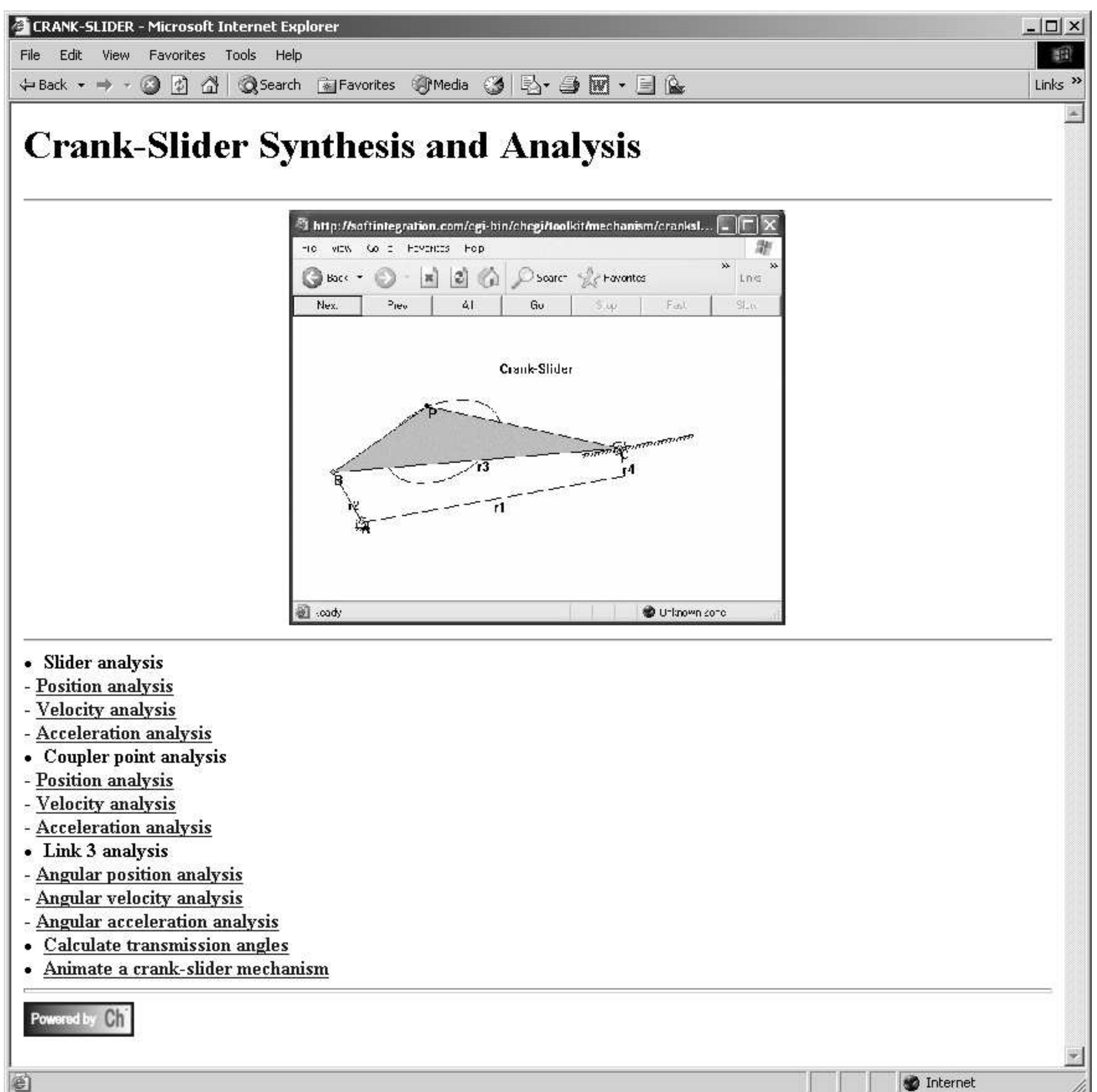


Figure 3.13: Main page for web-based crank-slider analysis.

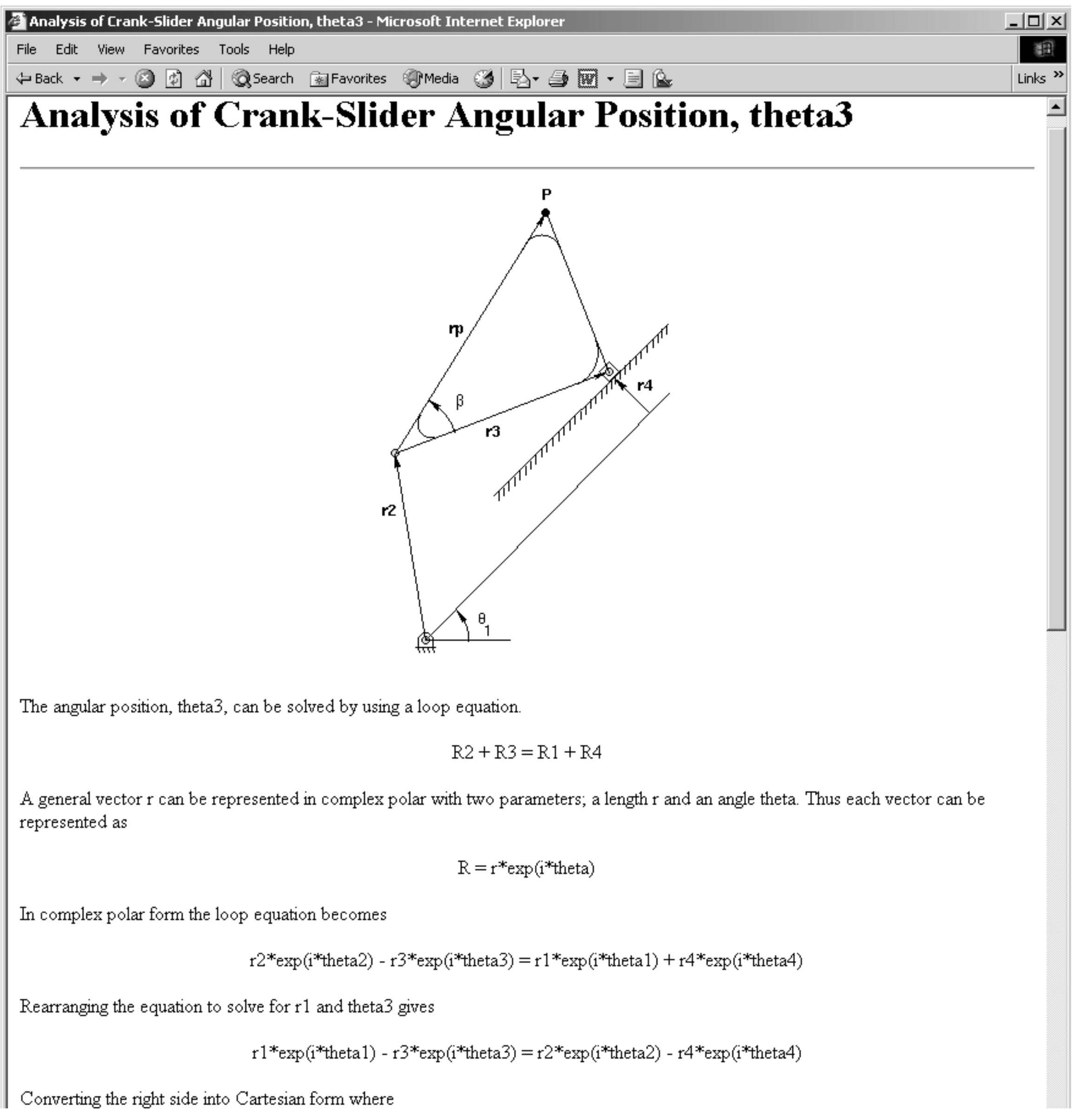


Figure 3.14: Slider-crank angular position analysis.

a=r2*cos(theta2) - r4*cos(theta4)

and

b=r2*sin(theta2) - r4*sin(theta4)

gives

r1*exp(i*theta1) - r3*exp(i*theta3) = a + i*b

Multiplying by exp(-i*theta1) gives

r1 - r3*exp[i(theta3-theta1)] = exp(-i*theta1)*(a + i*b)

and equating the imaginary parts of both sides eliminates r1 and produces

-r3*sin(theta3-theta1) = a*sin(-theta1) + b*cos(-theta1)

Solving for theta3 yields two solutions

theta3 = theta1 + [sin{(a*sin(theta1)-b*cos(theta1))/r3}]^-1

theta3 = theta1 + PI - [sin{(a*sin(theta1)-b*cos(theta1))/r3}]^-1

Please enter the data to calculate the angular position of link 3.

Unit Type: SI
Link lengths (m or ft): **r2:** 0.10 **r3:** 0.20 **r4:** 0 **rp:** 0.25
Angles: **theta1:** 45 **theta2:** 45 **beta:** 37 Degree Mode

Output option:

1. Display angular position
2. Display crank-slider position -- Branch Number: 1

Run Reset

Powered by Ch

Done Internet

Figure 3.14: Slider-crank angular position analysis (Contd.).

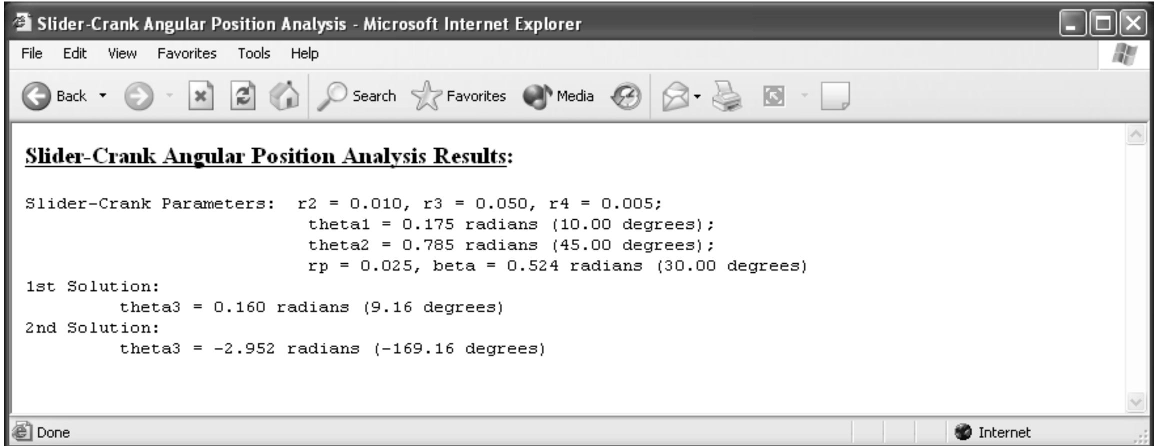


Figure 3.15: Output of angular position analysis.

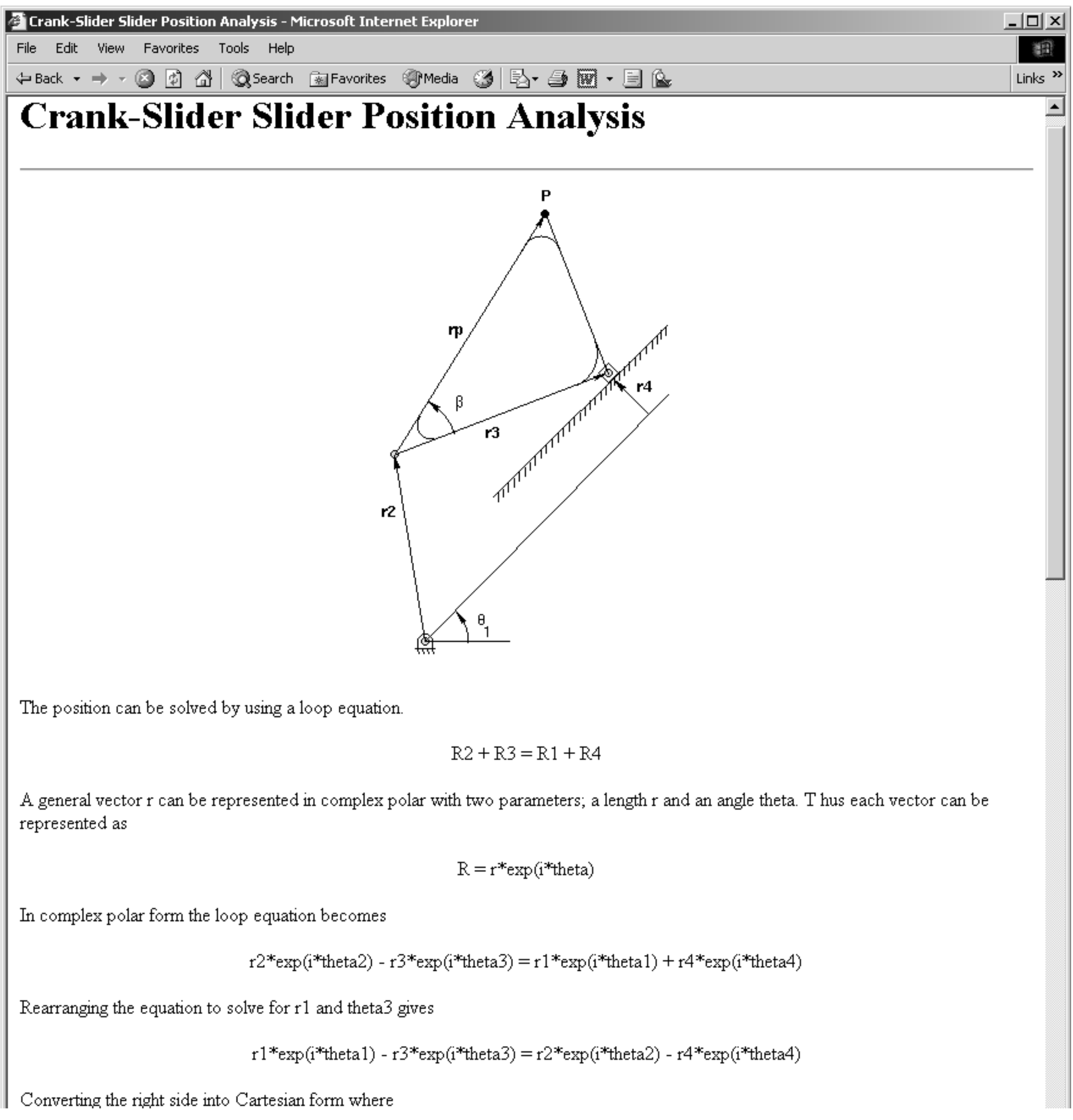


Figure 3.16: Slider position analysis.

a=r2*cos(theta2) - r4*cos(theta4)

and

b=r2*sin(theta2) - r4*sin(theta4)

gives

r1*exp(i*theta1) - r3*exp(i*theta3) = a + i*b

Multiplying by exp(-i*theta1) gives

r1 - r3*exp[i(theta3-theta1)] = exp(-i*theta1)*(a + i*b)

and equating the imaginary parts of both sides eliminates r1 and produces

-r3*sin(theta3-theta1) = a*sin(-theta1) + b*cos(-theta1)

Solving for theta3 yields two solutions

theta3 = theta1 + [sin{(a*sin(theta1)-b*cos(theta1))/r3}]^-1

theta3 = theta1 + PI - [sin{(a*sin(theta1)-b*cos(theta1))/r3}]^-1

Equating the real parts of the equation gives

r1*cos(theta1) - r3*cos(theta3) = a

and solving for r1 gives

r1 = [a + r3*cos(theta3)]/cos(theta1)

The slider position is defined as

R1 + R4

r1*exp(i*theta1) + r4*exp(i*theta1 + 90)

The x and y components of the slider are the real and imaginary parts of the equation.

Please enter the data to find the position of the slider.

Unit Type: SI
Link lengths (m or ft): **r2:** 0.1 **r3:** 0.2 **r4:** 0 **rp:** 0.25
Angles: **theta1:** 45 **theta2:** 45 **beta:** 37 Degree Mode

Run Reset

Powered by Ch

Done Internet

Figure 3.16: Slider position analysis (Contd.).

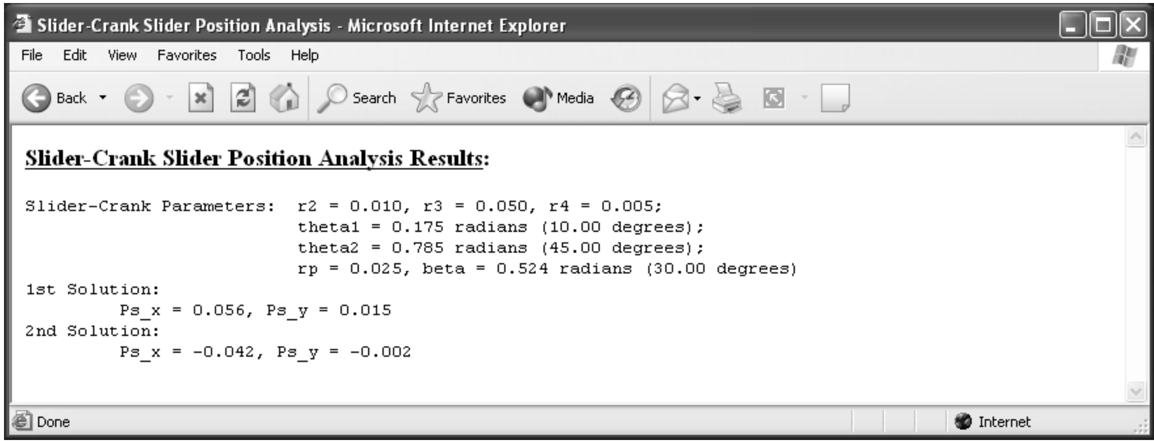


Figure 3.17: Output of slider position analysis.

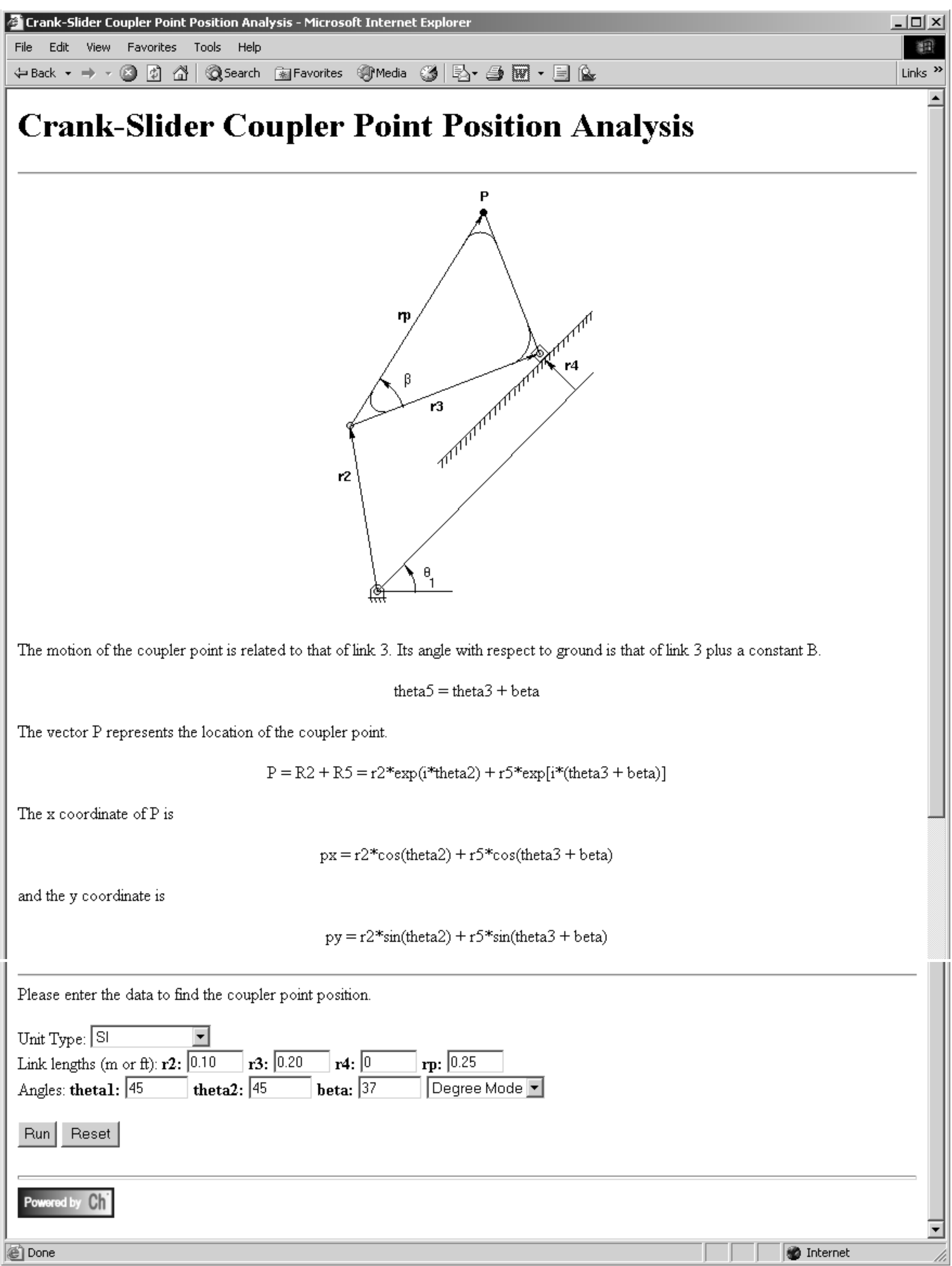


Figure 3.18: Coupler point position analysis.

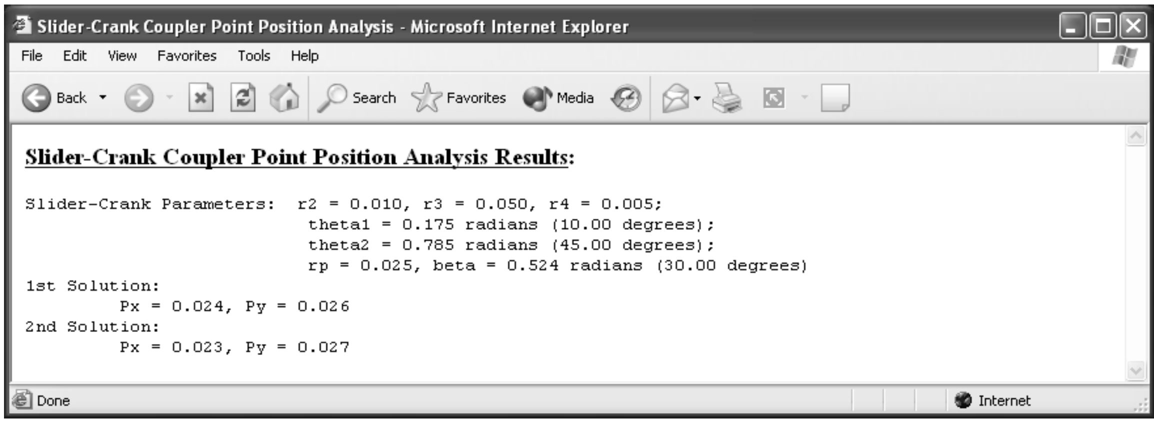


Figure 3.19: Output of coupler point position analysis.

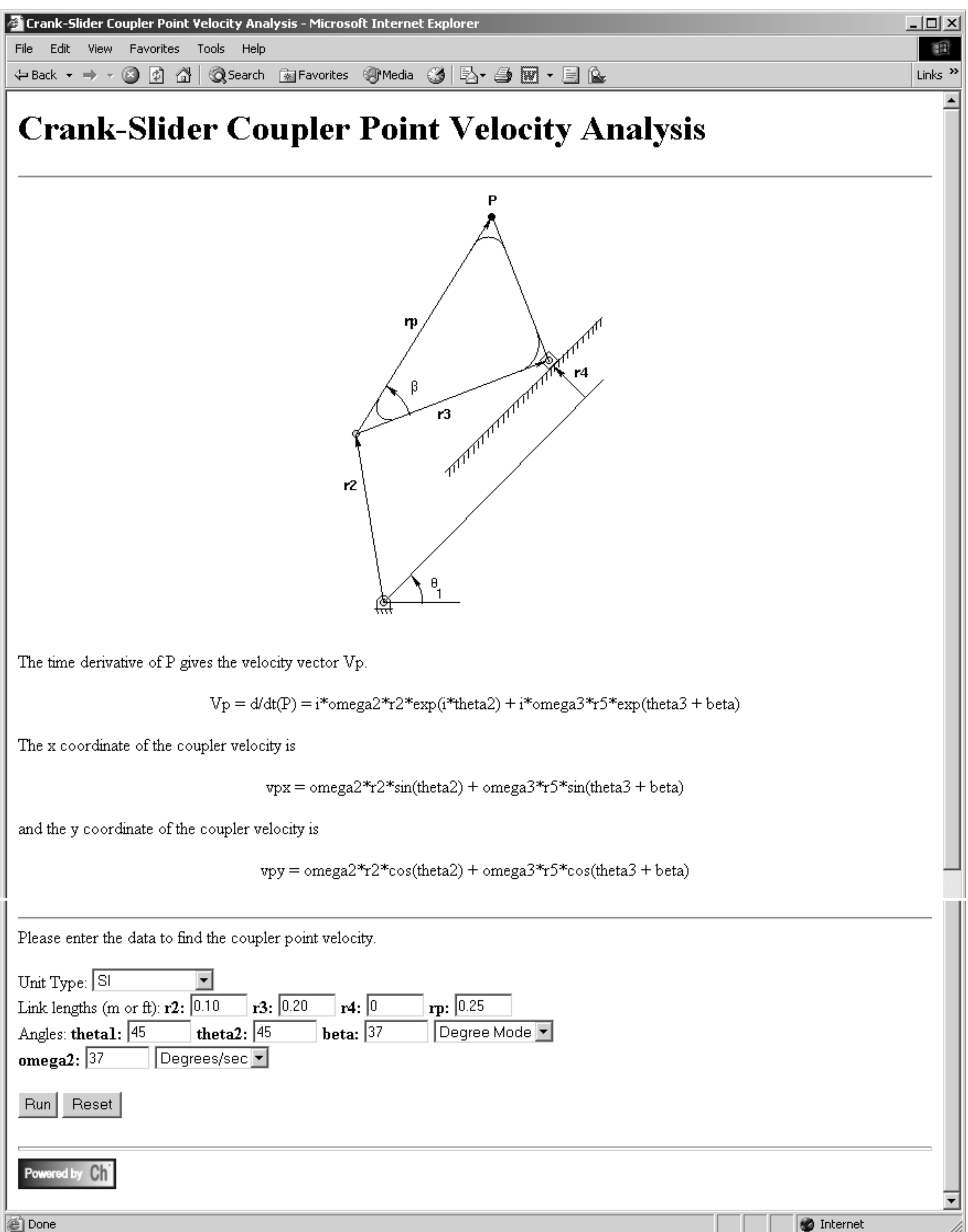


Figure 3.20: Coupler point velocity analysis.

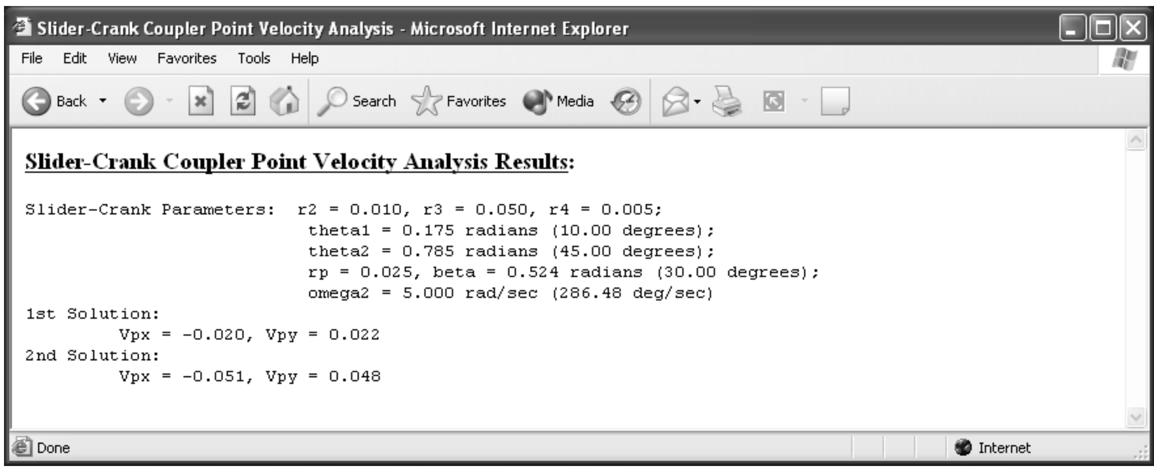


Figure 3.21: Output of coupler point velocity analysis.

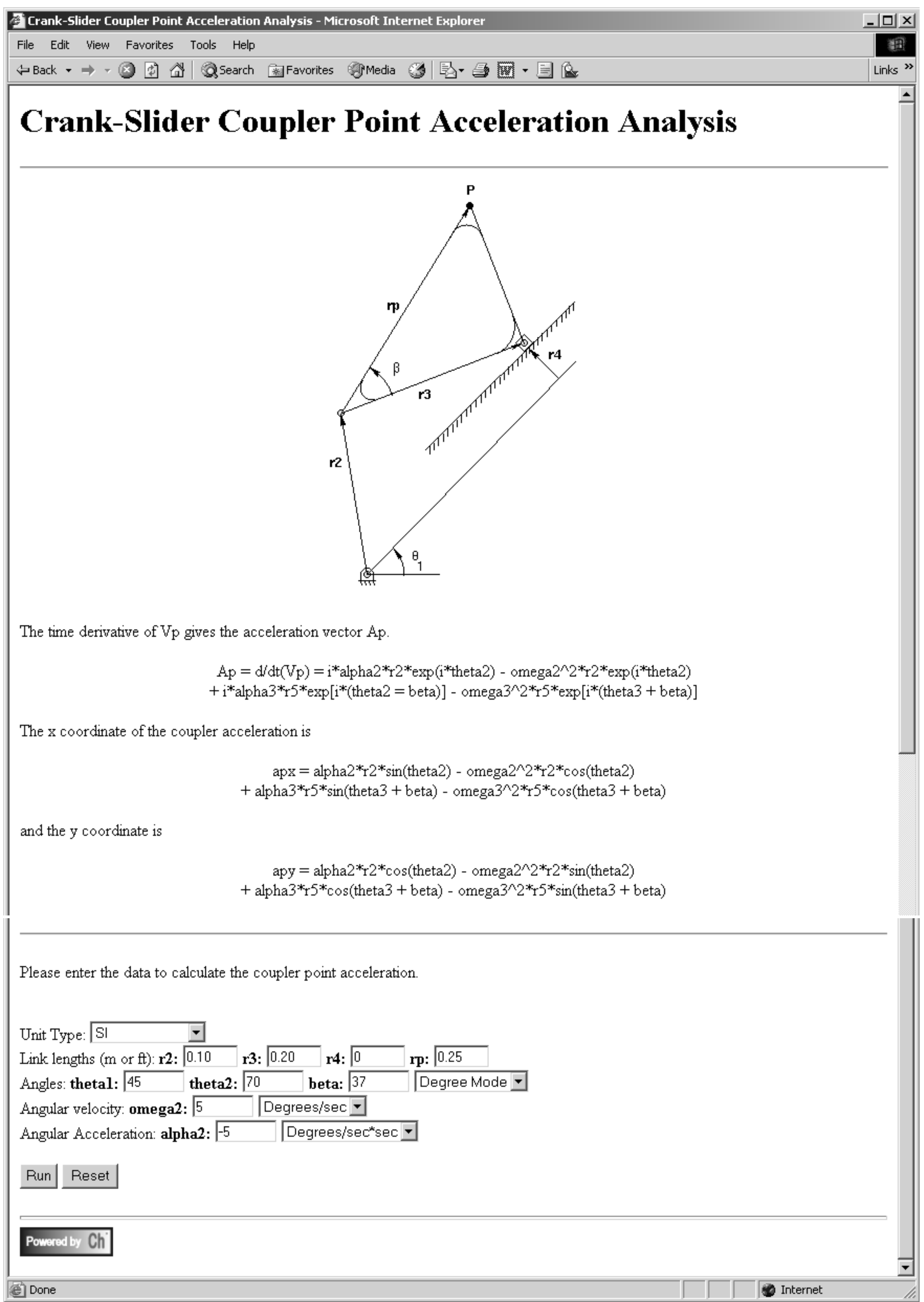


Figure 3.22: Coupler point acceleration analysis.

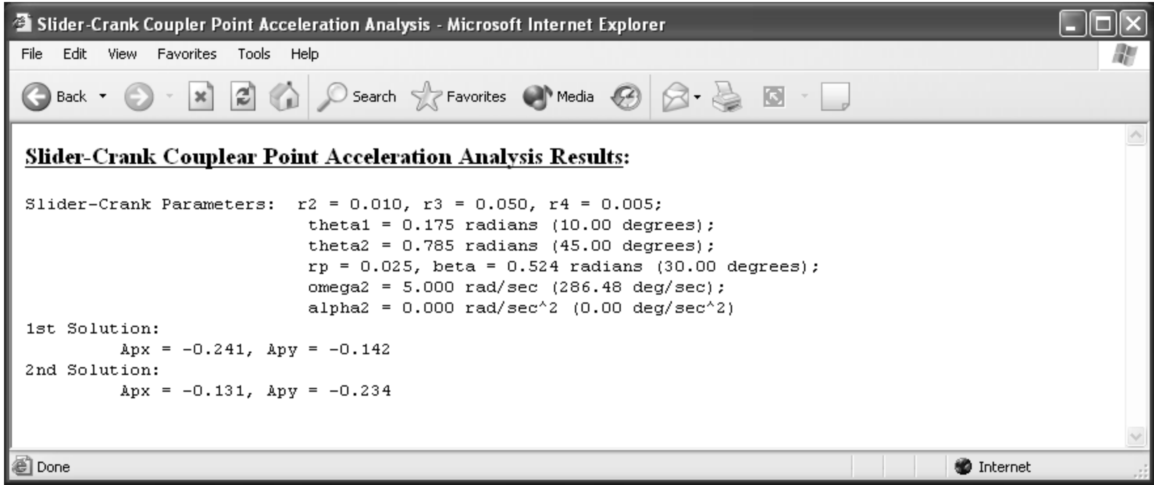


Figure 3.23: Output of coupler point acceleration analysis.

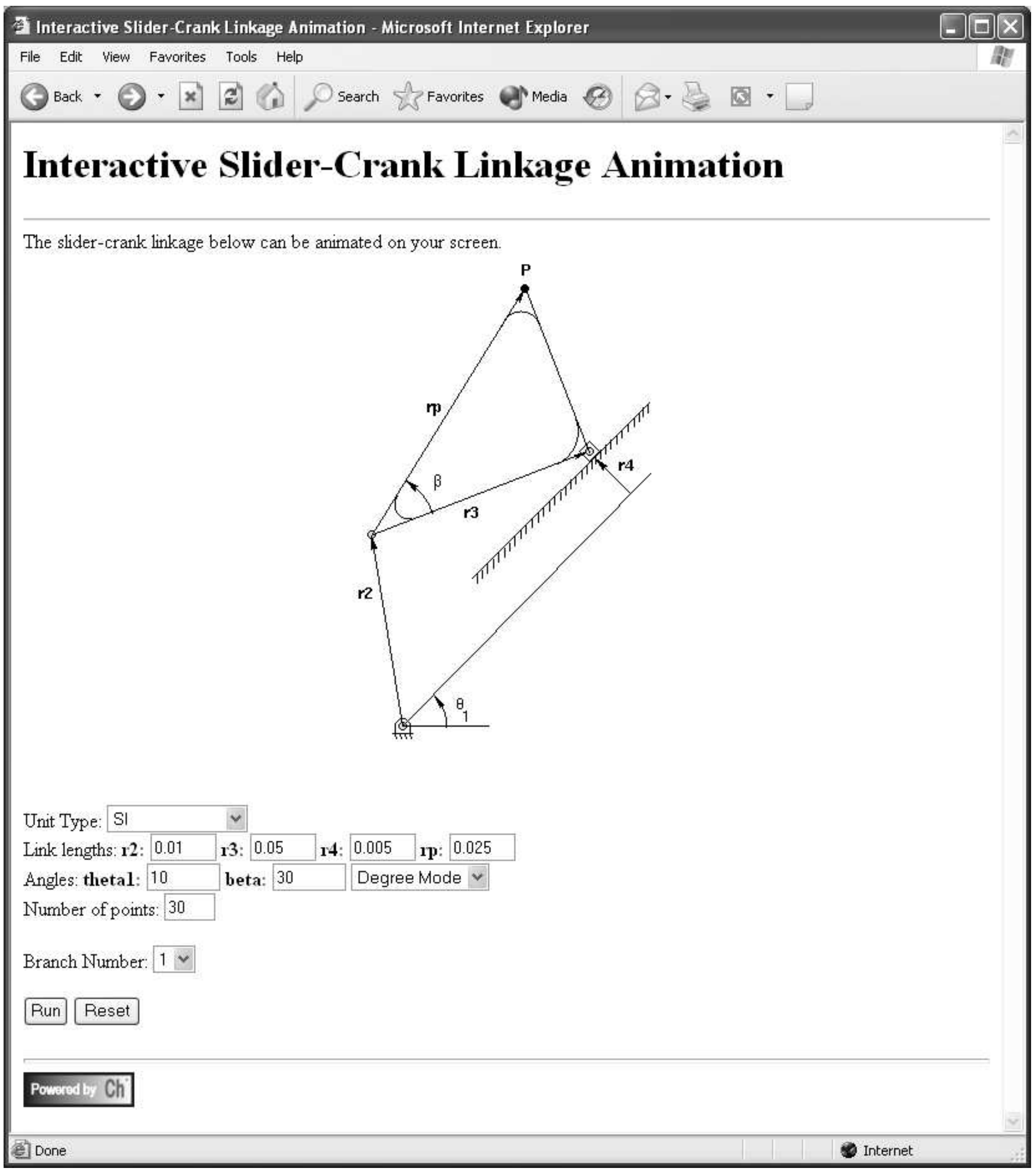


Figure 3.24: Slider-crank animation web page.

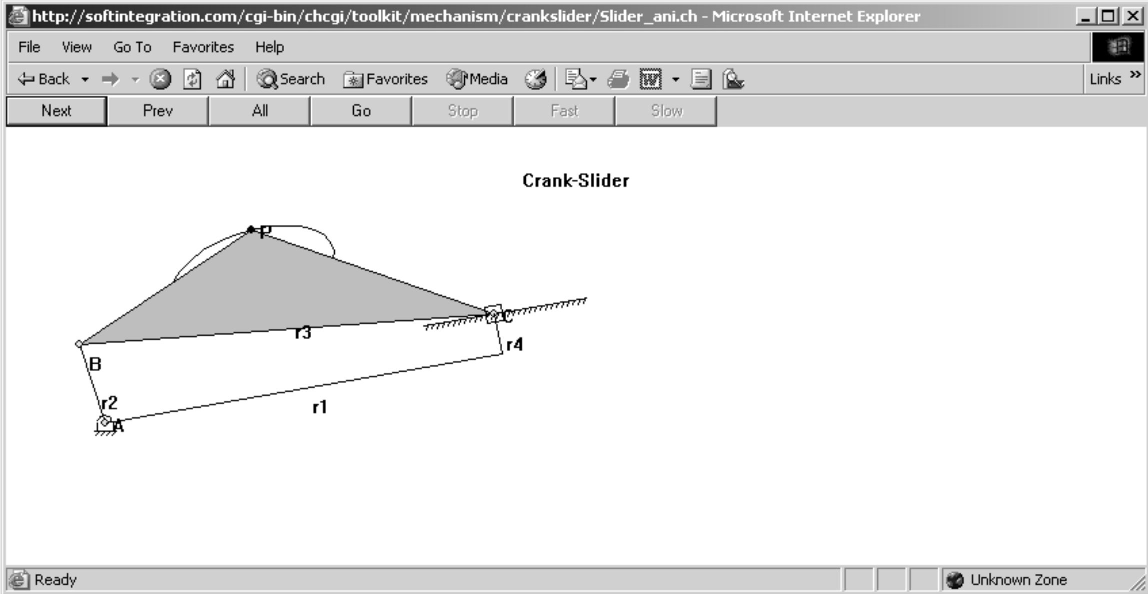


Figure 3.25: Slider-crank animation output.

Chapter 4

Geared Five-Bar Linkage

A geared fivebar mechanism is defined as a fivebar linkage attached to a pair of gears similar to the one shown in fig 4.1. There exists two different geometric inversions for the linkage. The origin of the Cartesian axes are located at point A_0 for both inversions. Similar to the fourbar and slidercrank mechanisms, analysis of the geared fivebar will produce a set of equations to determine the kinematic properties of each link as well as the coupler point.

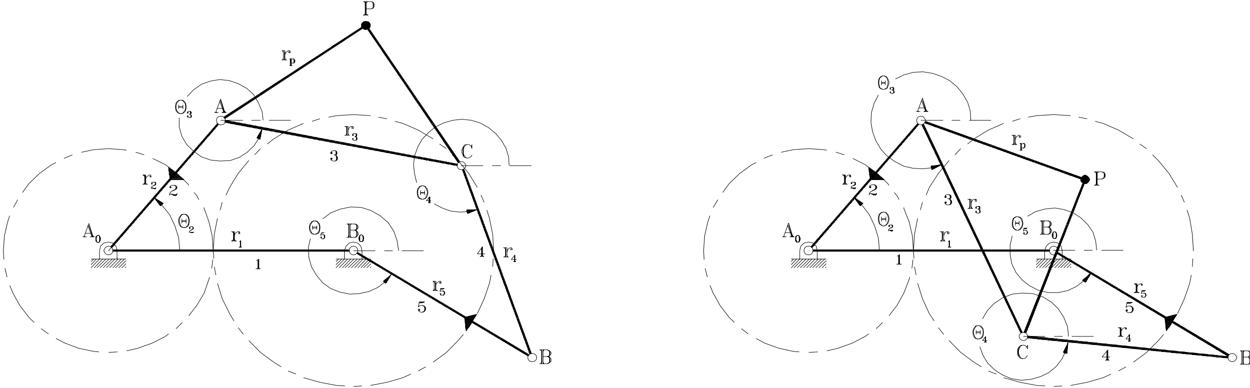


Figure 4.1: Both Inversions of the Geared Fivebar Linkage

4.1 Position Analysis

Angular position analysis of the geared fivebar linkage begin by first writing its vector equation (4.1) and then converting this equation to its polar form (4.2). The value for θ_5 can be written as a function of θ_2 to eliminate a variable in the vector equation. Since the gears rotate in opposite directions, the angular velocity and acceleration of link 5 are the negative values for link 2. Using the $\lambda\theta_2 + \phi$ relation will account for the negative value. The unknown terms of equation (4.2) are isolated to one side of the equation to produce equation (4.3).

$$\mathbf{r}_1 + \mathbf{r}_4 + \mathbf{r}_5 = \mathbf{r}_2 + \mathbf{r}_3 \tag{4.1}$$

$$r_1 e^{i\theta_1} + r_4 e^{i\theta_4} + r_5 e^{i(\lambda\theta_2+\phi)} = r_2 e^{i\theta_2} + r_3 e^{i\theta_3} \tag{4.2}$$

$$r_3 e^{i\theta_3} - r_4 e^{i\theta_4} = r_1 e^{i\theta_1} - r_2 e^{i\theta_2} + r_5 e^{i(\lambda\theta_2+\phi)} \tag{4.3}$$

With all the unknown quantities on the left-hand side, the polar equation (4.3) can now be solved using the `complexsolve()` function. Two sets of values for θ_3 and θ_4 are found by `complexsolve()`

where the inputs are $n1 = 2$, $n2 = 4$, $theta\ or\ r1 = r_3$, $theta\ or\ r2 = -r_4$, and $c3 = z = r_1e^{i\theta_1} - r_2e^{i\theta_2} + r_5e^{i(\lambda\theta_2+\phi)}$. Furthermore, the angular positions can be determined by the member function `angularPos()` of the geared fivebar class, `CGearedFivebar`. As an example, the following problem can be solved by using class `CGearedFivebar`.

> **Problem 1:** A geared fivebar linkage has the following dimensions and relations: $r_1 = 7cm, r_2 = 5cm, r_3 = 7cm, r_4 = 6cm, r_5 = 6cm$, $\theta_1 = 0$, $\theta_2 = 50^o$, $\lambda = -4/3$, $\phi = 35^o$. Determine the angular positions θ_3 and θ_4.

The solution to this problem is written in Ch code and is referred to as Program 20. The answers are as follow:

```
1st Circuit:
theta3 = -0.191 radians (-10.96 degrees)
theta4 = -1.227 radians (-70.32 degrees)
2nd Circuit:
theta3 = -1.140 radians (-65.30 degrees)
theta4 = -0.104 radians (-5.94 degrees)
```

4.2 Velocity Analysis

The angular velocities of the links are derived from the differential of equation (4.3). The derivative of this equation (4.4) is then multiplied by one of the unknown angle's exponential inverse, $e^{-i\theta_j}$, in order to isolate the two unknown $\omega's$ (4.5 and 4.6). The first one was multiplied by $e^{-i\theta_4}$ while the second one was multiplied by $e^{-i\theta_3}$. The i term factors out of these equations. The imaginary parts are separated to calculate the angular velocities (4.7 and 4.8). The angular velocities are then solved (4.9 and 4.10). The value of θ_5 is used instead of $\lambda\theta_2 + \phi$ because the value can be passed along with the $theta$ array. This reduces one less element that needs passing (ϕ). These calculations are carried out by member function `angularVel()`, whose prototype is as follow:

```
void CGearedFivebar::angularVel(double theta[1:5], omega[1:5]);
```

The first argument is an array of `double` type containing the angular positions of each link, and argument `omega` is an array storing the anguler velocity values. The calculated result of ω_3 and ω_4 is written into `omega[3]` and `omega[4]`, respectively.

$$ir_3\omega_3e^{i\theta_3} - ir_4\omega_4e^{i\theta_4} = -ir_2\omega_2e^{i\theta_2} + ir_5\omega_2\lambda e^{i\theta_5} \tag{4.4}$$

$$ir_3\omega_3e^{i(\theta_3-\theta_4)} - ir_4\omega_4 = -ir_2\omega_2e^{i(\theta_2-\theta_4)} + ir_5\omega_2\lambda e^{i(\theta_5-\theta_4)} \tag{4.5}$$

$$ir_3\omega_3 - ir_4\omega_4e^{i(\theta_4-\theta_3)} = -ir_2\omega_2e^{i(\theta_2-\theta_3)} + ir_5\omega_2\lambda e^{i(\theta_5-\theta_3)} \tag{4.6}$$

$$r_3\omega_3\sin(\theta_3-\theta_4) = -r_2\omega_2\sin(\theta_2-\theta_4) + r_5\omega_2\lambda\sin(\theta_5-\theta_4) \tag{4.7}$$

$$r_4\omega_4\sin(\theta_4-\theta_3) = r_2\omega_2\sin(\theta_2-\theta_4) + r_5\omega_2\lambda\sin(\theta_5-\theta_4) \tag{4.8}$$

$$\omega_3 = \frac{-r_2\omega_2\sin(\theta_2-\theta_4) + r_5\omega_2\lambda\sin(\theta_5-\theta_4)}{r_3\sin(\theta_3-\theta_4)} \tag{4.9}$$

$$\omega_4 = \frac{r_2\omega_2\sin(\theta_2-\theta_3) + r_5\omega_2\lambda\sin(\theta_5-\theta_3)}{r_4\sin(\theta_4-\theta_3)} \tag{4.10}$$

```
#include <fivebar.h>

int main()
{
   double r[1:5],
          theta_1[1:5], theta_2[1:5],
          phi, lambda;
   CGearedFivebar gearedbar;

   /* Setup geared fivebar linkage. */
   r[1] = 0.07;  r[2] = 0.05;  r[3] = 0.07;  r[4] = 0.06;  r[5] = 0.06;
   phi = 35*M_PI/180;  lambda = -4.0/3.0;
   theta_1[1] = 0;  theta_2[1] = 0;
   theta_1[2] = 50*(M_PI/180);  theta_2[2] = 50*(M_PI/180);

   /* Perform geared fivebar linkage analysis. */
   gearedbar.uscUnit(false);
   gearedbar.setLinks(r[1], r[2], r[3], r[4], r[5], theta_1[1]);
   gearedbar.setLambda(lambda);
   gearedbar.setPhi(phi);
   gearedbar.angularPos(theta_1, theta_2);

   /* May run into problem with theta4 being in opposite quadrant. */
   theta_1[4] += M_PI;  theta_2[4] += M_PI;
   if(theta_1[4] < -M_PI)
      theta_1[4] += 2*M_PI;
   if(theta_1[4] > M_PI)
      theta_1[4] -= 2*M_PI;
   if(theta_2[4] < -M_PI)
      theta_2[4] += 2*M_PI;
   if(theta_2[4] > M_PI)
      theta_2[4] -= 2*M_PI;

   /* Display the results. */
   printf("1st Circuit:\n");
   printf("\ttheta3 = %.3f radians (%.2f degrees)\n", theta_1[3], theta_1[3]*(180/M_PI));
   printf("\ttheta4 = %.3f radians (%.2f degrees)\n", theta_1[4], theta_1[4]*(180/M_PI));
   printf("2nd Circuit:\n");
   printf("\ttheta3 = %.3f radians (%.2f degrees)\n", theta_2[3], theta_2[3]*(180/M_PI));
   printf("\ttheta4 = %.3f radians (%.2f degrees)\n", theta_2[4], theta_2[4]*(180/M_PI));

  return 0;
}
```

Program 20: Program for computing θ_3 and θ_4 using class `CGearedFivebar`.

4.3 Acceleration Analysis

The angular accelerations are found by differentiating the angular velocity equation (4.4). This new equation (4.11) is multiplied by the same inverse angle values as the velocity equations had to isolate the two angular accelerations into two separate equations (4.12) and (4.13). The first one was multiplied by $e^{-i\theta_4}$ and the second by $e^{-i\theta_3}$. This time the real parts are separated to find the angular accelerations (4.14) and (4.15). The angular accelerations are then solved (4.16) and (4.17). Calculations are carried out by the member function, `angularAccel()`, whose prototype is shown below.

```
void CGearedFivebar::angularAccel(double theta[1:5], omega[1:5],
                                  alpha[1:5]);
```

Arguments `theta` and `omega` are arrays containing previously calculated or given values of the angular positions and velocities of the geared fivebar. `alpha` contains the angular acceleration values of the various linkages and outputs of member function `angularAccel()` are written to `alpha[3]` and `alpha[4]` for values, α_3 and α_4, respectively.

$$\begin{aligned} ir_3\alpha_3e^{i\theta_3} - r_3\omega_3^2e^{i\theta_3} - ir_4\alpha_4e^{i\theta_4} + r_4\omega_4^2e^{i\theta_4} &= r_2\omega_2^2e^{i\theta_2} - ir_2\alpha_2e^{i\theta_2} + ir_5\alpha_2\lambda e^{i\theta_5} \\ &\quad - r_5\omega_2^2\lambda^2e^{i\theta_5} \end{aligned} \tag{4.11}$$

$$\begin{aligned} ir_3\alpha_3e^{i(\theta_3-\theta_4)} - r_3\omega_3^2e^{i(\theta_3-\theta_4)} - ir_4\alpha_4 + r_4\omega_4^2 &= -ir_2\alpha_2e^{i(\theta_2-\theta_4)} + r_2\omega_2^2e^{i(\theta_2-\theta_4)} \\ &\quad + ir_5\alpha_2\lambda e^{i(\theta_5-\theta_4)} - r_5\omega_2^2\lambda^2e^{i(\theta_5-\theta_4)} \end{aligned} \tag{4.12}$$

$$\begin{aligned} ir_3\alpha_3 - r_3\omega_3^2 - ir_4\alpha_4e^{i(\theta_4-\theta_3)} + r_4\omega_4^2e^{i(\theta_4-\theta_3)} &= -ir_2\alpha_2e^{i(\theta_2-\theta_3)} + r_2\omega_2^2e^{i(\theta_2-\theta_3)} \\ &\quad + ir_5\alpha_2\lambda e^{i(\theta_5-\theta_3)} - r_5\omega_2^2\lambda^2e^{i(\theta_5-\theta_3)} \end{aligned} \tag{4.13}$$

$$\begin{aligned} -r_3\alpha_3\sin(\theta_3-\theta_4) - r_3\omega_3^2\cos(\theta_3-\theta_4) + r_4\omega_4^2 &= r_2\alpha_2\sin(\theta_2-\theta_4) + r_2\omega_2^2\cos(\theta_2-\theta_4) \\ &\quad - r_5\alpha_2\lambda\sin(\theta_5-\theta_4) \\ &\quad - r_5\omega_2^2\lambda\cos(\theta_5-\theta_4) \end{aligned} \tag{4.14}$$

$$\begin{aligned} -r_3\omega_3^2 + r_4\alpha_4\sin(\theta_4-\theta_3) + r_4\omega_4^2\cos(\theta_4-\theta_3) &= r_2\alpha_2\sin(\theta_2-\theta_3) + r_2\omega_2^2\cos(\theta_2-\theta_3) \\ &\quad - r_5\alpha_2\lambda\sin(\theta_5-\theta_3) \\ &\quad - r_5\omega_2^2\lambda\cos(\theta_5-\theta_3) \end{aligned} \tag{4.15}$$

$$\begin{aligned} \alpha_3 &= \frac{-r_2\alpha_2\sin(\theta_2-\theta_4) - r_2\omega_2^2\cos(\theta_2-\theta_4) - r_3\omega_3^2\cos(\theta_3-\theta_4)}{r_3\sin(\theta_3-\theta_4)} \\ &\quad + \frac{r_4\omega_4^2 + r_5\alpha_2\lambda\sin(\theta_5-\theta_4) + r_5\omega_2^2\lambda^2\cos(\theta_5-\theta_4)}{r_3\sin(\theta_3-\theta_4)} \end{aligned} \tag{4.16}$$

$$\begin{aligned} \alpha_4 &= \frac{r_2\alpha_2\sin(\theta_2-\theta_3) + r_2\omega_2^2\cos(\theta_2-\theta_3) + r_3\omega_3^2}{r_4\sin(\theta_4-\theta_3)} \\ &\quad + \frac{-r_4\omega_4^2\cos(\theta_4-\theta_3) - r_5\alpha_2\lambda\sin(\theta_5-\theta_3) - r_5\omega_2^2\lambda^2\cos(\theta_5-\theta_3)}{r_4\sin(\theta_4-\theta_3)} \end{aligned} \tag{4.17}$$

4.4 Coupler Point Analysis

The vector equation of the coupler point can be written in its polar form, and direct substitution of the angles that were solved in the earlier calculations will give the position of the coupler position. Likewise, the derivatives of this equation will give the coupler velocity and acceleration at the prescribed instant and will be solved for by substituting angular velocities and angular accelerations previously calculated. The calculations can be performed by member functions `couplerPointPos()`, `couplerPointVel()`, and `couplerPointAccel()`.

$$\mathbf{P} = \mathbf{r}_2 + \mathbf{r}_p \tag{4.18}$$

$$\mathbf{P} = r_2^{i\theta_2} + r_p e^{i(\theta_3+\beta)} \tag{4.19}$$

$$\dot{\mathbf{P}} = ir_2\omega_2 e^{i\theta_2} + ir_p\omega_3 e^{i(\theta_3+\beta)} \tag{4.20}$$

$$\ddot{\mathbf{P}} = ir_2\alpha_2 e^{i\theta_2} - r_2\omega_2^2 e^{i\theta_2} + ir_p\alpha_3 e^{i(\theta_3+\beta)} - r_p\omega_3^2 e^{i(\theta_3+\beta)} \tag{4.21}$$

Consider the following problem:

> **Problem 2:** Given a geared fivebar with parameters: $r_1 = 7cm, r_2 = 5cm, r_3 = 7cm, r_4 = 6cm, r_5 = 6cm, r_p = 5cm$, $\theta_1 = 0$, $\theta_2 = 50^o$, $\omega_2 = 5rad/sec$, $\alpha_2 = 0$, $\beta = 45^o$, $\lambda = -4/3$, and $\phi = 35^o$, find the coupler point position, velocity, and acceleration.

The above problem may be solved by Program 21 and the results are listed below.

```
Circuit 1:
   P = complex(0.082,0.046)
   Vp = complex(-0.189,0.147)
   Ap = complex(-0.807,-0.960)
Circuit 2:
   P = complex(0.067,0.003)
   Vp = complex(-0.191,0.161)
   Ap = complex(-0.805,-0.959)
```

4.5 Animation

Simulating the motion of the geared fivebar linkage can be performed with member function `animation()` of class `CGearedFiverbar`. It is prototyped as follows,

```
void CGearedFiverbar::animation(int branchnum, ...);
```

where `branchnum` corresponds to the branch number of the fivebar to animate. Similar to the other `animation()` member functions of classes `CFourbar` and `CSliderCrank`, this function also allows the user to save data generated for the animation into a file. For example, Program 22 can be used to animation a geared fivebar linkage with parameters: $r_1 = 7m$, $r_2 = 5m$, $r_3 = 10m$, $r_4 = 10m$, $r_5 = 2m$, $\theta_1 = 10^o$, $\phi = 35^o$, $\lambda = -2.5$, $r_p = 5m$, and $\beta = 45^o$. Figure 4.2 contain snapshots, or instances, of the animation for the two possible geometric inversions of the linkage.

```
#include <fivebar.h>

int main() {
    CGearedFivebar gearedfivebar;
    double r1 = 0.07, r2 = 0.05, r3 = 0.07, r4 = 0.06, r5 = 0.06,
           theta1 = 0;
    double rp = 0.05, beta = 20*M_PI/180;
    double phi = 35*M_PI/180;
    double lambda = -4.0/3.0;
    double theta_1[1:5], theta_2[1:5];
    double omega_1[1:5], omega_2[1:5];
    double alpha_1[1:5], alpha_2[1:5];
    double theta2 = 50*M_PI/180;
    double complex P[1:2], Vp[1:2], Ap[1:2];

    omega_1[2]=5;   /*  rad/sec  */
    alpha_1[2]=0;   /*  rad/sec*sec  */
    omega_2[2]=5;   /*  rad/sec  */
    alpha_2[2]=0;   /*  rad/sec*sec  */

    theta_1[1] = theta1;
    theta_1[2] = theta2; // theta2
    theta_2[1] = theta1;
    theta_2[2] = theta2; // theta2

    gearedfivebar.uscUnit(false);
    gearedfivebar.setLinks(r1, r2, r3, r4, r5, theta1);
    gearedfivebar.setCouplerPoint(COUPLER_LINK3, rp, beta);
    gearedfivebar.setLambda(lambda);
    gearedfivebar.setPhi(phi);

    gearedfivebar.angularPos(theta_1, theta_2);
    gearedfivebar.couplerPointPos(COUPLER_LINK3, theta2, P[1], P[2]);
    gearedfivebar.angularVel(theta_1, omega_1);
    gearedfivebar.angularVel(theta_2, omega_2);
    Vp[1] = gearedfivebar.couplerPointVel(COUPLER_LINK3, theta_1, omega_1);
    Vp[2] = gearedfivebar.couplerPointVel(COUPLER_LINK3, theta_2, omega_2);
    gearedfivebar.angularAccel(theta_1, omega_1, alpha_1);
    gearedfivebar.angularAccel(theta_2, omega_2, alpha_2);
    Ap[1] = gearedfivebar.couplerPointAccel(COUPLER_LINK3, theta_1, omega_1,
                                            alpha_1);
    Ap[2] = gearedfivebar.couplerPointAccel(COUPLER_LINK3, theta_2, omega_2,
                                            alpha_2);

    printf("Circuit 1:\n");
    printf("   P = %.3f\n", P[1]);
    printf("   Vp = %.3f\n", Vp[1]);
    printf("   Ap = %.3f\n", Ap[1]);
    printf("Circuit 2:\n");
    printf("   P = %.3f\n", P[2]);
    printf("   Vp = %.3f\n", Vp[2]);
    printf("   Ap = %.3f\n", Ap[2]);

    return 0;
}
```

Program 21: Program for computing the position, velocity, and acceleration of the coupler point of a geared fivebar linkage.

```
#include <fivebar.h>

int main()
{
   double r[1:5],
          theta_1[1:5], theta_2[1:5],
          rp, beta, phi, lambda;
   CGearedFivebar gearedbar;

   /* Setup geared fivebar linkage. */
   r[1] = 7;  r[2] = 5;  r[3] = 10;  r[4] = 10;  r[5] = 2;
   rp = 5;  beta = 45*(M_PI/180);
   phi = 35*M_PI/180;  lambda = -2.5;
   theta_1[1] = 10*(M_PI/180);  theta_2[1] = 10*(M_PI/180);
   theta_1[2] = 50*(M_PI/180);  theta_2[2] = 50*(M_PI/180);

   /* Perform geared fivebar linkage analysis. */
   gearedbar.setLinks(r[1], r[2], r[3], r[4], r[5], theta_1[1]);
   gearedbar.setCouplerPoint(COUPLER_LINK3, rp, beta);
   gearedbar.setLambda(lambda);
   gearedbar.setPhi(phi);
   gearedbar.setNumPoints(50);
   gearedbar.animation(1);
   gearedbar.animation(2);

  return 0;
}
```

Program 22: Program for animating the geared fivebar linkage.

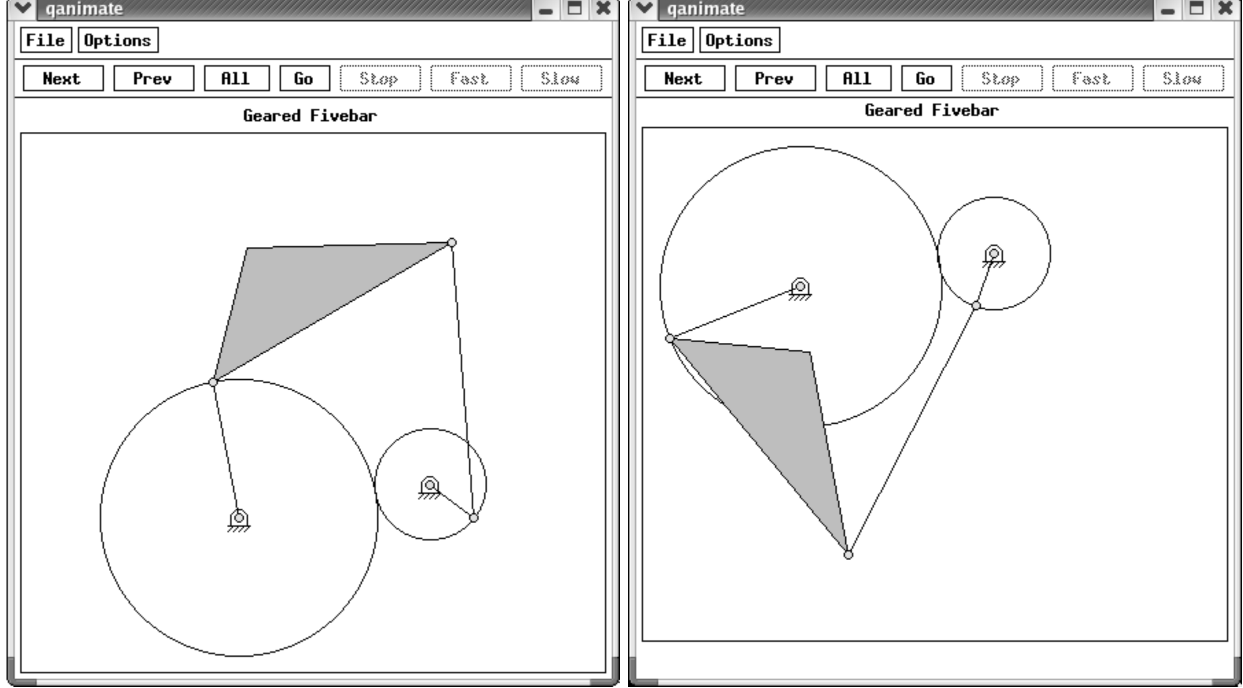


Figure 4.2: The animation from Program 22.

4.6 Web-Based Geared Fivebar Linkage Analysis

As with the fourbar and slider-crank mechanisms, there are also interactive web pages for analysis of the geared fivebar linkage. Figure 4.3 shows the main web page for fivebar linkage analysis. It contains two links: one for performing both the angular and coupler point analysis, as discussed in previous sections, and one for simulating the motion of a geared fivebar mechanism.

Figure 4.4 is the web page for calculating the angular and coupler point positions, velocities, and accelerations. Input values can be entered for the link lengths, coupler point parameters, base and input angles, θ_1 and θ_2, as well as the angular velocity and acceleration of input link 2. Furthermore, the angle parameters, λ and ϕ, may also be specified. Using the parameters for the geared fivebar linkage defined in Problem 2, Figure 4.5 shows the output of performing the web-based analysis for the angular and coupler point properties.

For animating the geared fivebar mechanism, the web page shown in Figure 4.6 can be used. Using the same parameters as in Section 4.5, a *snapshot*, or instant, of the fivebar linkage animation is shown in Figure 4.7.

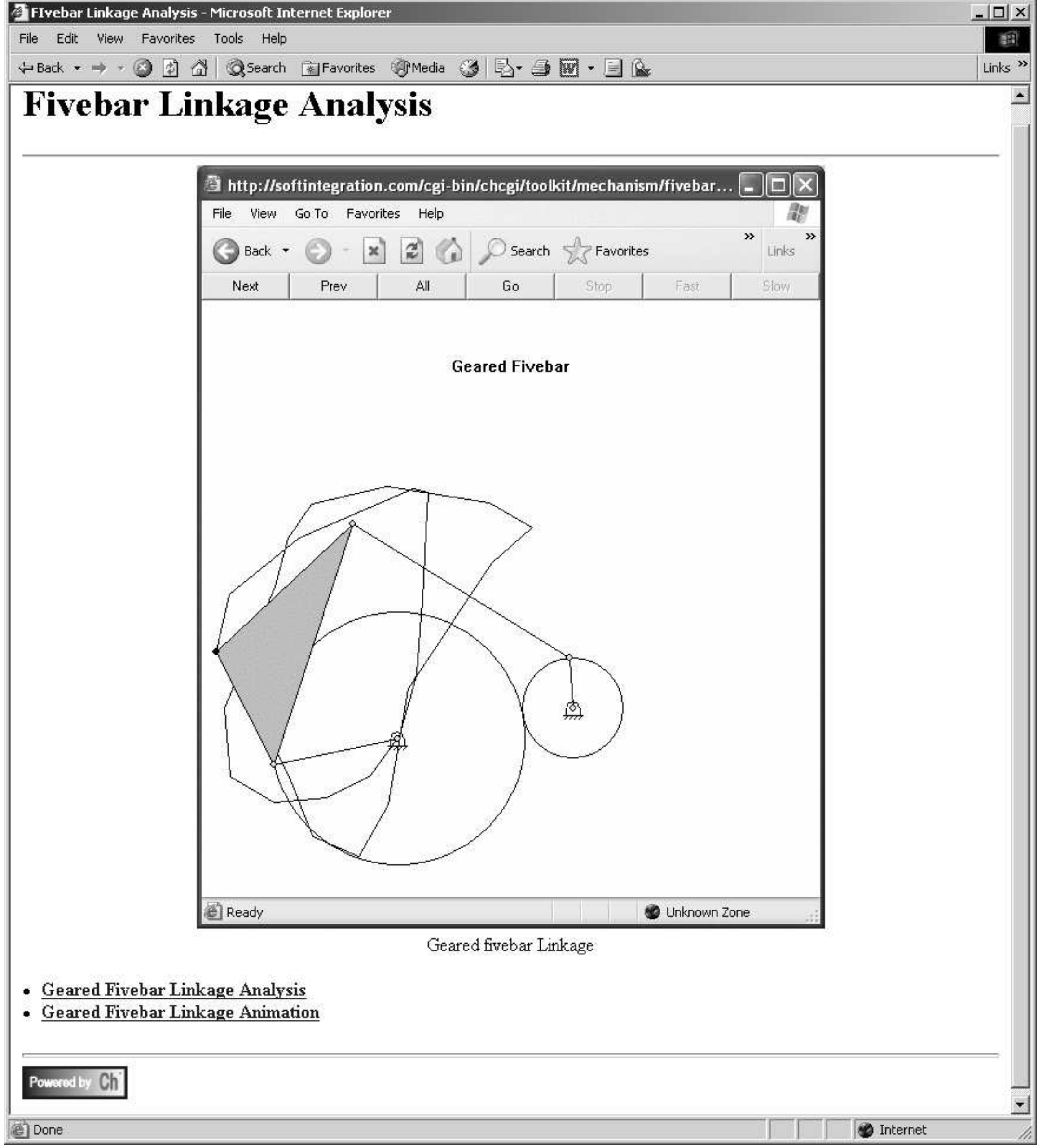


Figure 4.3: Main page for web-based geared fivebar linkage analysis.

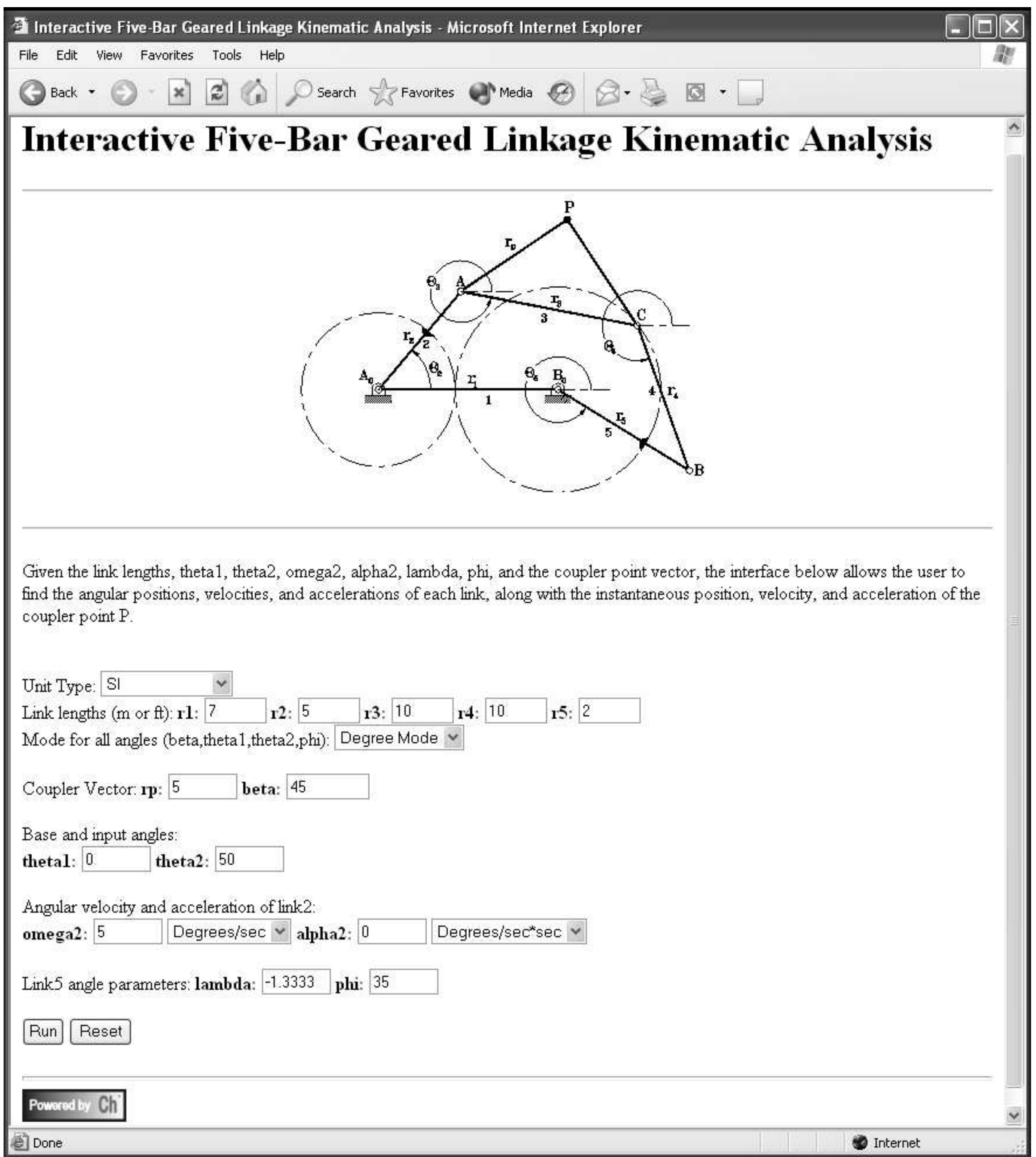


Figure 4.4: Web-based geared fivebar linkage analysis.

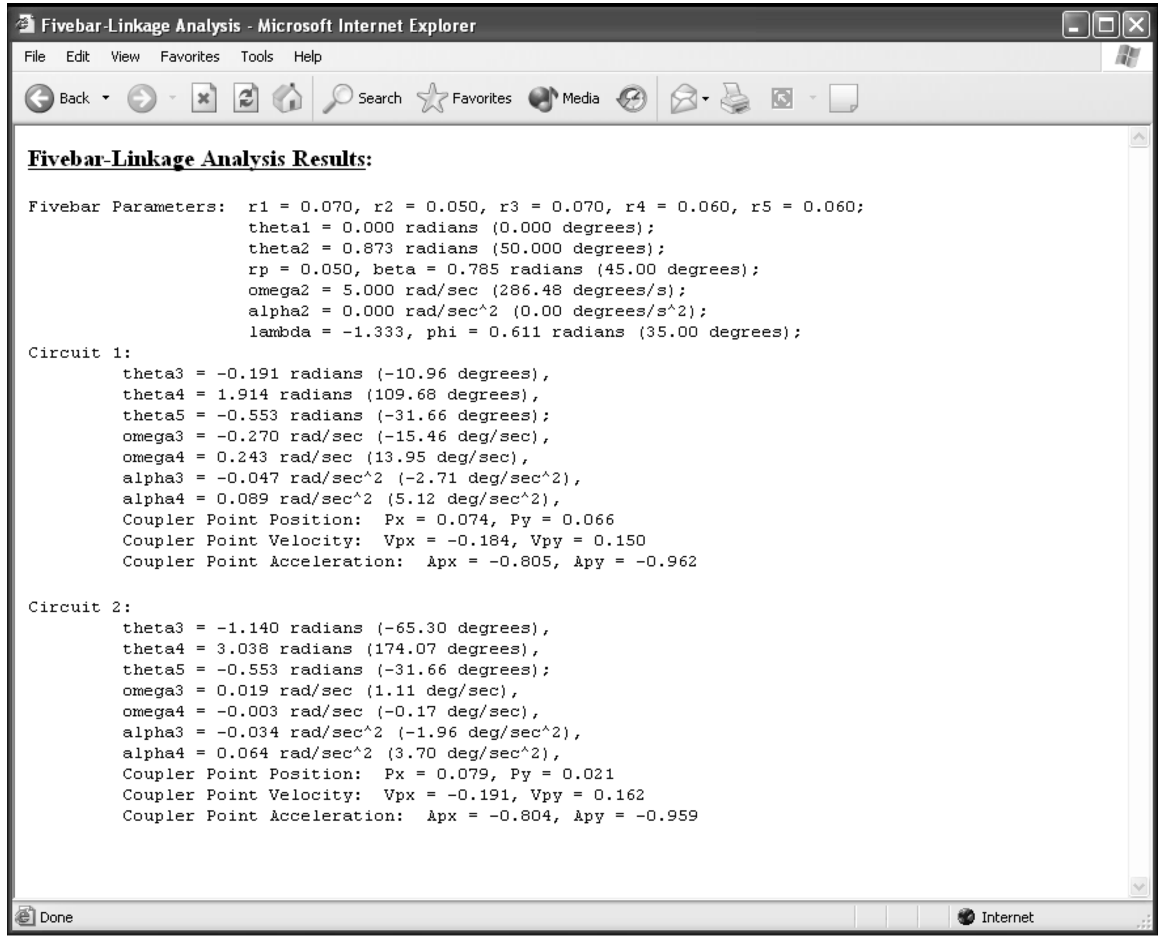


Figure 4.5: Output of web-based analysis.

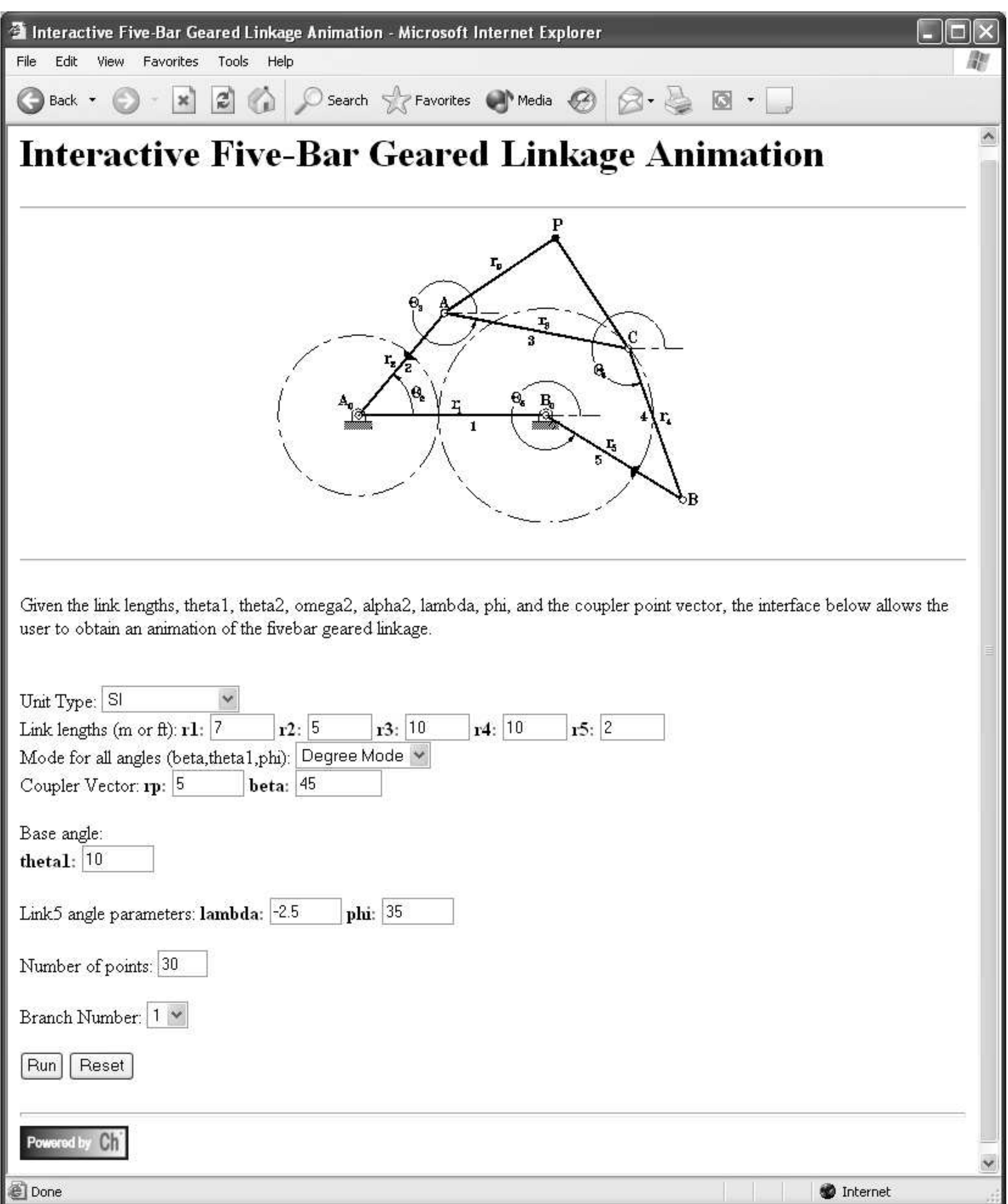


Figure 4.6: Web-based geared fivebar linkage animation.

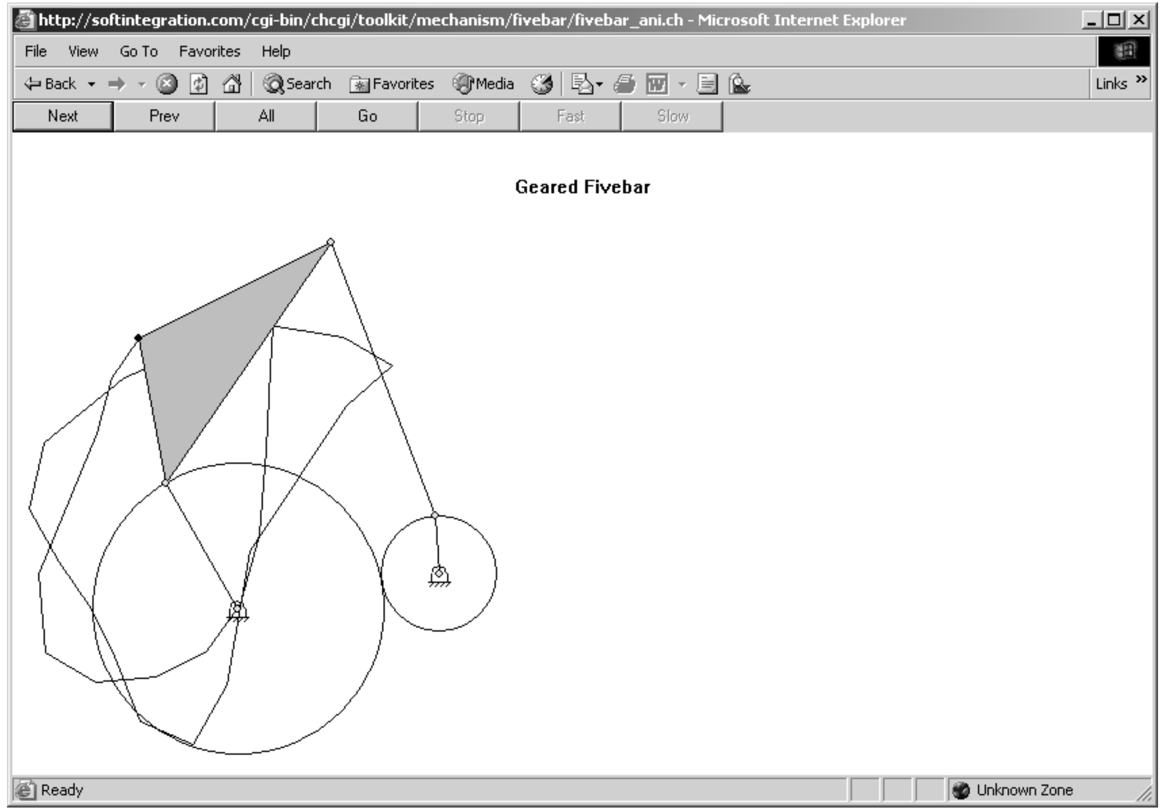


Figure 4.7: Output of web-based animation.

Chapter 5

Multi-Loop Six-Bar Linkages

5.1 Fourbar/Slider-Crank Linkage

A fourbar-slider linkage is assembled from a fourbar linkage and a slider-crank mechanism, where the output link of the fourbar linkage is also the input link of the slider linkage. For the fourbar-slider shown in Figure 5.1, a coupler is attached to the output link of the slider. Although the analysis of this mechanism is more complex than analyzing a simple fourbar linkage, the concepts applied to both are essentially the same.

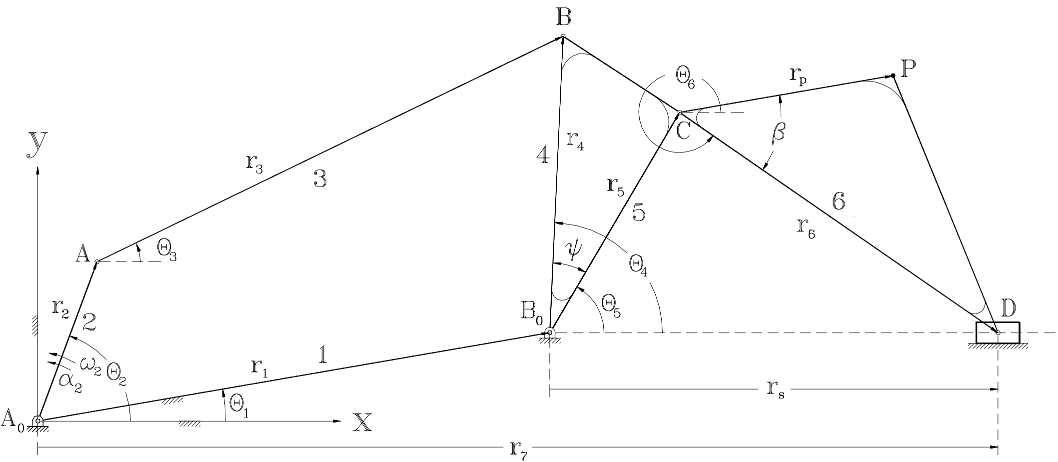


Figure 5.1: Barslider Linkage

5.1.1 Position Analysis

As with any linkage mechanism, position analysis of the fourbar-slider is performed by applying geometry, complex arithmetic, and vector analysis. Utilizing these three mathematical concepts allows the angular positions θ_3, θ_4, θ_5, and θ_6 as well as the slider position, r_7, to be determined. First, the vector equation for the fourbar section is written and reduced.

$$\mathbf{r}_1 + \mathbf{r}_4 = \mathbf{r}_2 + \mathbf{r}_3 \tag{5.1}$$

$$r_1 e^{i\theta_1} + r_4 e^{i\theta_4} = r_2 e^{i\theta_2} + r_3 e^{i\theta_3} \tag{5.2}$$

$$r_3 e^{i\theta_3} - r_4 e^{i\theta_4} = r_1 e^{i\theta_1} - r_2 e^{i\theta_2} \tag{5.3}$$

The latest equation (5.3) has all the unknown quantities on the left-hand side, which can now be solved using the `complexsolve` function. Two sets of values for θ_3 and θ_4 are found by `complexsolve()` where the inputs are $n1 = 2$, $n2 = 4$, $theta\ or\ r1 = r_3$, $theta\ or\ r2 = -r_4$, and $c3 = z = r_1 e^{i\theta_1} - r_2 e^{i\theta_2}$. Using each set of $\theta's$ separately, the angle θ_5 can be found by subtracting the angle ψ from θ_4. A second set of vector equations is derived from the slider section of the linkage.

$$\theta_5 = \theta_4 - \psi \tag{5.4}$$

$$\mathbf{r}_5 + \mathbf{r}_6 = \mathbf{r}_s \tag{5.5}$$

$$r_5 e^{i\theta_5} + r_6 e^{i\theta_6} = r_s e^0 \tag{5.6}$$

$$r_s - r_6 e^{i\theta_6} = r_5 e^{i\theta_5} \tag{5.7}$$

Again the latest equation (5.7) has the unknown terms on the left-hand side. Two sets of two values of r_s and θ_6 can then be determined. The value of r_7 is found by adding the horizontal component of r_1 to r_s. The angular positions and slider position can also be calculated by the fourbar-slider class, `CFourbarSlider`, member functions `angularPos()` and `sliderPos()`, respectively.

$$\mathbf{r}_7 = \mathbf{r}_{1x} + \mathbf{r}_s \tag{5.8}$$

$$r_7 = r_1 \cos\theta_1 + r_s \tag{5.9}$$

As an example, the following problem can be solved with the Ch code labeled as Program 23.

> **Problem 1:** The parameters of a fourbar-slider mechanism are $r_1 = 12cm, r_2 = 4cm, r_3 = 12cm, r_4 = 7cm, r_5 = 6cm, r_6 = 9cm$, $\theta_1 = 10^o$, $\theta_2 = 70^o$, and $\psi = 30^o$, determine the angular positions of the rest of the links. Also determine the horizontal distance between the origin, A_0, and the slider labeled as r_7 in Figure 5.1.

The solutions are as follow:

```
Circuit 1:
   theta3 = 0.459
   theta4 = 1.527
   theta5 = 1.003
   theta6 = -0.597
   r7 = 2.249
Circuit 2:
   theta3 = 0.459
   theta4 = 1.527
   theta5 = 1.003
   theta6 = -2.545
   r7 = 0.760
Circuit 3:
   theta3 = -0.777
   theta4 = -1.845
   theta5 = -2.368
   theta6 = 0.484
   r7 = 1.549
Circuit 4:
   theta3 = -0.777
   theta4 = -1.845
   theta5 = -2.368
   theta6 = 2.657
   r7 = -0.044
```

```
#include<stdio.h>
#include<sixbar.h>

int main()
{
   double r[1:6], theta[1:4][1:6];
   double r7[1:4];
   double psi;
   int i;
   CFourbarSlider fslider;

   /* Default specification of the fourbar-slider linkage. */
   r[1] = 0.12;  r[2] = 0.04;  r[3] = 0.12;
   r[4] = 0.07;  r[5] = 0.06;  r[6] = 0.09;
   for(i = 1; i <= 4; i++)
   {
      theta[i][1] = M_DEG2RAD(10.0);
      theta[i][2] = M_DEG2RAD(70.0);
   }
   psi = M_DEG2RAD(30.0);

   /* Perform fourbar-slider linkage analysis. */
   fslider.uscUnit(false);
   fslider.setLinks(r, theta[1][1], psi);
   fslider.angularPos(theta[1], theta[2], theta[3], theta[4]);
   r7[1] = fslider.sliderPos(theta[1]);
   r7[2] = fslider.sliderPos(theta[2]);
   r7[3] = fslider.sliderPos(theta[3]);
   r7[4] = fslider.sliderPos(theta[4]);

   /* Display the results. */
   printf("Circuit 1:\n");
   printf("   theta3 = %.3f\n", theta[1][3]);
   printf("   theta4 = %.3f\n", theta[1][4]);
   printf("   theta5 = %.3f\n", theta[1][5]);
   printf("   theta6 = %.3f\n", theta[1][6]);
   printf("   r7 = %.3f\n", r7[1]);
   printf("Circuit 2:\n");
   printf("   theta3 = %.3f\n", theta[2][3]);
   printf("   theta4 = %.3f\n", theta[2][4]);
   printf("   theta5 = %.3f\n", theta[2][5]);
   printf("   theta6 = %.3f\n", theta[2][6]);
   printf("   r7 = %.3f\n", r7[2]);
   printf("Circuit 3:\n");
   printf("   theta3 = %.3f\n", theta[3][3]);
   printf("   theta4 = %.3f\n", theta[3][4]);
   printf("   theta5 = %.3f\n", theta[3][5]);
   printf("   theta6 = %.3f\n", theta[3][6]);
   printf("   r7 = %.3f\n", r7[3]);
   printf("Circuit 4:\n");
   printf("   theta3 = %.3f\n", theta[4][3]);
   printf("   theta4 = %.3f\n", theta[4][4]);
   printf("   theta5 = %.3f\n", theta[4][5]);
   printf("   theta6 = %.3f\n", theta[4][6]);
   printf("   r7 = %.3f\n", r7[4]);

   return 0;
}
```

Program 23: Program for computing θ_3, θ_4, θ_5, θ_6, and r_7 using class `CFourbarSlider`.

5.1.2 Velocity Analysis

The first step in performing the velocity analysis of the fourbar-slider is to take the derivatives of the position vector equations of the two linkages. Taking the derivative of equations (5.3) and (5.7) gives the vector velocity equations (5.10) and (5.11).

$$ir_3\omega_3 e^{i\theta_3} - ir_4\omega_4 e^{i\theta_4} = -ir_2\omega_2 e^{i\theta_2} \tag{5.10}$$

$$\dot{r}_s - ir_6\omega_6 e^{i\theta_6} = ir_5\omega_5 e^{i\theta_5} \tag{5.11}$$

To find the values for ω_3, ω_4, ω_6, and $\dot{r}_7$, the velocity equations (5.10) and (5.11) must be multiplied by an $e^{-i\theta_j}$ term in order to isolate a particular ω_j term. Note that $\omega_5 = \omega_4$ and $\dot{r}_s = \dot{r}_7$. Multiplying equation (5.10) by $e^{-i\theta_4}$ and $e^{-i\theta_3}$ will generate equations (5.12) and (5.13). After factoring out the i term, the imaginary parts of equations (5.12) and (5.13) are separated in order to calculate the angular velocities (5.14) and (5.15). Rearranging these last two equations will give the values of ω_3 and ω_4 (5.16) and (5.17). The terms ω_6 and $\dot{r}_7$ are found by isolating the imaginary and real parts of equation (5.11). The imaginary parts are given in equation 5.18 and solves for ω_6 in equation (5.20), and the real terms are given in equation (5.19) and solves for $\dot{r}_7$ in equation (5.21). Note that $\dot{r}_7$ is independent of ω_3.

$$ir_3\omega_3 e^{i(\theta_3-\theta_4)} - ir_4\omega_4 = -ir_2\omega_2 e^{i(\theta_2-\theta_4)} \tag{5.12}$$

$$ir_3\omega_3 - ir_4\omega_4 e^{i(\theta_4-\theta_3)} = -ir_2\omega_2 e^{i(\theta_2-\theta_3)} \tag{5.13}$$

$$r_3\omega_3 \sin(\theta_3-\theta_4) = -r_2\omega_2 \sin(\theta_2-\theta_4) \tag{5.14}$$

$$r_4\omega_4 \sin(\theta_4-\theta_3) = r_2\omega_2 \sin(\theta_2-\theta_3) \tag{5.15}$$

$$\omega_3 = -\frac{r_2\omega_2 \sin(\theta_4-\theta_2)}{r_3 \sin(\theta_4-\theta_3)} \tag{5.16}$$

$$\omega_4 = \frac{r_2\omega_2 \sin(\theta_3-\theta_2)}{r_4 \sin(\theta_3-\theta_4)} \tag{5.17}$$

$$r_6\omega_6 \cos\theta_6 = -r_5\omega_4 \cos\theta_5 \tag{5.18}$$

$$\dot{r}_7 + r_6\omega_6 \sin\theta_6 = -r_5\omega_4 \sin\theta_5 \tag{5.19}$$

$$\omega_6 = -\frac{r_5\omega_4 \cos\theta_5}{r_6 \cos(\theta_6)} \tag{5.20}$$

$$\dot{r}_7 = -\left(r_6\omega_6 \sin\theta_6 + r_5\omega_4 \sin\theta_5\right) \tag{5.21}$$

The values of the ω_3, ω_4, and ω_6 are solved by member functions `angularVel()`, while the value of $\dot{r}_7$ is solved by member function `sliderVel()`. Thus given the link lengths and angular positions, which can be calculated by `CFourbarSlider::angularPos()`, and initial angular velocity, ω_2, the angular velocity of the rest of the links and the velocity of the slider can easily be determined.

5.1.3 Acceleration Analysis

The first step of acceleration analysis is to take the derivative of equations (5.10) and (5.11). Similar to the velocity equations, these derivatives (5.22) and (5.23) are then multiplied by an $e^{-i\theta_j}$ term in order to isolate a particular α_j term. Note that $\alpha_5 = \alpha_4$ and $\ddot{r}_s = \ddot{r}_7$. Multiplying equation (5.22) by $e^{-i\theta_4}$ and $e^{-i\theta_3}$ will generate equations (5.24) and (5.25). The real parts of equations (5.24) and (5.25) are separated in order to calculate the angular accelerations (5.26) and (5.27). Rearranging these last two equations will give the values of α_3 and α_4 (5.28) and (5.29). The terms α_6 and $\ddot{r}_7$ are found by isolating the imaginary and real parts of equation (5.23). The imaginary parts are given in equation (5.30) and solves for α_6 in equation (5.32), and the real terms are given in equation (5.31) and solves for $\ddot{r}_7$ in equation (5.33). Note that $\ddot{r}_7$ is independent of $\alpha 3$, but is dependent on ω_3. All the angular acceleration and slider acceleration derivations have been written to member functions `angularAccel()` and `sliderAccel()`. So instead of personally calculating the numerous angular acceleration properties of the fourbar slider (α_3, α_4, α_5, α_6, $\ddot{r}_7$), class `CFourbarSlider`'s member functions can be utilized to quickly determine the desired values.

$$(i\alpha_3 - \omega_3^2)r_3e^{i\theta_3} - (i\alpha_4 - \omega_4^2)r_4e^{i\theta_4} = (\omega_2^2 - i\alpha_2)r_2e^{i\theta_2} \tag{5.22}$$

$$\ddot{r}_s - (i\alpha_6 - \omega_6^2)r_6e^{i\theta_6} = (i\alpha_5 - \omega_5^2)r_5e^{i\theta_5} \tag{5.23}$$

$$ir_4\alpha_4 - r_4\omega_4^2 - ir_3\alpha_3e^{i(\theta_3-\theta_4)} + r_3\omega_3^2e^{i(\theta_3-\theta_4)} = ir_2\alpha_2e^{i(\theta_2-\theta_4)} - r_2\omega_2^2e^{i(\theta_2-\theta_4)} \tag{5.24}$$

$$ir_4\alpha_4e^{i(\theta_4-\theta_3)} - r_4\omega_4^2e^{i(\theta_4-\theta_3)} - ir_3\alpha_3 + r_3\omega_3^2 = ir_2\alpha_2e^{i(\theta_2-\theta_3)} - r_2\omega_2^2e^{i(\theta_2-\theta_3)} \tag{5.25}$$

$$r_3\alpha_3\sin(\theta_3-\theta_4) = -r_2\alpha_2\sin(\theta_2-\theta_4) - r_2\omega_2^2\cos(\theta_2-\theta_4) - r_3\omega_3^2\cos(\theta_3-\theta_4) + r_4\omega_4^2 \tag{5.26}$$

$$r_4\alpha_4\sin(\theta_4-\theta_3) = r_2\alpha_2\sin(\theta_2-\theta_3) + r_2\omega_2^2\cos(\theta_2-\theta_3) + r_3\omega_3^2 - r_4\omega_4^2\cos(\theta_4-\theta_3) \tag{5.27}$$

$$\alpha_3 = \frac{-r_2\omega_2^2\cos(\theta_2-\theta_4) - r_2\alpha_2\sin(\theta_2-\theta_4) - r_3\omega_3^2\cos(\theta_3-\theta_4) + r_4\omega_4^2}{r_3\sin(\theta_3-\theta_4)} \tag{5.28}$$

$$\alpha_4 = \frac{r_2\alpha_2\sin(\theta_2-\theta_3) + r_2\omega_2^2\cos(\theta_2-\theta_3) + r_3\omega_3^2 - r_4\omega_4^2\cos(\theta_4-\theta_3)}{r_4\sin(\theta_4-\theta_3)} \tag{5.29}$$

$$r_6\omega_6^2\sin\theta_6 - r_6\alpha_6\cos\theta_6 = r_5\alpha_4\cos\theta_5 - r_5\omega_4^2\sin\theta_5 \tag{5.30}$$

$$\ddot{r}_7 + r_6\omega_6^2\cos\theta_6 + r_6\alpha_6\sin\theta_6 = -r_5\alpha_5\sin\theta_5 - r_5\omega_4^2\cos\theta_5 \tag{5.31}$$

$$\alpha_6 = \frac{r_5\omega_5^2\sin\theta_5 - r_5\alpha_5\cos\theta_5 + r_6\omega_6^2\sin\theta_6}{r_6\cos\theta_6} \tag{5.32}$$

$$\ddot{r}_7 = -r_5\alpha_5\sin\theta_5 - r_5\omega_5^2\cos\theta_5 - r_6\alpha_6\sin\theta_6 - r_6\omega_6^2\cos\theta_6 \tag{5.33}$$

(c) **Coupler Position, Velocity, and Acceleration**: The vector equation of the coupler point can be written in its polar form, and direct substitution of the angles that were solved in the earlier calculations will give the position of the coupler position. Likewise, the derivatives of this equation will give the coupler velocity and acceleration at the prescribed instant and will be solved for by substituting angular velocities and angular accelerations previously calculated. The calculations are performed by member functions `couplerPointPos()`, `couplerPointVel()`, and `couplerPointAccel()`. Note again

that there are four possible coupler positions from two possible positions each of the fourbar linkage and the slidercrank linkage, and there are four possible angular velocities and accelerations for the system.

$$\mathbf{P} = \mathbf{r}_1 + \mathbf{r}_5 + \mathbf{r}_p \tag{5.34}$$

$$\mathbf{P} = r_1 e^{i\theta_1} + r_5 e^{i\theta_5} + r_p e^{i(\theta_6+\beta)} \tag{5.35}$$

$$\dot{\mathbf{P}} = i r_5 \omega_5 e^{i\theta_5} + i r_p \omega_6 e^{i(\theta_6+\beta)} \tag{5.36}$$

$$\ddot{\mathbf{P}} = i r_5 \alpha_5 e^{i\theta_5} - r_5 \omega_5^2 e^{i\theta_5} + i r_p \alpha_6 e^{i(\theta_6+\beta)} - r_p \omega_6^2 e^{i(\theta_6+\beta)} \tag{5.37}$$

Problem 2: A fourbar-slider has the following parameters: $r_1 = 12cm, r_2 = 4cm, r_3 = 12cm, r_4 = 7cm, r_5 = 6cm, r_6 = 9cm, r_p = 5cm$, $\theta_1 = 10^o$, $\theta_2 = 70^o$, $\omega_2 = 10rad/sec$, $\alpha_2 = 0$, $\beta = 45^o$, and $\psi = 30^o$. Calculate the coupler point position, velocity, and acceleration.

The solution to the above problem is Program 24, and the results are given below.

```
Circuit 1: Coupler Point Acceleration
   P = complex(0.200,0.081)
   Vp = complex(-0.210,0.049)
   Ap = complex(-2.741,-0.089)

Circuit 2: Coupler Point Acceleration
   P = complex(0.141,0.022)
   Vp = complex(-0.132,0.127)
   Ap = complex(-2.299,0.352)

Circuit 3: Coupler Point Acceleration
   P = complex(0.090,0.027)
   Vp = complex(-0.096,0.207)
   Ap = complex(2.736,-0.825)

Circuit 4: Coupler Point Acceleration
   P = complex(0.027,-0.036)
   Vp = complex(-0.201,0.102)
   Ap = complex(3.766,0.204)
```

```
#include<stdio.h>
#include<sixbar.h>


int main()
{
   double r[1:6], theta[1:4][1:6];
   double omega[1:4][1:6], alpha[1:4][1:6];
   double rp, beta;
   double psi;
   double complex P[1:4], Vp[1:4], Ap[1:4];
   int i;
   CFourbarSlider fslider;

   /* Default specification of the fourbar-slider linkage. */
   r[1] = 0.12;  r[2] = 0.04;  r[3] = 0.12;
   r[4] = 0.07;  r[5] = 0.06;  r[6] = 0.09;
   for(i = 1; i <= 4; i++)
   {
      theta[i][1] = M_DEG2RAD(10.0);
      theta[i][2] = M_DEG2RAD(70.0);
      omega[i][2] = 10.0;              // rad/sec
      alpha[i][2] = 0;                 // rad/sec^2
   }
   psi = M_DEG2RAD(30.0);
   rp = 0.05;  beta = M_DEG2RAD(45.0);

   /* Perform fourbar-slider linkage analysis. */
   fslider.uscUnit(false);
   fslider.setLinks(r, theta[1][1], psi);
   fslider.setCouplerPoint(COUPLER_LINK6, rp, beta);
   fslider.angularPos(theta[1], theta[2], theta[3], theta[4]);
   fslider.couplerPointPos(COUPLER_LINK6, theta[1][2], P);
   fslider.angularVel(theta[1], omega[1]);
   fslider.angularVel(theta[2], omega[2]);
   fslider.angularVel(theta[3], omega[3]);
   fslider.angularVel(theta[4], omega[4]);
   Vp[1] = fslider.couplerPointVel(COUPLER_LINK6, theta[1], omega[1]);
   Vp[2] = fslider.couplerPointVel(COUPLER_LINK6, theta[2], omega[2]);
   Vp[3] = fslider.couplerPointVel(COUPLER_LINK6, theta[3], omega[3]);
   Vp[4] = fslider.couplerPointVel(COUPLER_LINK6, theta[4], omega[4]);
   fslider.angularAccel(theta[1], omega[1], alpha[1]);
   fslider.angularAccel(theta[2], omega[2], alpha[2]);
   fslider.angularAccel(theta[3], omega[3], alpha[3]);
   fslider.angularAccel(theta[4], omega[4], alpha[4]);
   Ap[1] = fslider.couplerPointAccel(COUPLER_LINK6, theta[1], omega[1], alpha[1]);
   Ap[2] = fslider.couplerPointAccel(COUPLER_LINK6, theta[2], omega[2], alpha[2]);
   Ap[3] = fslider.couplerPointAccel(COUPLER_LINK6, theta[3], omega[3], alpha[3]);
   Ap[4] = fslider.couplerPointAccel(COUPLER_LINK6, theta[4], omega[4], alpha[4]);
```

Program 24: Program for computing the point position, velocity, and acceleration of the coupler point of a fourbar-slider mechanism.

```
    /* Display the results. */
    printf("Circuit 1: Coupler Point Acceleration\n");
    printf("   P = %.3f\n", P[1]);
    printf("   Vp = %.3f\n", Vp[1]);
    printf("   Ap = %.3f\n\n", Ap[1]);
    printf("Circuit 2: Coupler Point Acceleration\n");
    printf("   P = %.3f\n", P[2]);
    printf("   Vp = %.3f\n", Vp[2]);
    printf("   Ap = %.3f\n\n", Ap[2]);
    printf("Circuit 3: Coupler Point Acceleration\n");
    printf("   P = %.3f\n", P[3]);
    printf("   Vp = %.3f\n", Vp[3]);
    printf("   Ap = %.3f\n\n", Ap[3]);
    printf("Circuit 4: Coupler Point Acceleration\n");
    printf("   P = %.3f\n", P[4]);
    printf("   Vp = %.3f\n", Vp[4]);
    printf("   Ap = %.3f\n\n", Ap[4]);

    return 0;
}
```

Program 24: Program for computing the point position, velocity, and acceleration of the coupler point of a fourbar-slider mechanism (Contd.).

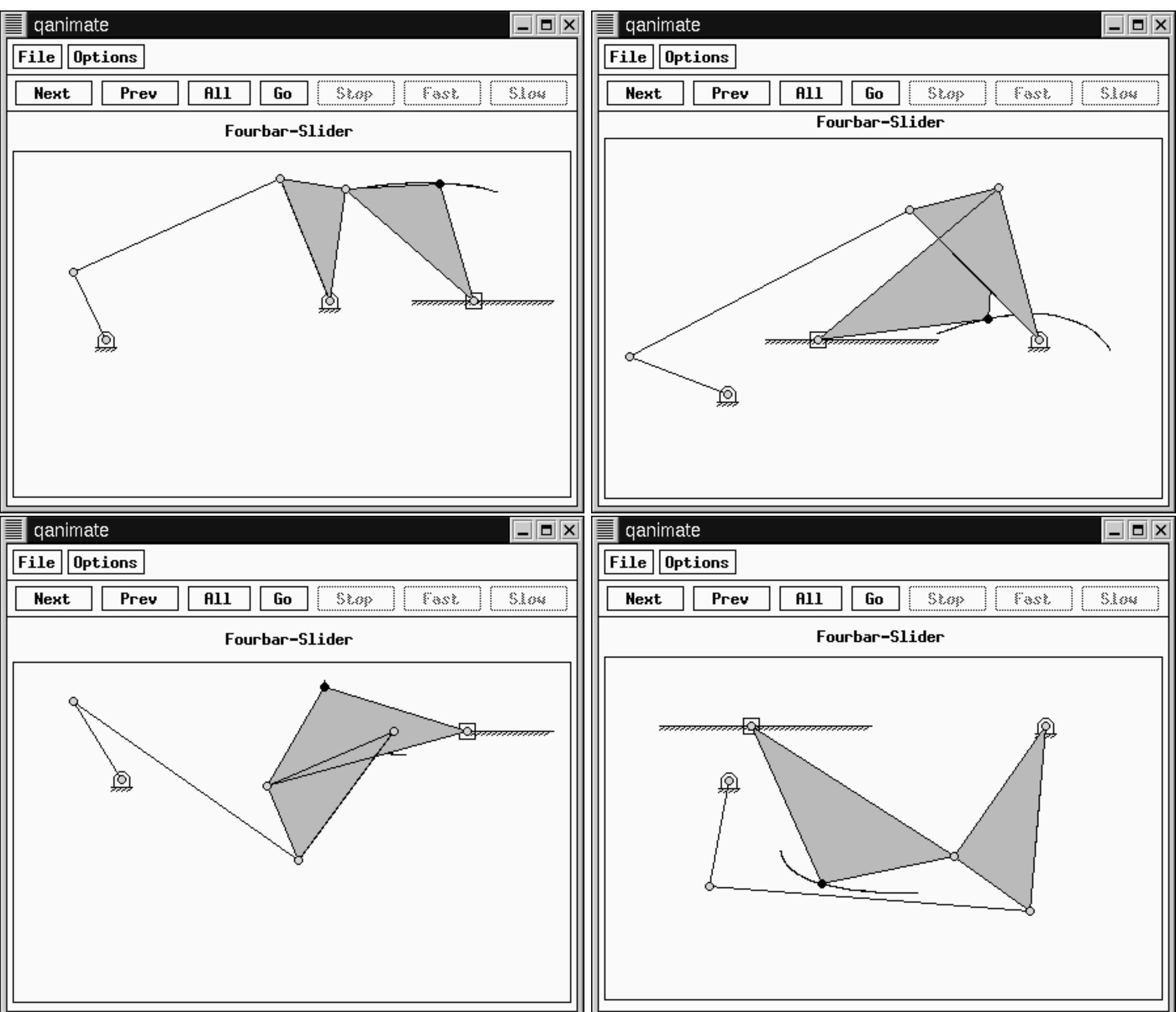
5.1.4 Animation

Animation of the fourbar-slider mechanism, or any other type of linkage, typically provides a better understanding of its operation. Simulating the movement of a fourbar-slider will help determine whether its parameters meet the desired design requirements. Analysis of the following problem problem will provide a better understanding of how animation can be handled by class `CFourbarSlider`.

> **Problem 2:** Given the dimensions of a barslider linkage, trace the path of the coupler point P throughout the allowable input range of link 2. The dimensions of the system are $r_1 = 12cm$, $r_2 = 4cm$, $r_3 = 12cm$, $r_4 = 7cm$, $r_5 = 6cm$, $r_6 = 9cm$, $r_p = 5cm$, $\theta_1 = 10\,\deg$, $\beta = 45\,\deg$, and $\psi = 30\,\deg$. (See Figure 5.1 for one of the layouts.)

The first step in creating an animation of the define the linkage parameters and set values to the fourbar-slider class. This is done with member functions `setLinks()`, `setCouplerPoint()`, and `setNumPoints()`. Notice that `setCouplerPoint()` has two macros as arguments. The first macro, `COUPLER_LINK6` specifies that the coupler is attached to link 6. The other macro argument, `TRACE_ON`, is a special macro for animation purposes. It specifies that the coupler point attached to link 6 should be traced, which is what the above problem requires. Member function `setNumPoints()` is also specifically for the animation feature of class `CFourbarSlider`. The argument `numpoints` specify the number of frames that the animation should include. Thus, after setting all the required parameters, the fourbar-slider can be animated with member function `animation()`, whose only argument describes the branch number of the mechanism to be animated.

Notice that the range of motion of input link 2 is determined internally by function `animation()`. However, if it was desired to calculate the input range of motion, class `CFourbar` member functions `grashof()` and `getJointLimits()` can be used. `grashof()` is used to determine the type of linkage, while `getJointLimits()` can compute the input/output ranges of the fourbar section of the mechanism. Also keep in mind that if the fourbar section was a Grashof Rocker-Rocker, then there would exist a total of eight branches for the fourbar-slider since there would be four possible input ranges.



5.1.5 Web-Based Fourbar-Slider Linkage Analysis

Figure 5.2 shows the internet web page for the analysis and animation of fourbar-slider mechanisms. The web page shown in Figure 5.3 can be used for performing analysis on the various links, slider, and coupler point of the fourbar-slider linkage. Using the parameters specified in Problem 2, the values for the slider, coupler point, and angular properties for the previously defined fourbar-slider mechanism are shown in Figure 5.4.

For simulating the motion of the fourbar-slider mechanism, Figure 5.5 can be used. Figure 5.6 is a snapshot for the first geometric inversion of the fourbar-slider linkage defined in the previous problems. Note that if the fourbar section of the fourbar-slider mechanism is classified as a Grashof Rocker-Rocker mechanism, there would be a total of 8 geometric inversions.

```
#include<sixbar.h>

int main()
{
   double r[1:6], theta[1:4][1:6];
   double rp, beta;
   double omega2, alpha2;
   double complex P[1:4];
   double psi;
   int numpoints = 50;
   int i;
   CFourbarSlider fslider;

   /* Default specification of the fourbar-slider linkage. */
   r[1] = 0.12;  r[2] = 0.04;  r[3] = 0.12;
   r[4] = 0.07;  r[5] = 0.06;  r[6] = 0.09;
   for(i = 1; i <= 4; i++)
   {
      theta[i][1] = M_DEG2RAD(10.0);
   }
   psi = M_DEG2RAD(30.0);
   rp = 0.05;  beta = M_DEG2RAD(45.0);
   omega2 = 5.0; // rad/sec
   alpha2 = 0.0; // rad/sec^2

   /* Perform fourbar-slider linkage analysis. */
   fslider.uscUnit(false);
   fslider.setLinks(r, theta[1][1], psi);
   fslider.setCouplerPoint(COUPLER_LINK6, rp, beta, TRACE_ON);
   fslider.setNumPoints(numpoints);
   fslider.animation(1);
   fslider.animation(2);
   fslider.animation(3);
   fslider.animation(4);

   return 0;
}
```

Program 25: Program using animation to trace the coupler curve attached to link 6 of the fourbar-slider mechanism.

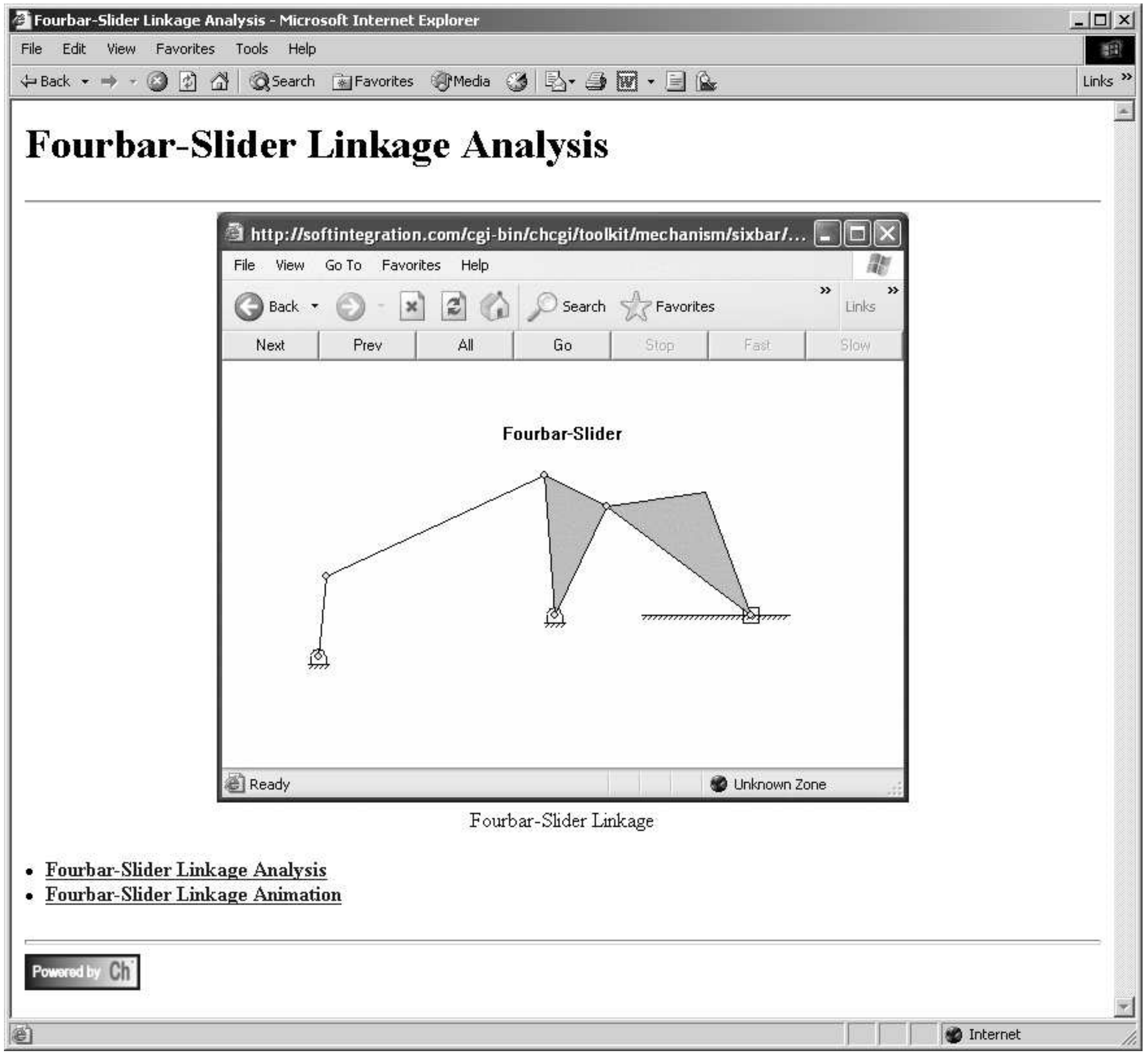


Figure 5.2: Main web page for fourbar-slider linkage analysis.

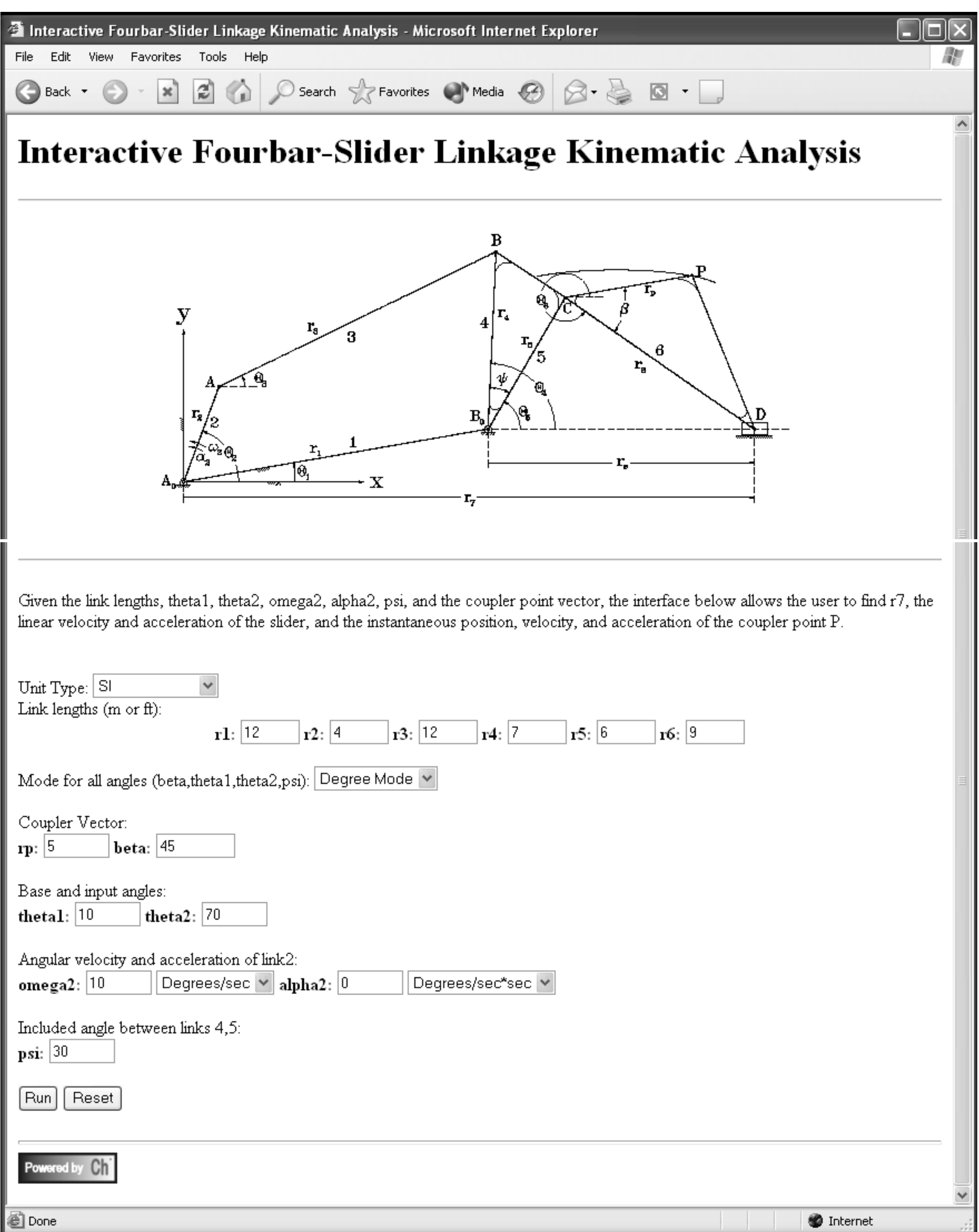


Figure 5.3: Web page for fourbar-slider linkage analysis.

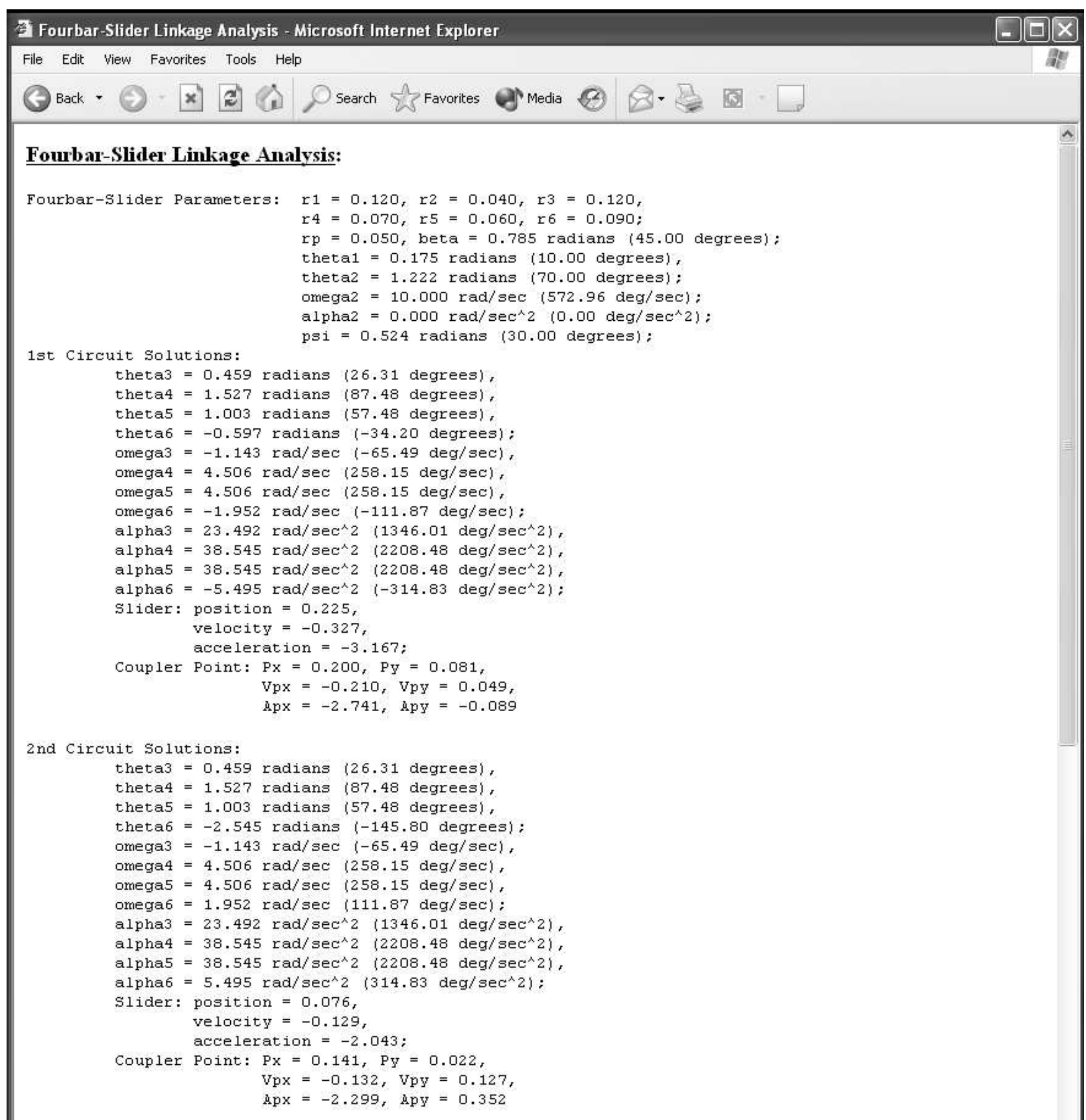


Figure 5.4: Output of web-based fourbar-slider linkage analysis.

```
3rd Circuit Solutions:
        theta3 = -0.777 radians (-44.52 degrees),
        theta4 = -1.845 radians (-105.70 degrees),
        theta5 = -2.368 radians (-135.70 degrees),
        theta6 = 0.484 radians (27.75 degrees);
        omega3 = -0.286 rad/sec (-16.36 deg/sec),
        omega4 = -5.934 rad/sec (-340.01 deg/sec),
        omega5 = -5.934 rad/sec (-340.01 deg/sec),
        omega6 = -3.199 rad/sec (-183.30 deg/sec);
        alpha3 = 61.343 rad/sec^2 (3514.69 deg/sec^2),
        alpha4 = 46.290 rad/sec^2 (2652.22 deg/sec^2),
        alpha5 = 46.290 rad/sec^2 (2652.22 deg/sec^2),
        alpha6 = 11.811 rad/sec^2 (676.72 deg/sec^2);
        Slider: position = 0.155,
                velocity = -0.115,
                acceleration = 2.142;
        Coupler Point: Px = 0.090, Py = 0.027,
                       Vpx = -0.096, Vpy = 0.207,
                       Apx = 2.736, Apy = -0.825

4th Circuit Solutions:
        theta3 = -0.777 radians (-44.52 degrees),
        theta4 = -1.845 radians (-105.70 degrees),
        theta5 = -2.368 radians (-135.70 degrees),
        theta6 = 2.657 radians (152.25 degrees);
        omega3 = -0.286 rad/sec (-16.36 deg/sec),
        omega4 = -5.934 rad/sec (-340.01 deg/sec),
        omega5 = -5.934 rad/sec (-340.01 deg/sec),
        omega6 = 3.199 rad/sec (183.30 deg/sec);
        alpha3 = 61.343 rad/sec^2 (3514.69 deg/sec^2),
        alpha4 = 46.290 rad/sec^2 (2652.22 deg/sec^2),
        alpha5 = 46.290 rad/sec^2 (2652.22 deg/sec^2),
        alpha6 = -11.811 rad/sec^2 (-676.72 deg/sec^2);
        Slider: position = -0.004,
                velocity = -0.383,
                acceleration = 4.762;
        Coupler Point: Px = 0.027, Py = -0.036,
                       Vpx = -0.201, Vpy = 0.102,
                       Apx = 3.766, Apy = 0.204
```

Done   Internet

Figure 5.4: Output of web-based fourbar-slider linkage analysis (Contd.).

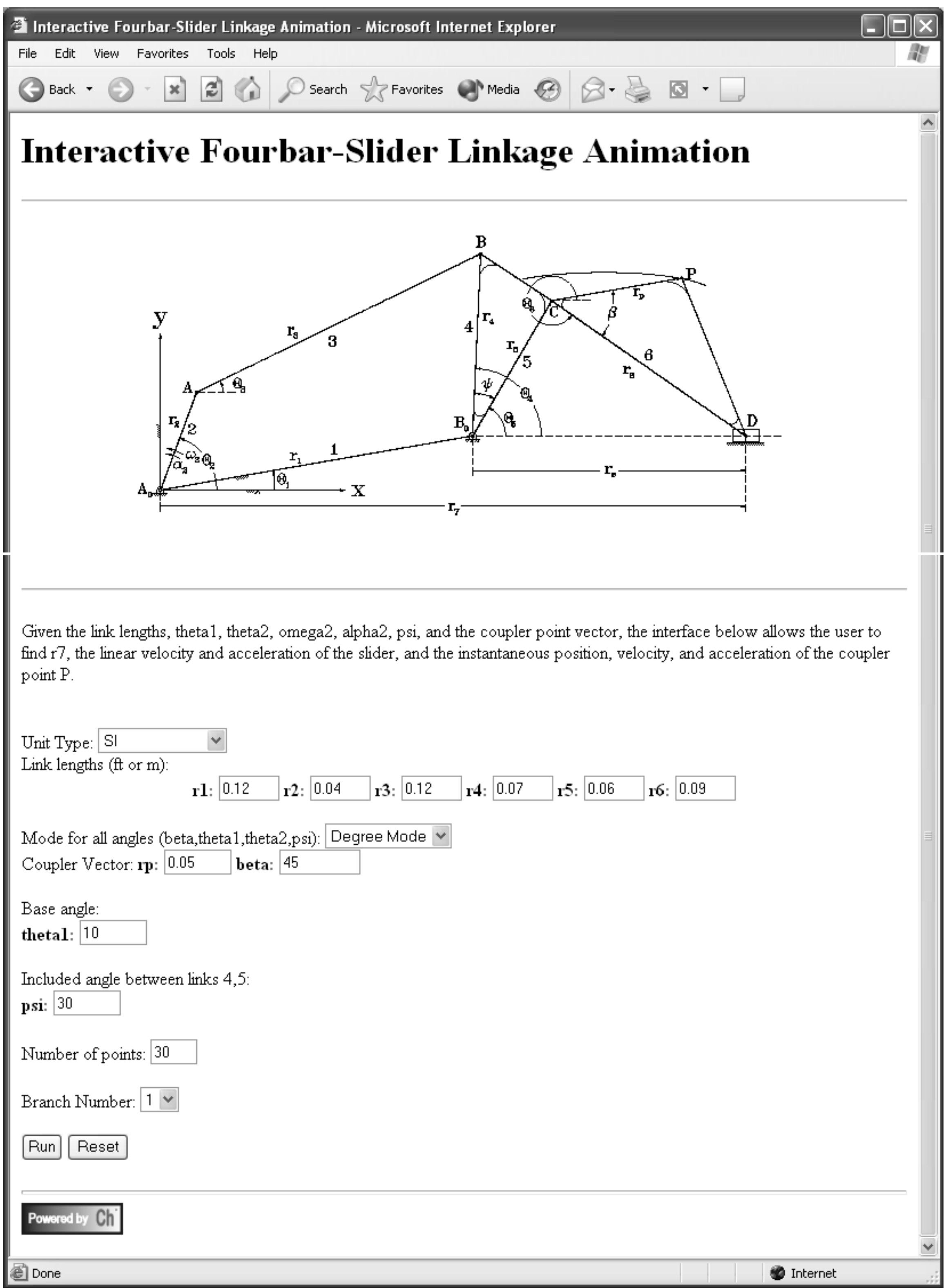


Figure 5.5: Web page for fourbar-slider linkage animation.

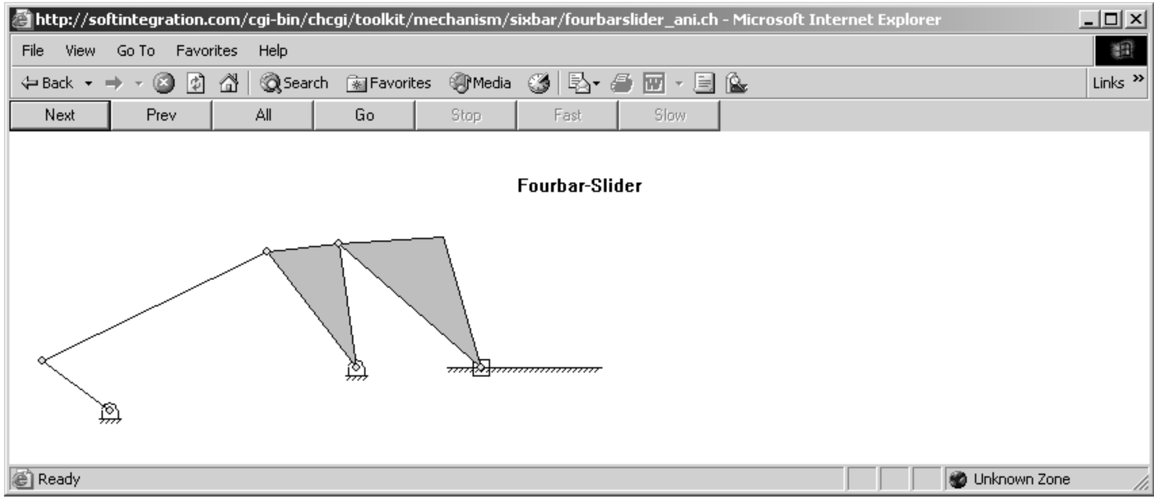


Figure 5.6: Snapshot of fourbar-slider animation for branch 1.

5.2 Watt Six-bar (I) Linkage

A Watt (I) sixbar linkage is assembled from two fourbar mechanisms. All four links $r"_3$, r'_4, r_5, and r_6 of the second fourbar linkage are movable. For the Watt (I) sixbar in Figure 5.7, note that there are two coupler points attached to the mechanism at links 5 and 6.

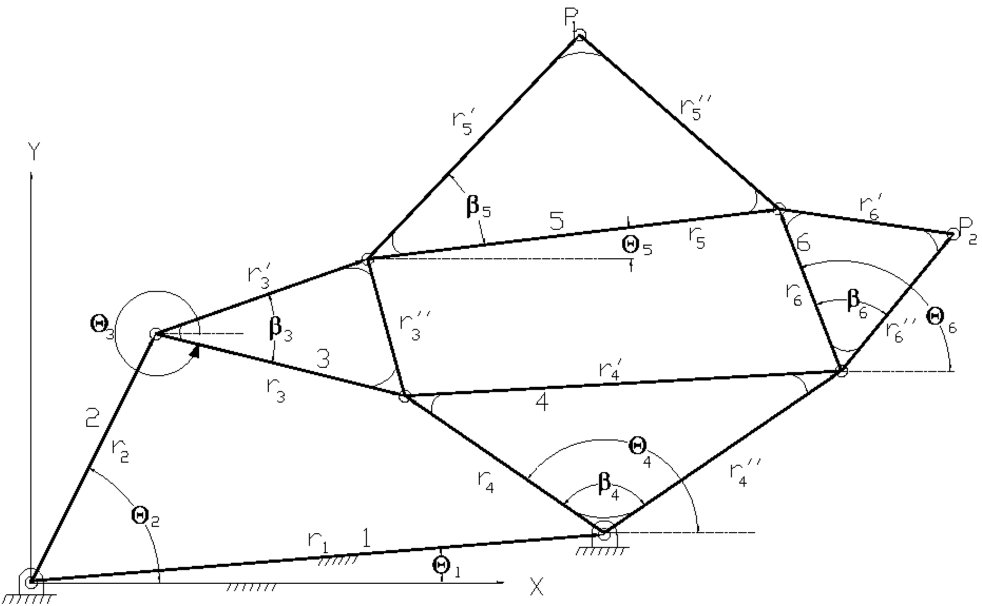


Figure 5.7: Watt (I) Sixbar Linkage

5.2.1 Position Analysis

By applying geometry, complex arithmetic and vector analysis similar to the previous sections, equations for calculating all unknown angles of the mechanism can be found. Analyzing the Watt (I) linkage shows that two loop closure equations may be utilized to perform the desired task. The first loop equation shown below can be used to solve for θ_3 and θ_4. The vector equation (5.38) is rewritten so that the two unknown angles are on the left-hand side of the equation (5.40).

$$\mathbf{r}_1 + \mathbf{r}_4 = \mathbf{r}_2 + \mathbf{r}_3 \tag{5.38}$$

$$r_1 e^{i\theta_1} + r_4 e^{i\theta_4} = r_2 e^{i\theta_2} + r_3 e^{i\theta_3} \tag{5.39}$$

$$r_3 e^{i\theta_3} - r_4 e^{i\theta_4} = r_1 e^{i\theta_1} - r_2 e^{i\theta_2} \tag{5.40}$$

From equation (5.40), the two sets of θ_3 and θ_4 can now be solved with function `complexsolve()`, where the inputs are $n1 = 2$, $n2 = 4$, $theta_or_r 1 = r_3$, $theta_or_r 2 = -r_4$, and $z = r_1 e^{i\theta_1} - r_2 e^{i\theta_2}$. By obtaining solutions for θ_3 and θ_4, the second loop equation (5.41) can be analyzed to determine values for θ_5 and θ_6. Rewritting equation (5.41) and again isolating all the terms with unknown values on the left-hand side, `complexsolve()` can be used to solve for θ_5 and θ_6.

$$\mathbf{r}_1 + \mathbf{r}''_4 + \mathbf{r}_6 = \mathbf{r}_2 + \mathbf{r}'_3 + \mathbf{r}_5 \tag{5.41}$$

$$r_1 e^{i\theta_1} + r''_4 e^{i\theta''_4} + r_6 e^{i\theta_6} = r_2 e^{i\theta_2} + r'_3 e^{i\theta'_3} + r_5 e^{i\theta_5} \tag{5.42}$$

$$r_5 e^{i\theta_5} - r_6 e^{i\theta_6} = r_1 e^{i\theta_1} - r_2 e^{i\theta_2} - r'_3 e^{i\theta'_3} + r''_4 e^{i\theta''_4} \tag{5.43}$$

where

$$\theta'_3 = \theta_3 + \beta_3 \tag{5.44}$$

$$\theta''_4 = \theta_4 - \beta_4 \tag{5.45}$$

Note that there are four sets of solutions for θ_5 and θ_6. The reason is that there are two geometric inversions for each fourbar linkage assembly. Calculating the angular position of the various links of the Watt (I) Sixbar linkage can also be done by using member fuction `angularPos()` of class `CWattSixbarI`. Its prototype is as follow:

```
void CWattSixbarI::angularPos(double theta_1[1:], theta_2[1:],
                              theta_3[1:], theta_4[1:]);
```

where each of the four arguments is an array of `double` type used to store the four different angular position solutions of the Watt (I) sixbar linkage.

5.2.2 Velocity Analysis

The angular velocity values of each link can be determined by first obtaining the derivatives of equations (5.40) and (5.43). The results of differentiating equations (5.40) and (5.43) are equations (5.46) and (5.47), respectively.

$$ir_3\omega_3 e^{i\theta_3} - ir_4\omega_4 e^{i\theta_4} = -ir_2\omega_2 e^{i\theta_2} \tag{5.46}$$

$$ir_5\omega_5 e^{i\theta_5} - ir_6\omega_6 e^{i\theta_6} = -ir_2\omega_2 e^{i\theta_2} - ir'_3\omega_3 e^{i\theta'_3} + ir''_4\omega_4 e^{i\theta''_4} \tag{5.47}$$

In order to solve for ω_3, ω_4, ω_5, and ω_6, the above equations must be multiplied by the term $e^{-i\theta_j}$ (for $j = 3, 4, 5, 6$). By separately multiplying equation (5.46) with $e^{-i\theta_3}$ and $e^{-i\theta_4}$, equations (5.48) and (5.49)

are generated. Notice that the imaginary i term has also been factored out of equations (5.46) and (5.47). By only considering the imaginary part of equations (5.48) and (5.49), ω_3 and ω_4 can then be calculated. Rearranging equations (5.50) and (5.51) to determine the analytical solution of ω_3 and ω_4 will result in equations (5.52) and (5.53).

$$r_3\omega_3 e^{i(\theta_3-\theta_4)} - r_4\omega_4 = -r_2\omega_2 e^{i(\theta_2-\theta_4)} \quad (5.48)$$

$$r_3\omega_3 - r_4\omega_4 e^{i(\theta_4-\theta_3)} = -r_2\omega_2 e^{i(\theta_2-\theta_3)} \quad (5.49)$$

$$r_3\omega_3 \sin(\theta_3-\theta_4) = -r_2\omega_2 \sin(\theta_2-\theta_4) \quad (5.50)$$

$$-r_4\omega_4 \sin(\theta_4-\theta_3) = -r_2\omega_2 \sin(\theta_2-\theta_3) \quad (5.51)$$

$$\omega_3 = -\frac{r_2\omega_2 \sin(\theta_2-\theta_4)}{r_3 \sin(\theta_3-\theta_4)} \quad (5.52)$$

$$\omega_4 = \frac{r_2\omega_2 \sin(\theta_2-\theta_3)}{r_4 \sin(\theta_4-\theta_3)} \quad (5.53)$$

After determining ω_3 and ω_4, the above process can be repeated on equation (5.47) to find ω_5 and ω_6. Equation (5.47) is separatedly multiplied by $e^{-i\theta_5}$ and $e^{-i\theta_6}$ in order to determine the two desired ω's (5.54) and (5.55). The terms ω_5 and ω_6 can now be found by isolating the imaginary parts of equations (5.54) and (5.55) to result in equations (5.56) and (5.57), respectively, and then rearranging these two equations to obtain equations (5.58) and (5.59). Similar to the calculating the angular positions, solutions to ω_3, ω_4, ω_5, and ω_6 can be determined by member function `angularVel()` of the Watt (I) Sixbar class. The prototype for function `angularVel()` is

```
void CWattSixbarI::angularVel(double theta[1:], omega[1:]);
```

where `theta` is an array that contains the angular position values of each link, and `omega` stores the angular velocity values. Note that each call of function `angularVel()` only produces one set of solution for angular velocity. Thus, `angularVel()` needs to be called four times, each time using a different set of `theta` solutions, to obtain the four possible angular velocity solutions.

$$r_5\omega_5 e^{i(\theta_5-\theta_6)} - r_6\omega_6 = -r_2\omega_2 e^{i(\theta_2-\theta_6)} - r_3'\omega_3 e^{i(\theta_3'-\theta_6)} + r_4''\omega_4 e^{i(\theta_4''-\theta_6)} \quad (5.54)$$

$$r_5\omega_5 - r_6\omega_6 e^{i(\theta_6-\theta_5)} = -r_2\omega_2 e^{i(\theta_2-\theta_5)} - r_3'\omega_3 e^{i(\theta_3'-\theta_5)} + r_4''\omega_4 e^{i(\theta_4''-\theta_5)} \quad (5.55)$$

$$r_5\omega_5 \sin(\theta_5-\theta_6) = -r_2\omega_2 \sin(\theta_2-\theta_6) - r_3'\omega 3 \sin(\theta_3'-\theta_6) + r_4''\omega_4 \sin(\theta_4''-\theta_6) \quad (5.56)$$

$$-r_6\omega_6 \sin(\theta_6-\theta_5) = -r_2\omega_2 \sin(\theta_2-\theta_5) - r_3'\omega_3 \sin(\theta_3'-\theta_5) + r_4''\omega_4 \sin(\theta_4''-\theta_5) \quad (5.57)$$

$$\omega_5 = -\frac{r_2\omega_2 \sin(\theta_2-\theta_6) + r_3'\omega_3 \sin(\theta_3'-\theta_6) - r_4''\omega_4 \sin(\theta_4''-\theta_6)}{r_5 \sin(\theta_5-\theta_6)} \quad (5.58)$$

$$\omega_6 = \frac{r_2\omega_2 \sin(\theta_2-\theta_5) + r_3'\omega_3 \sin(\theta_3'-\theta_5) - r_4''\omega_4 \sin(\theta_4''-\theta_5)}{r_6 \sin(\theta_6-\theta_5)} \quad (5.59)$$

5.2.3 Acceleration Analysis

The values for the angular accelerations of the links are found by differentiating the angular velocity equations (5.46) and (5.47). The results are equations (5.60) and (5.61). Similar to the velocity equations, these are then multiplied by an $e^{-i\theta_j}$ term to isolate a particular α_j term.

$$(i\alpha_3 - \omega_3^2) r_3 e^{i\theta_3} - (i\alpha_4 - \omega 4^2) r_4 e^{i\theta_4} = (-i\alpha_2 + \omega_2^2) r_2 e^{i\theta_2} \quad (5.60)$$

$$\begin{aligned}(i\alpha_5-\omega_5^2)r_5e^{i\theta_5}-(i\alpha_6-\omega_6^2)r_6e^{i\theta_6} &= (-i\alpha_2+\omega_2^2)r_2e^{i\theta_2}-(i\alpha_3-\omega_3^2)r_3'e^{i\theta_3'}\\ &\quad+(i\alpha_4-\omega_4^2)r_4''e^{i\theta_4''} \qquad (5.61)\end{aligned}$$

Multiplying equation (5.60) by $e^{-i\theta_3}$ and $e^{-i\theta_4}$ will generate equations (5.62) and (5.63), which can then be used to find for α_3 and α_4. By only considering the real parts of these equations, the desired angular acceleration values are isolated, and equations (5.64) and (5.65) can be evaluated and solve for α_3 and α_4, respectively (5.66) and (5.67). Note that the imaginary i term is replaced by $e^{i(\pi/2)}$, an equivalent term, in the subsequent equations.

$$\begin{aligned}r_3\alpha_3e^{i(\theta_3-\theta_4+\pi/2)}-r_4\alpha_4e^{i(\pi/2)} &= -r_2\alpha_2e^{i(\theta_2-\theta_4+\pi/2)}+r_2\omega_2^2e^{i(\theta_2-\theta_4)}+r_3\omega_3^2e^{i(\theta_3-\theta_4)}\\ &\quad-r_4\omega_4^2 \qquad (5.62)\\ r_3\alpha_3e^{i(\pi/2)}-r_4\alpha_4e^{i(\theta_4-\theta_3+\pi/2)} &= -r_2\alpha_2e^{i(\theta_2-\theta_4+\pi/2)}+r_2\omega_2^2e^{i(\theta_2-\theta_4)}+r_3\omega_3^2\\ &\quad-r_4\omega_4^2e^{i(\theta_4-\theta_3)} \qquad (5.63)\end{aligned}$$

$$\begin{aligned}-r_3\alpha_3\sin(\theta_3-\theta_4) &= r_2\alpha_2\sin(\theta_2-\theta_4)+r_2\omega_2^2\cos(\theta_2-\theta_4)+r_3\omega_3^2\cos(\theta_3-\theta_4)\\ &\quad-r_4\omega_4^2 \qquad (5.64)\end{aligned}$$

$$\begin{aligned}r_4\alpha_4\sin(\theta_4-\theta_3) &= r_2\alpha_2\sin(\theta_2-\theta_3)+r_2\omega_2^2\cos(\theta_2-\theta_3)+r_3\omega_3^2\\ &\quad-r_4\omega_4^2\sin(\theta_4-\theta_3) \qquad (5.65)\end{aligned}$$

$$\begin{aligned}\alpha_3 &= -\frac{r_2\alpha_2\sin(\theta_2-\theta_4)+r_2\omega_2^2\cos(\theta_2-\theta_4)+r_3\omega_3^2\cos(\theta_3-\theta_4)}{r_3\sin(\theta_3-\theta_4)}\\ &\quad+\frac{r_4\omega_4^2}{r_3\sin(\theta_3-\theta_4)} \qquad (5.66)\end{aligned}$$

$$\begin{aligned}\alpha_4 &= \frac{r_2\alpha_2\sin(\theta_2-\theta_3)+r_2\omega_2^2\cos(\theta_2-\theta_3)+r_3\omega_3^2}{r_4\sin(\theta_4-\theta_3)}\\ &\quad-\frac{r_4\omega_4^2\sin(\theta_4-\theta_3)}{r_4\sin(\theta_4-\theta_3)} \qquad (5.67)\end{aligned}$$

With a similar process, angular accelerations α_5 and α_6 can be solved from equation (5.61) by first multiplying it by $e^{-i\theta_5}$ and $e^{-i\theta_6}$ to result in equations (5.68) and (5.69). Again, the imaginary i terms have been replaced by $e^{i(\pi/2)}$.

$$\begin{aligned}r_5\alpha_5e^{i(\theta_5-\theta_6+\pi/2)}-r_6\alpha_6e^{i(\pi/2)} &= -r_2\alpha_2e^{i(\theta_2-\theta_6+\pi/2)}+r_2\omega_2^2e^{i(\theta_2-\theta_6)}-r_3'\alpha_3e^{i(\theta_3'-\theta_6+\pi/2)}\\ &\quad+r_3'\omega_3^2e^{i(\theta_3'-\theta_6)}+r_4''\alpha_4e^{i(\theta_4''-\theta_6+\pi/2)}-r_4''\omega_4^2e^{i(\theta_4''-\theta_6)}\\ &\quad+r_5\omega_5^2e^{i(\theta_5-\theta_6)}-r_6\omega_6^2 \qquad (5.68)\\ r_5\alpha_5e^{i(\pi/2)}-r_6\alpha_6e^{i(\theta_6-\theta_5+\pi/2)} &= -r_2\alpha_2e^{i(\theta_2-\theta_5+\pi/2)}+r_2\omega_2^2e^{i(\theta_2-\theta_5)}-r_3'\alpha_3e^{i(\theta_3'-\theta_5+\pi/2)}\\ &\quad+r_3'\omega_3^2e^{i(\theta_3'-\theta_5)}+r_4''\alpha_4e^{i(\theta_4''-\theta_6+\pi/2)}-r_4''\omega_4^2e^{i(\theta_4''-\theta_6)}\\ &\quad+r_5\omega_5^2-r_6\omega_6^2e^{i(\theta_6-\theta_5)} \qquad (5.69)\end{aligned}$$

Considering only the real parts of the previous two equations, α_5 and α_6 can be isolated and determined (5.72) and (5.73).

$$\begin{aligned}-r_5\alpha_5\sin(\theta_5-\theta_6) &= r_2\alpha_2\sin(\theta_2-\theta_6)+r_2\omega_2^2\cos(\theta_2-\theta_6)+r_3'\alpha_3\sin(\theta_3'-\theta_6)+r_3'\omega_3^2\cos(\theta_3'-\theta_6)\\ &\quad-r_4''\alpha_4\sin(\theta_4''-\theta_6)-r_4''\omega_4^2\cos(\theta_4''-\theta_6)+r_5\omega_5^2\cos(\theta_5-\theta_6)-r_6\omega_6^2 \qquad (5.70)\\ r_6\alpha_6\sin(\theta_6-\theta_5) &= r_2\alpha_2\sin(\theta_2-\theta_5)+r_2\omega_2^2\cos(\theta_2-\theta_5)+r_3'\alpha_3\sin(\theta_3'-\theta_5)+r_3'\omega_3^2\cos(\theta_3'-\theta_6)\\ &\quad-r_4''\alpha_4\sin(\theta_4''-\theta_6)-r_4''\omega_4^2\cos(\theta_4''-\theta_6)+r_5\omega_5^2-r_6\omega_6^2\cos(\theta_6-\theta_5) \qquad (5.71)\end{aligned}$$

$$\begin{aligned}\alpha_5 &= -\frac{r_2\alpha_2\sin(\theta_2-\theta_6)+r_2\omega_2^2\cos(\theta_2-\theta_6)+r_3'\alpha_3\sin(\theta_3'-\theta_6)+r_3'\omega_3^2\cos(\theta_3'-\theta_6)}{r_5\sin(\theta_5-\theta_6)}\\ &+\frac{r_4''\alpha_4\sin(\theta_4''-\theta_6)+r_4''\omega_4^2\cos(\theta_4''-\theta_6)-r_5\omega_5^2\cos(\theta_5-\theta_6)+r_6\omega_6^2}{r_5\sin(\theta_5-\theta_6)} \quad (5.72)\end{aligned}$$

$$\begin{aligned}\alpha_6 &= \frac{r_2\alpha_2\sin(\theta_2-\theta_5)+r_2\omega_2^2\cos(\theta_2-\theta_5)+r_3'\alpha_3\sin(\theta_3'-\theta_5)+r_3'\omega_3^2\cos(\theta_3'-\theta_5)}{}\\ &+\frac{-r_4''\alpha_4\sin(\theta_4''-\theta_6)-r_4''\omega_4^2\cos(\theta_4''-\theta_6)+r_5\omega_5^2-r_6\omega_6^2\cos(\theta_6-\theta_5)}{r_6\sin(\theta_6-\theta_5)} \quad (5.73)\end{aligned}$$

Note that the angular acceleration calculations can also be performed by member function `angularAccel()` with prototype

```
void CWattSixbarI::angularAccel(double theta[1:], omega[1:], alpha[1:]);
```

Arguments `theta` and `omega` are `double` arrays used to store the angular position and velocity values, respectively. The last argument `alpha` is used to store the angular acceleration values calculated by member function `angularAccel()`. Similar to `angularVel()`, this function also has to be called four times in order to acquire the four different set of angular acceleration solutions.

To help illustrate the benerfit of using class `CWattSixbarI`, consider the problem below.

> **Problem 1:** A Watt (I) sixbar linkage has parameters: $r_1 = 8cm, r_2 = 4cm, r_3 = 10cm, r_4 = 7cm, r_5 = 7cm, r_6 = 8cm$, $\theta_1 = 10^o$, $\theta_2 = 25^o$, $r_3' = 4cm, r_4'' = 9cm$, $\beta_3 = 30^o$, $\beta_4 = 50^o$, $\omega_2 = 10^o/sec$, and $\alpha_2 = 0$. Find the angular accelerations α_3, α_4, α_5, α_6.

Now, in order to find α_3, α_4, α_5, and α_6, the angular position and velocity of the various angles have to be known. Calculating all these values is time consuming if the previously derived equations are used instead of class `CWattSixbarI`. However, using the class specifically written for analysis of a Watt (I) sixbar linkage, the above probelm can easily be solved. Program 26 utilizes the member functions of class `CWattSixbarI` to calculate the desired values and the solution is shown below.

```
Solution #1:
 alpha5 = 0.008 rad/sec^2 (0.47 deg/sec^2),
 alpha6 = 0.117 rad/sec^2 (6.69 deg/sec^2);
Solution #2:
 alpha5 = 0.128 rad/sec^2 (7.32 deg/sec^2),
 alpha6 = 0.019 rad/sec^2 (1.10 deg/sec^2);
Solution #3:
 alpha5 = 0.069 rad/sec^2 (3.95 deg/sec^2),
 alpha6 = 0.026 rad/sec^2 (1.48 deg/sec^2);
Solution #4:
 alpha5 = 0.016 rad/sec^2 (0.93 deg/sec^2),
 alpha6 = 0.059 rad/sec^2 (3.40 deg/sec^2);
```

5.2.4 Coupler Position, Velocity, and Acceleration

The position of the two coupler points in Figure 5.7 can be derived directly from vector analysis. The position of the coupler point, P_1, is simply the vector sum of $\mathbf{r}_2$, $\mathbf{r}_3'$, and $\mathbf{r}_5'$ shown in equation (5.74). Likewise, the position of P_2 is written as equation (5.75). Writing these two equations in their polar form (5.76) and (5.77) reveal that the position of either point can be determined given the required link lengths and angles. The first and second derivatives of equations (5.76) and (5.77) will produce the equation for the coupler point velocities (5.78) and (5.79) and coupler point accelerations (5.80) and (5.81), respectively. Similar

```
#include<stdio.h>
#include<sixbar.h>

int main()
{
   double r[1:6];
   double rP3, beta3,
          rPP4, beta4;
   double theta[1:4][1:6], omega[1:4][1:6], alpha[1:4][1:6];
   double theta1 = M_DEG2RAD(10), theta2 = M_DEG2RAD(25);
   double omega2 = M_DEG2RAD(10), alpha2 = 0;
   CWattSixbarI wattI;
   int i;

   /* Define Watt (I) Sixbar linkage. */
   r[1] = 0.08;  r[2] = 0.04;
   r[3] = 0.10;  r[4] = 0.07;
   r[5] = 0.07;   r[6] = 0.08;
   rP3 = 0.04;  beta3 = M_DEG2RAD(30);
   rPP4 = 0.09;  beta4 = M_DEG2RAD(50);
   for(i = 1; i <= 4; i++)
   {
      theta[i][1] = theta1;
      theta[i][2] = theta2;
      omega[i][2] = omega2;
   }

   /* Perform analysis. */
   wattI.setLinks(r, theta1, rP3, beta3, rPP4, beta4);
   wattI.angularPos(theta[1], theta[2], theta[3], theta[4]);
   wattI.angularVel(theta[1], omega[1]);
   wattI.angularVel(theta[2], omega[2]);
   wattI.angularVel(theta[3], omega[3]);
   wattI.angularVel(theta[4], omega[4]);
   wattI.angularAccel(theta[1], omega[1], alpha[1]);
   wattI.angularAccel(theta[2], omega[2], alpha[2]);
   wattI.angularAccel(theta[3], omega[3], alpha[3]);
   wattI.angularAccel(theta[4], omega[4], alpha[4]);

   /* Display results. */
   for(i = 1; i <= 4; i++)
   {
      printf("Solution #%d:\n", i);
      printf("\t alpha5 = %.3f rad/sec^2 (%.2f deg/sec^2),\n",
              alpha[i][5], M_RAD2DEG(alpha[i][5]));
      printf("\t alpha6 = %.3f rad/sec^2 (%.2f deg/sec^2);\n",
              alpha[i][6], M_RAD2DEG(alpha[i][6]));
   }

   return 0;
}
```

Program 26: Program for computing α_3, α_4, α_5, and α_6 using class `CWattSixbarI`.

to the coupler point positions, the velocities and accelerations of the two coupler points can be determined given the required angular velocity and acceleration values. Coupler point position, velocity, and acceleration can also be calculated by using member functions `couplerPointPos()`, `couplerPointVel()`, and `couplerPointAccel()`, respectively.

$$\mathbf{P}_1 = \mathbf{r}_2 + \mathbf{r}'_3 + \mathbf{r}'_5 \tag{5.74}$$

$$\mathbf{P}_2 = \mathbf{r}_1 + \mathbf{r}''_4 + \mathbf{r}''_6 \tag{5.75}$$

$$\mathbf{P}_1 = r_2 e^{i\theta_2} + r'_3 e^{i(\theta_3+\beta_3)} + r'_5 e^{i(\theta_5+\beta_5)} \tag{5.76}$$

$$\mathbf{P}_2 = r_1 e^{i\theta_1} + r''_4 e^{i(\theta_4-\beta_4)} + r''_6 e^{i(\theta_6-\beta_6)} \tag{5.77}$$

$$\dot{\mathbf{P}}_1 = ir_2\omega_2 e^{i\theta_2} + ir'_3\omega_3 e^{i(\theta_3+\beta_3)} + ir'_5\omega_5 e^{i(\theta_5+\beta_5)} \tag{5.78}$$

$$\dot{\mathbf{P}}_2 = ir''_4\omega_4 e^{i(\theta_4-\beta_4)} + ir''_6\omega_6 e^{i(\theta_6-\beta_6)} \tag{5.79}$$

$$\ddot{\mathbf{P}}_1 = ir_2\alpha_2 e^{i\theta_2} - r_2\omega_2^2 e^{i\theta_2} + ir'_3\alpha_3 e^{i(\theta_3+\beta_3)} - r'_3\omega_3^2 e^{i(\theta_3+\beta_3)} + ir'_5\alpha_5 e^{i(\theta_5+\beta_5)} - r'_5\omega_5^2 e^{i(\theta_5+\beta_5)} \tag{5.80}$$

$$\ddot{\mathbf{P}}_2 = ir''_4\alpha_4 e^{i(\theta_4-\beta_4)} - r''_4\omega_4^2 e^{i(\theta_4-\beta_4)} + ir''_6\alpha_6 e^{i(\theta_6-\beta_6)} - r''_6\omega_6^2 e^{i(\theta_6-\beta_6)} \tag{5.81}$$

Problem 2: Consider a Watt (I) sixbar linkage with parameters given in Problem 1. If two coupler points are attached to links 5 and 6 (see Figure 5.7) with properties $r'_5 = 6cm$, $\beta_5 = 30^o$, $r''_6 = 4cm$, and $\beta_6 = 45^o$, find the coupler point position, velocity, and acceleration at this moment in time.

The solution to the above problem is listed as Program 27. Since there are two coupler points for the specified linkage, the member functions used to calculate the coupler point properties must be called for each point. The results is shown below.

```
Solution #1:
   P1 = complex(0.090,0.101), P2 = complex(0.173,0.058)
   Vp1 = complex(-0.001,-0.005), Vp2 = complex(0.003,-0.003)
   Ap1 = complex(-0.004,0.001), Ap2 = complex(-0.005,0.007)
Solution #2:
   P1 = complex(0.108,0.024), P2 = complex(0.133,0.036)
   Vp1 = complex(-0.008,0.003), Vp2 = complex(0.003,0.002)
   Ap1 = complex(0.005,0.001), Ap2 = complex(-0.000,0.006)
Solution #3:
   P1 = complex(0.133,-0.009), P2 = complex(0.083,-0.087)
   Vp1 = complex(-0.013,-0.003), Vp2 = complex(-0.024,0.001)
   Ap1 = complex(0.001,0.003), Ap2 = complex(0.001,0.007)
Solution #4:
   P1 = complex(0.029,-0.028), P2 = complex(0.057,-0.032)
   Vp1 = complex(-0.004,0.023), Vp2 = complex(-0.016,0.007)
   Ap1 = complex(-0.001,-0.001), Ap2 = complex(-0.001,0.005)
```

5.2.5 Animation

The Watt (I) sixbar linkage shown in Figure 5.7 can be animated with member function `animation()`. Using the parameters given in the problem statements above, Program 28 can be used to simulate the motion of the Watt (I) sixbar. After specifying the parameters for the sixbar, member function `animation()` is

```
#include<stdio.h>
#include<sixbar.h>

int main()
{
   double r[1:6];
   double rP3, beta3,
          rPP4, beta4;
   double rP5, beta5,
          rPP6, beta6;
   double theta[1:4][1:6], omega[1:4][1:6], alpha[1:4][1:6];
   double complex P1[1:4], P2[1:4],
                  Vp1[1:4], Vp2[1:4],
                  Ap1[1:4], Ap2[1:4];
   double theta1 = M_DEG2RAD(10), theta2 = M_DEG2RAD(25);
   double omega2 = M_DEG2RAD(10), alpha2 = 0;
   CWattSixbarI wattI;
   int i;

   /* Define Watt (I) Sixbar linkage. */
   r[1] = 0.08;  r[2] = 0.04;
   r[3] = 0.10;  r[4] = 0.07;
   r[5] = 0.07;   r[6] = 0.08;
   rP3 = 0.04;  beta3 = M_DEG2RAD(30);
   rPP4 = 0.09;  beta4 = M_DEG2RAD(50);
   rP5 = 0.06;  beta5 = M_DEG2RAD(30);
   rPP6 = 0.04;  beta6 = M_DEG2RAD(45);
   for(i = 1; i <= 4; i++)
   {
      theta[i][1] = theta1;
      theta[i][2] = theta2;
      omega[i][2] = omega2;
   }

   /* Perform analysis. */
   wattI.setLinks(r, theta1, rP3, beta3, rPP4, beta4);
   wattI.setCouplerPoint(COUPLER_LINK5, rP5, beta5);
   wattI.setCouplerPoint(COUPLER_LINK6, rPP6, beta6);
   wattI.angularPos(theta[1], theta[2], theta[3], theta[4]);
   wattI.couplerPointPos(COUPLER_LINK5, theta2, P1);
   wattI.couplerPointPos(COUPLER_LINK6, theta2, P2);
   wattI.angularVel(theta[1], omega[1]);
   wattI.angularVel(theta[2], omega[2]);
   wattI.angularVel(theta[3], omega[3]);
   wattI.angularVel(theta[4], omega[4]);
   Vp1[1] = wattI.couplerPointVel(COUPLER_LINK5, theta[1], omega[1]);
   Vp1[2] = wattI.couplerPointVel(COUPLER_LINK5, theta[2], omega[2]);
   Vp1[3] = wattI.couplerPointVel(COUPLER_LINK5, theta[3], omega[3]);
   Vp1[4] = wattI.couplerPointVel(COUPLER_LINK5, theta[4], omega[4]);
   Vp2[1] = wattI.couplerPointVel(COUPLER_LINK6, theta[1], omega[1]);
   Vp2[2] = wattI.couplerPointVel(COUPLER_LINK6, theta[2], omega[2]);
   Vp2[3] = wattI.couplerPointVel(COUPLER_LINK6, theta[3], omega[3]);
   Vp2[4] = wattI.couplerPointVel(COUPLER_LINK6, theta[4], omega[4]);
```

Program 27: Program for computing the positions, velocities, and accelerations of the two coupler points specified in Figure 5.7.

```
   wattI.angularAccel(theta[1], omega[1], alpha[1]);
   wattI.angularAccel(theta[2], omega[2], alpha[2]);
   wattI.angularAccel(theta[3], omega[3], alpha[3]);
   wattI.angularAccel(theta[4], omega[4], alpha[4]);
   Ap1[1] = wattI.couplerPointAccel(COUPLER_LINK5, theta[1], omega[1],
                                    alpha[1]);
   Ap1[2] = wattI.couplerPointAccel(COUPLER_LINK5, theta[2], omega[2],
                                    alpha[2]);
   Ap1[3] = wattI.couplerPointAccel(COUPLER_LINK5, theta[3], omega[3],
                                    alpha[3]);
   Ap1[4] = wattI.couplerPointAccel(COUPLER_LINK5, theta[4], omega[4],
                                    alpha[4]);
   Ap2[1] = wattI.couplerPointAccel(COUPLER_LINK6, theta[1], omega[1],
                                    alpha[1]);
   Ap2[2] = wattI.couplerPointAccel(COUPLER_LINK6, theta[2], omega[2],
                                    alpha[2]);
   Ap2[3] = wattI.couplerPointAccel(COUPLER_LINK6, theta[3], omega[3],
                                    alpha[3]);
   Ap2[4] = wattI.couplerPointAccel(COUPLER_LINK6, theta[4], omega[4],
                                    alpha[4]);

   /* Display results. */
   for(i = 1; i <= 4; i++)
   {
      printf("Solution #%d:\n", i);
      printf("   P1 = %.3f, P2 = %.3f\n", P1[i], P2[i]);
      printf("   Vp1 = %.3f, Vp2 = %.3f\n", Vp1[i], Vp2[i]);
      printf("   Ap1 = %.3f, Ap2 = %.3f\n", Ap1[i], Ap2[i]);
   }

   return 0;
}
```

Program 27: Program for computing the positions, velocities, and accelerations of the two coupler points specified in Figure 5.7 (Contd.).

called multiple times to produce animations of the defined sixbar for the four possible geometric inversions. Tracing is also set for motion of the coupler point of link 6, but not for link 5. Figure 5.8 contains snapshots of the animations for each geometric inversion.

5.2.6 Web-Based Analysis

All the analysis methods discussed in the previous sections for the Watt (I) sixbar linkage can be performed with a couple of interactive web pages. Figure 5.9 shows the main web page for web-based analysis of the Watt (I) sixbar linkage. The web page shown in Figure 5.10 can be used to calculate the angular properties of links 3-6 of the Watt (I) sixbar. Also, the position, velocity, and acceleration values can be determined for the coupler point(s) associated with link(s) 5 and/or 6. For example, consider the Watt (I) sixbar linkage defined in previous problems. By entering the given values for the link lengths, θ_1, β_3 and β_4, coupler point properties, and angular properties of the input link, link 2, the output of running the web-based analysis is shown in Figure 5.11.

Simulating the motion of the Watt (I) sixbar mechanism can also be done using the internet. Figure 5.12 shows the web page designed for animating the Watt (I) sixbar. Again, using the parameters specified by previous problem statements, Figure 5.13 is a snapshot of the animation for the first geometric inversion of the Watt (I) sixbar linkage.

```
#include<stdio.h>
#include<sixbar.h>

int main()
{
   double r[1:6];
   double rP3, beta3,
          rPP4, beta4,
          rP5, beta5,
          rPP6, beta6;
   double theta[1:4][1:6];
   double theta1 = M_DEG2RAD(10), theta2 = M_DEG2RAD(25);
   double complex P1[1:4], P2[1:4];
   CWattSixbarI wattI;
   int i;

   /* Define Watt (I) Sixbar linkage. */
   r[1] = 0.08;  r[2] = 0.04;
   r[3] = 0.10;  r[4] = 0.07;
   r[5] = 0.07;   r[6] = 0.08;
   rP3 = 0.04;  beta3 = M_DEG2RAD(30);
   rPP4 = 0.09;  beta4 = M_DEG2RAD(50);
   rP5 = 0.06;  beta5 = M_DEG2RAD(30);
   rPP6 = 0.04;  beta6 = M_DEG2RAD(45);
   for(i = 1; i <= 4; i++)
   {
      theta[i][1] = theta1;
      theta[i][2] = theta2;
   }

   /* Perform analysis. */
   wattI.setLinks(r, theta1, rP3, beta3, rPP4, beta4);
   wattI.setCouplerPoint(COUPLER_LINK5, rP5, beta5, TRACE_OFF);
   wattI.setCouplerPoint(COUPLER_LINK6, rPP6, beta6, TRACE_ON);
   wattI.animation(1);
   wattI.animation(2);
   wattI.animation(3);
   wattI.animation(4);

   return 0;
}
```

Program 28: Program for animating the Watt (I) sixbar linkage.

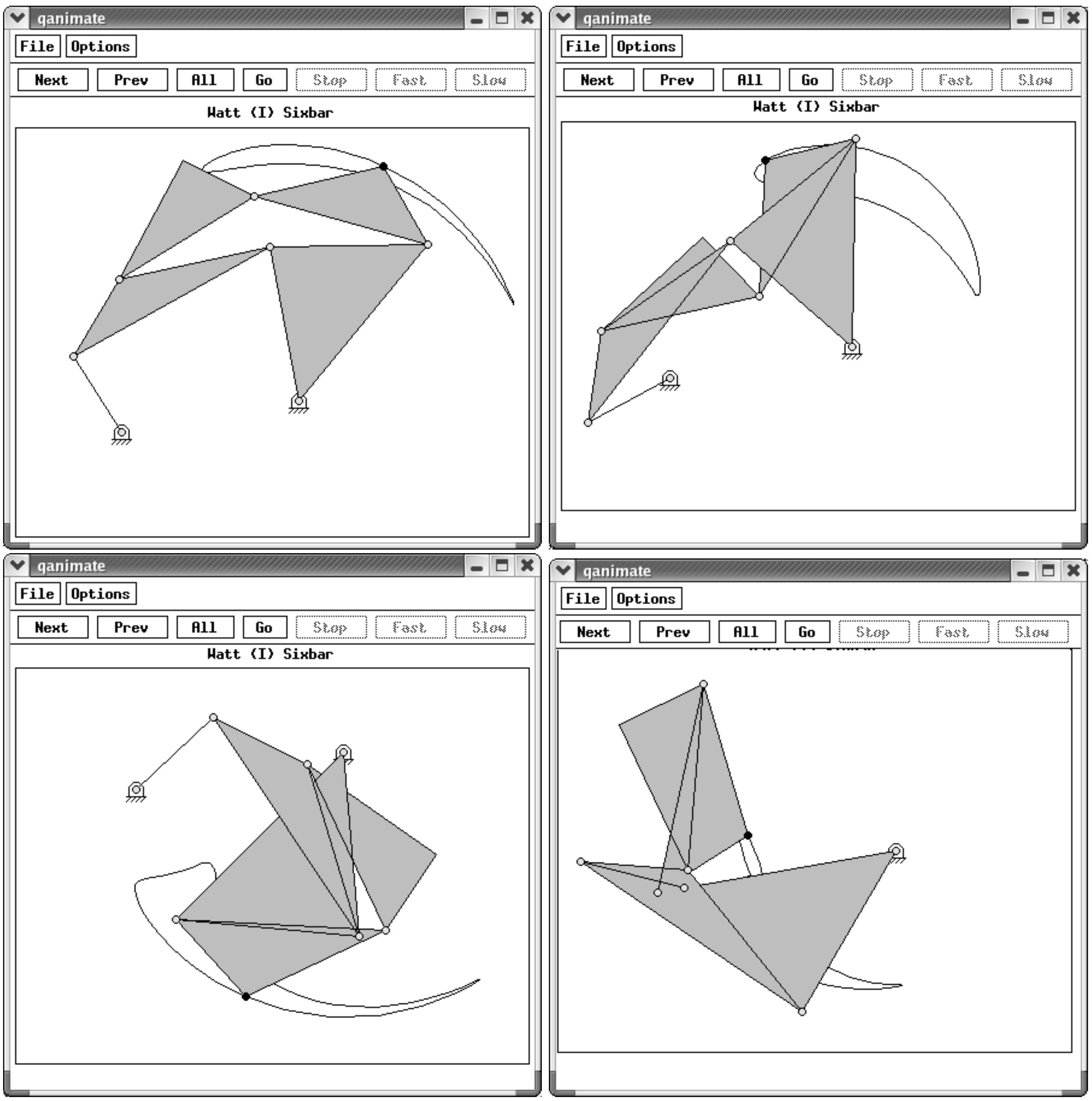


Figure 5.8: Output of Program 28.

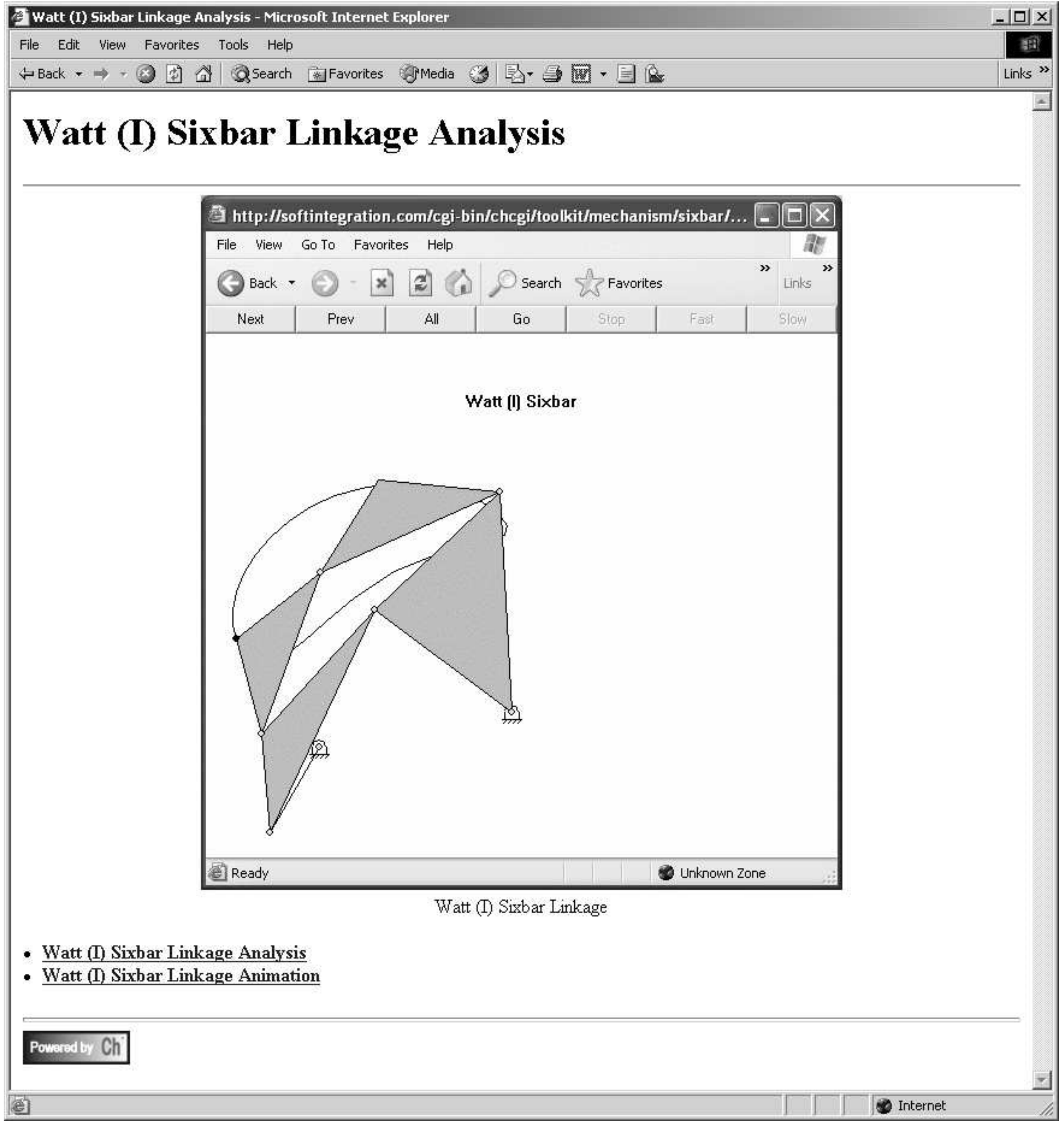


Figure 5.9: Main web page for Watt (I) sixbar analysis.

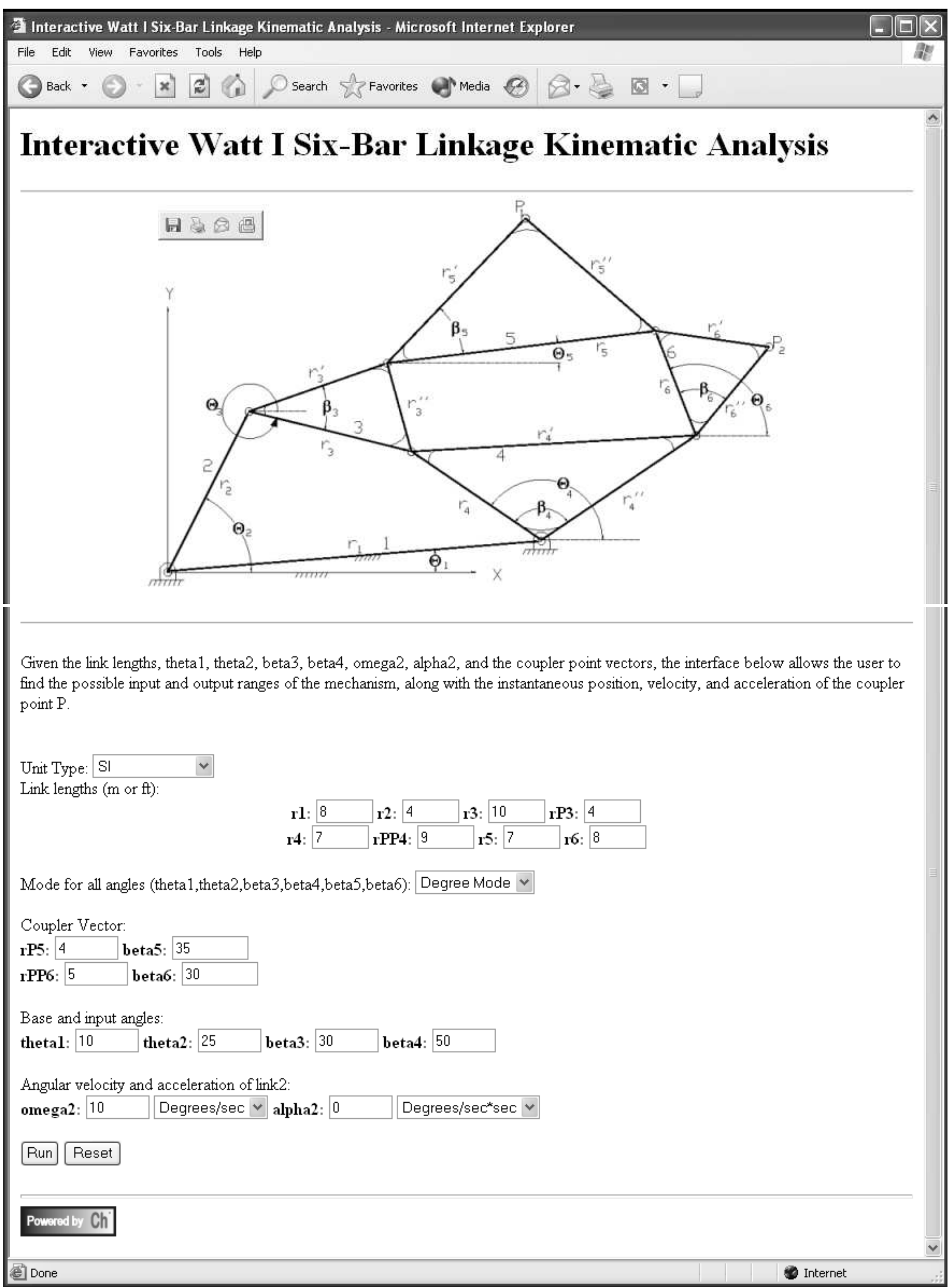


Figure 5.10: Web page for Watt (I) sixbar kinematic analysis.

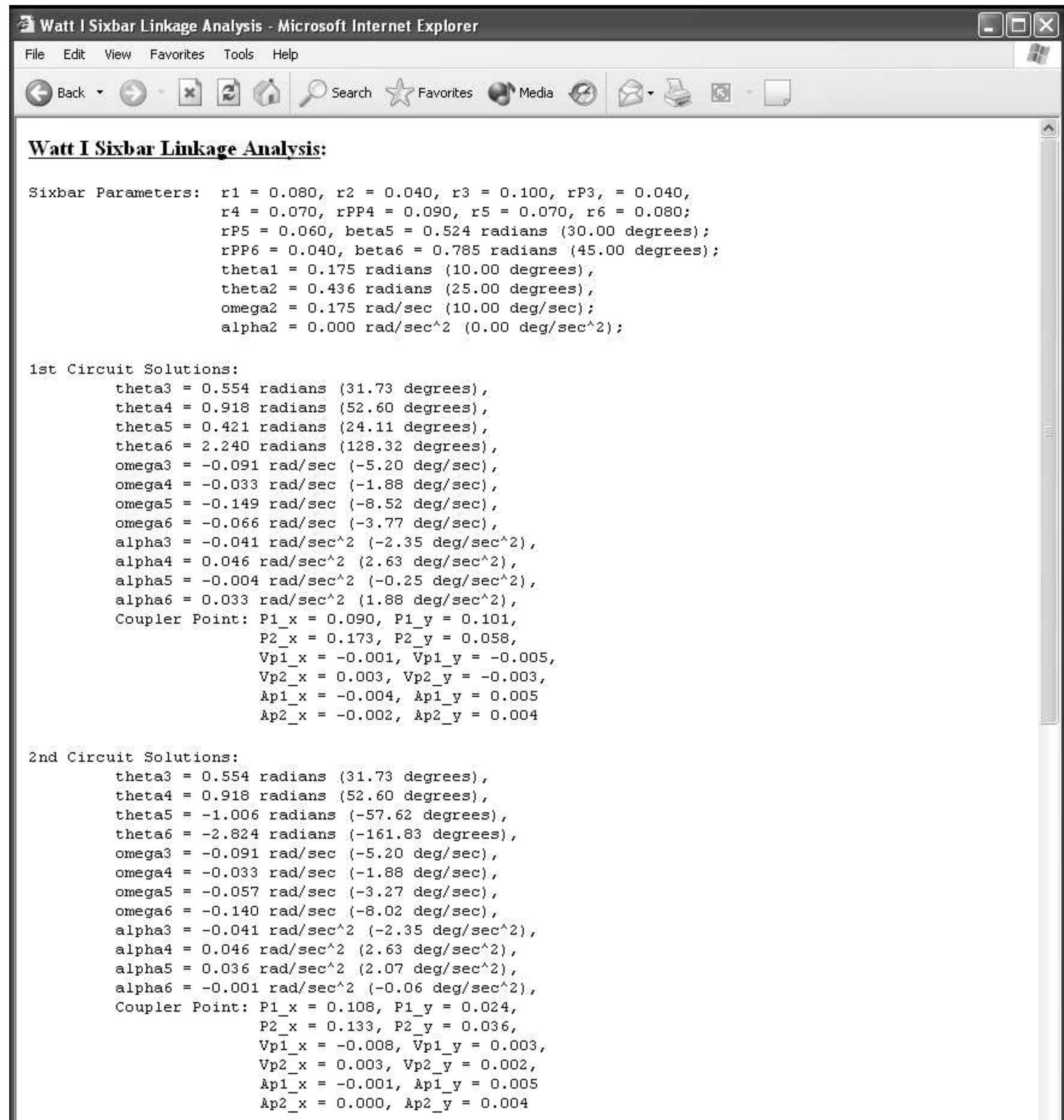


Figure 5.11: Output of web-based Watt (I) sixbar kinematic analysis.

```
3rd Circuit Solutions:
         theta3 = -0.695 radians (-39.83 degrees),
         theta4 = -1.059 radians (-60.70 degrees),
         theta5 = -0.848 radians (-48.59 degrees),
         theta6 = 0.356 radians (20.39 degrees),
         omega3 = -0.195 rad/sec (-11.20 deg/sec),
         omega4 = -0.253 rad/sec (-14.52 deg/sec),
         omega5 = -0.134 rad/sec (-7.68 deg/sec),
         omega6 = -0.184 rad/sec (-10.55 deg/sec),
         alpha3 = -0.172 rad/sec^2 (-9.86 deg/sec^2),
         alpha4 = -0.259 rad/sec^2 (-14.84 deg/sec^2),
         alpha5 = -0.035 rad/sec^2 (-2.01 deg/sec^2),
         alpha6 = -0.318 rad/sec^2 (-18.23 deg/sec^2),
         Coupler Point: P1_x = 0.133, P1_y = -0.009,
                        P2_x = 0.083, P2_y = -0.087,
                        Vp1_x = -0.013, Vp1_y = -0.003,
                        Vp2_x = -0.024, Vp2_y = 0.001,
                        Ap1_x = -0.009, Ap1_y = -0.000
                        Ap2_x = -0.026, Ap2_y = 0.003

4th Circuit Solutions:
         theta3 = -0.695 radians (-39.83 degrees),
         theta4 = -1.059 radians (-60.70 degrees),
         theta5 = -2.979 radians (-170.70 degrees),
         theta6 = 2.100 radians (120.32 degrees),
         omega3 = -0.195 rad/sec (-11.20 deg/sec),
         omega4 = -0.253 rad/sec (-14.52 deg/sec),
         omega5 = -0.194 rad/sec (-11.14 deg/sec),
         omega6 = -0.144 rad/sec (-8.27 deg/sec),
         alpha3 = -0.172 rad/sec^2 (-9.86 deg/sec^2),
         alpha4 = -0.259 rad/sec^2 (-14.84 deg/sec^2),
         alpha5 = -0.377 rad/sec^2 (-21.62 deg/sec^2),
         alpha6 = -0.094 rad/sec^2 (-5.39 deg/sec^2),
         Coupler Point: P1_x = 0.029, P1_y = -0.028,
                        P2_x = 0.057, P2_y = -0.032,
                        Vp1_x = -0.004, Vp1_y = 0.023,
                        Vp2_x = -0.016, Vp2_y = 0.007,
                        Ap1_x = 0.003, Ap1_y = 0.020
                        Ap2_x = -0.016, Ap2_y = 0.012
```

Done   Internet

Figure 5.11: Output of web-based Watt (I) sixbar kinematic analysis (Contd.).

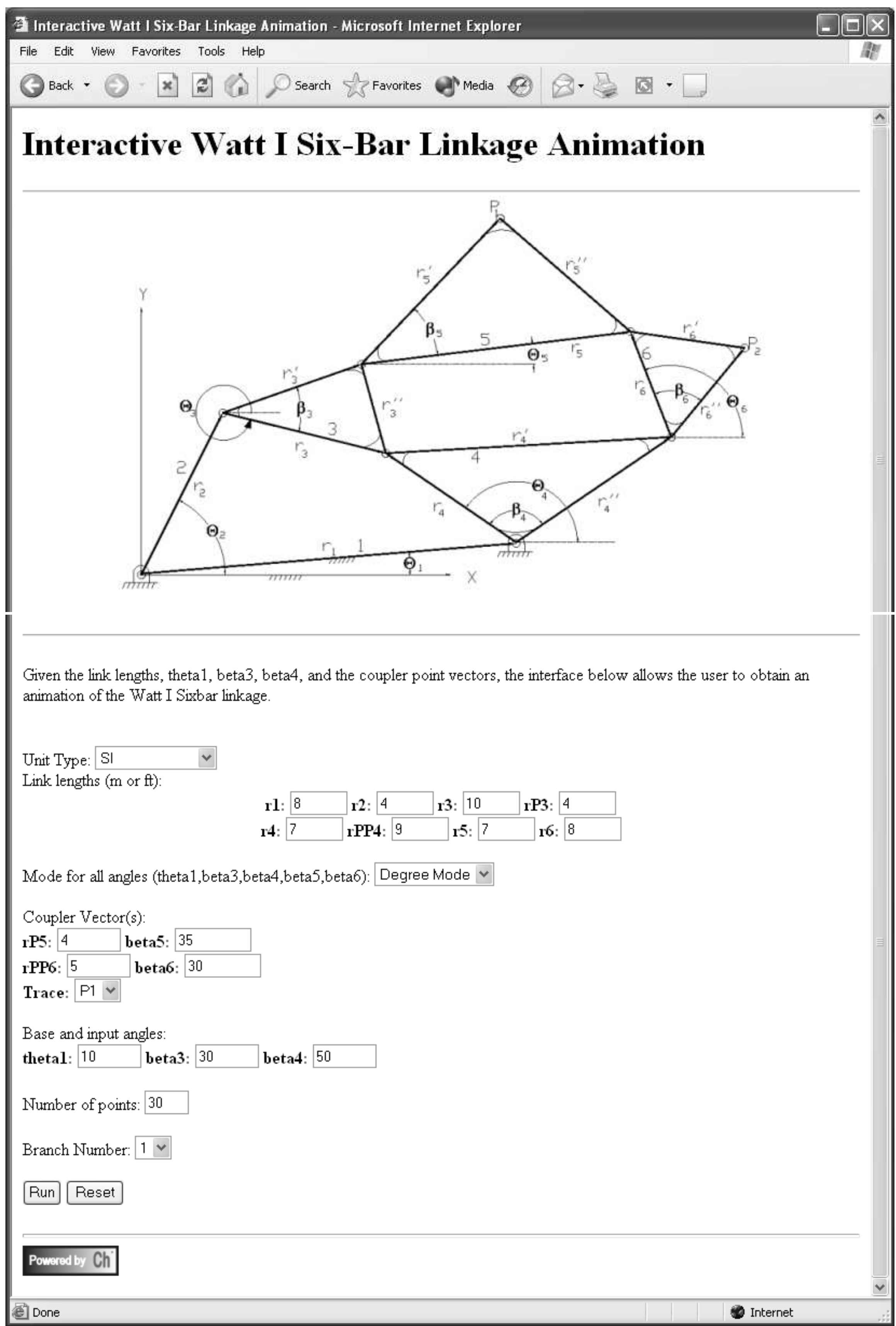


Figure 5.12: Web page for Watt (I) sixbar linkage animation.

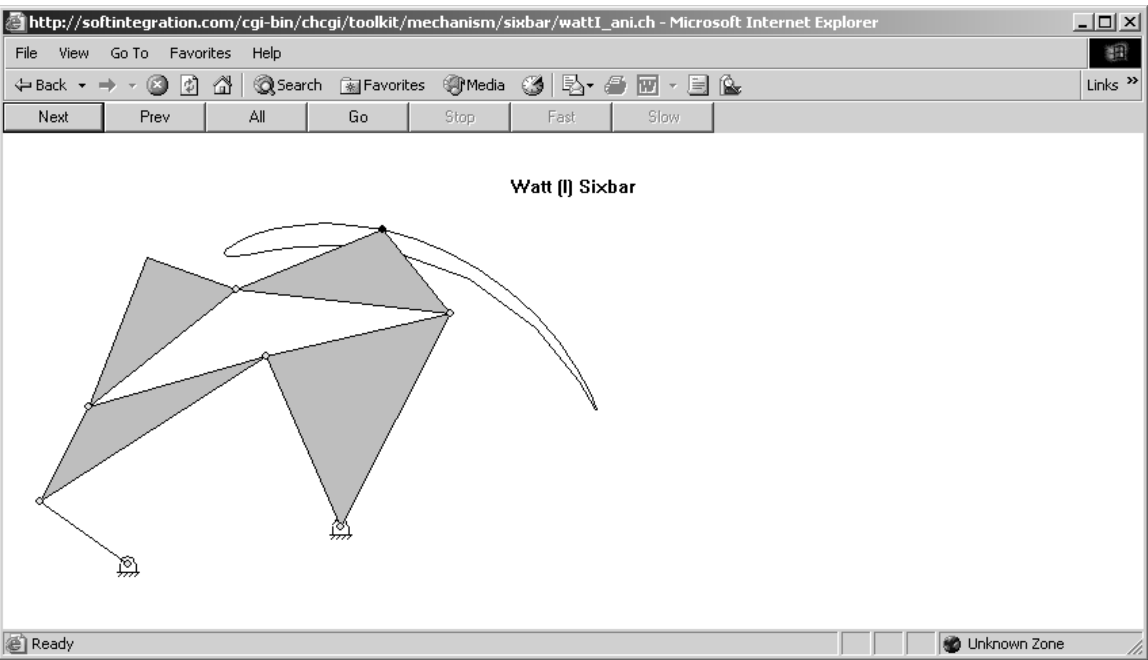


Figure 5.13: Snapshot of Watt (I) sixbar animation for branch 1.

5.3 Watt Six-bar (II) Linkage

A Watt (II) sixbar linkage is from two fourbar mechanisms where the output link of the first fourbar is also the input link of the second fourbar linkage. For the Watt (II) linkage in Figure 5.14, note that there is also a coupler point located on the floating link of the second fourbar linkage.

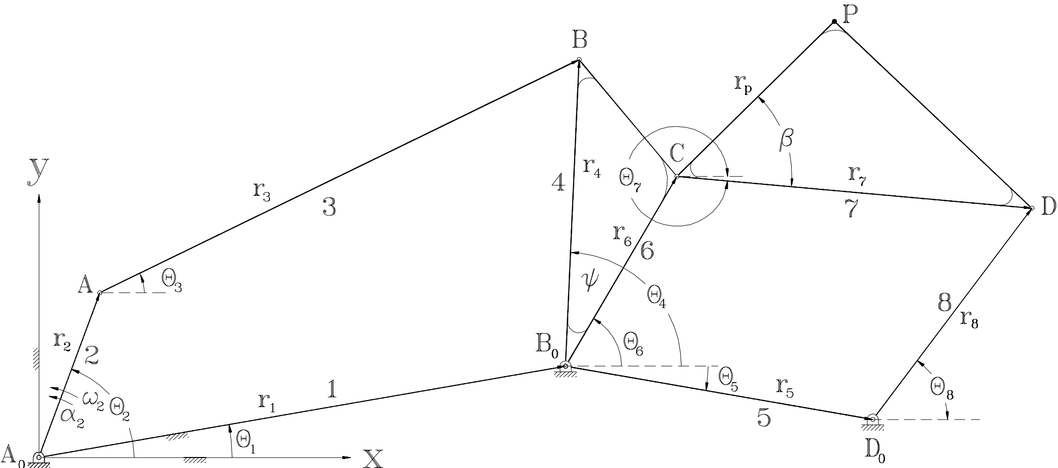


Figure 5.14: Watt (II) Sixbar Linkage

5.3.1 Position Analysis

Derivation of the angular positions of the links begin by applying geometry, complex arithmetic and vector analysis. Thus, the values θ_3, θ_4, θ_6, θ_7, and θ_8 can be found. First, the vector equation for the fourbar

section is written and reduced.

$$\mathbf{r}_1 + \mathbf{r}_4 = \mathbf{r}_2 + \mathbf{r}_3 \tag{5.82}$$

$$r_1 e^{i\theta_1} + r_4 e^{i\theta_4} = r_2 e^{i\theta_2} + r_3 e^{i\theta_3} \tag{5.83}$$

$$r_3 e^{i\theta_3} - r_4 e^{i\theta_4} = r_1 e^{i\theta_1} - r_2 e^{i\theta_2} \tag{5.84}$$

The latest equation has all the unknown quantities on the left-hand side, which can now be solved using the `complexsolve()` function. Two sets of values for θ_3 and θ_4 are found by `complexsolve()` where the inputs are $n1 = 2$, $n2 = 4$, $theta_o r_r 1 = r_3$, $theta_o r_r 2 = -r_4$, and $c3 = z = r_1 e^{i\theta_1} - r_2 e^{i\theta_2}$. Using each set of $\theta's$ separately, the angle θ_6 can be found by subtracting the angle ψ from θ_4. A similar set of vector equations is derived for the second fourbar section.

$$\theta_6 = \theta_4 - \psi \tag{5.85}$$

$$\mathbf{r}_5 + \mathbf{r}_8 = \mathbf{r}_6 + \mathbf{r}_7 \tag{5.86}$$

$$r_5 e^{i\theta_5} + r_8 e^{i\theta_8} = r_6 e^{i\theta_6} + r_7 e^{i\theta_7} \tag{5.87}$$

$$r_7 e^{i\theta_7} - r_8 e^{i\theta_8} = r_5 e^{i\theta_5} - r_6 e^{i\theta_6} \tag{5.88}$$

Again, the latest equation has all the unknown quantities on the left-hand side, which can now be solved using the `complexsolve` function. Two sets of values for θ_7 and θ_8 are found by `complexsolve()` where the inputs are $n1 = 2$, $n2 = 4$, $theta_o r_r 1 = r_7$, $theta_o r_r 2 = -r_8$, and $c3 = z = r_5 e^{i\theta_5} - r_6 e^{i\theta_6}$. Note that angular positions can also be determined by `angularPos()`, which is a member function of the Watt (II) sixbar class, `CWattSixbarII`. This class is similar to the class `CWattSixbarI`, but it specifically handles analysis for a Watt (II) sixbar.

With the given dimensions of the two fourbar linkages, it can be found that four different Watt (II) sixbar linkages can be assembled from them. This is possible because each fourbar linkage has two geometric inversions, and the output ranges of the first fourbar linkage intersect with the input ranges of the second with consideration to the included angle ψ.

5.3.2 Velocity Analysis

The values of the rotational velocities of the linkage can be found by taking the derivative of position vector equations of the two fourbar linkages. Taking the derivative of equations (5.84) and (5.88) gives the vector velocity equations (5.89) and (5.90).

$$i r_3 \omega_3 e^{i\theta_3} - i r_4 \omega_4 e^{i\theta_4} = -i r_2 \omega_2 e^{i\theta_2} \tag{5.89}$$

$$i r_7 \omega_7 e^{i\theta_7} - i r_8 \omega_8 e^{i\theta_8} = -i r_6 \omega_6 e^{i\theta_6} \tag{5.90}$$

To find the values for ω_3, ω_4, ω_7, and ω_8, the equations must be multiplied by an $e^{-i\theta_j}$ term in order to isolate a particular ω_j term. Note that $\omega_6 = \omega_4$. Multiplying equation (5.89) by $e^{-i\theta_4}$ and $e^{-i\theta_3}$ will generate equations (5.91) and (5.92). After factoring out the i term, the imaginary parts of equations 5.91 and 5.92 are separated in order to calculate the angular velocities (5.93) and (5.94). Rearranging these last two equations will give the values of ω_3 and ω_4 (5.95) and (5.96). The terms ω_7 and ω_8 are found in the same manner as ω_3 and ω_4. The difference is that the "3" and "4" subscripts are replaced by those of "7" and "8," respectively. The equations derived from equation (5.90) are equations (5.97) through (5.102). The angular velocity values for ω_3, ω_4, ω_7, and ω_8 are calculated in member function `angularVel()`.

Member function `angularVel()` can also be used to calculate the angular velocity values for ω_3, ω_4, ω_7, and ω_8 given the link lengths, their angular positions and angular velocity of the input link.

$$ir_3\omega_3e^{i(\theta_3-\theta_4)} - ir_4\omega_4 = -ir_2\omega_2e^{i(\theta_2-\theta_4)} \tag{5.91}$$

$$ir_3\omega_3 - ir_4\omega_4e^{i(\theta_4-\theta_3)} = -ir_2\omega_2e^{i(\theta_2-\theta_3)} \tag{5.92}$$

$$r_3\omega_3\sin(\theta_3-\theta_4) = -r_2\omega_2\sin(\theta_2-\theta_4) \tag{5.93}$$

$$r_4\omega_4\sin(\theta_4-\theta_3) = r_2\omega_2\sin(\theta_2-\theta_3) \tag{5.94}$$

$$\omega_3 = -\frac{r_2\omega_2\sin(\theta_4-\theta_2)}{r_3\sin(\theta_4-\theta_3)} \tag{5.95}$$

$$\omega_4 = \frac{r_2\omega_2\sin(\theta_3-\theta_2)}{r_4\sin(\theta_3-\theta_4)} \tag{5.96}$$

$$ir_7\omega_7e^{i(\theta_7-\theta_8)} - ir_8\omega_8 = -ir_6\omega_6e^{i(\theta_6-\theta_8)} \tag{5.97}$$

$$ir_7\omega_7 - ir_8\omega_8e^{i(\theta_8-\theta_7)} = -ir_6\omega_6e^{i(\theta_6-\theta_7)} \tag{5.98}$$

$$r_7\omega_7\sin(\theta_7-\theta_8) = -r_6\omega_6\sin(\theta_6-\theta_8) \tag{5.99}$$

$$r_8\omega_8\sin(\theta_8-\theta_7) = r_6\omega_6\sin(\theta_6-\theta_7) \tag{5.100}$$

$$\omega_7 = -\frac{r_6\omega_6\sin(\theta_8-\theta_6)}{r_7\sin(\theta_8-\theta_7)} \tag{5.101}$$

$$\omega_8 = \frac{r_6\omega_6\sin(\theta_7-\theta_6)}{r_8\sin(\theta_7-\theta_8)} \tag{5.102}$$

5.3.3 Acceleration Analysis

The acceleration relationships of the Watt (II) sixbar linkage can be found by performing the derivatives of equations (5.89) and (5.90). Similar to the velocity equations, these derivatives (5.103) and (5.104) are then multiplied by an $e^{-i\theta_j}$ term in order to isolate a particular α_j term. Note that $\alpha_6 = \alpha_4$. Multiplying equation (5.103) by $e^{-i\theta_4}$ and $e^{-i\theta_3}$ will generate equations (5.105) and (5.106). The real parts of equations (5.105) and (5.106) are separated in order to calculate the angular accelerations (5.107) and (5.108). Rearranging these last two equations will give the values of α_3 and α_4 (5.109) and (5.110). The values of α_7 and α_8 are found in the same manner by performing similar steps on equation (5.104) to get equations (5.111) through (5.116). Similar to the ω_i values, the angular acceleration values can be determined by member function `angularAccel()`.

$$(i\alpha_3-\omega_3^2)r_3e^{i\theta_3} - (i\alpha_4-\omega_4^2)r_4e^{i\theta_4} = (\omega_2^2-i\alpha_2)r_2e^{i\theta_2} \tag{5.103}$$

$$(i\alpha_7-\omega_7^2)r_7e^{i\theta_7} - (i\alpha_8-\omega_8^2)r_8e^{i\theta_8} = (\omega_6^2-i\alpha_6)r_6e^{i\theta_6} \tag{5.104}$$

$$ir_4\alpha_4 - r_4\omega_4^2 - ir_3\alpha_3e^{i(\theta_3-\theta_4)} + r_3\omega_3^2e^{i(\theta_3-\theta_4)} = ir_2\alpha_2e^{i(\theta_2-\theta_4)} - r_2\omega_2^2e^{i(\theta_2-\theta_4)} \tag{5.105}$$

$$ir_4\alpha_4e^{i(\theta_4-\theta_3)} - r_4\omega_4^2e^{i(\theta_4-\theta_3)} - ir_3\alpha_3 + r_3\omega_3^2 = ir_2\alpha_2e^{i(\theta_2-\theta_3)} - r_2\omega_2^2e^{i(\theta_2-\theta_3)} \tag{5.106}$$

$$r_3\alpha_3\sin(\theta_3-\theta_4) = -r_2\alpha_2\sin(\theta_2-\theta_4) - r_2\omega_2^2\cos(\theta_2-\theta_4) - r_3\omega_3^2\cos(\theta_3-\theta_4) + r_4\omega_4^2 \quad (5.107)$$

$$r_4\alpha_4\sin(\theta_4-\theta_3) = r_2\alpha_2\sin(\theta_2-\theta_3) + r_2\omega_2^2\cos(\theta_2-\theta_3) + r_3\omega_3^2 - r_4\omega_4^2\cos(\theta_4-\theta_3) \quad (5.108)$$

$$\alpha_3 = \frac{-r_2\omega_2^2\cos(\theta_2-\theta_4) - r_2\alpha_2\sin(\theta_2-\theta_4) - r_3\omega_3^2\cos(\theta_3-\theta_4) + r_4\omega_4^2}{r_3\sin(\theta_3-\theta_4)} \quad (5.109)$$

$$\alpha_4 = \frac{r_2\alpha_2\sin(\theta_2-\theta_3) + r_2\omega_2^2\cos(\theta_2-\theta_3) + r_3\omega_3^2 - r_4\omega_4^2\cos(\theta_4-\theta_3)}{r_4\sin(\theta_4-\theta_3)} \quad (5.110)$$

$$ir_8\alpha_8 - r_8\omega_8^2 - ir_7\alpha_7e^{i(\theta_7-\theta_8)} + r_7\omega_7^2e^{i(\theta_7-\theta_8)} = ir_6\alpha_6e^{i(\theta_6-\theta_8)} - r_6\omega_6^2e^{i(\theta_6-\theta_8)} \quad (5.111)$$

$$ir_8\alpha_8e^{i(\theta_8-\theta_7)} - r_8\omega_8^2e^{i(\theta_8-\theta_7)} - ir_7\alpha_7 + r_7\omega_7^2 = ir_6\alpha_6e^{i(\theta_6-\theta_7)} - r_6\omega_6^2e^{i(\theta_6-\theta_7)} \quad (5.112)$$

$$r_7\alpha_7\sin(\theta_7-\theta_8) = -r_6\alpha_6\sin(\theta_6-\theta_8) - r_6\omega_6^2\cos(\theta_6-\theta_8) - r_7\omega_7^2\cos(\theta_7-\theta_8) + r_8\omega_8^2 \quad (5.113)$$

$$r_8\alpha_8\sin(\theta_8-\theta_7) = r_6\alpha_6\sin(\theta_6-\theta_7) + r_6\omega_6^2\cos(\theta_6-\theta_7) + r_7\omega_7^2 - r_8\omega_8^2\cos(\theta_8-\theta_7) \quad (5.114)$$

$$\alpha_7 = \frac{-r_6\omega_6^2\cos(\theta_6-\theta_8) - r_6\alpha_6\sin(\theta_6-\theta_8) - r_7\omega_7^2\cos(\theta_7-\theta_8) + r_8\omega_8^2}{r_7\sin(\theta_7-\theta_8)} \quad (5.115)$$

$$\alpha_8 = \frac{r_6\alpha_6\sin(\theta_6-\theta_7) + r_6\omega_6^2\cos(\theta_6-\theta_7) + r_7\omega_7^2 - r_8\omega_8^2\cos(\theta_8-\theta_7)}{r_8\sin(\theta_8-\theta_7)} \quad (5.116)$$

Problem 1: A Watt (II) sixbar mechanism has the following parameters: $r_1 = 12cm, r_2 = 4cm, r_3 = 12cm, r_4 = 7cm, r_5 = 7cm, r_6 = 5cm, r_7 = 8cm, r_8 = 6cm$, $\theta_1 = 10^o$, $\theta_2 = 70^o$, $\theta_5 = -10^o$, $\psi = 30^o$, $\omega_2 = 10^o/sec$, and $\alpha_2 = 0$. Determine the angular accelerations of the various links for the sixbar linkage.

In order to calculate the angular acceleration values for the Watt (II) sixbar specified in the problem, each link's angular position and velocity has to be determined. This can be accomplished by utilizing member functions `angularPos()` and `angularVel()` of class `CWattSixbarII`. Then member function `angularAccel()` would be used to calculate α_3, α_4, α_6, α_7, and α_8. The solutions to the above problem are listed below. Program 29 was used to determine these values.

```
 Circuit 1: Angular Accelerations

alpha[3]= 0.007
alpha[4]= 0.012
alpha[6]= 0.012
alpha[7]= 0.001
alpha[8]= 0.010

 Circuit 2: Angular Accelerations

alpha[3]= 0.007
alpha[4]= 0.012
alpha[6]= 0.012
```

```
alpha[7]= 0.009
alpha[8]=-0.000

 Circuit 3: Angular Accelerations

alpha[3]= 0.019
alpha[4]= 0.014
alpha[6]= 0.014
alpha[7]= 0.009
alpha[8]=-0.003

 Circuit 4: Angular Accelerations

alpha[3]= 0.019
alpha[4]= 0.014
alpha[6]= 0.014
alpha[7]= 0.001
alpha[8]= 0.013
```

5.3.4 Coupler Point Position, Velocity, and Acceleration

The position of the coupler point can be determined by first writing its vector equation (5.117), which is then converted to polar form (5.118). Direct substitution of the link lengths and angles previously derived will give the position of the coupler point. The equations for the coupler point velocity and acceleration (5.119) and (5.120), respectively, are simply the first and second derivatives of equation (5.118). Exact numbers can be determined by substituting known and calculated values into the respective equations. Instead of direct calculation, the properties of the coupler point can be found by using member functions `couplerPointPos()`, `couplerPointVel()`, and `couplerPointAccel()`.

$$\mathbf{P} = \mathbf{r}_1 + \mathbf{r}_6 + \mathbf{r}_p \tag{5.117}$$

$$\mathbf{P} = r_1 e^{i\theta_1} + r_6 e^{i\theta_6} + r_p e^{i(\theta_7+\beta)} \tag{5.118}$$

$$\dot{\mathbf{P}} = i r_6 \omega_6 e^{i\theta_6} + i r_p \omega_7 e^{i(\theta_7+\beta)} \tag{5.119}$$

$$\ddot{\mathbf{P}} = i r_6 \alpha_6 e^{i\theta_6} - r_6 \omega_6^2 e^{i\theta_6} + i r_p \alpha_7 e^{i(\theta_7+\beta)} - r_p \omega_7^2 e^{i(\theta_7+\beta)} \tag{5.120}$$

For a better understanding of how class `CWattSixbarII` can be used to determine the coupler point position, velocity, and acceleration, refer to the following problem.

> **Problem 2:** For the Watt (II) sixbar in figure 5.14 with parameters: $r_1 = 12cm, r_2 = 4cm, r_3 = 12cm, r_4 = 7cm, r_5 = 7cm, r_6 = 5cm, r_7 = 8cm, r_8 = 6cm, r_p = 5cm$, $\theta_1 = 10^o$, $\theta_2 = 70^o$, $\theta_5 = -10^o$, $\omega_2 = 10^o/sec$, $\alpha_2 = 0$, $\beta = 45^o$, and $\psi = 30^o$, determine the position, velocity, and acceleration of the coupler point.

The solutions for this problem is show below. Program 30 was used to find these values.

```
P[1] = complex(0.184,0.095)
Vp[1] = complex(-0.004,0.002)
Ap[1] = complex(-0.001,0.000)
P[2] = complex(0.174,0.023)
Vp[2] = complex(-0.002,0.003)
Ap[2] = complex(-0.000,0.000)
```

```
#include <sixbar.h>

int main() {
    CWattSixbarII wattbar;
    double r[1:8], theta[1:8],theta_2[1:8],theta_3[1:8],theta_4[1:8];
    double omega[1:8],omega_2[1:8],omega_3[1:8],omega_4[1:8];
    double alpha[1:8],alpha_2[1:8],alpha_3[1:8],alpha_4[1:8];
    double theta1, theta5;
    double psi;
    int i;

    /* default specification of the four-bar linkage */
    r[1] = 0.12; r[2] = 0.04; r[3] = 0.12; r[4] = 0.07;
    r[5] = 0.07; r[6] = 0.05; r[7] = 0.08; r[8] = 0.06;
    theta1 = 10*M_PI/180;
    theta5 = -10*M_PI/180;
    theta[2]=70*M_PI/180;
    omega[2]=10*M_PI/180;  /*  rad/sec  */
    alpha[2]=0;            /*  rad/sec*sec  */
    psi = 30*M_PI/180;


    theta_2[2] = theta_3[2] = theta_4[2] = theta[2];
    omega_2[2] = omega_3[2] = omega_4[2] = omega[2];
    alpha_2[2] = alpha_3[2] = alpha_4[2] = alpha[2];
    wattbar.setLinks(r, theta1, theta5, psi);
    wattbar.angularPos(theta, theta_2,theta_3,theta_4);
    wattbar.angularVel(theta, omega);
    wattbar.angularAccel(theta, omega, alpha);
    printf("\n Circuit 1: Angular Accelerations\n\n");
    for(i=3; i<=8; i++) {
       if(i!=5)
           printf("alpha[%d]=%6.3f\n", i, alpha[i]);
    }
    wattbar.angularVel(theta_2, omega_2);
    wattbar.angularAccel(theta_2, omega_2, alpha_2);
    printf("\n Circuit 2: Angular Accelerations\n\n");
    for(i=3; i<=8; i++) {
       if(i!=5)
           printf("alpha[%d]=%6.3f\n", i, alpha_2[i]);
    }
    wattbar.angularVel(theta_3, omega_3);
    wattbar.angularAccel(theta_3, omega_3, alpha_3);
    printf("\n Circuit 3: Angular Accelerations\n\n");
    for(i=3; i<=8; i++) {
       if(i!=5)
           printf("alpha[%d]=%6.3f\n", i, alpha_3[i]);
    }
    wattbar.angularVel(theta_4, omega_4);
    wattbar.angularAccel(theta_4, omega_4, alpha_4);
    printf("\n Circuit 4: Angular Accelerations\n\n");
    for(i=3; i<=8; i++) {
       if(i!=5)
           printf("alpha[%d]=%6.3f\n", i, alpha_4[i]);
    }
}
```

Program 29: Program for computing the individual link's angular acceleration for the Watt (II) sixbar.

```
P[3] = complex(0.082,0.036)
Vp[3] = complex(-0.000,0.004)
Ap[3] = complex(0.000,-0.000)
P[4] = complex(0.128,0.006)
Vp[4] = complex(-0.003,0.003)
Ap[4] = complex(0.001,-0.000)
```

```
#include <sixbar.h>

int main() {
  CWattSixbarII wattbar;
  double r[1:8];
  double theta1 = 10*M_PI/180;
  double theta5 = -10*M_PI/180;
  double rp = 0.5, beta = 45*M_PI/180;
  double psi = 30*M_PI/180;
  double theta_1[1:8], theta_2[1:8], theta_3[1:8], theta_4[1:8];
  double omega_1[1:8], omega_2[1:8], omega_3[1:8], omega_4[1:8];
  double alpha_1[1:8], alpha_2[1:8], alpha_3[1:8], alpha_4[1:8];
  double theta2 = 70*M_PI/180;
  double complex p[1:4], vp[1:4], ap[1:4];
  int i;

  r[1] = 0.12, r[2] = 0.04, r[3] = 0.12, r[4] = 0.07;
  r[5] = 0.07,  r[6] = 0.05, r[7] = 0.08, r[8] = 0.06;
  omega_1[2] = 10*M_PI/180;  /*  rad/sec  */
  alpha_1[2] = 0;  /*  rad/sec*sec  */
  theta_1[1] = theta1;
  theta_1[2] = theta2; // theta2
  omega_2[2] = omega_3[2] = omega_4[2] = omega_1[2];
  alpha_2[2] = alpha_3[2] = alpha_4[2] = alpha_1[2];

  wattbar.setLinks(r, theta1, theta5, psi);
  wattbar.setCouplerPoint(COUPLER_LINK7, rp, beta);
  wattbar.angularPos(theta_1, theta_2, theta_3, theta_4);
  wattbar.couplerPointPos(COUPLER_LINK7, theta_1[2], p);

  /* first solution */
  wattbar.angularVel(theta_1, omega_1);
  wattbar.angularAccel(theta_1, omega_1, alpha_1);
  vp[1] = wattbar.couplerPointVel(COUPLER_LINK7, theta_1, omega_1);
  ap[1] = wattbar.couplerPointAccel(COUPLER_LINK7, theta_1, omega_1, alpha_1);

  /* second solution */
  wattbar.angularVel(theta_2, omega_2);
  wattbar.angularAccel(theta_2, omega_2, alpha_2);
  vp[2] = wattbar.couplerPointVel(COUPLER_LINK7, theta_2, omega_2);
  ap[2] = wattbar.couplerPointAccel(COUPLER_LINK7, theta_2, omega_2, alpha_2);

  /* third solution */
  wattbar.angularVel(theta_3, omega_3);
  wattbar.angularAccel(theta_3, omega_3, alpha_3);
  vp[3] = wattbar.couplerPointVel(COUPLER_LINK7, theta_3, omega_3);
  ap[3] = wattbar.couplerPointAccel(COUPLER_LINK7, theta_3, omega_3, alpha_3);

  /* fouth solution */
  wattbar.angularVel(theta_4, omega_4);
  wattbar.angularAccel(theta_4, omega_4, alpha_4);
  vp[4] = wattbar.couplerPointVel(COUPLER_LINK7, theta_4, omega_4);
  ap[4] = wattbar.couplerPointAccel(COUPLER_LINK7, theta_4, omega_4, alpha_4);
```

Program 30: Program for computing the coupler point properties using class `CWattSixbarII`.

```
  /* print solutions */
  for (i=1; i<=4; i++) {
    printf("P[%d] = %.3f\n", i, p[i]);
    printf("Vp[%d] = %.3f\n", i, vp[i]);
    printf("Ap[%d] = %.3f\n", i, ap[i]);
  }

  return 0;
}
```

Program 30: Program for computing the coupler point properties using class `CWattSixbarII` (Contd.).

5.3.5 Input/Output Ranges

Often times, it is desirable to know the input/output ranges of a mechanism. A mechanism's range of motion is usually part of the criteria of a design. To determine the possible input/output ranges of the Watt (II) sixbar, each fourbar portion of the linkage must be considered. The output range of the first fourbar linkage must intersect the input range of the second fourbar linkage with respect to the included angle ψ. These ranges are found by applying Grashof's Criteria of fourbar-linkage rotability. This is done by applying the member functions `grashof()` and `getJointLimits()` of class `CFourbar`. These functions will determine the fourbar linkage type and its consequent input and output ranges. Class `CFourbar` member function `getJointLimits()` will also calculate the input and output ranges of each geometric inversion should more than one linkage inversion exist.

The possible output range of the first fourbar linkage and the input range of the second linkage is then compared to find the intersection range. The comparison may run into a little trouble if negative angle values are compared to positive angle values. For comparison purposes only, all negative range values were made positive by adding 2π (1 revolution) to prevent any comparison error.

After finding the intersection ranges, the input ranges of the first fourbar linkage and the output ranges of the second fourbar linkage that corresponds to these intersection ranges can be found. The vector equations (5.82) and (5.86) can be solved to find these values. This is accomplished by inputting the necessary conditions into the function `complexsolve()`. This Watt (II) sixbar linkage has four different possible input/output ranges. Each input/output range corresponds to one of the possible geometric-inversion combinations of the two fourbar linkages. The input and output ranges of the Watt (II) sixbar linkage can also be determined by member function `getIORanges()`. Its function prototype is as follows,

```
int CWattSixbarII::getIORanges(double in[1:][:], end[1:][:]);
```

where `in` and `end` are two-dimensional arrays containing the range of motion for the input and output links of a Watt (II) sixbar linkage. As shown in Figure 5.14, the input and output links refer to links 2 and 8, respectively. Note that if the first fourbar portion is a Grashof Rocker-Rocker, there will exist eight possible input/output ranges.

> **Problem 3:** Determine all the possible input/output ranges of the Watt (II) sixbar defined in Problem 1.

If the input/output ranges of a Watt (II) sixbar is desired, such as in Problem 3, a Ch program like Program 31 can be used to find these values. After specifying the parameters for the sixbar linkage with memeber function `setLinks()`, member function `getIORanges()` is called to determine the input/output ranges of the linkage. For the Watt (II) sixbar defined in Problem 1, there are four possible input/output ranges, since there are four possible geometric inversions . Thus, variables `input` and `output` are defined as

```
#include <sixbar.h>

int main()
{
  double r[1:8], theta[1:8], psi, beta, rp;
  double input[1:4][2], output[1:4][2];
  int i, j, num_range;
  CWattSixbarII wattbar;

  /* specifications of the first four-bar linkage */
  r[1] = 0.12; r[2] = 0.04; r[3] = 0.12; r[4] = 0.07;
  theta[1] = 10*M_PI/180;

  /* specifications of the second four-bar linkage */
  r[5] = 0.07; r[6] = 0.05; r[7] = 0.08; r[8] = 0.06;
  theta[5] = -10*M_PI/180;

  /* adjoining angle */
  psi = 30*M_PI/180;

  /* input values */
  theta[2] = 70*M_PI/180;

  /* setup Watt (II) sixbar linkage. */
  wattbar.setLinks(r, theta[1], theta[5], psi);
  wattbar.getIORanges(r, theta, psi, input, output);

  return 0;
}
```

Program 31: Program for determining the input/output ranges of the Watt (II) sixbar defined in Problem 1.

4×2 arrays, where the first index refers to the four possible inversions, and the second index corresponds to the lower and upper range limits for the input/output links. For example, the input/output range for the first geometric inversion is $in[1][0] \leq \theta_2 \leq in[1][1]$ and $out[1][0] \leq \theta_8 \leq out[1][1]$. The results of this program is shown below.

```
 0.591 <= theta2 <=  3.911,  0.746 <= theta8 <=  1.808
 0.593 <= theta2 <=  3.911, -2.640 <= theta8 <= -2.873
 2.724 <= theta2 <=  6.040,  2.485 <= theta8 <=  2.430
 2.718 <= theta2 <=  6.040, -2.402 <= theta8 <= -1.405
```

5.3.6 Animation

Like the other classes for multiloop linkage analysis, class `CWattSixbarII` has member function `animation()` to simulate the motion of a Watt (II) sixbar mechanism. Its function prototype is the same as those that have already been discussed, which is shown below.

```
int CWattSixbarII::animation(int branchnum, ...);
```

Again, argument `branchnum` specifies the branch number of the Watt (II) sixbar linkage to animate. Program 32 is an example program that simulates the motion of the Watt (II) sixbar defined in previous problem statements. Snapshots of animation for the various branches are shown in Figure 5.15. Note that for

```
#include <sixbar.h>

int main() {
    /* default specification of the four-bar linkage */
    double r[1:8];
    r[1] = 0.12;  r[2] = 0.04;  r[3] = 0.12;  r[4] = 0.07;
    r[5] = 0.07;  r[6] = 0.05;  r[7] = 0.08;  r[8] = 0.06;
    double theta1 = 10*M_PI/180, theta5 = -10*M_PI/180;
    double psi = 30*M_PI/180;
    double rp = 0.05, beta = 45*M_PI/180;
    double omega2 = 10*M_PI/180;  /*  rad/sec  */
    double alpha2 = 0;  /*  rad/sec*sec  */
    int numpoints = 50;
    CWattSixbarII WattSixbar;

    WattSixbar.setLinks(r, theta1, theta5, psi);
    WattSixbar.setCouplerPoint(COUPLER_LINK7, rp, beta, TRACE_ON);
    WattSixbar.setNumPoints(numpoints);
    WattSixbar.animation(1);
    WattSixbar.animation(2);
    WattSixbar.animation(3);
    WattSixbar.animation(4);
}
```

Program 32: Program for animating the Watt (II) sixbar linkage.

Program 32, the first argument of member function `setCouplerPoint()` indicates the position of the coupler link, which is link 7 in this case.

5.3.7 Web-Based Analysis

Figure 5.16 shows the main web page for web-based analysis of the Watt (II) sixbar mechanism. The web pages for analyzing and animating the Watt (II) sixbar are shown in Figures 5.17 and 5.18, respectively. Based on the Watt (II) sixbar defined in the problem statement of Problem 2, the result of executing the analysis of Figure 5.17 is shown in Figure 5.19. The output of the Watt (II) linkage analysis web page contains values for the angular positions, velocities, and accelerations of all the links as well as values for the coupler point position, velocity, and acceleration. Note that the coupler link is assumed to be link 7, which corresponds to the Watt (II) sixbar figure shown on the web page.

For the animation web page shown in Figure 5.18, Figure 5.20 shows the output of running the web page to simulate the motion of Watt (II) sixbar linkage. The output is a snapshot of animation for the first geometric inversion of the defined mechanism.

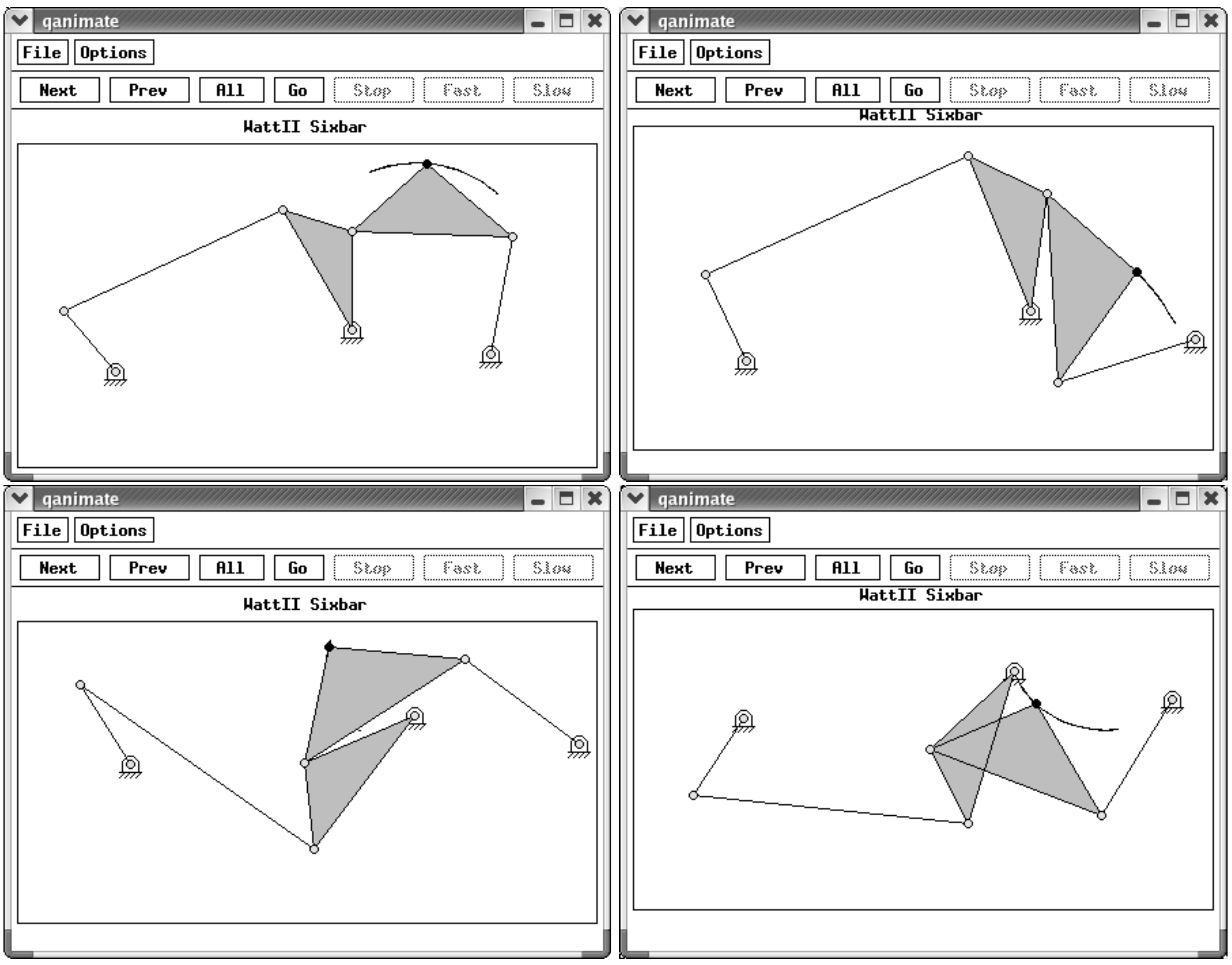


Figure 5.15: Output of Program 32.

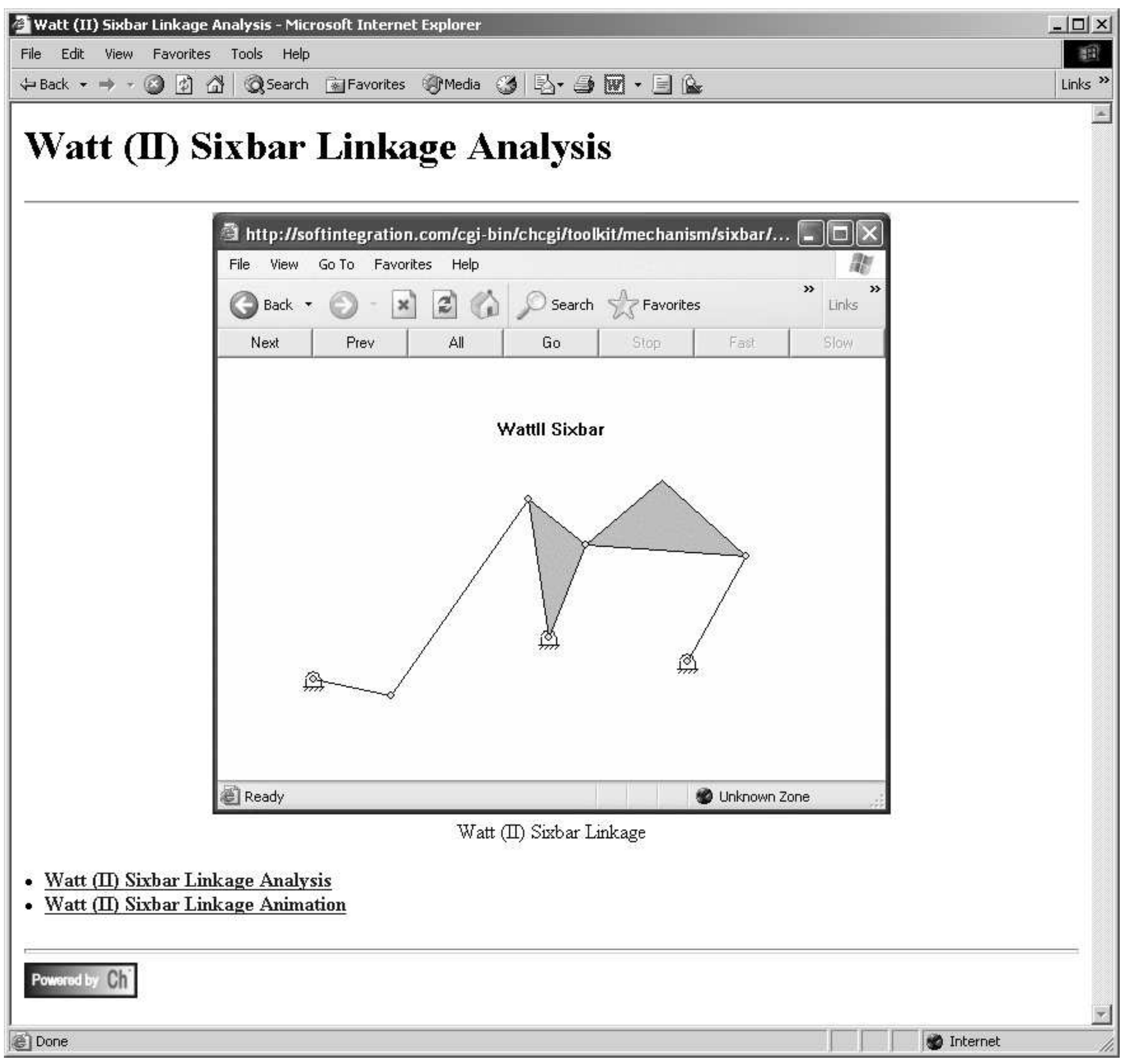


Figure 5.16: Main web page for the Watt (II) sixbar.

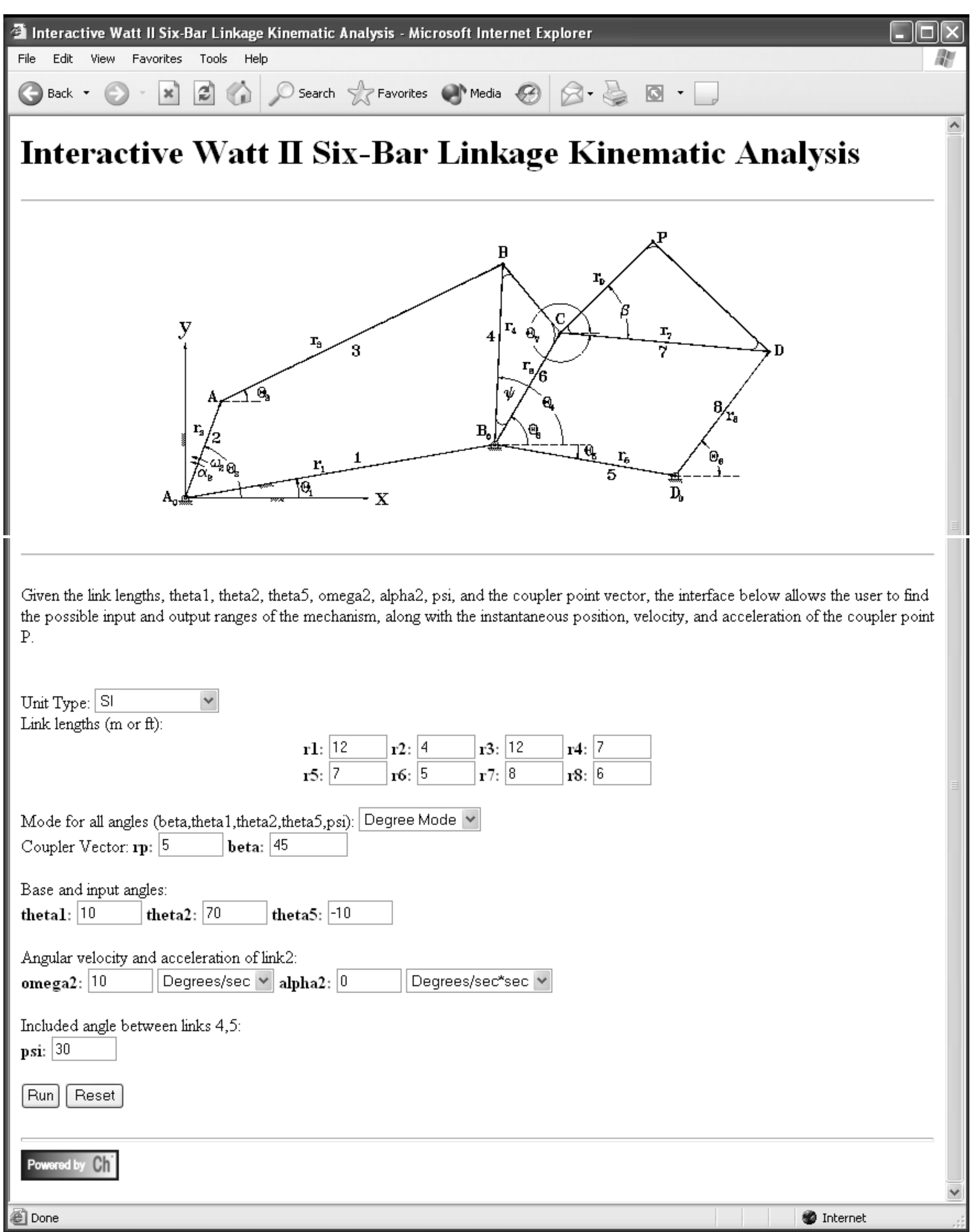


Figure 5.17: Web page for Watt (II) sixbar kinematic analysis.

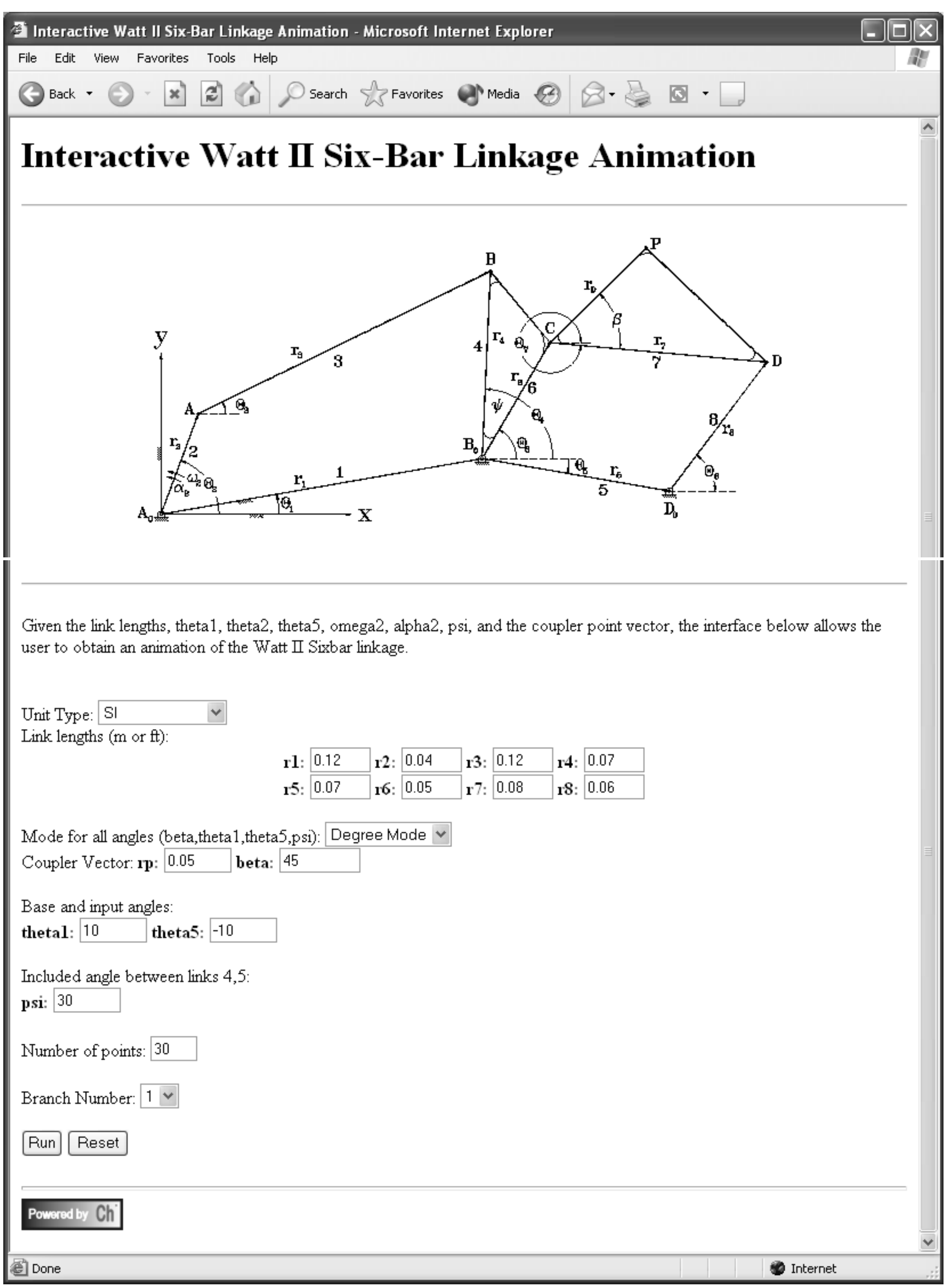


Figure 5.18: Web page for Watt (II) sixbar linkage animation.

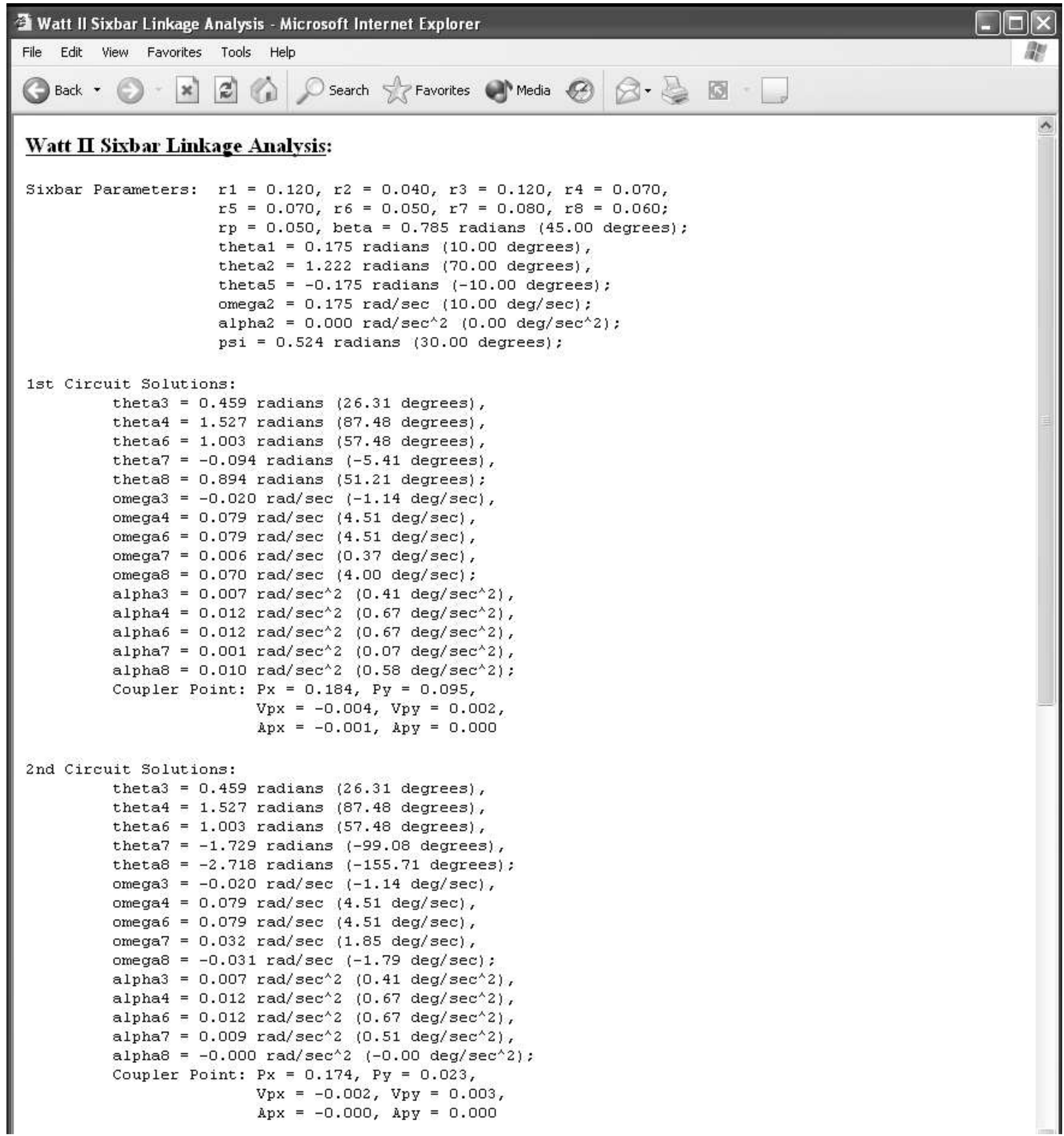


Figure 5.19: Output of web-based Watt (II) sixbar kinematic analysis.

```
3rd Circuit Solutions:
         theta3 = -0.777 radians (-44.52 degrees),
         theta4 = -1.845 radians (-105.70 degrees),
         theta6 = -2.368 radians (-135.70 degrees),
         theta7 = 0.800 radians (45.85 degrees),
         theta8 = 2.526 radians (144.74 degrees);
         omega3 = -0.005 rad/sec (-0.29 deg/sec),
         omega4 = -0.104 rad/sec (-5.93 deg/sec),
         omega6 = -0.104 rad/sec (-5.93 deg/sec),
         omega7 = -0.064 rad/sec (-3.69 deg/sec),
         omega8 = -0.002 rad/sec (-0.13 deg/sec);
         alpha3 = 0.019 rad/sec^2 (1.07 deg/sec^2),
         alpha4 = 0.014 rad/sec^2 (0.81 deg/sec^2),
         alpha6 = 0.014 rad/sec^2 (0.81 deg/sec^2),
         alpha7 = 0.009 rad/sec^2 (0.54 deg/sec^2),
         alpha8 = -0.003 rad/sec^2 (-0.18 deg/sec^2);
         Coupler Point: Px = 0.082, Py = 0.036,
                        Vpx = -0.000, Vpy = 0.004,
                        Apx = 0.000, Apy = -0.000

4th Circuit Solutions:
         theta3 = -0.777 radians (-44.52 degrees),
         theta4 = -1.845 radians (-105.70 degrees),
         theta6 = -2.368 radians (-135.70 degrees),
         theta7 = -0.372 radians (-21.32 degrees),
         theta8 = -2.098 radians (-120.21 degrees);
         omega3 = -0.005 rad/sec (-0.29 deg/sec),
         omega4 = -0.104 rad/sec (-5.93 deg/sec),
         omega6 = -0.104 rad/sec (-5.93 deg/sec),
         omega7 = -0.017 rad/sec (-1.00 deg/sec),
         omega8 = -0.080 rad/sec (-4.56 deg/sec);
         alpha3 = 0.019 rad/sec^2 (1.07 deg/sec^2),
         alpha4 = 0.014 rad/sec^2 (0.81 deg/sec^2),
         alpha6 = 0.014 rad/sec^2 (0.81 deg/sec^2),
         alpha7 = 0.001 rad/sec^2 (0.04 deg/sec^2),
         alpha8 = 0.013 rad/sec^2 (0.75 deg/sec^2);
         Coupler Point: Px = 0.128, Py = 0.006,
                        Vpx = -0.003, Vpy = 0.003,
                        Apx = 0.001, Apy = -0.000
```

Done   Internet

Figure 5.19: Output of web-based Watt (II) sixbar kinematic analysis (Contd.).

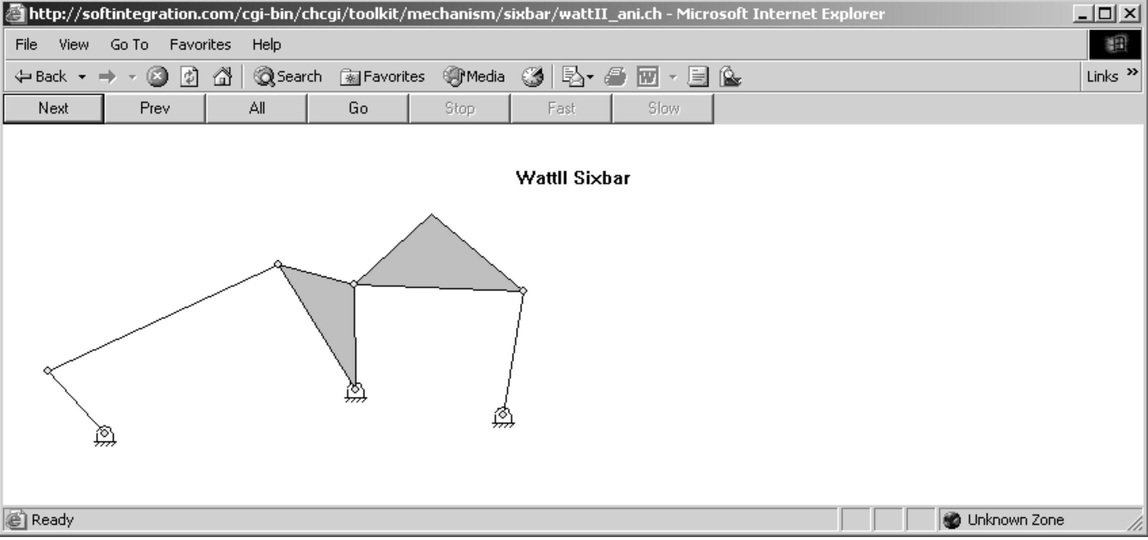


Figure 5.20: Snapshot of Watt (II) sixbar animation for branch 1.

5.4 Stephenson Six-bar (I) Linkage

A Stephenson (I) mechanism composed of a fourbar linkage and a restricted moving fivebar linkage is shown in Figure 5.21. The rigid-body input and output links of fourbar linkage are also the moving links of the fivebar linkage, thus the inputs and outputs of the two linkages are related.

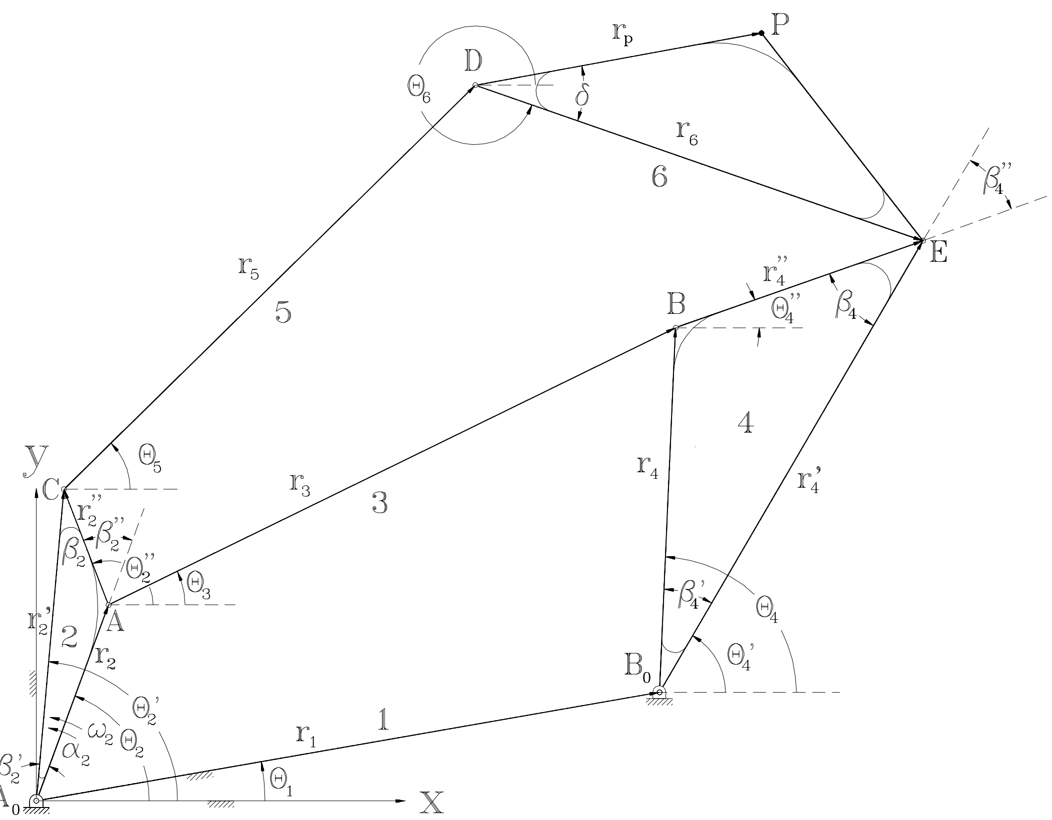


Figure 5.21: Stephenson (I) Sixbar Linkage

5.4.1 Position Analysis

By applying geometry, complex arithmetic and vector analysis, the angular positions θ_3, θ_4, θ_5, and θ_6 can be found. From vector analysis, we find that there are two sets of loop equations that will solve the system. Each of these sets of loop equations will generate the same answers. Both sets of loop equations will be derived in the discussion. The next step is to write and reduce the vector equations (5.121)-(5.123) of the fourbar linkage.

$$\mathbf{r}_1 + \mathbf{r}_4 = \mathbf{r}_2 + \mathbf{r}_3 \tag{5.121}$$

$$r_1 e^{i\theta_1} + r_4 e^{i\theta_4} = r_2 e^{i\theta_2} + r_3 e^{i\theta_3} \tag{5.122}$$

$$r_3 e^{i\theta_3} - r_4 e^{i\theta_4} = r_1 e^{i\theta_1} - r_2 e^{i\theta_2} \tag{5.123}$$

The last equation has all the unknown quantities on the left-hand side, which can now be solved using the `complexsolve` function. Two sets of values for θ_3 and θ_4 are found by `complexsolve()` where the inputs are $n1 = 2$, $n2 = 4$, $theta_o r_r 1 = r_3$, $theta_o r_r 2 = -r_4$, and $c3 = z = r_1 e^{i\theta_1} - r_2 e^{i\theta_2}$. The angles θ_2', θ_2'', θ_4', and θ_4'' can be found by subtracting the various unknown angles β, β', and β'' from the $\theta' s$.

$$\beta_2'' = \beta_2 + \beta_2' \quad (5.124)$$

$$\beta_4'' = \beta_4 \quad (5.125)$$

$$\theta_2' = \theta_2 + \beta_2' \quad (5.126)$$

$$\theta_4' = \theta_4 - \beta_4' \quad (5.127)$$

$$\theta_2'' = \theta_2 + \beta_2'' \quad (5.128)$$

$$\theta_4'' = \theta_4' - \beta_4'' \quad (5.129)$$

Now, we proceed to a second loop equation to find the values of θ_5 and θ_6. We have a choice of which fivebar linkage to use in this vector equation. The two possible equations are given in equations (5.130) and (5.131). The derivations of these two different equations to solve for the angles of links 5 and 6 are very similar, so we will proceed deriving the first one in its entirety (positions, velocities, and accelerations) for now and briefly go over the second later. Let's start by translating equation (5.130) into its polar form (5.132). After doing this, we identify the unknown terms and isolate them to one side of the equation (5.133). With the two unknowns on the left-hand side of the equation, the polar equation can be solved by the `complexsolve` function. The inputs for `complexsolve()` are: $n1 = 2$, $n2 = 4$, $theta_or_r1 = r_5$, $theta_or_r2 = r_6$, and $c3 = r_1e^{i\theta_1} - r_2'e^{i\theta_4'} + r_4'e^{i\theta_4'}$. There are two sets of values for θ_5 and θ_6 solved for each of the two θ_4's which totals four sets of values. The values of θ_5 and θ_6, along with θ_3 and θ_4 are solved by the member function `angularPos()` of the Stephenson (I) sixbar class, `CStevSixbarI`. This function incorporates the process mentioned above.

$$\mathbf{r}_2' + \mathbf{r}_5 + \mathbf{r}_6 = \mathbf{r}_1 + \mathbf{r}_4' \quad (5.130)$$

$$\mathbf{r}_2'' + \mathbf{r}_5 + \mathbf{r}_6 = \mathbf{r}_3 + \mathbf{r}_4'' \quad (5.131)$$

$$r_2'e^{i\theta_2'} + r_5e^{i\theta_5} + r_6e^{i\theta_6} = r_1e^{i\theta_1} + r_4'e^{i\theta_4'} \quad (5.132)$$

$$r_5e^{i\theta_5} + r_6e^{i\theta_6} = r_1e^{i\theta_1} - r_2'e^{i\theta_2'} + r_4'e^{i\theta_4'} \quad (5.133)$$

5.4.2 Velocity Analysis

The values of the angular velocities of the floating links 5 and 6 can be found through equations (5.123) and (5.133). Taking the derivative of equations (5.123) and (5.133) gives the vector velocity equations (5.134) and (5.135). Note that $\omega_2' = \omega_2$ and $\omega_4' = \omega_4$.

$$ir_3\omega_3e^{i\theta_3} - ir_4\omega_4e^{i\theta_4} = -ir_2\omega_2e^{i\theta_2} \quad (5.134)$$

$$ir_5\omega_5e^{i\theta_5} + ir_6\omega_6e^{i\theta_6} = -ir_2'\omega_2e^{i\theta_2'} + ir_4'\omega_4e^{i\theta_4'} \quad (5.135)$$

To find the values for ω_3, ω_4, ω_5, and ω_6, the equations must be multiplied by an $e^{-i\theta_j}$ term in order to isolate a particular ω_j term. Multiplying equation (5.134) by $e^{-i\theta_4}$ and $e^{-i\theta_3}$ separately will generate equations (5.136) and (5.137). These equations have the exponential product term eliminated from the ω_3 and ω_4 terms. The imaginary i term can be factored out of the equation also. After factoring out the i term, the imaginary parts of equations (5.136) and (5.137) are separated in order to calculate the angular velocities (5.138) and (5.139). Rearranging these last two equations will give the values of ω_3 and ω_4 (5.140) and (5.141). Knowing these values, we can then find ω_5 and ω_6. Again, we want to eliminate the exponential

term to isolate the $\omega' s$, so equation (5.135) is multiplied by $e^{-i\theta_6}$ and $e^{-i\theta_5}$ separately to generate equations (5.142) and (5.143). The terms ω_5 and ω_6 are found by isolating the imaginary parts of equations (5.142) and (5.143). Since all terms in equations (5.142) and (5.143) have a common imaginary term, this term can be factored out from both sides resulting in equations without the i term in the product. The imaginary parts are given in equations (5.144) and (5.145). Using these last two equations, we can isolate and solve for ω_5 and ω_6 (5.146) and (5.147).

$$ir_3\omega_3 e^{i(\theta_3-\theta_4)} - ir_4\omega_4 = -ir_2\omega_2 e^{i(\theta_2-\theta_4)} \tag{5.136}$$

$$ir_3\omega_3 - ir_4\omega_4 e^{i(\theta_4-\theta_3)} = -ir_2\omega_2 e^{i(\theta_2-\theta_3)} \tag{5.137}$$

$$r_3\omega_3 \sin(\theta_3-\theta_4) = -r_2\omega_2 \sin(\theta_2-\theta_4) \tag{5.138}$$

$$r_4\omega_4 \sin(\theta_4-\theta_3) = r_2\omega_2 \sin(\theta_2-\theta_3) \tag{5.139}$$

$$\omega_3 = -\frac{r_2\omega_2 \sin(\theta_4-\theta_2)}{r_3 \sin(\theta_4-\theta_3)} \tag{5.140}$$

$$\omega_4 = \frac{r_2\omega_2 \sin(\theta_3-\theta_2)}{r_4 \sin(\theta_3-\theta_4)} \tag{5.141}$$

$$ir_5\omega_5 e^{i(\theta_5-\theta_6)} + ir_6\omega_6 = -ir_2'\omega_2 e^{i(\theta_2'-\theta_6)} + ir_4'\omega_4 e^{i(\theta_4'-\theta_6)} \tag{5.142}$$

$$ir_5\omega_5 + ir_6\omega_6 e^{i(\theta_6-\theta_5)} = -ir_2'\omega_2 e^{i(\theta_2'-\theta_5)} + ir_4'\omega_4 e^{i(\theta_4'-\theta_5)} \tag{5.143}$$

$$r_5\omega_5 \sin(\theta_5-\theta_6) = -r_2'\omega_2 \sin(\theta_2'-\theta_6) + r_4'\omega_4 \sin(\theta_4'-\theta_6) \tag{5.144}$$

$$r_6\omega_6 \sin(\theta_6-\theta_5) = -r_2'\omega_2 \sin(\theta_2'-\theta_5) + r_4'\omega_4 \sin(\theta_4'-\theta_5) \tag{5.145}$$

$$\omega_5 = \frac{-r_2'\omega_2 \sin(\theta_2'-\theta_6) + r_4'\omega_4 \sin(\theta_4'-\theta_6)}{r_5 \sin(\theta_5-\theta_6)} \tag{5.146}$$

$$\omega_6 = \frac{-r_2'\omega_2 \sin(\theta_2'-\theta_5) + r_4'\omega_4 \sin(\theta_4'-\theta_5)}{r_6 \sin(\theta_6-\theta_5)} \tag{5.147}$$

The values for ω_3, ω_4, ω_5, and ω_6 are solved by member function `angularVel()`. Note that ω_5 and ω_6 are independent of ω_3 for this particular loop equation (5.130). However, this will not be the case if the second loop equation (5.131) was used instead.

5.4.3 Acceleration Analysis

The angular accelerations of the floating links are found by differentiating the angular velocity equations (5.134) and (5.135). Similar to the velocity equations, these derivatives (5.148) and (5.149) are then multiplied by an $e^{-i\theta_j}$ term in order to isolate a particular α_j term. Note that $\alpha_2' = \alpha_2$ and $\alpha_5' = \alpha_5$. Multiplying equation (5.148) by $e^{-i\theta_4}$ and $e^{-i\theta_3}$ will generate equations (5.150) and (5.151). The real parts of equations (5.150) and (5.151) are separated in order to calculate the angular accelerations (5.152) and (5.153). Rearranging these last two equations will give the values of α_3 and α_4 (5.154) and (5.155). To find the

angular accelerations of links 5 and 6, we multiply equation 5.149 by $e^{-i\theta_6}$ and $e^{-i\theta_5}$ separately (5.156) and (5.157). The terms α_5 and α_6 are then found by isolating the real parts of equations (5.156) and (5.157). The real parts are given in equations (5.158) and (5.159). These last two equations are rearranged to solve for α_5 and α_6 (5.160) and (5.161). The values for α_3, α_4, α_5, and α_6 are solved by member function `angularAccel()`. Similar to the angular velocities, note that α_5 and α_6 are independent of α_3 for this loop equation.

$$(i\alpha_3 - \omega_3^2)r_3e^{i\theta_3} - (i\alpha_4 - \omega_4^2)r_4e^{i\theta_4} = (\omega_2^2 - i\alpha_2)r_2e^{i\theta_2} \tag{5.148}$$

$$(i\alpha_5 - \omega_5^2)r_5e^{i\theta_5} + (i\alpha_6 - \omega_6^2)r_6e^{i\theta_6} = -(i\alpha_2 - \omega_2^2)r_2'e^{i\theta_2'} + (i\alpha_4 - \omega_4^2)r_4'e^{i\theta_4'} \tag{5.149}$$

$$ir_4\alpha_4 - r_4\omega_4^2 - ir_3\alpha_3e^{i(\theta_3-\theta_4)} + r_3\omega_3^2e^{i(\theta_3-\theta_4)} = ir_2\alpha_2e^{i(\theta_2-\theta_4)} - r_2\omega_2^2e^{i(\theta_2-\theta_4)} \tag{5.150}$$

$$ir_4\alpha_4e^{i(\theta_4-\theta_3)} - r_4\omega_4^2e^{i(\theta_4-\theta_3)} - ir_3\alpha_3 + r_3\omega_3^2 = ir_2\alpha_2e^{i(\theta_2-\theta_3)} - r_2\omega_2^2e^{i(\theta_2-\theta_3)} \tag{5.151}$$

$$\begin{aligned} r_3\alpha_3\sin(\theta_3-\theta_4) &= -r_2\alpha_2\sin(\theta_2-\theta_4) - r_2\omega_2^2\cos(\theta_2-\theta_4) - r_3\omega_3^2\cos(\theta_3-\theta_4) \\ &\quad + r_4\omega_4^2 \end{aligned} \tag{5.152}$$

$$r_4\alpha_4\sin(\theta_4-\theta_3) = r_2\alpha_2\sin(\theta_2-\theta_3) + r_2\omega_2^2\cos(\theta_2-\theta_3) + r_3\omega_3^2 - r_4\omega_4^2\cos(\theta_4-\theta_3) \tag{5.153}$$

$$\alpha_3 = \frac{-r_2\omega_2^2\cos(\theta_2-\theta_4) - r_2\alpha_2\sin(\theta_2-\theta_4) - r_3\omega_3^2\cos(\theta_3-\theta_4) + r_4\omega_4^2}{r_3\sin(\theta_3-\theta_4)} \tag{5.154}$$

$$\alpha_4 = \frac{r_2\alpha_2\sin(\theta_2-\theta_3) + r_2\omega_2^2\cos(\theta_2-\theta_3) + r_3\omega_3^2 - r_4\omega_4^2\cos(\theta_4-\theta_3)}{r_4\sin(\theta_4-\theta_3)} \tag{5.155}$$

$$\begin{aligned} (ir_5\alpha_5 - r_5\omega_5^2)e^{i(\theta_5-\theta_6)} + ir_6\alpha_6 - r_6\omega_6^2 &= -(ir_2'\alpha_2 - r_2'\omega_2^2)e^{i(\theta_2'-\theta_6)} \\ &\quad + (ir_4'\alpha_4 - r_4'\omega_4^2)e^{i(\theta_4'-\theta_6)} \end{aligned} \tag{5.156}$$

$$\begin{aligned} ir_5\alpha_5 - r_5\omega_5^2 + (ir_6\alpha_6 - r_6\omega_6^2)e^{i(\theta_6-\theta_5)} &= -(ir_2'\alpha_2 - r_2'\omega_2^2)e^{i(\theta_2'-\theta_5)} \\ &\quad + (ir_4'\alpha_4 - r_4'\omega_4^2)e^{i(\theta_4'-\theta_5)} \end{aligned} \tag{5.157}$$

$$\begin{aligned} -r_5\alpha_5\sin(\theta_5-\theta_6) - r_5\omega_5^2\cos(\theta_5-\theta_6) - r_6\omega_6^2 &= r_2'\alpha_2\sin(\theta_2'-\theta_6) + r_2'\omega_2^2\cos(\theta_2'-\theta_6) \\ &\quad - r_4'\alpha_4\sin(\theta_4'-\theta_6) \\ &\quad - r_4'\omega_4^2\cos(\theta_4'-\theta_6) \end{aligned} \tag{5.158}$$

$$\begin{aligned} -r_5\omega_5^2 - r_6\alpha_6\sin(\theta_6-\theta_5) - r_6\omega_6^2\cos(\theta_6-\theta_5) &= r_2'\alpha_2\sin(\theta_2'-\theta_5) + r_2'\omega_2^2\cos(\theta_2'-\theta_5) \\ &\quad - r_4'\alpha_4\sin(\theta_4'-\theta_5) \\ &\quad - r_4'\omega_4^2\cos(\theta_4'-\theta_5) \end{aligned} \tag{5.159}$$

$$\begin{aligned} \alpha_5 &= \frac{-r_2'\alpha_2\sin(\theta_2'-\theta_6) - r_2'\omega_2^2\cos(\theta_2'-\theta_6) + r_4'\alpha_4\sin(\theta_4'-\theta_6)}{r_5\sin(\theta_5-\theta_6)} \\ &\quad + \frac{r_4'\omega_4^2\cos(\theta_4'-\theta_6) - r_5\omega_5^2\cos(\theta_5-\theta_6) - r_6\omega_6^2}{r_5\sin(\theta_5-\theta_6)} \end{aligned} \tag{5.160}$$

$$\begin{aligned} \alpha_6 &= \frac{-r_2'\alpha_2\sin(\theta_2'-\theta_5) - r_2'\omega_2^2\cos(\theta_2'-\theta_5) + r_4'\alpha_4\sin(\theta_4'-\theta_5)}{r_6\sin(\theta_6-\theta_5)} \\ &\quad + \frac{r_4'\omega_4^2\cos(\theta_4'-\theta_5) - r_5\omega_5^2 - r_6\omega_6^2\cos(\theta_6-\theta_5)}{r_6\sin(\theta_6-\theta_5)} \end{aligned} \tag{5.161}$$

If we had chosen equation (5.131) as our second loop equation, the derivations for the link properties of links 5 and 6 would differ from those derived from equation (5.130) by having extra terms relating to link 3. These extra terms exist because link 3 is a moving link whereas link 1 in equation (5.130) is stationary. The velocity and acceleration terms for link 1 are equal to zero, so they drop out of the derivation. Also similar to the first loop equation, the terms ω_j'' and α_j'' in the second loop equation are equal to ω_j and α_j, respectively. Equations (5.162) and (5.163) are the position derivations of the second loop equation, equations (5.164) through (5.170) are the velocity derivations, and equations (5.171) through (5.177) are the acceleration derivations. The answers calculated by using the second loop equation would the same as those calculated using equation (5.130).

$$r_2''e^{i\theta_2''} + r_5e^{i\theta_5} + r_6e^{i\theta_6} = r_3e^{i\theta_3} + r_4''e^{i\theta_4''} \tag{5.162}$$

$$r_5e^{i\theta_5} + r_6e^{i\theta_6} = -r_2''e^{i\theta_2''} + r_3e^{i\theta_3} + r_4''e^{i\theta_4''} \tag{5.163}$$

$$ir_5\omega_5e^{i\theta_5} + ir_6\omega_6e^{i\theta_6} = -ir_2''\omega_2e^{i\theta_2''} + ir_3\omega_3e^{i\theta_3} + ir_4''\omega_4e^{i\theta_4''} \tag{5.164}$$

$$ir_5\omega_5e^{i(\theta_5-\theta_6)} + ir_6\omega_6 = -ir_2''\omega_2e^{i(\theta_2''-\theta_6)} + ir_3\omega_3e^{i(\theta_3-\theta_6)} + ir_4''\omega_4e^{i(\theta_4''-\theta_6)} \tag{5.165}$$

$$ir_5\omega_5 + ir_6\omega_6e^{i(\theta_6-\theta_5)} = -ir_2''\omega_2e^{i(\theta_2''-\theta_5)} + ir_3\omega_3e^{i(\theta_3-\theta_5)} + ir_4''\omega_4e^{i(\theta_4''-\theta_5)} \tag{5.166}$$

$$r_5\omega_5\sin(\theta_5-\theta_6) = -r_2''\omega_2\sin(\theta_2''-\theta_6) + r_3\omega_3\sin(\theta_3-\theta_6) + r_4''\omega_4\sin(\theta_4''-\theta_6) \tag{5.167}$$

$$r_6\omega_6\sin(\theta_6-\theta_5) = -r_2''\omega_2\sin(\theta_2''-\theta_5) + r_3\omega_3\sin(\theta_3-\theta_5) + r_4''\omega_4\sin(\theta_4''-\theta_5) \tag{5.168}$$

$$\omega_5 = \frac{-r_2''\omega_2\sin(\theta_2''-\theta_6) + r_3\omega_3\sin(\theta_3-\theta_6) + r_4''\omega_4\sin(\theta_4''-\theta_6)}{r_5\sin(\theta_5-\theta_6)} \tag{5.169}$$

$$\omega_6 = \frac{-r_2''\omega_2\sin(\theta_2''-\theta_5) + r_3\omega_3\sin(\theta_3-\theta_5) + r_4''\omega_4\sin(\theta_4''-\theta_5)}{r_6\sin(\theta_6-\theta_5)} \tag{5.170}$$

$$\begin{aligned}(i\alpha_5-\omega_5^2)r_5e^{i\theta_5} + (i\alpha_6-\omega_6^2)r_6e^{i\theta_6} &= -(i\alpha_2-\omega_2^2)r_2''e^{i\theta_2''} + (i\alpha_3-\omega_3^2)r_3e^{i\theta_3} \\ &\quad + (i\alpha_4-\omega_4^2)r_4''e^{i\theta_4''}\end{aligned} \tag{5.171}$$

$$\begin{aligned}(ir_5\alpha_5 - r_5\omega_5^2)e^{i(\theta_5-\theta_6)} + ir_6\alpha_6 - r_6\omega_6^2 &= -(ir_2''\alpha_2 - r_2''\omega_2^2)e^{i(\theta_2''-\theta_6)} + (ir_3\alpha_3 - r_3\omega_3^2)e^{i(\theta_3-\theta_6)} \\ &\quad + (ir_4''\alpha_4 - r_4''\omega_4^2)e^{i(\theta_4''-\theta_6)}\end{aligned} \tag{5.172}$$

$$\begin{aligned}ir_5\alpha_5 - r_5\omega_5^2 + (ir_6\alpha_6 - r_6\omega_6^2)e^{i(\theta_6-\theta_5)} &= -(ir_2''\alpha_2 - r_2''\omega_2^2)e^{i(\theta_2''-\theta_5)} + (ir_3\alpha_3 - r_3\omega_3^2)e^{i(\theta_3-\theta_5)} \\ &\quad + (ir_4''\alpha_4 - r_4''\omega_4^2)e^{i(\theta_4''-\theta_5)}\end{aligned} \tag{5.173}$$

$$\begin{aligned}-r_5\alpha_5\sin(\theta_5-\theta_6) - r_5\omega_5^2\cos(\theta_5-\theta_6) - r_6\omega_6^2 &= r_2''\alpha_2\sin(\theta_2''-\theta_6) + r_2''\omega_2^2\cos(\theta_2''-\theta_6) \\ &\quad - r_3\alpha_3\sin(\theta_3-\theta_6) - r_3\omega_3^2\cos(\theta_3-\theta_6) \\ &\quad - r_4''\alpha_4\sin(\theta_4''-\theta_6) \\ &\quad - r_4''\omega_4^2\cos(\theta_4''-\theta_6)\end{aligned} \tag{5.174}$$

$$\begin{aligned}-r_5\omega_5^2 - r_6\alpha_6\sin(\theta_6-\theta_5) - r_6\omega_6^2\cos(\theta_6-\theta_5) &= r_2''\alpha_2\sin(\theta_2''-\theta_5) + r_2''\omega_2^2\cos(\theta_2''-\theta_5) \\ &\quad - r_3\alpha_3\sin(\theta_3-\theta_5) - r_3\omega_3^2\cos(\theta_3-\theta_5) \\ &\quad - r_4''\alpha_4\sin(\theta_4''-\theta_5) \\ &\quad - r_4''\omega_4^2\cos(\theta_4''-\theta_5)\end{aligned} \tag{5.175}$$

$$\alpha_5 = \frac{-r_2''\alpha_2 \sin(\theta_2'' - \theta_6) - r_2''\omega_2^2 \cos(\theta_2'' - \theta_6) + r_3\alpha_3 \sin(\theta_3 - \theta_6) + r_3\omega_3^2 \cos(\theta_3 - \theta_6)}{r_5 \sin(\theta_5 - \theta_6)} + \frac{r_4''\alpha_4 \sin(\theta_4'' - \theta_6) + r_4''\omega_4^2 \cos(\theta_4'' - \theta_6) - r_5\omega_5^2 \cos(\theta_5 - \theta_6) - r_6\omega_6^2}{r_5 \sin(\theta_5 - \theta_6)} \quad (5.176)$$

$$\alpha_6 = \frac{-r_2''\alpha_2 \sin(\theta_2'' - \theta_5) - r_2''\omega_2^2 \cos(\theta_2'' - \theta_5) + r_3\alpha_3 \sin(\theta_3 - \theta_5) + r_3\omega_3^2 \cos(\theta_3 - \theta_5)}{r_6 \sin(\theta_6 - \theta_5)} + \frac{r_4''\alpha_4 \sin(\theta_4'' - \theta_5) + r_4''\omega_4^2 \cos(\theta_4'' - \theta_5) - r_5\omega_5^2 - r_6\omega_6^2 \cos(\theta_6 - \theta_5)}{r_6 \sin(\theta_6 - \theta_5)} \quad (5.177)$$

Problem 1: Consider a Stephenson (I) sixbar linkage with the following parameters: $r_1 = 12cm, r_2 = 4cm, r_2' = 6cm, r_3 = 12cm, r_4 = 7cm, r_4' = 10cm, r_5 = 11cm, r_6 = 9cm$, $\theta_1 = 10^o$, $\theta_2 = 70^o$, $\omega_2 = 10^o/sec$, $\alpha_2 = 0$, $\beta_2' = 15^o$, and $\beta_4' = 30^o$. Determine the angular acceleration of the various links of the sixbar.

The solution to the above problem is Program 33. This program utilizes class `CStevSixbarI` to solve the problem. The results are listed below. Note that the units for the angular accelerations are rad/sec^2.

```
Solution Set #1:
alpha3 = 0.007, alpha4 = 0.012,
alpha5 = 0.010, alpha6 = 0.015

Solution Set #2:
alpha3 = 0.007, alpha4 = 0.012,
alpha5 = 0.014, alpha6 = 0.009

Solution Set #3:
alpha3 = 0.019, alpha4 = 0.014,
alpha5 = 0.023, alpha6 = 0.028

Solution Set #4:
alpha3 = 0.019, alpha4 = 0.014,
alpha5 = 0.025, alpha6 = 0.020
```

```
#include<sixbar.h>

int main()
{
   double r[1:6];
   double theta[1:4][1:6], omega[1:4][1:6], alpha[1:4][1:6];
   double theta1, theta2, omega2, alpha2;
   double rp2, rp4, betaP2, betaP4, rp, delta;
   CStevSixbarI stevbar;
   int i;

   /* Specifications for the Stephenson I sixbar linkage. */
   r[1] = 0.12;  r[2] = 0.04;  r[3] = 0.12;  r[4] = 0.07;  r[5] = 0.11;  r[6] = 0.09;
   rp2 = 0.06; rp4 = 0.10;
   theta1 = 10*M_PI/180;  theta2 = 70*M_PI/180;
   betaP2 = 15*M_PI/180;  betaP4 = 30*M_PI/180;
   rp = 0.05;  delta = 30*M_PI/180;
   omega2 = 10*M_PI/180;                /* rad/sec */
   alpha2 = 0;                         /* rad/sec^2 */

   for(i = 1; i <= 4; i++)
   {
      theta[i][1] = theta1;  theta[i][2] = theta2;
      omega[i][2] = omega2;  alpha[i][2] = alpha2;
   }

   /* Perform analysis. */
   stevbar.setLinks(r, rp2, rp4, betaP2, betaP4, theta1);
   stevbar.angularPos(theta[1], theta[2], theta[3], theta[4]);
   stevbar.angularVel(theta[1], omega[1]);
   stevbar.angularVel(theta[2], omega[2]);
   stevbar.angularVel(theta[3], omega[3]);
   stevbar.angularVel(theta[4], omega[4]);
   stevbar.angularAccel(theta[1], omega[1], alpha[1]);
   stevbar.angularAccel(theta[2], omega[2], alpha[2]);
   stevbar.angularAccel(theta[3], omega[3], alpha[3]);
   stevbar.angularAccel(theta[4], omega[4], alpha[4]);

   /* Display the results. */
   for(i = 1; i <= 4; i++)
   {
      printf("Solution Set #%d:\n", i);
      printf("\talpha3 = %.3f, alpha4 = %.3f,\n", alpha[i][3], alpha[i][4]);
      printf("\talpha5 = %.3f, alpha6 = %.3f\n\n", alpha[i][5], alpha[i][6]);
   }

   return 0;
}
```

Program 33: Program for computing α_3, α_4, α_5, and α_6 using class `CStevSixbarI`.

5.4.4 Coupler Point Position, Velocity, and Acceleration

The vector equation for the coupler point, P, in Figure 5.21 is shown as equation (5.178). Rewriting this equation into polar form results in equation (5.179). Thus, given the required link lengths and θ's, the coupler point position can easily be determined. Likewise, one can find the coupler point velocity and acceleration by taking the derivatives of equation (5.179) and substituting the required values. The equations for the coupler point velocity and acceleration are listed as equations (5.180) and (5.181), respectively. Member functions `couplerPointPos()`, `couplerPointVel()`, and `couplerPointAccel()` utilizes equations (5.179)-(5.181) to determine the coupler point properties. Thus, one can use these member functions to determine the coupler point position, velocity, and accelerartion instead of the derived equations.

$$\mathbf{P} = \mathbf{r}'_2 + \mathbf{r}_5 + \mathbf{r}_p \tag{5.178}$$

$$\mathbf{P} = re^{i(\theta_2+\beta'_2)} + r_5 e^{i\theta_5} + r_p e^{i(\theta_6+\delta)} \tag{5.179}$$

$$\dot{\mathbf{P}} = ir'_2\omega_2 e^{i(\theta_2+\beta'_2)} + ir_5\omega_5 e^{i\theta_5} + ir_p\omega_6 e^{i(\theta_6+\delta)} \tag{5.180}$$

$$\ddot{\mathbf{P}} = ir'_2\alpha_2 e^{i(\theta_2+\beta'_2)} - r'_2\omega_2^2 e^{i(\theta_2+\beta'_2)} + ir_5\alpha_5 e^{i\theta_5} - r_5\omega_5^2 e^{i\theta_5} + ir_p\alpha_6 e^{i(\theta_6+\delta)} - r_p\omega_6^2 e^{i(\theta_6+\delta)} \tag{5.181}$$

As an example, consider the following problem.

> **Problem 2:** Given $r_p = 5cm$ and $\delta = 30^o$, determine the coupler point position, velocity, and acceleration of the Stephenson (I) sixbar shown in Figure 5.21.

The above problem can easily be solved if class `CStevSixbarI` is utilized. It contains the necessary member functions to calculate all the coupler point properties. Additionally, all the intermediate steps, such as calculating the links' angular positions, can be handled by this class. Below are the solutions for this problem, which was calculated by Program 34.

```
Solution Set #1:
 P = complex(0.122,0.086)
 Vp = complex(-0.007,0.006)
 Ap = complex(-0.001,-0.001)
Solution Set #2:
 P = complex(0.136,0.144)
 Vp = complex(-0.009,0.002)
 Ap = complex(-0.002,-0.000)
Solution Set #3:
 P = complex(0.003,-0.019)
 Vp = complex(-0.010,0.004)
 Ap = complex(0.001,-0.002)
Solution Set #4:
 P = complex(0.096,-0.031)
 Vp = complex(-0.007,0.008)
 Ap = complex(0.001,0.001)
```

5.4.5 Animation

Member function `animation()` of class `CStevSixbarI` with function prototype

```
int CStevSixbarI::animation(int branchnum, ...);
```

```
#include<sixbar.h>

int main()
{
   double r[1:6];
   double theta[1:4][1:6], omega[1:4][1:6], alpha[1:4][1:6];
   double theta1, theta2, omega2, alpha2;
   double rp2, rp4;
   double betaP2, betaP4, rp, delta;
   double complex P[1:4], Vp[1:4], Ap[1:4];
   CStevSixbarI stevbar;
   int i;

   /* Specifications for the Stephenson I sixbar linkage. */
   r[1] = 0.12;  r[2] = 0.04;  r[3] = 0.12;  r[4] = 0.07;
   r[5] = 0.11;  r[6] = 0.09;
   rp2 = 0.06; rp4 = 0.10;
   theta1 = 10*M_PI/180;  theta2 = 70*M_PI/180;
   betaP2 = 15*M_PI/180;  betaP4 = 30*M_PI/180;
   rp = 0.05;  delta = 30*M_PI/180;
   omega2 = 10*M_PI/180;                /* rad/sec */
   alpha2 = 0;                         /* rad/sec^2 */

   for(i = 1; i <= 4; i++)
   {
      theta[i][1] = theta1;  theta[i][2] = theta2;
      omega[i][2] = omega2;  alpha[i][2] = alpha2;
   }

   /* Perform analysis. */
   stevbar.setLinks(r, rp2, rp4, betaP2, betaP4, theta1);
   stevbar.setCouplerPoint(COUPLER_LINK6, rp, delta);
   stevbar.angularPos(theta[1], theta[2], theta[3], theta[4]);
   stevbar.angularVel(theta[1], omega[1]);
   stevbar.angularVel(theta[2], omega[2]);
   stevbar.angularVel(theta[3], omega[3]);
   stevbar.angularVel(theta[4], omega[4]);
   stevbar.angularAccel(theta[1], omega[1], alpha[1]);
   stevbar.angularAccel(theta[2], omega[2], alpha[2]);
   stevbar.angularAccel(theta[3], omega[3], alpha[3]);
   stevbar.angularAccel(theta[4], omega[4], alpha[4]);
```

Program 34: Program for computing the position, velocity, and acceleration of the coupler point using class `CStevSixbarI`.

```
    /* Determine coupler point properties. */
    stevbar.couplerPointPos(COUPLER_LINK6, theta2, P);
    for(i = 1; i <= 4; i++)
    {
       Vp[i] = stevbar.couplerPointVel(COUPLER_LINK6, theta[i], omega[i]);
       Ap[i] = stevbar.couplerPointAccel(COUPLER_LINK6, theta[i], omega[i],
                                          alpha[i]);
    }

    /* Display the results. */
    for(i = 1; i <= 4; i++)
    {
       printf("Solution Set #%d:\n", i);
       printf("\t P = %.3f\n", P[i]);
       printf("\t Vp = %.3f\n", Vp[i]);
       printf("\t Ap = %.3f\n", Ap[i]);
    }

    return 0;
}
```

Program 34: Program for computing the position, velocity, and acceleration of the coupler point using class `CStevSixbarI` (Contd.).

can be used to simulate the motion of a Stephenson (I) sixbar linkage. For the Stephenson (I) sixbar mechanism defined in earlier problem statements, with the coupler link being link 6, Program 35 can be used to simulate its motion. Snapshots of animation for the four possible geometric inversions are shown in Figure 5.22. Note that if the fourbar portion of the sixbar linkage was a Grashof Rocker-Rocker, the possible number of geometric inversions would be 8, not 4.

5.4.6 Web-Based Analysis

Analysis of the Stephenson (I) sixbar linkage can also be performed through the world wide web. Figure 5.23 is the main web page for analyzing the Stephenson (I) sixbar. Figure 5.24 shows the internet web page used for kinematic analysis of the sixbar mechanism discussed in this section. Figure 5.25 shows the result of using the web page of Figure 5.24 to analyze the Stephenson (I) sixbar linkage defined in the problems above.

For animating the Stephenson (I) sixbar linkage, the web page shown in Figure 5.26 can be used. Similar to the other web pages for simulating the motion of linkage mechanisms, this page requires of the input of parameters to define the Stephenson (I) sixbar. The number of animation frames to generate as well as the branch number may also be specified. Again, using the parameters defined in previous examples, Figure 5.27 shows an instant of animation for one branch of the Stephenson (I) sixbar mechanism.

```
#include <sixbar.h>

int main() {
    /* default specification of the StevesonI linkage */
    double r[1:6];
    r[1] = 0.12;  r[2] = 0.04;  r[3] = 0.12;
    r[4] = 0.07;  r[5] = 0.11;  r[6] = 0.09;
    double theta1 = 10*M_PI/180;
    double rp2 = 0.06, rp4 = 0.10;
    double betaP2 = 15*M_PI/180, betaP4 = 30*M_PI/180;
    double rp = 0.05, delta = 30*M_PI/180;
    double omega2=10*M_PI/180;  /*  rad/sec  */
    double alpha2=0;            /*  rad/sec*sec  */
    int numpoints =50;
    CStevSixbarI StevSixbar;

    StevSixbar.setLinks(r, rp2, rp4, betaP2, betaP4, theta1);
    StevSixbar.setCouplerPoint(COUPLER_LINK6, rp, delta, TRACE_ON);
    StevSixbar.setNumPoints(numpoints);
    StevSixbar.animation(1);
    StevSixbar.animation(2);
    StevSixbar.animation(3);
    StevSixbar.animation(4);
}
```

Program 35: Program for animating the Stephenson (I) sixbar linkage.

5.5 Stephenson Six-bar (III) Linkage

A Stephenson (III) mechanism is composed of a normal fourbar linkage and a restricted moving fourbar as shown in Figure 5.28. Similar to a Watt (II) sixbar, the output link of the first fourbar linkage, r_4, becomes the input link of the second fourbar linkage. The limited motion of the fourth link results in the restrictive motion of the second fourbar.

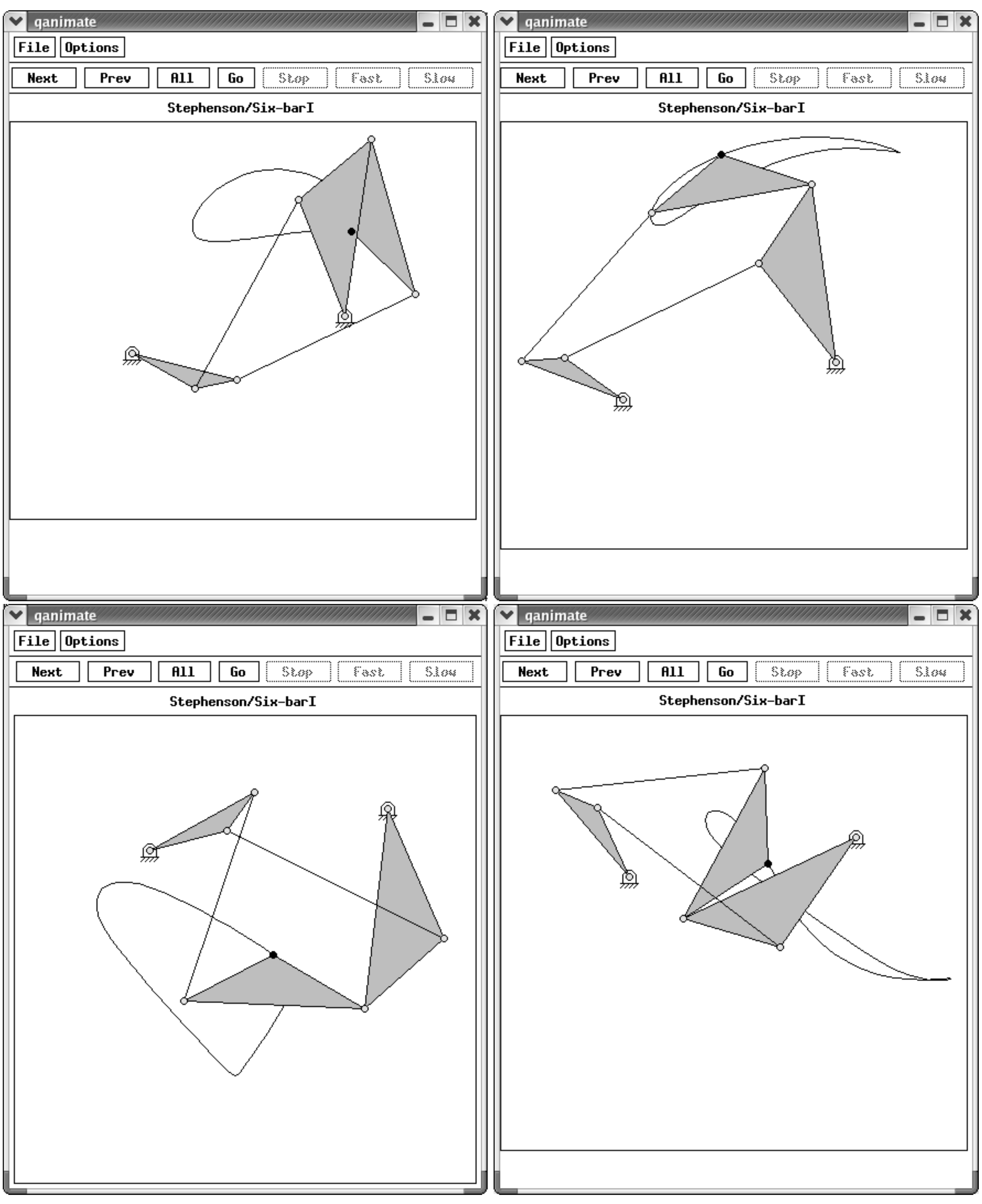


Figure 5.22: Output of Program 35.

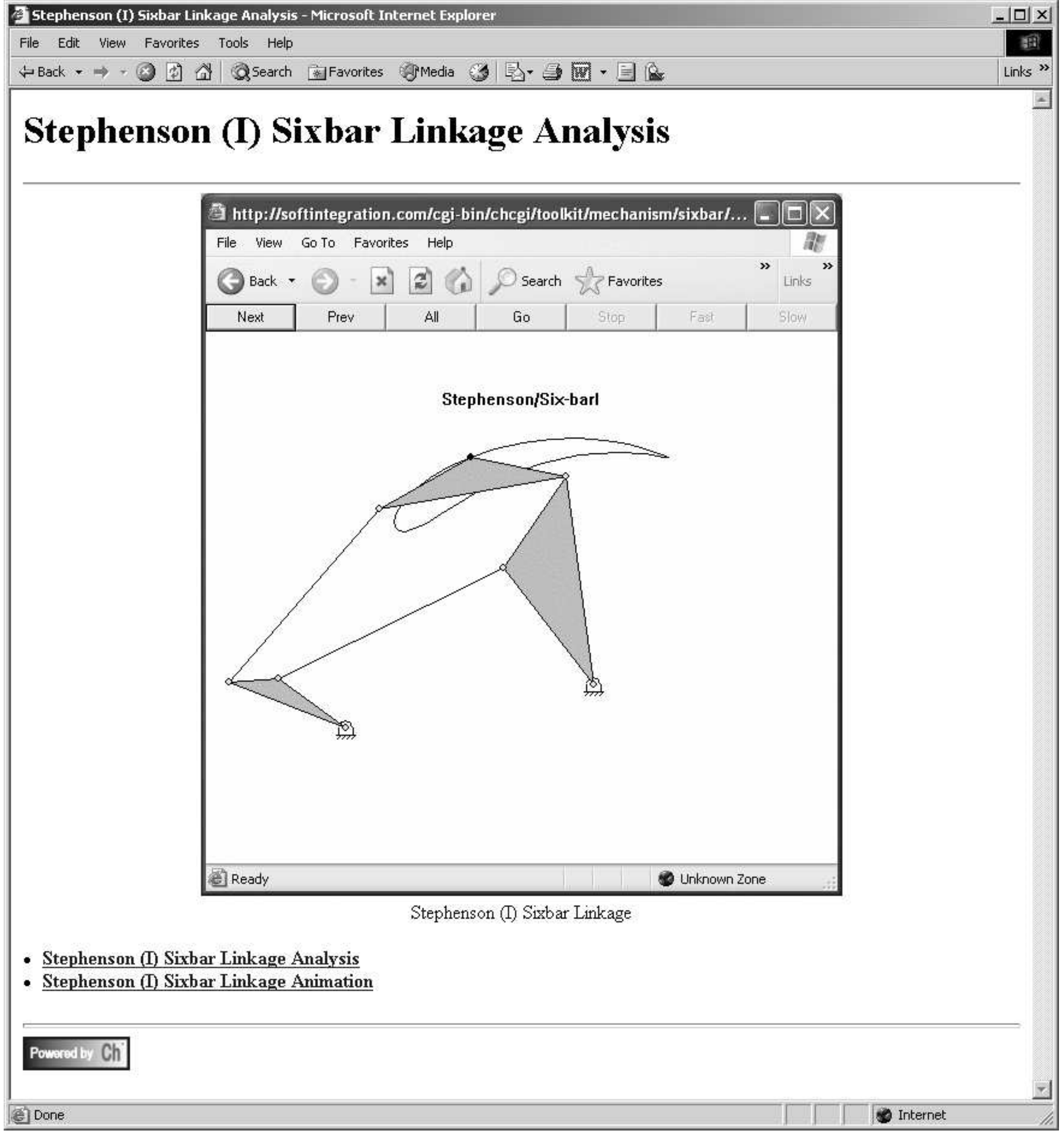


Figure 5.23: Main web page for the Stephenson (I) sixbar.

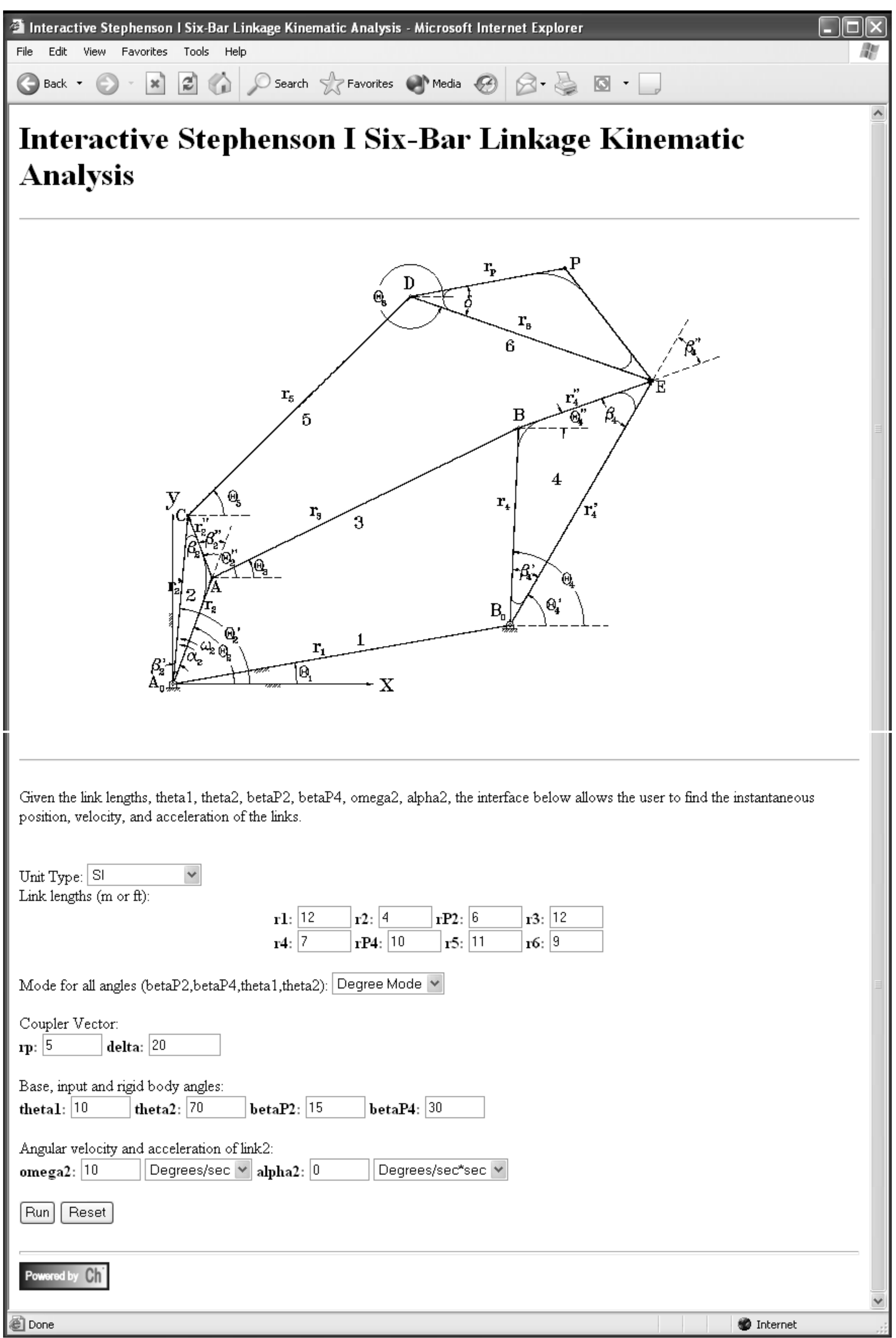


Figure 5.24: Web page for Stephenson (I) sixbar kinematic analysis.

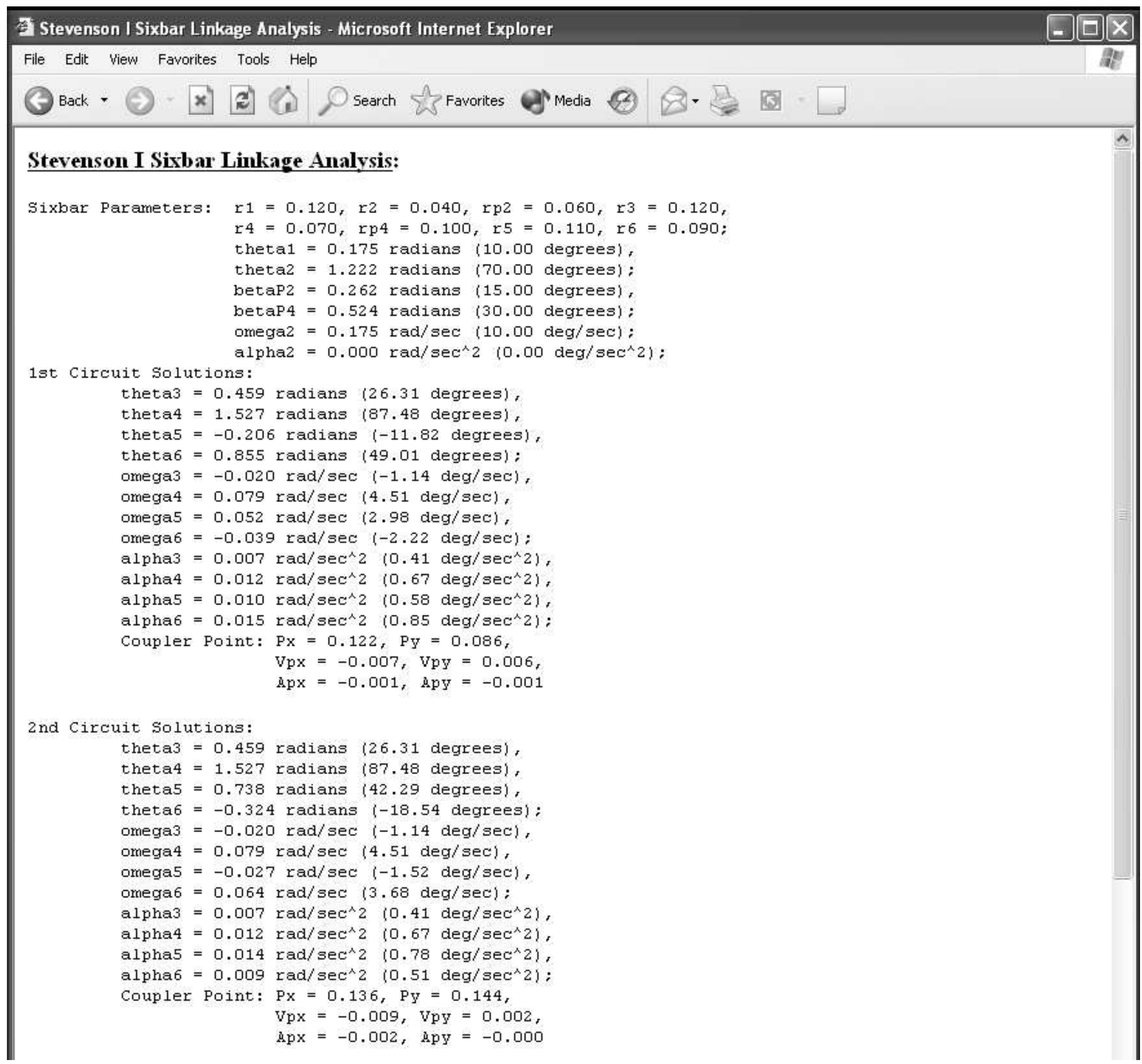


Figure 5.25: Output of web-based Stephenson (I) sixbar kinematic analysis.

```
3rd Circuit Solutions:
        theta3 = -0.777 radians (-44.52 degrees),
        theta4 = -1.845 radians (-105.70 degrees),
        theta5 = -2.023 radians (-115.92 degrees),
        theta6 = -0.110 radians (-6.28 degrees);
        omega3 = -0.005 rad/sec (-0.29 deg/sec),
        omega4 = -0.104 rad/sec (-5.93 deg/sec),
        omega5 = 0.024 rad/sec (1.36 deg/sec),
        omega6 = 0.085 rad/sec (4.90 deg/sec);
        alpha3 = 0.019 rad/sec^2 (1.07 deg/sec^2),
        alpha4 = 0.014 rad/sec^2 (0.81 deg/sec^2),
        alpha5 = 0.023 rad/sec^2 (1.31 deg/sec^2),
        alpha6 = 0.028 rad/sec^2 (1.62 deg/sec^2);
        Coupler Point: Px = 0.003, Py = -0.019,
                       Vpx = -0.010, Vpy = 0.004,
                       Apx = 0.001, Apy = -0.002

4th Circuit Solutions:
        theta3 = -0.777 radians (-44.52 degrees),
        theta4 = -1.845 radians (-105.70 degrees),
        theta5 = -0.391 radians (-22.43 degrees),
        theta6 = -2.305 radians (-132.06 degrees);
        omega3 = -0.005 rad/sec (-0.29 deg/sec),
        omega4 = -0.104 rad/sec (-5.93 deg/sec),
        omega5 = 0.067 rad/sec (3.85 deg/sec),
        omega6 = 0.006 rad/sec (0.32 deg/sec);
        alpha3 = 0.019 rad/sec^2 (1.07 deg/sec^2),
        alpha4 = 0.014 rad/sec^2 (0.81 deg/sec^2),
        alpha5 = 0.025 rad/sec^2 (1.44 deg/sec^2),
        alpha6 = 0.020 rad/sec^2 (1.13 deg/sec^2);
        Coupler Point: Px = 0.096, Py = -0.031,
                       Vpx = -0.007, Vpy = 0.008,
                       Apx = 0.001, Apy = 0.001
```

Done Internet

Figure 5.25: Output of web-based Stephenson (I) sixbar kinematic analysis (Contd).

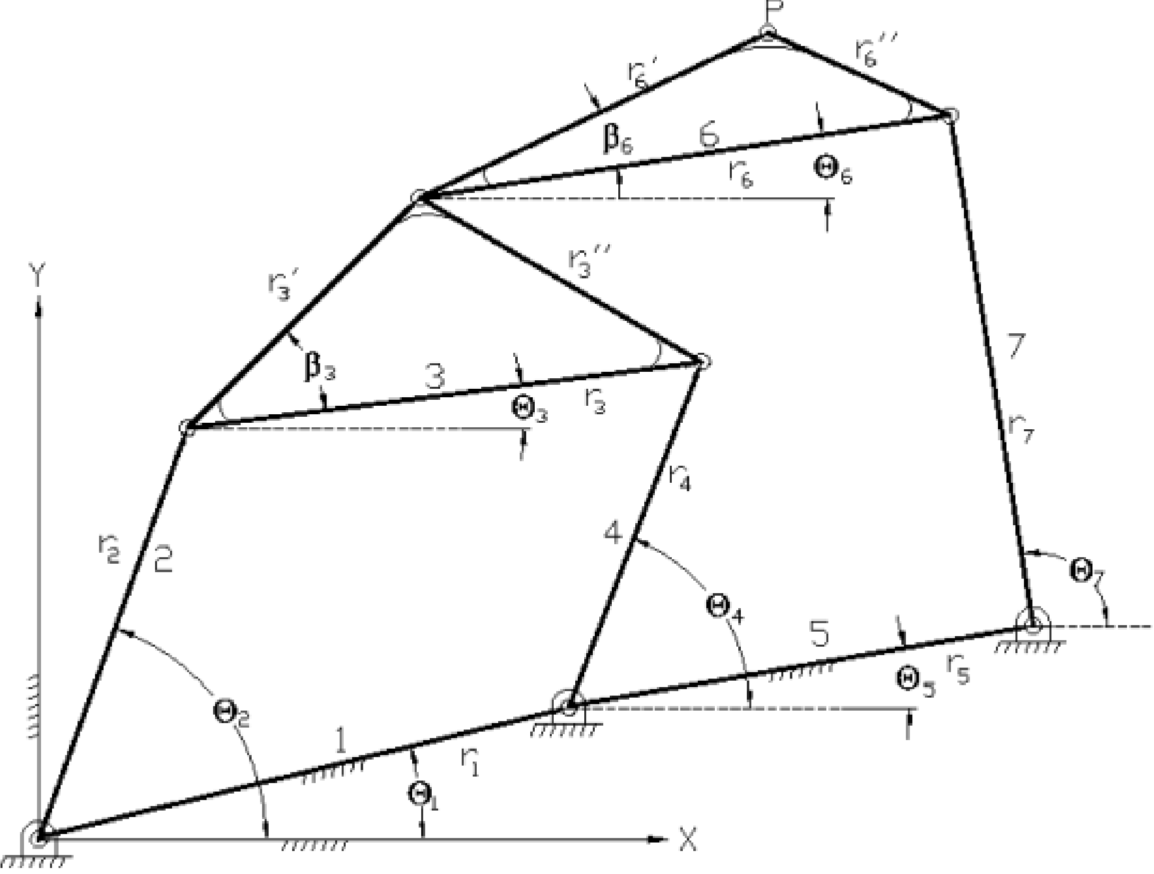


Figure 5.28: Stephenson (III) Sixbar Linkage

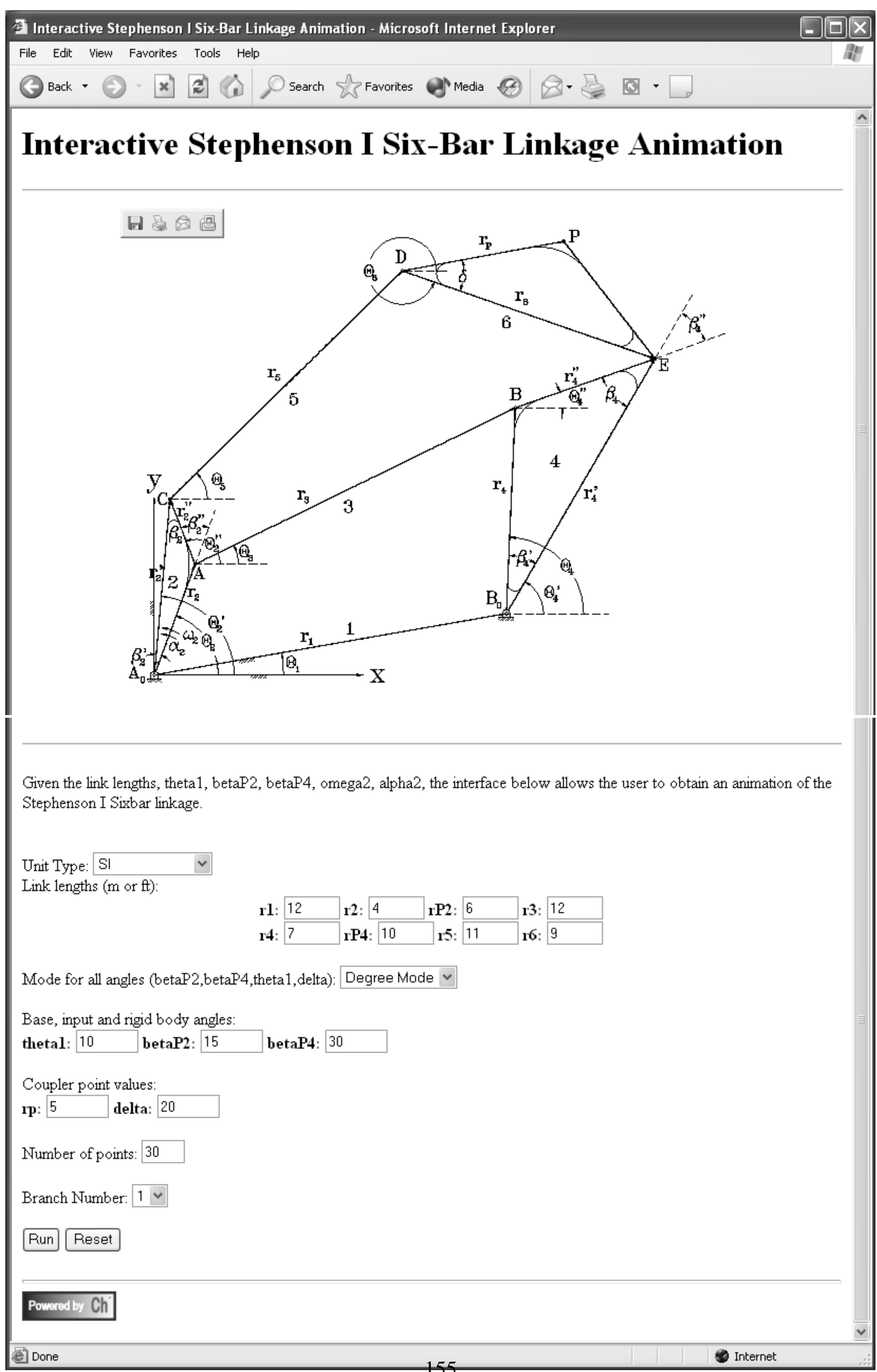


Figure 5.26: Web page for Stephenson (I) sixbar linkage animation.

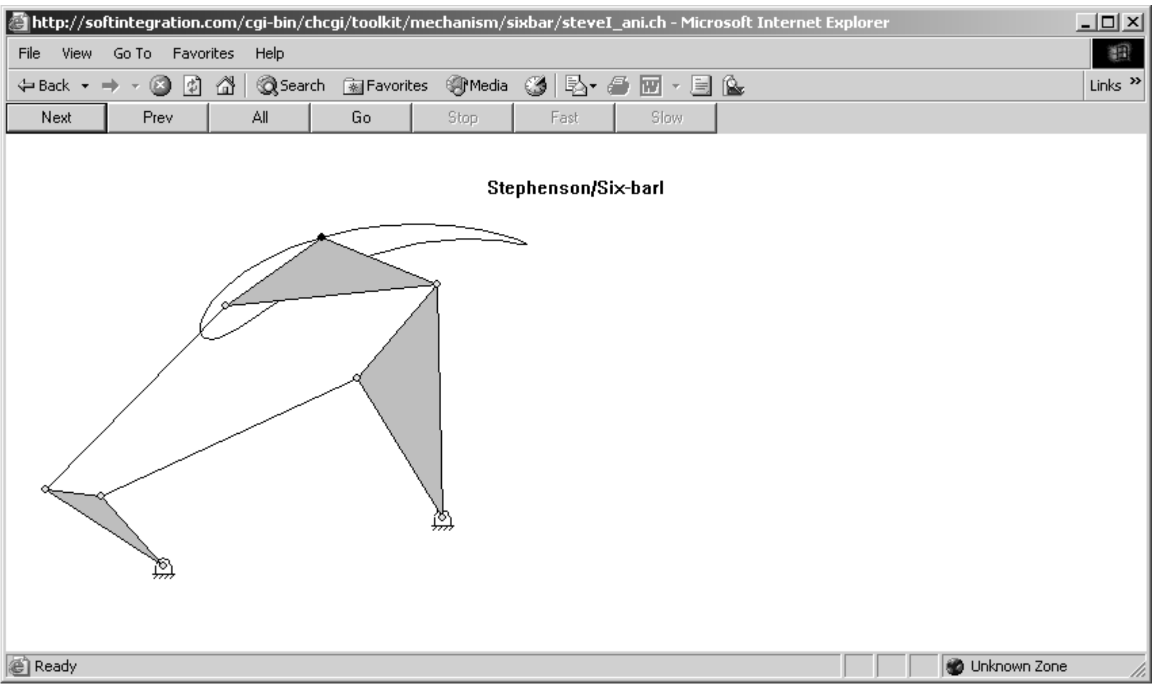


Figure 5.27: Snapshot of a Stephenson (I) sixbar animation.

5.5.1 Position Analysis

In order to determine the angular position (θ_3, θ_4, θ_6, and θ_7) of the various links of a Stephenson (III) sixbar, two loop closure equations (5.182 and 5.183) are needed. Since the analysis of the fourbar linkage has previously been described, consider that angular positions θ_3 and θ_4 are known values so that only the second loop equation need to be analyzed to find a solution for θ_6 and θ_7.

$$\mathbf{r}_1 + \mathbf{r}_4 = \mathbf{r}_2 + \mathbf{r}_3 \tag{5.182}$$

$$\mathbf{r}_2 + \mathbf{r}_3' + \mathbf{r}_6 = \mathbf{r}_1 + \mathbf{r}_5 + \mathbf{r}_7 \tag{5.183}$$

By rewriting and isolating the unknown values, equation (5.183) becomes equation (5.185). In this form, the solutions for θ_6 and θ_7 can be determined by function `complexsolve()`. Note that there are four possible solutions to the assembled linkage, two geometric inversions for each fourbar linkage. Also, the angular positions of this sixbar linkage can be determined by member function `angularPos()` of class `CStevSixbarIII`.

$$r_2 e^{i\theta_2} + r_3' e^{i(\theta_3+\beta_3)} + r_6 e^{i\theta_6} = r_1 e^{i\theta_1} + r_5 e^{i\theta_5} + r_7 e^{i\theta_7} \tag{5.184}$$

$$r_6 e^{i\theta_6} - r_7 e^{i\theta_7} = r_1 e^{i\theta_1} - r_2 e^{i\theta_2} - r_3' e^{i(\theta_3+\beta_3)} + r_5 e^{i\theta_5} \tag{5.185}$$

5.5.2 Velocity Analysis

The initial step of velocity analysis of the Stephenson (III) sixbar linkage is to differentiate equation (5.185) to determine ω_6 as well as ω_7 (assume that ω_3 and ω_4 are known values). With the unknown terms already on the left-hand side, equation (5.187) is multiplied by $e^{-i\theta_6}$ and $e^{-i\theta_7}$ so that equations (5.188) and (5.189) can be used to solve for ω_6 and ω_7, respectively. The solutions for these values are shown below (5.192)

and (5.193). Calculating the angular velocities, ω_3, ω_4, ω_6, and ω_7 can also be done with member function `angularVel()`.

$$ir_6\omega_6 e^{i\theta_6} - ir_7\omega_7 e^{i\theta_7} = -ir_2\omega_2 e^{i\theta_2} - ir'_3\omega_3 e^{i(\theta_3+\beta_3)} \quad (5.186)$$

$$r_6\omega_7 e^{i\theta_6} - r_7\omega_7 e^{i\theta_7} = -r_2\omega_2 e^{i\theta_2} - r'_3\omega_3 e^{i(\theta_3+\beta_3)} \quad (5.187)$$

$$r_6\omega_6 e^{i(\theta_6-\theta_7)} - r_7\omega_7 = -r_2\omega_2 e^{i(\theta_2-\theta_7)} - r'_3\omega_3 e^{i(\theta_3+\beta_3-\theta_7)} \quad (5.188)$$

$$r_6\omega_6 - r_7\omega_7 e^{i(\theta_7-\theta_6)} = -r2\omega_2 e^{i(\theta_2-\theta_6)} - r'_3\omega_3 e^{i(\theta_3+\beta_3-\theta_6)} \quad (5.189)$$

$$r_6\omega_6 \sin(\theta_6-\theta_7) = -r_2\omega_2 \sin(\theta_2-\theta_7) - r'_3\omega_3 \sin(\theta_3+\beta_3-\theta_7) \quad (5.190)$$

$$-r_7\omega_7 \sin(\theta_7-\theta_6) = -r_2\omega_2 \sin(\theta_2-\theta_6) - r'_3\omega_3 \sin(\theta_3+\beta_3-\theta_6) \quad (5.191)$$

$$\omega_6 = -\frac{r_2\omega_2 \sin(\theta_2-\theta_7) + r'_3\omega_3 \sin(\theta_3+\beta_3-\theta_7)}{r_6 \sin(\theta_6-\theta_7)} \quad (5.192)$$

$$\omega_7 = \frac{r_2\omega_2 \sin(\theta_2-\theta_6) + r'_3\omega_3 \sin(\theta_3+\beta_3-\theta_6)}{r_7 \sin(\theta_7-\theta_6)} \quad (5.193)$$

5.5.3 Acceleration Analysis

Acceleration analysis of the Stephenson (III) sixbar can be simplified by assuming that α_3 and α_4 have already been solved by evaluating the second derivative to the first loop closure equation. Thus, angular accelerations, α_6 and α_7, can be determined by considering the second derivative of the second loop closure equation (5.183). Afer some rearrangement to place the two unknown terms on the left-hand side, this becomes equation (5.194). Multiplying equation (5.194) by $e^{-i\theta_6}$ and $e^{-i\theta_7}$ and then only considering the real part of equations (5.199) and (5.200) will allow for the isolation of α_6 and α_7, respectively. With the two desired angular acceleration values isolated, solutions can be found for these two values using equations (5.201) and (5.202).

$$\begin{aligned} ir_6\alpha_6 e^{i\theta_6} - ir_7\alpha_7 e^{i\theta_7} &= -ir_2\alpha_2 e^{i\theta_2} + r_2\omega_2^2 e^{i\theta_2} - ir'_3\alpha_3 e^{i(\theta_3+\beta_3)} \\ &\quad + r'_3\omega_3^2 e^{i(\theta_3+\beta_3)} + r_6\omega_6^2 e^{i\theta_6} - r_7\omega_7^2 e^{i\theta_7} \end{aligned} \quad (5.194)$$

$$\begin{aligned} r_6\alpha_6 e^{i(\theta_6-\theta_7+\pi/2)} - r_7\alpha_7 e^{i(\pi/2)} &= -r_2\alpha_2 e^{i(\theta_2-\theta_7+\pi/2)} + r_2\omega_2^2 e^{i(\theta_2-\theta_7)} \\ &\quad - r'_3\alpha_3 e^{i(\theta_3+\beta_3-\theta_7+\pi/2)} + r'_3\omega_3^2 e^{i(\theta_3+\beta_3-\theta_7)} \\ &\quad + r_6\omega_6^2 e^{i(\theta_6-\theta_7)} - r_7\omega_7^2 \end{aligned} \quad (5.195)$$

$$\begin{aligned} r_6\alpha_6 e^{i(\pi/2)} - r_7\alpha_7 e^{i(\theta_7-\theta_6+\pi/2)} &= -r_2\alpha_2 e^{i(\theta_2-\theta_6+\pi/2)} + r_2\alpha_2 e^{i(\theta_2-\theta_6)} \\ &\quad - r'_3\alpha_3 e^{i(\theta_3+\beta_3-\theta_6+\pi/2)} + r'_3\omega_3^2 e^{i(\theta_3+\beta_3-\theta_6+\pi/2)} \\ &\quad + r_6\omega_6^2 - r_7\omega_7^2 e^{i(\theta_7-\theta_6)} \end{aligned} \quad (5.196)$$

$$\begin{aligned} r_6\alpha_6 \cos(\theta_6-\theta_7+\pi/2) &= -r_2\alpha_2 \cos(\theta_2-\theta_7+\pi/2) + r_2\omega_2^2 \cos(\theta_2-\theta_7) \\ &\quad - r'_3\alpha_3 \cos(\theta_3+\beta_3-\theta_7+\pi/2) + r'_3\omega_3^2 \cos(\theta_3+\beta_3-\theta_7) \\ &\quad + r_6\omega_6^2 \cos(\theta_6-\theta_7) - r_7\omega_7^2 \end{aligned} \quad (5.197)$$

$$\begin{aligned} -r_7\alpha_7 \cos(\theta_7-\theta_6+\pi/2) &= -r_2\alpha_2 \cos(\theta_2-\theta_6+\pi/2) + r_2\omega_2^2 \cos(\theta_2-\theta_6) \\ &\quad - r'_3\alpha_3 \cos(\theta_3+\beta_3-\theta_6+\pi/2) + r'_3\omega_3^2 \cos(\theta_3+\beta_3-\theta_6) \\ &\quad + r_6\omega_6^2 - r_7\omega_7^2 \cos(\theta_7-\theta_6) \end{aligned} \quad (5.198)$$

$$-r_6\alpha_6 \sin(\theta_6-\theta_7) = r_2\alpha_2 \cos(\theta_2-\theta_7) + r_2\omega_2^2 \cos(\theta_2-\theta_7) + r'_3\alpha_3 \sin(\theta_3+\beta_3-\theta_7)$$

$$+ r_3'\omega_3^2\cos(\theta_3+\beta_3-\theta_7) + r_6\omega_6^2\cos(\theta_6-\theta_7) - r_7\omega_7^2 \tag{5.199}$$

$$\begin{aligned} r_7\alpha_7\sin(\theta_7-\theta_6) &= r_2\alpha_2\sin(\theta_2-\theta_6) + r_2\omega_2^2\cos(\theta_2-\theta_6) + r_3'\alpha_3\sin(\theta_3+\beta_3-\theta_6) \\ &\quad + r_3'\omega_2^2\cos(\theta_3+\beta_3-\theta_6) + r_6\omega_6^2 - r_7\omega_7^2\cos(\theta_7-\theta_6) \end{aligned} \tag{5.200}$$

$$\begin{aligned} \alpha_6 &= -\frac{r_2\alpha_2\sin(\theta_2-\theta_7) + r_2\omega_2^2\cos(\theta_2-\theta_7) + r_3'\alpha_3\sin(\theta_3+\beta_3-\theta_7)}{r_6\sin(\theta_6-\theta_7)} \\ &\quad + \frac{-r_3'\omega_3^2\cos(\theta_3+\beta_3-\theta_7) + r_6\omega_6^2\cos(\theta_6-\theta_7) - r_7\omega_7^2}{r_6\sin(\theta_6-\theta_7)} \end{aligned} \tag{5.201}$$

$$\begin{aligned} \alpha_7 &= \frac{r_2\alpha_2\sin(\theta_2-\theta_6) + r_2\omega_2^2\cos(\theta_2-\theta_6) + r_3'\alpha_3\sin(\theta_3+\beta_3-\theta_6)}{r_7\sin(\theta_7-\theta_6)} \\ &\quad + \frac{r_3'\omega_3^2\cos(\theta_3+\beta_3-\theta_6) + r_6\omega_6^2 - r_7\omega_7^2\cos(\theta_7-\theta_6)}{r_7\sin(\theta_7-\theta_6)} \end{aligned} \tag{5.202}$$

Similar to the position and velocity analysis, member function `angularAccel()` can be used to solve for θ_3, θ_4, θ_6, and θ_7.

> **Problem 1:** The Stephenson (III) sixbar shown in Figure 5.28 has parameters $r_1 = 9cm, r_2 = 4cm, r_3 = 10cm, r_3' = 3cm, r_4 = 6cm, r_5 = 8cm, r_6 = 9cm, r_7 = 12cm$, $\theta_1 = 0$, $\theta_2 = 25^o$, $\omega_2 = 10^o/sec$, $\alpha_2 = 0$, $\theta_5 = 15^o$, and $\beta_3 = 20^o$. Compute the angular acceleration of links 3,4,6, and 7.

Although α_3, α_4, α_5, and α_7 can be calculated with the equations previously derived, the process is tedious and undesirable since intermediate terms, such as angular positions and velocities, must also be calculated. That is why class `CStevSixbarIII` is available to simply this task. Program 36 utilizes class `CStevSixbarIII` to calculate the angular acceleration values of the Stephenson (III) sixbar linkage, and the solutions are listed below.

```
Solution #1:
 alpha3 = 0.027 rad/s^2, alpha4 = 0.047 rad/s^2,
 alpha6 = -0.017 rad/s^2, alpha7 = 0.006 rad/s^2
Solution #2:
 alpha3 = 0.027 rad/s^2, alpha4 = 0.047 rad/s^2,
 alpha6 = 0.018 rad/s^2, alpha7 = -0.004 rad/s^2
Solution #3:
 alpha3 = -0.027 rad/s^2, alpha4 = -0.074 rad/s^2,
 alpha6 = -0.014 rad/s^2, alpha7 = 0.012 rad/s^2
Solution #4:
 alpha3 = -0.027 rad/s^2, alpha4 = -0.074 rad/s^2,
 alpha6 = 0.026 rad/s^2, alpha7 = 0.001 rad/s^2
```

```
#include<sixbar.h>

int main()
{
   double r[1:7];
   double rP3, beta3;
   double theta[1:4][1:7], omega[1:4][1:7], alpha[1:4][1:7];
   double theta1 = 0, theta5 = M_DEG2RAD(15), theta2 = M_DEG2RAD(25);
   double omega2 = M_DEG2RAD(10), alpha2 = 0;
   CStevSixbarIII stevIII;
   int i;

   /* Define Stephenson (III) Sixbar linkage. */
   r[1] = 0.09;  r[2] = 0.04;
   r[3] = 0.10;  r[4] = 0.06;
   r[5] = 0.08;   r[6] = 0.09;
   r[7] = 0.12;
   rP3 = 0.03;  beta3 = M_DEG2RAD(20);
   for(i = 1; i <= 4; i++)
   {
      theta[i][1] = theta1;
      theta[i][2] = theta2;
      theta[i][5] = theta5;
      omega[i][2] = omega2;
   }

   /* Perform analysis. */
   stevIII.setLinks(r, rP3, beta3, theta1, theta5);
   stevIII.angularPos(theta[1], theta[2], theta[3], theta[4]);
   stevIII.angularVel(theta[1], omega[1]);
   stevIII.angularVel(theta[2], omega[2]);
   stevIII.angularVel(theta[3], omega[3]);
   stevIII.angularVel(theta[4], omega[4]);
   stevIII.angularAccel(theta[1], omega[1], alpha[1]);
   stevIII.angularAccel(theta[2], omega[2], alpha[2]);
   stevIII.angularAccel(theta[3], omega[3], alpha[3]);
   stevIII.angularAccel(theta[4], omega[4], alpha[4]);

   /* Display results. */
   for(i = 1; i <= 4; i++)
   {
      printf("Solution #%d:\n", i);
      printf("\t alpha3 = %.3f rad/s^2, alpha4 = %.3f rad/s^2,\n",
             alpha[i][3], alpha[i][4]);
      printf("\t alpha6 = %.3f rad/s^2, alpha7 = %.3f rad/s^2\n",
             alpha[i][6], alpha[i][7]);
   }

   return 0;
}
```

Program 36: Program for computing α_3, α_4, α_6, and α_7 using class `CStevSixbarIII`.

5.5.4 Coupler Point Position, Velocity, and Acceleration

Coupler point analysis begin by writing the vector equation of coupler point, P, and then converting it to polar form (5.204). This is the equation for the coupler point position. The velocity and acceleration of the coupler point can be determined by taking the derivatives of equation (5.204). Thus, equations (5.205) and (5.206) are equations for the coupler point velocity and acceleration, respectively. The coupler point position can be calculated by member function `couplerPointPos()` of class `CStevSixbarIII`. Similarly, this class also have member functions to determine the coupler point velocity and acceleration (`couplerPointVel()` and `couplerPointAccel()`, respectively).

$$\mathbf{P} = \mathbf{r}_2 + \mathbf{r}_3' + \mathbf{r}_p \tag{5.203}$$

$$\mathbf{P} = r_2 e^{i\theta_2} + r_3' e^{i(\theta_3+\beta_3)} + r_p e^{i(\theta_6+\beta)} \tag{5.204}$$

$$\mathbf{V}_p = i r_2 \omega_2 e^{i\theta_2} + i r_3' \omega_3 e^{i(\theta_3+\beta_3)} + i r_p e^{i(\theta_6+\beta)} \tag{5.205}$$

$$\begin{aligned}\mathbf{A}_p = {} & i r_2 \alpha_2 e^{i\theta_2} - r_2 \omega_2^2 e^{i\theta_2} + i r_3' \alpha_3 e^{i(\theta_3+\beta_3)} - r_3' \omega_2^2 e^{i(\theta_3+\beta_3)} \\ & + i r_p \alpha_6 e^{i(\theta_6+\beta)} - r_p \omega_6^2 e^{i(\theta_6+\beta)}\end{aligned} \tag{5.206}$$

The member functions that handles coupler point analysis of a Stephenson (III) sixbar linkage are utilized to solve the following problem.

> **Problem 2:** The Stephenson (III) sixbar shown in Figure 5.28 has parameters $r_1 = 9cm, r_2 = 4cm, r_3 = 10cm, r_3' = 3cm, r_4 = 6cm, r_5 = 8cm, r_6 = 9cm, r_7 = 12cm$, $\theta_1 = 0$, $\theta_2 = 25^o$, $\omega_2 = 10^o/sec$, $\alpha_2 = 0$, $\theta_5 = 15^o$, and $\beta_3 = 20^o$. The coupler point parameters are $r_p = 5cm$ and $\beta = 30^o$. Compute the coupler point position, velocity, and acceleration.

The Ch code that solves this problem is labeled as Program 37, and its results are shown below.

```
Solution #1:
 P = complex(0.067,0.084)
 Vp = complex(0.001,0.006)
 Ap = complex(-0.001,0.000)
Solution #2:
 P = complex(0.087,-0.009)
 Vp = complex(-0.004,0.003)
 Ap = complex(-0.001,0.001)
Solution #3:
 P = complex(0.051,0.051)
 Vp = complex(-0.003,0.004)
 Ap = complex(-0.001,-0.001)
Solution #4:
 P = complex(0.098,-0.033)
 Vp = complex(-0.006,0.002)
 Ap = complex(-0.001,0.000)
```

5.5.5 Animation

The `CStevSixbarIII` class contains member function `animation()` for the purpose of simulating the motion of the Stephenson (III) sixbar linkage. The function prototype for `animation()` is as follows.

```
int CStevSixbarIII::animation(int branchnum, ...);
```

```
#include<stdio.h>
#include<sixbar.h>

int main()
{
   double r[1:7], rP3, beta3;
   double rp, beta;
   double theta[1:4][1:7], omega[1:4][1:7], alpha[1:4][1:7];
   double theta1 = 0, theta5 = M_DEG2RAD(15), theta2 = M_DEG2RAD(25);
   double omega2 = M_DEG2RAD(10), alpha2 = 0;
   double complex P[1:4], Vp[1:4], Ap[1:4];
   CStevSixbarIII stevIII;
   int i;

   /* Define Stephenson (III) Sixbar linkage. */
   r[1] = 0.09;  r[2] = 0.04;
   r[3] = 0.10;  r[4] = 0.06;
   r[5] = 0.08;   r[6] = 0.09;
   r[7] = 0.12;
   rP3 = 0.03;  beta3 = M_DEG2RAD(20);
   rp = 0.05;  beta = M_DEG2RAD(30);
   for(i = 1; i <= 4; i++)
   {
      theta[i][1] = theta1;
      theta[i][2] = theta2;
      theta[i][5] = theta5;
      omega[i][2] = omega2;
   }

   /* Perform analysis. */
   stevIII.setLinks(r, rP3, beta3, theta1, theta5);
   stevIII.setCouplerPoint(rp, beta);
   stevIII.angularPos(theta[1], theta[2], theta[3], theta[4]);
   stevIII.angularVel(theta[1], omega[1]);
   stevIII.angularVel(theta[2], omega[2]);
   stevIII.angularVel(theta[3], omega[3]);
   stevIII.angularVel(theta[4], omega[4]);
   stevIII.angularAccel(theta[1], omega[1], alpha[1]);
   stevIII.angularAccel(theta[2], omega[2], alpha[2]);
   stevIII.angularAccel(theta[3], omega[3], alpha[3]);
   stevIII.angularAccel(theta[4], omega[4], alpha[4]);

   /* Determine coupler point properties. */
   stevIII.couplerPointPos(theta2, P);
   for(i = 1; i <= 4; i++)
   {
      Vp[i] = stevIII.couplerPointVel(theta[i], omega[i]);
      Ap[i] = stevIII.couplerPointAccel(theta[i], omega[i], alpha[i]);
   }
```

Program 37: Program for calculating the coupler point position, velocity, and acceleration.

```
   /* Display results. */
   for(i = 1; i <= 4; i++)
   {
      printf("Solution #%d:\n", i);
      printf("\t P = %.3f\n", P[i]);
      printf("\t Vp = %.3f\n", Vp[i]);
      printf("\t Ap = %.3f\n", Ap[i]);
   }

   return 0;
}
```

Program 37: Program for calculating the coupler point position, velocity, and acceleration (Contd.).

The first argument, `branchnum`, corresponds to the branch number of the Stephenson (III) sixbar linkage. That is, the value of `branchnum` specifies which geometric inversion to animate. Similar to the animation functions of the other classes for mechanism analysis, additional arguments may be inputted into member function `animation()` to store the generated animation data into a file for later use. For example, Program 38 can be used to simulate the motion of the Stephenson (III) sixbar linkage that has been defined in previous examples. Figure 5.29 contains snapshots of the four possible geometric inversions for this sixbar linkage. Note that the number of frames of animation is specified by member function `setNumPoints()` in Program 38.

5.5.6 Web-Based Analysis

Analysis of the Stephenson (III) sixbar linkage can also be performed interactively on the internet. Figure 5.30 shows the main web page for web-based analysis of the sixbar mechanism. Figure 5.31 is the web page used for calculating the unknown instantaneous angular positions, velocities, and accelerations of the Stephenson (III) sixbar linkages. Also, if a coupler is attached to the mechanism, this web page can calculate the coupler point position, velocity, and acceleration as well. The user only needs to specify the link lengths, angles θ_1, θ_5, and β_3, and angular position, velocity, and acceleration of the input link, link 2. Using the specifications of the Stephenson (III) sixbar linkage defined in the above problem statements, the result of using the web-based kinematic analysis is shown in Figure 5.32. Note that for the purpose of this example, the coupler vector was defined as $r_p = 5cm$ and $\beta = 30^o$.

For web-based animation of the Stephenson (III) sixbar linkage, Figure 5.33 can be used. Similar to animating the sixbar mechanism with class `CStevSixbarIII` in the previous section, the parameters of the sixbar needs to be specified along with the number of animation frames to generate. Furthermore, the specific branch number of the Stephenson (III) sixbar may also be indicated prior to execution. Again, using the parameters of the linkage that has already been defined, a snapshot of the animation for the first geometric inversion is shown in Figure 5.34.

```
#include<stdio.h>
#include<sixbar.h>

#define NUMPOINTS 50

int main()
{
   double r[1:7];
   double rP3, beta3,
          rp, beta;
   double theta1 = 0, theta5 = M_DEG2RAD(15);
   CStevSixbarIII stevIII;
   int i;

   /* Define Stephenson (III) Sixbar linkage. */
   r[1] = 0.09;  r[2] = 0.04;
   r[3] = 0.10;  r[4] = 0.06;
   r[5] = 0.08;   r[6] = 0.09;
   r[7] = 0.12;
   rP3 = 0.03;  beta3 = M_DEG2RAD(20);
   rp = 0.05;  beta = M_DEG2RAD(30);

   /* Perform analysis. */
   stevIII.setLinks(r, rP3, beta3, theta1, theta5);
   stevIII.setCouplerPoint(rp, beta, TRACE_ON);
   stevIII.setNumPoints(NUMPOINTS);
   stevIII.animation(1);
   stevIII.animation(2);
   stevIII.animation(3);
   stevIII.animation(4);

   return 0;
}
```

Program 38: Program for animating the Stephenson (III) sixbar linkage.

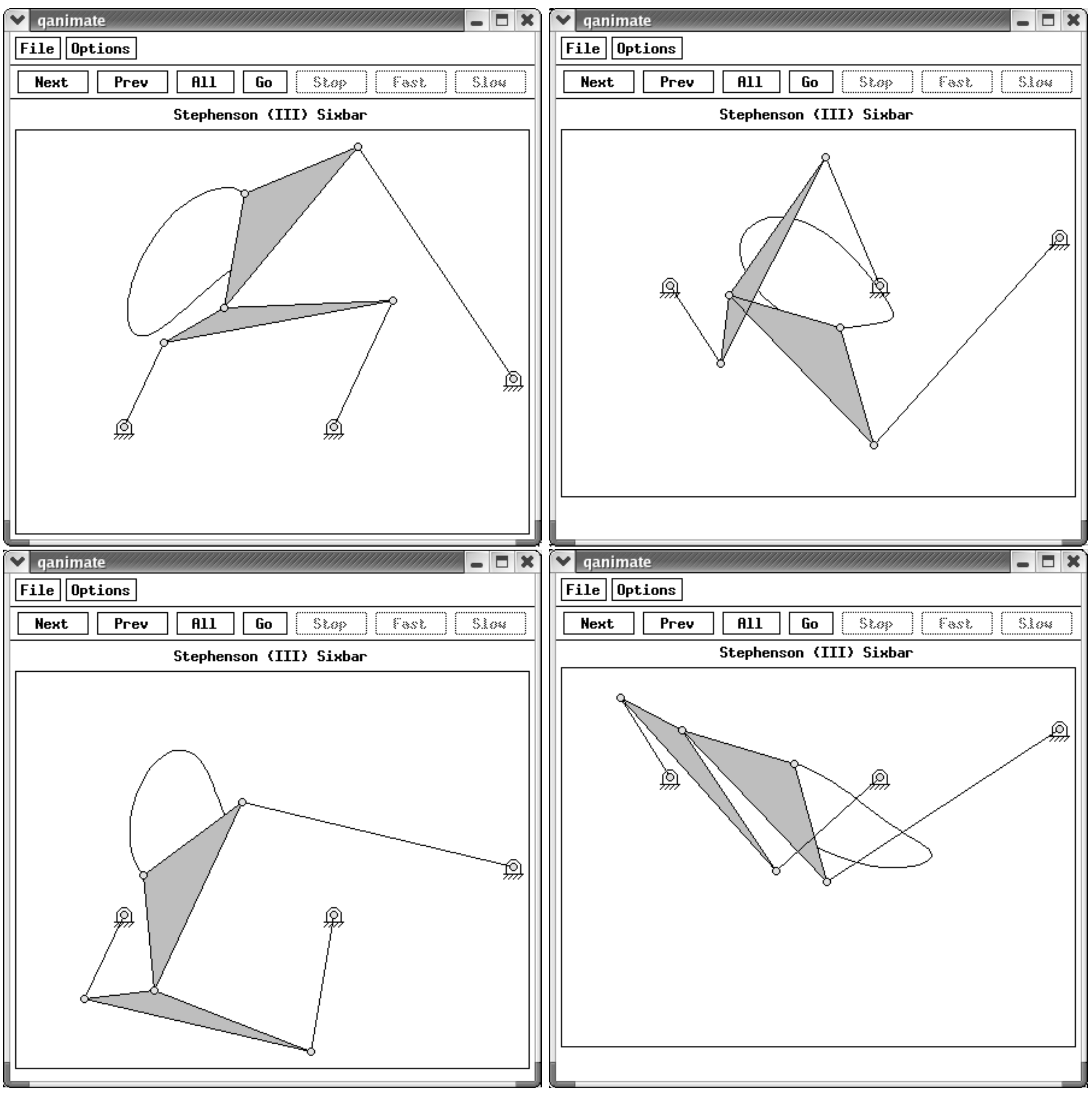


Figure 5.29: Output of Program 38.

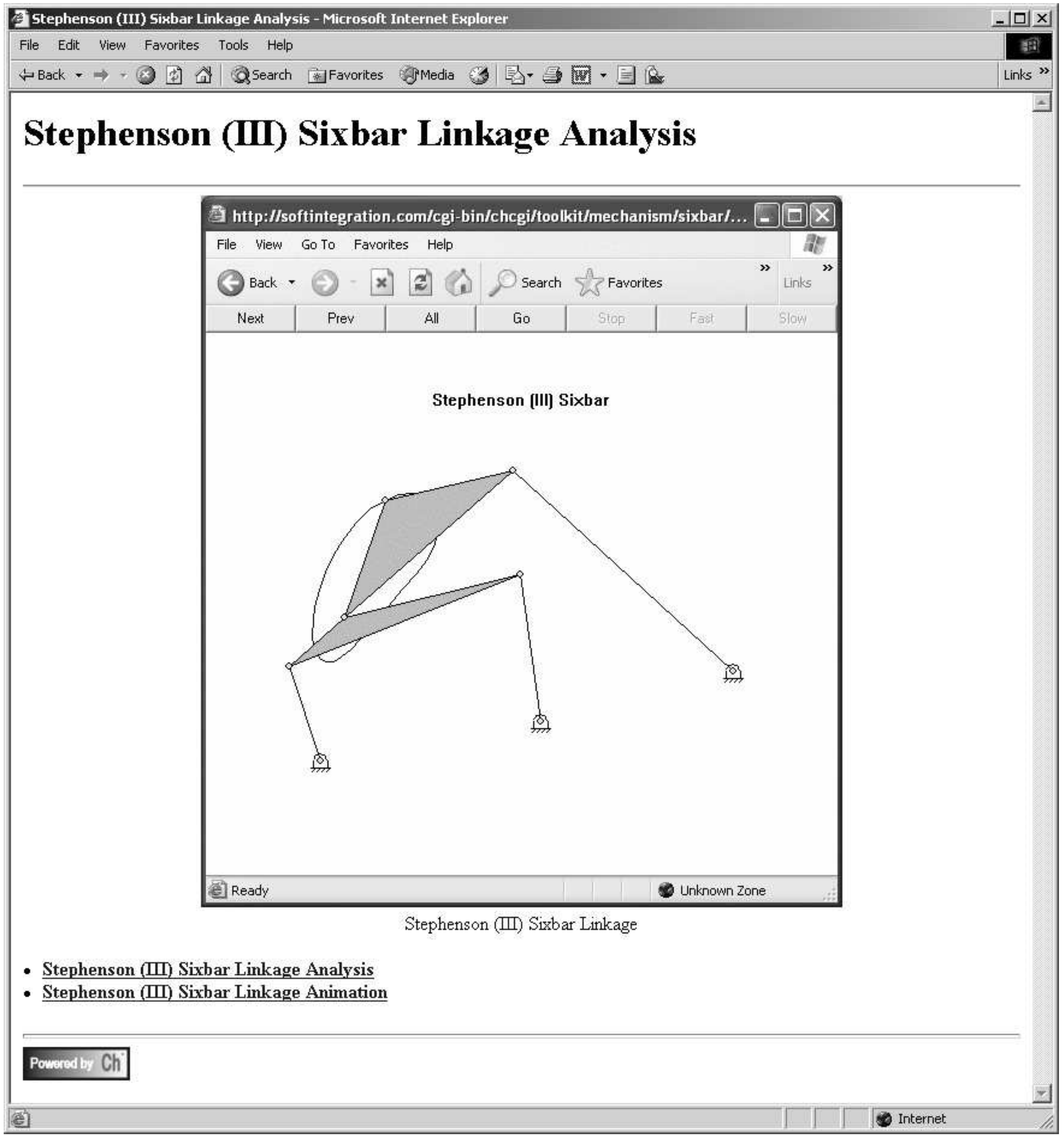


Figure 5.30: Main web page for the Stephenson (III) sixbar linkage.

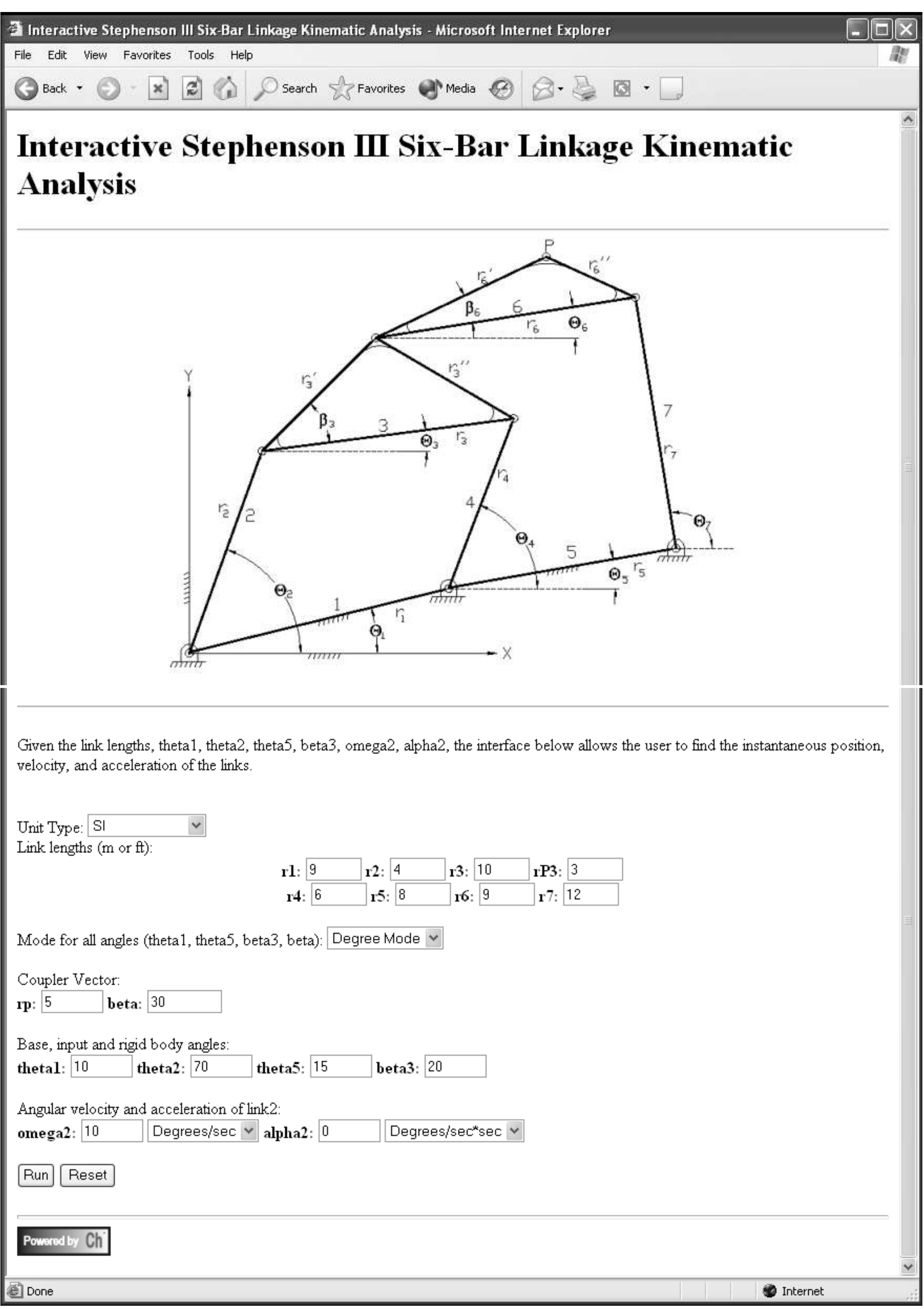
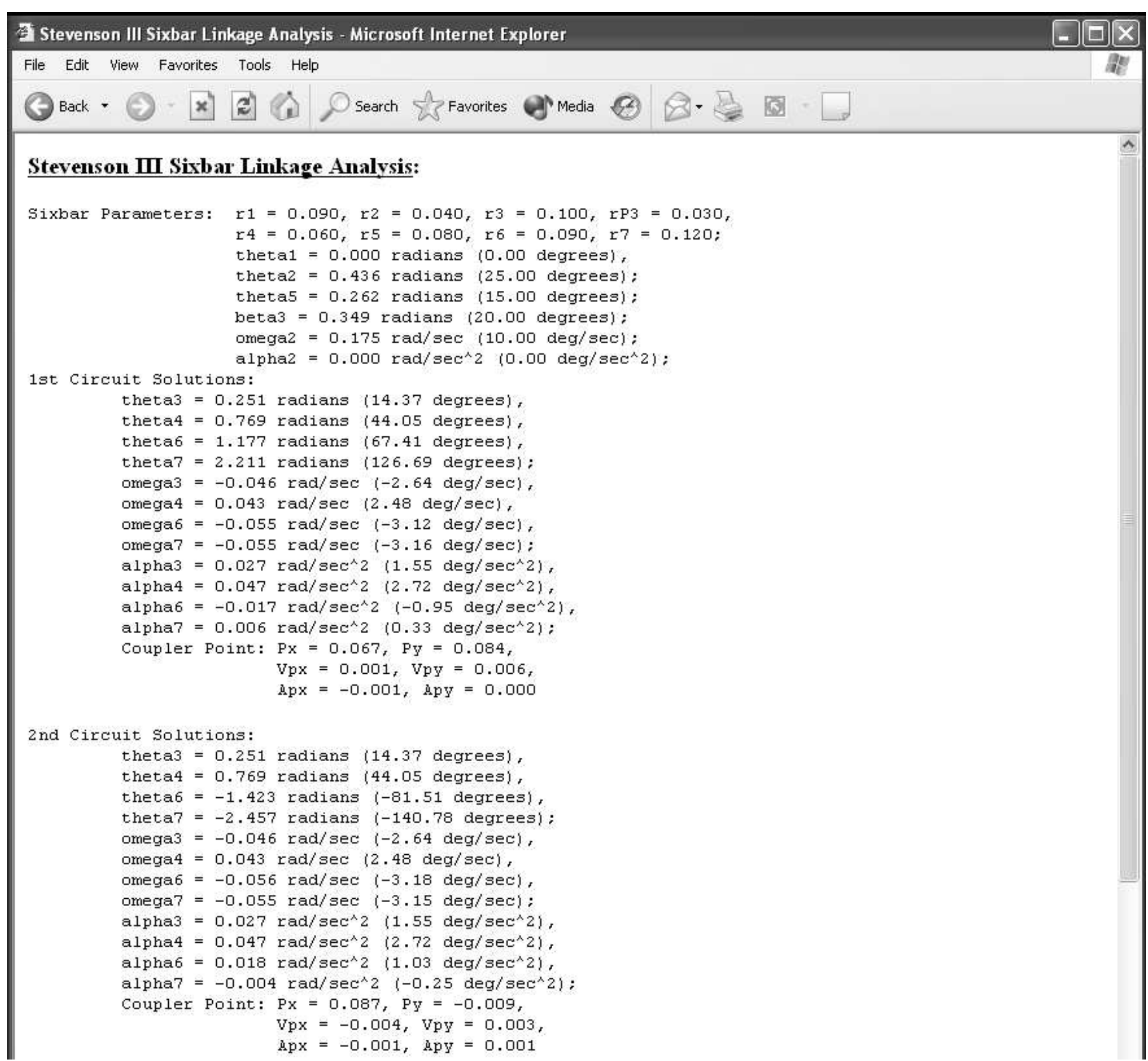


Figure 5.31: Web page for Stephenson (III) sixbar kinematic analysis.



Stevenson III Sixbar Linkage Analysis:

```
Sixbar Parameters:  r1 = 0.090, r2 = 0.040, r3 = 0.100, rP3 = 0.030,
                    r4 = 0.060, r5 = 0.080, r6 = 0.090, r7 = 0.120;
                    theta1 = 0.000 radians (0.00 degrees),
                    theta2 = 0.436 radians (25.00 degrees);
                    theta5 = 0.262 radians (15.00 degrees);
                    beta3 = 0.349 radians (20.00 degrees);
                    omega2 = 0.175 rad/sec (10.00 deg/sec);
                    alpha2 = 0.000 rad/sec^2 (0.00 deg/sec^2);
1st Circuit Solutions:
         theta3 = 0.251 radians (14.37 degrees),
         theta4 = 0.769 radians (44.05 degrees),
         theta6 = 1.177 radians (67.41 degrees),
         theta7 = 2.211 radians (126.69 degrees);
         omega3 = -0.046 rad/sec (-2.64 deg/sec),
         omega4 = 0.043 rad/sec (2.48 deg/sec),
         omega6 = -0.055 rad/sec (-3.12 deg/sec),
         omega7 = -0.055 rad/sec (-3.16 deg/sec);
         alpha3 = 0.027 rad/sec^2 (1.55 deg/sec^2),
         alpha4 = 0.047 rad/sec^2 (2.72 deg/sec^2),
         alpha6 = -0.017 rad/sec^2 (-0.95 deg/sec^2),
         alpha7 = 0.006 rad/sec^2 (0.33 deg/sec^2);
         Coupler Point: Px = 0.067, Py = 0.084,
                        Vpx = 0.001, Vpy = 0.006,
                        Apx = -0.001, Apy = 0.000

2nd Circuit Solutions:
         theta3 = 0.251 radians (14.37 degrees),
         theta4 = 0.769 radians (44.05 degrees),
         theta6 = -1.423 radians (-81.51 degrees),
         theta7 = -2.457 radians (-140.78 degrees);
         omega3 = -0.046 rad/sec (-2.64 deg/sec),
         omega4 = 0.043 rad/sec (2.48 deg/sec),
         omega6 = -0.056 rad/sec (-3.18 deg/sec),
         omega7 = -0.055 rad/sec (-3.15 deg/sec);
         alpha3 = 0.027 rad/sec^2 (1.55 deg/sec^2),
         alpha4 = 0.047 rad/sec^2 (2.72 deg/sec^2),
         alpha6 = 0.018 rad/sec^2 (1.03 deg/sec^2),
         alpha7 = -0.004 rad/sec^2 (-0.25 deg/sec^2);
         Coupler Point: Px = 0.087, Py = -0.009,
                        Vpx = -0.004, Vpy = 0.003,
                        Apx = -0.001, Apy = 0.001
```

Figure 5.32: Output of web-based Stephenson (III) sixbar kinematic analysis.

```
3rd Circuit Solutions:
        theta3 = -0.860 radians (-49.28 degrees),
        theta4 = -1.378 radians (-78.97 degrees),
        theta6 = 1.479 radians (84.76 degrees),
        theta7 = 2.507 radians (143.64 degrees);
        omega3 = -0.137 rad/sec (-7.84 deg/sec),
        omega4 = -0.226 rad/sec (-12.96 deg/sec),
        omega6 = -0.047 rad/sec (-2.70 deg/sec),
        omega7 = -0.031 rad/sec (-1.79 deg/sec);
        alpha3 = -0.027 rad/sec^2 (-1.54 deg/sec^2),
        alpha4 = -0.074 rad/sec^2 (-4.25 deg/sec^2),
        alpha6 = -0.014 rad/sec^2 (-0.78 deg/sec^2),
        alpha7 = 0.012 rad/sec^2 (0.70 deg/sec^2);
        Coupler Point: Px = 0.051, Py = 0.051,
                       Vpx = -0.003, Vpy = 0.004,
                       Apx = -0.001, Apy = -0.001

4th Circuit Solutions:
        theta3 = -0.860 radians (-49.28 degrees),
        theta4 = -1.378 radians (-78.97 degrees),
        theta6 = -1.130 radians (-64.77 degrees),
        theta7 = -2.158 radians (-123.65 degrees);
        omega3 = -0.137 rad/sec (-7.84 deg/sec),
        omega4 = -0.226 rad/sec (-12.96 deg/sec),
        omega6 = -0.022 rad/sec (-1.28 deg/sec),
        omega7 = -0.038 rad/sec (-2.19 deg/sec);
        alpha3 = -0.027 rad/sec^2 (-1.54 deg/sec^2),
        alpha4 = -0.074 rad/sec^2 (-4.25 deg/sec^2),
        alpha6 = 0.026 rad/sec^2 (1.52 deg/sec^2),
        alpha7 = 0.001 rad/sec^2 (0.03 deg/sec^2);
        Coupler Point: Px = 0.098, Py = -0.033,
                       Vpx = -0.006, Vpy = 0.002,
                       Apx = -0.001, Apy = 0.000
```

Done Internet

Figure 5.32: Output of web-based Stephenson (III) sixbar kinematic analysis (Contd.).

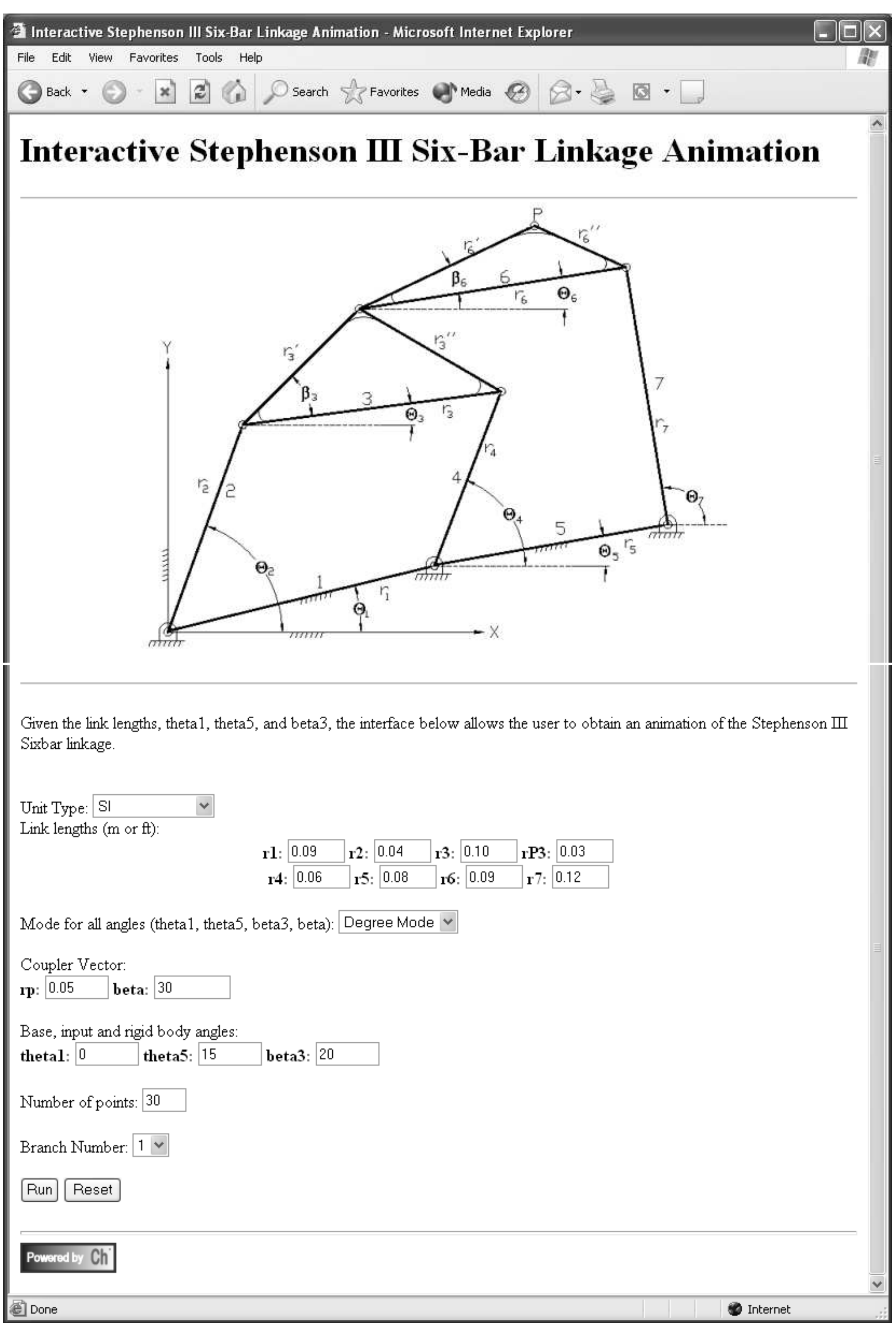


Figure 5.33: Web page for Stephenson (III) sixbar linkage animation.

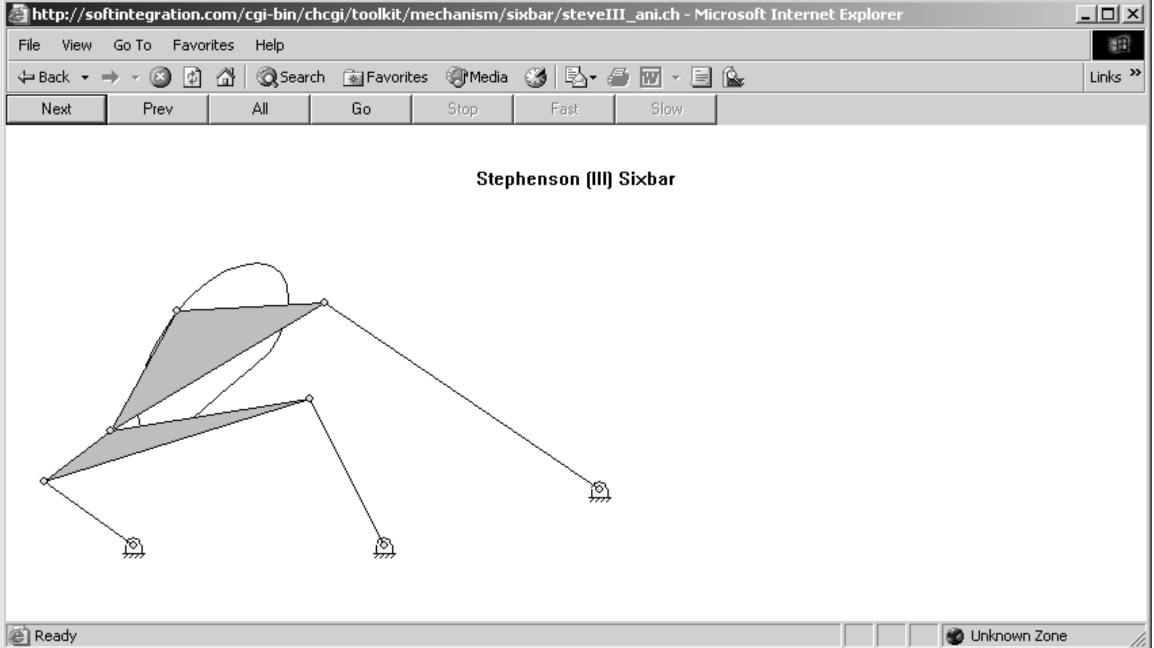


Figure 5.34: Snapshot of a Stephenson (III) sixbar animation.

Chapter 6

Cam Design

6.1 Introduction to Cam Design

Cams are some of the most commonly used mechanisms in automation and assembly systems. A cam mechanism, consisting of a cam lobe and a follower, typically transforms rotational motion to oscillatory motion, translational motion, or a combination of both. Cam synthesis is the process of designing a cam which moves a follower in the desired manner.

Header file **cam.h** contains functions for design of a cam/follower system. The member functions of the cam class allow for selection of cam and follower parameters, CNC manufacturing parameters, and output options. The cam class can be used directly or accessed through the **cam** program using command line arguments or through a web browser.

All of the parameters used in Ch programs and web interface described in this chapter, for both the translating and oscillating follower cams, are outlined below. Figures 6.1 — 6.4 illustrate these parameters.

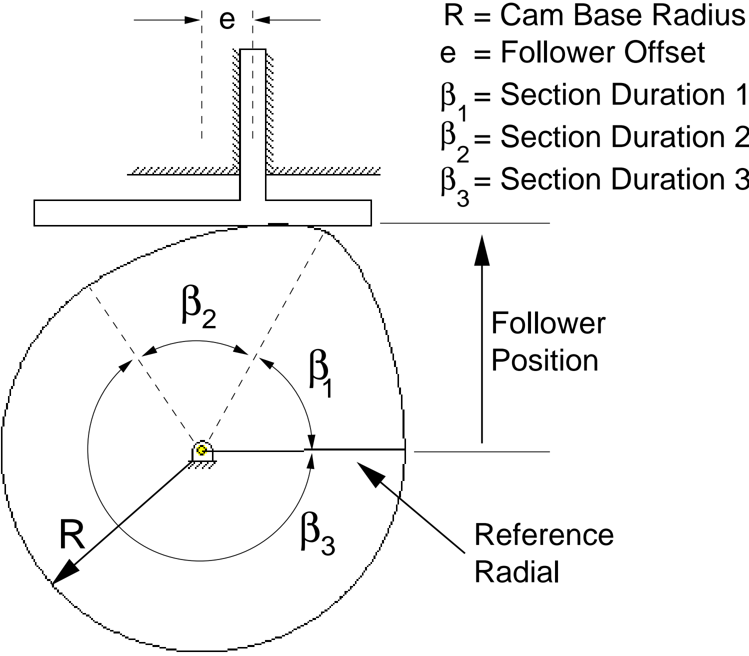


Figure 6.1: Translating flat-faced follower parameters

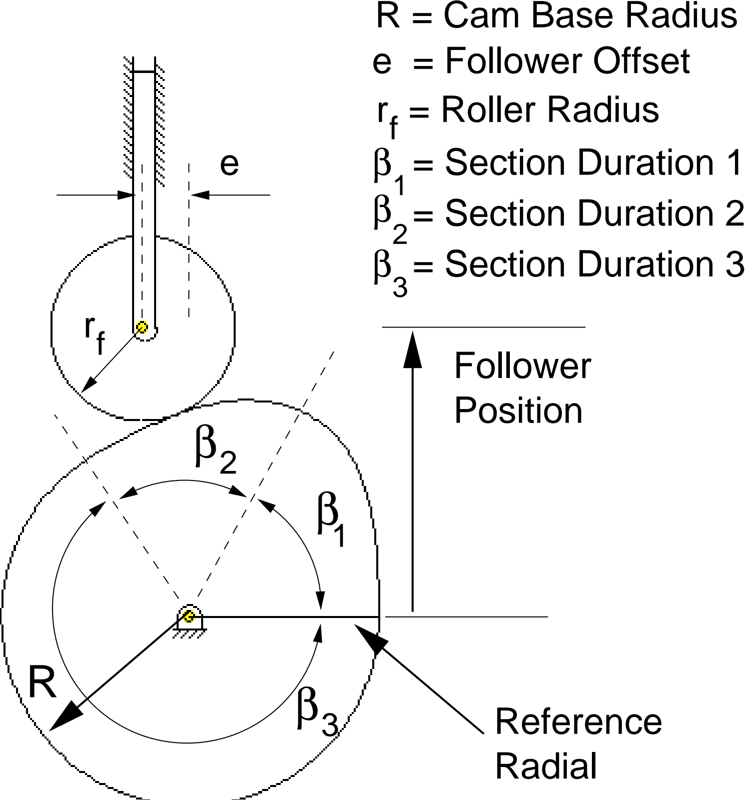


Figure 6.2: Translating roller follower parameters

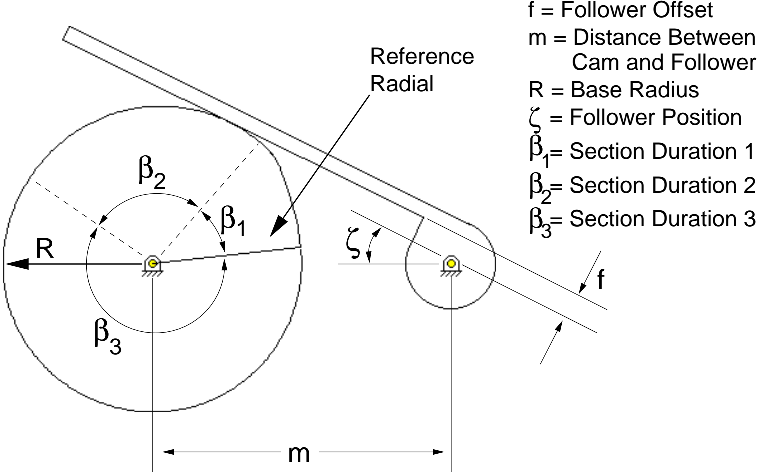


Figure 6.3: Oscillating flat-faced follower parameters

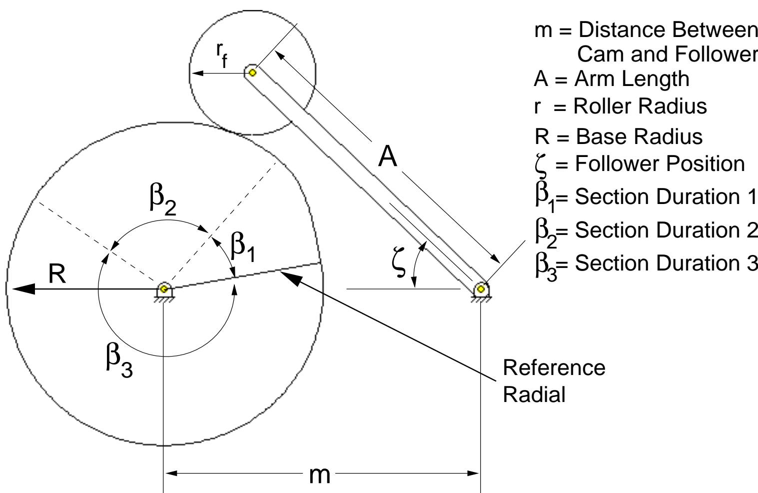


Figure 6.4: Oscillating roller follower parameters

- **The following cam parameters are common to both the translating and oscillating follower types.**

Base Radius The base radius is the initial radius of the cam. The cam profile is built up from a disk of this size.

Profile Points The number of points used to calculate the cam profile.

Cam Angular Velocity The angular velocity of the cam in rad/s. Positive clockwise.

Duration The duration is the angular size of the section.

Motion Type The shape of the displacement profile may be chosen to be either harmonic or cycloidal. For sections with a Lift or Oscillation Angle of zero, the section is circular and this parameter has no effect.

- **The translating follower specific parameters are listed below.**

Follower Offset The distance between the cam center and the follower line of motion. Positive to the right for flat followers and positive to the left for roller followers.

Roller Radius For roller followers, the radius of the roller.

Lift Lift specifies the change in the follower output for the section. This is specified as a positive number if the follower is to move away from the cam, or as a negative number if the cam is to move towards the cam. A lift of zero is entered if the follower displacement is to remain constant for the duration of the cam section. For the last section of the cam, the duration and change in lift will be chosen automatically to form a continuous cam profile.

- **The oscillating follower parameters are listed below.**

Follower Offset For a flat-face follower, the follower offset is the distance from the follower face to the follower pivot point.

Distance Between the Cam and Follower The distance between the cam and follower is measured from the cam center to the follower pivot point.

Follower Arm Length For roller followers, the length of the arm connecting the pivot point and the roller center.

Roller Radius For roller followers, the radius of the roller.

Oscillation Angle Oscillation angle specifies the change in the follower output for the section. This is specified as a positive number if the follower is to move away from the cam, or as a negative number if the cam is to move towards the cam. A oscillation angle of zero is entered if the follower displacement is to remain constant for the duration of the cam section. For the last section of the cam, the duration and change in oscillation angle will be chosen automatically to form a continuous cam profile.

- **Parameters for generation of CNC code are listed below.**

Cutter Radius As shown in Figure 6.5, the cutter radius is the radius of the mill bit used by the CNC machine (in).

Cutter Length As shown in Figure 6.5, the cutter length is the length of the mill bit used by the CNC machine (in).

Cam Thickness This parameter is the thickness of the material used to manufacture the cam, and, for CNC code generation, is also used as the depth to which the cam is cut (in).

Feedrate Feedrate is the rate at which the workpiece is moved during machining (in/min).

Spindle Speed The spindle speed is rotational speed of the cutter (rpm).

CNC Home Position Offset The CNC home position offset may be changed if the CNC home position does not coincide with the desired location of the cam center. As shown in Figure 6.6, the home position offset is measured from the old home position to the new home position. It is important that these parameters be chosen properly, incorrect selection can cause damage to the tools and CNC machine.

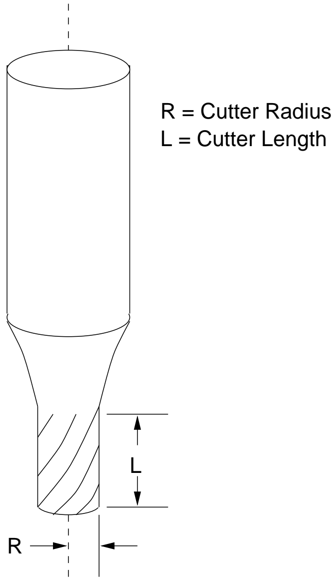


Figure 6.5: Cutter dimensions

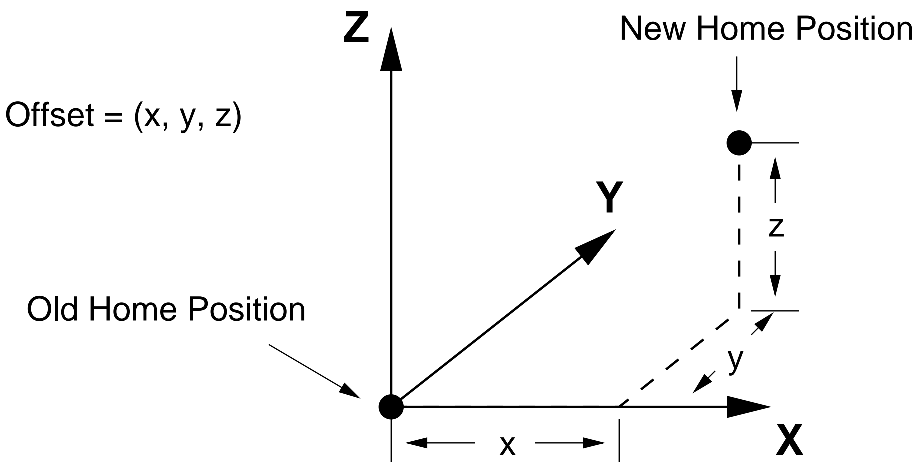


Figure 6.6: Home position offset

6.2 Cam Design with Class CCam

As with the classes for analyzing and designing linkage mechanisms, class **CCam** can be used to aid in the design of cam/follower systems. After specifying the parameters for the cam mechanism, various member functions can be called to plot or obtain data for the cam profile, follower position, follower velocity, follower acceleration, and transmission angle. Furthermore, CNC code can be produced to manufacture the desired cam. Animation of the cam/follower system may also be generated with class **CCam**.

The base radius and angular velocity of the cam can be specified by member function **baseRadius**() and **angularVel**(), respectively. Their function prototypes are as follows.

```
void CCam::baseRadius(double base_radius);
void CCam::angularVel(double omega);
```

Member function **followerType**() is used to indicate the follower type and specify the parameter(s) associated with the given type. The follower may be a flat-face or roller type and translating or oscillating. The function prototype for member function **followerType**() is shown below.

```
int CCam::followerType(int follower_type, ...
                        /* [double e],
                           [double e, double rf],
                           [double m, double f],
                           [double m, double A, double rf] */);
```

Argument `follower_type` may be either **CAM_FOLLOWER_TRANS_FLAT**, **CAM_FOLLOWER_TRANS_ROLL**, **CAM_FOLLOWER_OSC_FLAT**, or **CAM_FOLLOWER_OSC_ROLL** for a translating flat-face follower, translating roller follower, oscillating flat-face follower, and oscillating roller follower, respectively. For a translating follower, parameter `e` specifies the distance from the cam center to the line of follower motion. If the translating follower is a roller type, then parameter `rf` is used to indicate its radius as well. For an oscillating flat-face follower, parameter `m` specifies the distance between the cam center and the follower pivot point, whereas parameter `f` is the value of the follower face offset measured from the follower pivot point. If the follower is an oscillating roller type, then parameters `m`, `A`, and `rf` needs to be indicated, where `A` is the length of the arm connecting the pivot point and roller center.

Another member function of class **CCam** is **addSection**(), which is called to add a cam section to a previously declared instance of the cam class. It is prototyped as follows.

```
int CCam::addSection(double duration, double displacement,
                     int motion_type);
```

Argument `duration` is the angular duration of the cam section in degrees, and `displacement` is the change in position of the cam follower. A positive value indicates that the added section is away from the cam center, whereas a negative value means that it is towards the center. The `displacement` value is measured in inches for translating followers and degrees for oscillating followers. The last argument, `motion_type`, is used to describe the shape of the displacement profile for the cam section. The value of `motion_type` may be either one of two macros: **CAM_MOTION_HARMONIC** or **CAM_MOTION_CYCLOIDAL** for harmonic and cycloidal motion, respectively.

The member function that performs all the calculations for generating the cam data is **makeCam**(). This function shall be called after all the desired cam parameters have been set and prior to any output function calls. The function prototype for member function **makeCam**() is shown below.

```
int CCam::makeCam(int steps);
```

Argument `steps` indicates the number of steps to use in calculating cam results, such as the cam profile. The cam results are stored internally within arrays which may be later accessed by the user or for plotting purposes. Furthermore, if a file name was specified by member function **CNCCode**(), then calling **makeCam**() would generate and store CNC code for manufacturing the cam into a file specified by the given file name. Other results that member function **makeCam**() calculates are the cam profile, follower position, follower velocity, follower acceleration, and transmission angle. For example, consider the following problem statement.

> **Problem 1:** Using class **CCam**, generate a cam profile for a translating flat-face follower. The cam should have a base radius of 2.25 inches, no follower offset, and an angular velocity of 1 rad/s. During the first 90° of cam rotation the follower should move outward 0.75 inches with harmonic motion. During the next 90° the follower should move inward 0.75 inches with harmonic motion. For the remainder of the cam rotation the follower should not change position. 360 points should be used to calculate the results. Generate plots for the follower position, follower velocity, follower acceleration, the transmission angle, and the cam profile. Also generate an animation of the cam using 12 positions and produce CNC code using the following parameters. The cutter has a radius of 0.25 inches and a length of 0.75 inches. The spindle speed should be set to 4000 RPM and the feedrate should be 15 inches/minute. The thickness of the cam is 0.375 inches. No home position offset is used.

Problem 1 defines a cam mechanism with a translating flat-face follower. Given a set of parameters, the program requires various outputs, including plot of the cam profile, CNC code, and animation of the cam/follower system. The solution for Problem 1 is listed as Program 39. Member function **uscUnit**() is called prior to any of the other member functions to indicate that the cam analysis needs to be performed with US Customary units rather than SI units. Note that member functions **cutter**(), **spindleSpeed**(), **feedrate**(), **cutDepth**(), and **cutterOffset**() are used to set the cutter parameters, spindle speed, feedrate, cut depth, and cutter home position offset for CNC code generation, respectively. The cutter parameters include the radius and length, in feet, of the cutter as well as the tool number of the cutter used. The function prototypes for each of the above member functions are shown below.

```
void CCam::cutter(double cutter_radius, double cutter_length,
                  int tool_num);
void CCam::spindleSpeed(double spindle_speed);
void CCam::feedrate(double feedrate);
void CCam::cutDepth(double cut_depth);
void CCam::cutterOffset(double x_offset, double y_offset,
                        double z_offset);
```

Member functions **plotCamProfile**(), **plotFollowerPos**(), **plotFollowerVel**(), **plotFollowerAccel**(), and **plotTransAngle**() are called to output the desired plots. Each plotting function requires a single argument, which is a pointer to an object of class **CPlot**, or **CPlot ***. For simulating the motion of the cam mechanism, member function **animation**() is used. Its functionality and syntax is similar to those of the classes for four-bar and other linkage mechanisms. The difference is that its required input argument indicates the number of frames to generate for the animation. Member function **animation**() may also have an additional second and/or third input argument. The second optional argument is one of the following macros:
QANIMATE_OUTPUTTYPE_DISPLAY, **QANIMATE_OUTPUTTYPE_FILE**, or
QANIMATE_OUTPUTTYPE_STREAM for sending the animation to the monitor screen, saving the animation data to a file, or streaming the animation to the standard output. If the second optional argument is **QANIMATE_OUTPUTTYPE_FILE**, a string may be entered as the third optional argument to specify the file name to store the animation data. The outputs of Problem 1 are listed as Figures 6.7-6.13, where Figure 6.13 is an instance of the cam animation. Figure 6.14 shows a cam manufactured from the CNC code generated by Program 39.

Problem 2: Using class **CCam**, generate a cam profile for a translating flat-face follower. The cam should have a base radius of 5.72 cm, no follower offset, and an angular velocity of 1 rad/s. During the first 90° of cam rotation the follower should move outward 1.91 cm with harmonic motion. During the next 90° the follower should move inward 1.91 cm with harmonic motion. For the remainder of the cam rotation the follower should not change position. 360 points should be used to calculate the results. Generate plots for the follower position, follower velocity, follower acceleration, the transmission angle, and the cam profile. Also generate an animation of the cam using 12 positions and produce CNC code using the following parameters. The cutter has a radius of 0.64 cm and a length of 1.91 cm. The spindle speed should be set to 4000 RPM and the feedrate should be 38.10 cm/minute. The thickness of the cam is 0.95 cm. No home position offset is used.

The above problem statement is similar to that of Problem 1. The difference is that parameters for the cam mechanism is defined in SI units instead of US Customary units. The solution to Problem 2 is Program 40. The program's outputs are equivalent to the outputs of Program 39 in SI units.

Problem 3: Using class **CCam**, generate a cam profile for an oscillating roller follower. The base radius of the cam should be four inches. The follower should have a radius of two inches, should have an arm length of 12 inches and should be located 10 inches from the cam center. 400 points and an angular velocity of 1 rad/s should be used. During the first 120° of cam rotation the follower should move out five degrees with cycloidal motion. During the next 120° the follower should move back to its original position with cycloidal motion. The follower will remain at rest for the remainder of the cam rotation. Plot the follower position, follower velocity, follower acceleration, transmission angle and the cam profile. Animate the cam/follower system.

```
#include <cam.h>

int main() {
    class CCam cam;
    class CPlot plotc, plotp, plotv, plota, plott;
    int steps = 360;
    double x[steps+1], y[steps+1];
    int i;

    cam.uscUnit(true);
    cam.followerType(CAM_FOLLOWER_TRANS_FLAT, 0);
    cam.baseRadius(2.25/12.0);
    cam.angularVel(1);
    cam.addSection(90, .75/12.0, CAM_MOTION_HARMONIC);
    cam.addSection(90, -.75/12.0, CAM_MOTION_HARMONIC);
    cam.addSection(CAM_DURATION_FILL, 0, CAM_MOTION_HARMONIC);
    cam.CNCCode("cam_code.nc");
    cam.cutter(.25/12.0, .75/12.0, 1);
    cam.spindleSpeed(4000);
    cam.feedrate(15/12.0);
    cam.cutDepth(.375/12.0);
    cam.cutterOffset(0, 0, 0);
    cam.makeCam(steps);
    cam.plotFollowerPos(&plotp);
    cam.plotFollowerVel(&plotv);
    cam.plotFollowerAccel(&plota);
    cam.plotTransAngle(&plott);
    cam.plotCamProfile(&plotc);
    cam.animation(12);
}
```

Program 39: Ch program for Problem 1.

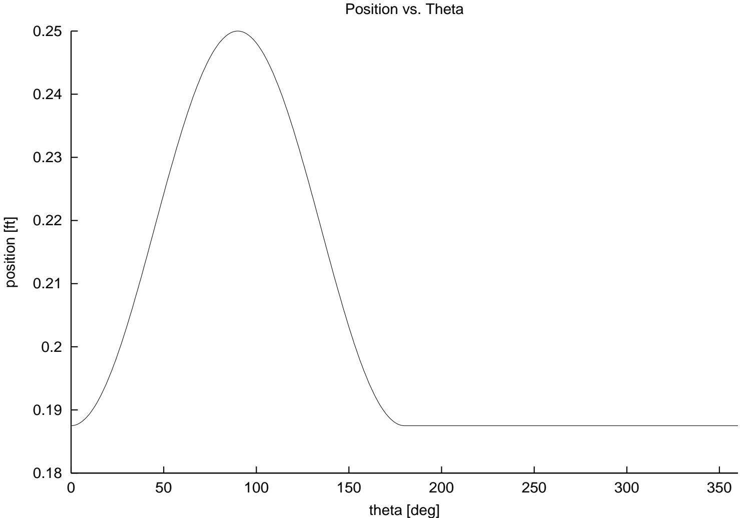


Figure 6.7: Translating follower position.

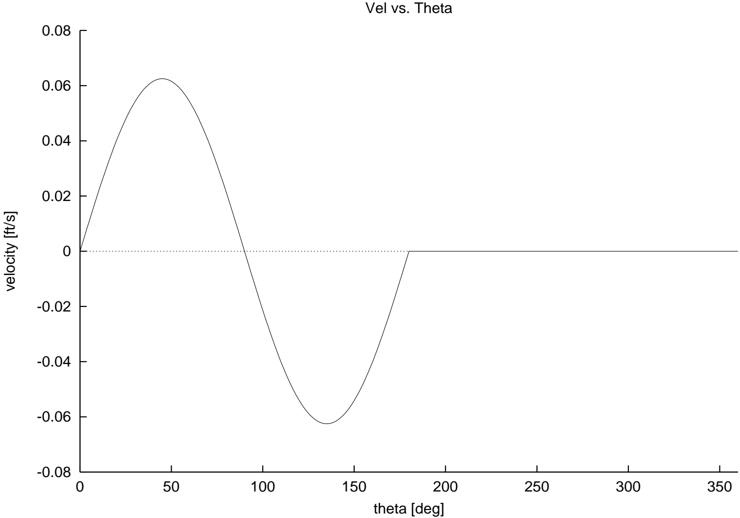


Figure 6.8: Translating follower velocity.

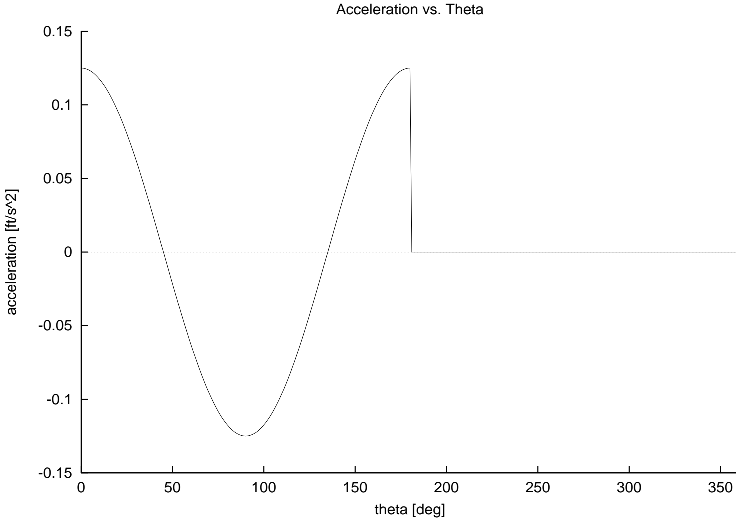


Figure 6.9: Translating follower acceleration.

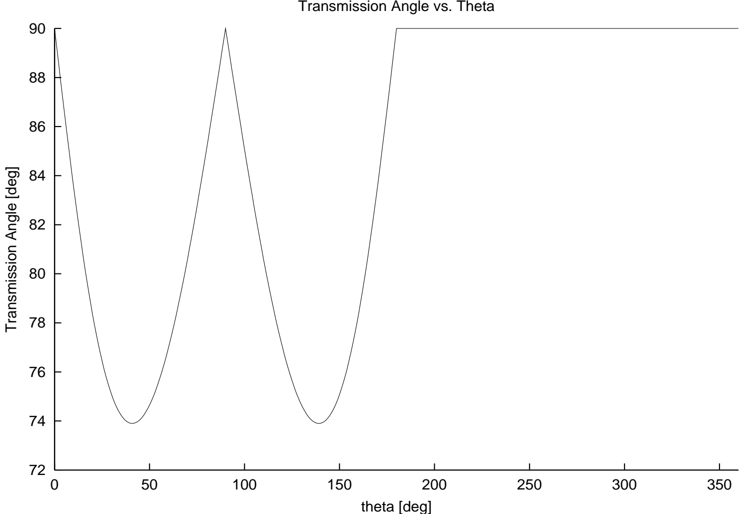


Figure 6.10: Transmission angle of the cam.

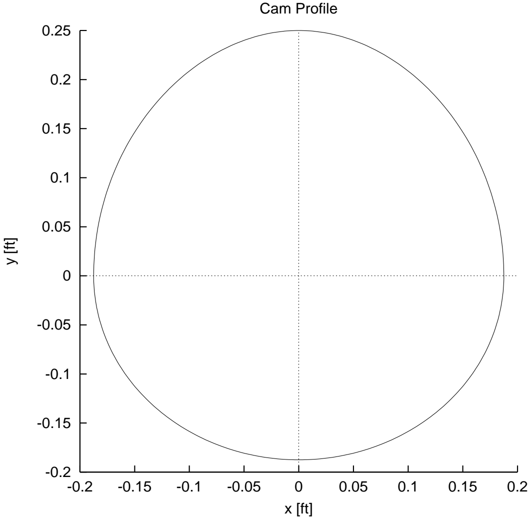


Figure 6.11: Cam profile.

```
N10 G90 G00 X0.000 Y0.000 Z0.000; move to home
N20 M08; coolant on
N25 S0 M3; spindle on
N30 X0.000 Y0.000; move to starting point
N35 G01 Z-0.010 F0.381;
N40 X0.064 Y0.000
N45 X0.063 Y0.002
N50 X0.063 Y0.004
N55 X0.063 Y0.005
N60 X0.063 Y0.007
N65 X0.063 Y0.009
N70 X0.063 Y0.011
N75 X0.063 Y0.012
N80 X0.063 Y0.014
N85 X0.062 Y0.016
N90 X0.062 Y0.018
N95 X0.062 Y0.019
N100 X0.061 Y0.021
...
N1830 X0.063 Y-0.002
N1835 X0.063 Y-0.001
N1840 X0.064 Y-0.000
N1845 G90 G01 Z0.000;
N1850 M05; stop spindle
N1855 M09; coolant off
N1860 G90 X0.000 Y0.000;
N1865 M22
```

Figure 6.12: CNC code for manufacturing the cam shown in Figure 6.14

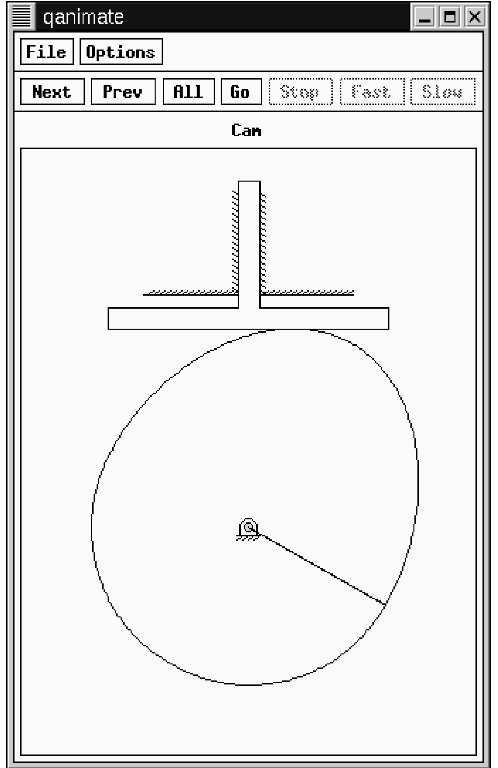


Figure 6.13: Animation of cam with translating follower.

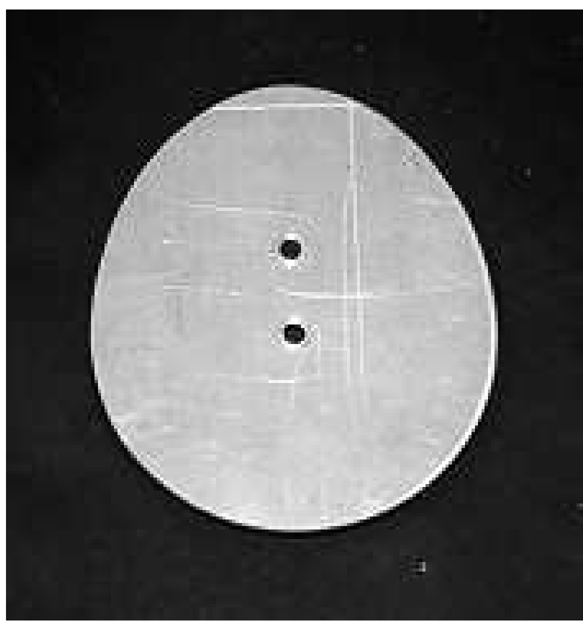


Figure 6.14: Manufactured cam.

```
#include <cam.h>

int main() {
    class CCam cam;
    class CPlot plotc, plotp, plotv, plota, plott;
    int steps = 360;
    double x[steps+1], y[steps+1];
    int i;

    cam.uscUnit(false);
    cam.followerType(CAM_FOLLOWER_TRANS_FLAT, 0);
    cam.baseRadius(0.0572);
    cam.angularVel(1);
    cam.addSection(90, 0.0191, CAM_MOTION_HARMONIC);
    cam.addSection(90, -0.0191, CAM_MOTION_HARMONIC);
    cam.addSection(CAM_DURATION_FILL, 0, CAM_MOTION_HARMONIC);
    cam.CNCCode("cam_code.nc_test");
    cam.cutter(0.0064, 0.0191, 0.0254);
    cam.spindleSpeed(4000);
    cam.feedrate(0.3810);
    cam.cutDepth(0.0095);
    cam.cutterOffset(0, 0, 0);
    cam.makeCam(steps);
    cam.plotFollowerPos(&plotp);
    cam.plotFollowerVel(&plotv);
    cam.plotFollowerAccel(&plota);
    cam.plotTransAngle(&plott);
    cam.plotCamProfile(&plotc);
    cam.animation(12);
}
```

Program 40: Ch program for Problem 1 using SI units.

```
#include <cam.h>

int main() {
    class CCam cam;
    class CPlot plotc, plotp, plotv, plota, plott;

    cam.uscUnit(true);
    cam.followerType(CAM_FOLLOWER_OSC_ROLL, 10/12.0, 12/12.0, 2/12.0);
    cam.baseRadius(4/12.0);
    cam.angularVel(1);
    cam.addSection(120, 5, CAM_MOTION_CYCLOIDAL);
    cam.addSection(120, -5, CAM_MOTION_CYCLOIDAL);
    cam.addSection(CAM_DURATION_FILL, 0, CAM_MOTION_CYCLOIDAL);
    cam.spindleSpeed(4000);
    cam.makeCam(400);
    cam.plotFollowerPos(&plotp);
    cam.plotFollowerVel(&plotv);
    cam.plotFollowerAccel(&plota);
    cam.plotTransAngle(&plott);
    cam.plotCamProfile(&plotc);
    cam.animation(12);
}
```

Program 41: Ch program for Problem 2.

Another example to consider is Problem 3. This problem statement is similar to Problem 1 since it requires the same types of outputs. However, the cam being specified consists of an oscillating roller follower, which means that the parameters to set up are slightly different. For example, member function **followerType**() now requires a few additional parameters to clearly define the oscillating roller follower. Likewise, cycloidal motion needs to be specified when calling the **addSection**() member function. The number of steps for calculating the cam results is now 400 instead of 360 according to the previous problem statement. The solution to Problem 3 is Program 41. The outputs for this program are Figures 6.15-6.20.

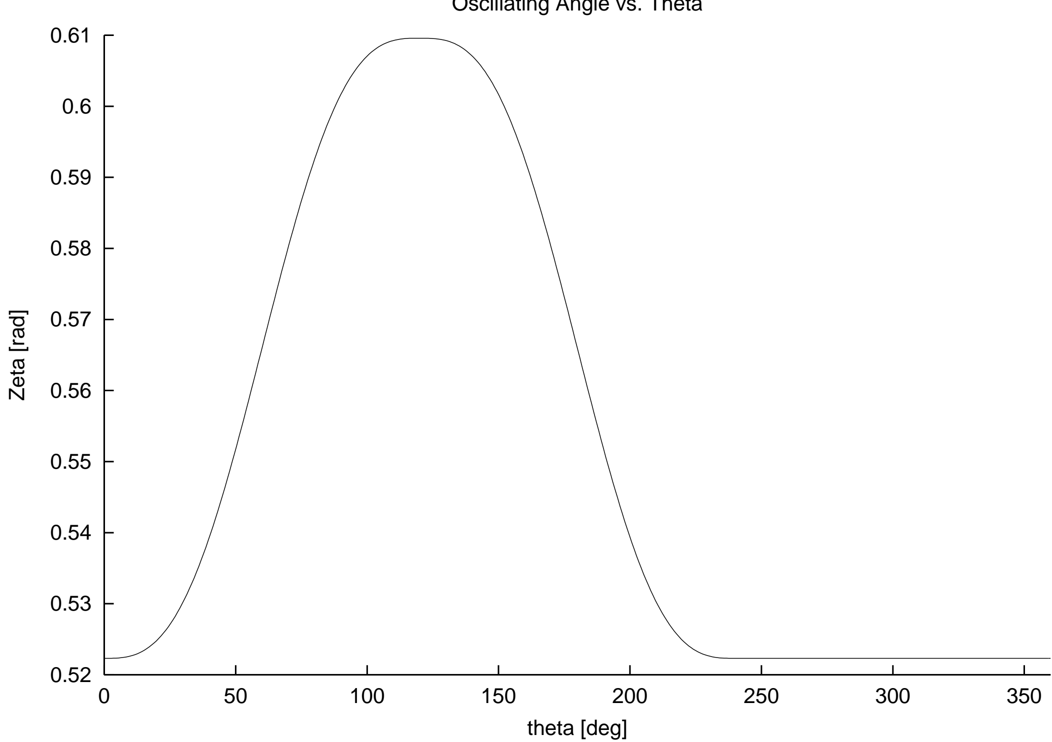


Figure 6.15: Oscillating follower position.

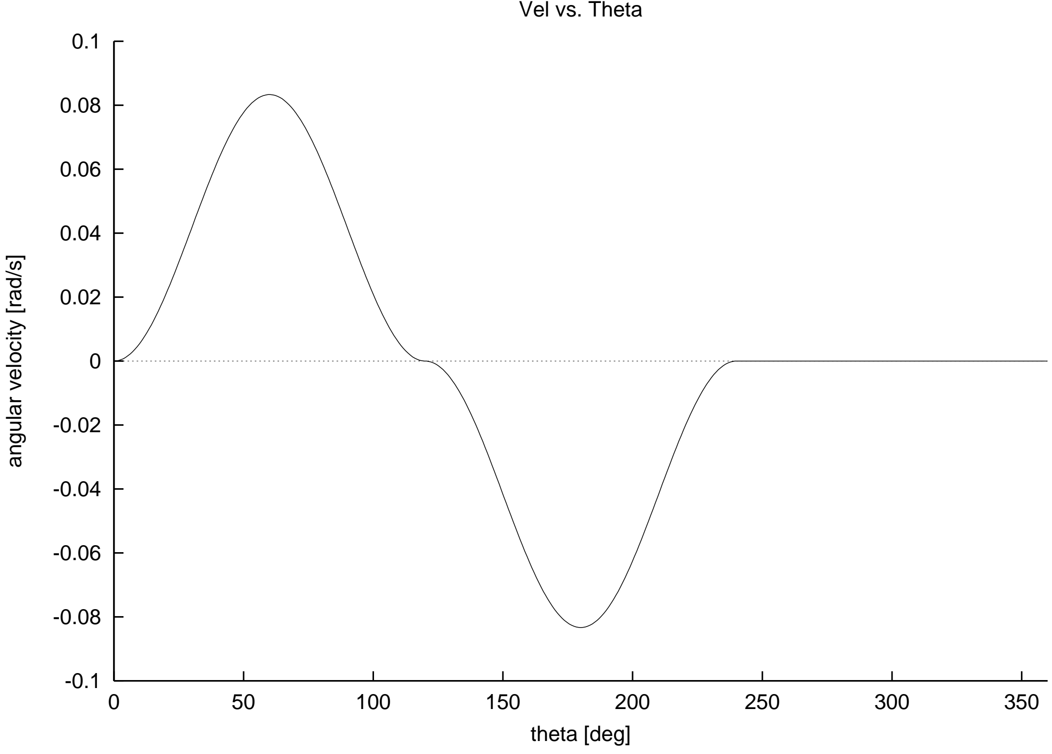


Figure 6.16: Oscillating follower velocity.

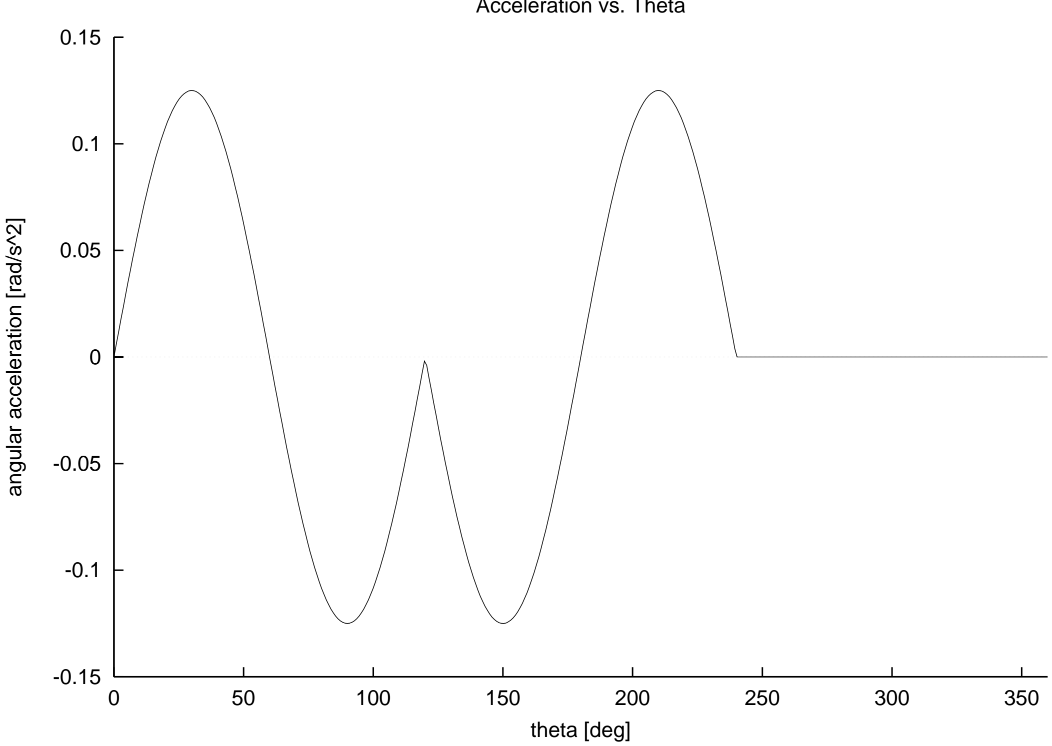


Figure 6.17: Oscillating follower acceleration.

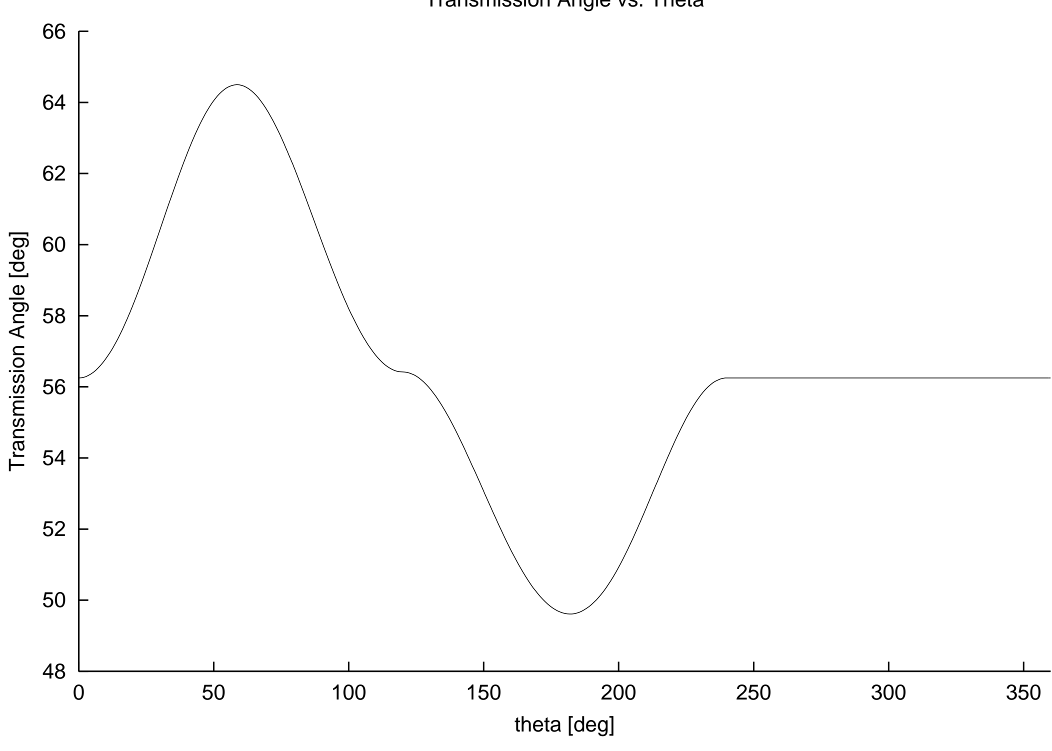


Figure 6.18: Transmission angle of the cam.

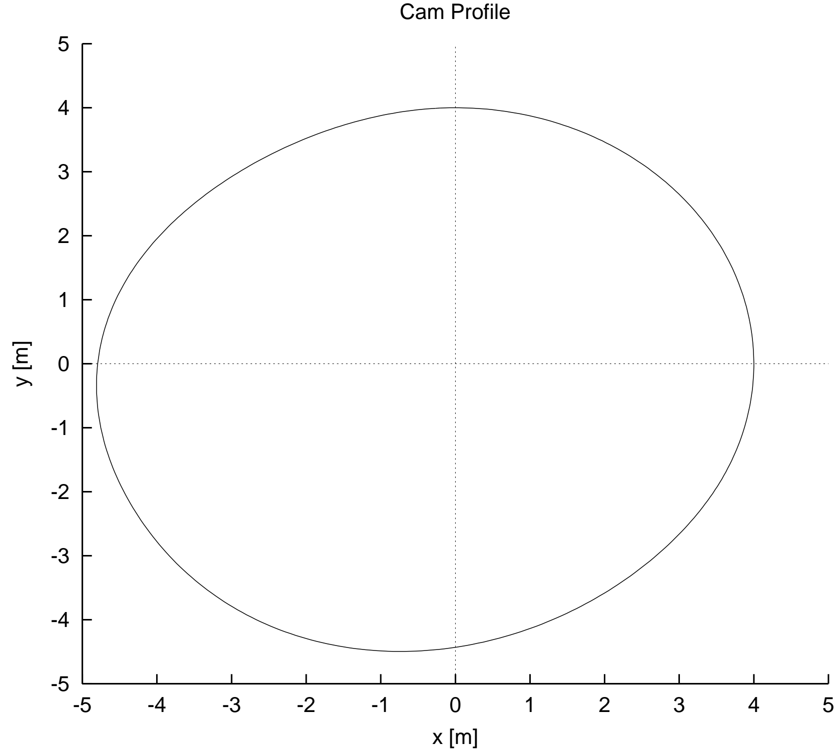


Figure 6.19: Cam profile.

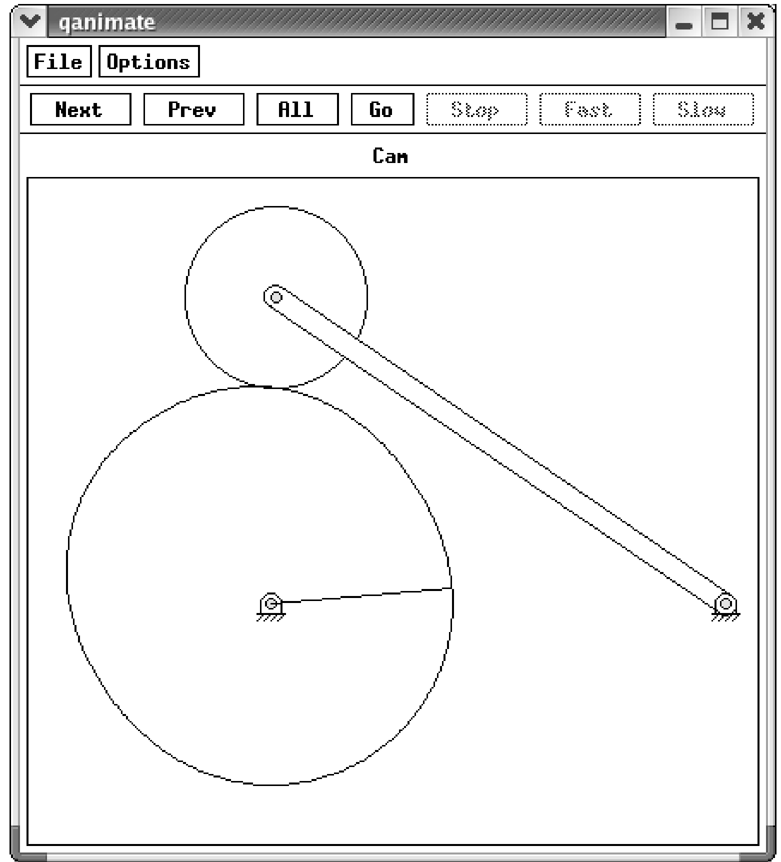


Figure 6.20: Animation of cam with Oscillating follower.

6.3 Web-Based Cam Design

User Interface

The web-based user interface provides a quick and easy means of generating a cam profile. The web interface consists of a number of web pages, some of which are created dynamically, to guide the user through the creation of the cam.

As shown in Figure 6.21, the first of these pages allows the user to select the type of cam follower motion that is desired. These choices are presented in a graphical format representing a flat face radially translating follower, which converts rotational motion to linear motion, and a flat faced oscillating follower, which converts rotational motion to angular motion.

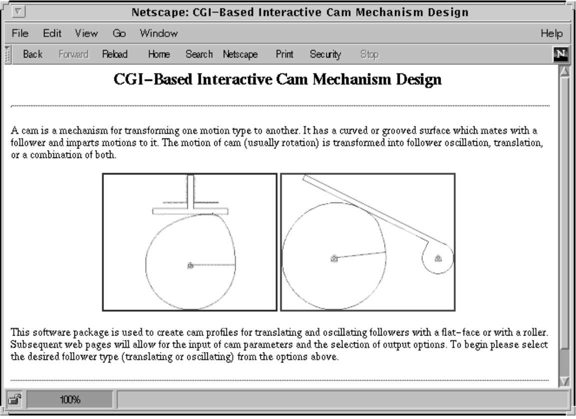


Figure 6.21: Opening cam synthesis page

After one of these choices is selected, the user is next prompted to input the follower type and the number of cam sections (a change in follower position or a dwell) that is desired, as shown in Figure 6.22. Alternatively, the user may select a sample cam design with all parameters preselected.

Figures 6.23 and 6.24 show the subsequent page where the user can input the cam parameters, some of which are dependent on the follower type, and the CNC cutter parameters. Note that Figures 6.23, 6.24, and 6.25 are the same Web page. All cam parameters must be set for the program to work properly. However, if the user does not want the CNC code for manufacturing the designed cam as an output, no modifications to the CNC parameters are needed, as they do not affect the calculations of the cam profile.

Following selection of the cam parameters, as shown in Figure 6.25, the output options are chosen. The output options include: plots of cam profile, follower position, follower velocity, and follower acceleration, transmission angle, CNC code for manufacturing of the designed cam, and animation of the cam/follower system.

When the cam parameters are submitted to the WWW server, they are sent to CGI programs `cgi_osc_results.ch` or `cgi_trans_results.ch`, depending on the follower type, and processed. These CGI programs then execute a second CGI program named `cgi_make_osc_cam.ch` or `cgi_make_trans_cam.ch` which in turn executes `cam.ch` on the server and sends it the processed parameters. The `cam` program then performs the calculations for cam design and generates the data for the desired output. Plots of the results, CNC code, and animations can be viewed, but only one of them can be viewed for each submission to the `cam` program.

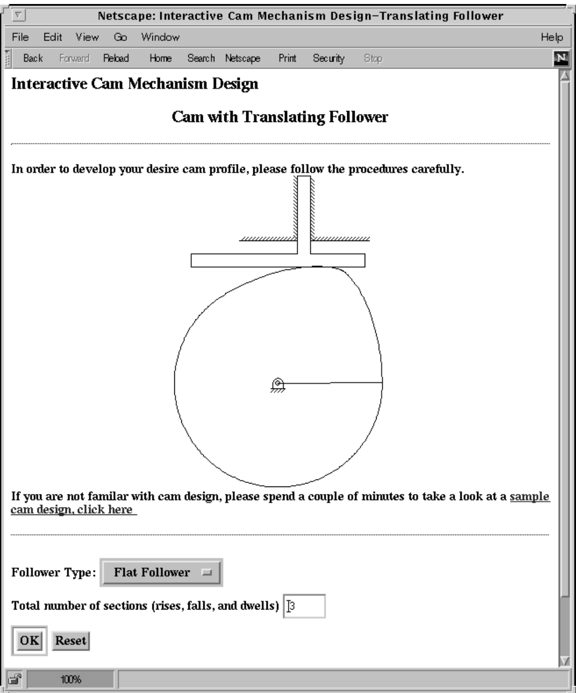


Figure 6.22: Translating cam page

Plots of the cam profile, follower position, follower velocity, follower acceleration, and transmission angle are sent to the browser. These plots are also generated using the Ch CPlot class. The `cam.ch` program passes the data to the CPlot class with an option set for generating an image as the standard output stream. The Web browser then takes the data from this stream and displays it directly in the browser window. The CNC code, if selected, is also displayed in the browser window, where it may be saved for later use.

Sample Cam Generation

In this section two sample cam design problems will be presented to illustrate the features and applications of Web-based cam design.

Problem 4: Repeat Problem 1 using the CGI-based cam design Web pages.

Problem 1 required the generation of plots for the follower position, follower velocity, and follower acceleration, transmission angle, as well as a plot for the cam profile. It also asked for the output of the CNC code and animation of the follower and cam. The Web pages used for entry of the parameters for this problem are shown in Figures 6.21– 6.25 in the previous sections. The outputs are identical to those for Problem 1.

Problem 5: Using the cam design web pages, generate a cam profile for an oscillating roller follower. The parameters for this cam/follower system is the same as those defined for the system in Problem 3. Also plot the follower position, follower velocity, follower acceleration, transmission angle and the cam profile as well as animate the cam/follower system.

The Web pages for entry of the oscillating follower parameters are similar to those for the translating follower. The primary difference is that the oscillating follower has a few additional parameters. To produce the desired output, the parameters must be submitted a total of six times, once for each plot and once for the animation. The resulting plots of follower position, follower velocity, and follower acceleration are shown

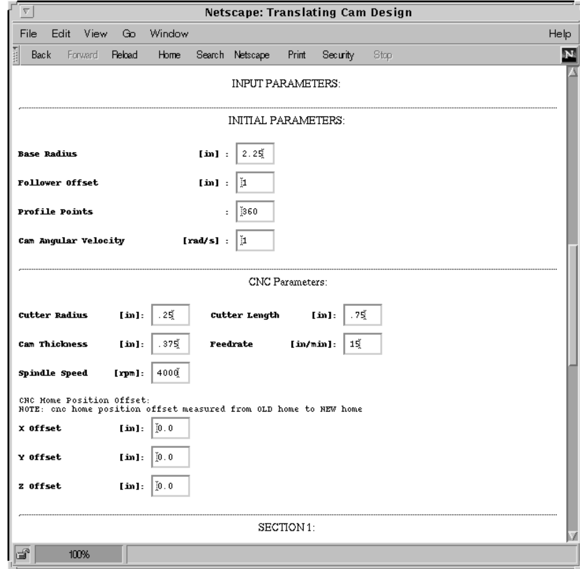


Figure 6.23: Base Radius and CNC Manufacturing Parameters

in Figures 6.26–6.28. The plot of the transmission angle is shown in Figure 6.29. The plot in Figure 6.30 shows the cam profile and the animation is shown in Figure 6.31.

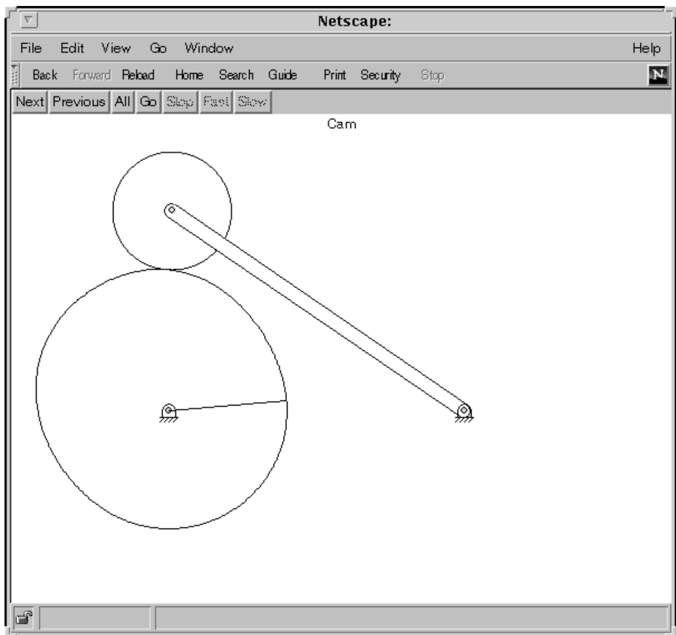


Figure 6.31: Animation of cam with Oscillating follower.

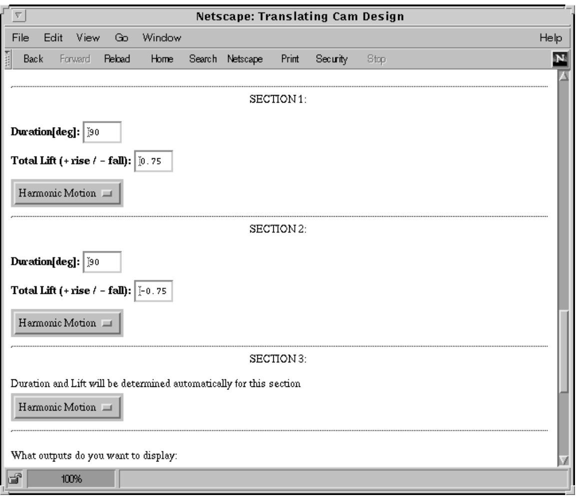


Figure 6.24: Section Parameters

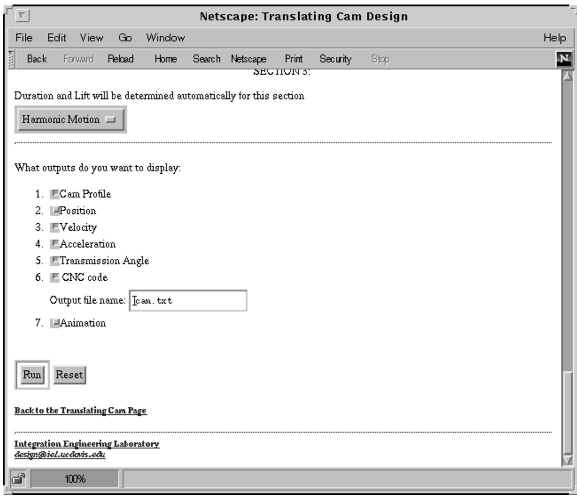


Figure 6.25: Translating cam output options

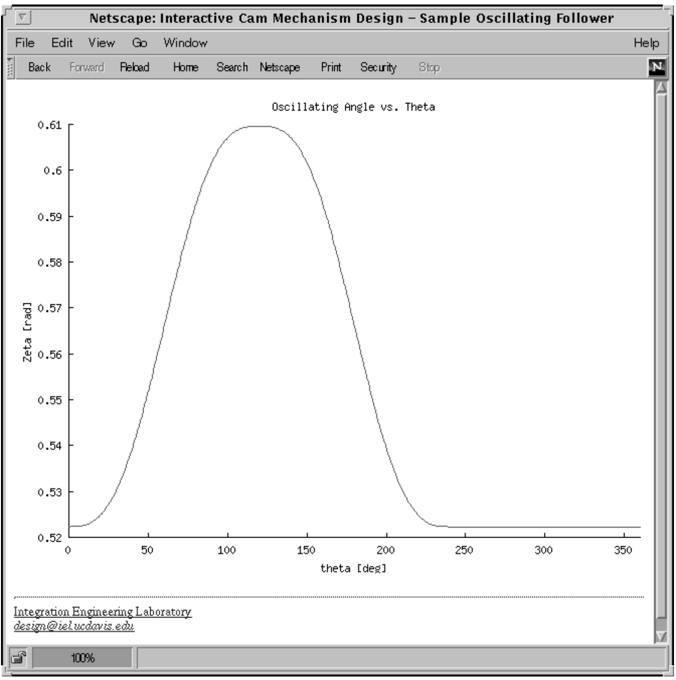


Figure 6.26: Oscillating follower position.

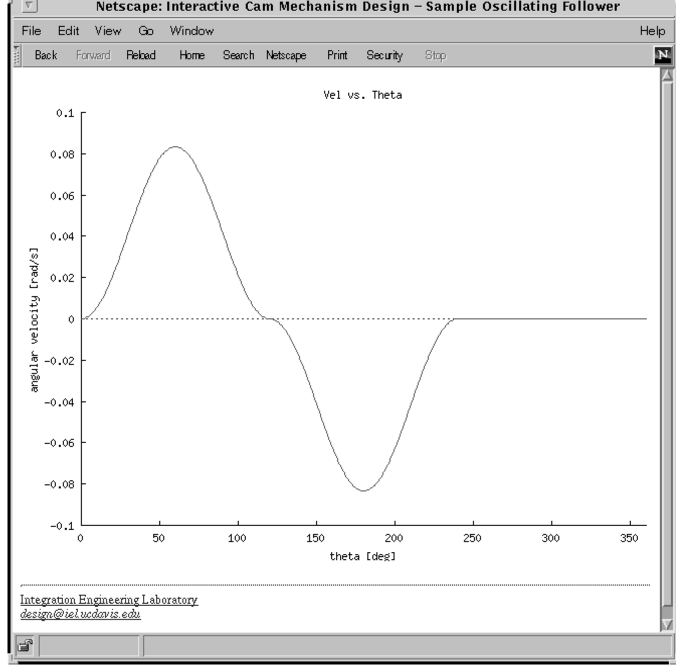


Figure 6.27: Oscillating follower velocity.

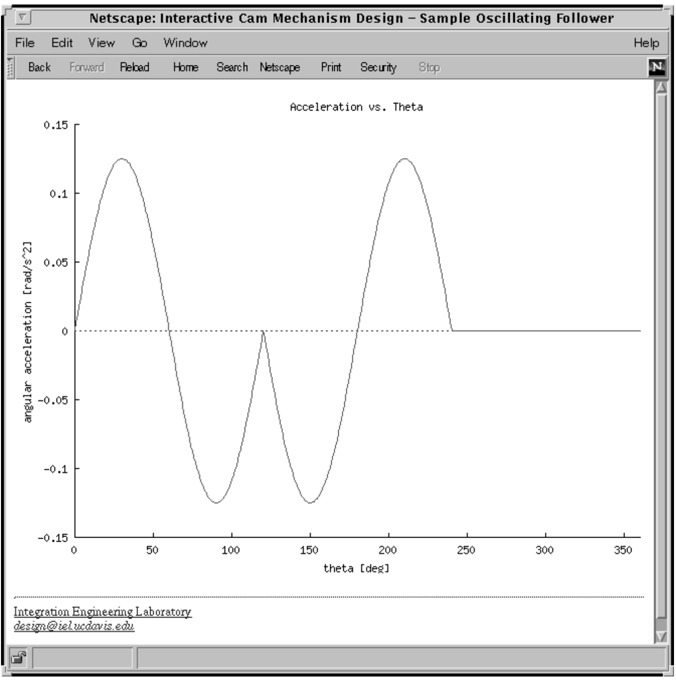


Figure 6.28: Oscillating follower acceleration.

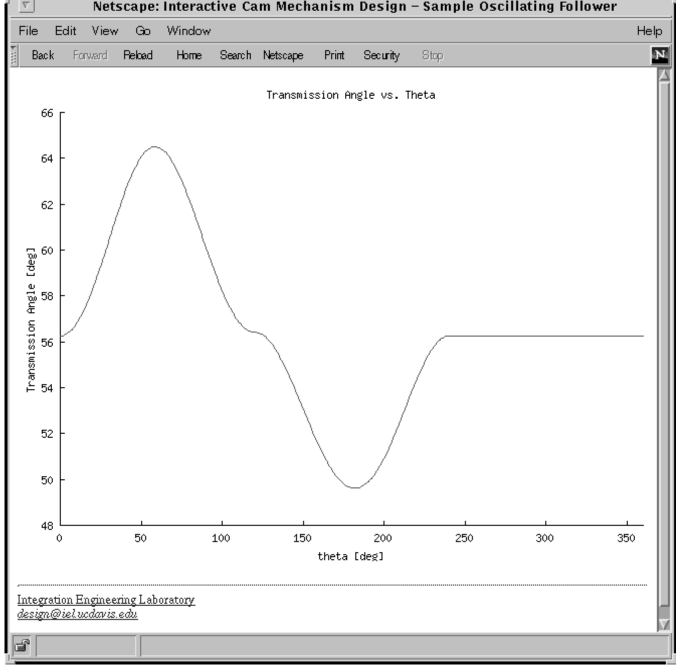


Figure 6.29: Transmission angle of the cam.

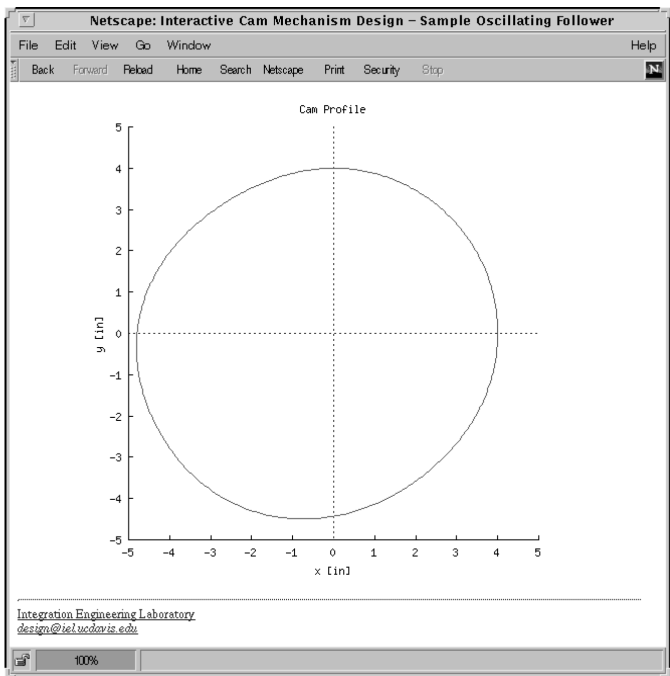


Figure 6.30: Cam profile.

Chapter 7

Quick Animation

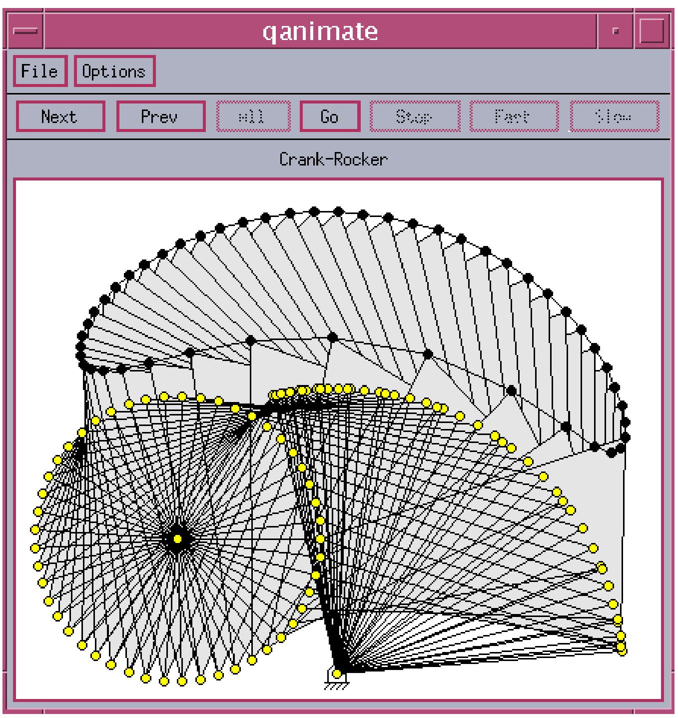


Figure 7.1: Quick Animation window showing a four-bar linkage

All animations of Ch Mechanism Toolkit are handled by QuickAnimation™. QuickAnimation™ can display and animate various planar mechanical systems based on specified x-y coordinate data. For example, the QuickAnimation™ window shown in Figure 7.1 displays a menu bar and a four-bar mechanism. The mechanism is drawn in the largest area of the window with the name of the mechanism above it. Links are drawn as lines, joints are drawn as open circles, and coupler points are drawn as filled circles. The triangular area represents one solid link to which a coupler point is attached. The coupler curve is drawn to indicate the motion of the coupler point. The details on how to use the QuickAnimation™ program of `qanimate` for animation of planar mechanisms are described in this chapter.

7.1 Input Data Format

The typical format for a Quick Animation data file is displayed in Figure 7.2. It is specified with the following typographical notation:

- `Typewriter` text specifies actual keywords.
- *`Emphasized`* text is specified by the user.
- Text between square brackets '`[ ]`' are optional.
- The line character '`|`' specifes an "OR" condition.

The character '`#`' on the first line delimits a comment. Quick Animation will ignore anything on that line following the '`#`' character. The title of the mechanical system is specified by the `title` keyword followed by the title string delimited by the double quotation character, '`"`'. Keyword `fixture` allows the following line to define how the mechanical system will be fixed. `primitives` are commands used to define general mechanical components of the animated system. `animate` begins the inputting of of data for animation. Each line following keyword `animate` represents one position of the mechanical system, as indicated by the superscript on `primitive`. The `primitives` following the keyword `stopped` will be displayed only when animation is stopped.

```
# comment
title "title string"
fixture
primitives
animate [ restart | reverse ]
primitives^1 [ stopped primitives^1 ]
primitives^2 [ stopped primitives^2 ]

                  .
                  .
                  .

primitives^n [ stopped primitives^n ]
```

Figure 7.2: Quick Animation data format.

7.1.1 General Drawing Primitives

General drawing primitives were built into Quick Animation for ease of creating typical components, such as the springs and joints, of a mechanical system. Figure 7.3 shows the various drawing primitives available for Quick Animation. These primitives allow for the drawing of an arc, line, segment, circle, polygon, and rectangle as well as the insertion of text into a Quick Animation program. The syntax for drawing such primitives are displayed in Figure 7.4. As an example, consider the sytax for drawing a line. One may specify a line by typing `line` followed by the x- and y-coordinates of the starting and ending points of the line (i.e. `line 0 0 2 3` draws a line from the origin to point (2,3) in the Cartesian coordinate system). Multiple lines may be linked together by adding more coordinate points after the `line` statement. Similarly, a circle may be drawn by specifying its center point and radius according to the syntax in Figure 7.4. The various options available for each general drawing primitives are displayed in Figure 7.5, and an example of color and font options is listed in Figure 7.6.

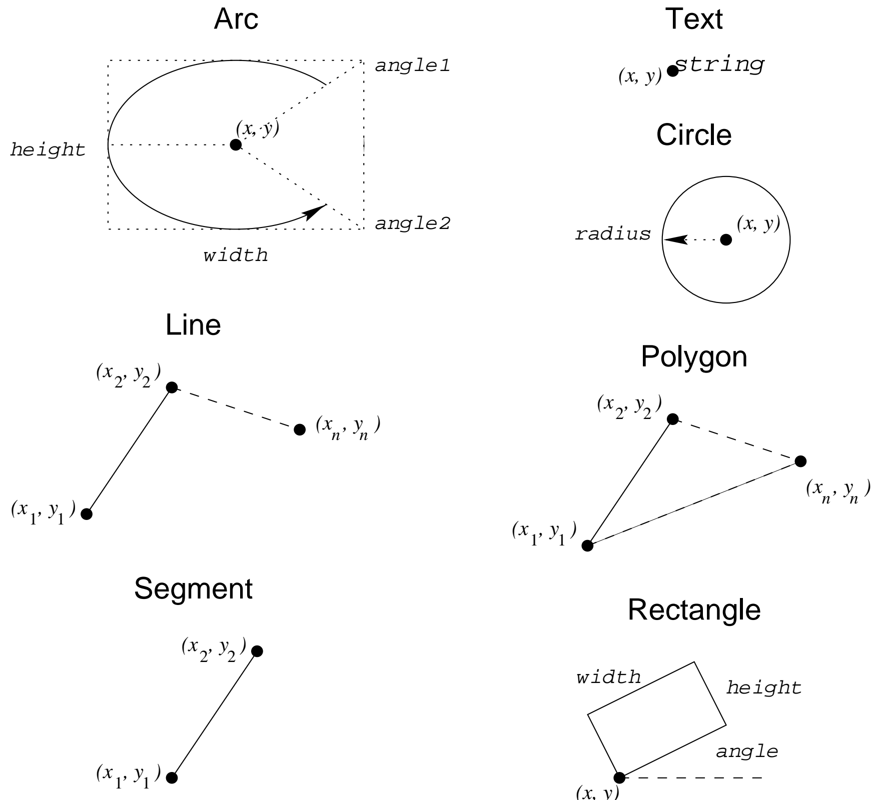


Figure 7.3: Graphical representation of general drawing primitives

```
line x1 y1  x2 y2  [ ...  xn yn ]
arc x y width height angle1 angle2
segment x1 y1 x2 y2
rectangle x y width height [ angle angle ]
polygon x1 y1  x2 y2  x3 y3  ...  xn yn
text x y string
circle x y radius
dot x y
```

Figure 7.4: Syntax for general drawing primitives

```
line
segment   ...
          [ pen color ]
          [ linewidth pixelwidth ]
          [ linestyle solid |
                      dashed [ length pixellength ] |
                      dotted [ gap pixelgap ] ]
          [ capstyle butt | round | projecting ]
          [ joinstyle miter | round | bevel ]
          [ depth depth ]
arc
circle
polygon
rectangle ...
          [ pen color ]
          [ fill color [ intensity percent ]
                       [ pattern number ] ]
          [ linewidth pixelwidth ]
          [ linestyle solid |
                      dashed [ length pixellength ] |
                      dotted [ gap pixelgap ] ]
          [ capstyle butt | round | projecting ]
          [ joinstyle miter | round | bevel ]
          [ depth depth ]
text      ...
          [ pen color ]
          [ depth depth ]
          [ font fontname ]
dot       ...
          [ pen color ]
          [ depth depth ]
```

Figure 7.5: Options for general drawing primitives

```
...  color { red | blue | yellow | white | black | grey90 ...  }
...  font { fixed | 6x13 | 6x13bold | lucidasanstypewriter-12 ...
}
```

Figure 7.6: Example color and font options

Option Notes:

- Colors are specified by the X Window System. A listing of color names can be found in the file `rgb.txt` located in directory `/usr/X11R6/lib/X11` in Linux and `/usr/openwin/lib` in Solaris.
- Fonts are specified by the X Window System.

7.1.2 Mechanical Drawing Primitives

The mechanical drawing primitives available in Quick Animation are derived from the general drawing primitives. For example, a link is a combination of two circles connected by a line. All the available mechanical drawing primitives are shown in Fig. 7.7. These primitives are the primary tools used for creating animations of mechanical systems.

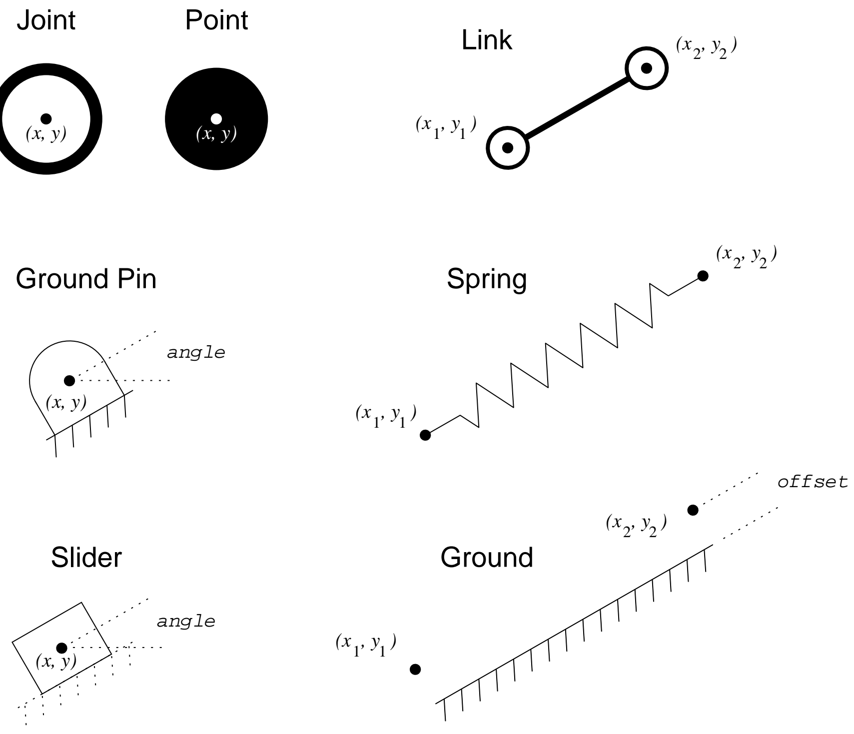


Figure 7.7: Graphical representation of mechanical drawing primitives

Point and Joint

The `point` primitive is basically circle with a filled-in center. It is usually used to emphasize a point on a mechanical system. The general syntax for a `point` is

`point` $x_1\, y_1$ `[`$x_2\, y_2$ `...` $x_n\, y_n$`] [trace]`,

where $x_1 y_1 ... x_n y_n$ specify the coordinate(s) of the joint(s), `trace` is an optional parameter used to specify whether the point is to be traced during animation. For example, to create a point at coordinate (1,3) with a trace the following command would be required:

```
point 1 3 trace.
```

Primitive `joint` is very similar to `point`. It is syntatically the same as the `point` primitive, but is comprised of a circle that is not filled-in. The joint represents a connection between two links or other mechanical components.

Link

As previously mentioned, the `link` primitive is a mechanical component formed by two circle primitives and a line primitive. This primitive is normally used for generating animations of mechanical linkages such as fourbar mechanisms. The general syntax for a `link` is given by the following:

link x_1 x_2 x_2 y_2 [... x_n y_n].

The coordinates of the endpoints of the first link is specified by (x_1,y_1) and (x_2,y_2). Addition links may be attached to the last link by indicating the coordinates of the links' other endpoints. //primitive. A typical example of creating two links adjoined at a common endpoint would be

```
link 1 1  1 4  4 4
```

In this example, the endpoints of the first link are at coordinates (1,1) and (1,4). The second link is then attached to the first link at (1,4), and its other endpoint is located at (4,4). Note that the extra space between the endpoint coordinates are ignored during execution of the Quick Animation program. They are present in the example to help distinguish the endpoints.

Ground

The `ground` primitive represents a reference area of the animation. It is stationary and fixed to its location. The syntax for `ground` is

ground x_1 y_1 x_2 y_2 [offset *pixeloffset*]
[ticks *forward* | *backward*]

For option `offset`, *pixeloffset* specifies the distance that the ground should be placed away for the x- and y-coordinates of the ground. Additionally, if the `ticks` option is used, and its value is "forward", then the ground is specified as going from (x_1,y_1) to (x_2,y_2). Likewise, the opposite is true if the value of `ticks` is "backward". The default value for option `ticks` is "forward". For example,

```
ground 0 0  10 0 offset 2
```

will produce a "ground" section from x=0 to x=10 and two units below the line, y=0.

Ground Pin

In order to directly connect a mechanical system to "ground", the `groundpin` primitive is used to specify the desired connection. The syntax for this primitive is given below as

groundpin *x y* [angle *angle*]

Coordinate (x,y) is the center point of the ground pin, and the optional argument `angle` *angle* describes the angular offset, in radians, relative to a horizontal position. In order to create a ground pin at the origin with a $45°$ rotational offset, the following statement should be declared:

```
groundpin 0 0  angle 3.14159/4
```

Slider

The `slider` primitive is generated from the rectangle drawing primitive. It represents a block member of a mechanical system that is only capable of translation displacement. Similar to the ground pin, the slider can have an angular displacement that would allow it to translate on a sloped surface. Its syntax is defined as follow:

`slider` *x y* `[angle` *angle*`]`

As an example, consider a crank-slider mechanism that requires the slider to slide on a sloped surface, located at (3,4) that is about 30° relative to the ground. The slider portion of the mechanism can be created with the following statement:

```
slider 3 4 angle 3.14159/6
```

Spring

The spring is a typical component of many mechanical systems. The availability of a `spring` primitive in Quick Animation greatly increases the number of mechanical systems that can be modeled and animated. Its syntax is given as

`spring` $x_1\, y_1\;\; x_2\, y_2$,

where coordinates (x_1,y_1) and (x_2,y_2) specifies the endpoints of the spring. To create a spring from (1,1) to (3,5), the following should be entered in the Quick Animation data file:

```
spring 1 1  3 5
```

7.2 Quick Animation Examples

The data file in Figure 7.8 illusrates how low-level and mechanical primitives are specified in a quick animation file. Figure 7.9 show the display when this data file is processed by the program.

The sample code shown in Figure 7.10 can be used to create an animation of a fourbar linkage similar to the one in Figure 7.11. It should be noted that the first line in an animation file must start with #**qanimate**. Next, the title of the animation is set, and the fixtures are specified as ground pins at points (A_x, A_y) and (D_x, D_y). Line `animate restart` is used to begin the input of the animation. Each line beginning with `(line ...)` and ending after `(polygon ...)` represents one snap shot, or one position, of the fourbar linkage animation. The links are drawn with the `link` keyword. Each coordinate pair represents a joint of the mechanism. On each line, the coordinate pairs are arranged in order so that two adjacent joints form a link. The path of the coupler point, specified as point P in Figure 7.11, is shown with a `point trace`. The `polygon` keyword was used to create the coupler attachment. The desired number of frames will determine the number of data sets that should be entered into the Quick Animation data file. Hence, *n* number of frames will require *n* number of data sets.

```
# this is a comment
title "This is a Title"

fixture
#no fixture

animate

# low level primitives:
line 0 0 1 1.5 2 2 pen red \
    line 3 3 4 4
line 5 5 12 5 linestyle dashed length 2 pen green linewidth 1
line 5 6 12 6 linestyle dashed length 5 pen green linewidth 1
line 5 7 12 7 linestyle dotted gap 1 pen red linewidth 2
line 5 8 12 8 linestyle dotted gap 5 pen red linewidth 2
arc 11 11 4 4 0 270 fill grey90 linewidth 5
arc 12 12 10 11 0 90  13 13 5 5 0 360 linewidth 2 pen blue
segment 14 14 15 15  16 16 17 17 pen red
#color of text cannot be changed in Windows for now
text 18 5 string1 pen rgb:ffff/ffff/0
text 18 7 "This is a string2" pen red
text 18 9 "This is a string3" \
    font -*-lucidatypewriter-medium-*-*-*-12-*-*-*-*-*-*-*
circle 22 16 2 \
    stopped line 14 17 17 20 text 17.2 20 "center of circle"
rectangle 15 18 1 1 pen red fill grey
rectangle 17 20 2 1 angle 30

# higher linkage primitives
joint 18 18
point 19 19
link 20 20 21 21
groundpin 22 22 25 25 angle 30
link 22 22 25 25
polygon 4 10  5 10 6 13 3.5 14 fill green
spring 10 1  15 1
ground 17 1.0  19 2.0
# The traced trajectory shown on the upper left
point 0 20 trace
point 3 23 trace
point 6 25 trace
point 10 20 trace
```

Figure 7.8: A sample data file `sample.lnm`.

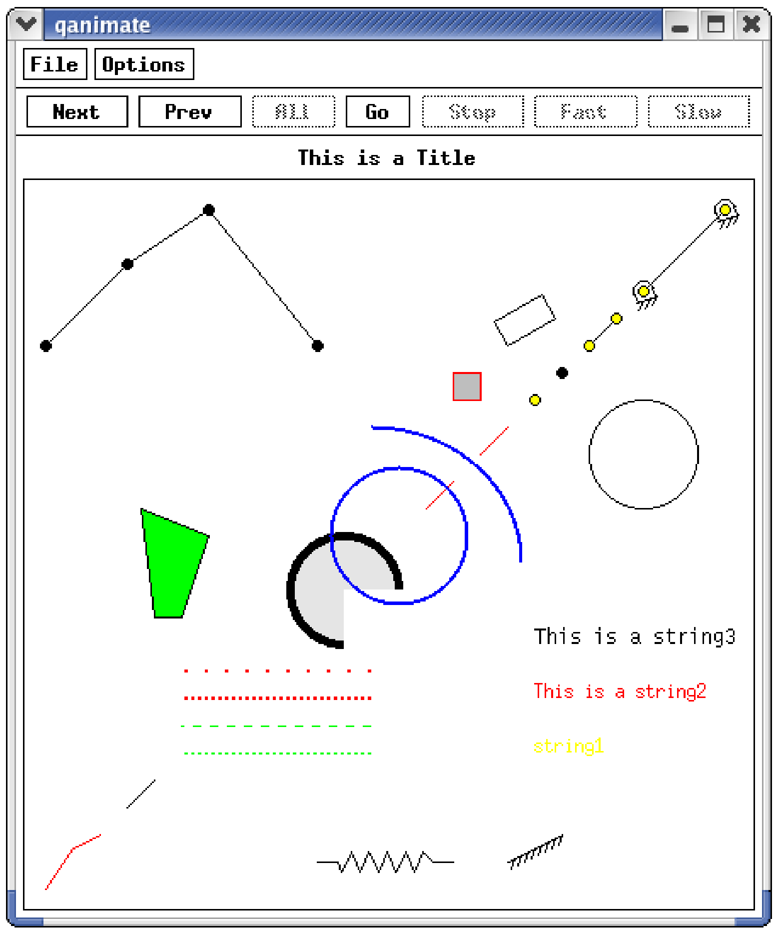


Figure 7.9: The quick animation display based on the sample data file `sample.qnm`.

#qanimate animation file
title "*Crank-Rocker*"

fixture
groundpin $A_x\ A_y \quad D_x\ D_y$

animate restart

link $A_x^1\ A_y^1 \quad B_x^1\ B_y^1 \quad C_x^1\ C_y^1 \quad D_x^1\ D_y^1$ \
point trace $P_x^1\ P_y^1$ \
polygon fill grey $B_x^1\ B_y^1 \quad C_x^1\ C_y^1 \quad P_x^1\ P_y^1$

link $A_x^2\ A_y^2 \quad B_x^2\ B_y^2 \quad C_x^2\ C_y^2 \quad D_x^2\ D_y^2$ \
point trace $P_x^2\ P_y^2$ \
polygon fill grey $B_x^2\ B_y^2 \quad C_x^2\ C_y^2 \quad P_x^2\ P_y^2$

.
.
.

link $A_x^n\ A_y^n \quad B_x^n\ B_y^n \quad C_x^n\ C_y^n \quad D_x^n\ D_y^n$ \
point trace $P_x^n\ P_y^n$ \
polygon fill grey $B_x^n\ B_y^n \quad C_x^n\ C_y^n \quad P_x^n\ P_y^n$

Figure 7.10: Input data format for a four-bar linkage with a single coupler point

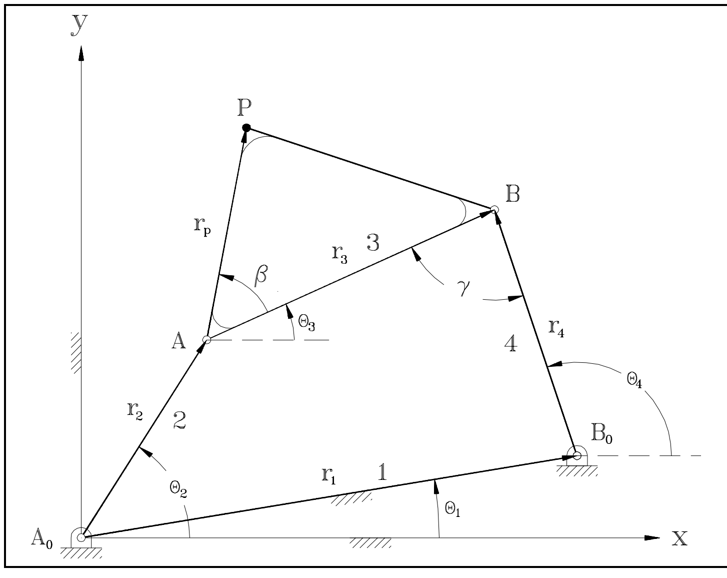


Figure 7.11: General fourbar linkage

Before viewing a complete Ch program for creating a Quick Animation data file, consider the Ch code

listed as Figure 7.12. This Ch code can be executed to create one frame of animation data. First, the variables for the fourbar parameters are declared and initialized. Then **printf()** statements are used to write the typical heading of an animation data file. Note that the links and coupler point vector are represented as complex variables `R` and `P`, where `R` is an array of complex type. The positions of the links and coupler point are calculated by the **complexsolve()** function and vector analysis. Additional calls to function **printf()** completes writing to the animation data file. The resulting animation code of the program shown in Figure 7.12 can be used by the Quick Animation application to generate a single frame of animation, which is shown as Figure 7.13.

```
/* Filename: simple_ani.ch */
#include <stdio.h>
#include <math.h>
#include <complex.h>
#define deg2rad(a) ((a)*M_PI/180.0)       // macros
#define rad2deg(a) ((a)*180.0/M_PI)

int main() {
    double theta1 = deg2rad(20.0),       // given data
           r1 = 5.5,
           r2 = 1.5,
           r3 = 4.5,
           r4 = 5,
           rp = 3,
           beta = deg2rad(25.0);
    int n1 = 2, n2 = 4;
    double theta2,theta3,theta4, theta3_2, theta4_2;
    double complex z;
    double complex R[1:3]; // link vectors
    double complex P; // coupler vector

    R[1] = polar(r1,theta1);

    /* The first line of the animation file must start with #qanimate */
    printf("#qanimate animation data\n");
    /* The title displayed on the animation */
    printf("title Crank-Rocker\n");
    printf("fixture\n");
    /* The primitives following fixture */
    printf("groundpin 0 0 %g %g\n", real(R[1]), imag(R[1]));
    /* For crank-rocker, crank-crank, use animate restart.
    For rocker-rocker, rocker-crank, use animate reverse */
    printf("animate restart\n");

    theta2 = deg2rad(10.0);
    z = polar(r1,theta1) - polar(r2,theta2);
    complexsolve(n1,n2,r3,-r4,z,theta3, theta4, theta3_2, theta4_2);
    R[2] = polar(r2,theta2);
    R[3] = R[2] + polar(r3,theta3);
    P = R[2] + polar(rp,theta3+beta);

    /* output coordinates for animation */
    printf("link 0 0  %f %f  %f %f  %f %f \\\n",
            real(R[2]), imag(R[2]),
            real(R[3]), imag(R[3]),
            real(R[1]), imag(R[1]) );
    printf("point trace %f %f \\\n", real(P), imag(P));
    printf("polygon fill grey90 %f %f %f %f %f %f \\\n",
            real(R[2]), imag(R[2]),
            real(R[3]), imag(R[3]),
            real(P), imag(P));
    printf("\n");
}
```

Figure 7.12: List of program simple_ani.ch.

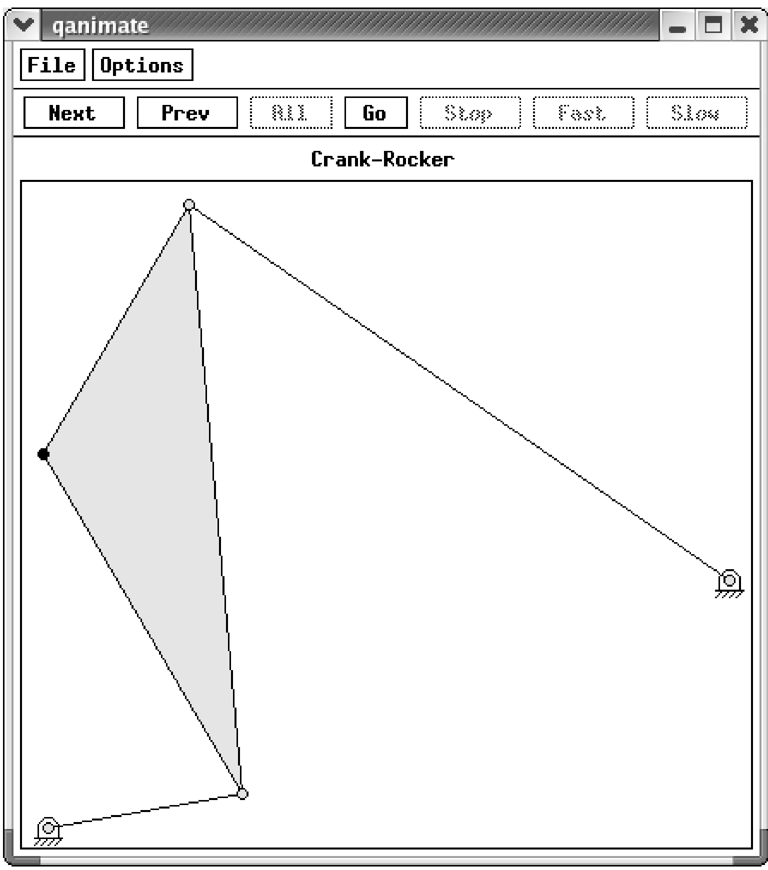


Figure 7.13: Single animation frame of a fourbar linkage.

Figure 7.14 lists a sample Ch code used to create a Quick Animation data file. The animation can be created by typing the following in the Ch command prompt:

```
animate.ch | qanimate
```

or

```
animate.ch > tmp1.qnm
qanimate tmp1.qnm
```

The result of `animate.ch` is shown in Figure 7.15. The data sets for creating the fourbar linkage animation is typically generated by a `for`-loop in Ch code. Figure 7.16 displays a snapshot of the Quick Animation animation generated by the animation data file.

```
/*******************************************************************
 * animate1.ch
 *
 * Purpose:
 *  - fourbar linkage analysis
 *  - output animation coordinate data
 * Run this program in CH as follows:
* in Unix:
 *      animate1.ch | qanimate
 *                or
 *      animate1.ch > tmp1.qnm
 *      qanimate tmp1.qnm
 * in Windows:
 *      animate1.ch | qanimate
 *                or
 *      animate1.ch > tmp1.qnm
 *      qanimate tmp1.qnm
 *                or
 *      animate1.ch > tmp1.qnm
 *      tmp1.qnm
 *******************************************************************/
#include <stdio.h>
#include <math.h>
#include <complex.h>
#define deg2rad(a) ((a)*M_PI/180.0)       // macros
#define rad2deg(a) ((a)*180.0/M_PI)

int main() {
    double theta1 = deg2rad(20.0),       // given data
           r1 = 5.5,
           r2 = 1.5,
           r3 = 4.5,
           r4 = 5,
           rp = 3,
           beta = deg2rad(25.0);
    int n1 = 2, n2 = 4;
    double theta2,theta3,theta4, theta3_2, theta4_2;
    double complex z;
    double complex R[1:3]; // link vectors
    double complex P; // coupler vector

    R[1] = polar(r1,theta1);

    /* The first line of the animation file must start with #qanimate */
    printf("#qanimate animation data\n");
    /* The title displayed on the animation */
    printf("title Crank-Rocker\n");
    printf("fixture\n");
    /* The primitives following fixture */
    printf("groundpin 0 0 %g %g\n", real(R[1]), imag(R[1]));
    /* For crank-rocker, crank-crank, use animate restart.
    For rocker-rocker, rocker-crank, use animate reverse */
    printf("animate restart\n");

    for (theta2=0.0; theta2<=deg2rad(360.0); theta2+=deg2rad(10.0)) {
        z = polar(r1,theta1) - polar(r2,theta2);
        complexsolve(n1,n2,r3,-r4,z,theta3, theta4, theta3_2, theta4_2);
        R[2] = polar(r2,theta2);
        R[3] = R[2] + polar(r3,theta3);
        P = R[2] + polar(rp,theta3+beta);

        /* output coordinates for animation */
        printf("link 0 0  %f %f  %f %f  %f %f \\\n",
                real(R[2]), imag(R[2]),
                real(R[3]), imag(R[3]),
                real(R[1]), imag(R[1]) );
        printf("point trace %f %f \\\n", real(P), imag(P));
        printf("polygon fill grey90 %f %f %f %f %f %f \\\n",
                real(R[2]), imag(R[2]),
                real(R[3]), imag(R[3]),
                real(P), imag(P));
        printf("\n");
    }
}
```

```
#qanimate animation data
title Crank-Rocker
fixture
groundpin 0 0 5.16831 1.88111
animate restart
link 0 0  1.500000 0.000000  0.883211 4.457530  5.168309 1.881111 \
point trace -0.128556 2.519485 \
polygon fill grey90 1.500000 0.000000 0.883211 4.457530 -0.128556 2.519485 \

link 0 0  1.477212 0.260472  1.067510 4.741783  5.168309 1.881111 \
point trace -0.032922 2.852672 \
polygon fill grey90 1.477212 0.260472 1.067510 4.741783 -0.032922 2.852672 \

.
.
.

link 0 0  1.477211 -0.260473  0.726293 4.176431  5.168309 1.881111 \
point trace -0.226575 2.208759 \
polygon fill grey90 1.477211 -0.260473 0.726293 4.176431 -0.226575 2.208759 \

link 0 0  1.500000 -0.000001  0.883210 4.457529  5.168309 1.881111 \
point trace -0.128557 2.519484 \
polygon fill grey90 1.500000 -0.000001 0.883210 4.457529 -0.128557 2.519484 \
```

Figure 7.15: Sample of data generated by program `animate1.ch`.

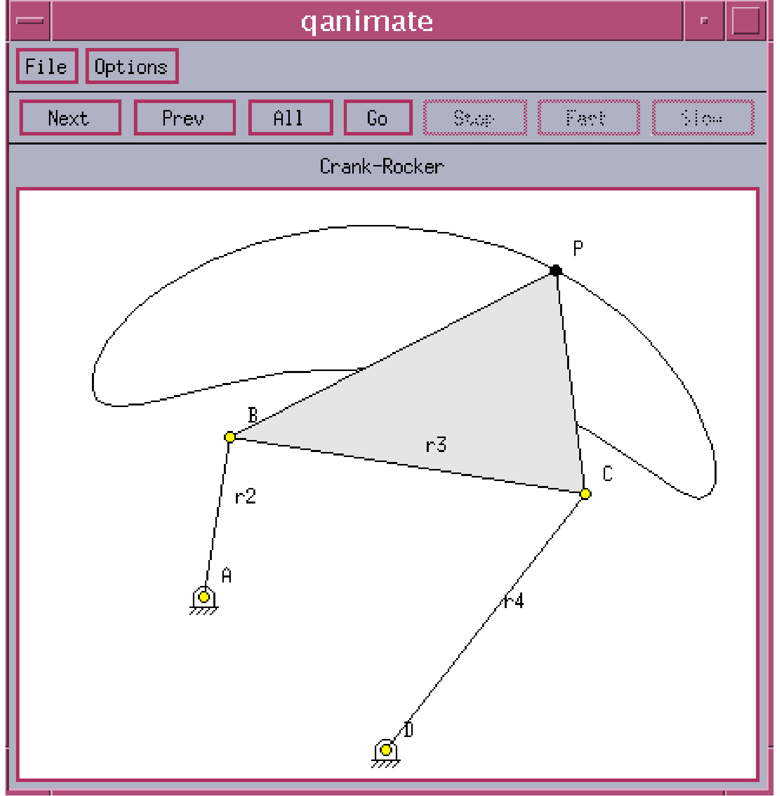


Figure 7.16: Quick Animation window showing a four-bar linkage.

Two additional examples of creating Quick Animation animation files are given below. They generate the same fourbar linkage animation as that of program `animate1.ch`, but with special handling of the animation data. Program `animate2.ch` outputs the animation coordinate data to two data files `animate1.qnm` and `animate2.qnm`, one file for each kinematic inversion of the fourbar linkage. These animation data files are then processed by program `qanimate`. Program `animate3.ch` will also produce two data files, but these data files are piped directly to the Quick Animation program to automatically generate the fourbar linkage animations. No extra commands are required to implement animation for the third sample program. Execution of the program will simply generate the desired animations. However, program `animate3.ch` works only in Unix/Linux/Mac. It does not work in Windows in the current version of Ch.

Listing of program `animate2.ch`

```
/**********************************************************************
 * animate2.ch
 * Purpose:
 *   - fourbar linkage analysis
 *   - output animation coordinate data to two data files and run animation
 * Run this program in Ch as follows:
 *      animate2.ch
 *********************************************************************/
#include <stdio.h>
#include <math.h>
#include <complex.h>
#define deg2rad(a) ((a)*M_PI/180.0)       // macros
#define rad2deg(a) ((a)*180.0/M_PI)

int main() {
    double theta1 = deg2rad(20.0),       // given data
           r1 = 5.5,
           r2 = 1.5,
           r3 = 4.5,
           r4 = 5,
           rp = 3,
           beta = deg2rad(25.0);
    int i, n1 = 2, n2 = 4;
    double theta2,theta3,theta4,theta3_2,theta4_2;
    double complex z;
    double complex R[1:3]; // link vectors
    double complex P; // coupler vector
    FILE *out[2];

    R[1] = polar(r1,theta1);
    out[0] = fopen("animate1.qnm","w"); // open animate1.qnm
    if (!out[0]) {
        perror("fopen()");
        exit(1);
    }

    out[1] = fopen("animate2.qnm","w"); // open animate2.qnm
    if (!out[1]) {
        perror("fopen()");
        exit(1);
    }

    for (i=0; i<2; i++) {
        /* The first line of the animation file must start with #qanimate */
        fprintf(out[i],"#qanimate animation data\n");
```

```
        /* The title displayed on the animation */
        fprintf(out[i],"title Crank-Rocker\n");
        fprintf(out[i],"fixture\n");
        /* The primitives following fixture */
        fprintf(out[i],"groundpin 0 0 %g %g\n", real(R[1]), imag(R[1]));
        /* For crank-rocker, crank-crank, use animate restart.
        For rocker-rocker, rocker-crank, use animate reverse */
        fprintf(out[i],"animate restart\n");
    }

    for (theta2=0.0; theta2<=deg2rad(360.0); theta2+=deg2rad(10.0)) {
        R[2] = polar(r2,theta2);

        z = polar(r1,theta1) - polar(r2,theta2);
        complexsolve(n1,n2,r3,-r4,z,theta3,theta4, theta3_2, theta4_2);

        /* first solution */
        R[3] = R[2] + polar(r3,theta3);
        P = R[2] + polar(rp,theta3+beta);

/* output coordinates for animation */
        fprintf(out[0],"link 0 0  %f %f  %f %f  %f %f \\\n",
                real(R[2]), imag(R[2]),
                real(R[3]), imag(R[3]),
                real(R[1]), imag(R[1]) );
        fprintf(out[0],"point trace %f %f \\\n", real(P), imag(P));
        fprintf(out[0],"polygon fill grey90 %f %f %f %f %f %f \\\n",
                real(R[2]), imag(R[2]),
                real(R[3]), imag(R[3]),
                real(P), imag(P));
        fprintf(out[0],"\n");

        /* second solution */
        R[3] = R[2] + polar(r3,theta3_2);
        P = R[2] + polar(rp,theta3_2+beta);

        /* output coordinates for animation */
        fprintf(out[1],"link 0 0  %f %f  %f %f  %f %f \\\n",
                real(R[2]), imag(R[2]),
                real(R[3]), imag(R[3]),
                real(R[1]), imag(R[1]) );
        fprintf(out[1],"point trace %f %f \\\n", real(P), imag(P));
        fprintf(out[1],"polygon fill grey90 %f %f %f %f %f %f \\\n",
                real(R[2]), imag(R[2]),
                real(R[3]), imag(R[3]),
                real(P), imag(P));
        fprintf(out[1],"\n");
    }

    fclose(out[0]);
    fclose(out[1]);
    qanimate animate1.qnm
    remove("animate1.qnm"); // remove the data file
    qanimate animate2.qnm
    remove("animate2.qnm"); // remove the data file
}
```

Listing of program `animate3.ch`

```
/*******************************************************************
 * animate3.ch
 *
 * Purpose:
 *  - fourbar linkage analysis
 *  - output animation coordinate data to two pipes
 * Run this program in Ch as follows:
 *      animate3.ch
 *******************************************************************/
#include <stdio.h>
#include <math.h>
#include <complex.h>
#define deg2rad(a) ((a)*M_PI/180.0)       // macros
#define rad2deg(a) ((a)*180.0/M_PI)

int main() {
    double theta1 = deg2rad(20.0),       // given data
           r1 = 5.5,
           r2 = 1.5,
           r3 = 4.5,
           r4 = 5,
           rp = 3,
           beta = deg2rad(25.0);
    int i, n1 = 2, n2 = 4;
    double theta2,theta3,theta4,theta3_2,theta4_2;
    double complex z;
    double complex R[1:3]; // link vectors
    double complex P;         // coupler vector
    FILE *out[2];

    R[1] = polar(r1,theta1);
    out[0] = popen("qanimate","w");     // open qanimate pipe
    if (!out[0]) {
        perror("popen()");
        exit(1);
    }

    out[1] = popen("qanimate","w");
    if (!out[1]) {
        perror("popen()");
        exit(1);
    }

    for (i=0; i<2; i++) {
        /* The first line of the animation file must start with #qanimate */
        fprintf(out[i],"#qanimate animation data\n");
        /* The title displayed on the animation */
        fprintf(out[i],"title Crank-Rocker\n");
        fprintf(out[i],"fixture\n");
        /* The primitives following fixture */
        fprintf(out[i],"groundpin 0 0 %f %f\n", real(R[1]), imag(R[1]));
        /* For crank-rocker, crank-crank, use animate restart.
        For rocker-rocker, rocker-crank, use animate reverse */
        fprintf(out[i],"animate restart\n");
    }

    for (theta2=0.0; theta2<=deg2rad(360.0); theta2+=deg2rad(10.0)) {
        R[2] = polar(r2,theta2);
```

```
        z = polar(r1,theta1) - polar(r2,theta2);
        complexsolve(n1,n2,r3,-r4,z,theta3,theta4, theta3_2, theta4_2);

        /* first solution */
        R[3] = R[2] + polar(r3,theta3);
        P = R[2] + polar(rp,theta3+beta);

        /* output coordinates for animation */
        fprintf(out[0],"link 0 0  %f %f  %f %f  %f %f \\\n",
                real(R[2]), imag(R[2]),
                real(R[3]), imag(R[3]),
                real(R[1]), imag(R[1]) );
        fprintf(out[0],"point trace %f %f \\\n", real(P), imag(P));
        fprintf(out[0],"polygon fill grey90 %f %f %f %f %f %f \\\n",
                real(R[2]), imag(R[2]),
                real(R[3]), imag(R[3]),
                real(P), imag(P));
        fprintf(out[0],"\n");

        /* second solution */
        R[3] = R[2] + polar(r3,theta3_2);
        P = R[2] + polar(rp,theta3_2+beta);

        /* output coordinates for animation */
        fprintf(out[1],"link 0 0  %f %f  %f %f  %f %f \\\n",
                real(R[2]), imag(R[2]),
                real(R[3]), imag(R[3]),
                real(R[1]), imag(R[1]) );
        fprintf(out[1],"point trace %f %f \\\n", real(P), imag(P));
        fprintf(out[1],"polygon fill grey90 %f %f %f %f %f %f \\\n",
                real(R[2]), imag(R[2]),
                real(R[3]), imag(R[3]),
                real(P), imag(P));
        fprintf(out[1],"\n");
    }

    pclose(out[0]);
    pclose(out[1]);
}
```

Chapter 8

Implementations of Interactive Web Pages for Mechanism Design and Analysis

Interactive web-based mechanism analysis for various linkages have been introduced in the previous chapters. From those brief introductions, it can be seen that web-based analysis was a convenient tool for performing kinematic and dynamic analysis of the mechanisms. Users only have to input specific values for the mechanism they want to study and then click on the "run" button to begin the mechanism analysis. This saves the user valuable time since the computer is performing all the required calculations, and he/she does not need to have any prior programming experiences.

This chapter will provide a more in depth discussion on how the interactive mechanism analysis web pages are designed and implemented. At the end of the chapter, the user should have a clear understanding of the functionality of interactive web-based analysis programs and be able to develop similar web pages.

8.1 Introduction to CGI Programming

The basis behind the concept of interactive web-based mechanism analysis is *CGI programming*. CGI, or *Common Gateway Interface*, allows web servers to execute scripts and programs using data sent by a client and then returns the results to the client. The web pages for mechanism analysis utilizes Ch scripts to perform the desired analysis. CGI programming involves the interaction between two files: an HTML file and a script/program file.

8.1.1 Writing HTML Files

HTML files allows for the creation of internet web pages that web servers utilize to prompt users to input data required by CGI scripts. For example, consider the "fill-out form" web page shown in Figure 8.1. It is a simple HTML document for sending data through a web page.

HTML is not a programming language, but a formatting language. Formatting is specified by *tags* in HTML files. Tags can be easily distinguished from other text in HTML files because they are enclosed within a pair of angle brackets, < . . . >. For example, <TITLE> and <BODY> are two HTML tags used in Program 42. Furthermore, tags and other elements in HTML code are case insensitive. Thus, <TITLE> is equivalent to <Title> or <Title>. Table 8.1 lists some of the common HTML tags along with a brief description. Note that some format tags requires an ending statement, such as <BODY>...</BODY>.

All HTML documents should have a clearly defined head and body. The head of an HTML document usually includes the title of the document. Note that the title does not appear inside the body of document, but instead, displayed in the window's title bar for the web browser. The head and title are delimited by

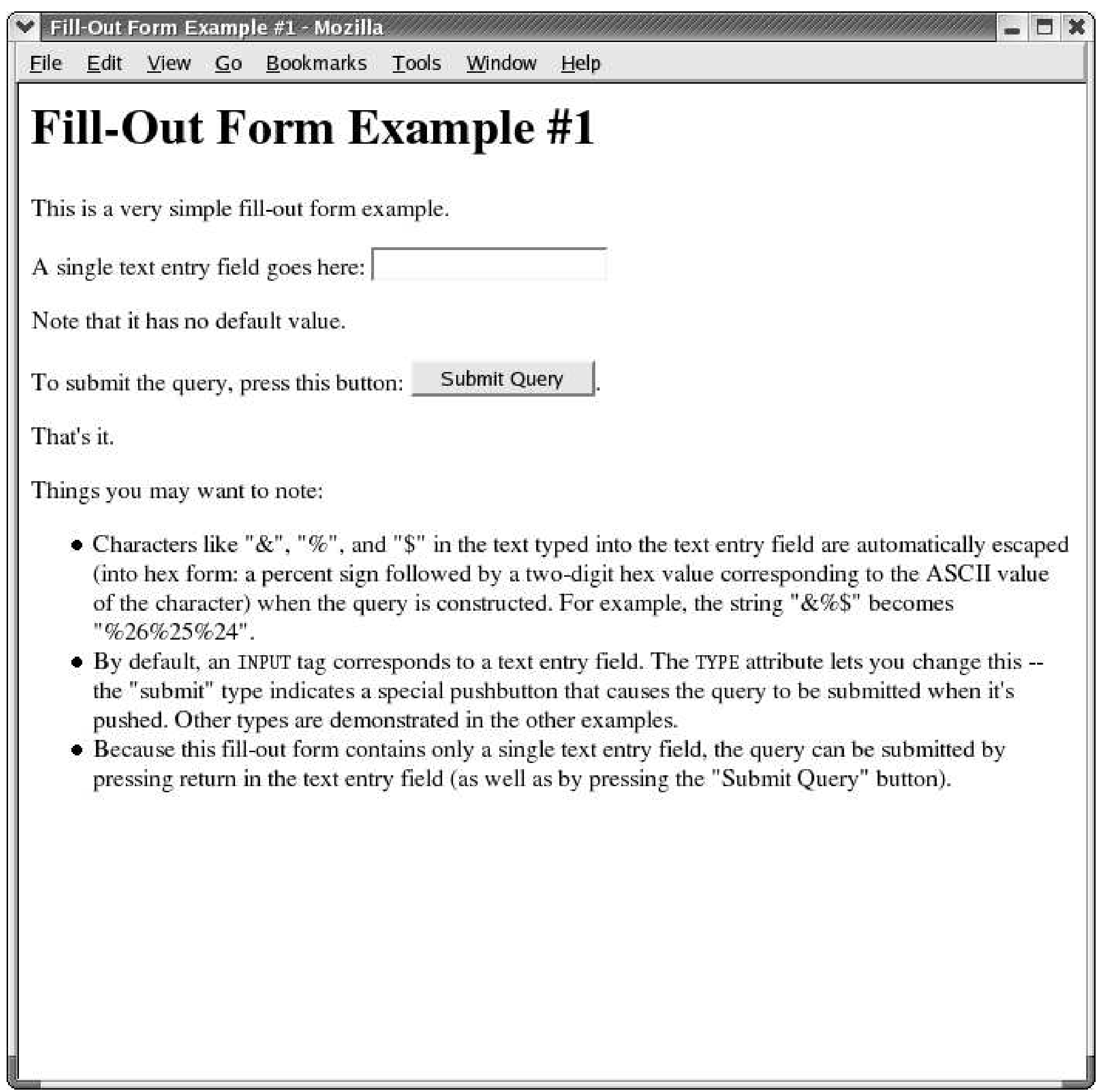


Figure 8.1: Fill-out form example.

```
<HEAD>
<TITLE>Fill-Out Form Example #1</TITLE>
</HEAD>
<BODY bgcolor="#FFFFFF" text="#000000" vlink="#FF0000">
<H1>Fill-Out Form Example #1</H1>

This is a very simple fill-out form example. <P>

<FORM METHOD="POST" ACTION="/cgi-bin/chcgi/form.ch">

A single text entry field goes here: <INPUT NAME="entry"> <P>

Note that it has no default value. <P>

To submit the query, press this button: <INPUT TYPE="submit"
VALUE="Submit Query">. <P>

</FORM>

That’s it. <P>

Things you may want to note: <P>

<UL>
<LI> Characters like "&amp;", "%", and "$" in the text typed into the
     text entry field are automatically escaped (into hex form: a
     percent sign followed by a two-digit hex value corresponding to
     the ASCII value of the character) when the query is constructed.
     For example, the string "&amp;%$" becomes "%26%25%24".
<LI> By default, an <CODE>INPUT</CODE> tag corresponds to a text entry
     field.  The <CODE>TYPE</CODE> attribute lets you change this --
     the "submit" type indicates a special pushbutton that causes the
     query to be submitted when it’s pushed.  Other types are
     demonstrated in the other examples.
<LI> Because this fill-out form contains only a single text entry
     field, the query can be submitted by pressing return in the text
     entry field (as well as by pressing the "Submit Query" button).
</UL>
</BODY>
```

Program 42: HTML source code for Figure 8.1.

`<HEAD>...</HEAD>` and `<TITLE>...</TITLE>`, respectively. For example, the title of the HTML document shown in Figure 8.1 is "Fill-Out Form Example #1".

The source code for the HTML document of Figure 8.1 is Program 42. As can be seen from this source code, the body of an HTML document contains all the information displayed on the internet web page. Notice that some attributes have been specified for the HTML document within the `<BODY>` tag. The `bgcolor=` attribute sets the background color of the web page, whereas attribute `text=` defines the color of the text to be displayed. The last attribute, `vlink=`, specifies the color of a previously visited hyperlink. In order to have a visible title displayed on the web page, the header tag `<H1>...</H1>` has been used.

Creating a form begins with the `<FORM>` tag. Within the `<FORM>` tag, there is typically two attributes associated with it: `method=` and `action=`. For the example of Program 42, the form method is set to `post`, which means that the data are to be passed via the standard input method. The `action=` attribute is used to indicate the program or script to process the data collected by the HTML form. Inside the form, data entry fields are specified with the `<INPUT>` tag. Note that data entry fields may be distinguished by names with the `name=` attribute, as in Program 42. Submission of data is accomplished through the `<INPUT>` tag with `type=` attribute set to "submit".

Table 8.1: Description of HTML tags.

Tags	Description
`<TITLE>...</TITLE>`	Specify title of an HTML document.
`<HEAD>...</HEAD>`	Indicates the head section of an HTML document.
`<BODY>...</BODY>`	Indicates the body section of an HTML document.
`<H[#]>...</H[#]>`	Heading format; value of *[#]* can range from 1 to 6, with `<H1>` being the highest setting.
`<HR>`	Horizontal rule; inserts a horizontal line between sections of text.
`<CENTER>...</CENTER>`	Horizontal centering.
`<IMG>`	Insert an in-line image.
`<PRE>...</PRE>`	Display text in preformatted form.
`<P>`	Paragraph tag; inserts a blank line between sections of text.
`<UL>...<LI>...</UL>`	Unordered, or unnumbered, list.
` `	Break tag; starts new line, but does not insert a blank one.
`<B>...</B>`	Bold text.
`<I>...</I>`	Italics text.
`<U>...</U>`	Underline text.
`<FORM>...</FORM>`	Define a form field within an HTML document.
`<INPUT>`	Define an input field.
`<SELECT>...</SELECT>`	Define a selection field.

8.1.2 Writing CGI Script Files

Processing data sent from an HTML document with the `<FORM>` tag requires a script file, which is located on the server side. Once data are passed to the server through CGI, the necessary procedures are performed and the result is returned to the client. As an example, Program 43 is the Ch script associated with the

```
#!/bin/ch
#include <cgi.h>

int main() {
  int i, num;
  chstrarray name, value;
  class CResponse Response;
  class CRequest Request;

  Response.begin();
  Response.title("CGI FORM results");
  printf("<H1>CGI FORM test script reports:</H1>\n");

  num = Request.getFormNameValue(name, value);
  if(num == 0) {
    printf("No name/value has been submitted<P>\n");
    exit(0);
  }
  else if(num < 0) {
    printf("Error: in Request.getFormNameValue() <P>\n");
    exit(0);
  }
  printf("The following %d name/value pairs are submitted<P>\n",num);
  printf("<UL>\n");
  for(i=0; i < num; i++) {
    printf("<LI> <CODE>%s = ",name[i]);
    if(value[i])
      printf("%s",value[i]);
    printf("</CODE>\n");
  }
  printf("</UL>\n");
  Response.end();
}
```

Program 43: Example CGI program.

form filling example in the above section. It is used to process the data collected from the HTML document shown in Figure 8.1, which is generated by Program 42.

The first line of Program 43, `#!/bin/ch`, indicates that this program is a Ch script file. Header file **cgi.h** contains the definitions for classes **CResponse** and **CRequest**, which are convenient for CGI programming. Class **CRequest** can be used to obtain data from HTML documents, whereas class **CResponse** can be used to output data as an HTML document. Two generic data types **chchar** and **chstrarray** are also type-defined in this header file. They are type-defined as follows.

```
typedef char   chchar;
typedef char** chstrarray;
```

Program 43 declares two variables, `name` and `value`, of data type **chstrarray** to store the name and value data passed by the form of Program 42. Member function **begin()** of class **CResponse** indicates the start of the output. The title of the dynamically generated HTML document may be added with member function **title()**, which requires a string for its input argument. For the document generated by Program 43, the title is specified as `"CGI FORM results"`. Lines of HTML codes are added with the C **printf()** function. The names and values submitted in a fill-out form can be obtained by the CGI program by member functions **CResquest::getForm()**, **CResquest::getForms()**, or **CResquest::getFormNameValue()**. For

Figure 8.2: Output of submitting form of Figure 8.1.

the example of Program 43, member function **getFormNameValue()** is used. The prototype for member function **getFormNameValue()** is shown below.

```
int CRequest::getFormNameValue(chstrarray &names,
                               chstrarray &values);
```

The **getFormNameValue()** member function obtains all the name/value pairs from the HTML form document and stores them into arguments `names` and `values`, which are treated as string arrays. After obtaining the name/value pair(s) from the form-filling document, the rest of the code is used for checking input errors and displaying the values that have been entered. Note that the primary function of HTML tag `<CODE>` is to display codes extracted from programming files. Text are displayed as they are typed within the tag. Figure 8.2 shows the dynamically generated HTML document after submitting the form shown in Figure 8.1 with the entered value ”something”.

8.2 Web-Based Animation Example

Although the form-filling example in the above section provides a good introduction to CGI programming, an in depth example is given in this section to show how to write a more effective CGI program. One such example would be the fourbar animation web page discussed in Section 2.9. The HTML document for animating the fourbar linkage mechanism via the internet is shown in Figure 2.20. Its source code is listed as Program 44.

Program 44 concisely specifies the layout of Figure 2.20. After the head and title of the HTML document, a figure of the fourbar linkage is displayed with the `<IMG>` tag, where option `src=` indicates the location of the image file on the web server. Next, the `<FORM>` tag allows for the user to input data required for generating the fourbar animation. The initial `<SELECT>` tag allow users to specify the use of SI of US Customary units. Text input areas are created for link lengths and ground angle θ_1. Furthermore, input areas have been allocated for the coupler point properties if the user decides to specify a coupler link. Note that each `<INPUT>` tag is associated with a name describing the required parameter. This allows the CGI script, `fourbar_ani.ch`, to easily identify the type of data being passed into it. For example, the CGI script would be able to distinguish between link lengths r_1 and r_2. Along with the `name=` option for tag `<INPUT>`, the options `value=` and `size=` specifies the default values and the field-width for each input, respectively. Thus, the input for link length r_4 has a default value of `"5.0"` and field width size of 5.

Following the link lengths area, ground angle θ_1 and the phase angle for the coupler point β may be specified. The `<SELECT>` tag allows the user to specify whether the angles are in degrees or radians. The branch number selection of the fourbar linkage is also created in this manner.

The CGI script for processing the data collected from the HTML form created by Program 44 is listed as Program 45. Program 45 consists of the **main()** function and two other functions, `errorHandler()` and `fourbar_ani()`. In the **main()** function, an object of class **CRequest**, `Request`, is instantiated in order to obtain the data submitted by the HTML form for animation of the fourbar mechanism. Member function **Request::getForm()** is used to acquire the submitted data one value at a time. Member function **getForm()** takes the name associated with the value to be acquired as an input value, and then the acquired value becomes the member function's return value. Note that the data values are read and stored as strings. If, for some reason, the value submitted from the HTML form is invalid, function `errorHandler()` is called to display an error message in HTML document form by using class **CResponse**.

If the values collected from the HTML form are all valid, then they are passed into function `fourbar_ani()` to simulate the motion of the fourbar linkage via the world wide web. In function `fourbar_ani()`, the data values passed in as strings are now converted to double precision floating-point numbers, with the exceptions of arguments `mode` and `branchnum`. These two variables can be used directly as strings. If the values for θ_1 and β were submitted in degrees, then they are converted to radians in function `fourbar_ani()`. The next step to simulate the motion of the fourbar mechanism is to set the parameters of object `fourbar` of class **CFourbar**. Once this is done, and the branch number of the fourbar is determined from string `branchnum`, class **CResponse** can be used to stream the linkage animation to an HTML document. Note that only a Rocker-Rocker mechanism may have a branch number greater than 2. The statement

```
Response.setContentType("application/x-qnm");
```

indicates that the output is a Quick Animation application. Recall that member function **CFourbar:::animation()** has a variable argument list. Also recall that if the second argument of member function **animation()** is `QANIMATE_OUTPUTTYPE_FILE`, then the data for generating the fourbar animation can be saved into a file whose name is specified by a third function argument. For Program 45, the second argument of member function **animation()** is `QANIMATE_OUTPUTTYPE_STREAM`, which indicates that the animation data should be streamed directly to the HTML document.

An alternative CGI program for animating the fourbar linkage is Program 46. It is similar to Program 45, except that this CGI program independently generates the animation data rather than use member function **CFourbar::animation()**. However, Program 46 still incorporates the use of class **CFourbar** in order to determine the joint limits and type of the fourbar. Vectors of **double complex** types are used to define the various positions of the links and coupler point. In order to stream the animation to the web browser file pointer `animation` is set to point to **stdout**. Program 46 is provided to show the reader how to create and display animations on the internet with Quick Animation and CGI programming.

8.2.1 Configuration and Setup of Web Servers

In order to run the Quick Animation application from a Netscape Web server in a Unix operating system, the following line has to be added to the Netscape WWW server configuration file **mime.types** located in directory `server_home_dir/https-80_or_http/config`.

```
type=application/x-qnm     exts=qnm
```

For the Apache Web server, the line

```
<HEAD>
<TITLE>
Interactive Four-Bar Linkage Animation
</TITLE>
</HEAD>
<BODY bgcolor="#FFFFFF">
<H1>
Interactive Four-Bar Linkage Animation
</H1>

<HR>

The four-bar linkage below can be animated on your screen.

<CENTER>
<IMG SRC="/chhtml/toolkit/mechanism/fig/fourbar/fourbar_trans.gif">
</CENTER>

<P>
<HR>

<FORM method="POST" action="/cgi-bin/chcgi/toolkit/mechanism/fourbar/fourbar_ani.ch">

<BR>
Unit Type:
<SELECT name="unit">
<OPTION value="SI">SI
<OPTION value="USC">US Customary
</SELECT>

<BR>
Link lengths (m or ft):
<B>r1:</B> <INPUT name="r1" value="0.12" size=5>
<B>r2:</B> <INPUT name="r2" value="0.04" size=5>
<B>r3:</B> <INPUT name="r3" value="0.10" size=5>
<B>r4:</B> <INPUT name="r4" value="0.07" size=5>
<B>rp:</B> <INPUT name="rp" value="0.05" size=5>
<BR>

Angles:
<B>theta1:</B> <INPUT name="th" value="0" size=6>
<B>beta:</B> <INPUT name="beta" value="20" size=6>
<SELECT name="mode">
<OPTION value="deg">Degree Mode
<OPTION value="rad">Radian Mode
</SELECT>
<BR>

Number of points: <INPUT name="n" value="30" size=3>
<BR><BR>
```

Program 44: Source code for web-based fourbar linkage animation.

```
Branch Number:
<SELECT name="branchnum">
<OPTION value="1st">1
<OPTION value="2nd">2
<OPTION value="3rd">3
<OPTION value="4th">4
</SELECT>
<BR>

<P>

<INPUT TYPE=submit VALUE="Run">
<INPUT TYPE=reset VALUE="Reset">
</FORM>

<P>

<HR size=4>
<A HREF="http://www.softintegration.com" target="_top">
<img src="/chhtml/images/poweredbych.gif" alt="Powered by Ch"></A>
</FONT>
</BODY>
```

Program 44: Source code for web-based fourbar linkage animation (Contd.).

```
application/x-qnm      qnm
```

may be added to file `server_home_dir/conf/mime.types`. Note that the Web server needs to restart in order for the changes to be effective.

```
#!/bin/ch

#include<stdio.h>
#include<string.h>
#include<cgi.h>
#include<fourbar.h>

void errorHandler(char *reason)
{
    class CResponse Response;

    Response.begin();
    Response.title("Fourbar Animation");

    fprintf stdout << ENDFILE
      <H3>Fourbar Animation Failed</H3>
      Your fourbar animation has not been generated
       because: $reason.<BR>
       <A HREF="/chhtml/toolkit/mechanism/fourbar/fourbar_ani.html">Try again.</A>
       <P><HR SIZE=4>
       <hr>
       <A HREF="http://www.softintegration.com" target="_top">
       <img src="/chhtml/images/poweredbych.gif" alt="Powered by Ch"></A>
     ENDFILE
     Response.end();
     exit(0);
}


void fourbar_ani(char *unit, char *rr1, char *rr2, char *rr3, char *rr4, char *rrp,
                 char *mode, char* ttheta1, char *bbeta, char *n,
                 char *branchnum)
{
   class CResponse Response;
   double r[1:4], rpp;
   double beta;
   double theta2_min[2], theta2_max[2],
          theta4_min[2], theta4_max[2];
   double theta[1:2][1:4];
   double complex P1, P2;
   int num_pts;
   int theta_id = FOURBAR_LINK2;
   int fourbartype;
   FILE *animation;
   CFourbar fourbar;
   double complex R[1:4];
   int i;

   /* Define the fourbar-linkage. */
   r[1] = streval(rr1);  r[2] = streval(rr2);
   r[3] = streval(rr3);  r[4] = streval(rr4);
   rpp = streval(rrp);
   theta[1][1] = streval(ttheta1);
   beta = streval(bbeta);
   num_pts = streval(n);
```

Program 45: CGI program for web-based fourbar linkage animation.

```
   if(strcmp(mode, "deg") == 0)
   {
      theta[1][1] = M_DEG2RAD(theta[1][1]);
      beta = M_DEG2RAD(beta);
   }
   theta[2][1] = theta[1][1];

   /* Perform position analysis of fourbar. */
   if(strcmp(unit, "USC") == 0)
     fourbar.uscUnit(true);
   fourbar.setLinks(r[1], r[2], r[3], r[4], theta[1][1]);
   fourbar.setCouplerPoint(rpp, beta, TRACE_ON);
   fourbartype = fourbar.getJointLimits(theta2_min, theta2_max, theta4_min, theta4_max);
   fourbar.setNumPoints(num_pts);

   /* Determine branch number. */
   if(strcmp(branchnum, "1st") == 0)
      i = 1;
   else if(strcmp(branchnum, "2nd") == 0)
      i = 2;
   else if(strcmp(branchnum, "3rd") == 0)
      i = 3;
   else
      i = 4;

   /* Determine whether branchnum > 2 is possible. */
   if((fourbartype != FOURBAR_ROCKERROCKER) && (i > 2))
      errorHandler("This linkage only have two branches.");

   /* Display the results onto the browser. */
   Response.setContentType("application/x-qnm");
   Response.begin();
   fourbar.animation(i, QANIMATE_OUTPUTTYPE_STREAM);
   Response.end();
}

int main()
{
   class CRequest Request;
   chchar *r1, *r2, *r3, *r4;
   chchar *theta1;
   chchar *rp, *beta;
   chchar *mode, *n, *branchnum;
   chchar *unit;

   unit = Request.getForm("unit");
   r1 = Request.getForm("r1");
   if(!r1)
      errorHandler("You didn't input a 'r1' value.");
   else if(!isnum(r1))
      errorHandler("'r1' is not a valid number.");
   r2 = Request.getForm("r2");
   if(!r2)
      errorHandler("You didn't input a 'r2' value.");
   else if(!isnum(r2))
      errorHandler("'r2' is not a valid number.");
```

Program 45: CGI program for web-based fourbar linkage animation (Contd.).

```
   r3 = Request.getForm("r3");
   if(!r3)
      errorHandler("You didn’t input a ’r3’ value.");
   else if(!isnum(r3))
      errorHandler("’r3’ is not a valid number.");
   r4 = Request.getForm("r4");
   if(!r4)
      errorHandler("You didn’t input a ’r4’ value.");
   else if(!isnum(r4))
      errorHandler("’r4’ is not a valid number.");
   rp = Request.getForm("rp");
   if(!rp)
      errorHandler("You didn’t input a ’rp’ value.");
   else if(!isnum(rp))
      errorHandler("’rp’ is not a valid number.");
   theta1 = Request.getForm("th");
   if(!theta1)
      errorHandler("You didn’t input a ’th’ value.");
   else if(!isnum(theta1))
      errorHandler("’th’ is not a valid number.");
   beta = Request.getForm("beta");
   if(!beta)
      errorHandler("You didn’t input a ’beta’ value.");
   else if(!isnum(beta))
      errorHandler("’beta’ is not a valid number.");
   mode = Request.getForm("mode");
   n = Request.getForm("n");
   if(!n)
      errorHandler("You didn’t input a ’n’ value.");
   else if(!isnum(n))
      errorHandler("’n’ is not a valid number.");
   branchnum = Request.getForm("branchnum");

   fourbar_ani(unit, r1, r2, r3, r4, rp, mode, theta1, beta, n, branchnum);

   return 0;
}
```

Program 45: CGI program for web-based fourbar linkage animation (Contd.).

```
#!/bin/ch

#include<stdio.h>
#include<string.h>
#include<cgi.h>
#include<fourbar.h>

#define deg2rad(a) (a*(M_PI/180))

void errorHandler(char *reason)
{
    class CResponse Response;

    Response.begin();
    Response.title("Fourbar Animation");

    fprintf stdout << ENDFILE
      <H3>Fourbar Animation Failed</H3>
      Your fourbar animation has not been generated
       because: $reason.<BR>
       <A HREF="/chhtml/toolkit/mechanism/fourbar/fourbar_ani.html">Try again.</A>
       <P><HR SIZE=4>
       <hr>
       <A HREF="http://www.softintegration.com" target="_top">
       <img src="/chhtml/images/poweredbych.gif" alt="Powered by Ch"></A>
     ENDFILE
     Response.end();
     exit(0);
}


void fourbar_ani(char *unit, char *rr1, char *rr2, char *rr3, char *rr4,
                 char *rrp, char *mode, char* ttheta1, char *bbeta, char *n,
                 char *branchnum)
{
   class CResponse Response;
   double r[1:4], rpp;
   double beta;
   double theta2;
   double theta2_min[2], theta2_max[2],
          theta4_min[2], theta4_max[2];
   double min, max;
   double theta[1:2][1:4];
   double incr;
   int num_pts;
   int fourbartype;
   string_t grashof_name;
   FILE *animation;
   double complex R[1:4];
   double complex z;
   double complex P;
   int n1 = 2, n2 = 4;
   CFourbar fourbar;
   int i;
```

Program 46: Alternative CGI program for web-based fourbar linkage animation.

```
/* Define the fourbar-linkage. */
r[1] = streval(rr1);  r[2] = streval(rr2);
r[3] = streval(rr3);  r[4] = streval(rr4);
rpp = streval(rrp);
theta[1][1] = streval(ttheta1);
beta = streval(bbeta);
num_pts = streval(n);

if(strcmp(mode, "deg") == 0)
{
   theta[1][1] = deg2rad(theta[1][1]);
   beta = deg2rad(beta);
}
theta[2][1] = theta[1][1];

/* Determine branch number. */
if(strcmp(branchnum, "1st") == 0)
   i = 1;
else if(strcmp(branchnum, "2nd") == 0)
   i = 2;
else if(strcmp(branchnum, "3rd") == 0)
   i = 3;
else
   i = 4;

/* Simulate motion of fourbar. */
if(strcmp(unit, "USC") == 0)
  fourbar.uscUnit(true);
fourbar.setLinks(r[1], r[2], r[3], r[4], theta[1][1]);
fourbartype = fourbar.getJointLimits(theta2_min, theta2_max, theta4_min, theta4_max);
fourbar.grashof(grashof_name);

/* Determine whether branchnum > 2 is possible. */
if((fourbartype != FOURBAR_ROCKERROCKER) && (i > 2))
   errorHandler("This linkage only have two branches.");

if(i < 3) {
   min = theta2_min[0];
   max = theta2_max[0];
}
else {
   min = theta2_min[1];
   max = theta2_max[1];
}

animation = stdout;

Response.setContentType("application/x-qnm");
Response.begin();
R[1] = polar(r[1], theta[1][1]);
fprintf(animation, "#qanimate animation data\n");
fprintf(animation, "title \"%s\"\n", grashof_name);
fprintf(animation, "fixture\n");
fprintf(animation, "groundpin 0 0 %f %f\n", real(R[1]), imag(R[1]));
```

Program 46: Alternative CGI program for web-based fourbar linkage animation (Contd.).

```
   if((fourbartype == FOURBAR_ROCKERROCKER)
       || (fourbartype == FOURBAR_ROCKERCRANK))
   {
      fprintf(animation, "animate reverse \n");
   }
   else
   {
      fprintf(animation, "animate restart \n");
   }

   incr = abs((max-min)/num_pts);
   for(theta2 = min; theta2 <= max; theta2+=incr) {
      theta[1][2] = theta2;  theta[2][2] = theta2;
      R[2] = polar(r[2], theta2);

      z = R[1] - R[2];
      complexsolve(n1, n2, r[3], -r[4], z, theta[1][3], theta[1][4],
                   theta[2][3], theta[2][4]);

      if((i == 1) || (i == 3)) {
         R[3] = R[2] + polar(r[3], theta[1][3]);
         P = R[2] + polar(rpp, theta[1][3]+beta);
      }
      else {
         R[3] = R[2] + polar(r[3], theta[2][3]);
         P = R[2] + polar(rpp, theta[2][3]+beta);
      }

      /* Animation coordinates. */
      fprintf(animation, "link 0 0  %f %f  %f %f  %f %f \\\n",
              real(R[2]), imag(R[2]),
              real(R[3]), imag(R[3]),
              real(R[1]), imag(R[1]));
      fprintf(animation, "point trace %f %f \\\n", real(P), imag(P));
      fprintf(animation, "polygon fill grey90 %f %f  %f %f  %f %f \\\n",
              real(R[2]), imag(R[2]),
              real(R[3]), imag(R[3]),
              real(P), imag(P));
      fprintf(animation, "\n");
   }
   Response.end();
}


int main()
{
   class CRequest Request;
   chchar *r1, *r2, *r3, *r4;
   chchar *theta1;
   chchar *rp, *beta;
   chchar *mode, *n, *branchnum;
   chchar *unit;
```

Program 46: Alternative CGI program for web-based fourbar linkage animation (Contd.).

```
   unit = Request.getForm("unit");
   r1 = Request.getForm("r1");
   if(!r1)
      errorHandler("You didn’t input a ’r1’ value.");
   else if(!isnum(r1))
      errorHandler("’r1’ is not a valid number.");
   r2 = Request.getForm("r2");
   if(!r2)
      errorHandler("You didn’t input a ’r2’ value.");
   else if(!isnum(r2))
      errorHandler("’r2’ is not a valid number.");
   r3 = Request.getForm("r3");
   if(!r3)
      errorHandler("You didn’t input a ’r3’ value.");
   else if(!isnum(r3))
      errorHandler("’r3’ is not a valid number.");
   r4 = Request.getForm("r4");
   if(!r4)
      errorHandler("You didn’t input a ’r4’ value.");
   else if(!isnum(r4))
      errorHandler("’r4’ is not a valid number.");
   rp = Request.getForm("rp");
   if(!rp)
      errorHandler("You didn’t input a ’rp’ value.");
   else if(!isnum(rp))
      errorHandler("’rp’ is not a valid number.");
   theta1 = Request.getForm("th");
   if(!theta1)
      errorHandler("You didn’t input a ’th’ value.");
   else if(!isnum(theta1))
      errorHandler("’th’ is not a valid number.");
   beta = Request.getForm("beta");
   if(!beta)
      errorHandler("You didn’t input a ’beta’ value.");
   else if(!isnum(beta))
      errorHandler("’beta’ is not a valid number.");
   mode = Request.getForm("mode");
   n = Request.getForm("n");
   if(!n)
      errorHandler("You didn’t input a ’n’ value.");
   else if(!isnum(n))
      errorHandler("’n’ is not a valid number.");
   branchnum = Request.getForm("branchnum");

   fourbar_ani(unit, r1, r2, r3, r4, rp, mode, theta1, beta, n, branchnum);

   return 0;
}
```

Program 46: Alternative CGI program for web-based fourbar linkage animation (Contd.).

Chapter 9

References

1. Hrones, J.A., and Nelson, G. L., *Analysis of the Four-Bar Linkage*, MIT Press and John Wiley and Sons, Inc., New York, 1951.

2. Shigley, J. E., and Uicker, J. J., Jr., *Theory of Machines and Mechanisms*, McGraw-Hill Inc., New York, 1980.

3. Erdman, A. G., Sandor, G. N, and Kota, S., Mechanism Design, Analysis and Synthesis, vol. 1, 4th edition, Prentice Hall, Englewood Cliffs, New Jersey, 2001.

4. Cheng, H. H., Pedagogically Effective Programming Environment for Teaching Mechanism Design, *Computer Applications in Engineering Education*, Vol. 2, No. 1, 1994, pp. 23-39.

5. Cheng, H. H., Scientific Computing in the Ch Programming Language, *Scientific Programming*, Vol. 2, No. 3, Fall, 1993, pp. 49-75.

6. Cheng, H. H., Handling of Complex Numbers in the Ch Programming Language, *Scientific Programming*, Vol. 2, No. 3, Fall, 1993, pp. 76-106.

7. Harry H. Cheng, *The Ch Language Environment, — User's Guide*, version 3.0, SoftIntegration, Inc., 2002; http://www.softintegration.com.

8. *The Ch Language Environment, — Reference Guide*, version 3.0, SoftIntegration, Inc., 2002.

9. *Th Ch Language Environment CGI Toolkit, — User's Guide*, version 3.0, SoftIntegration, Inc., 2002

Appendix A

Header File linkage.h

linkage.h

The header file **linkage.h** contains macros and definitions commonly used by all the mechanism classes. It is included in each class header file.

Functions

Macros	Description
M_DEG2RAD(a)	Converts input value a from degrees to radians.
M_RAD2DEG(a)	Converts input value a from radians to degrees.

Constants

Macros	Descriptions
M_FT2M	Multiplication factor for conversion from foot (ft) to meter (m).
M_LB2N	Multiplication factor for conversion from pound (lb) to Newton (N).
M_SLUG2KG	Multiplication factor for conversion from slug to kilogram (kg).
M_LBFTSS2KGMM	Multiplication factor for conversion from $lb-ft/s^2$ to $kg-m^2$.
M_LBFT2NM	Multiplication factor for conversion from $lb-ft$ to $N-m$.
TRACE_OFF	Identifier for tracing of coupler curve option to be ”off” for animation.
TRACE_ON	Identifier for tracing of coupler curve option to be ”on” for animation.
QANIMATE_OUTPUTTYPE_DISPLAY	Output qanimate to the screen.

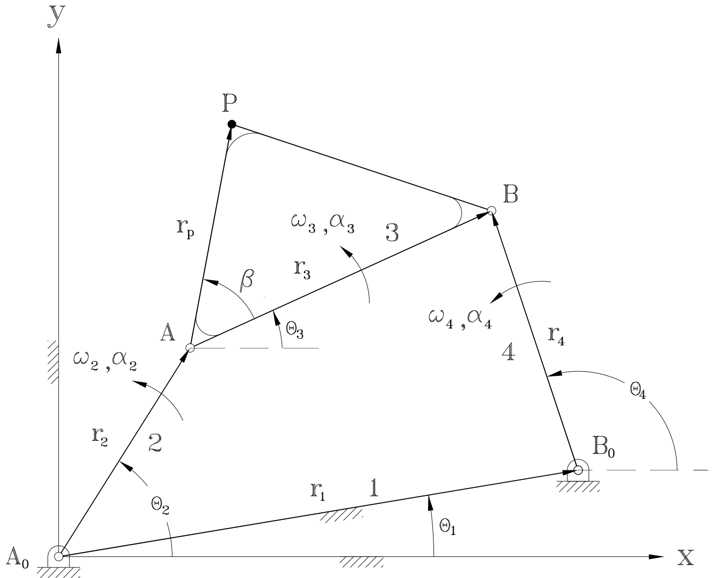
QANIMATE_OUTPUTTYPE_FILE	Save qanimate data to file.
QANIMATE_OUTPUTTYPE_STREAM	Output qanimate to the standard output stream.
COUPLER_LINK3	Coupler attached to link 3.
COUPLER_LINK4	Coupler attached to link 4.
COUPLER_LINK5	Coupler attached to link 5.
COUPLER_LINK6	Coupler attached to link 6.
COUPLER_LINK7	Coupler attached to link 7.

Appendix B

Class CFourbar

CFourbar

The header file **fourbar.h** includes header file **linkage.h**. The header file **fourbar.h** also contains a declaration of class **CFourbar**. The **CFourbar** class provides a means to analyze four-bar linkage within a Ch language environment.



Public Data
None.

Public Member Functions

Functions	Descriptions
angularAccel	Given the angular acceleration of one link, calculate the angular acceleration of other

	links.
angularAccels	Calculate angular acceleration values of links 3 and 4 with respect to time.
angularPos	Given the angle of one link, caculate the angle of other links.
angularPoss	Calculate angular poisition values for links 3 and 4 with respect to time.
angularVel	Given the angular velocity of one link, calculate the angular velocity of other links.
angularVels	Calculate angular velocity values for links 3 and 4 with respect to time.
animation	Fourbar linkage animation.
couplerCurve	Calculate the coordinates of the coupler curve.
couplerPointAccel	Calculate the acceleration of the coupler point.
couplerPointPos	Calculate the position of the coupler point.
couplerPointVel	Calculate the velocity of the coupler point.
forceTorque	Caculate the joint force and output torque at a given point.
forceTorques	Caculate the joint force and output torque in the valid range of motion.
getAngle	Calculate the angle of a given link.
getJointLimits	Calculate fourbar linkage input and output joint limits.
grashof	Calculate joint limits and decide Grashof type.
plotAngularAccels	Plot angular accelerations α_3 and α_4 vs time.
plotAngularPoss	Plot angular positions θ_3 and θ_4 vs time.
plotAngularVels	Plot angular velocities ω_3 and ω_4 vs time.
plotCouplerCurve	Plot the coupler curves.
plotForceTorques	Plot the joint forces and input torque curves.
plotTransAngles	Plot transmission angle vs θ_2.
printJointLimits	Print the joint limits of the input and output joints.
setCouplerPoint	Set parameters for the coupler point.
setGravityCenter	Set parameters for mass centers of links.
setInertia	Set inerita parameters of links.
setAngularVel	Set angular velocity of linkage 2.
setLinks	Set lengths of links.
setMass	Set masses of links.
setNumPoints	Set number of points for animation and plot coupler curve.
synthesis	Fourbar linkage position synthesis.
transAngle	Given input link position, calculate the transmission angle.
transAngles	Calculate transmission angle values for the valid range of motion.
uscUnit	Specify the use of SI or US Customary units.

Constants

The following defined constants are used by the **CFourbar** class.

Macros	Descriptions
FOURBAR_LINK1	Identifier for link 1.
FOURBAR_LINK2	Identifier for link 2.
FOURBAR_LINK3	Identifier for link 3.
FOURBAR_LINK4	Identifier for link 4.
FOURBAR_INVALID	Not a valid linkage.
FOURBAR_CRANKROCKER	Identifier for Crank-Rocker.

FOURBAR_CRANKCRANK	Identifier for Crank-Crank.
FOURBAR_ROCKERCRANK	Identifier for Rocker-Crank.
FOURBAR_ROCKERROCKER	Identifier for Rocker-Rocker.
FOURBAR_INWARDINWARD	Identifier for Inward-Inward limited.
FOURBAR_INWARDOUTWARD	Identifier for Inward-Outward limited.
FOURBAR_OUTWARDINWARD	Identifier for Outward-Inward limited.
FOURBAR_OUTWARDOUTWARD	Identifier for Outward-Outward limited.

See Also

CFourbar::angularAccel

Synopsis
#include <fourbar.h>
int angularAccel(**double** *theta[1:4]*, **double** *omega[1:4]*, **double** *alpha[1:4],* **int** *alpha_id*);

Purpose
Given the angular acceleration of one link, calculate the angular acceleration of other links.

Parameters

theta An array of double data type with angles of links.

omega An array of double data type with angular velocities of links.

alpha An array of double data type with angular accelerations of links.

alpha_id An integer number indicating the link number with a given angular acceleration.

Return Value
This function returns 0 on success and -1 on failure.

Description
Given the angular acceleration of one link, this function calculates the angular acceleration of the remaining two moving links of the fourbar. *theta* is a one-dimensional array of size 4 which stores the angle of each link. *omega* is a one-dimensional array of size 4 which stores the angular velocity of each link. *alpha* is a one-dimensional array of size 4 which stores the angular acceleration of each link. The result of calculation is stored in array *alpha*.

Example
A fourbar linkage has link lengths $r_1 = 12m, r_2 = 4m, r_3 = 12m, r_4 = 7m$, and $\theta_1 = 0$. Given the angle θ_2, the angular velocity ω_2 and the angular acceleration α_2, determine the Grashof type of the fourbar linkage, and calculate the angular acceleration α_3 and α_4 of link3 and link4 for each circuit of the linkage.

```
/****************************************************************
*  This example is for calculating the angular acceleration of
*  link3 and link4.
****************************************************************/
```

```
#include <math.h>
#include <stdio.h>
#include <fourbar.h>

int main() {
    double r[1:4], theta[1:4],theta_2[1:4],omega[1:4],omega_2[1:4];
    double alpha[1:4],alpha_2[1:4];

    /* default specification of the four-bar linkage */
    r[1] = 12; r[2] = 4; r[3] = 12; r[4] = 7;
    theta[1] = 0*M_PI/180; theta[2]=150*M_PI/180;
    omega[2]=5;  /*  rad/sec  */
    alpha[2]=-5;  /*  rad/sec*sec  */

    int fourbartype;
    CFourbar fourbar;
    fourbartype = fourbar.grashof(NULL);

    if (fourbartype== FOURBAR_INVALID) {
      exit(1);
    }

    //###############################
    //     print_alpha34( )
    //###############################
    void print_alpha34(double a3,a4)
    {
      printf("\tDeg/sec*sec:\t alpha3=%6.3f, \t alpha4=%6.3f\t\n",a3*180/M_PI,
                 a4*180/M_PI);
      printf("\tRad/sec*sec:\t alpha3=%6.4f, \t alpha4=%6.4f\t\n",a3,a4);
    }

    theta_2[2] = theta[2];
    omega_2[2] = omega[2];
    alpha_2[2] = alpha[2];
    /*find alpha3, alpha4*/
    fourbar.angularPos(theta, theta_2, FOURBAR_LINK2);
    if((fourbartype==FOURBAR_ROCKERROCKER)||(fourbartype==FOURBAR_ROCKERCRANK)) {
       fourbar.angularVel(theta, omega, FOURBAR_LINK2);
       fourbar.angularAccel(theta, omega, alpha, FOURBAR_LINK2);
       fourbar.angularVel(theta_2, omega_2, FOURBAR_LINK2);
       fourbar.angularAccel(theta_2, omega_2, alpha_2, FOURBAR_LINK2);
       printf("\n Circuit 1: Angular Accelerations\n\n");
       print_alpha34(alpha[3],alpha[4]);
       printf("\n Circuit 2: Angular Accelerations\n\n");
       print_alpha34(alpha_2[3],alpha_2[4]);
    } else {
       fourbar.angularVel(theta, omega, FOURBAR_LINK2);
       fourbar.angularAccel(theta, omega, alpha, FOURBAR_LINK2);
       fourbar.angularVel(theta_2, omega_2, FOURBAR_LINK2);
       fourbar.angularAccel(theta_2, omega_2, alpha_2, FOURBAR_LINK2);
       printf("\n Circuit 1: Angular Accelerations\n\n");
       print_alpha34(alpha[3],alpha[4]);
       printf("\n Circuit 2: Angular Accelerations\n\n");
       print_alpha34(alpha_2[3],alpha_2[4]);
    }
}
```

Output

```
 Circuit 1: Angular Accelerations

Deg/sec*sec:  alpha3=230.218,   alpha4=-558.101
Rad/sec*sec:  alpha3=4.0181,   alpha4=-9.7407

 Circuit 2: Angular Accelerations

Deg/sec*sec:  alpha3=-217.011,   alpha4=571.309
Rad/sec*sec:  alpha3=-3.7875,   alpha4=9.9712
```

CFourbar::angularAccels

Synopsis
#include <fourbar.h>
int angularAccels(int *branchnum*, **double** *time[:]*, **double** *alpha3[:]*, **double** *alpha4[:]*);

Purpose
Calculate α_3 and α_4 values for the valid range of motion.

Parameters

branchnum An integer used to indicate the branch of the fourbar.

time An array for time values.

alpha3 An array for α_3 values.

alpha4 An array for α_4 values.

Return Value
This function returns 0 on success and -1 on failure.

Description
This function calculates the values for α_3 and α_4 for the valid range of motion. `branchnum` is the branch of the fourbar. `time` is an array to record time. `alpha3` and `alpha4` are arrays for storing the angular position values for links 3 and 4.

Example
For a fourbar linkage with link lengths $r_1 = 12m, r_2 = 4m, r_3 = 12m, r_4 = 7m$, and angle $\theta_1 = 10^o$, plot the angular acceleration curves for links 3 and 4.

```
/****************************************************************
*   This example is for generating angular acceleration values
*   for alpha3 and alpha4 within the valid range of motion.
****************************************************************/
#include <math.h>
#include <fourbar.h>
```

```
int main() {
    CFourbar fourbar;
    double r1 = 12, r2 = 4, r3 =12, r4 = 7, theta1 = 10*M_PI/180;
    double omega2 = 5;      // rad/sec
    int i, numpoints = 50;
    double t[numpoints], alpha3[numpoints], alpha4[numpoints];
    CPlot plot;

    fourbar.setLinks(r1, r2, r3, r4, theta1);
    fourbar.setAngularVel(omega2);
    fourbar.angularAccels(1, t, alpha3, alpha4);
    for(i = 0; i < numpoints; i++) {
       alpha3[i] = M_RAD2DEG(alpha3[i]);
       alpha4[i] = M_RAD2DEG(alpha4[i]);
    }
    plot.data2D(t, alpha3);
    plot.data2D(t, alpha4);
    plot.title("Angular Accelerations Plots");
    plot.label(PLOT_AXIS_X, "time (sec)");
    plot.label(PLOT_AXIS_Y, "alpha (deg/sec^2)");
    plot.legend("alpha3", 0);
    plot.legend("alpha4", 1);
    plot.border(PLOT_BORDER_ALL, PLOT_ON);
    plot.plotting();
}
```

Output
see output for **CFourbar**::**plotAngularAccels**() example.

CFourbar::angularPos

Synopsis
#include <fourbar.h>
int angularPos(**double** *theta_1[1:]*, **double** *theta_2[1:]*, **int** *theta_id*);

Purpose
Given the angle of one link, calculate the angle of other links.

Parameters

theta_1 A double array with dimension size of 4 for the first solution.

theta_2 A double array with dimension size of 4 for the second solution.

theta_id An integer number indicating the known link number with a position angle.

Return Value
This function returns 0 on success and -1 on failure.

Description
Given the angular position of one link of a fourbar linkage, this function computes the angular positions of the remaining two moving links. *theta_1* is a one-dimensional array of size 4 which stores the first solution of angular. *theta_2* is a one-dimensional array of size 4 which stores the second solution of angular. *theta_id*

is an integer number indicating the link number of the known angular position.

Example
A fourbar linkage has link lengths $r_1 = 12m, r_2 = 4m, r_3 = 12m, r_4 = 7m$, and $\theta_1 = 10$. Given the angle θ_2, calculate the angular position θ_3 and θ_4 of link3 and link4, and the coupler point position for each circuit of the linkage.

```
/*************************************************************
*   This example is for calculating the theta3 and theta4 with
*   given theta2.
*************************************************************/
#include <math.h>
#include <fourbar.h>
int main() {
  CFourbar fourbar;
  double r1 = 12, r2 = 4, r3 =12, r4 = 7, theta1 = 10*M_PI/180;
  double rp = 5, beta = 20*M_PI/180;
  double theta_1[1:4], theta_2[1:4];
  double complex p1, p2;  // tow solution of coupler point P
  double theta2 = 70*M_PI/180;

  theta_1[1] = theta1;
  theta_1[2] = theta2; // theta2
  theta_2[1] = theta1;
  theta_2[2] = theta2; // theta2
  fourbar.setLinks(r1, r2, r3, r4, theta1);
  fourbar.setCouplerPoint(rp, beta);
  fourbar.angularPos(theta_1, theta_2, FOURBAR_LINK2);
  fourbar.couplerPointPos(theta2, p1, p2);

  /**** the first set of solutions ****/
  printf("theta3 = %6.3f, theta4 = %6.3f, P = %6.3f \n", theta_1[3], theta_1[4], p1);
  /**** the second set of solutions ****/
  printf("theta3 = %6.3f, theta4 = %6.3f, P = %6.3f \n", theta_2[3], theta_2[4], p2);
}
```

Output

```
theta3 =  0.459, theta4 =  1.527, P = complex( 4.822, 7.374)
theta3 = -0.777, theta4 = -1.845, P = complex( 5.917, 1.684)
```

CFourbar::angularPoss

Synopsis
#include <fourbar.h>
int angularPoss(int *branchnum*, **double** *time[:]*, **double** *theta3[:]*, **double** *theta4[:]*);

Purpose
Calculate θ_3 and θ_4 values for the valid range of motion.

Parameters

branchnum An integer used to indicate the branch of the fourbar.

time An array for time values.

theta3 An array for θ_3 values.

theta4 An array for θ_4 values.

Return Value
This function returns 0 on success and -1 on failure.

Description
This function calculates the values for θ_3 and θ_4 for the valid range of motion. `branchnum` is the branch of the fourbar. `time` is an array to record time. `theta3` and `theta4` are arrays for storing the angular position values for links 3 and 4.

Example
For a fourbar linkage with link lengths $r_1 = 12m, r_2 = 4m, r_3 = 12m, r_4 = 7m$, and angle $\theta_1 = 10^o$, plot the angular position curves for links 3 and 4.

```
/**************************************************************
*   This example is for generating angular position values
*   for theta3 and theta4 within the valid range of motion.
**************************************************************/
#include <math.h>
#include <fourbar.h>

int main() {
    CFourbar fourbar;
    double r1 = 12, r2 = 4, r3 =12, r4 = 7, theta1 = 10*M_PI/180;
    double omega2 = 5;      // rad/sec
    int i, numpoints = 50;
    double t[numpoints], theta3[numpoints], theta4[numpoints];
    CPlot plot;

    fourbar.setLinks(r1, r2, r3, r4, theta1);
    fourbar.setAngularVel(omega2);
    fourbar.angularPoss(1, t, theta3, theta4);
    for(i = 0; i < numpoints; i++) {
       theta3[i] = M_RAD2DEG(theta3[i]);
       theta4[i] = M_RAD2DEG(theta4[i]);
    }
    plot.data2D(t, theta3);
    plot.data2D(t, theta4);
    plot.title("Angular Position Plots");
    plot.label(PLOT_AXIS_X, "time (sec)");
    plot.label(PLOT_AXIS_Y, "theta (deg)");
    plot.legend("theta3", 0);
    plot.legend("theta4", 1);
    plot.border(PLOT_BORDER_ALL, PLOT_ON);
    plot.plotting();
}
```

Output
see output for **CFourbar::plotAngularPoss**() example.

CFourbar::angularVel

Synopsis
#include <fourbar.h>
int angularVel(double *theta[1:4],* **double** *omega[1:4],* **int** *omega_id*);

Purpose
Given the angular velocity of one link, calculate the angular velocity of other links.

Parameters

theta A double array used for the input angle of links.

omega A doulbe array used for the angular velocities of links.

omega_id An integer number indicating the link number with a given angular velocity.

Return Value
This function returns 0 on success and -1 on failure.

Description
Given the angular velocity of one link, this function calculates the angular velocities of the remaining two moving links of the fourbar. *theta* is an array for link positions. *omega* is an array for angular velocity of links. *omega_id* is an integer number for identifier for known link angular velocity.

Example
A fourbar linkage has link lengths $r_1 = 12m, r_2 = 4m, r_3 = 12m, r_4 = 7m$, and $\theta_1 = 0$. Given the angle θ_2 and the angular velocity ω_2, determine the Grashof type of the fourbar linkage, and calculate the angular velocities ω_3 and ω_4 of link3 and link4 for each circuit of the linkage.

```
/*******************************************************************
*   This example is for calculating the angular velocity of
*   link3 and link4.
*******************************************************************/
#include <math.h>
#include <stdio.h>
#include <fourbar.h>

int main(int argc, char* argv[]) {
    double r[1:4], theta[1:4],theta_2[1:4],omega[1:4],omega_2[1:4];
    int fourbartype;
    CFourbar fourbar;

    /* default specification of the four-bar linkage */
    r[1] = 12; r[2] = 4; r[3] = 12; r[4] = 7;
    theta[1] = 0*M_PI/180; theta[2]=150*M_PI/180;
    omega[2]=5;  /*  rad/sec  */
    fourbar.setLinks(r[1], r[2], r[3], r[4], theta[1]);
    /* grashof analysis */
    fourbartype = fourbar.grashof(NULL);

    if (fourbartype == FOURBAR_INVALID) {
       exit(1);
```

```
    }
    theta_2[2]=theta[2];              /*  second circuit  */
    fourbar.angularPos(theta, theta_2, FOURBAR_LINK2);


    //##############################
    //     print_omega34( )
    //##############################
    void print_omega34(double w3,w4)
    {
       printf("\tDeg/sec:\t omega3=%6.3f, \t omega4=%6.3f\t\n",w3*180/M_PI,
                w4*180/M_PI);
       printf("\tRad/sec:\t omega3=%6.4f, \t omega4=%6.4f\t\n",w3,
                w4);
    }
    /*find omega3, omega4*/
    omega_2[2] = omega[2];
    if((fourbartype == FOURBAR_ROCKERCRANK)||(fourbartype == FOURBAR_ROCKERROCKER)) {
       fourbar.angularVel(theta, omega, FOURBAR_LINK2);
       fourbar.angularVel(theta_2, omega_2, FOURBAR_LINK2);
       printf("\n Circuit 1: Angular Velocities\n\n");
       print_omega34(omega[3],omega[4]);
       printf("\n Circuit 2: Angular Velocities\n\n");
       print_omega34(omega_2[3],omega_2[4]);
    }  else {
       fourbar.angularVel(theta, omega, FOURBAR_LINK2);
       fourbar.angularVel(theta_2, omega_2, FOURBAR_LINK2);
       printf("\n Circuit 1: Angular Velocities\n\n");
       print_omega34(omega[3],omega[4]);
       printf("\n Circuit 2: Angular Velocities\n\n");
       print_omega34(omega_2[3],omega_2[4]);
    }
}
```

Output

```

 Circuit 1: Angular Velocities

Deg/sec:  omega3=41.706,   omega4=127.465
Rad/sec:  omega3=0.7279,   omega4=2.2247

 Circuit 2: Angular Velocities

Deg/sec:  omega3=93.957,   omega4= 8.197
Rad/sec:  omega3=1.6399,   omega4=0.1431
```

CFourbar::angularVels

Synopsis

#include <fourbar.h>

int angularVels(int *branchnum*, **double** *time[:]*, **double** *omega3[:]*, **double** *omega4[:]*);

Purpose

Calculate ω_3 and ω_4 values for the valid range of motion.

Parameters

branchnum An integer used to indicate the branch of the fourbar.

time An array for time values.

omega3 An array for ω_3 values.

omega4 An array for ω_4 values.

Return Value

This function returns 0 on success and -1 on failure.

Description

This function calculates the values for ω_3 and ω_4 for the valid range of motion. `branchnum` is the branch of the fourbar. `time` is an array to record time. `omega3` and `omega4` are arrays for storing the angular positions values for links 3 and 4.

Example

For a fourbar linkage with link lengths $r_1 = 12m, r_2 = 4m, r_3 = 12m, r_4 = 7m$, and angle $\theta_1 = 10^o$, plot the angular velocity curves for links 3 and 4.

```
/***************************************************************
*   This example is for generating angular velocity values
*   for omega3 and omega4 within the valid range of motion.
***************************************************************/
#include <math.h>
#include <fourbar.h>

int main() {
    CFourbar fourbar;
    double r1 = 12, r2 = 4, r3 =12, r4 = 7, theta1 = 10*M_PI/180;
    double omega2 = 5;     // rad/sec
    int i, numpoints = 50;
    double t[numpoints], omega3[numpoints], omega4[numpoints];
    CPlot plot;

    fourbar.setLinks(r1, r2, r3, r4, theta1);
    fourbar.setAngularVel(omega2);
    fourbar.angularVels(1, t, omega3, omega4);
    for(i = 0; i < numpoints; i++) {
       omega3[i] = M_RAD2DEG(omega3[i]);
       omega4[i] = M_RAD2DEG(omega4[i]);
    }
    plot.data2D(t, omega3);
    plot.data2D(t, omega4);
    plot.title("Angular Velocity Plots");
    plot.label(PLOT_AXIS_X, "time (sec)");
    plot.label(PLOT_AXIS_Y, "omega (deg/rad)");
    plot.legend("omega3", 0);
    plot.legend("omega4", 1);
    plot.border(PLOT_BORDER_ALL, PLOT_ON);
    plot.plotting();
}
```

Output
see output for **CFourbar**::**plotAngularVels**() example.

CFourbar::animation

Synopsis
#include <fourbar.h>
int animation(**int** *branchnum*, ... /* [**int** *outputtype*, **string_t** *datafilename*] */);

Syntax
animation(*branchnum*)
animation(*branchnum*, *outputtype*)
animation(*branchnum*, *outputtype*, *datafilename*)

Purpose
An animation of a fourbar mechanism.

Parameters

branchnum an integer used to indicate which branch will be drawn. Each fourbar linkage has 2 branches except the Rocker-Rocker, which has 4 branches.

outputtype an optional parameter to specify the output type of the animation.

datafilename an optional parameter to specify the output file name.

Return Value
This function returns 0 on success and -1 on failure.

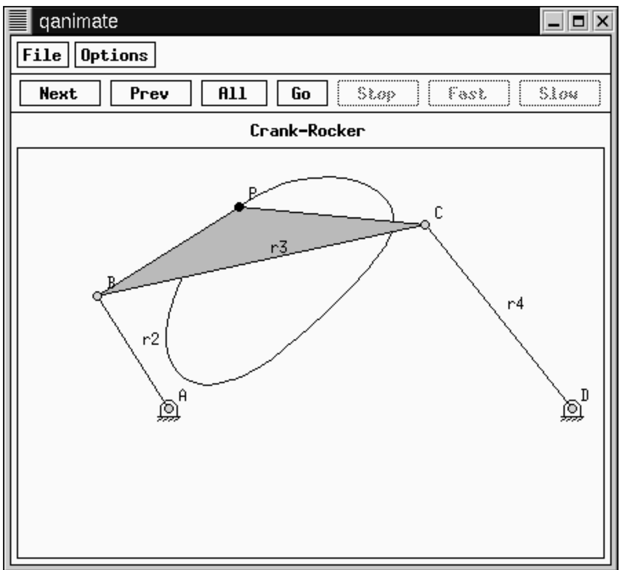
Description
This function simulates the motion of a fourbar mechanism. *branchnum* is an integer number which indicates the branch to be drawn. *outputtype* is an optional parameter used to specify how the animation should be outputted. *outputtype* can be either of the following macros: QANIMATE_OUTPUTTYPE_DISPLAY, QANIMATE_OUTPUTTYPE_FILE, QANIMATE_OUTPUTTYPE_STREAM.
QANIMATE_OUTPUTTYPE_DISPLAY outputs the animation onto the computer terminal.
QANIMATE_OUTPUTTYPE_FILE writes the animation data onto a file.
QANIMATE_OUTPUTTYPE_STREAM outputs the animation to the standard out. *datafilename* is an optional parameter to specify the output file name.

Example 1
For a Crank-Rocker fourbar linkage with $r_1 = 12m, r_2 = 4m, r_3 = 12m, r_4 = 7m, and \theta_1 = 0$, simulate the motion of the fourbar linkage.

```
/* Crank - Rocker */
#include <stdio.h>
#include <fourbar.h>

int main() {
    /* default specification of the four-bar linkage */
    double r1 = 12, r2 = 4, r3 = 10, r4= 7;//cranker-rocker
    double theta1 = 0*M_PI/180;
```

```
    double rp = 5, beta = 20*M_PI/180;
    int numpoints =50;
    CFourbar fourbar;

    fourbar.setLinks(r1, r2, r3, r4, theta1);
    fourbar.setCouplerPoint(rp, beta, TRACE_ON);
    fourbar.setNumPoints(numpoints);
    fourbar.animation(1);
}
```

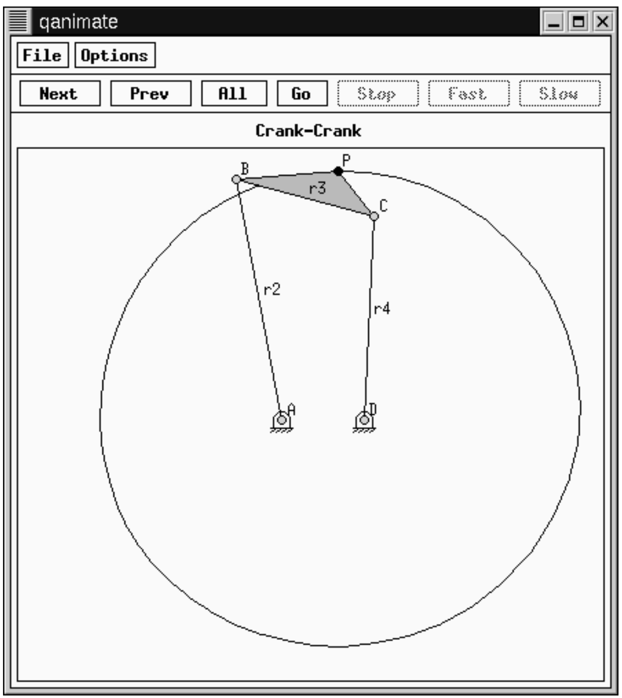
Output



Example 2
For a Crank-Crank fourbar linkage with $r_1 = 4m, r_2 = 12m, r_3 = 7m, r_4 = 10m, and \theta_1 = 0$, simulate the motion of the fourbar linkage.

```
/* Crank - Crank */
#include <stdio.h>
#include <fourbar.h>

int main() {
    /* default specification of the four-bar linkage */
    double r1 = 4, r2 = 12, r3 = 7, r4 = 10; // crank-crank
    double theta1 = 0*M_PI/180;
    double rp = 5, beta = 20*M_PI/180;
    int numpoints =50;
    CFourbar fourbar;

    fourbar.setLinks(r1, r2, r3, r4, theta1);
    fourbar.setCouplerPoint(rp, beta, TRACE_ON);
    fourbar.setNumPoints(numpoints);
    fourbar.animation(1);
}
```

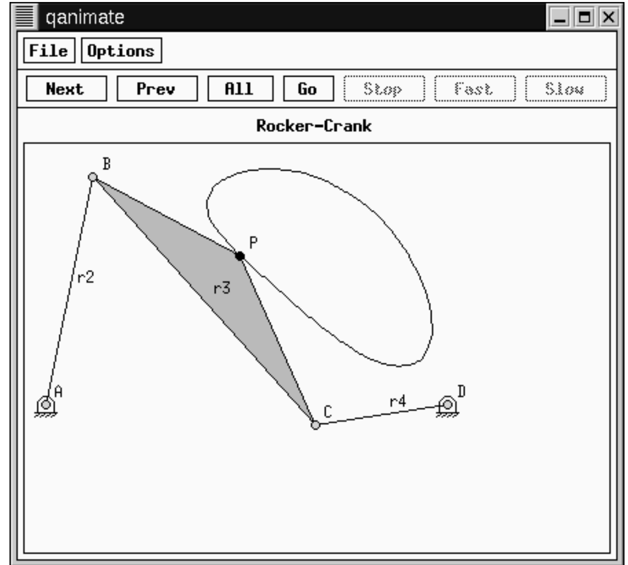
Output



Example 3
For a Rocker-Crank fourbar linkage with $r_1 = 12m, r_2 = 7m, r_3 = 10m, r_4 = 4m, and \theta_1 = 0$, simulate the motion of the fourbar linkage.

```
/* Rocker - Crank */
#include <stdio.h>
#include <fourbar.h>

int main() {
    /* default specification of the four-bar linkage */
    double r1 = 12, r2 = 7, r3 = 10, r4= 4; // rocker - crank
    double theta1 = 0*M_PI/180;
    double rp = 5, beta = 20*M_PI/180;
    int numpoints =50;
    CFourbar fourbar;

    fourbar.setLinks(r1, r2, r3, r4, theta1);
    fourbar.setCouplerPoint(rp, beta, TRACE_ON);
    fourbar.setNumPoints(numpoints);
    fourbar.animation(1);
}
```

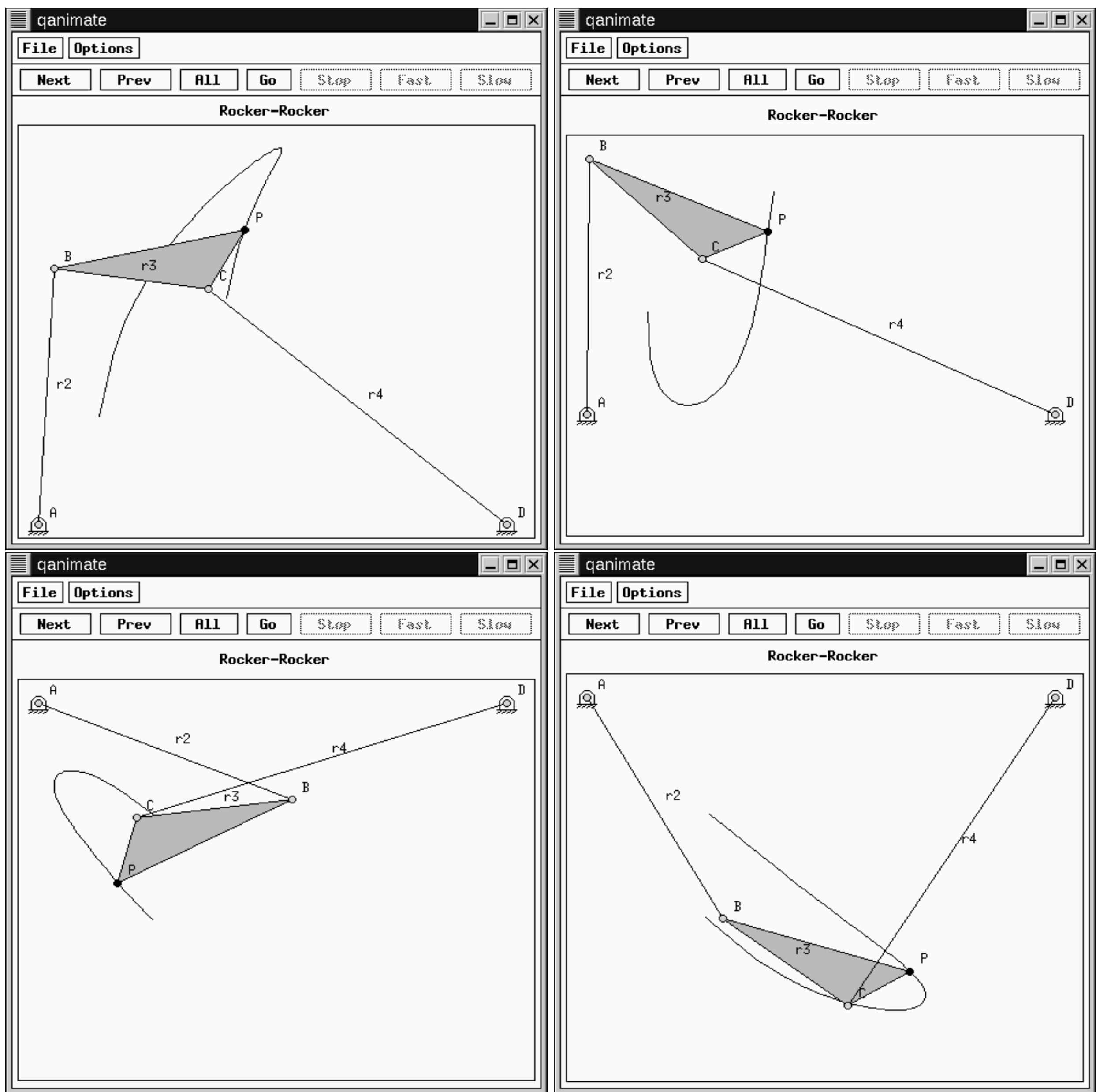
Output



Example 4
For a Rocker-Rocker fourbar linkage with $r_1 = 12m, r_2 = 7m, r_3 = 4m, r_4 = 10m, and\theta_1 = 0$, simulate the motion of the fourbar linkage.

```
/* Rocker - Rocker */
#include <stdio.h>
#include <fourbar.h>

int main() {
    /* default specification of the four-bar linkage */
    double r1 = 12, r2 = 7, r3 = 4, r4= 10; // rocker - rocker
    double theta1 = 0*M_PI/180;
    double rp = 5, beta = 20*M_PI/180;
    int numpoints =50;
    CFourbar fourbar;

    fourbar.setLinks(r1, r2, r3, r4, theta1);
    fourbar.setCouplerPoint(rp, beta, TRACE_ON);
    fourbar.setNumPoints(numpoints);
    fourbar.animation(1);
    fourbar.animation(2, QANIMATE_OUTPUTTYPE_FILE, "data.qnm"); // output the animation
                                                                // data to file
    fourbar.animation(3);
    fourbar.animation(4);
}
```

Output

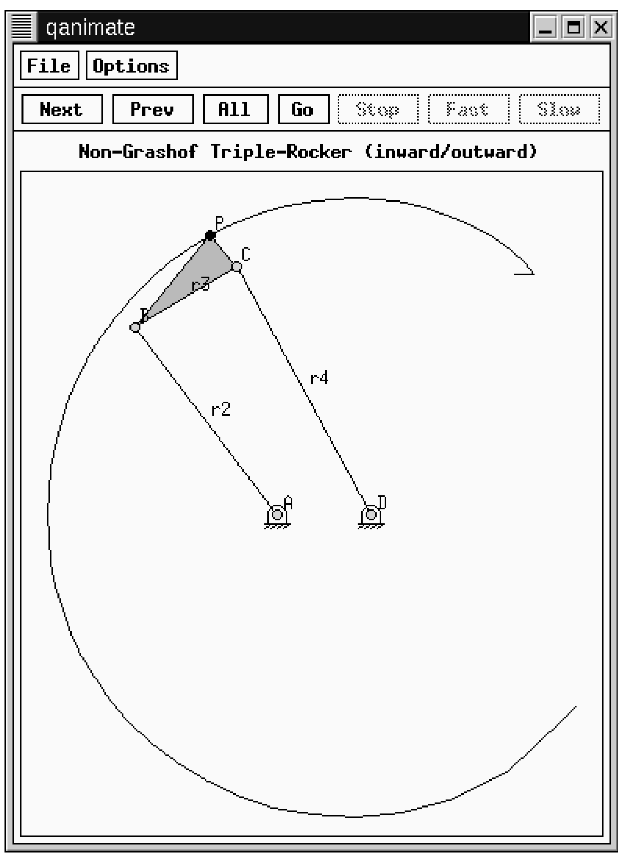


Example 5
For an Inward-Outward fourbar linkage with $r_1 = 4m, r_2 = 10m, r_3 = 5m, r_4 = 12m, and \theta_1 = 0$, simulate the motion of the fourbar linkage.

```
/* Inward - Outward */
#include <stdio.h>
#include <fourbar.h>

int main() {
    /* default specification of the four-bar linkage */
    double r1 = 4, r2 = 10, r3 = 5, r4 = 12; // inward /outward
    double theta1 = 0*M_PI/180;
    double rp = 5, beta = 20*M_PI/180;
    int numpoints = 50;
    CFourbar fourbar;
```

```
    fourbar.setLinks(r1, r2, r3, r4, theta1);
    fourbar.setCouplerPoint(rp, beta, TRACE_ON);
    fourbar.setNumPoints(numpoints);
    fourbar.animation(1);
}
```

Output

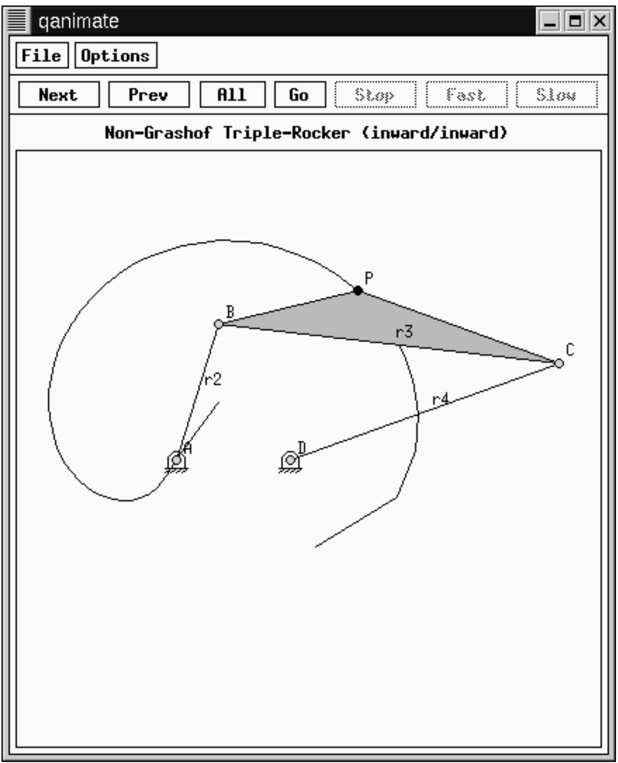


Example 6
For an Inward-Inward fourbar linkage with $r_1 = 4m, r_2 = 5m, r_3 = 12m, r_4 = 10m, and \theta_1 = 0$, simulate the motion of the fourbar linkage.

```
/* Inward - Inward */
#include <stdio.h>
#include <fourbar.h>

int main() {
    /* default specification of the four-bar linkage */
    double r1 = 4, r2 = 5, r3 = 12, r4 = 10; // inward /inward
    double theta1 = 0*M_PI/180;
    double rp = 5, beta = 20*M_PI/180;
    int numpoints =50;
    CFourbar fourbar;

    fourbar.setLinks(r1, r2, r3, r4, theta1);
    fourbar.setCouplerPoint(rp, beta, TRACE_ON);
    fourbar.setNumPoints(numpoints);
```

```
    fourbar.animation(1);
}
```

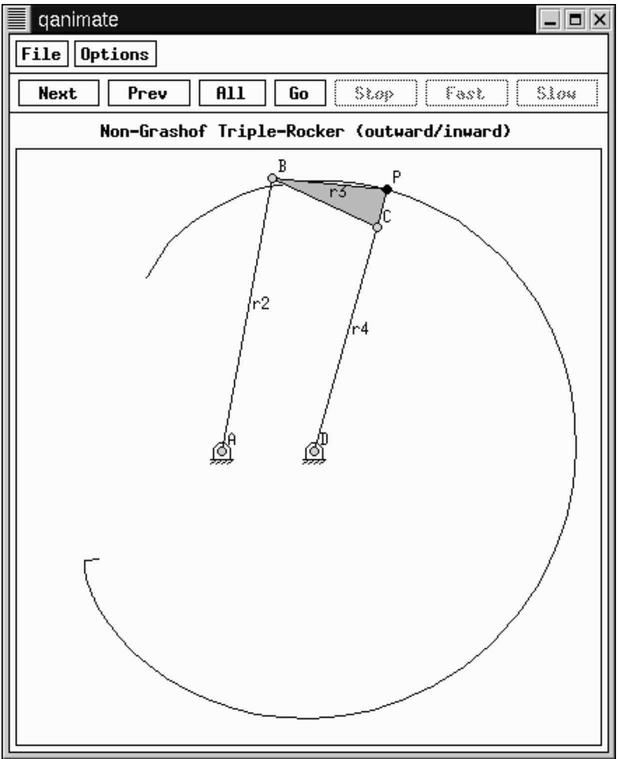
Output



Example 7
For an Outward-Inward fourbar linkage with $r_1 = 4m, r_2 = 12m, r_3 = 5m, r_4 = 10m, and \theta_1 = 0$, simulate the motion of the fourbar linkage.

```
/* Outward - Inward */
#include <stdio.h>
#include <fourbar.h>

int main() {
    /* default specification of the four-bar linkage */
    double r1 = 4, r2 = 12, r3 = 5, r4 = 10; // outward /inward
    double theta1 = 0*M_PI/180;
    double rp = 5, beta = 20*M_PI/180;
    int numpoints =50;
    CFourbar fourbar;

    fourbar.setLinks(r1, r2, r3, r4, theta1);
    fourbar.setCouplerPoint(rp, beta, TRACE_ON);
    fourbar.setNumPoints(numpoints);
    fourbar.animation(1);
}
```

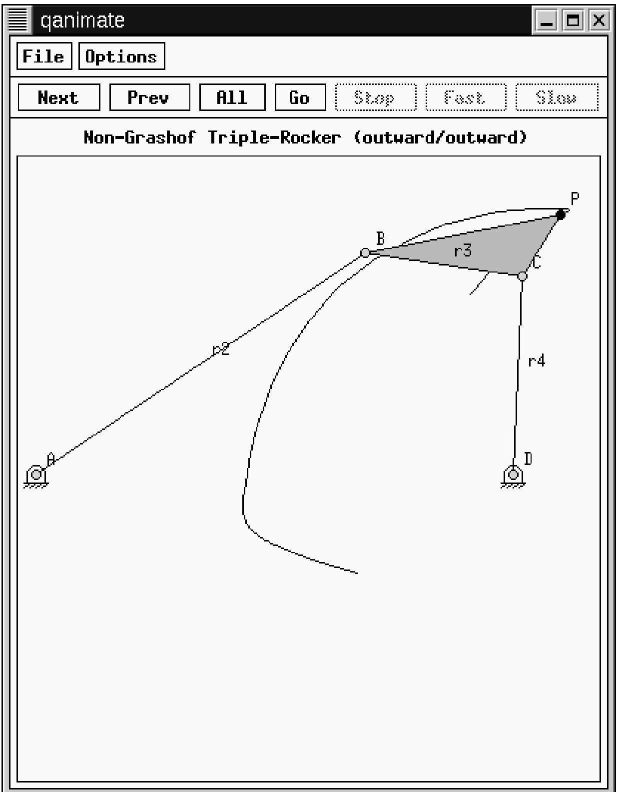
Output



Example 8
For an Outward-Outward fourbar linkage with $r_1 = 12m, r_2 = 10m, r_3 = 4m, r_4 = 5m, and \theta_1 = 0$, simulate the motion of the fourbar linkage.

```
/* Outward - Outward */
#include <stdio.h>
#include <fourbar.h>

int main() {
    /* default specification of the four-bar linkage */
    double r1 = 12, r2 = 10, r3 = 4, r4 = 5; // outward /outward
    double theta1 = 0*M_PI/180;
    double rp = 5, beta = 20*M_PI/180;
    int numpoints =50;
    CFourbar fourbar;

    fourbar.setLinks(r1, r2, r3, r4, theta1);
    fourbar.setCouplerPoint(rp, beta, TRACE_ON);
    fourbar.setNumPoints(numpoints);
    fourbar.animation(1);
}
```

Output



CFourbar::couplerCurve

Synopsis
#include <fourbar.h>
void couplerCurve(int *branchnum,* **double** *curvex[:],* **double** *curvey[:]*);

Purpose
Calculate the coordinates of the coupler curve.

Parameters

branchnum An integer number used for the branch which will be calculated.

curvex[:] A double array used for the x coordinate of coupler point with different input angles.

curvey[:] A double array used for the y coordinate of coupler point with different input angles.

Return Value
None.

Description
This function calculate the coupler point position by looping the input angle. *curvex ,curvey* is the solution of the coupler point positon.

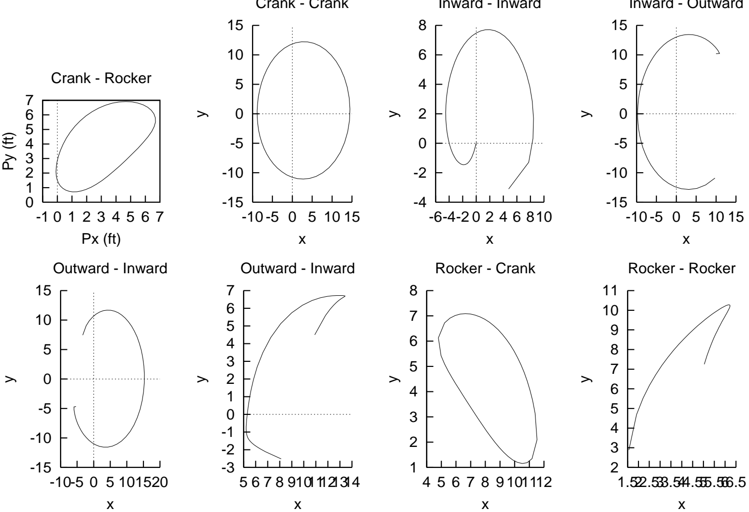
Example

For different types of the fourbar linkages, given the length of each link, draw the coupler point position curve.

```
/* Crank - Crank */
#include <fourbar.h>

int main() {
    CFourbar fourbar;
    double r1, r2, r3, r4;
    double theta1 = 0;
    double rp = 5, beta = 20*M_PI/180;
    class CPlot pl, *spl;
    int num_range;

    double RetCurvex[50], RetCurvey[50];


    fourbar.setCouplerPoint(rp, beta);
    fourbar.setNumPoints(50);

    pl.subplot(2,4);
    // Crank - Rocker
    r1 = 12; r2 = 4; r3 = 10; r4= 7;
    spl = pl.getSubplot(0,0);
    fourbar.setLinks(r1, r2, r3, r4, theta1);
    fourbar.couplerCurve(1, RetCurvex, RetCurvey);
    spl->label(PLOT_AXIS_X,"Px (ft)");
    spl->label(PLOT_AXIS_Y,"Py (ft)");
    spl->title("Crank - Rocker");
    spl->data2D(RetCurvex, RetCurvey);

    // Crank - Crank
    r1 = 4; r2 = 12; r3 = 7; r4 = 10;
    spl = pl.getSubplot(0,1);
    fourbar.setLinks(r1, r2, r3, r4, theta1);
    fourbar.couplerCurve(1, RetCurvex, RetCurvey);
    spl->title("Crank - Crank");
    spl->data2D(RetCurvex, RetCurvey);

    //Inward - Inward
    r1 = 4; r2 = 5; r3 = 12; r4 = 10;
    spl = pl.getSubplot(0,2);
    fourbar.setLinks(r1, r2, r3, r4, theta1);
    fourbar.couplerCurve(1, RetCurvex, RetCurvey);
    spl->title("Inward - Inward");
    spl->data2D(RetCurvex, RetCurvey);

    //Inward - Outward
    r1 = 4; r2 = 10; r3 = 5; r4 = 12;
    spl = pl.getSubplot(0,3);
    fourbar.setLinks(r1, r2, r3, r4, theta1);
    fourbar.couplerCurve(1, RetCurvex, RetCurvey);
    spl->title("Inward - Outward");
    spl->data2D(RetCurvex, RetCurvey);

    //Outward - Inward
    r1 = 4; r2 = 12; r3 = 5; r4 = 10;
    spl = pl.getSubplot(1,0);
```

```
    fourbar.setLinks(r1, r2, r3, r4, theta1);
    fourbar.couplerCurve(1, RetCurvex, RetCurvey);
    spl->title("Outward - Inward");
    spl->data2D(RetCurvex, RetCurvey);

    //Outward - Outward
    r1 = 12; r2 = 10; r3 = 4; r4 = 5;
    spl = pl.getSubplot(1,1);
    fourbar.setLinks(r1, r2, r3, r4, theta1);
    fourbar.couplerCurve(1, RetCurvex, RetCurvey);
    spl->title("Outward - Inward");
    spl->data2D(RetCurvex, RetCurvey);

    //Rocker - Crank
    r1 = 12; r2 = 7; r3 = 10; r4= 4;
    spl = pl.getSubplot(1,2);
    fourbar.setLinks(r1, r2, r3, r4, theta1);
    fourbar.couplerCurve(1, RetCurvex, RetCurvey);
    spl->title("Rocker - Crank");
    spl->data2D(RetCurvex, RetCurvey);

    //Rocker - Rocker
    r1 = 12; r2 = 7; r3 = 4; r4= 10;
    spl = pl.getSubplot(1,3);
    fourbar.setLinks(r1, r2, r3, r4, theta1);
    fourbar.couplerCurve(1, RetCurvex, RetCurvey);
    spl->title("Rocker - Rocker");
    spl->data2D(RetCurvex, RetCurvey);

    pl.sizeRatio(-1);
    pl.border(PLOT_BORDER_ALL, PLOT_ON);
    pl.plotting();
}
```

Output



See Also

CFourbar::couplerPointAccel

Synopsis
#include <fourbar.h>
double complex couplerPointAccel(double *theta2,* **double** *theta3,* **double** *omega2,* **double** *omega3,* **double** *alpha2,* **double** *alpha3*);

Purpose
Calculate the acceleration of the coupler point.

Parameters

theta2 A double number used for the angle of link 2.

theta3 A double number used for the angle of link 3.

omega2 A double number used for the angular velocity of link 2.

omega3 A double number used for the angular velocity of link 3.

alpha2 A double number used for the angular acceleration of link 2.

alpha3 A double number used for the angular acceleration of link 3.

Return Value
This function returns the acceleration of the coupler.

Description
This function calculates the acceleration of the coupler point. *theta2, theta3, omega2, omega3, alpha2, alpha3* are double numbers. The return value is a complex number.

Example
A fourbar linkage has link lengths $r_1 = 12m, r_2 = 4m, r_3 = 12m, r_4 = 7m, r_p = 5m$, and angles $\beta = 20$, $\theta_1 = 10$. Given the angle θ_2, the angular velocity ω_2 and the angular acceleration α_2, determine the Grashof type of the fourbar linkage, and calculate the acceleration of the coupler point for each circuit, respectively.

```
/****************************************************************
*  This example calculates the acceleration of coupler point.
****************************************************************/
#include <fourbar.h>

int main()
{
    CFourbar fourbar;
    int fourbartype;
    double r1 = 12, r2 = 4, r3 =12, r4 = 7, theta1 = 10*M_PI/180;
    double rp = 5, beta = 20*M_PI/180;
    double theta_1[1:4], theta_2[1:4];
    double omega[1:4], alpha[1:4], omega_2[1:4], alpha_2[1:4];
    double theta2 = 70*M_PI/180;
    double complex Ap[1:2];

    omega[2]=5;  /*  rad/sec  */
    alpha[2]=-5;  /*  rad/sec*sec  */
```

```
    theta_1[1] = theta1;
    theta_1[2] = theta2; // theta2
    theta_2[1] = theta1;
    theta_2[2] = theta2; // theta2

    fourbar.setLinks(r1, r2, r3, r4, theta1);
    fourbar.setCouplerPoint(rp, beta);

    fourbartype = fourbar.grashof(NULL);

    if (fourbartype == FOURBAR_INVALID) {
      exit(1);
    }

    omega_2[2] = omega[2];
    alpha_2[2] = alpha[2];

    /*find alpha3, alpha4*/
    fourbar.angularPos(theta_1, theta_2, FOURBAR_LINK2);
    if((fourbartype == FOURBAR_ROCKERCRANK) || (fourbartype == FOURBAR_ROCKERROCKER)) {
       fourbar.angularVel(theta_1, omega, FOURBAR_LINK2);
       fourbar.angularAccel(theta_1, omega, alpha, FOURBAR_LINK2);
       fourbar.angularVel(theta_2, omega_2, FOURBAR_LINK2);
       fourbar.angularAccel(theta_2, omega_2, alpha_2, FOURBAR_LINK2);
       Ap[1] = fourbar.couplerPointAccel(theta_1[2], theta_1[3],
                      omega[2], omega[3], alpha[2], alpha[3]);
       printf("Circuit 1: \n  CouplerAccleration: %f \n", Ap[1]);
       Ap[2] = fourbar.couplerPointAccel(theta_2[2], theta_2[3],
                      omega_2[2], omega_2[3], alpha_2[2], alpha_2[3]);
       printf("Circuit 2: \n  CouplerAccleration: %f \n", Ap[2]);

    } else  {
       fourbar.angularVel(theta_1, omega, FOURBAR_LINK2);
       fourbar.angularAccel(theta_1, omega, alpha, FOURBAR_LINK2);
       fourbar.angularVel(theta_2, omega_2, FOURBAR_LINK2);
       fourbar.angularAccel(theta_2, omega_2, alpha_2, FOURBAR_LINK2);
       Ap[1] = fourbar.couplerPointAccel(theta_1[2], theta_1[3],

                      omega[2], omega[3], alpha[2], alpha[3]);
       printf("Circuit 1: \n  CouplerAccleration: %f \n", Ap[1]);
       Ap[2] = fourbar.couplerPointAccel(theta_2[2], theta_2[3],
                      omega_2[2], omega_2[3], alpha_2[2], alpha_2[3]);
       printf("Circuit 2: \n  CouplerAccleration: %f \n", Ap[2]);
    }
}
```

Output

```
Circuit 1:
  CouplerAccleration: complex(-39.835109,-79.731271)
Circuit 2:
  CouplerAccleration: complex(16.618619,-30.354696)
```

CFourbar::couplerPointPos

Synopsis

#include <fourbar.h>

void couplerPointPos(double *theta2*, **double complex** &*p1,* **double complex** &*p2*);

Purpose
Calculate the position of coupler point.

Parameters

theta2 A double number used for the input angle of link.

p1 A double complex number for the first solution of coupler point.

p2 A double complex number for the second solution of coupler point. x.

Return Value
None.

Description
This function calculates the position of the coupler point. *theta2* is the input angle. *p1,p2* are the two solutions of coupler point position, respectively. Each is a complex number indicating the vector of the coupler point.

Example
A fourbar linkage has link lengths $r_1 = 12m, r_2 = 4m, r_3 = 12m, r_4 = 7m, r_p = 5m$, and angles $\beta = 20$, and $\theta_1 = 10$. Given the angle θ_2, calculate the position of the coupler point for each circuit, respectively.

```
see CFourbar::angularPos().
```

CFourbar::couplerPointVel

Synopsis
#include <fourbar.h>
double complex couplerPointVel(double *theta2,* **double** *theta3,* **double** *omega2,* **double** *omega3*);

Purpose
Calculate the velocity of the coupler point.

Parameters

theta2 A double number used for the angle of link 2.

theta3 A double number used for the angle of link 3.

omega2 A double number used for the angular velocity of link 2.

omega3 A double number used for the angular velocity of link 3.

Return Value
This function returns the vector of the coupler point's velocity.

Description
This function calculates the vector of the coupler point's velocity. *theta2* is the angle of link2, *theta3* is the angle of link3. *omega2* is the angular velocity of link2, and *omega3* is the angular velocity of link3. The

vector of coupler point's velocity is returned.

Example

A fourbar linkage has link lengths $r_1 = 12m, r_2 = 4m, r_3 = 12m, r_4 = 7m, r_p = 5m$, and angles $\beta = 20$, and $\theta_1 = 10$. Given the angle θ_2 and the angular velocity ω_2, determine the Grashof type of the fourbar linkage, and calculate the velocity of the coupler point for each circuit, respectively.

```
/*******************************************************************
*  This example calculates the velocity of coupler point.
*******************************************************************/
#include <fourbar.h>

int main()
{
    CFourbar fourbar;
    int fourbartype;
    double r1 = 12, r2 = 4, r3 =12, r4 = 7, theta1 = 10*M_PI/180;
    double rp = 5, beta = 20*M_PI/180;
    double theta_1[1:4], theta_2[1:4];
    double omega[1:4], alpha[1:4], omega_2[1:4], alpha_2[1:4];
    double theta2 = 70*M_PI/180;
    double complex Vp[1:2];

    omega[2]=5;  /*  rad/sec  */
    alpha[2]=-5;  /*  rad/sec*sec  */

    theta_1[1] = theta1;
    theta_1[2] = theta2; // theta2
    theta_2[1] = theta1;
    theta_2[2] = theta2; // theta2

    fourbar.setLinks(r1, r2, r3, r4, theta1);
    fourbar.setCouplerPoint(rp, beta);

    fourbartype = fourbar.grashof(NULL);

    if (fourbartype == FOURBAR_INVALID) {
      exit(1);
    }

    omega_2[2] = omega[2];
    alpha_2[2] = alpha[2];

    /*find alpha3, alpha4*/
    fourbar.angularPos(theta_1, theta_2, FOURBAR_LINK2);
    if((fourbartype == FOURBAR_ROCKERCRANK)||(fourbartype == FOURBAR_ROCKERROCKER)) {
       fourbar.angularVel(theta_1, omega, FOURBAR_LINK2);
       fourbar.angularVel(theta_2, omega_2, FOURBAR_LINK2);
       Vp[1] = fourbar.couplerPointVel(theta_1[2], theta_1[3], omega[2], omega[3]);
       printf("Circuit 1: \n  CouplerVelocity: %f \n", Vp[1]);
       Vp[2] = fourbar.couplerPointVel(theta_2[2], theta_2[3], omega_2[2], omega_2[3]);
       printf("Circuit 2: \n  CouplerVelocity: %f \n", Vp[2]);

    } else {
       fourbar.angularVel(theta_1, omega, FOURBAR_LINK2);
       fourbar.angularVel(theta_2, omega_2, FOURBAR_LINK2);
       Vp[1] = fourbar.couplerPointVel(theta_1[2], theta_1[3], omega[2], omega[3]);
       printf("Circuit 1: \n  CouplerVelocity: %f \n", Vp[1]);
```

```
        Vp[2] = fourbar.couplerPointVel(theta_2[2], theta_2[3], omega_2[2], omega_2[3]);
        printf("Circuit 2: \n  CouplerVelocity: %f \n", Vp[2]);
    }
}
```

Output

```
Circuit 1:
  CouplerVelocity: complex(-16.727763,4.866514)
Circuit 2:
  CouplerVelocity: complex(-19.090173,6.190806)
```

CFourbar::displayPosition

Synopsis
#include <fourbar.h>
int displayPosition(double *theta2*, **double** *theta3*, **double** *theta4*, ... /* [**int** *outputtype* [, [**char** * *filename*]] */);

Syntax
displayPosition(*theta2*, *theta3*, *theta4*)
displayPosition(*theta2*, *theta3*, *theta4*, *outputtype*)
displayPosition(*theta2*, *theta3*, *theta4*, *outputtype*, *filename*)

Purpose
Given θ_2, θ_3, and θ_4, display the current position of the fourbar linkage.

Parameters

theta2 θ_2 angle.

theta3 θ_3 angle.

theta4 θ_4 angle.

outputtype an optional argument for the output type.

filename an optional argument for the output file name.

Return Value
This function returns 0 on success and -1 on failure.

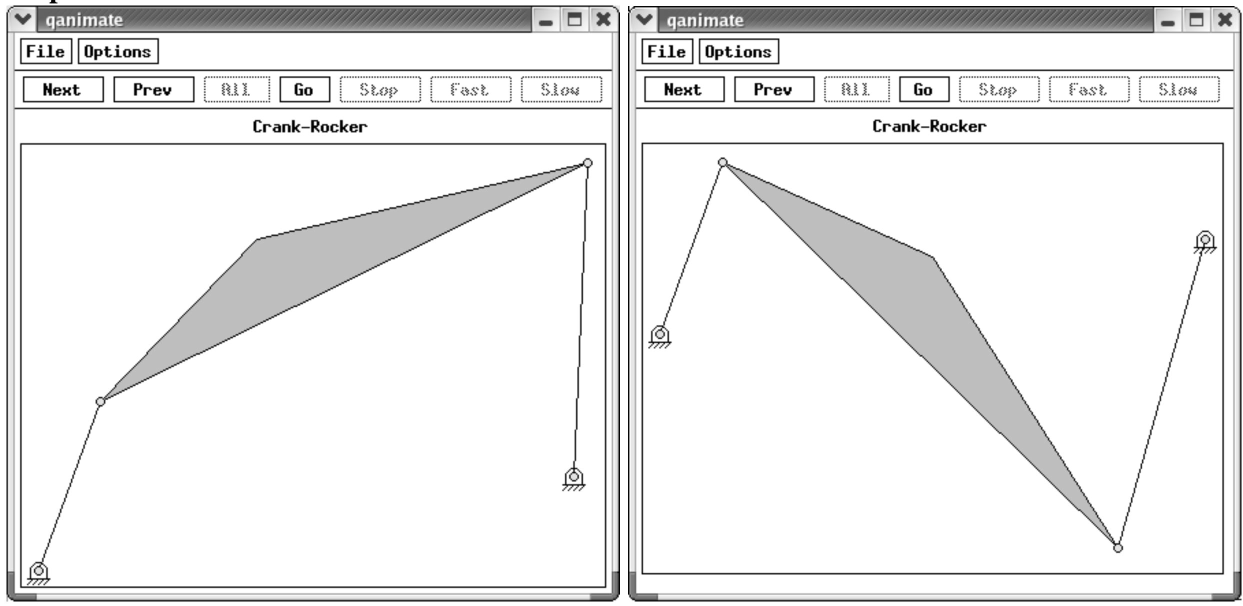
Description
Given θ_2, θ_3, and θ_4, display the current position of the fourbar linkage. `outputtype` is an optional parameter used to specify how the output should be handled. It may be one of the following macros: QANIMATE_OUTPUTTYPE_DISPLAY, QANIMATE_OUTPUTTYPE_FILE, QANIMATE_OUTPUTTYPE_STREAM. QANIMATE_OUTPUTTYPE_DISPLAY outputs the figure to the computer terminal. QANIMATE_OUTPUTTYPE_FILE writes the qanimate data onto a file. QANIMATE_OUTPUTTYPE_STREAM outputs the qanimate data to the standard out stream. `filename` is an optional parameter to specify the output file name.

Example

A fourbar linkage has link lengths $r_1 = 12m, r_2 = 4m, r_3 = 12m, r_4 = 7m$, $\theta_1 = 10$ and coupler point properties $r_p = 5, \beta = 20^o$. Given the angle θ_2, calculate the angular position θ_3 and θ_4 of link3 and link4, display the fourbar linkage in its current position.

```
/**************************************************************
*   This example displays the current position of the fourbar
*   mechanism given theta2, theta3, and theta4.
**************************************************************/
#include <math.h>
#include <fourbar.h>
int main() {
  CFourbar fourbar;
  double r1 = 12, r2 = 4, r3 =12, r4 = 7, theta1 = 10*M_PI/180;
  double rp = 5, beta = 20*M_PI/180;
  double theta_1[1:4], theta_2[1:4];
  double theta2 = 70*M_PI/180;

  theta_1[1] = theta1;
  theta_1[2] = theta2; // theta2
  theta_2[1] = theta1;
  theta_2[2] = theta2; // theta2
  fourbar.setLinks(r1, r2, r3, r4, theta1);
  fourbar.setCouplerPoint(rp, beta);
  fourbar.angularPos(theta_1, theta_2, FOURBAR_LINK2);

  fourbar.displayPosition(theta_1[2], theta_1[3], theta_1[4]);
  fourbar.displayPosition(theta_2[2], theta_2[3], theta_2[4]);

  return 0;
}
```

Output



CFourbar::displayPositions

Synopsis
#include <fourbar.h>
int displayPositions(double *theta2[:]*, **double** *theta3[:]*, **double** *theta4[:]*, ... /* [**int** *outputtype* [, [**char *** *filename*]] */);

Syntax
displayPositions(*theta2*, *theta3*, *theta4*)
displayPositions(*theta2*, *theta3*, *theta4*, *outputtype*)
displayPositions(*theta2*, *theta3*, *theta4*, *outputtype*, *filename*)

Purpose
Given θ_2s, θ_3s, and θ_4s, display the respective positions of the fourbar linkage.

Parameters

theta2 θ_2 angles.

theta3 θ_3 angles.

theta4 θ_4 angles.

outputtype an optional argument for the output type.

filename an optional argument for the output file name.

Return Value
This function returns 0 on success and -1 on failure.

Description
Given θ_2s, θ_3s, and θ_4s, display the relative positions of the fourbar linkage. `outputtype` is an optional parameter used to specify how the output should be handled. It may be one of the following macros: QANIMATE_OUTPUTTYPE_DISPLAY, QANIMATE_OUTPUTTYPE_FILE, QANIMATE_OUTPUTTYPE_STREAM. QANIMATE_OUTPUTTYPE_DISPLAY outputs the figure to the computer terminal. QANIMATE_OUTPUTTYPE_FILE writes the qanimate data onto a file. QANIMATE_OUTPUTTYPE_STREAM outputs the qanimate data to the standard out stream. `filename` is an optional parameter to specify the output file name.

Example
see **CFourbar::synthesis**().

CFourbar::forceTorque

Synopsis
#include <fourbar.h>
void forceTorque(double *theta[1:4],* **double** *omega[1:4],* **double** *alpha[1:4],* **double** *tl*, **array double** *x[9]*);

Purpose
Calculate the joint force and output torque at a given point.

Parameters

theta A double array used for angles of links.

omega A double array used for the angular velocities of links.

alpha A double array used for the angular accelerations of links.

tl A double number used for load torque.

x A double array used for forces in four joints and output torque on link 2 as represented
$$\mathbf{x} = (F_{12x}, F_{12y}, F_{23x}, F_{23y}, F_{34x}, F_{34y}, F_{14x}, F_{14y}, T_s)^T$$

Return Value
None.

Description
This function calculates the joint force and ouput torque at a given load torque. *theta* is a one-dimensional array of size 4 for the angles of links. *omega* is a one-dimensional array of size 4 for the angular velocities of links. *alpha* is a one-dimensional array of size 4 for the angular acclerations of links. *tl* is the load torque. x contains the force and output torque.

Example
A fourbar linkage has link lengths $r_1 = 12in, r_2 = 4in, r_3 = 12in, r_4 = 7in, r_p = 5$, and angles $\beta = 20$, and $\theta_1 = 10$. Given the angle θ_2, angular velocity ω_2, and angular acceleration α_2, calculate the required torque applied to the input link 2 in order to achieve the desired motion. Also calculate the joint forces exerted on the ground from links 1 and 4.

```
/*************************************************************
* This example calcuates the joint force and output torque at
* a given point.
*************************************************************/
#include <math.h>
#include <fourbar.h>

int main() {
    CFourbar fourbar;
    double r1 = 12/12.0, r2 = 4/12.0, r3 =12/12.0, r4 = 7/12.0, theta1 = 0;
    double rp = 5/12.0, beta = 20*M_PI/180;
    double theta_1[1:4], theta_2[1:4], omega[1:4], alpha[1:4];
    array double X[9];
    double g=32.2;
    double rg2 = 2/12.0, rg3 = 6/12.0, rg4 = 3.5/12.0;
    double delta2 = 0.0, delta3 = 0, delta4 = 0.0;
    double m2 = 0.8/g, m3 = 2.4/g, m4 = 1.4/g;
    double ig2 = 0.012/12, ig3 = 0.119/12, ig4 = 0.038/12, tl=0;

    /* initialization of link parameters and
    inertia properties */

    theta_1[1] = 0; theta_1[2]=150*M_PI/180;
    theta_2[1] = 0; theta_2[2]=150*M_PI/180;
    omega[2] = 5;  alpha[2] = -5;
    fourbar.uscUnit(true);
```

```
    fourbar.setLinks(r1, r2, r3, r4, theta1);
    fourbar.setCouplerPoint(rp, beta);
    fourbar.setGravityCenter(rg2, rg3, rg4, delta2, delta3, delta4);
    fourbar.setInertia(ig2, ig3, ig4);
    fourbar.setMass(m2, m3, m4);

    // find theta3, theta4
    fourbar.angularPos(theta_1, theta_2, FOURBAR_LINK2);
    // find omega3, omega4
    fourbar.angularVel(theta_1, omega, FOURBAR_LINK2);
    // find alpha3, alpha4
    fourbar.angularAccel(theta_1, omega, alpha, FOURBAR_LINK2);
    // find forces, torque
    fourbar.forceTorque(theta_1, omega, alpha, tl, X);
    printf("first solution X = %.4f \n", X);

    // find omega3, omega4
    fourbar.angularVel(theta_2, omega, FOURBAR_LINK2);
    // find alpha3, alpha4
    fourbar.angularAccel(theta_2, omega, alpha, FOURBAR_LINK2);
    // find forces, torque
    fourbar.forceTorque(theta_2, omega, alpha, tl, X);
    printf("second solution X = %.4f \n", X);
}
```

Output

```
first solution X = 1.7993 2.3553 1.6993 1.5895 1.1643 -0.7428 -1.0273 2.1624 -0.8659

second solution X = -0.6161 2.4778 -0.7161 1.7120 -1.1555 -0.4193 1.2368 1.7218 -0.4987
```

CFourbar::forceTorques

Synopsis
#include <fourbar.h>
void forceTorques(int *branchnum,* **double** *tl,***array double** *time[:],* **array double** *f12x[:],* **array** double *f12y[:],* **array double** *f23x[:],* **array double** *f23y[:],* **array double** *f34x[:],* **array double** *f34y[:],* **array double** *f14x[:],* **array double** *f14y[:],* **array double** *ts[:]*);

Purpose
Calculate the joint force and output torque in the valid range of motion.

Parameters

branchnum An integer number used to indicate the branch to be calculated.

tl A double number for the load torque.

time A double array to record time.

f12x, f12y, f23x, f23y, f34x, f34y, f14x, f14y Double arrays for forces.

ts A double array for input torque.

Return Value
None.

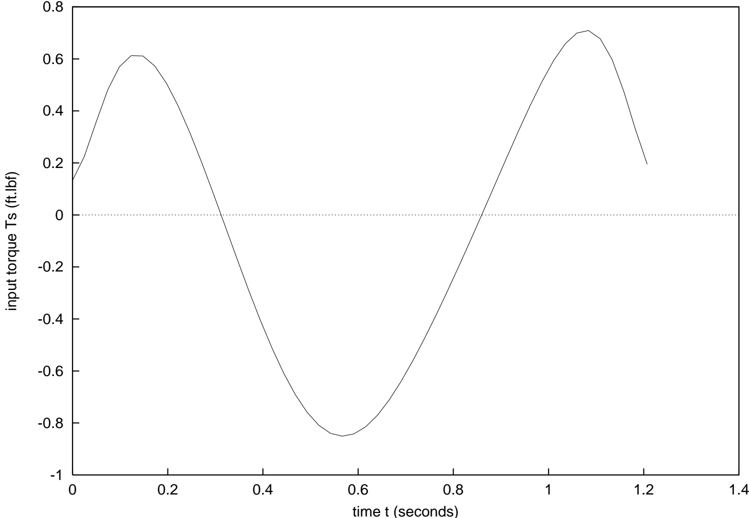
Description
This function calculates the joint force and output torque in the valid range of motion. *branchnum* is the branch which will be plotted. *tl* is the load torque. *time* is an array to record time. *f12x, f12y, f23x, f23y, f34x, f34y, f14x, f14y* are arrays for forces. *ts* is a double array for input torque.

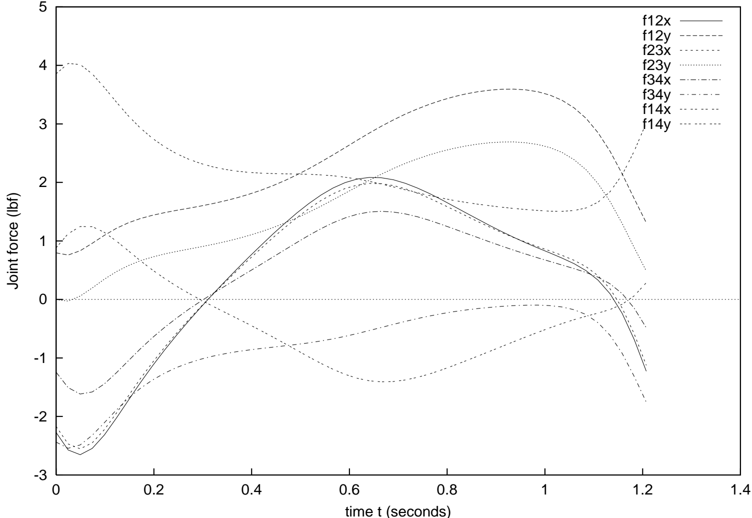
Example
A fourbar linkage has link lengths $r_1 = 12in, r_2 = 4in, r_3 = 12in, r_4 = 7in, r_p = 5in$, and angles $\beta = 20$, and $\theta_1 = 10$. Given the angle θ_2 and constant angular velocity ω_2, using a loop to calculate the required torques applied to the input link 2 in order to achieve the constant angular velocity for link 2. Also calculate the joint forces exerted on the ground from links 1 and 4.

```
/****************************************************************
* This example calculates the joint force and output torque
* in the valid range of motion.
****************************************************************/
#include <math.h>
#include  <fourbar.h>

int main()
{
    CFourbar fourbar;
    double r1 = 12/12.0, r2 = 4/12.0, r3 =12/12.0, r4 = 7/12.0, theta1 = 0;
    double rp = 5/12.0, beta = 20*M_PI/180;
    double g=32.2;
    double rg2 = 2/12.0, rg3 = 6/12.0, rg4 = 3.5/12.0;
    double delta2 = 0.0, delta3 = 0, delta4 = 0.0;
    double m2 = 0.8/g, m3 = 2.4/g, m4 = 1.4/g;
    double ig2 = 0.012/12, ig3 = 0.119/12, ig4 = 0.038/12, tl=0;
    int numpoint = 50;
    double  omega2 = 5;  /* constant omega2 */
    array double  time[numpoint], ts[numpoint];
    array double  f12x[numpoint], f12y[numpoint];
    array double  f23x[numpoint], f23y[numpoint];
    array double  f34x[numpoint], f34y[numpoint];
    array double  f14x[numpoint], f14y[numpoint];
    int branchnum = 1;
    int i;
    class CPlot pl;

    /* initialization of link parameters and
    inertia properties */
    fourbar.uscUnit(true);
    fourbar.setLinks(r1, r2, r3, r4, theta1);
    fourbar.setCouplerPoint(rp, beta);
    fourbar.setGravityCenter(rg2, rg3, rg4, delta2, delta3, delta4);
    fourbar.setInertia(ig2, ig3, ig4);
    fourbar.setMass(m2, m3, m4);
    fourbar.setNumPoints(numpoint);
    fourbar.setAngularVel(omega2);


    fourbar.forceTorques(branchnum, tl, time, f12x,
              f12y, f23x, f23y, f34x, f34y, f14x, f14y, ts);  // calculate the forces
                                                              // and torque
```

```
    plotxy(time, ts, NULL, "time t (seconds)",  "input torque Ts (ft.lbf)", &pl);
    pl.border(PLOT_BORDER_ALL, PLOT_ON);
    pl.plotting();
    plotxy(time, f12x, NULL, "time t (seconds)",  "Joint force (lbf)", &pl);
    pl.data2D(time, f12y);
    pl.data2D(time, f23x);
    pl.data2D(time, f23y);
    pl.data2D(time, f34x);
    pl.data2D(time, f34y);
    pl.data2D(time, f14x);
    pl.data2D(time, f14y);
    pl.legend("f12x",0);
    pl.legend("f12y",1);
    pl.legend("f23x",2);
    pl.legend("f23y",3);
    pl.legend("f34x",4);
    pl.legend("f34y",5);
    pl.legend("f14x",6);
    pl.legend("f14y",7);
    pl.border(PLOT_BORDER_ALL, PLOT_ON);
    pl.plotting();

}
```

Output





CFourbar::getAngle

Synopsis
#include <fourbar.h>
int getAngle(**double** *theta[1:]* **int** *theta_id*);

Purpose
Given angles of two moving links, calculate the angle of the remaining moving link.

Parameters

theta A double array with dimension size of 4 for angular positions.

theta_id An integer number indicating the link number of the unknown position angle.

Return Value
This function returns 0 on success and -1 on failure.

Description
Given the angular positions of two links of a fourbar linkage, this function computes the angular position of the remaining moving link. *theta* is a one-dimensional array of size 4 which stores the angular positions of the links. *theta_id* is an integer number indicating the link number of the known angular position.

Example
see **CFourbar::synthesis**().

CFourbar::getJointLimits

Synopsis
#include <fourbar.h>

int getJointLimits(**double** *inputmin[2],* **double** *inputmax[2]*, **double** *ouputmin[2],* **double** *outputmax[2]*);

Purpose
Calculate the input and output joint limits of a fourbar linkage.

Parameters

inputmin A double array used for the minimum input angle.

inputmax A double array used for the maximum input angle.

outputmin A double array used for the minimum output angle.

outputmax A double array used for the maximum output angle.

Return Value
This function returns the type of linkage.
The return value are: **FOURBAR_INVALID**, **FOURBAR_CRANKCRANK**, **FOURBAR_ROCKERCRANK**, **FOURBAR_ROCKERROCKER**, **FOURBAR_INWARDINWARD**, **FOURBAR_INWARDOUTWARD**, **FOURBAR_OUTWARDINWARD**, **FOURBAR_OUTWARDOUTWARD**

Description
This function calculates fourbar linkage input and output joint limits. *inputmin, inputmax, outputmin outputmax* are arrays for the limit of the fourbar linkage. The function returns the type of linkage.

Example
For a fourbar linkage with link lengths $r_1 = 12m, r_2 = 4m, r_3 = 12m, r_4 = 7m$, and angle $\theta_1 = 0$, determine the input limit and output limit for each circuit.

```
/*****************************************************************
*  This example calculates the input and output joint limits
*  of fourbar linkage.
*****************************************************************/
#include <math.h>
#include <fourbar.h>

int main() {
    CFourbar fourbar;
    double r1 = 12, r2 = 4, r3 =12, r4 = 7, theta1 = 0;
    class CPlot pl;
    int fourbartype;

    double inputlimitmin[2], inputlimitmax[2],
           outputlimitmin[2], outputlimitmax[2];

    fourbar.setLinks(r1, r2, r3, r4, theta1);

    fourbartype = fourbar.getJointLimits(inputlimitmin, inputlimitmax,
                             outputlimitmin, outputlimitmax);
    printf("input Range:\n");
    printf("    Circuit:  %8s %8s\n","1","2");
    printf("              %8s %8s\n","(deg)", "(deg)");
    printf("    Lower limit: %7.2f %8.2f\n",
            inputlimitmin[0]*180./M_PI, inputlimitmin[1]*180./M_PI);
    printf("    Upper limit: %7.2f %8.2f\n",
```

```
  inputlimitmax[0]*180./M_PI, inputlimitmax[1]*180./M_PI);

    printf("output Range:\n");
    printf("    Circuit:  %8s %8s\n","1","2");
    printf("              %8s %8s\n","(deg)", "(deg)");
    printf("    Lower limit: %7.2f %8.2f\n",
            outputlimitmin[0]*180./M_PI, outputlimitmin[1]*180./M_PI);
    printf("    Upper limit: %7.2f %8.2f\n",
  outputlimitmax[0]*180./M_PI, outputlimitmax[1]*180./M_PI);
}
```

Output

```
input Range:
    Circuit:         1        2
                 (deg)    (deg)
    Lower limit:    0.00     0.00
    Upper limit:  360.00   360.00
output Range:
    Circuit:         1        2
                 (deg)    (deg)
    Lower limit:   67.98  -140.16
    Upper limit:  140.16   -67.98
```

CFourbar::grashof

Synopsis

#include <fourbar.h>
int grashof(string_t &*name*);

Purpose

Determine the Grashof type of the fourbar linkage.

Parameters

name A string indicating the Grashof type of linkage.

Return Value

This function returns the Grashof type of the linkage.
The return values are: **FOURBAR_INVALID**, **FOURBAR_CRANKROCKER**, **FOURBAR_CRANKCRANK**, **FOURBAR_ROCKERCRANK**, **FOURBAR_ROCKERROCKER**, **FOURBAR_INWARDINWARD**, **FOURBAR_INWARDOUTWARD**, **FOURBAR_OUTWARDINWARD**, **FOURBAR_OUTWARDOUTWARD**

Description

This function determines the Grashof type of the linkage. *name* is a string indicating the Grashof type. The fuction returns the type of linkage.

Example

For a fourbar linkage with link lengths $r_1 = 12m, r_2 = 4m, r_3 = 12m, r_4 = 7m$, and angle $\theta_1 = 10$, determine the Grashof type of the linkage.

```
/********************************************************************
```

```
*  This example calculate joint limits and decide Grashof type of
*  linkage.
*****************************************************************/
#include <stdio.h>
#include <fourbar.h>

int main() {
    double theta1, theta2;
    double gamma1, gamma2;
    int fourbartype;                            /* the type of linkage */
    string_t GrashofName;
    CFourbar fourbar;

    /* default specification of the four-bar linkage */
    double r1 = 12, r2 = 4, r3 = 10, r4= 7;//cranker-rocker
    theta1 = 10*M_PI/180; theta2=45*M_PI/180;

    fourbar.setLinks(r1, r2, r3, r4, theta1);

    /* Grashof Analysis */
    fourbartype = fourbar.grashof(GrashofName);
    printf("linkage type is %s \n", GrashofName);
}
```

Output

```
linkage type is Crank-Rocker
```

CFourbar::plotAngularAccels

Synopsis
#include <fourbar.h>
void plotAngularAccels(CPlot **plot*, **int** *branchnum*);

Purpose
Plot angular accelerations α_3 and α_4 with respect to time.

Parameters

&plot A pointer to a CPlot class variable for formatting the plot of the branch to be drawn.

branchnum An integer used to indicate the branch to be drawn.

Return Value
None.

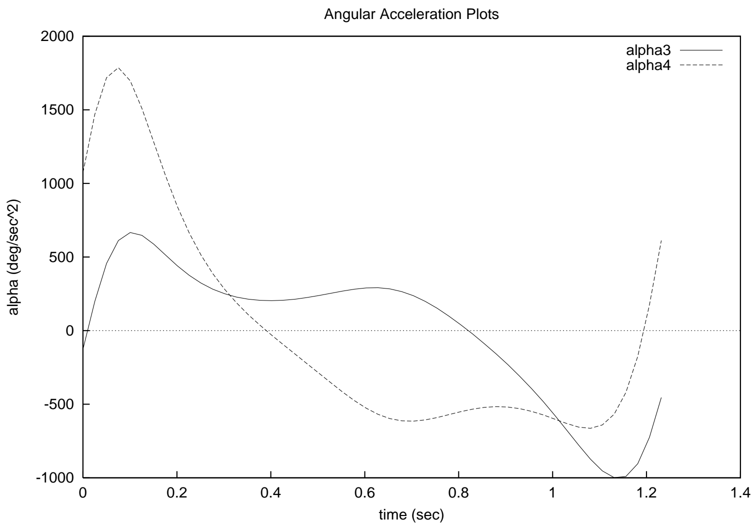
Description
This function plots angular accelerations α_3 and α_4 with respect to time. *&plot* is a pointer to a CPlot class variable for formatting the plot of the branch to be drawn. *branchnum* is an integer number used to indicate the branch to be drawn.

Example
For a fourbar linkage with link lengths $r_1 = 12m, r_2 = 4m, r_3 = 12m, r_4 = 7m$, angle $\theta_1 = 10^o$, and

constant angular velocity $\omega_2 = 5rad/s$, plot the angular acceleration curves for links 3 and 4.

```
/**********************************************************************
*   This example is for plotting angular accelerations alpha3 and
*   alpha4 with respect to time.
**********************************************************************/
#include <math.h>
#include <stdio.h>
#include <fourbar.h>

int main() {
    double r[1:4], theta1;
    double omega2;
    int numpoints = 50;
    CFourbar fourbar;
    CPlot plot;

    /* default specification of the four-bar linkage */
    r[1] = 12; r[2] = 4; r[3] = 12; r[4] = 7;
    theta1 = 10*M_PI/180;
    omega2 = 5;  /*  rad/sec  */

    fourbar.setLinks(r[1], r[2], r[3], r[4], theta1);
    fourbar.setAngularVel(omega2);
    fourbar.setNumPoints(numpoints);
    fourbar.plotAngularAccels(&plot, 1);
}
```

Output



CFourbar::plotAngularPoss

Synopsis
#include <fourbar.h>
void plotAngularPoss(CPlot **plot*, **int** *branchnum*);

Purpose

Plot angular positions θ_3 and θ_4 with respect to time.

Parameters

&plot A pointer to a CPlot class variable for formatting the plot of the branch to be drawn.

branchnum An integer used to indicate the branch to be drawn.

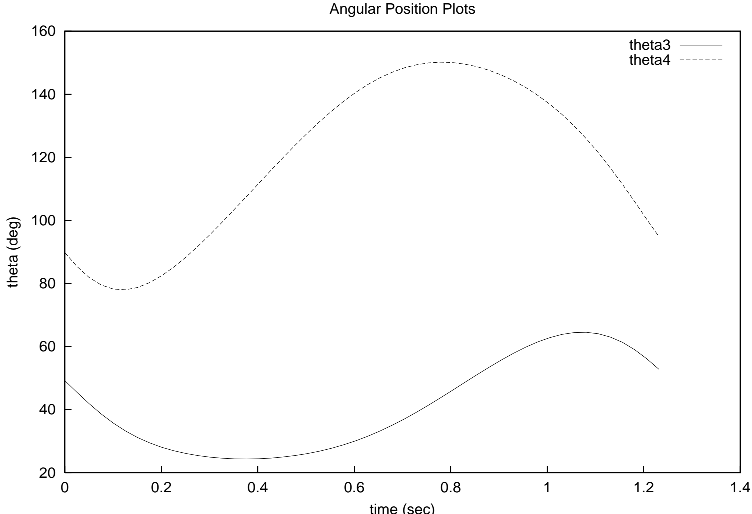
Return Value
None.

Description
This function plots angular positions θ_3 and θ_4 with respect to time. *&plot* is a pointer to a CPlot class variable for formatting the plot of the branch to be drawn. *branchnum* is an integer number used to indicate the branch to be drawn.

Example
For a fourbar linkage with link lengths $r_1 = 12m, r_2 = 4m, r_3 = 12m, r_4 = 7m$, and angle $\theta_1 = 10^o$, plot the angular position curves for links 3 and 4.

```
/****************************************************************
*   This example is for plotting angular positions theta3 and
*   theta4 with respect to time.
****************************************************************/
#include <math.h>
#include <fourbar.h>
int main() {
    CFourbar fourbar;
    double r1 = 12, r2 = 4, r3 =12, r4 = 7, theta1 = 10*M_PI/180;
    double omega2 = 5;     // rad/sec
    int numpoints = 50;
    CPlot plot;

    fourbar.setLinks(r1, r2, r3, r4, theta1);
    fourbar.setAngularVel(omega2);
    fourbar.setNumPoints(numpoints);
    fourbar.plotAngularPoss(&plot, 1);
}
```

Output



CFourbar::plotAngularVels

Synopsis
#include <fourbar.h>
void plotAngularVels(CPlot **plot*, **int** *branchnum*);

Purpose
Plot angular velocities ω_3 and ω_4 with respect to time.

Parameters

&plot A pointer to a CPlot class variable for formatting the plot of the branch to be drawn.

branchnum An integer used to indicate the branch to be drawn.

Return Value
None.

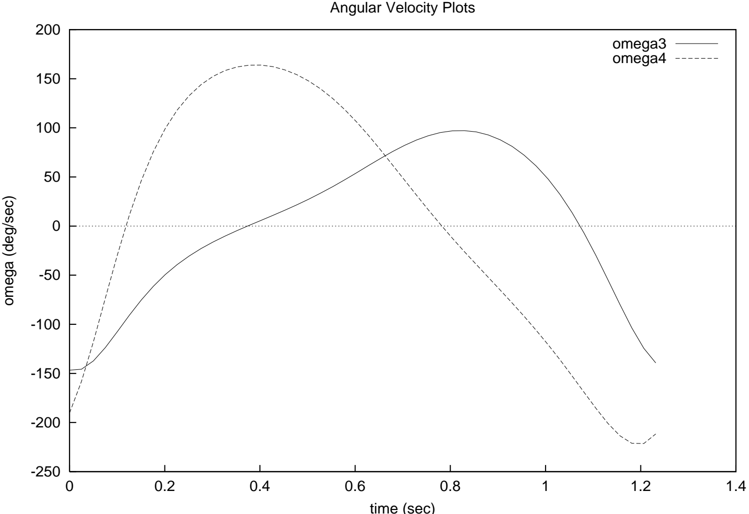
Description
This function plots angular velocities ω_3 and ω_4 with respect to time. *&plot* is a pointer to a CPlot class variable for formatting the plot of the branch to be drawn. *branchnum* is an integer number used to indicate the branch to be drawn.

Example
For a fourbar linkage with link lengths $r_1 = 12m, r_2 = 4m, r_3 = 12m, r_4 = 7m$, angle $\theta_1 = 10^o$, and constant angular velocity $\omega_2 = 5rad/s$, plot the angular velocity curves for links 3 and 4.

```
/**********************************************************************
*   This example is for plotting angular velocities omega3 and omega4
*   with respect to time.
**********************************************************************/
#include <math.h>
#include <stdio.h>
```

```
#include <fourbar.h>

int main() {
    double r[1:4], theta1;
    double omega2;
    int numpoints = 50;
    CFourbar fourbar;
    CPlot plot;

    /* default specification of the four-bar linkage */
    r[1] = 12; r[2] = 4; r[3] = 12; r[4] = 7;
    theta1 = 10*M_PI/180;
    omega2 = 5;  /*  rad/sec  */

    fourbar.setLinks(r[1], r[2], r[3], r[4], theta1);
    fourbar.setAngularVel(omega2);
    fourbar.setNumPoints(numpoints);
    fourbar.plotAngularVels(&plot, 1);
}
```

Output



CFourbar::plotCouplerCurve

Synopsis

#include <fourbar.h>
void plotCouplerCurve(CPlot **plot*, **int** *branchnum*);

Purpose

Plot the coupler curve.

Parameters

&plot A pointer to a CPlot class variable for formatting the plot of the branch to be drawn.

branchnum An integer used to indicate the branch to be drawn.

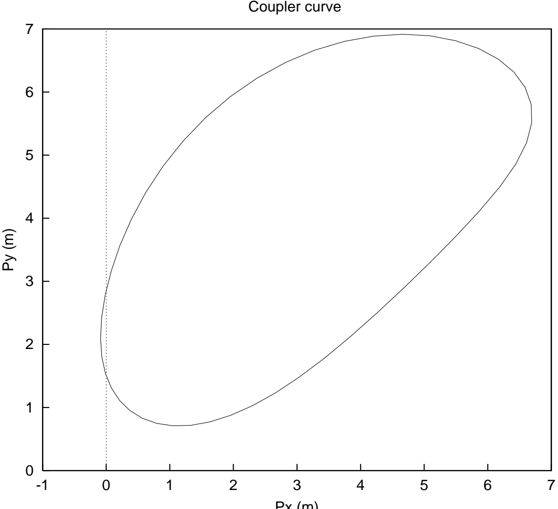
Return Value
None.

Description
This function plots the coupler curve. *&plot* is a pointer to a CPlot class variable for formatting the plot of the branch to be drawn. *branchnum* is an integer number used to indicate the branch to be drawn.

Example
For a fourbar linkage with link lengths $r_1 = 12m, r_2 = 4m, r_3 = 10m, r_4 = 7m, r_p = 5$, and angles $\theta_1 = 0$, and $\beta = 20$, plot the curve of the coupler point's position.

```
/**********************************************************
* This example plot the coupler curve.
**********************************************************/
#include <math.h>
#include <fourbar.h>

int main() {
    CFourbar fourbar;
    double r1 = 12, r2 = 4, r3 = 10, r4= 7;//cranker-rocker
    double theta1 = 0;
    double rp = 5, beta = 20*M_PI/180;
    class CPlot plot;

    fourbar.setLinks(r1, r2, r3, r4, theta1);
    fourbar.setCouplerPoint(rp, beta);
    fourbar.setNumPoints(50);
    fourbar.plotCouplerCurve(&plot, 1);  //display a coupler curve

}
```

Output



CFourbar::plotForceTorques

Synopsis
#include <fourbar.h>
void plotForceTorques(CPlot **plot*, **int** *branchnum*, **double** *tl*);

Purpose
Plot output torque curve.

Parameters

&plot A pointer to a class used for formatting plot of the branch to be drawn.

branchnum An integer number used for indicating the branch that will be plotted.

tl A double number for the load torque.

Return Value
None.

Description
This function plots the output torque curve. *&plot* is a pointer to a CPlot class for formatting the plot of the branch to be plotted. *branchnum* is an integer for the branch that will be plotted. *tl* is a double number for the load torque. The first point and the last point are removed when plotting for all linkages except types crank-rocker and crank-crank because of sigularities.

Example
A fourbar linkage has link lengths $r_1 = 12in, r_2 = 4in, r_3 = 10in, r_4 = 7in$, and angle $\theta_1 = 10^o$. Given constant angular velocity $\omega_2 = 5rad/sec$, load torque $t_l = 0$ and inertia properties of the fourbar, plot the joint forces and output torque versus time t.
See Program 12.
Output
See Figure 2.11.

CFourbar::plotTransAngles

Synopsis
#include <fourbar.h>
void plotTransAngles(CPlot **plot*, **int** *branchnum*);

Purpose
Plot the transmission angle γ with respect to θ_2.

Parameters

&plot A pointer to a CPlot class variable for formatting the plot of the branch to be drawn.

branchnum An integer used to indicate the branch to be drawn.

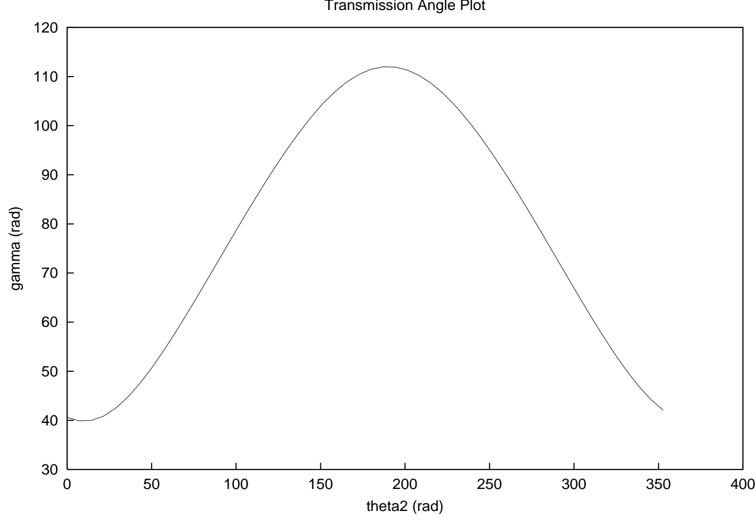
Return Value
None.

Description
This function plots the transmission angle curve as a function of θ_2. *&plot* is a pointer to a CPlot class variable for formatting the plot of the branch to be drawn. *branchnum* is an integer number used to indicate the branch to be drawn.

Example
For a fourbar linkage with link lengths $r_1 = 12m, r_2 = 4m, r_3 = 12m, r_4 = 7m$, and angle $\theta_1 = 10^o$, plot the transmission angle curve for the first branch.

```
/****************************************************************
*   This example is for plotting the transmission angle vs
*   theta2.
****************************************************************/
#include <math.h>
#include <fourbar.h>

int main() {
    CFourbar fourbar;
    double r1 = 12, r2 = 4, r3 = 12, r4 = 7, theta1 = 10*M_PI/180;
    int numpoints = 50;
    CPlot plot;

    fourbar.setLinks(r1, r2, r3, r4, theta1);
    fourbar.setNumPoints(numpoints);
    fourbar.plotTransAngles(&plot, 1);
}
```

Output



CFourbar::printJointLimits

Synopsis
#include <fourbar.h>
void printJointLimits();

Purpose
Prints the range of motion for the input and output joints.

Parameters None.

Return Value
None.

Description
This function prints the range of motion of the input and output links.
Example
For a fourbar linkage with link lengths $r_1 = 12m, r_2 = 4m, r_3 = 12m, r_4 = 7m$, and angle $\theta_1 = 10$, print out the joint limits for the input and output joints.

```
/*******************************************************************
*  This example calculate joint limits and decide Grashof type of
*  linkage.
******************************************************************/
#include <stdio.h>
#include <fourbar.h>

int main() {
    double theta1, theta2;
    double gamma1, gamma2;
    int fourbartype;                            /* the type of linkage */
    string_t GrashofName;
    CFourbar fourbar;

    /* default specification of the four-bar linkage */
    double r1 = 12, r2 = 4, r3 = 10, r4= 7;//cranker-rocker
    theta1 = 10*M_PI/180; theta2=45*M_PI/180;

    fourbar.setLinks(r1, r2, r3, r4, theta1);

    /* Grashof Analysis */
    fourbar.printJointLimits();
}
```

Output

```
Input Characteristics: Input 360 degree rotation
Output Range:
    Circuit:         1        2
                   (deg)    (deg)
    Lower limit:   98.98  -149.15
    Upper limit:  169.15   -78.98
```

CFourbar::setAngularVel

Synopsis
#include <fourbar.h>
void setAngularVel(double *omega2***);**

Purpose
Set the constant angular velocity of link 2.

Parameters

omega2 A double number used for the constant input angular velocity of link 2.

Return Value
None.

Description
This function sets the constant angular velocity of link 2. It is used in conjunction with member functions **plotAngularPoss**(), **plotAngularVels**(), **plotAngularAccels**(), **plotForceTorques**(), and **forceTorques**(). The relationships between θ_2, ω_2, and α_2 are as follows,

$$\begin{aligned} \theta_2 &= \omega_0 t + \theta_{2,min} \\ \omega_2 &= \omega_0 \\ \alpha_2 &= 0 \end{aligned}$$

where $\theta_{2,min}$ is the minimum angular position of the input link, and ω_0 is a constant.

Example
see **CFourbar::plotForceTorques**().

CFourbar::setCouplerPoint

Synopsis
#include <fourbar.h>
void setCouplerPoint(**double** *rp, beta*, ... /* [**int** trace] */);

Syntax
setCouplerPoint(*rp*, *beta*)
setCouplerPoint(*rp*, *beta*, *trace*)

Purpose
Set parameters for the coupler point.

Parameters

rp A double number used for rp.

beta A double number for beta.

it trace An optional parameter of **int** type specifying either macro TRACE_OFF or TRACE_ON to indicate whether the coupler curve should be traced during animation.

Return Value
None.

Description
This function sets the parameters of the coupler point.

Example
see **CFourbar::plotCouplerCurve**(), **CFourbar::animation**().

CFourbar::setGravityCenter

Synopsis
#include <fourbar.h>
void setGravityCenter(double *rg2,* **double** *rg3,* **double** *rg4,* **double** *delta2,* **double** *delta3,* **double** *delta4*);

Purpose
Set parameters for the mass centers of links.

Parameters

rg2 A double number used for the distance from joint A_0 to the center of gravity of link 2.

rg3 A double number used for the distance from joint A to the center of gravity of link 3.

rg4 A double number used for the distance from joint B_0 to the center of gravity of link 4.

delta2 A double used for the angle between vector rg_2 and link 2.

delta3 A double used for the angle between vector rg_3 and link 3.

delta4 A double used for the angle between vector rg_4 and link 4.

Return Value
None.

Description
This function sets parameters for the mass centers of links.

Example
see **CFourbar::plotForceTorques**().

CFourbar::setInertia

Synopsis
#include <fourbar.h>
void setInertia(double *ig2,* **double** *ig3,* **double** *ig4*);

Purpose
Set inertia parameters of the links.

Parameters

ig2 A double number used for the inertia of link 2.

ig3 A double number used for the inertia of link 3.

ig4 A double number used for the inertia of link 4.

Return Value
None.

Description
This function sets inertia parameters of the links.

Example
see **CFourbar**::**plotForceTorques**().

CFourbar::setLinks

Synopsis
#include <fourbar.h>
int setLinks(double *r1,* **double** *r2,* **double** *r3,* **double** *r4,* **double** *theta1*);

Purpose
Set the lengths of links.

Parameters

r1,r2,r3,r4 A double number used for the length of links.

theta1 A double number for the angle between link1 and horizontal.

Return Value
This function returns 0 on success and -1 on failure.

Description
This function sets the lengths of links and determines if the links can construct a fourbar mechanism. If not, return -1.

Example
see **CFourbar**::**plotCouplerCurve**().

CFourbar::setMass

Synopsis
#include <fourbar.h>
void setMass(double *m2,* **double** *m3,* **double** *m4*);

Purpose
Set masses of the links.

Parameters

m2,m3,m4 double numbers used for the masses of links.

Return Value
None.

Description
This function sets masses of links.

Example
see **CFourbar**::**plotForceTorques**().

CFourbar::**setNumPoints**

Synopsis
#include <fourbar.h>
void setNumPoints(**int** *numpoints*);

Purpose
Set the number of points for animation and plotting coupler curves.

Parameters

numpoints An integer number used for the number of points.

Return Value
None.

Description
This function sets the number of points for animation and plotting coupler curves.

Example
see **CFourbar**::**animation**().

CFourbar::**synthesis**

Synopsis
#include <fourbar.h>
int synthesis(**double** *r[1:4],* **double** *phi[:],* **double** *psi[:]*);

Purpose
Fourbar linkage position synthesis.

Parameters

r A double array used for the length of links.

phi A double array used for the input angles.

psi A double array used for the output angles.

Return Value
This function returns 0 on success and -1 on failure.

Description

This function synthesizes fourbar linkage position. *r* is an array for the length of the links. *phi* is an array for the input angles. *psi* is an array for the output angles.

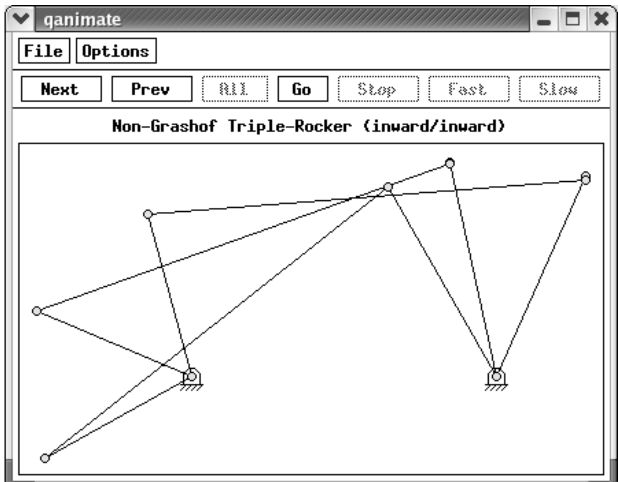
Example

Given input angles $\psi_1 = 66.27, \psi_2 = 102.42, \psi_3 = 119.67$ and output angles $\phi_1 = 105, \phi_2 = 157, \phi_3 = 209$, calculate the length of the links. Then display the fourbar at these three positions.

```
#include <stdio.h>
#include <fourbar.h>

int main()
{
    double r[1:4];
    double psi[1:3], phi[1:3];
    double theta[1:4], theta3[1:3];
    CFourbar fourbar;

    /* specify input/output relation for a four-bar linkage */
    r[1] = 1;
    psi[1]=66.27*M_PI/180; psi[2]=102.42*M_PI/180; psi[3]=119.67*M_PI/180;
    phi[1]=105*M_PI/180; phi[2]=157*M_PI/180; phi[3]=209*M_PI/180;
    fourbar.synthesis(r,phi,psi);

    /* display link lengths */
    printf("r2 = %.3f, r3 = %.3f, r4 = %.3f\n", r[2], r[3], r[4]);

    /* obtain theta3 in three positions and display these positions */
    theta[1] = 0;
    fourbar.setLinks(r[1], r[2], r[3], r[4], theta[1]);
    theta[2]=phi[1];  theta[4] = psi[1];
    fourbar.getAngle(theta, FOURBAR_LINK3);
    theta3[1] = theta[3];
    theta[2] = phi[2]; theta[4] = psi[2];
    fourbar.getAngle(theta, FOURBAR_LINK3); theta3[2] = theta[3];
    theta[2] = phi[3]; theta[4] = psi[3];
    fourbar.getAngle(theta, FOURBAR_LINK3); theta3[3] = theta[3];
    fourbar.displayPositions(phi, theta3, psi);

    return 0;
}
```

Output

```
Results:  Interactive Four-Bar Linkage Position Synthesis

      Link 1 =  1.000
      Link 2 =  0.555
      Link 3 =  1.441
      Link 4 =  0.725
```



CFourbar::transAngle

Synopsis
#include <fourbar.h>
void transAngle(double &*gamma1,* **double** &*gamma2,* **double** *theta,* **int** *given_theta*);

Purpose
Given input link position, calculate the transmission angle.

Parameters

gamma1 A double number used for the first solution.

gamma2 A double number used for the second solution.

theta A double number used for a given link position.

given_theta An integer number used as an identifier for a known link.

Return Value
None.

Description
This function calculates the transmission angle, given the position of the input link. *gamm1, gamm2* are used for the two solutions of the transmission angle for each circuit of the linkage, respectively. *theta* is the given input link position. *given_theta* is an idetifier for the known angle theta.

Example
A fourbar linkage has link lengths $r_1 = 5m, r_2 = 1.5m, r_3 = 4m, r_4 = 4.5m$, and an angle $\theta_1 = 10$. Given the angle θ_2, determine the Grashof type of the fourbar linkage and calculate the transmission angle for each circuit, respectively.

```
/************************************************************************
*  This example calculate the transmission angle with given input
*  link position.
************************************************************************/
```

```
#include <math.h>
#include <stdio.h>
#include <fourbar.h>

int main() {
    double r[1:4], theta1, theta2;
    double gamma1, gamma2;
    int fourbartype;
    CFourbar fourbar;

    /* default specification of the four-bar linkage */
    r[1] = 5; r[2] = 1.5; r[3] = 4; r[4] = 4.5;
    theta1 = 10*M_PI/180; theta2=45*M_PI/180;

    printf("Results: Interactive Four-Bar Linkage Transmission Angle Analysis\n\n");

    fourbar.setLinks(r[1], r[2], r[3], r[4], theta1);

       /* Grashof Analysis */
    fourbartype = fourbar.grashof(NULL);
    if (fourbartype == FOURBAR_INVALID) exit(1);
    fourbar.transAngle(gamma1, gamma2, theta2, FOURBAR_LINK2);
    printf("\n Circuit 1: Transmission Angle\n\n");
    printf("\tDegrees:\t gamma=%6.3f \n",gamma1*180/M_PI);
    printf("\tRadians:\t gamma=%6.4f \n",gamma1);
    printf("\n Circuit 2: Transmission Angle\n\n");
    printf("\tDegrees:\t gamma=%6.3f \n",gamma2*180/M_PI);
    printf("\tRadians:\t gamma=%6.4f \n",gamma2);
}
```

Output

```
Results: Interactive Four-Bar Linkage Transmission Angle Analysis


 Circuit 1: Transmission Angle

Degrees:  gamma=53.750
Radians:  gamma=0.9381

 Circuit 2: Transmission Angle

Degrees:  gamma=-53.750
Radians:  gamma=-0.9381
```

CFourbar::transAngles

Synopsis

#include <fourbar.h>

void transAngles(int *branchnum*, **double** *theta2[:]*, **double** *gamma[:]*);

Purpose

Calculate transmission angle values for the valid range of motion.

Parameters

branchnum An integer used to indicate the branch of the fourbar.

theta2 An array for θ_2 values.

gamma An array for γ values.

Return Value
This function returns 0 on success and -1 on failure.

Description
This function calculates the values for γ for the valid range of motion. `branchnum` is the branch of the fourbar. `theta2` is an array for θ_2 values. `theta3` are arrays for storing the calculated transmission angle values.

Example
For a fourbar linkage with link lengths $r_1 = 12m, r_2 = 4m, r_3 = 12m, r_4 = 7m$, and angle $\theta_1 = 10^o$, plot the transmission angle curve for the first branch.

```
/**************************************************************
*   This example is for generating transmission angle  values
*   for the valid range of motion.
**************************************************************/
#include <math.h>
#include <fourbar.h>

int main() {
    CFourbar fourbar;
    double r1 = 12, r2 = 4, r3 =12, r4 = 7, theta1 = 10*M_PI/180;
    int i, numpoints = 50;
    double theta2[numpoints], gamma[numpoints];
    CPlot plot;

    fourbar.setLinks(r1, r2, r3, r4, theta1);
    fourbar.transAngles(1, theta2, gamma);
    for(i = 0; i < numpoints; i++) {
       theta2[i] = M_RAD2DEG(theta2[i]);
       gamma[i] = M_RAD2DEG(gamma[i]);
    }
    plot.data2D(theta2, gamma);
    plot.title("Transmission Angle Plot");
    plot.label(PLOT_AXIS_X, "theta2 (deg)");
    plot.label(PLOT_AXIS_Y, "gamma (deg)");
    plot.border(PLOT_BORDER_ALL, PLOT_ON);
    plot.plotting();
}
```

Output
see output for **CFourbar::plotTransAngles**() example.

CFourbar::uscUnit

Synopsis
#include <fourbar.h>

void uscUnit(**bool** *unit*);

Purpose
Specify the use of SI or US Customary units in analysis.

Parameters

unit A boolean argument, where `true` indicates US Customary units are desired and `false` indicates SI units.

Return Value
None.

Description
This function specifies the whether SI or US Customary units are used. If `unit = true`, then US Customary units are used; otherwise, SI units are used. By default, SI units are assumed. This member function shall be called prior any other member function calls.

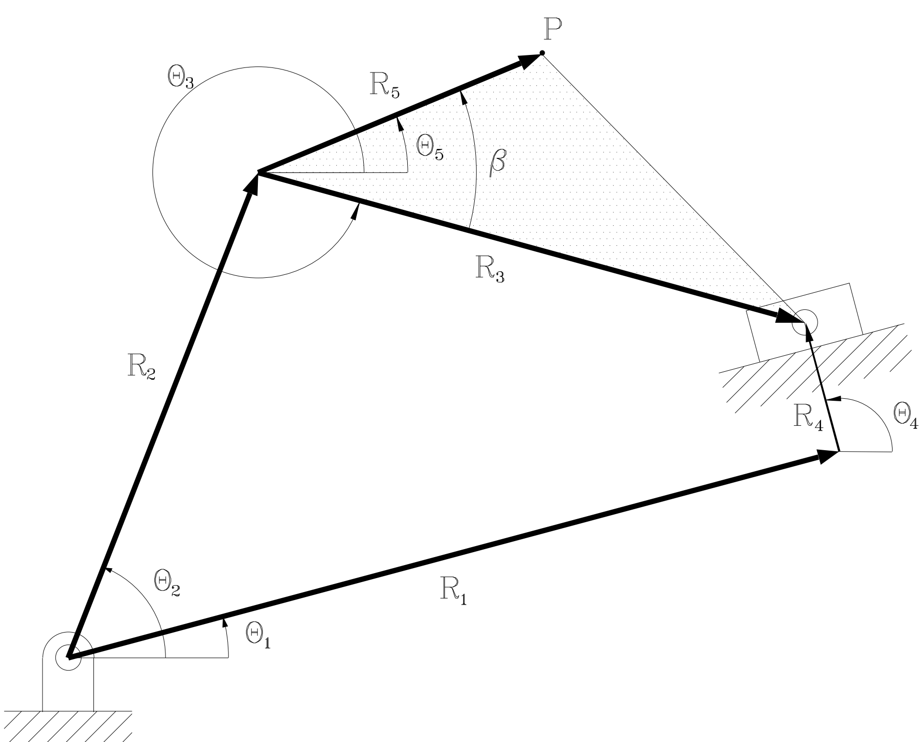
Example
see **CFourbar**::**forceTorque**().

Appendix C

Class CCrankSlider

CCrankSlider

The header file **crankslider.h** includes header file **linkage.h**. The header file **crankslider.h** also contains a declaration of class **CCrankSlider**. The **CCrankSlider** class provides a means to analyze crank-slider within a Ch language environment.



Public Data
None.

Public Member Functions

Functions	Descriptions

angularAccel	Given the angular acceleration of link2, calculate the angular acceleration of link3.
angularPos	Given the angle of link2, calculate the angle of link3.
angularVel	Given the angular velocity of link2, calculate the angular velocity of link3.
animation	Crank-slider animation.
couplerCurve	Calculate the coordinates of the coupler curve.
couplerPointAccel	Calculate the acceleration of the coupler point.
couplerPointPos	Calculate the position of the coupler point.
couplerPointVel	Calculate the velocity of the coupler point.
displayPosition	Display position of the crank-slider mechanism.
forceTorque	Calculate the joint forces and input torque at a given point.
forceTorques	Calculate the joint forces and input torque in the valid range of motion.
getJointLimits	Calculate crank-slider linkage input and output joint limits.
plotCouplerCurve	Plot the coupler curves.
plotForceTorques	Plot the joint forces and input torque curves.
setCouplerPoint	Set parameters for the coupler point.
setGravityCenter	Set parameters for mass centers of links.
setInertia	Set inerita parameters of links.
setAngularVel	Set constant angular velocity of linkage 2.
setLinks	Set lengths of links.
setMass	Set masses of links.
setNumPoints	Set number of points for animation and plot coupler curve.
sliderAccel	Given the angular acceleration of link2, calculate the acceleration of the slider.
sliderPos	Given the angular position of link2, calculate the position of the slider.
sliderVel	Given the angular velocity of link 2, calculate the velocity of the slider.
transAngle	Given input link position, calculate the transmission angle.
uscUnit	Specify the use of SI or US Customary units.

See Also

CCrankSlider::angularAccel

Synopsis
#include <crankslider.h>
double angularAccel(double *theta2*, **double** *theta3*, **double** *omega2*, double *omega3*, **double** *alpha2*);

Purpose
Given the angular acceleration of link2, calculate the angular acceleration of link 3.

Parameters

theta2, theta3 Double numbers indicating the angular position of link2 and link 3, respective.

omega2, omega3 Double numbers indicating the angular velocity of link2 and link3, respective.

alpha2 A double number indicating the angular acceleration of link2.

Return Value
This function returns the angular acceleration of link3.

Description

Given the angular acceleration of one link, this function calculates the angular acceleration of link3 of the crank-slider. *theta2, theta3* are double numbers which store the angle of link2 and link3, respective. *omega2, omega3* are double numbers which store the angular velocity of link2 and link3, respective. *alpha2* is a double number which stores the angular acceleration of link2. The returned value is the result of calculation.

Example

A crank-slider linkage has link lengths $r_2 = 1m, r_3 = 2m, r_4 = 0.5m$, and angle $\theta_1 = 10^o$. Given the angle θ_2, the angular velocity ω_2 and the angular acceleration α_2, calculate the angular acceleration α_3 of link3 for one circuit.

```
#include <math.h>
#include <stdio.h>
#include <crankslider.h>

int main() {
    double r2 = 1, r3 = 2, r4 = 0.5, theta1 = 10*M_PI/180;
    double rp = 2.5, beta = 20*M_PI/180;
    double theta2 = 50*M_PI/180;
    double omega2 = 5;  /*  rad/sec  */
    double alpha2 =-5;  /*  rad/sec*sec  */

    CCrankSlider crankslider;
    double first_theta3, sec_theta3;
    double omega3;
    double alpha3;

    crankslider.setLinks(r2, r3, r4, theta1);
    crankslider.angularPos(theta2, first_theta3, sec_theta3);

    //first solution
    omega3 = crankslider.angularVel(theta2, first_theta3, omega2);
    alpha3 = crankslider.angularAccel(theta2, omega2, first_theta3, omega3, alpha2);
    printf("First Solution:\n");
    printf("\tDeg/sec*sec:\t alpha3=%6.3f\n",alpha3*180/M_PI);
    printf("\tRad/sec*sec:\t alpha3=%6.4f\n",alpha3);

    //second solution
    omega3 = crankslider.angularVel(theta2, sec_theta3, omega2);
    alpha3 = crankslider.angularAccel(theta2, omega2, sec_theta3, omega3, alpha2);
    printf("Second Solution:\n");
    printf("\tDeg/sec*sec:\t alpha3=%6.3f\n",alpha3*180/M_PI);
    printf("\tRad/sec*sec:\t alpha3=%6.4f\n",alpha3);
}
```

Output

```
First Solution:
Deg/sec*sec:  alpha3=556.431
Rad/sec*sec:  alpha3=9.7115
Second Solution:
Deg/sec*sec:  alpha3=-556.431
Rad/sec*sec:  alpha3=-9.7115
```

CCrankSlider::angularPos

Synopsis
#include <crankslider.h>
void angularPos(**double** *theta2*, **double** &*first_solution,* **double** &*sec_solution*);

Purpose
Given the angle of one link, calculate the angle of the other links.

Parameters

theta2 A double number used for the angular positon of link2.

first_solution A double number used for the first solution of theta3.

sec_solution A double number used for the second solution of theta3.

Return Value
This function returns 0 on success and -1 on failure.

Description
Given the angular position of link2, this function computes the angular positions of link3.
Example
A crank-slider linkage has link lengths $r_2 = 1m, r_3 = 2m, r_4 = 0.5m$, and angle $\theta_1 = 10^o$. Given the angle θ_2, calculate the angular position θ_3 and θ_4 of link3 and link4, as well as, the coupler point position for each circuit, respectively.

```
#include <math.h>
#include <crankslider.h>
int main()
{
  CCrankSlider crankslider;
  double r2 = 1, r3 = 2, r4 = 0.5, theta1 = 10*M_PI/180;
  double rp = 2.5, beta = 20*M_PI/180;
  double theta2 = 70*M_PI/180;
  double first_theta3, sec_theta3;
  double complex p1, p2;  //two solution of coupler point P

  crankslider.setLinks(r2, r3, r4, theta1);
  crankslider.setCouplerPoint(rp, beta);
  crankslider.angularPos(theta2, first_theta3, sec_theta3);
  crankslider.couplerPointPos(theta2, p1, p2);

  /**** the first set of solutions ****/
  printf("theta3 = %6.3f, P = %6.3f \n", first_theta3, p1);
  /**** the second set of solutions ****/
  printf("theta3 = %6.3f, P = %6.3f \n", sec_theta3, p2);
}
```

Output

```
theta3 = -0.010, P = complex( 2.707, 1.750)
theta3 = -2.783, P = complex( 1.551, 3.128)
```

CCrankSlider::angularVel

Synopsis
#include <crankslider.h>
double angularVel(double *theta2,* **double** *theta2,* **double** *omega2*);

Purpose
Given the angular velocity of link2, calculate the angular velocity of link3.

Parameters

theta2, theta3 Double numbers used for the input angles of link2 and link3 respectively.

omega2 A double number used for the angular velocity of link2.

Return Value
This function returns the angular velocity of link3.

Description
Given the angular velocity of link2, this function calculates the angular velocities of link3. *theta2, theta3* are double numbers for link positions. *omega2* is a double number for angular velocity of link2.

Example
A crank-slider linkage has link lengths $r_2 = 1m, r_3 = 2m, r_4 = 0.5m$, and angle $\theta_1 = 10^o$. Given the angle θ_2 and the angular velocity ω_2, determine the Grashof type of the crank-slider linkage and calculate the angular velocities ω_3 and ω_4 of link3 and link4 for each circuit, respectively.

```
#include <math.h>
#include <stdio.h>
#include <crankslider.h>

int main()
{
    CCrankSlider crankslider;

    double r2 =1, r3 =2, r4 = 0.5, theta1 = 10*M_PI/180;
    double theta2 = 45*M_PI/180;
    double omega2 = 5;  /*  rad/sec  */
    double first_theta3, sec_theta3;
    double omega3;

    crankslider.setLinks(r2, r3, r4, theta1);
    crankslider.angularPos(theta2, first_theta3, sec_theta3);

    omega3 = crankslider.angularVel(theta2, first_theta3, omega2);
    printf("First Solution:\n");
    printf("\tDeg/sec:\t omega3=%6.3f\n", omega3*180/M_PI);
    printf("\tRad/sec:\t omega3=%6.3f\n", omega3);

    omega3 = crankslider.angularVel(theta2, sec_theta3, omega2);
    printf("Second Solution:\n");
    printf("\tDeg/sec:\t omega3=%6.3f\n", omega3*180/M_PI);
    printf("\tRad/sec:\t omega3=%6.3f\n", omega3);
}
```

Output

```
First Solution:
Deg/sec:  omega3=-117.414
Rad/sec:  omega3=-2.049
Second Solution:
Deg/sec:  omega3=117.414
Rad/sec:  omega3= 2.049
```

CCrankSlider::animation

Synopsis
#include <crankslider.h>
int animation(int *branchnum*, ... /* [**int** *outputtype*, **string_t** *datafilename*] */);

Syntax
animation(*branchnum*)
animation(*branchnum*, *outputtype*)
animation(*branchnum*, *outputtype*, *datafilename*)

Purpose
An animation of the crank-slider mechanism.

Parameters

branchnum an integer used for indicating which branch you want to draw. Each crank-slider has 2 branches.

outputtype an optional parameter to specify the output type of the animation.

datafilename an optional parameter to specify the file name of output.

Return Value
This function returns 0 on success and -1 on failure.

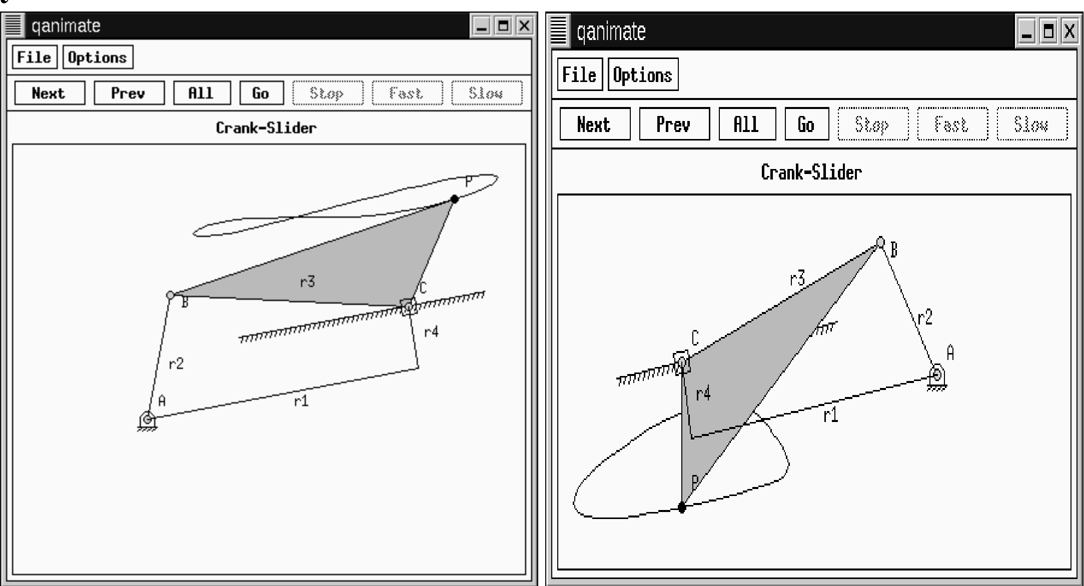
Description
This function simulates the motion of a crank-slider mechanism. *branchnum* is an integer which indicates the branch you want to draw. *outputtype* is an optional parameter used to specify how the animation should be outputted. *outputtype* can be either of the following macros: QANIMATE_OUTPUTTYPE_DISPLAY, QANIMATE_OUTPUTTYPE_FILE, QANIMATE_OUTPUTTYPE_STREAM. Specifying macro QANIMATE_OUTPUTTYPE_DISPLAY displays an animation on the screen. With macro, QANIMATE_OUTPUTTYPE_FILE, the animation data can be written to a file. Macro QANIMATE_OUTPUTTYPE_STREAM outputs the animation to the standard out. *datafilename* is an optional parameter to specify the output file name if you want to output the data to a file.

Example
For a Crank-Rocker crank-slider linkage with link lengths $r_2 = 1m, r_3 = 2m, r_4 = 0.5m$,and angle $\theta_1 = 10^o$ and coupler parameters: $r_p = 2.5m$ and $\beta = 20^o$, simulate the motion of the crank-slider linkage.

```
/*crankslider - Rocker */
#include <stdio.h>
```

```
#include <crankslider.h>

int main() {
    /* default specification of the four-bar linkage */
    double  r2 = 1, r3 = 2, r4= 0.5;//cranker-rocker
    double theta1 = 10*M_PI/180;
    double rp = 2.5, beta = 20*M_PI/180;
    int numpoints = 50;
    CCrankSlider crankslider;

    crankslider.setLinks(r2, r3, r4, theta1);
    crankslider.setCouplerPoint(rp, beta, TRACE_ON);
    crankslider.setNumPoints(numpoints);
    crankslider.animation(1);
    crankslider.animation(2);
}
```

Output



CCrankSlider::couplerCurve

Synopsis
#include <crankslider.h>
void couplerCurve(int *branchnum,* **double** *curvex[:],* **double** *curvey[:]*);

Purpose
Calculate the coordinates of the coupler curve.

Parameters

branchnum An integer used to indicatel the branch to be calculated.

curvex[:] A double array used for the x coordinate of coupler point through different input angles.

curvey[:] A double array used for the y coordinate of coupler point through different input angle.

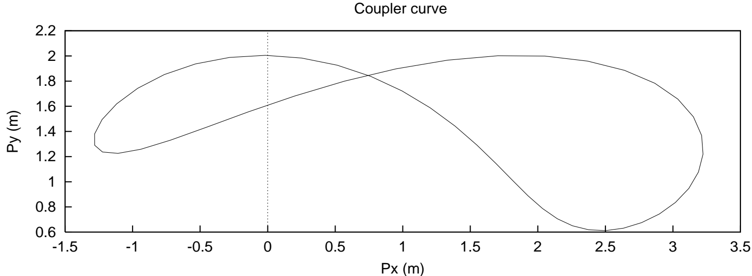
Return Value
None.

Description
This function calculates the coupler point position by looping the input angle. *curvex ,curvey* are the coordinate solutions of the coupler point positon.

Example
For a crank-slider with link lengths $r_2 = 1m, r_3 = 2m, r_4 = 0.5m, r_p = 2.5m$ and angles $\theta_1 = 0$, and $\beta = 20^o$, plot the position curve of the coupler point.

```
#include <stdio.h>
#include <crankslider.h>
#include <chplot.h>

int main() {
    CCrankSlider crankslider;
    double r2 = 1, r3 = 2, r4 = 0.5; // crank-crank
    double theta1 = 0;
    double rp = 2.5, beta = 20*M_PI/180;
    class CPlot pl;


    double RetCurvex[50], RetCurvey[50];

    crankslider.setLinks(r2, r3, r4, theta1);
    crankslider.setCouplerPoint(rp, beta);
    crankslider.setNumPoints(50);

    // use coupler points plot curve
    crankslider.couplerCurve(1, RetCurvex, RetCurvey);
    plotxy(RetCurvex,RetCurvey,"Coupler curve ",
           "Px (m)", "Py (m)", &pl);
    pl.sizeRatio(-1);
    pl.border(PLOT_BORDER_ALL, PLOT_ON);
    pl.plotting();
}
```

Output



See Also

CCrankSlider::couplerPointAccel

Synopsis
#include <crankslider.h>
double complex couplerPointAccel(double *theta2,* **double** *theta3,* **double** *omega2,* **double** *omega3,* **double** *alpha2,* **double** *alpha3*);

Purpose
Calculate the acceleration of the coupler point.

Parameters

theta2 A double number used for the angle of link 2.

theta3 A double number used for the angle of link 3.

omega2 A double number used for the angular velocity of link 2.

omega3 A double number used for the angular velocity of link 3.

alpha2 A double number used for the angular acceleration of link 2.

alpha3 A double number used for the angular acceleration of link 3.

Return Value
This function returns the acceleration of the coupler point.

Description
This function calculates the acceleration of the coupler point. *theta2, theta3, omega2, omega3, alpha2, alpha3* are double numbers. The return value is a complex number.

Example
For a crank-slider linkage with link lengths $r_2 = 1m, r_3 = 2m, r_4 = 0.5m, r_p = 2.5m$, and angles $\beta = 20^o$,

and $\theta_1 = 10^o$. Given the angle θ_2, the angular velocity ω_2 and the angular acceleration α_2, calculate the acceleration of the coupler point for each circuit, respectively.

```
#include <math.h>
#include <crankslider.h>

int main() {
    CCrankSlider crankslider;
    double r2 = 1, r3 = 2, r4 = 0.5, theta1 = 10*M_PI/180;
    double rp = 2.5, beta = 20*M_PI/180;
    double theta2 = 70*M_PI/180;
    double complex Ap[1:2];
    double omega2 = 5;  /*  rad/sec  */
    double alpha2 =-5;  /*  rad/sec*sec  */
    double first_theta3, sec_theta3;
    double first_omega3, sec_omega3;
    double first_alpha3, sec_alpha3;


    crankslider.setLinks(r2, r3, r4, theta1);
    crankslider.setCouplerPoint(rp, beta);
    crankslider.angularPos(theta2, first_theta3, sec_theta3);
    first_omega3 = crankslider.angularVel(theta2, first_theta3, omega2);
    sec_omega3 = crankslider.angularVel(theta2, sec_theta3, omega2);
    first_alpha3 = crankslider.angularAccel(theta2, omega2, first_theta3, first_omega3,
                                            alpha2);
    sec_alpha3 = crankslider.angularAccel(theta2, omega2, sec_theta3, sec_omega3,
                                          alpha2);
    Ap[1] = crankslider.couplerPointAccel(theta2, first_theta3, omega2, first_omega3,
                                          alpha2, first_alpha3);
    Ap[2] = crankslider.couplerPointAccel(theta2, sec_theta3, omega2, sec_omega3, alpha2,
                                          sec_alpha3);
    printf("Circuit 1: \n  CouplerAccleration: %f \n", Ap[1]);
    printf("Circuit 2: \n  CouplerAccleration: %f \n", Ap[2]);
}
```

Output

```
Circuit 1:
  CouplerAccleration: complex(-17.383000,1.825762)
Circuit 2:
  CouplerAccleration: complex(20.415886,-43.221212)
```

CCrankSlider::couplerPointPos

Synopsis
#include <crankslider.h>
void couplerPointPos(double *theta2*, **double complex** &*p1,* **double complex** &*p2*);

Purpose
Calculate the position of the coupler point.

Parameters

theta2 A double number used for the input angle of link2.

p1 A double complex number for the first solution of the coupler point.

p2 A double complex number for the second solution of the coupler point. *x*.

Return Value
None.

Description
This function calculates the position of the coupler point. *theta2* is the input angle. *p1,p2* are the two solutions of the coupler point position, respectively. Each is a complex number indicating the vector of the coupler point.

Example

```
see  CCrankSlider::angularPos().
```

CCrankSlider::couplerPointVel

Synopsis
#include <crankslider.h>
double complex couplerPointVel(double *theta2,* **double** *theta3,* **double** *omega2,* **double** *omega3*);

Purpose
Calculate the velocity of the coupler point.

Parameters

theta2 A double number used for the angle of link 2.

theta3 A double number used for the angle of link 3.

omega2 A double number used for the angular velocity of link 2.

omega3 A double number used for the angular velocity of link 3.

Return Value
This function returns the vector of the coupler velocity.

Description
This function calculates the vector of the coupler velocity. *theta2* is the angle of link2, *theta3* is the angle of link3. *omega2* is the angular velocity of link2, *omega3* is the angular velocity of link3. The vector ofthe coupler point velocity is returned.

Example
For a crank-slider linkage with link lengths $r_2 = 1m, r_3 = 2m, r_4 = 0.5m, r_p = 2.5m$, and angles $\beta = 20^o$, and $\theta_1 = 10^o$. Given the angle θ_2 and the angular velocity ω_2, calculate the velocity of the coupler point for each circuit, respectively.

```
#include <math.h>
#include <crankslider.h>

int main() {
    CCrankSlider crankslider;
    double r2 = 1, r3 = 2, r4 = 0.5, theta1 = 10*M_PI/180;
    double rp = 2.5, beta = 20*M_PI/180;
    double theta2 = 70*M_PI/180;
    double complex Vp[1:2];
    double omega2 = 5;  /*  rad/sec  */
    double first_theta3, sec_theta3;
    double first_omega3, sec_omega3;

    crankslider.setLinks(r2, r3, r4, theta1);
    crankslider.setCouplerPoint(rp, beta);
    crankslider.angularPos(theta2, first_theta3, sec_theta3);
    first_omega3 = crankslider.angularVel(theta2, first_theta3, omega2);
    sec_omega3 = crankslider.angularVel(theta2, sec_theta3, omega2);
    Vp[1] = crankslider.couplerPointVel(theta2, first_theta3, omega2, first_omega3);
    Vp[2] = crankslider.couplerPointVel(theta2, sec_theta3, omega2, sec_omega3);
    printf("Circuit 1: \n  CouplerVelocity: %f \n", Vp[1]);
    printf("Circuit 2: \n  CouplerVelocity: %f \n", Vp[2]);
}
```

Output

```
Circuit 1:
  CouplerVelocity: complex(-3.668337,-1.297038)
Circuit 2:
  CouplerVelocity: complex(-7.481037,3.246762)
```

CCrankSlider::displayPosition

Synopsis
#include <crankslider.h>
int displayPosition(double *theta2*, **double** *theta3*, ... /* [**int** *outputtype* [, [**char *** *filename*]] */);

Syntax
displayPosition(*theta2*, *theta3*)
displayPosition(*theta2*, *theta3*, *outputtype*)
displayPosition(*theta2*, *theta3*, *outputtype*, *filename*)

Purpose
Given θ_2 and θ_3, display the current position of the crank-slider mechanism.

Parameters

theta2 θ_2 angle.

theta3 θ_3 angle.

outputtype an optional argument for the output type.

filename an optional argument for the output file name.

Return Value

This function returns 0 on success and -1 on failure.

Description

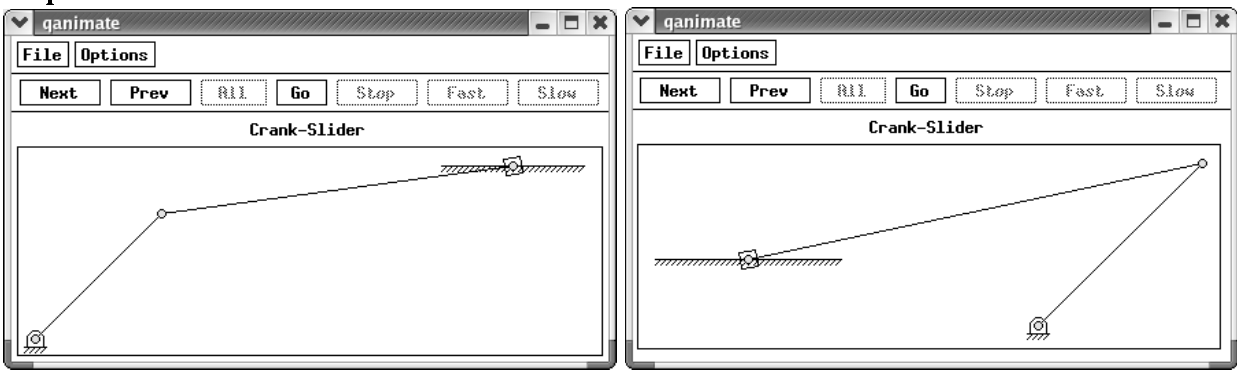
Given θ_2 and θ_3, display the current position of the crank-slider mechanism. `outputtype` is an optional parameter used to specify how the output should be handled. It may be one of the following macros: QANIMATE_OUTPUTTYPE_DISPLAY, QANIMATE_OUTPUTTYPE_FILE, QANIMATE_OUTPUTTYPE_STREAM. QANIMATE_OUTPUTTYPE_DISPLAY outputs the figure to the computer terminal. QANIMATE_OUTPUTTYPE_FILE writes the qanimate data onto a file. QANIMATE_OUTPUTTYPE_STREAM outputs the qanimate data to the standard out stream. `filename` is an optional parameter to specify the output file name.

Example

A crank-slider mechanism has link lengths $r_2 = 1m, r_3 = 2m, r_4 = 0.5m$, and $\theta_1 = 10^o$. Given the angle $\theta_2 = 45^o$, calculate the angular position θ_3 and display the crank-slider mechanism in its current position.

```
#include <math.h>
#include <crankslider.h>
int main()
{
  CCrankSlider crankslider;
  double r2 = 1, r3 = 2, r4 = 0.5, theta1 = 10*M_PI/180;
  double theta2 = 45*M_PI/180;
  double theta3_1, theta3_2;
  double complex rs_1, rs_2;

  crankslider.setLinks(r2, r3, r4, theta1);
  crankslider.angularPos(theta2, theta3_1, theta3_2);
  crankslider.displayPosition(theta2, theta3_1);
  crankslider.displayPosition(theta2, theta3_2);

   return 0;
}
```

Output



CCrankSlider::forceTorque

Synopsis

#include <crankslider.h>

void forceTorque(double *theta2,* **double** *theta3,* **double** *omega2,* **double** *omega3,* **double** *alpha2,* **double** *alpha3,* **double** *Fl*, **double complex** *as,* **array double** *x[9]*);

Purpose
Calculate the joint forces and input torque at a given point.

Parameters

theta2, theta3 Double numbers used for the angles of link2 and link3 respectively.

omega2, omega3 Double numbers used for the angular velocities of link2 and link3 respectively.

alpha2, alpha3 Double numbers used for the angular accelerations of link2 and link3 respectively.

Fl A double number used for the load force.

as A double number used for the acceleration of the slider.

x A double array used for forces and torque.

Return Value
None.

Description
This function calculates the joint forces and input torque for a given load torque. *theta* is a one-dimensional array of size 4 for the angles of the links. *omega* is a one-dimensional array of size 4 for the angular velocities of the links. *alpha* is a one-dimensional array of size 4 for the angular acclerations of the links. *Fl* is the load force. x contains the joint forces and input torque.

Example
For a crank-slider linkage with link lengths $r_2 = 1", r_3 = 2", r_4 = 0.5", r_p = 5"$, and angles $\beta = 20^o$, and $\theta_1 = 10^o$. Given the angle θ_2, the angular velocity ω_2 and the angular acceleration α_2, calculate the required torque applied to the input link2 in order to achieve the constant angular velocity for link2. Also calculate the joint forces exerted on the ground from links 1 and 4.

```
#include <math.h>
#include <crankslider.h>

int main() {
    CCrankSlider crankslider;
    double r2 = 1/12.0, r3 = 2/12.0, r4 = 0.5/12.0, theta1 = 10*M_PI/180;
    double rp = 2.5/12.0, beta = 20*M_PI/180;
    array double X[9];
    double g = 32.2;
    double rg2 = 2/12.0, rg3 = 6/12.0;
    double m2 = 0.8/g, m3 = 2.4/g, m4 = 1.4/g;
    double ig2 = 0.012/12.0, ig3 = 0.119/12.0, Fl=0;
    double theta2 = 45*M_PI/180;
    double omega2 = 5;  /*  rad/sec  */
    double alpha2 =-5;  /*  rad/sec*sec  */
    double first_theta3, sec_theta3;
    double first_omega3, sec_omega3;
    double first_alpha3, sec_alpha3;
```

```
    double complex first_as, sec_as;

    /* initialization of link parameters and
    inertia properties */

    crankslider.uscUnit(1);
    crankslider.setLinks(r2, r3, r4, theta1);
    crankslider.setCouplerPoint(rp, beta);
    crankslider.setGravityCenter(rg2, rg3);
    crankslider.setInertia(ig2, ig3);
    crankslider.setMass(m2, m3, m4);

    crankslider.angularPos(theta2, first_theta3, sec_theta3);
    first_omega3 = crankslider.angularVel(theta2, first_theta3, omega2);
    sec_omega3 = crankslider.angularVel(theta2, sec_theta3, omega2);
    first_alpha3 = crankslider.angularAccel(theta2, omega2, first_theta3, first_omega3,
                                            alpha2);
    sec_alpha3 = crankslider.angularAccel(theta2, omega2, sec_theta3, sec_omega3,
                                          alpha2);
    first_as = crankslider.sliderAccel(theta2, first_theta3, omega2, first_omega3,
                                       alpha2, first_alpha3);
    sec_as = crankslider.sliderAccel(theta2, sec_theta3, omega2, sec_omega3, alpha2,
                                     sec_alpha3);

    crankslider.forceTorque(theta2, first_theta3, omega2, first_omega3, alpha2,
                            first_alpha3, Fl, first_as, X);
    printf("first solution X = %.4f \n", X);


    crankslider.forceTorque(theta2, sec_theta3, omega2, sec_omega3, alpha2, sec_alpha3,
                            Fl, sec_as, X);
    printf("second solution X = %.4f \n", X);
}
```

Output

```
first solution X = 1.8977 -3.7569 1.8391 -4.6443 1.5499 -6.8617 -1.4595 8.2770 -0.2894

second solution X = 1.5349 -3.6799 1.4763 -4.5674 1.4706 -6.7348 -1.4354 8.1403 -0.2634
```

CCrankSlider::forceTorques

Synopsis

#include <crankslider.h>

void forceTorques(int *branchnum,* **double** *Fl,***array double** *time[:],* **array double** *f12x[:],* **array** double *f12y[:],* **array double** *f23x[:],* **array double** *f23y[:],* **array double** *f34x[:],* **array double** *f34y[:],* **array double** *f14x[:],* **array double** *f14y[:],* **array double** *ts[:]*);

Purpose

Calculate the joint forces and input torque in the valid range of motion.

Parameters

branchnum An integer number used for the branch that will be calculated.

Fl A double number for the load force.

time A double array to record time.

f12x, f12yble f23x, f23y, f34x, f34y, f14x, f14y Double arrays for forces.

ts A double array for input torque.

Return Value
None.

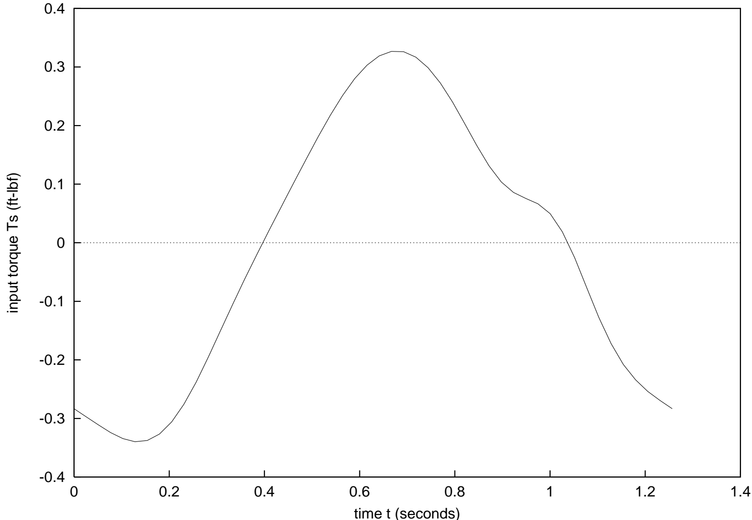
Description
This function calculates the joint forces and input torque in the valid range of motion. *branchnum* is the branch which will be plotted. *Fl* is the load force. *time* is an array to record time. *f12x, f12y, f23x, f23y, f34x, f34y, f14x, f14y* are arrays for forces. *ts* is a double array for input torque.

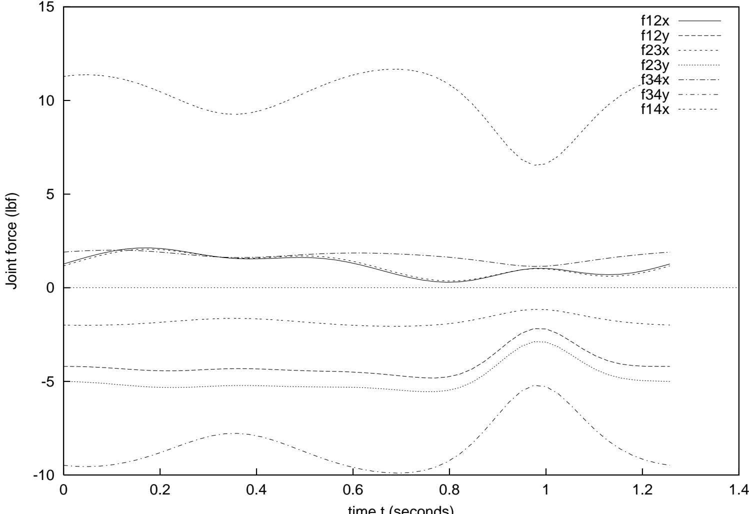
Example
For a crank-slider linkage with link lengths $r_2 = 1", r_3 = 2", r_4 = 0.5", r_p = 2.5"$, and angles $\beta = 20^o$, and $\theta_1 = 10^o$. Given the angle θ_2, use a loop to calculate the required torque applied to the input link2 in order to achieve the constant angular velocity for link2. Also calculate the joint forces exerted on the ground from links 1 and 4.

```
#include <math.h>
#include <crankslider.h>

int main()
{
    CCrankSlider crankslider;
    double r2 = 1/12.0, r3 = 2/12.0, r4 = 0.5/12.0, theta1 = 10*M_PI/180;
    double rp = 2.5/12.0, beta = 20*M_PI/180;
    double g = 32.2;
    double rg2 = 2/12.0, rg3 = 6/12.0;
    double m2 = 0.8/g, m3 = 2.4/g, m4 = 1.4/g;
    double ig2 = 0.012/12.0, ig3 = 0.119/12.0, Fl=0;
    int numpoint = 50;

    double  omega2 = 5;  /* constant omega2 */
    array double  time[numpoint], ts[numpoint];
    array double  f12x[numpoint], f12y[numpoint];
    array double  f23x[numpoint], f23y[numpoint];
    array double  f34x[numpoint], f34y[numpoint];
    array double  f14x[numpoint], f14y[numpoint];
    int branchnum = 2;
    int i;
    class CPlot pl;

    /* initialization of link parameters and
    inertia properties */
    crankslider.uscUnit(1);
    crankslider.setLinks(r2, r3, r4, theta1);
    crankslider.setCouplerPoint(rp, beta);
    crankslider.setGravityCenter(rg2, rg3);
    crankslider.setInertia(ig2, ig3);
    crankslider.setMass(m2, m3, m4);
    crankslider.setNumPoints(numpoint);
    crankslider.setAngularVel(omega2);
```

```
    crankslider.forceTorques(branchnum, Fl, time, f12x, f12y, f23x, f23y,
                             f34x, f34y, f14x, f14y, ts);  // calculate the forces and torque
    plotxy(time, ts, NULL, "time t (seconds)", "input torque Ts (ft-lbf)", &pl);
    pl.border(PLOT_BORDER_ALL, PLOT_ON);
    pl.plotting();
    plotxy(time, f12x, NULL, "time t (seconds)",  "Joint force (lbf)", &pl);
    pl.data2D(time, f12y);
    pl.data2D(time, f23x);
    pl.data2D(time, f23y);
    pl.data2D(time, f34x);
    pl.data2D(time, f34y);
    pl.data2D(time, f14x);
    pl.data2D(time, f14y);
    pl.legend("f12x",0);
    pl.legend("f12y",1);
    pl.legend("f23x",2);
    pl.legend("f23y",3);
    pl.legend("f34x",4);
    pl.legend("f34y",5);
    pl.legend("f14x",6);
    pl.border(PLOT_BORDER_ALL, PLOT_ON);
    pl.plotting();

}
```

Output





CCrankSlider::getJointLimits

Synopsis
#include <crankslider.h>
int getJointLimits(**double** &*inputmin,* **double** &*inputmax*);

Purpose
Calculate the crank-slider linkage input and output joint limits.

Parameters

inputmin A double number used for the minimum input angle.

inputmax A double number used for the maximum input angle.

Return Value
If the crank can fully rotate, return 1, otherwise return **FOURBAR_INVALID**.

Description
This function calculates the crank-slider linkage input limits. *inputmin, inputmax* are numbers for the limits of the crank-slider linkage.

Example
For a crankslider linkage with link lengths $r_2 = 1m, r_3 = 2m, r_4 = 0.5m$, and angle $\theta_1 = 0$, determine the input limit and output limit for each circuit.

```
#include <math.h>
#include <crankslider.h>

int main() {
    CCrankSlider crankslider;
    double r2 = 1, r3 = 2, r4 = 0.5, theta1 = 0;
    double inputlimitmin, inputlimitmax;
```

```
    crankslider.setLinks(r2, r3, r4, theta1);

    crankslider.getJointLimits(inputlimitmin, inputlimitmax);

    printf("input Range:\n");
    printf("    Lower limit: %7.2f\n",
                inputlimitmin*180./M_PI);
    printf("    Upper limit: %7.2f\n",
                inputlimitmax*180./M_PI);
}
```

Output

```
input Range:
    Lower limit:    0.00
    Upper limit:  360.00
```

CCrankSlider::plotCouplerCurve

Synopsis
#include <crankslider.h>
void plotCouplerCurve(**CPlot** **plot*, **int** *branchnum*);

Syntax
plotCouplerCurve(*&plot*, *branchnum*)

Purpose
Plot the coupler curve.

Parameters

&plot A pointer to a CPlot class variable used to format the plot of the branch to be drawn.

branchnum An integer used for indicating the branch which you want to draw.

Return Value
None.

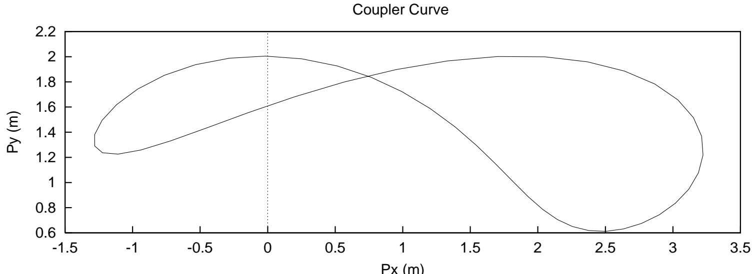
Description
This function plots the coupler curve. *&plot* is a pointer to a CPlot class variable used to format the plot of the branch to be drawn. *branchnum* is an integer number which indicates the branch you want to draw.

Example
For a crank-slider linkage with link lengths $r_2 = 1m, r_3 = 2m, r_4 = 0.5m, r_p = 2.5m$, and angles $\theta_1 = 0$, and $\beta = 20^o$, plot the position curve of the coupler point.

```
#include <stdio.h>
#include <crankslider.h>

int main() {
```

```
    CCrankSlider Slidercrank;
    double r2 = 1, r3 = 2, r4 = 0.5;
    double theta1 = 0;
    double rp = 2.5, beta = 20*M_PI/180;
    class CPlot plot;

    Slidercrank.setLinks(r2, r3, r4, theta1);
    Slidercrank.setCouplerPoint(rp, beta);
    Slidercrank.setNumPoints(50);
    Slidercrank.plotCouplerCurve(&plot, 1);  //display a coupler curve
}
```

Output



CCrankSlider::plotForceTorques

Synopsis
#include <crankslider.h>
void plotForceTorques(**CPlot** **plot*, **int** *branchnum*, **double** *Fl*);

Syntax
plotForceTorques(*&plot*, *branchnum*,*Fl*)

Purpose
Plot the joint forces and input torque curves.

Parameters

&plot A CPlot class variable used to format the plot of the branch to be drawn.

branchnum An integer number used for idicating the branch which will be plotted.

Fl A double number for the load Force.

Return Value

None.

Description

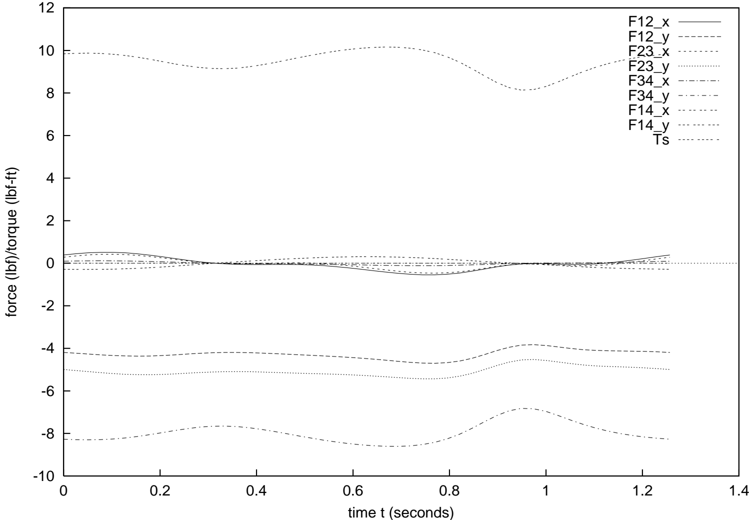
This function plots the joint forces and input torque curve. *&plot* is a CPlot class variable used to format the plot of the branch to be drawn. *branchnum* is an integer for the branch which will be plotted.

Example

A crank-slider linkage has link length $r_2 = 1”, r_3 = 2”, r_4 = 0.5”$, and angle $\theta_1 = 0^o$. Plot the joint forces and input torque versus time t, when the input link θ_2 is rotated counterclockwise with a constant input angular velocity.

```
#include <math.h>
#include <crankslider.h>

int main()
{
    CCrankSlider crankslider;
    double r2 = 1/12.0, r3 = 2/12.0, r4= 0.5/12.0;//cranker-rocker
    double theta1 = 0;
    double g = 32.2;
    double rg2 = 2/12.0, rg3 = 6/12.0;
    double m2 = 0.8/g, m3 = 2.4/g, m4 = 1.4/g;
    double ig2 = 0.012/12.0, ig3 = 0.119/12.0, Fl=0;
    int numpoint = 50;
    double  omega2 = 5;  /* constant omega2 */
    class CPlot plot;

    /* initialization of link parameters and
    inertia properties */

    crankslider.uscUnit(1);
    crankslider.setLinks(r2, r3, r4, theta1);
    crankslider.setGravityCenter(rg2, rg3);
    crankslider.setInertia(ig2, ig3);
    crankslider.setMass(m2, m3, m4);
    crankslider.setNumPoints(numpoint);
    crankslider.setAngularVel(omega2);
    crankslider.plotForceTorques(&plot,1,Fl); //first branch
}
```

Output



CCrankSlider::setCouplerPoint

Synopsis
#include <crankslider.h>
void setCouplerPoint(double *rp, beta*, ... /* [**int** trace] */);

Syntax
setCouplerPoint(*rp*, *beta*)
setCouplerPoint(*rp*, *beta*, *trace*)

Purpose
Set parameters for the coupler point.

Parameters

rp A double number used for rp.

beta A double number for beta.

trace An optional parameter of **int** type specifying either macro TRACE_OFF or TRACE_ON to indicte whether the coupler curve should be traced during animation.

Return Value
None.

Description
This function sets the parameters of the coupler point.

Example
see **CCrankSlider::plotCouplerCurve**().

CCrankSlider::setGravityCenter

Synopsis
#include <crankslider.h>
void setGravityCenter(double *rg2,* **double** *rg3*);

Purpose
Set the mass center parameters of the links.

Parameters

rg2 A double number used for the distance from joint A_0 to the center of gravity of link 2.

rg3 A double number used for the distance from joint A to the center of gravity of link 3.

Return Value
None.

Description
This function sets parameters for mass centers of links.

Example
see **CCrankSlider::plotForceTorques**().

CCrankSlider::setInertia

Synopsis
#include <crankslider.h>
void setInertia(double *ig2,* **double** *ig3*);

Purpose
Set inertia parameters of the links.

Parameters

ig2 A double number used for the inertia of link 2.

ig3 A double number used for the inertia of link 3.

Return Value
None.

Description
This function sets inertia parameters of the links.

Example
see **CCrankSlider::plotForceTorques**().

CCrankSlider::setAngularVel

Synopsis
#include <crankslider.h>

void setAngularVel(**double** *omega2*);

Purpose
Set the constant angular velocity of link2.

Parameters

omega2 A double number used for the constant input angular velocity of link2.

Return Value
None.

Description
This function sets the constant angular velocity of link2. It is used in conjunction with member functions **forceTorques**() and **plotForceTorques**().

Example
see **CCrankSlider**::**plotForceTorques**().

CCrankSlider::setLinks

Synopsis
#include <crankslider.h>
int setLinks(**double** *r2,* **double** *r3,* **double** *r4,* **double** *theta1*);

Purpose
Set the lengths of the links.

Parameters

r2,r3,r4 Double numbers used for the lengths of each link.

theta1 A double number representing the angle between link1 and horizontal.

Return Value
This function returns 0 on success and -1 on failure.

Description
This function sets the lengths of links and detects if the lengths can construct a crank-slider mechanism. If not, return -1.

Example
see **CCrankSlider**::**plotCouplerCurve**().

CCrankSlider::setMass

Synopsis
#include <crankslider.h>
void setMass(**double** *m2,* **double** *m3,* **double** *m4*);

Purpose
Set the mass of each link.

Parameters

m2,m3,m4 Double numbers used for the mass of the links.

Return Value
None.

Description
This function sets the masses of links.

Example
see **CCrankSlider**::**plotForceTorques**().

CCrankSlider::setNumPoints

Synopsis
#include <crankslider.h>
void setNumPoints(**int** *numpoints*);

Purpose
Set the number of points for the animation and the coupler curve.

Parameters

numpoints An integer number used for the number of points.

Return Value
None.

Description
This function sets the number of points for the animation and the coupler curve plot.

Example
see **CCrankSlider**::**animation**().

CCrankSlider::sliderAccel

Synopsis
#include <crankslider.h>
double sliderAccel(**double** *theta2*, **double** *theta3*, **double** *omega2*, double *omega3*, **double** *alpha2*, **double** *alpha3*);

Purpose
Given the angular acceleration of link2, calculate the acceleration of the slider.

Parameters

theta2, theta3 Double numbers indicating the angular position of link2 and link 3, respective.

omega2, omega3 Double numbers indicating the angular velocity of link2 and link3, respective.

alpha2, alpha3 Double numbers indicating the angular acceleration of link2 and link3, respective.

Return Value
This function returns the acceleration of the slider.

Description
Given the angular acceleration of one link, this function calculates the acceleration of slider. *theta2, theta3* are double numbers which store the angle of link2 and link3, respective. *omega2, omega3* are double numbers which store the angular velocity of link2 and link3, respective. *alpha2, alpha3* are double numbers which store the angular acceleration of link2. The returned value is the result of calculation.

Example
A crank-slider linkage has link lengths $r_2 = 1m, r_3 = 2m, r_4 = 0.5m$, and angle $\theta_1 = 10^o$. Given the angle θ_2, the angular velocity ω_2 and the angular acceleration α_2, calculate the acceleration of the slider for each circuit.

```
#include <math.h>
#include <crankslider.h>

int main() {
    CCrankSlider crankslider;
    double r2 = 1, r3 = 2, r4 = 0.5, theta1 = 10*M_PI/180;
    double rp = 2.5, beta = 20*M_PI/180;
    double theta2 = 70*M_PI/180;
    double complex As[1:2];
    double omega2 = 5;  /*  rad/sec  */
    double alpha2 =-5;  /*  rad/sec*sec  */
    double first_theta3, sec_theta3;
    double first_omega3, sec_omega3;
    double first_alpha3, sec_alpha3;


    crankslider.setLinks(r2, r3, r4, theta1);
    crankslider.setCouplerPoint(rp, beta);
    crankslider.angularPos(theta2, first_theta3, sec_theta3);
    first_omega3 = crankslider.angularVel(theta2, first_theta3, omega2);
    sec_omega3 = crankslider.angularVel(theta2, sec_theta3, omega2);
    first_alpha3 = crankslider.angularAccel(theta2, omega2, first_theta3,
                                             first_omega3, alpha2);
    sec_alpha3 = crankslider.angularAccel(theta2, omega2, sec_theta3, sec_omega3,
                                           alpha2);
    As[1] = crankslider.sliderAccel(theta2, first_theta3, omega2, first_omega3,
                                     alpha2, first_alpha3);
    As[2] = crankslider.sliderAccel(theta2, sec_theta3, omega2, sec_omega3,
                                     alpha2, sec_alpha3);

    printf("Circuit 1: \n  SliderAccleration: %f \n", As[1]);
    printf("Circuit 2: \n  SliderAccleration: %f \n", As[2]);
}
```

Output

```
Circuit 1:
  SliderAcceleration: complex(-6.857122, -1.209097)
Circuit 2:
  SliderAcceleration: complex(-9.234371, -1.628269)
```

CCrankSlider::sliderPos

Synopsis
#include <crankslider.h>
void sliderPos(double *theta2,* **double complex** *&first_solution,* **double complex** *&sec_solution,*);

Purpose
Given the angular velocity of link2, calculate the position of the slider.

Parameters

theta2 Double numbers used for the input angle of link2.

first_solution Double complex number used to store the first solution of the slider position.

sec_solution Double complex number used to store the second solution of the slider position.

Return Value
No return value.

Description
Given the angular position of link2, this function calculates the position of the slider. *theta2* is a double number for the position of link 2. *first_solution, sec_solution* are double complex numbers for the two possible solutions to the position of the slider.

Example
A crank-slider linkage has link lengths $r_2 = 1m, r_3 = 2m, r_4 = 0.5m$, and angle $\theta_1 = 10^o$. Given the angle θ_2, determine the position of the slider for each circuit.

```
#include <math.h>
#include <crankslider.h>

int main() {
    CCrankSlider crankslider;
    double r2 = 1, r3 = 2, r4 = 0.5, theta1 = 10*M_PI/180;
    double rp = 2.5, beta = 20*M_PI/180;
    double theta2 = 70*M_PI/180;
    double complex Ps[1:2];
    double first_theta3, sec_theta3;

    crankslider.setLinks(r2, r3, r4, theta1);
    crankslider.setCouplerPoint(rp, beta);
    crankslider.angularPos(theta2, first_theta3, sec_theta3);
    crankslider.sliderPos(theta2, Ps[1], Ps[2]);
    printf("Circuit 1: \n  Slider Position: %f \n", Ps[1]);
    printf("Circuit 2: \n  Slider Position: %f \n", Ps[2]);

    return 0;
}
```

Output

```
Circuit 1:
  Slider Position: complex(2.341929,0.920659)
Circuit 2:
  Slider Position: complex(-1.530770,0.237797)
```

CCrankSlider::sliderVel

Synopsis
#include <crankslider.h>
double sliderVel(double *theta2,* **double** *theta3,* **double** *omega2,* **double** *omega3*);

Purpose
Given the angular velocity of link2, calculate the velocity of the slider.

Parameters

theta2, theta3 Double numbers used for the input angles of link2 and link3 respectively.

omega2, omega3 Double numbers used for the angular velocity of link2 and link3 respectively.

Return Value
This function returns the velocity of the slider.

Description
Given the angular velocity of link2, this function calculates the velocity of the slider. *theta2, theta3* are double numbers for link positions. *omega2, omega3* are double numbers for the angular velocities of links.

Example
A crank-slider linkage has link lengths $r_2 = 1m, r_3 = 2m, r_4 = 0.5m$, and angle $\theta_1 = 10^o$. Given the angle θ_2 and the angular velocity ω_2, determine the slider velocity for each circuit.

```
#include <math.h>
#include <crankslider.h>

int main() {
    CCrankSlider crankslider;
    double r2 = 1, r3 = 2, r4 = 0.5, theta1 = 10*M_PI/180;
    double rp = 2.5, beta = 20*M_PI/180;
    double theta2 = 70*M_PI/180;
    double complex Vs[1:2];
    double omega2 = 5;  /*  rad/sec  */
    double first_theta3, sec_theta3;
    double first_omega3, sec_omega3;

    crankslider.setLinks(r2, r3, r4, theta1);
    crankslider.setCouplerPoint(rp, beta);
    crankslider.angularPos(theta2, first_theta3, sec_theta3);
    first_omega3 = crankslider.angularVel(theta2, first_theta3, omega2);
    sec_omega3 = crankslider.angularVel(theta2, sec_theta3, omega2);
    Vs[1] = crankslider.sliderVel(theta2, first_theta3, omega2, first_omega3);
    Vs[2] = crankslider.sliderVel(theta2, sec_theta3, omega2, sec_omega3);
```

```
    printf("Circuit 1: \n  SliderVelocity: %f \n", Vs[1]);
    printf("Circuit 2: \n  SliderVelocity: %f \n", Vs[2]);
}
```

Output

```
Circuit 1:
  SliderVelocity: complex(-4.722665, -0.832733)
Circuit 2:
  SliderVelocity: complex(-3.806022, -0.671104)
```

CCrankSlider::transAngle

Synopsis
#include <crankslider.h>
void transAngle(**double** &*gamma1,* **double** &*gamma2,* **double** *theta2*);

Purpose
Given the position of the input link, calculate the transmission angle.

Parameters

gamma1 A double number used for the first solution.

gamma2 A double number used for the second solution.

theta2 A double number used for the angular positon of link2.

Return Value
None.

Description
Given the position of the input link, this function calculates the transmission angle.
Example
A crank-slider linkage has link lengths $r_2 = 1m, r_3 = 2m, r_4 = 0.5m$, and angle $\theta_1 = 10^o$. Given the angle θ_2, calculate the transmission angle for each circuit, respectively.

```
#include <math.h>
#include <stdio.h>
#include <crankslider.h>

int main() {
    double r[1:4], theta1, theta2;
    double gamma1, gamma2;
    CCrankSlider crankslider;

    /* default specification of the four-bar linkage */
    r[2] = 1; r[3] = 2; r[4] = 0.5;
    theta1 = 10*M_PI/180; theta2=45*M_PI/180;

    printf("Results: Interactive Four-Bar Linkage Transmission Angle Analysis\n\n");
```

```
    crankslider.setLinks(r[2], r[3], r[4], theta1);

    crankslider.transAngle(gamma1, gamma2, theta2);
    printf("\n Circuit 1: Transmission Angle\n\n");
    printf("\tDegrees:\t gamma=%6.3f \n",gamma1*180/M_PI);
    printf("\tRadians:\t gamma=%6.4f \n",gamma1);
    printf("\n Circuit 2: Transmission Angle\n\n");
    printf("\tDegrees:\t gamma=%6.3f \n",gamma2*180/M_PI);
    printf("\tRadians:\t gamma=%6.4f \n",gamma2);
}
```

Output

```
Results: Interactive Four-Bar Linkage Transmission Angle Analysis


 Circuit 1: Transmission Angle

Degrees:  gamma=87.892
Radians:  gamma=1.5340

 Circuit 2: Transmission Angle

Degrees:  gamma=-87.892
Radians:  gamma=-1.5340
```

CCrankSlider::uscUnit

Synopsis
#include <crankslider.h>
void uscUnit(bool *unit*);

Purpose
Specify the use of SI or US Customary units in analysis.

Parameters

unit A boolean argument, where `true` indicates US Customary units are desired and `false` indicates SI units.

Return Value
None.

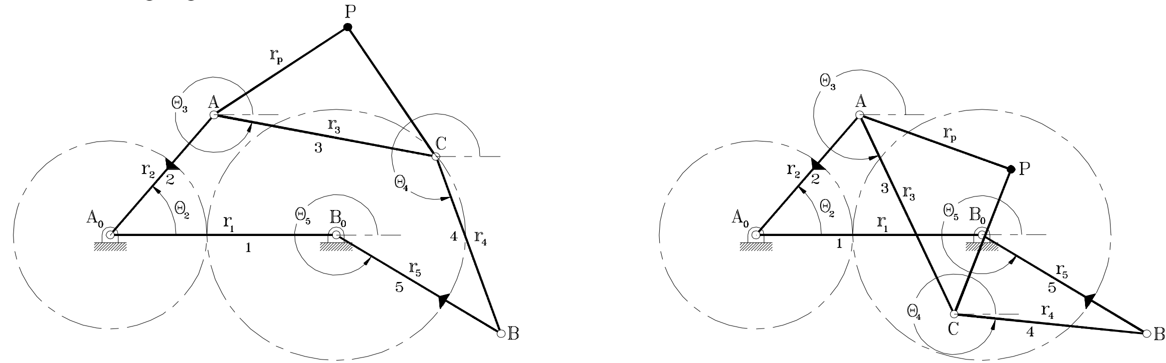
Description
This function specifies the whether SI or US Customary units are used.If `unit = true`, then US Customary units are used; otherwise, SI units are used. By default, SI units are assumed. This member function shall be called prior any other member function calls.

Appendix D

Class CGearedFivebar

CGearedFivebar

The header file **fivebar.h** includes header file **linkage.h**. The header file **fivebar.h** also contains a declaration of class **CGearedFivebar**. The **CGearedFivebar** class provides a means to analyze geared fivebar linkage within a Ch language environment.



Public Data
None.

Public Member Functions

Functions	Descriptions
angularAccel	Given the angular acceleration of one link, calculate the angular acceleration of other links.
angularPos	Given the angle of one link, caculate the angle of other links.
angularVel	Given the angular velocity of one link, calculate the angular velocity of other links.
animation	Fourbar linkage animation.
couplerCurve	Calculate the coordinates of the coupler curve.
couplerPointAccel	Calculate the acceleration of the coupler point.
couplerPointPos	Calculate the position of the coupler point.
couplerPointVel	Calculate the velocity of the coupler point.

displayPosition	Display the geared fivebar positions.
plotCouplerCurve	Plot the coupler curves.
setCouplerPoint	Set parameters for the coupler point.
setAngularVel	Set constant angular velocity of linkage 2.
setLinks	Set lengths of links.
setNumPoints	Set number of points for animation and plot coupler curve.
uscUnit	Specify the use of SI or US Customary units.

See Also

CGearedFivebar::angularAccel

Synopsis
#include <fivebar.h>
void angularAccel(double *theta[1:5]*, **double** *omega[1:5]*, **double** *alpha[1:5]*);

Purpose
Given the angular acceleration of the input link, calculate the angular accelerations of other links.

Parameters

theta An array of double data type with angles of links.

omega An array of double data type with angular velocities of links.

alpha An array of double data type with angular accelerations of links.

Return Value
None.

Description
Given the angular acceleration of the input link, this function calculates the angular acceleration of the remaining moving links of the geared fivebar. *theta* is a one-dimensional array of size 5 which stores the angle of each link. *omega* is a one-dimensional array of size 5 which stores the angular velocity of each link. *alpha* is a one-dimensional array of size 5 which stores the angular acceleration of each link. The result of calculation is stored in array *alpha*.

Example
A geared fivebar linkage has parameters $r_1 = 7m, r_2 = 5m, r_3 = 10m, r_4 = 10m, r_5 = 2m$, $\theta_1 = 10^o$, $\lambda = -2.5$, $\phi = 35^o$. Given the angle θ_2, the angular velocity ω_2 and the angular acceleration α_2, calculate the angular accelerations of the other links.

```
/****************************************************************
 *  This example is for calculating the angular acceleration of  *
 *  link3 and link4.                                             *
 ****************************************************************/

#include <fivebar.h>
```

```
int main()
{
   double r[1:5],
          theta_1[1:5], theta_2[1:5],
          omega_1[1:5], omega_2[1:5],
          alpha_1[1:5], alpha_2[1:5],
          phi, lambda;
   double complex P[1:2], Vp[1:2], Ap[1:2];
   CGearedFivebar gearedbar;

   /* Setup geared fivebar linkage. */
   r[1] = 7;  r[2] = 5;  r[3] = 10;  r[4] = 10;  r[5] = 2;
   phi = 35*M_PI/180;  lambda = -2.5;
   theta_1[1] = 10*(M_PI/180);  theta_2[1] = 10*(M_PI/180);
   theta_1[2] = 50*(M_PI/180);  theta_2[2] = 50*(M_PI/180);
   omega_1[2] = 5*(M_PI/180);  omega_2[2] = 5*(M_PI/180);
   alpha_1[2] = 0;  alpha_2[2] = 0;
   theta_1[5] = lambda*theta_1[2] + phi;
   theta_2[5] = lambda*theta_2[2] + phi;
   if(theta_1[5] < -M_PI)
   {
      theta_1[5] += 2*M_PI;
      theta_2[5] += 2*M_PI;
   }
   if(theta_1[5] > M_PI)
   {
      theta_1[5] -= 2*M_PI;
      theta_2[5] -= 2*M_PI;
   }

   /* Perform geared fivebar linkage analysis. */
   gearedbar.setLinks(r[1], r[2], r[3], r[4], r[5], theta_1[1]);
   gearedbar.setLambda(lambda);
   gearedbar.setPhi(phi);
   gearedbar.angularPos(theta_1, theta_2);
   gearedbar.angularVel(theta_1, omega_1);
   gearedbar.angularVel(theta_2, omega_2);
   gearedbar.angularAccel(theta_1, omega_1, alpha_1);
   gearedbar.angularAccel(theta_2, omega_2, alpha_2);

   /* print results on screen */
   printf("1st Circuit:\n");
   printf("\talpha3 = %.3f rad/sec^2 (%.2f deg/sec^2)\n", alpha_1[3],
          alpha_1[3]*(180/M_PI));
   printf("\talpha4 = %.3f rad/sec^2 (%.2f deg/sec^2)\n\n", alpha_1[4],
          alpha_1[4]*(180/M_PI));
   printf("2nd Circuit:\n");
   printf("\talpha3 = %.3f rad/sec^2 (%.2f deg/sec^2)\n", alpha_2[3],
          alpha_2[3]*(180/M_PI));
   printf("\talpha4 = %.3f rad/sec^2 (%.2f deg/sec^2)\n\n", alpha_2[4],
          alpha_2[4]*(180/M_PI));

  return 0;
}
```

Output

```
1st Circuit:
alpha3 = 0.003 rad/sec^2 (0.20 deg/sec^2)
alpha4 = -0.001 rad/sec^2 (-0.04 deg/sec^2)
```

```
2nd Circuit:
alpha3 = 0.013 rad/sec^2 (0.72 deg/sec^2)
alpha4 = 0.020 rad/sec^2 (1.17 deg/sec^2)
```

CGearedFivebar::angularPos

Synopsis
#include <fivebar.h>
void angularPos(double *theta_1[1:5]*, **double** *theta_2[1:5]*);

Purpose
Given the angle of the input link, calculate the angles of other links.

Parameters

theta_1 A double array with of size 5 for the first solution.

theta_2 A double array of size 5 for the second solution.

Return Value
None.

Description
Given the angular position of one link of a fivebar linkage, this function computes the angular positions of the remaining moving links. *theta_1* is a one-dimensional array of size 5 which stores the first solution of the angular positions. *theta_2* is a one-dimensional array of size 5 which stores the second solution of angular positions.

Example
A geared fivebar linkage has parameters $r_1 = 7m, r_2 = 5m, r_3 = 10m, r_4 = 10m, r_5 = 2m$, $\theta_1 = 10^o$, $\lambda = -2.5$, and $\phi = 35^o$. Given the angle θ_2, calculate the angular positions of the other links for each respective circuit.

```
/************************************************************
 *  This example is for calculating the angular position of  *
 *  link3 and link4.                                         *
 ************************************************************/

#include <fivebar.h>

int main()
{
   double r[1:5],
          theta_1[1:5], theta_2[1:5],
          phi, lambda;
   CGearedFivebar gearedbar;

   /* Setup geared fivebar linkage. */
   r[1] = 7;  r[2] = 5;  r[3] = 10;  r[4] = 10;  r[5] = 2;
   phi = 35*M_PI/180;  lambda = -2.5;
   theta_1[1] = 10*(M_PI/180);  theta_2[1] = 10*(M_PI/180);
```

```
   theta_1[2] = 50*(M_PI/180);  theta_2[2] = 50*(M_PI/180);
   theta_1[5] = lambda*theta_1[2] + phi;
   theta_2[5] = lambda*theta_2[2] + phi;
   if(theta_1[5] < -M_PI)
   {
      theta_1[5] += 2*M_PI;
      theta_2[5] += 2*M_PI;
   }
   if(theta_1[5] > M_PI)
   {
      theta_1[5] -= 2*M_PI;
      theta_2[5] -= 2*M_PI;
   }

   /* Perform geared fivebar linkage analysis. */
   gearedbar.setLinks(r[1], r[2], r[3], r[4], r[5], theta_1[1]);
   gearedbar.setLambda(lambda);
   gearedbar.setPhi(phi);
   gearedbar.angularPos(theta_1, theta_2);

   /* May run into problem with theta4 being in opposite quadrant. */
   theta_1[4] += M_PI;  theta_2[4] += M_PI;
   if(theta_1[4] < -M_PI)
      theta_1[4] += 2*M_PI;
   if(theta_1[4] > M_PI)
      theta_1[4] -= 2*M_PI;
   if(theta_2[4] < -M_PI)
      theta_2[4] += 2*M_PI;
   if(theta_2[4] > M_PI)
      theta_2[4] -= 2*M_PI;

   /* print results on screen */
   printf("1st Circuit:\n");
   printf("\ttheta3 = %.3f radians (%.2f degrees)\n", theta_1[3],
          theta_1[3]*(180/M_PI));
   printf("\ttheta4 = %.3f radians (%.2f degrees)\n", theta_1[4],
          theta_1[4]*(180/M_PI));
   printf("\ttheta5 = %.3f radians (%.2f degrees)\n\n", theta_1[5],
          theta_1[5]*(180/M_PI));
   printf("2nd Circuit:\n");
   printf("\ttheta3 = %.3f radians (%.2f degrees)\n", theta_2[3],
          theta_2[3]*(180/M_PI));
   printf("\ttheta4 = %.3f radians (%.2f degrees)\n", theta_2[4],
          theta_2[4]*(180/M_PI));
   printf("\ttheta5 = %.3f radians (%.2f degrees)\n\n", theta_2[5],
          theta_2[5]*(180/M_PI));

  return 0;
}
```

Output

```
1st Circuit:
theta3 = 0.374 radians (21.40 degrees)
theta4 = -2.169 radians (-124.27 degrees)
theta5 = -1.571 radians (-90.00 degrees)

2nd Circuit:
theta3 = -2.169 radians (-124.27 degrees)
```

```
theta4 = 0.374 radians (21.40 degrees)
theta5 = -1.571 radians (-90.00 degrees)
```

CGearedFivebar::angularVel

Synopsis
#include <fivebar.h>
void angularVel(double *theta[1:5],* **double** *omega[1:5]*);

Purpose
Given the angular velocity of one link, calculate the angular velocities of other links.

Parameters

theta A double array used for the input angle of links.

omega A double array used for the angular velocities of links.

Return Value
None.

Description
Given the angular velocity of one link, this function calculates the angular velocities of the remaining two moving links of the fivebar. *theta* is an array for link positions. *omega* is an array for angular velocity of links.

Example
A geared fivebar linkage has parameters $r_1 = 7m, r_2 = 5m, r_3 = 10m, r_4 = 10m, r_5 = 2m$, $\theta_1 = 10^o$, $\lambda = -2.5$, and $\phi = 35^o$. Given the angle θ_2 and angular velocity ω_2, calculate the angular velocities ω_3, ω_4 of link3, and link4 for each circuit, respectively.

```
/************************************************************
 *  This example is for calculating the angular velocity of  *
 *  link3 and link4.                                         *
 ************************************************************/

#include <fivebar.h>

int main()
{
   double r[1:5],
          theta_1[1:5], theta_2[1:5],
          omega_1[1:5], omega_2[1:5],
          phi, lambda;
   CGearedFivebar gearedbar;

   /* Setup geared fivebar linkage. */
   r[1] = 7;  r[2] = 5;  r[3] = 10;  r[4] = 10;  r[5] = 2;
   phi = 35*M_PI/180;  lambda = -2.5;
   theta_1[1] = 10*(M_PI/180);  theta_2[1] = 10*(M_PI/180);
   theta_1[2] = 50*(M_PI/180);  theta_2[2] = 50*(M_PI/180);
   omega_1[2] = 5*(M_PI/180);  omega_2[2] = 5*(M_PI/180);
```

```
   theta_1[5] = lambda*theta_1[2] + phi;
   theta_2[5] = lambda*theta_2[2] + phi;
   if(theta_1[5] < -M_PI)
   {
      theta_1[5] += 2*M_PI;
      theta_2[5] += 2*M_PI;
   }
   if(theta_1[5] > M_PI)
   {
      theta_1[5] -= 2*M_PI;
      theta_2[5] -= 2*M_PI;
   }

   /* Perform geared fivebar linkage analysis. */
   gearedbar.setLinks(r[1], r[2], r[3], r[4], r[5], theta_1[1]);
   gearedbar.setLambda(lambda);
   gearedbar.setPhi(phi);
   gearedbar.angularPos(theta_1, theta_2);
   gearedbar.angularVel(theta_1, omega_1);
   gearedbar.angularVel(theta_2, omega_2);

   /* print results on screen */
   printf("1st Circuit:\n");
   printf("\tomega3 = %.3f rad/sec (%.2f deg/sec)\n", omega_1[3], omega_1[3]*(180/M_PI));
   printf("\tomega4 = %.3f rad/sec (%.2f deg/sec)\n", omega_1[4], omega_1[4]*(180/M_PI));
   printf("2nd Circuit:\n");
   printf("\tomega3 = %.3f rad/sec (%.2f deg/sec)\n", omega_2[3], omega_2[3]*(180/M_PI));
   printf("\tomega4 = %.3f rad/sec (%.2f deg/sec)\n", omega_2[4], omega_2[4]*(180/M_PI));

   return 0;
}
```

Output

```
1st Circuit:
omega3 = -0.051 rad/sec (-2.94 deg/sec)
omega4 = 0.109 rad/sec (6.25 deg/sec)
2nd Circuit:
omega3 = -0.035 rad/sec (-2.01 deg/sec)
omega4 = 0.036 rad/sec (2.05 deg/sec)
```

CGearedFivebar::animation

Synopsis
#include <fivebar.h>
int animation(int *branchnum*, ... /* [**int***outputtype*, **string_t** *datafilename*] */);

Syntax
animation(*branchnum*)
animation(*branchnum*, *outputtype*)
animation(*branchnum*, *outputtype*, *datafilename*)

Purpose
An animation of a geared fivebar mechanism.

Parameters

branchnum an integer used for indicating which branch you want to draw.

outputtype an optional parameter to specify the output type of the animation.

datafilename an optional parameter to specify the file name of output.

Return Value
This function returns 0 on success and -1 on failure.

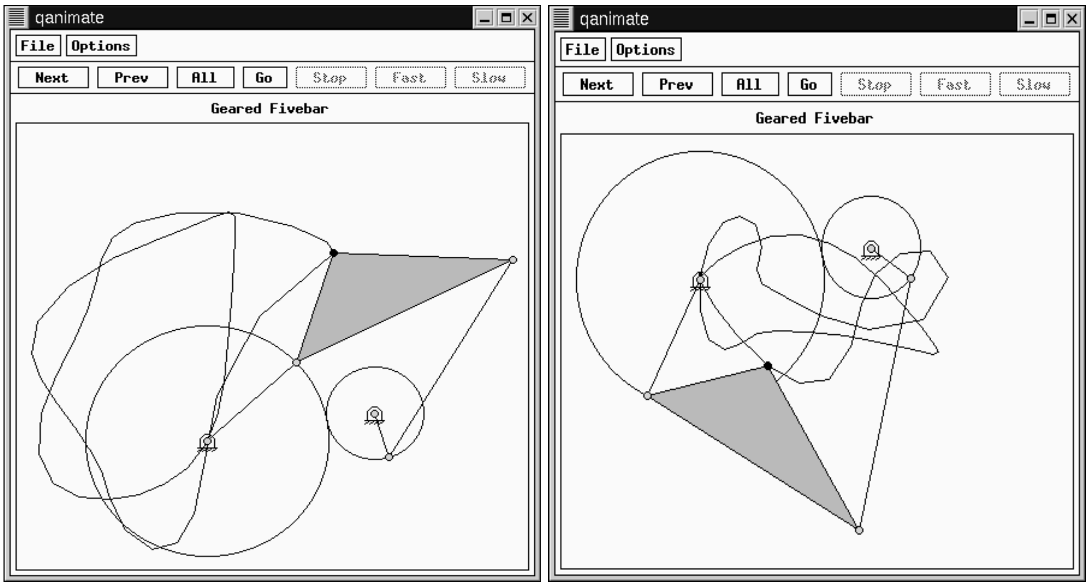
Description
This function simulates the motion of a fivebar mechanism. *branchnum* is an integer number which indicates the branch you want to draw. *outputtype* is an optional parameter used to specify how the animation should be outputted. *outputtype* can be either of the following macros:
QANIMATE_OUTPUTTYPE_DISPLAY, QANIMATE_OUTPUTTYPE_FILE,
QANIMATE_OUTPUTTYPE_STREAM. QANIMATE_OUTPUTTYPE_DISPLAY creates an animation on the screen. QANIMATE_OUTPUTTYPE_FILE writes the animation data onto a file, and
QANIMATE_OUTPUTTYPE_STREAM outputs the animation to the standard out. *datafilename* is an optional parameter to specify the file name of output if you want to output the data to a file.

Example 1
For a geared fivebar linkage with parameters $r_1 = 7m, r_2 = 5m, r_3 = 10m, r_4 = 10m, r_5 = 2m$, $\theta_1 = 0$, $\lambda = -2.5$, $\phi = 35^o$, $\omega_2 = 5rad/sec$, and $\alpha_2 = 0$, simulate the motion of the fivebar linkage. Also trace the motion of the coupler point attached to link 3 with parameters $r_p = 5m$ and $\beta = 45^o$.

```
/**************************************************
 *  This example is to simulate the motion of the  *
 *  geared fivebar linkage.                        *
 **************************************************/

#include <fivebar.h>

int main()
{
   double r[1:5],
          theta_1[1:5], theta_2[1:5],
          rp, beta, phi, lambda;
   double omega2, alpha2;
   CGearedFivebar gearedbar;

   /* Setup geared fivebar linkage. */
   r[1] = 7;  r[2] = 5;  r[3] = 10;  r[4] = 10;  r[5] = 2;
   rp = 5;  beta = 45*(M_PI/180);
   phi = 35*M_PI/180;  lambda = -2.5;
   theta_1[1] = 10*(M_PI/180);  theta_2[1] = 10*(M_PI/180);
   theta_1[2] = 50*(M_PI/180);  theta_2[2] = 50*(M_PI/180);
   theta_1[5] = lambda*theta_1[2] + phi;
   theta_2[5] = lambda*theta_2[2] + phi;
   if(theta_1[5] < -M_PI)
   {
      theta_1[5] += 2*M_PI;
      theta_2[5] += 2*M_PI;
   }
```

```
   if(theta_1[5] > M_PI)
   {
      theta_1[5] -= 2*M_PI;
      theta_2[5] -= 2*M_PI;
   }

   /* Perform geared fivebar linkage analysis. */
   gearedbar.setLinks(r[1], r[2], r[3], r[4], r[5], theta_1[1]);
   gearedbar.setCouplerPoint(COUPLER_LINK3, rp, beta, TRACE_ON);
   gearedbar.setLambda(lambda);
   gearedbar.setPhi(phi);
   gearedbar.setNumPoints(50);
   gearedbar.animation(1);
   gearedbar.animation(2);

  return 0;
}
```

Output



CGearedFivebar::couplerCurve

Synopsis
#include <fivebar.h>
void couplerCurve(int *couplerLink*, **int** *branchnum,* **double** *curvex[:],* **double** *curvey[:]*);

Purpose
Calculate the coordinates of the coupler curve.

Parameters

couplerLink An *int* value specifying a macro to indicate which link the coupler is attached to.

branchnum An integer number used for the branch which will be calculated.

curvex[:] A double array used for the x coordinate of the coupler point with different input angle.

curvey[:] A double array used for the y coordinate of the coupler point with different input angle.

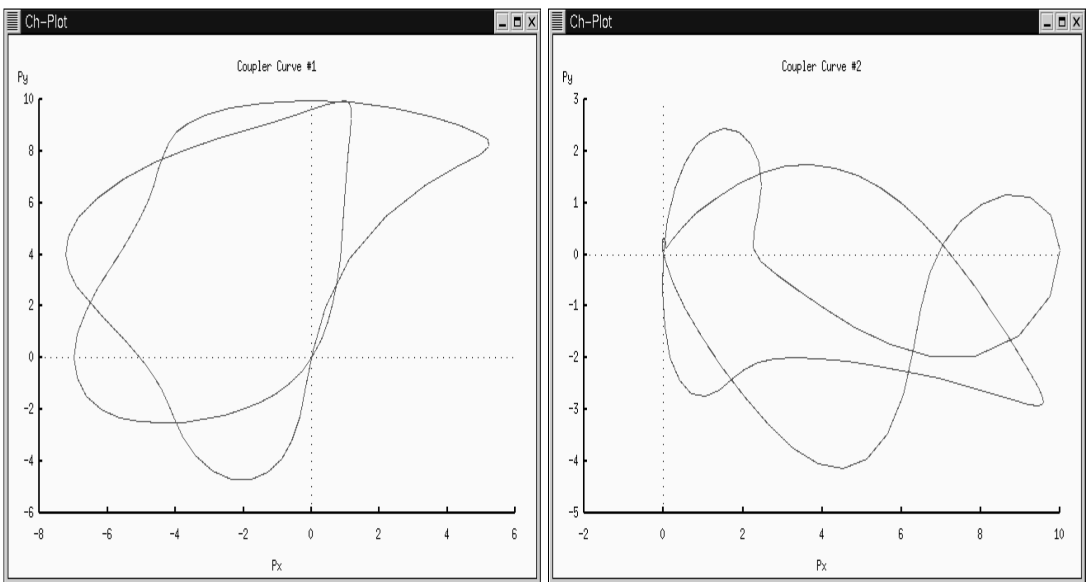
Return Value
None.

Description
This function calculates the coupler point position while looping the input angle. *curvex ,curvey* is the solution of the coupler point positon.

Example
For a geared fivebar linkage, given the lengths of the links and the fact that the coupler is attached to link 3, draw the position curve of the coupler point for each respective circuit.

```
/************************************************************
 *  This example is for determining the coupler curve of the  *
 *  geared fivebar linkage.                                   *
 ************************************************************/

#include <fivebar.h>

int main()
{
   double r[1:5],
          theta1,
          rp, beta, phi, lambda;
   int numpoint = 100;
   double RetCurvex_1[numpoint], RetCurvey_1[numpoint];
   double RetCurvex_2[numpoint], RetCurvey_2[numpoint];
   CGearedFivebar gearedbar;

   /* Setup geared fivebar linkage. */
   r[1] = 7;  r[2] = 5;  r[3] = 10;  r[4] = 10;  r[5] = 2;
   rp = 5;  beta = 45*(M_PI/180);
   phi = 35*M_PI/180;  lambda = -2.5;
   theta1 = 10*(M_PI/180);

   /* Perform geared fivebar linkage analysis. */
   gearedbar.setLinks(r[1], r[2], r[3], r[4], r[5], theta1);
   gearedbar.setCouplerPoint(COUPLER_LINK3, rp, beta);
   gearedbar.setLambda(lambda);
   gearedbar.setPhi(phi);
   gearedbar.setNumPoints(numpoint);
   gearedbar.couplerCurve(COUPLER_LINK3, 1, RetCurvex_1, RetCurvey_1);
   gearedbar.couplerCurve(COUPLER_LINK3, 2, RetCurvex_2, RetCurvey_2);

   /* Plot coupler curve. */
   plotxy(RetCurvex_1, RetCurvey_1, "Coupler Curve #1", "Px", "Py");
   plotxy(RetCurvex_2, RetCurvey_2, "Coupler Curve #2", "Px", "Py");

  return 0;
}
```

Output



See Also
CGearedFivebar::couplerPointPos().

CGearedFivebar::couplerPointAccel

Synopsis
#include <fivebar.h>
double complex couplerPointAccel(int *couplerLink*, **double** *theta[1:]*, **double** *omega[1:]*, **double** *alpha[1:]*);

Purpose
Calculate the acceleration of the coupler point.

Parameters

couplerLink An *int* value specifying a macro to indicate which link the coupler is attached to.

theta An array of *double* type used to store the angular position values of the various links.

oemga An array of *double* type used to store the angular velocity values of the various links.

alpha An array of *double* type used to store the angular acceleration values of the various links.

Return Value
This function returns the acceleration of the coupler point.

Description
This function calculates the acceleration of the coupler point. The return value is a complex number.

Example
For a geared fivebar linkage with properties $r_1 = 7m, r_2 = 5m, r_3 = 10m, r_4 = 10m, r_5 = 2m, r_p = 5m$, $\lambda = -2.5$, $\phi = 35^o$, $\beta = 20^o$, and $\theta_1 = 10^o$. Given the angle θ_2, the angular velocity ω_2 and the angular

acceleration α_2, calculate the acceleration of the coupler point for each circuit given that the coupler is attached to link 3.

```
/***************************************************
 *  This example is for calculating the acceleration  *
 *  of the coupler point of a geared fivebar          *
 *  linkage.                                          *
 ***************************************************/
#include <fivebar.h>

int main() {
    CGearedFivebar gearedfivebar;
    double r1 = 7, r2 = 5, r3 =10, r4 = 10, r5 =2.0, theta1 = 10*M_PI/180;
    double rp = 5, beta = 20*M_PI/180;
    double phi = 35*M_PI/180;
    double lambda = -2.5;
    double theta_1[1:5], theta_2[1:5];
    double omega_1[1:5], omega_2[1:5];
    double alpha_1[1:5], alpha_2[1:5];
    double theta2 = 70*M_PI/180;
    double complex Ap[1:2];

    omega_1[2]=5;   /*  rad/sec  */
    alpha_1[2]=-5;   /*  rad/sec*sec  */
    omega_2[2]=5;   /*  rad/sec  */
    alpha_2[2]=-5;   /*  rad/sec*sec  */

    theta_1[1] = theta1;
    theta_1[2] = theta2; // theta2
    theta_2[1] = theta1;
    theta_2[2] = theta2; // theta2

    gearedfivebar.setLinks(r1, r2, r3, r4, r5, theta1);
    gearedfivebar.setCouplerPoint(COUPLER_LINK3, rp, beta);
    gearedfivebar.setLambda(lambda);
    gearedfivebar.setPhi(phi);

    gearedfivebar.angularPos(theta_1, theta_2);
    gearedfivebar.angularVel(theta_1, omega_1);
    gearedfivebar.angularVel(theta_2, omega_2);
    gearedfivebar.angularAccel(theta_1, omega_1, alpha_1);
    gearedfivebar.angularAccel(theta_2, omega_2, alpha_2);
    Ap[1] = gearedfivebar.couplerPointAccel(COUPLER_LINK3, theta_1, omega_1,
                                            alpha_1);
    Ap[2] = gearedfivebar.couplerPointAccel(COUPLER_LINK3, theta_2, omega_2,
                                            alpha_2);
    printf("Circuit 1: \n  Coupler Acceleration: %.2f \n", Ap[1]);
    printf("Circuit 2: \n  Coupler Acceleration: %.2f \n", Ap[2]);

    return 0;
}
```

Output

```
Circuit 1:
  Coupler Acceleration: complex(-160.66,2.98)
Circuit 2:
  Coupler Acceleration: complex(267.71,-185.63)
```

CGearedFivebar::couplerPointPos

Synopsis
#include <fivebar.h>
void couplerPointPos(int *couplerLink*, **double** *theta2*, **double complex** &*p1,* **double complex** &*p2*);

Purpose
Calculate the position of the coupler point.

Parameters

couplerLink An *int* value specifying a macro to indicate which link the coupler is attached to.

theta2 A double number used for the input angle of link2.

p1 A double complex number for the first solution of the coupler point.

p2 A double complex number for the second solution of the coupler point. *x*.

Return Value
None.

Description
This function calculates the position of the coupler point. *theta2* is the input angle. *p1,p2* are the two solutions of the coupler point position, respectively, which is a complex number indicating the vector of the coupler point.

Example
A geared fivebar linkage has parameters $r_1 = 7m, r_2 = 5m, r_3 = 10m, r_4 = 10m, r_5 = 2m, r_p = 5m$, $\lambda = -2.5$, $\phi = 35^o$, $\beta = 20^o$, and $\theta_1 = 10^o$. Given the angle θ_2. Given that the coupler is attached to link 3, calculate the position of the coupler point for each circuit, respectively.

```
/******************************************************
 *  This example is for calculating the coupler point  *
 *  position of the geared fivebar linkage.            *
 ******************************************************/
#include <fivebar.h>

int main() {
    CGearedFivebar gearedfivebar;
    double r1 = 7, r2 = 5, r3 =10, r4 = 10, r5 =2.0, theta1 = 10*M_PI/180;
    double rp = 5, beta = 20*M_PI/180;
    double phi = 35*M_PI/180;
    double lambda = -2.5;
    double theta_1[1:5], theta_2[1:5];
    double theta2 = 70*M_PI/180;
    double complex P[1:2];

    theta_1[1] = theta1;
    theta_1[2] = theta2; // theta2
    theta_2[1] = theta1;
    theta_2[2] = theta2; // theta2
```

```
    gearedfivebar.setLinks(r1, r2, r3, r4, r5, theta1);
    gearedfivebar.setCouplerPoint(COUPLER_LINK3, rp, beta);
    gearedfivebar.setLambda(lambda);
    gearedfivebar.setPhi(phi);

    gearedfivebar.angularPos(theta_1, theta_2);
    gearedfivebar.couplerPointPos(COUPLER_LINK3, theta2, P[1], P[2]);

    printf("Circuit 1: \n  Coupler Position: %.2f \n", P[1]);
    printf("Circuit 2: \n  Coupler Position: %.2f \n", P[2]);

    return 0;
}
```

Output

```
Circuit 1:
  Coupler Position: complex(5.54,7.91)
Circuit 2:
  Coupler Position: complex(0.41,-0.13)
```

CGearedFivebar::couplerPointVel

Synopsis

#include <fivebar.h>
double complex couplerPointVel(int *couplerLink*, **double** *theta2,* **double** *theta3,* **double** *omega2,* **double** *omega3*);

Purpose

Calculate the velocity of the coupler point.

Parameters

couplerLink An *int* value specifying a macro to indicate which link the coupler is attached to.

theta An array of *double* type used to store the angular position values of the links.

omega An array of *double* type used to store the angular velocity values of the links.

Return Value

This function returns the vector of the coupler velocity.

Description

This function calculates the vector of the coupler velocity. The vector of the coupler point velocity is returned.

Example

A geared fivebar linkage has properties $r_1 = 7m, r_2 = 5m, r_3 = 10m, r_4 = 10m, r_5 = 2m, r_p = 5m$, $\lambda = -2.5$, $\phi = 35^o$, $\beta = 20$, and $\theta_1 = 10$. Given the angle θ_2 and the angular velocity ω_2, calculate the velocity of the coupler point for each circuit given that the coupler is attached to link 3.

```
/************************************************
 *  This example is for calculating the velocity  *
 *  of the coupler point of a geared fivebar      *
 *  linkage.                                      *
 ************************************************/
#include <fivebar.h>

int main() {
    CGearedFivebar gearedfivebar;
    double r1 = 7, r2 = 5, r3 =10, r4 = 10, r5 =2.0, theta1 = 10*M_PI/180;
    double rp = 5, beta = 20*M_PI/180;
    double phi = 35*M_PI/180;
    double lambda = -2.5;
    double theta_1[1:5], theta_2[1:5];
    double omega_1[1:5], omega_2[1:5];
    double theta2 = 70*M_PI/180;
    double complex Vp[1:2];

    omega_1[2]=5;   /*  rad/sec  */
    omega_2[2]=5;   /*  rad/sec  */

    theta_1[1] = theta1;
    theta_1[2] = theta2; // theta2
    theta_2[1] = theta1;
    theta_2[2] = theta2; // theta2

    gearedfivebar.setLinks(r1, r2, r3, r4, r5, theta1);
    gearedfivebar.setCouplerPoint(COUPLER_LINK3, rp, beta);
    gearedfivebar.setLambda(lambda);
    gearedfivebar.setPhi(phi);

    gearedfivebar.angularPos(theta_1, theta_2);
    gearedfivebar.angularVel(theta_1, omega_1);
    gearedfivebar.angularVel(theta_2, omega_2);
    Vp[1] = gearedfivebar.couplerPointVel(COUPLER_LINK3, theta_1, omega_1);
    Vp[2] = gearedfivebar.couplerPointVel(COUPLER_LINK3, theta_2, omega_2);
    printf("Circuit 1: \n  Coupler Velocity: %.2f \n", Vp[1]);
    printf("Circuit 2: \n  Coupler Velocity: %.2f \n", Vp[2]);

    return 0;
}
```

Output

```
Circuit 1:
  Coupler Velocity: complex(-30.75,17.21)
Circuit 2:
  Coupler Velocity: complex(-14.56,6.14)
```

CGearedFivebar::displayPosition

Synopsis

#include <fivebar.h>
int displayPosition(double *theta2*, **double** *theta3*, **double** *theta4*, ... /* [**int** *outputtype* [, [**char *** *filename*]] */);

Syntax
displayPosition(*theta2*, *theta3*, *theta4*)
displayPosition(*theta2*, *theta3*, *theta4*, *outputtype*)
displayPosition(*theta2*, *theta3*, *theta4*, *outputtype*, *filename*)

Purpose
Given θ_2, θ_3, and θ_4, display the current position of the geared fivebar linkage.

Parameters

theta2 θ_2 angle.

theta3 θ_3 angle.

theta4 θ_4 angle.

outputtype an optional argument for the output type.

filename an optional argument for the output file name.

Return Value
This function returns 0 on success and -1 on failure.

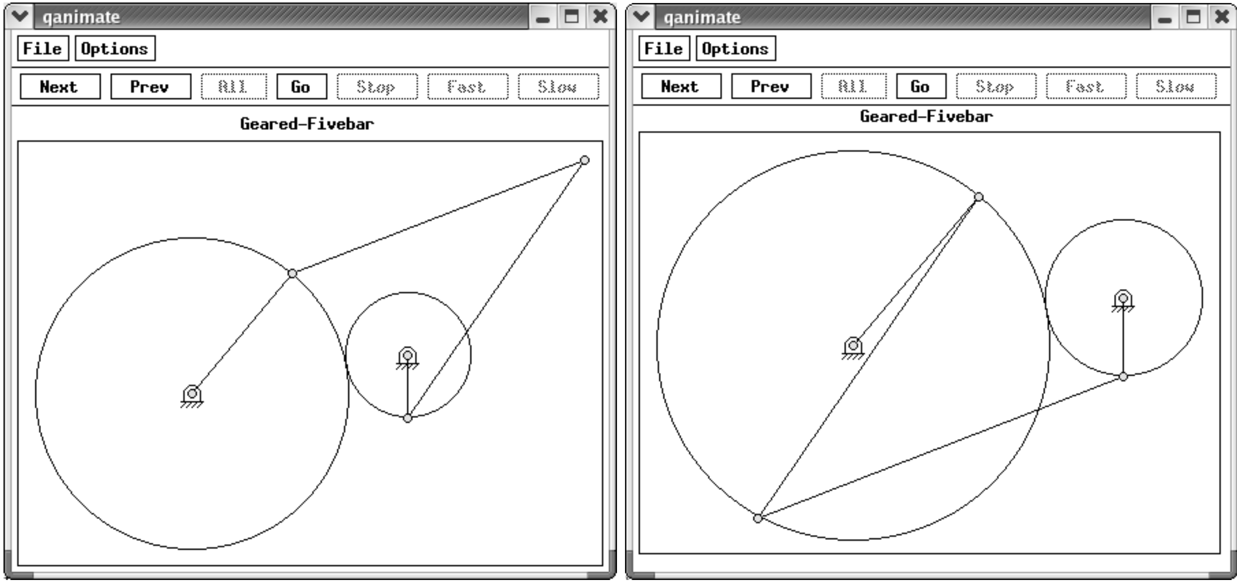
Description
Given θ_2, θ_3, and θ_4, display the current position of the geared fivebar linkage. `outputtype` is an optional parameter used to specify how the output should be handled. It may be one of the following macros: QANIMATE_OUTPUTTYPE_DISPLAY, QANIMATE_OUTPUTTYPE_FILE, QANIMATE_OUTPUTTYPE_STREAM. QANIMATE_OUTPUTTYPE_DISPLAY outputs the figure to the computer terminal. QANIMATE_OUTPUTTYPE_FILE writes the qanimate data onto a file. QANIMATE_OUTPUTTYPE_STREAM outputs the qanimate data to the standard out stream. `filename` is an optional parameter to specify the output file name.

Example
A geared-fivebar mechanism has link lengths $r_1 = 7m, r_2 = 5m, r_3 = 10m, r_4 = 10m, r_5 = 2m$, $\theta_1 = 10^o$, $\phi = 10^o$, and $\lambda = -2.5$. Given the angle θ_2, calculate the angular position θ_3 and θ_4 of link3 and link4, display the geared fivebar linkage in its current position.

```
/***********************************************************
 *  This example is for displaying the current position of   *
 *  the geared fivebar linkage.                              *
 ***********************************************************/

#include <fivebar.h>

int main()
{
   double r[1:5],
          theta_1[1:5], theta_2[1:5],
          phi, lambda;
   CGearedFivebar gearedbar;

   /* Setup geared fivebar linkage. */
```

```
    r[1] = 7;  r[2] = 5;  r[3] = 10;  r[4] = 10;  r[5] = 2;
    phi = 35*M_PI/180;  lambda = -2.5;
    theta_1[1] = 10*(M_PI/180);  theta_2[1] = 10*(M_PI/180);
    theta_1[2] = 50*(M_PI/180);  theta_2[2] = 50*(M_PI/180);
    theta_1[5] = lambda*theta_1[2] + phi;
    theta_2[5] = lambda*theta_2[2] + phi;
    if(theta_1[5] < -M_PI)
    {
       theta_1[5] += 2*M_PI;
       theta_2[5] += 2*M_PI;
    }
    if(theta_1[5] > M_PI)
    {
       theta_1[5] -= 2*M_PI;
       theta_2[5] -= 2*M_PI;
    }

    /* Perform geared fivebar linkage analysis. */
    gearedbar.setLinks(r[1], r[2], r[3], r[4], r[5], theta_1[1]);
    gearedbar.setLambda(lambda);
    gearedbar.setPhi(phi);
    gearedbar.angularPos(theta_1, theta_2);

    gearedbar.displayPosition(theta_1[2], theta_1[3], theta_1[4]);
    gearedbar.displayPosition(theta_2[2], theta_2[3], theta_2[4]);

   return 0;
}
```

Output



CGearedFivebar::plotCouplerCurve

Synopsis
#include <fivebar.h>
void plotCouplerCurve(CPlot **plot*, **int** *couplerLink*, **int** *branchnum*);

Syntax
plotCouplerCurve(*&plot*, *couplerLink*, *branchnum*)

Purpose
Plot the coupler curve.

Parameters

&plot A pointer to a CPlot class variable for formatting the plot of hte branch to be drawn.

couplerLink An *int* value specifying a macro to indicate which link the coupler is attached to.

branchnum An integer used for indicating the branch which you want to draw.

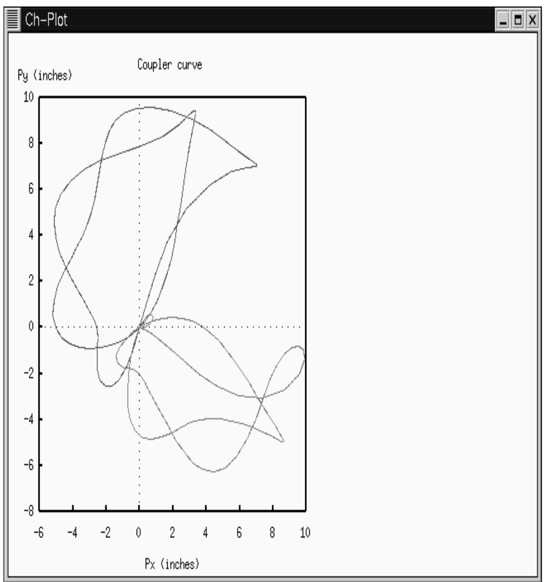
Return Value
None.

Description
This function plots the coupler curve. *&plot* is a pointer to a CPlot class variable for formatting the plot of the branch to be drawn. *couplerLink* is a macro specifying which link the coupler is attached to and *branchnum* is an integer number which indicates the branch you want to draw.

Example
For a geared fivebar linkage with parameters $r_1 = 7m, r_2 = 5m, r_3 = 10m, r_4 = 10m, r_5 = 2m, r_p = 5m$, $\lambda = -2.5$, $\phi = 35^o$, $\theta_1 = 0$, and $\beta = 20^o$, plot the position curve of the coupler point for each respective circuit given that the coupler is attached to link 3.

```
/**************************************************
 *  This example is for plotting the coupler curve  *
 *  of the geared fivebar linkage.                  *
 **************************************************/
#include <fivebar.h>

int main() {
    CGearedFivebar gearedfivebar;
    double r1 = 7, r2 = 5, r3 = 10, r4 = 10, r5 = 2;
    double theta1 = 10*M_PI/180;
    double rp = 5, beta = 20*M_PI/180;
    double phi = 35*M_PI/180;
    double lambda = -2.5;
    int numpoint =100;
    CPlot plot1, plot2;

    gearedfivebar.setLinks(r1,r2, r3, r4, r5, theta1);
    gearedfivebar.setCouplerPoint(COUPLER_LINK3, rp, beta);
    gearedfivebar.setNumPoints(numpoint);
    gearedfivebar.setLambda(lambda);
    gearedfivebar.setPhi(phi);
    gearedfivebar.plotCouplerCurve(&plot1, COUPLER_LINK3, 1);  //display a coupler curve
    gearedfivebar.plotCouplerCurve(&plot2, COUPLER_LINK3, 2);  //display a coupler curve }

    return 0;
}
```

Output



CGearedFivebar::setCouplerPoint

Synopsis
#include <fivebar.h>
void setCouplerPoint(**int** *couplerLink*, **double** *rp, beta*, ... [**int** *trace*] */);

Syntax
setCouplerPoint(*couplerLink*, *rp*, *beta*)
setCouplerPoint(*couplerLink*, *rp*, *beta*, *trace*)

Purpose
Set parameters for the coupler point.

Parameters

couplerLink An *int* value specifying a macro to indicate which link the coupler is attached to.

rp A double number used for the link length connected to the coupler point.

beta A double number specifying the angular position of the coupler point relative to the link it is attached to.

it trace An optional parameter of **int** type specifying either macro TRACE_OFF or TRACE_ON to indicate whether the coupler curve should be traced during animation.

Return Value
None.

Description
This function sets the parameters of the coupler point.

Example
see **CGearedFivebar**::**couplerPointAccel**().

CGearedFivebar::setAngularVel

Synopsis
#include <fivebar.h>
void setAngularVel(**double** *omega2*);

Purpose
Set the constant angular velocity of link2.

Parameters

omega2 A double number used for the constant input angular velocity of link2.

Return Value
None.

Description
This function sets the constant angular velocity of link2.

Example
see **CGearedFivebar**::**couplerPointAccel**().

CGearedFivebar::setLinks

Synopsis
#include <fivebar.h>
int setLinks(**double** *r1,* **double** *r2,* **double** *r3,* **double** *r4,* **double** *r5,* **double** *theta1*);

Purpose
Set the lengths of the links.

Parameters

r1,r2,r3,r4, r5 Double numbers used for the lengths of links.

theta1 A double number for the angle between link1 and the horizontal.

Return Value

Description
This function sets the lengths of links and angle θ_1.

Example
see **CGearedFivebar**::**plotCouplerCurve**().

CGearedFivebar::setNumPoints

Synopsis
#include <fivebar.h>
void setNumPoints(int *numpoints***);**

Purpose
Set the number of points for the animation and the plot of the coupler curve.

Parameters

numpoints An integer number used for the number of points.

Return Value
None.

Description
This function sets the number of points for the animation and the plot of the coupler curve.

Example
see **CGearedFivebar::animation**().

CGearedFivebar::uscUnit

Synopsis
#include <fivebar.h>
void uscUnit(bool *unit***);**

Purpose
Specify the use of SI or US Customary units in analysis.

Parameters

unit A boolean argument, where `true` indicates US Customary units are desired and `false` indicates SI units.

Return Value
None.

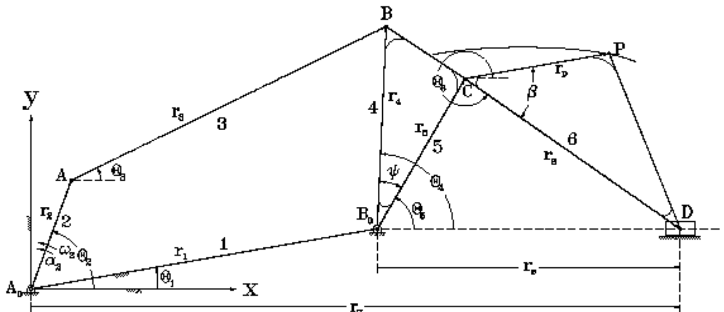
Description
This function specifies the whether SI or US Customary units are used.If `unit = true`, then US Customary units are used; otherwise, SI units are used. By default, SI units are assumed. This member function shall be called prior any other member function calls.

Appendix E

Class CFourbarSlider

CFourbarSlider

The header file **sixbar.h** includes header file **linkage.h**. The header file **sixbar.h** also contains a declaration of class **CFourbarSlider**. The **CFourbarSlider** class provides a means to analyze a fourbar-slider linkage within a Ch language environment.



Public Data
None.

Public Member Functions

Functions	Descriptions
angularAccel	Given the angular accelartion of link 2, calculate the angular acceleration of other links.
angularPos	Given the angle of link 2, calculate the angle of other links.
angularVel	Given the angular velocity of link 2, calculate the angular velocity of other links.
animation	Fourbar-slider linkage animation.
couplerPointAccel	Calculate the acceleration of the coupler point.
couplerPointPos	Calculate the position of the coupler point.
couplerPointVel	Calculate the velocity of the coupler point.
displayPosition	Display position of the fourbar-slider.
setCouplerPoint	Set parameters for the coupler point.

setAngularVel	Set constant angular velocity of linkage 2.
setLinks	Set lengths of links.
setNumPoints	Set number of points for animation.
sliderAccel	Calculate the linear acceleration of the slider.
sliderPos	Calculate the distance, r_7.
sliderVel	Calculate the linear velocity of the slider.
uscUnit	Specify the use of SI or US Customary units.

See Also

CFourbarSlider::angularAccel

Synopsis
#include <sixbar.h>
void angularAccel(double *theta[1:6],* **double** *omega[1:6],* **double** *alpha[1:6]*);

Purpose
Given the angular acceleration of link 2, calculate the angular acceleration of other links.

Parameters

theta An array of double data type with angles of links.

omega An array of double data type with angular velocities of links.

alpha An array of double data type with angular accelerations of links.

Return Value
None.

Description
Given the angular acceleration of link 2, this function calculates the angular acceleration of the remaining links of the fourbar-slider mechanism. *theta* is a one-dimensional array of size 6 which stores the angle of each link. *omega* is a one-dimensional array of size 6 which stores the angular velocity of each link. *alpha* is a one-dimensional array of size 6 which stores the angular acceleration of each link. The results of calculation are stored in this array.

Example
A fourbar-slider linkage has link lengths $r_1 = 12m, r_2 = 4m, r_3 = 12m, r_4 = 7m, r_5 = 6m, r_6 = 9m$, $\theta_1 = 10^o$, $\psi = 30^o$. Given the angle θ_2, the angular velocity ω_2 and the angular acceleration α_2, calculate the angular accelerations α_3, α_4, and α_6 of links 3, 4, and 6 for each circuit of the linkage.

```
/*****************************************************************
 *  This example is for calculating the angular acceleration of  *
 *  link3, link4, and link6.                                     *
 *****************************************************************/

#include<stdio.h>
```

```
#include<sixbar.h>

/******************************
 *        print_alpha()
 ******************************/
void print_alpha(double alp[1:6])
{
   printf("   deg/sec^2:\t alpha3 = %.3f,\t alpha4 = %.3f,\t alpha6 = %.3f\n",
          M_RAD2DEG(alp[3]), M_RAD2DEG(alp[4]), M_RAD2DEG(alp[6]));
   printf("   rad/sec^2:\t alpha3 = %.3f,\t alpha4 = %.3f,\t alpha6 = %.3f\n",
          alp[3], alp[4], alp[6]);
   printf("\n");
}


int main()
{
   double r[1:6], theta[1:4][1:6];
   double omega[1:4][1:6], alpha[1:4][1:6];
   double psi;
   int i;
   CFourbarSlider fslider;

   /* Default specification of the fourbar-slider linkage. */
   r[1] = 12.0;  r[2] = 4.0;  r[3] = 12.0;
   r[4] = 7.0;  r[5] = 6.0;  r[6] = 9.0;
   for(i = 1; i <= 4; i++)
   {
      theta[i][1] = M_DEG2RAD(10.0);
      theta[i][2] = M_DEG2RAD(70.0);
      omega[i][2] = M_DEG2RAD(10.0); // rad/sec
      alpha[i][2] = 0; // rad/sec^2
   }
   psi = M_DEG2RAD(30.0);

   /* Perform fourbar-slider linkage analysis. */
   fslider.setLinks(r, theta[1][1], psi);
   fslider.angularPos(theta[1], theta[2], theta[3], theta[4]);
   fslider.angularVel(theta[1], omega[1]);
   fslider.angularVel(theta[2], omega[2]);
   fslider.angularVel(theta[3], omega[3]);
   fslider.angularVel(theta[4], omega[4]);
   fslider.angularAccel(theta[1], omega[1], alpha[1]);
   fslider.angularAccel(theta[2], omega[2], alpha[2]);
   fslider.angularAccel(theta[3], omega[3], alpha[3]);
   fslider.angularAccel(theta[4], omega[4], alpha[4]);

   /* Display the results. */
   printf("Circuit 1: Angular Accelerations\n");
   print_alpha(alpha[1]);
   printf("Circuit 2: Angular Accelerations\n");
   print_alpha(alpha[2]);
   printf("Circuit 3: Angular Accelerations\n");
   print_alpha(alpha[3]);
   printf("Circuit 4: Angular Accelerations\n");
   print_alpha(alpha[4]);

   return 0;
}
```

Output

```
Circuit 1: Angular Accelerations
   deg/sec^2:  alpha3 = 0.410, alpha4 = 0.673, alpha6 = -0.096
   rad/sec^2:  alpha3 = 0.007, alpha4 = 0.012, alpha6 = -0.002

Circuit 2: Angular Accelerations
   deg/sec^2:  alpha3 = 0.410, alpha4 = 0.673, alpha6 = 0.096
   rad/sec^2:  alpha3 = 0.007, alpha4 = 0.012, alpha6 = 0.002

Circuit 3: Angular Accelerations
   deg/sec^2:  alpha3 = 1.071, alpha4 = 0.808, alpha6 = 0.206
   rad/sec^2:  alpha3 = 0.019, alpha4 = 0.014, alpha6 = 0.004

Circuit 4: Angular Accelerations
   deg/sec^2:  alpha3 = 1.071, alpha4 = 0.808, alpha6 = -0.206
   rad/sec^2:  alpha3 = 0.019, alpha4 = 0.014, alpha6 = -0.004
```

CFourbarSlider::angularPos

Synopsis
#include <sixbar.h>
void angularPos(double *theta_1[1:6],* **double** *theta_2[1:6],* **double** *theta_3[1:6],* **double** *theta_4[1:6]*);

Purpose
Given the angle of link 2, calculate the angles of the other links.

Parameters

theta_1 A double array with dimension size of 6 for the first solution.

theta_2 A double array with dimension size of 6 for the second solution.

theta_3 A double array with dimension size of 6 for the third solution.

theta_4 A double array with dimension size of 6 for the fourth solution.

Return Value
None.

Description
Given the angular position of link 2 of a fourbar-slider linkage, this function computes the angular positions of the remaining links. *theta_1* is a one-dimensional array of size 6 which stores the first solution of angular position. *theta_2* is a one-dimensional array of size 6 which stores the second solution of angular position. *theta_3* is a one-dimensional array of size 6 which stores the third solution of angular position. *theta_4* is a one-dimensional array of size 6 which stores the fourth solution of angular position.

Example
For a fourbar-slider linkage with link lengths $r_1 = 12m, r_2 = 4m, r_3 = 12m, r_4 = 7m, r_5 = 6m, r_6 = 9m$, $\theta_1 = 10^o$, $\psi = 30^o$. Given the angle θ_2, calculate the angular positions θ_3, θ_4, and θ_6 of links 3, 4, and 6 for

each circuit.

```
/************************************************************
 *  This example is for calculating the angular position of  *
 *  link3, link4, and link6.                                 *
 ************************************************************/

#include<stdio.h>
#include<sixbar.h>

/*******************************
 *       print_theta()
 *******************************/
void print_theta(double th[1:6])
{
   printf("   degrees:\t theta3 = %.3f,\t theta4 = %.3f,\t theta6 = %.3f\n",
          M_RAD2DEG(th[3]), M_RAD2DEG(th[4]), M_RAD2DEG(th[6]));
   printf("   radians:\t theta3 = %.3f,\t theta4 = %.3f,\t theta6 = %.3f\n",
          th[3], th[4], th[6]);
   printf("\n");
}


int main()
{
   double r[1:6], theta[1:4][1:6];
   double complex P[1:4];
   double psi;
   int i;
   CFourbarSlider fslider;

   /* Default specification of the fourbar-slider linkage. */
   r[1] = 12;  r[2] = 4;  r[3] = 12;
   r[4] = 7;  r[5] = 6;  r[6] = 9;
   for(i = 1; i <= 4; i++)
   {
      theta[i][1] = M_DEG2RAD(10.0);
      theta[i][2] = M_DEG2RAD(70.0);
   }
   psi = M_DEG2RAD(30.0);

   /* Perform fourbar-slider linkage analysis. */
   fslider.setLinks(r, theta[1][1], psi);
   fslider.angularPos(theta[1], theta[2], theta[3], theta[4]);

   /* Display the results. */
   printf("Circuit 1: Angular Positions\n");
   print_theta(theta[1]);
   printf("Circuit 2: Angular Positions\n");
   print_theta(theta[2]);
   printf("Circuit 3: Angular Positions\n");
   print_theta(theta[3]);
   printf("Circuit 4: Angular Positions\n");
   print_theta(theta[4]);

   return 0;
}
```

Output

```
Circuit 1: Angular Positions
   degrees:  theta3 = 26.307, theta4 = 87.482, theta6 = -34.204
   radians:  theta3 = 0.459, theta4 = 1.527, theta6 = -0.597

Circuit 2: Angular Positions
   degrees:  theta3 = 26.307, theta4 = 87.482, theta6 = -145.796
   radians:  theta3 = 0.459, theta4 = 1.527, theta6 = -2.545

Circuit 3: Angular Positions
   degrees:  theta3 = -44.521, theta4 = -105.695, theta6 = 27.752
   radians:  theta3 = -0.777, theta4 = -1.845, theta6 = 0.484

Circuit 4: Angular Positions
   degrees:  theta3 = -44.521, theta4 = -105.695, theta6 = 152.248
   radians:  theta3 = -0.777, theta4 = -1.845, theta6 = 2.657
```

CFourbarSlider::angularVel

Synopsis
#include <sixbar.h>
void angularVel(double *theta[1:6],* **double** *omega[1:6]*);

Purpose
Given the angular velocity of link 2, calculate the angular velocity of the other links.

Parameters

theta A double array of size 6 used for the input angle of links.

omega A double array of size 6 used for the angular velocities of links.

Return Value
None.

Description
Given the angular velocity of link 2, this function calculates the angular velocities of the remaining links of the fourbar-slider linkage. *theta* is an array for the angular positions of the six links. *omega* is an array for the links' angular velocity values. The results of the calculation are stored in array *omega*.

Example
For a fourbar-slider linkage with link lengths $r_1 = 12m, r_2 = 4m, r_3 = 12m, r_4 = 7m, r_5 = 6m, r_6 = 9m$, $\theta_1 = 10^o$, $\psi = 30^o$. Given the angle θ_2 and then angular velocity ω_2, calculate the angular velocities ω_3, ω_4, and ω_6 of links 3, 4, and 6 for each circuit.

```
/************************************************************
 *  This example is for calculating the angular velocity of  *
 *  link3, link4, and link6.                                 *
 ************************************************************/
```

```
#include<stdio.h>
#include<sixbar.h>

/*******************************
 *        print_omega()
 *******************************/
void print_omega(double om[1:6])
{
   printf("   deg/sec:\t omega3 = %.3f,\t omega4 = %.3f,\t omega6 = %.3f\n",
          M_RAD2DEG(om[3]), M_RAD2DEG(om[4]), M_RAD2DEG(om[6]));
   printf("   rad/sec:\t omega3 = %.3f,\t omega4 = %.3f,\t omega6 = %.3f\n",
          om[3], om[4], om[6]);
   printf("\n");
}


int main()
{
   double r[1:6], theta[1:4][1:6];
   double omega[1:4][1:6];
   double psi;
   int i;
   CFourbarSlider fslider;

   /* Default specification of the fourbar-slider linkage. */
   r[1] = 12;  r[2] = 4;  r[3] = 12;
   r[4] = 7;  r[5] = 6;  r[6] = 9;
   for(i = 1; i <= 4; i++)
   {
      theta[i][1] = M_DEG2RAD(10.0);
      theta[i][2] = M_DEG2RAD(70.0);
      omega[i][2] = M_DEG2RAD(10.0); // rad/sec
   }
   psi = M_DEG2RAD(30.0);

   /* Perform fourbar-slider linkage analysis. */
   fslider.setLinks(r, theta[1][1], psi);
   fslider.angularPos(theta[1], theta[2], theta[3], theta[4]);
   fslider.angularVel(theta[1], omega[1]);
   fslider.angularVel(theta[2], omega[2]);
   fslider.angularVel(theta[3], omega[3]);
   fslider.angularVel(theta[4], omega[4]);

   /* Display the results. */
   printf("Circuit 1: Angular Velocities\n");
   print_omega(omega[1]);
   printf("Circuit 2: Angular Velocities\n");
   print_omega(omega[2]);
   printf("Circuit 3: Angular Velocities\n");
   print_omega(omega[3]);
   printf("Circuit 4: Angular Velocities\n");
   print_omega(omega[4]);

   return 0;
}
```

Output

```
Circuit 1: Angular Velocities
```

```
   deg/sec:  omega3 = -1.143, omega4 = 4.506, omega6 = -1.952
   rad/sec:  omega3 = -0.020, omega4 = 0.079, omega6 = -0.034

Circuit 2: Angular Velocities
   deg/sec:  omega3 = -1.143, omega4 = 4.506, omega6 = 1.952
   rad/sec:  omega3 = -0.020, omega4 = 0.079, omega6 = 0.034

Circuit 3: Angular Velocities
   deg/sec:  omega3 = -0.286, omega4 = -5.934, omega6 = -3.199
   rad/sec:  omega3 = -0.005, omega4 = -0.104, omega6 = -0.056

Circuit 4: Angular Velocities
   deg/sec:  omega3 = -0.286, omega4 = -5.934, omega6 = 3.199
   rad/sec:  omega3 = -0.005, omega4 = -0.104, omega6 = 0.056
```

CFourbarSlider::animation

Synopsis
#include <sixbar.h>
void animation(int *branchnum*, ... /* [**int** *animationtype*, **string_t** *datafilename*] */);

Purpose
An animation of a fourbar-slider mechanism.

Parameters

branchnum an integer used to indicate which branch will be drawn.

it animationtype an optional parameter to specify the output type of the animation.

datafilename an optional parameter to specify the output file name.

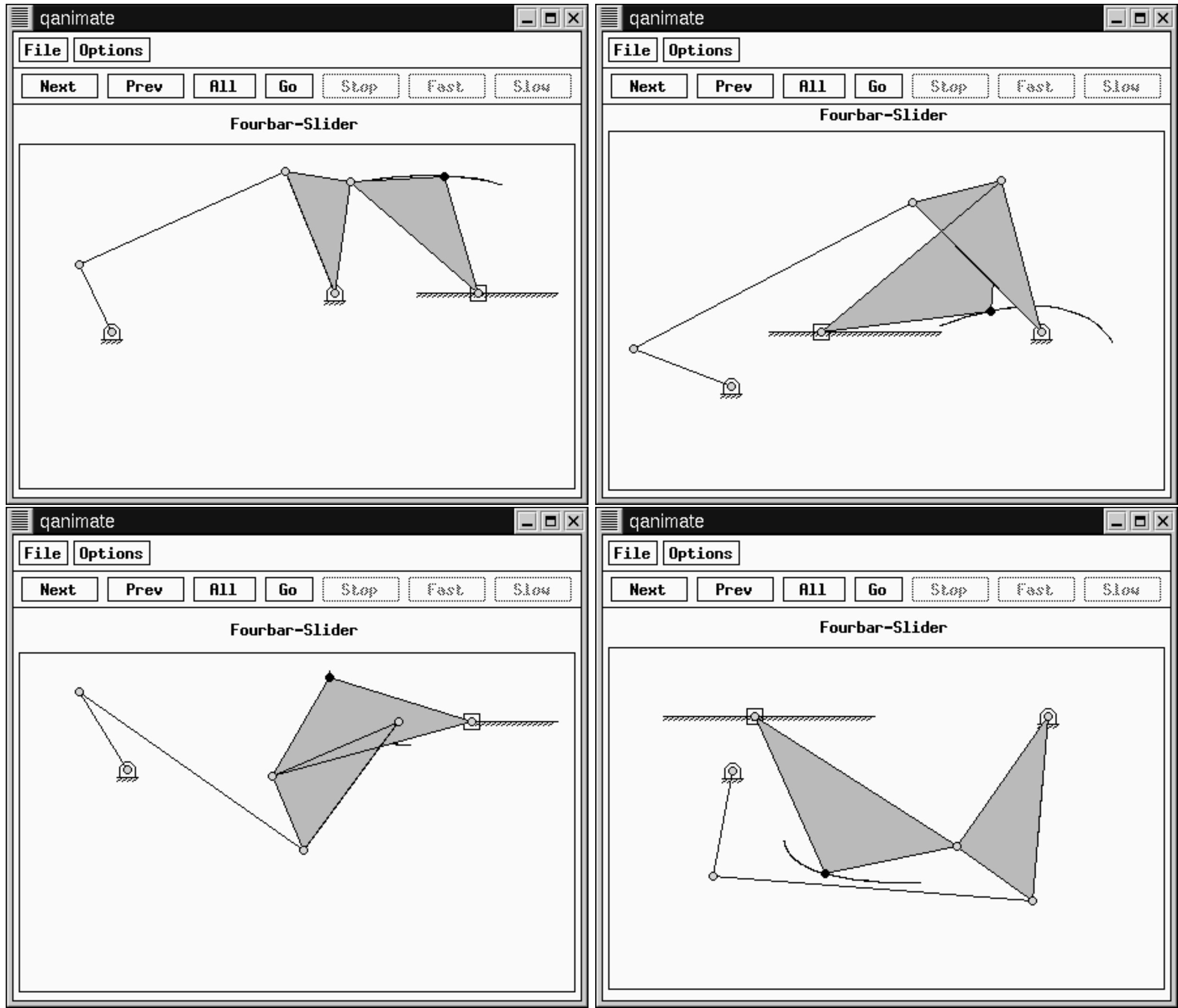
Return Value
None.

Description
This function simulates the motion of a fourbar-slider mechanism. *branchnum* is an integer number which indicates the branch to be drawn. *animationtype* is an optional parameter used to specify how the animation should be outputted. *animationtype* can be either of the following macros:
QANIMATE_OUTPUTTYPE_DISPLAY, QANIMATE_OUTPUTTYPE_FILE,
QANIMATE_OUTPUTTYPE_STREAM. QANIMATE_OUTPUTTYPE_DISPLAY display an animation on the screen. QANIMATE_OUTPUTTYPE_FILE writes the animation data onto a file.
QANIMATE_OUTPUTTYPE_STREAM outputs the animation to the standard out. *datafilename* is an optional parameter to specify the output file name.

Example
For a fourbar-slider linkage with link lengths $r_1 = 12m, r_2 = 4m, r_3 = 12m, r_4 = 7m, r_5 = 6m, r_6 = 9m$, $\theta_1 = 10^o$, $\psi = 30^o$, $\omega_2 = 5rad/sec$, $\alpha_2 = 0$, simulate the motion of the mechanism with a coupler attached to link 6 with parameters $r_p = 5$ and $\beta = 45^o$.

```
/*************************************************
 *  This example is for simulating the motion of a  *
 *  fourbar-slider linkage.                         *
 *************************************************/

#include<stdio.h>
#include<sixbar.h>

int main()
{
   double r[1:6], theta[1:4][1:6];
   double rp, beta;
   double omega2, alpha2;
   double complex P[1:4];
   double psi;
   int numpoints = 50;
   int i;
   CFourbarSlider fslider;

   /* Default specification of the fourbar-slider linkage. */
   r[1] = 12;  r[2] = 4;  r[3] = 12;
   r[4] = 7;  r[5] = 6;  r[6] = 9;
   for(i = 1; i <= 4; i++)
   {
      theta[i][1] = M_DEG2RAD(10.0);
      theta[i][2] = M_DEG2RAD(70.0);
   }
   psi = M_DEG2RAD(30.0);
   rp = 5.0;  beta = M_DEG2RAD(45.0);
   omega2 = 5.0; // rad/sec
   alpha2 = 0.0; // rad/sec^2

   /* Perform fourbar-slider linkage analysis. */
   fslider.setLinks(r, theta[1][1], psi);
   fslider.setCouplerPoint(COUPLER_LINK6, rp, beta, TRACE_ON);
   fslider.setNumPoints(numpoints);
   fslider.animation(1);
   fslider.animation(2);
   fslider.animation(3);
   fslider.animation(4);

   return 0;
}
```

Output



CFourbarSlider::couplerPointAccel

Synopsis
#include <sixbar.h>
double complex couplerPointAccel(int *couplerLink*, **double** *theta[1:6],* **double** *omega[1:6],* **double** *alpha[1:6]*);

Purpose
Calculate the acceleration of the coupler point.

Parameters

couplerLink An *int* value specifying a macro to indicate which link the coupler is attached to.

theta A double array used to store the angular positions of the links.

omega A double array used to store the angular velocities of the links.

alpha A double array used to store the angular accelerations of the links.

Return Value
This function returns the acceleration of the the coupler point.

Description
This function calculates the acceleration of the coupler point. The return value is a complex number.

Example
A fourbar-slider mechanism has properties $r_1 = 12m, r_2 = 4m, r_3 = 12m, r_4 = 7m, r_5 = 6m, r_6 = 9m, r_p = 5$, $\beta = 45^o$, $\theta_1 = 10^o$, $\psi = 30^o$. Given the angle θ_2, the angular velocity ω_2 and the angular acceleration α_2, calculate the acceleration of the coupler point for different circuits. Note that the coupler point is attached to link 6.

```
/*************************************************************
 *  This example is for calculating the acceleration of the  *
 *  coupler point of a fourbar-slider linkage.               *
 *************************************************************/

#include<stdio.h>
#include<sixbar.h>


int main()
{
   double r[1:6], theta[1:4][1:6];
   double omega[1:4][1:6], alpha[1:4][1:6];
   double rp, beta;
   double psi;
   double complex Ap[1:4];
   int i;
   CFourbarSlider fslider;

   /* Default specification of the fourbar-slider linkage. */
   r[1] = 12;  r[2] = 4;  r[3] = 12;
   r[4] = 7;  r[5] = 6;  r[6] = 9;
   for(i = 1; i <= 4; i++)
   {
      theta[i][1] = M_DEG2RAD(10.0);
      theta[i][2] = M_DEG2RAD(70.0);
      omega[i][2] = M_DEG2RAD(10.0); // rad/sec
      alpha[i][2] = 0; // rad/sec^2
   }
   psi = M_DEG2RAD(30.0);
   rp = 5.0;  beta = M_DEG2RAD(45.0);

   /* Perform fourbar-slider linkage analysis. */
   fslider.setLinks(r, theta[1][1], psi);
   fslider.setCouplerPoint(COUPLER_LINK6, rp, beta);
   fslider.angularPos(theta[1], theta[2], theta[3], theta[4]);
   fslider.angularVel(theta[1], omega[1]);
   fslider.angularVel(theta[2], omega[2]);
   fslider.angularVel(theta[3], omega[3]);
   fslider.angularVel(theta[4], omega[4]);
   fslider.angularAccel(theta[1], omega[1], alpha[1]);
```

```
    fslider.angularAccel(theta[2], omega[2], alpha[2]);
    fslider.angularAccel(theta[3], omega[3], alpha[3]);
    fslider.angularAccel(theta[4], omega[4], alpha[4]);
    Ap[1] = fslider.couplerPointAccel(COUPLER_LINK6, theta[1], omega[1], alpha[1]);
    Ap[2] = fslider.couplerPointAccel(COUPLER_LINK6, theta[2], omega[2], alpha[2]);
    Ap[3] = fslider.couplerPointAccel(COUPLER_LINK6, theta[3], omega[3], alpha[3]);
    Ap[4] = fslider.couplerPointAccel(COUPLER_LINK6, theta[4], omega[4], alpha[4]);

    /* Display the results. */
    printf("Circuit 1: Coupler Point Acceleration\n");
    printf("   Ap = %.3f\n\n", Ap[1]);
    printf("Circuit 2: Coupler Point Acceleration\n");
    printf("   Ap = %.3f\n\n", Ap[2]);
    printf("Circuit 3: Coupler Point Acceleration\n");
    printf("   Ap = %.3f\n\n", Ap[3]);
    printf("Circuit 4: Coupler Point Acceleration\n");
    printf("   Ap = %.3f\n\n", Ap[4]);

    return 0;
}
```

Output

```
Circuit 1: Coupler Point Acceleration
   Ap = complex(-0.083,-0.003)

Circuit 2: Coupler Point Acceleration
   Ap = complex(-0.070,0.011)

Circuit 3: Coupler Point Acceleration
   Ap = complex(0.083,-0.025)

Circuit 4: Coupler Point Acceleration
   Ap = complex(0.115,0.006)
```

CFourbarSlider::couplerPointPos

Synopsis

#include <sixbar.h>
void couplerPointPos(int *couplerLink*, **double** *theta2*, **double complex** *p[1:4]*);

Purpose

Calculate the position of the coupler point.

Parameters

couplerLink An *int* value specifying a macro to indicate which link the coupler is attached to.

theta2 A double number used for the input angle of link.

p A double complex array of size 4 to store the different solutions of the coupler point position.

Return Value

None.

Description

This function calculates the position of the coupler point. *theta2* is the input angle. Array *p* contains the four possible solutions of the coupler point position. The results of the calculation are stored in this array.

Example

A fourbar-slider mechanism has properties $r_1 = 12m, r_2 = 4m, r_3 = 12m, r_4 = 7m, r_5 = 6m, r_6 = 9m, r_p = 5m$, $\beta = 45^o$, $\theta_1 = 10^o$, $\psi = 30^o$. Given the angle θ_2 and the fact that the coupler is attached to link 6, calculate the coupler point position of the different circuits.

```
/*********************************************************
 *  This example is for calculating the position of the  *
 *  coupler point of a fourbar-slider linkage.           *
 *********************************************************/

#include<stdio.h>
#include<sixbar.h>


int main()
{
   double r[1:6], theta[1:4][1:6];
   double rp, beta;
   double psi;
   double complex P[1:4];
   int i;
   CFourbarSlider fslider;

   /* Default specification of the fourbar-slider linkage. */
   r[1] = 12.0;  r[2] = 4.0;  r[3] = 12.0;
   r[4] = 7.0;  r[5] = 6.0;  r[6] = 9.0;
   for(i = 1; i <= 4; i++)
   {
      theta[i][1] = M_DEG2RAD(10.0);
      theta[i][2] = M_DEG2RAD(70.0);
   }
   psi = M_DEG2RAD(30.0);
   rp = 5.0;  beta = M_DEG2RAD(45.0);

   /* Perform fourbar-slider linkage analysis. */
   fslider.setLinks(r, theta[1][1], psi);
   fslider.setCouplerPoint(COUPLER_LINK6, rp, beta);
   fslider.angularPos(theta[1], theta[2], theta[3], theta[4]);
   fslider.couplerPointPos(COUPLER_LINK6, theta[1][2], P);

   /* Display the results. */
   printf("Circuit 1: Coupler Point Position\n");
   printf("   P = %.3f\n\n", P[1]);
   printf("Circuit 2: Coupler Point Position\n");
   printf("   P = %.3f\n\n", P[2]);
   printf("Circuit 3: Coupler Point Position\n");
   printf("   P = %.3f\n\n", P[3]);
   printf("Circuit 4: Coupler Point Position\n");
   printf("   P = %.3f\n\n", P[4]);
```

```
   return 0;
}
```

Output

```
Circuit 1: Coupler Point Position
   P = complex(19.955,8.080)

Circuit 2: Coupler Point Position
   P = complex(14.107,2.232)

Circuit 3: Coupler Point Position
   P = complex(9.006,2.668)

Circuit 4: Coupler Point Position
   P = complex(2.749,-3.590)
```

CFourbarSlider::couplerPointVel

Synopsis
#include <sixbar.h>
double complex couplerPointVel(int *couplerLink*, **double** *theta[1:6],* **double** *omega[1:6]*);

Purpose
Calculate the velocity of the coupler point.

Parameters

theta A double array of size 6 used to store the angular positions of the links.

omega A double array of size 6 used to store the angular velocities of the links.

Return Value
This function returns the velocity of the coupler point in vector form.

Description
This function calculates the velocity of the coupler point. The value is returned as of type `double complex`.

Example
A fourbar-slider mechanism has properties $r_1 = 12m, r_2 = 4m, r_3 = 12m, r_4 = 7m, r_5 = 6m, r_6 = 9m, r_p = 5m$, $\beta = 45^o$, $\theta_1 = 10^o$, $\psi = 30^o$. Given the angle θ_2 and the angular velocity ω_2, calculate the velocity of the coupler point for the different circuits. The coupler is attached to link 6.

```
/*********************************************************
 *  This example is for calculating the velocity of the  *
 *  coupler point of a fourbar-slider linkage.           *
 *********************************************************/

#include<stdio.h>
#include<sixbar.h>
```

```
int main()
{
   double r[1:6], theta[1:4][1:6];
   double omega[1:4][1:6];
   double rp, beta;
   double psi;
   double complex Vp[1:4];
   int i;
   CFourbarSlider fslider;

   /* Default specification of the fourbar-slider linkage. */
   r[1] = 12;  r[2] = 4;  r[3] = 12;
   r[4] = 7;  r[5] = 6;  r[6] = 9;
   for(i = 1; i <= 4; i++)
   {
      theta[i][1] = M_DEG2RAD(10.0);
      theta[i][2] = M_DEG2RAD(70.0);
      omega[i][2] = M_DEG2RAD(10.0); // rad/sec
   }
   psi = M_DEG2RAD(30.0);
   rp = 5.0;  beta = M_DEG2RAD(45.0);

   /* Perform fourbar-slider linkage analysis. */
   fslider.setLinks(r, theta[1][1], psi);
   fslider.setCouplerPoint(COUPLER_LINK6, rp, beta);
   fslider.angularPos(theta[1], theta[2], theta[3], theta[4]);
   fslider.angularVel(theta[1], omega[1]);
   fslider.angularVel(theta[2], omega[2]);
   fslider.angularVel(theta[3], omega[3]);
   fslider.angularVel(theta[4], omega[4]);
   Vp[1] = fslider.couplerPointVel(COUPLER_LINK6, theta[1], omega[1]);
   Vp[2] = fslider.couplerPointVel(COUPLER_LINK6, theta[2], omega[2]);
   Vp[3] = fslider.couplerPointVel(COUPLER_LINK6, theta[3], omega[3]);
   Vp[4] = fslider.couplerPointVel(COUPLER_LINK6, theta[4], omega[4]);

   /* Display the results. */
   printf("Circuit 1: Coupler Point Velocity\n");
   printf("   Vp = %.3f\n\n", Vp[1]);
   printf("Circuit 2: Coupler Point Velocity\n");
   printf("   Vp = %.3f\n\n", Vp[2]);
   printf("Circuit 3: Coupler Point Velocity\n");
   printf("   Vp = %.3f\n\n", Vp[3]);
   printf("Circuit 4: Coupler Point Velocity\n");
   printf("   Vp = %.3f\n\n", Vp[4]);

   return 0;
}
```

Output

```
Circuit 1: Coupler Point Velocity
   Vp = complex(-0.366,0.086)

Circuit 2: Coupler Point Velocity
   Vp = complex(-0.230,0.222)

Circuit 3: Coupler Point Velocity
```

```
   Vp = complex(-0.167,0.362)

Circuit 4: Coupler Point Velocity
   Vp = complex(-0.351,0.178)
```

CFourbarSlider::displayPosition

Synopsis
#include <sixbar.h>
int displayPosition(double *theta2*, **double** *theta3*, **double** *theta4*, **double** *theta6*, ... /* [**int** *outputtype* [, [**char** * *filename*]] */);

Syntax
displayPosition(*theta2*, *theta3*, *theta4*, *theta6*)
displayPosition(*theta2*, *theta3*, *theta4*, *theta6*, *outputtype*)
displayPosition(*theta2*, *theta3*, *theta4*, *theta6*, *outputtype*, *filename*)

Purpose
Given θ_2, θ_3, θ_4, and θ_6, display the current position of the fourbar-slider linkage.

Parameters

theta2 θ_2 angle.

theta3 θ_3 angle.

theta4 θ_4 angle.

theta6 θ_6 angle.

outputtype an optional argument for the output type.

filename an optional argument for the output file name.

Return Value
This function returns 0 on success and -1 on failure.

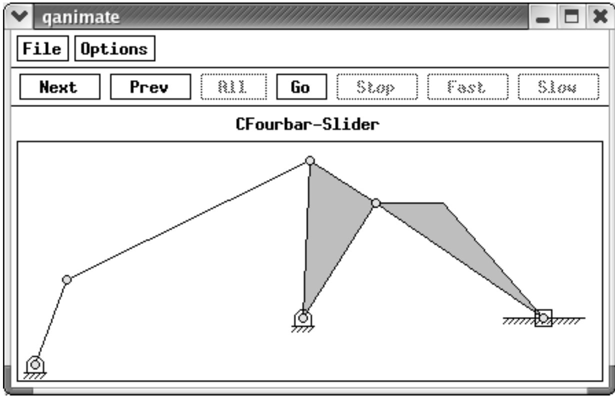
Description
Given θ_2, θ_3, θ_4, and θ_6 display the current position of the fourbar-slider linkage. `outputtype` is an optional parameter used to specify how the output should be handled. It may be one of the following macros: QANIMATE_OUTPUTTYPE_DISPLAY, QANIMATE_OUTPUTTYPE_FILE, QANIMATE_OUTPUTTYPE_STREAM. QANIMATE_OUTPUTTYPE_DISPLAY outputs the figure to the computer terminal. QANIMATE_OUTPUTTYPE_FILE writes the qanimate data onto a file. QANIMATE_OUTPUTTYPE_STREAM outputs the qanimate data to the standard out stream. `filename` is an optional parameter to specify the output file name.

Example
A fourbar linkage has link lengths $r_1 = 12m, r_2 = 4m, r_3 = 12m, r_4 = 7m, r_5 = 6m, r_6 = 9m$, $\theta_1 = 10^o$, and coupler properties $r_p = 3$ and $\beta = 35^o$ attached to link 6. Given the angle θ_2, display the fourbarslider

linkage in its current position for the first branch.

```
/************************************************************
 *  This example is for display the position of the         *
 *  fourbar-slider mechanism.                               *
 ************************************************************/

#include<stdio.h>
#include<sixbar.h>

int main()
{
   double r[1:6], theta[1:4][1:6];
   double rp6, beta6;
   double complex P[1:4];
   double psi;
   int i;
   CFourbarSlider fslider;

   /* Default specification of the fourbar-slider linkage. */
   r[1] = 12;  r[2] = 4;  r[3] = 12;
   r[4] = 7;  r[5] = 6;  r[6] = 9;
   rp6 = 3;
   for(i = 1; i <= 4; i++)
   {
      theta[i][1] = M_DEG2RAD(10.0);
      theta[i][2] = M_DEG2RAD(70.0);
      beta6 = M_DEG2RAD(35.0);
   }
   psi = M_DEG2RAD(30.0);

   /* Perform fourbar-slider linkage analysis. */
   fslider.setLinks(r, theta[1][1], psi);
   fslider.setCouplerPoint(COUPLER_LINK6, rp6, beta6);
   fslider.angularPos(theta[1], theta[2], theta[3], theta[4]);
   fslider.displayPosition(theta[1][2], theta[1][3], theta[1][4], theta[1][6]);

   return 0;
}
```

Output



CFourbarSlider::setCouplerPoint

Synopsis
#include <sixbar.h>
void setCouplerPoint(int *couplerLink*, **double** *rp*, **double** *beta*, ... /* [**int** *trace*] */);

Syntax
setCouplerPoint(*couplerLink*, *rp*, *beta*)
setCouplerPoint(*couplerLink*, *rp*, *beta*, *trace*)

Purpose
Set parameters for the coupler point.

Parameters

couplerLink An *int* value specifying a macro to indicate which link the coupler is attached to.

rp A double number used for the link length connected to the coupler point.

beta A double number specifying the angular position of the coupler point relative to the link it is attached to.

it trace An optional parameter of **int** type specifying either macro TRACE_OFF or TRACE_ON to indicate whether the coupler curve should be traced during animation.

Return Value
None.

Description
This function sets the parameters of the coupler point.

Example
See **CFourbarSlider::couplerPointPos()**.

CFourbarSlider::setAngularVel

Synopsis
#include <sixbar.h>
void setAngularVel(double *omega2*);

Purpose
Set the constant angular velocity of link 2.

Parameters

omega2 A double number used for the constant input angular velocity of link 2.

Description
This function sets the constant angular velocity of link 2.

Example
see **CFourbarSlider::angularVel()**.

CFourbarSlider::setLinks

Synopsis
#include <sixbar.h>
void setLinks(double *r[1:6],* **double** *theta1,* **double** *psi*);

Purpose
Setup the links of the fourbar-slider mechanism.

Parameters

r A double array of size 6 used for the length of the links.

theta1 A double number for the angle between link 1 and the horizontal.

psi A double number for the included angle between links 4 and 5.

Return Value
None.

Description
This function sets the links of the fourbar-slider mechanism as well as angles θ_1 and ψ.

Example
See **CFourbarSlider::angularPos()**.

CFourbarSlider::setNumPoints

Synopsis
#include <sixbar.h>
void setNumPoints(int *numpoints*);

Purpose
Set the number of points for animation.

Parameters

numpoints An integer number used for the number of points.

Return Value
None.

Description
This function sets the number of points for animation of the fourbar-slider mechanism.

Example
See **CFourbarSlider::animation()**.

CFourbarSlider::sliderAccel

Synopsis
#include <sixbar.h>
double sliderAccel(double *theta[1:6],* **double** *omega[1:6],* **double** *alpha[1:6]***);**

Purpose
Calculates the acceleration of the slider.

Parameters

theta A double array of size 6 used to store the angular positions of the links.

omega A double array of size 6 used to store the angular velocities of the links.

alpha A double array of size 6 used to store the angular acceleration of the links.

Return Value
This function returns the linear acceleration of the slider.

Description
This function calculates the linear acceleration of the slider parallel to the horizontal. The return value is of `double` type.

Example
A fourbar-slider mechanism has the following properties: $r_1 = 12m, r_2 = 4m, r_3 = 12m, r_4 = 7m, r_5 = 6m, r_6 = 9m$, $\theta_1 = 10^o$, $\psi = 30^o$. Given the angle θ_2, the angular velocity ω_2 and the angular acceleration α_2, calculate the linear acceleration of the slider, $\ddot{r}_7$.

```
/*******************************************************
 *  This example is for calculating the acceleration of  *
 *  the slider.                                           *
 *******************************************************/

#include<stdio.h>
#include<sixbar.h>


int main()
{
   double r[1:6], theta[1:4][1:6];
   double omega[1:4][1:6], alpha[1:4][1:6];
   double r7ddot[1:4];
   double psi;
   int i;
   CFourbarSlider fslider;

   /* Default specification of the fourbar-slider linkage. */
   r[1] = 12;  r[2] = 4;  r[3] = 12;
   r[4] = 7;  r[5] = 6;  r[6] = 9;
   for(i = 1; i <= 4; i++)
   {
      theta[i][1] = M_DEG2RAD(10.0);
```

```
        theta[i][2] = M_DEG2RAD(70.0);
        omega[i][2] = M_DEG2RAD(10.0); // rad/sec
        alpha[i][2] = 0; // rad/sec^2
    }
    psi = M_DEG2RAD(30.0);

    /* Perform fourbar-slider linkage analysis. */
    fslider.setLinks(r, theta[1][1], psi);
    fslider.angularPos(theta[1], theta[2], theta[3], theta[4]);
    fslider.angularVel(theta[1], omega[1]);
    fslider.angularVel(theta[2], omega[2]);
    fslider.angularVel(theta[3], omega[3]);
    fslider.angularVel(theta[4], omega[4]);
    fslider.angularAccel(theta[1], omega[1], alpha[1]);
    fslider.angularAccel(theta[2], omega[2], alpha[2]);
    fslider.angularAccel(theta[3], omega[3], alpha[3]);
    fslider.angularAccel(theta[4], omega[4], alpha[4]);
    r7ddot[1] = fslider.sliderAccel(theta[1], omega[1], alpha[1]);
    r7ddot[2] = fslider.sliderAccel(theta[2], omega[2], alpha[2]);
    r7ddot[3] = fslider.sliderAccel(theta[3], omega[3], alpha[3]);
    r7ddot[4] = fslider.sliderAccel(theta[4], omega[4], alpha[4]);

    /* Display the results. */
    printf("Circuit 1: Slider Acceleration\n");
    printf("   r7ddot = %.3f\n\n", r7ddot[1]);
    printf("Circuit 2: Slider Acceleration\n");
    printf("   r7ddot = %.3f\n\n", r7ddot[2]);
    printf("Circuit 3: Slider Acceleration\n");
    printf("   r7ddot = %.3f\n\n", r7ddot[3]);
    printf("Circuit 4: Slider Acceleration\n");
    printf("   r7ddot = %.3f\n\n", r7ddot[4]);

    return 0;
}
```

Output

```
Circuit 1: Slider Acceleration
   r7ddot = -0.096

Circuit 2: Slider Acceleration
   r7ddot = -0.062

Circuit 3: Slider Acceleration
   r7ddot = 0.065

Circuit 4: Slider Acceleration
   r7ddot = 0.145
```

CFourbarSlider::sliderPos

Synopsis

#include <sixbar.h>
double sliderPos(double *theta[1:6]***);**

Purpose
Calculate the positon of the slider.

Parameters

theta A double array of size 6 used to store the angular positions of the links.

Return Value
This function returns the horizontal position of the slider relative to A_0, the joint where r_1 and r_2 are fixed to ground (See Fig. 5.1).

Description
This function calculates the horizontal distance from A_0 to the slider. The return value is of `double` type.

Example
A fourbar-slider mechanism has the following propertities: $r_1 = 12m, r_2 = 4m, r_3 = 12m, r_4 = 7m, r_5 = 6m, r_6 = 9m$, $\theta_1 = 10^o$, $\psi = 30^o$. Given the angle θ_2, calculate the horizontal distance of the slider from link joint A_0.

```
/*****************************************************
 *  This example is for calculating the position of  *
 *  of the slider.                                   *
 *****************************************************/

#include<stdio.h>
#include<sixbar.h>


int main()
{
   double r[1:6], theta[1:4][1:6];
   double r7[1:4];
   double psi;
   int i;
   CFourbarSlider fslider;

   /* Default specification of the fourbar-slider linkage. */
   r[1] = 12;  r[2] = 4;  r[3] = 12;
   r[4] = 7;  r[5] = 6;  r[6] = 9;
   for(i = 1; i <= 4; i++)
   {
      theta[i][1] = M_DEG2RAD(10.0);
      theta[i][2] = M_DEG2RAD(70.0);
   }
   psi = M_DEG2RAD(30.0);

   /* Perform fourbar-slider linkage analysis. */
   fslider.setLinks(r, theta[1][1], psi);
   fslider.angularPos(theta[1], theta[2], theta[3], theta[4]);
   r7[1] = fslider.sliderPos(theta[1]);
   r7[2] = fslider.sliderPos(theta[2]);
   r7[3] = fslider.sliderPos(theta[3]);
   r7[4] = fslider.sliderPos(theta[4]);

   /* Display the results. */
```

```
    printf("Circuit 1: Slider Position\n");
    printf("   r7 = %.3f\n\n", r7[1]);
    printf("Circuit 2: Slider Position\n");
    printf("   r7 = %.3f\n\n", r7[2]);
    printf("Circuit 3: Slider Position\n");
    printf("   r7 = %.3f\n\n", r7[3]);
    printf("Circuit 4: Slider Position\n");
    printf("   r7 = %.3f\n\n", r7[4]);

    return 0;
}
```

Output

```
Circuit 1: Slider Position
   r7 = 22.486

Circuit 2: Slider Position
   r7 = 7.600

Circuit 3: Slider Position
   r7 = 15.489

Circuit 4: Slider Position
   r7 = -0.441
```

CFourbarSlider::sliderVel

Synopsis
#include <sixbar.h>
double sliderVel(double *theta[1:6],* **double** *omega[1:6]***);**

Purpose
Calculates the velocity of the slider.

Parameters

theta A double array of size 6 used to store the angular positions of the links.

omega A double array of size 6 used to store the angular velocities of the links.

Return Value
This function returns the linear velocity of the slider.

Description
This function calculates the linear velcity of the slider parallel to the horizontal. The return value is of `double` type.

Example
A fourbar-slider mechanism has the following properties: $r_1 = 12m, r_2 = 4m, r_3 = 12m, r_4 = 7m, r_5 = 6m, r_6 = 9m$, $\theta_1 = 10^o$, $\psi = 30^o$. Given the angle θ_2 and the angular velocity ω_2, calculate the linear velocity of the slider, $\dot{r}_7$.

```
/*************************************************
 *  This example is for calculating the velocity of  *
 *  the slider.                                      *
 *************************************************/

#include<stdio.h>
#include<sixbar.h>


int main()
{
   double r[1:6], theta[1:4][1:6];
   double omega[1:4][1:6];
   double r7dot[1:4];
   double psi;
   int i;
   CFourbarSlider fslider;

   /* Default specification of the fourbar-slider linkage. */
   r[1] = 12;  r[2] = 4;  r[3] = 12;
   r[4] = 7;  r[5] = 6;  r[6] = 9;
   for(i = 1; i <= 4; i++)
   {
      theta[i][1] = M_DEG2RAD(10.0);
      theta[i][2] = M_DEG2RAD(70.0);
      omega[i][2] = M_DEG2RAD(10.0); // rad/sec
   }
   psi = M_DEG2RAD(30.0);

   /* Perform fourbar-slider linkage analysis. */
   fslider.setLinks(r, theta[1][1], psi);
   fslider.angularPos(theta[1], theta[2], theta[3], theta[4]);
   fslider.angularVel(theta[1], omega[1]);
   fslider.angularVel(theta[2], omega[2]);
   fslider.angularVel(theta[3], omega[3]);
   fslider.angularVel(theta[4], omega[4]);
   r7dot[1] = fslider.sliderVel(theta[1], omega[1]);
   r7dot[2] = fslider.sliderVel(theta[2], omega[2]);
   r7dot[3] = fslider.sliderVel(theta[3], omega[3]);
   r7dot[4] = fslider.sliderVel(theta[4], omega[4]);

   /* Display the results. */
   printf("Circuit 1: Slider Velocity\n");
   printf("   r7dot = %.3f\n\n", r7dot[1]);
   printf("Circuit 2: Slider Velocity\n");
   printf("   r7dot = %.3f\n\n", r7dot[2]);
   printf("Circuit 3: Slider Velocity\n");
   printf("   r7dot = %.3f\n\n", r7dot[3]);
   printf("Circuit 4: Slider Velocity\n");
   printf("   r7dot = %.3f\n\n", r7dot[4]);

   return 0;
}
```

Output

```
Circuit 1: Slider Velocity
   r7dot = -0.570
```

```
Circuit 2: Slider Velocity
   r7dot = -0.225

Circuit 3: Slider Velocity
   r7dot = -0.200

Circuit 4: Slider Velocity
   r7dot = -0.668
```

CFourbarSlider::uscUnit

Synopsis
#include <sixbar.h>
void uscUnit(bool *unit*);

Purpose
Specify the use of SI or US Customary units in analysis.

Parameters

unit A boolean argument, where `true` indicates US Customary units are desired and `false` indicates SI units.

Return Value
None.

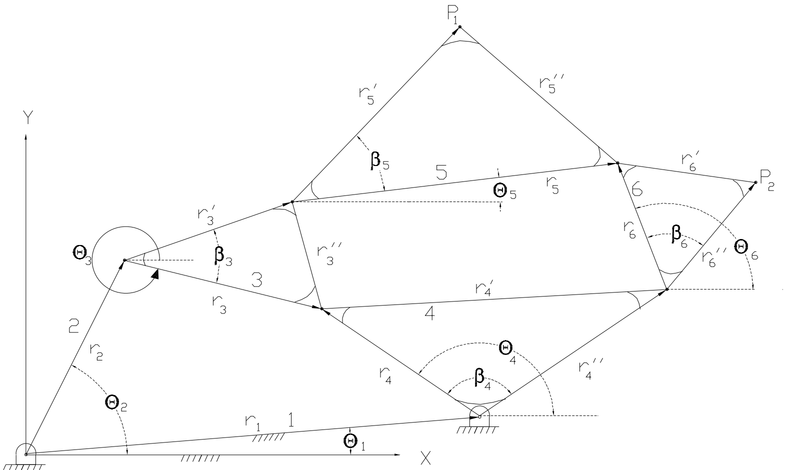
Description
This function specifies the whether SI or US Customary units are used.If `unit = true`, then US Customary units are used; otherwise, SI units are used. By default, SI units are assumed. This member function shall be called prior any other member function calls.

Appendix F

Class CWattSixbarI

CWattSixbarI

The header file **sixbar.h** includes header file **linkage.h**. The header file **sixbar.h** also contains a declaration of class **CWattSixbarI**. The **CWattSixbarI** class provides a means to analyze a Watt (I) sixbar linkage within a Ch language environment.



Public Data
None.

Public Member Functions

Functions	Descriptions
angularAccel	Given the angular acceleration of the input link, calculate the angular acceleration of other links.
angularPos	Given the angle of the input link, calculate the angle of other links.
angularVel	Given the angular velocity of the input link, calculate the angular velocity of other links.

animation	Watt (I) Sixbar linkage animation.
couplerPointAccel	Calculate the coupler point acceleration.
couplerPointPos	Calculate the position of the coupler point.
couplerPointVel	Calculate the coupler point velocity.
displayPosition	Display the position of the Watt (I) sixbar linkage.
setCouplerPoint	Set parameters for the coupler point(s).
setAngularVel	Set constant angular velocity of linkage 2.
setLinks	Set lengths of links.
setNumPoints	Set number of points for animation.
uscUnit	Specify the use of SI or US Customary units.

See Also

CWattSixbarI::angularAccel

Synopsis
#include <sixbar.h>
void angularAccel(double *theta[1:6]*, **double** *omega[1:6]*, **double** *alpha[1:6]*);

Purpose
Given the angular acceleration of the input link, calculate the angular acceleration of the other links.

Parameters

theta An array of double data type with angles of links.

omega An array of double data type with angular velocities of links.

alpha An array of double data type with angular accelerations of links.

Return Value
None.

Description
Given the angular acceleration of the input link, this function calculates the angular acceleration of the remaining moving links of the sixbar. *theta* is a one-dimensional array of size 6 which stores the angle of each link. *omega* is a one-dimensional array of size 6 which stores the angular velocity of each link. *alpha* is a one-dimensional array of size 6 which stores the angular acceleration of each link. The result of caculation is stored in array *alpha*.

Example
A Watt (I) sixbar linkage has parameters $r_1 = 8m, r_2 = 4m, r_3 = 10m, r_4 = 7m, r_5 = 7m, r_6 = 8m$, $r'_3 = 4m, \beta_3 = 30^o, r''_4 = 9m, \beta_4 = 50^o$, and $\theta_1 = 10^o$, . Given the angle θ_2, the angular velocity ω_2 and the angular acceleration α_2, calculate the angular acceleration of the other links.

```
#include<stdio.h>
#include<sixbar.h>

int main()
```

```
{
   double r[1:6];
   double rP3, beta3,
          rPP4, beta4;
   double theta[1:4][1:6], omega[1:4][1:6], alpha[1:4][1:6];
   double theta1 = M_DEG2RAD(10), theta2 = M_DEG2RAD(25);
   double omega2 = M_DEG2RAD(10), alpha2 = 0;
   CWattSixbarI wattI;
   int i;

   /* Define Watt (I) Sixbar linkage. */
   r[1] = 8;  r[2] = 4;
   r[3] = 10;  r[4] = 7;
   r[5] = 7;   r[6] = 8;
   rP3 = 4;  beta3 = M_DEG2RAD(30);
   rPP4 = 9;  beta4 = M_DEG2RAD(50);
   for(i = 1; i <= 4; i++)
   {
      theta[i][1] = theta1;
      theta[i][2] = theta2;
      omega[i][2] = omega2;
   }

   /* Perform analysis. */
   wattI.setLinks(r, theta1, rP3, beta3, rPP4, beta4);
   wattI.angularPos(theta[1], theta[2], theta[3], theta[4]);
   wattI.angularVel(theta[1], omega[1]);
   wattI.angularVel(theta[2], omega[2]);
   wattI.angularVel(theta[3], omega[3]);
   wattI.angularVel(theta[4], omega[4]);
   wattI.angularAccel(theta[1], omega[1], alpha[1]);
   wattI.angularAccel(theta[2], omega[2], alpha[2]);
   wattI.angularAccel(theta[3], omega[3], alpha[3]);
   wattI.angularAccel(theta[4], omega[4], alpha[4]);

   /* Display results. */
   for(i = 1; i <= 4; i++)
   {
      printf("Solution #%d:\n", i);
      printf("\t alpha3 = %.3f rad/s^2 (%.2f deg/s^2), alpha4 = %.3f rad/s^2 (%.2f deg/s^2),\n",
             alpha[i][3], M_RAD2DEG(alpha[i][3]),
             alpha[i][4], M_RAD2DEG(alpha[i][4]));
      printf("\t alpha5 = %.3f rad/s^2 (%.2f deg/s^2), alpha6 = %.3f rad/s^2 (%.2f deg/s^2)\n",
             alpha[i][5], M_RAD2DEG(alpha[i][5]),
             alpha[i][6], M_RAD2DEG(alpha[i][6]));
   }

   return 0;
}
```

Output

```
Solution #1:
 alpha3 = 0.050 rad/s^2 (2.86 deg/s^2), alpha4 = 0.079 rad/s^2 (4.51 deg/s^2),
 alpha5 = 0.008 rad/s^2 (0.47 deg/s^2), alpha6 = 0.117 rad/s^2 (6.69 deg/s^2)
Solution #2:
 alpha3 = 0.050 rad/s^2 (2.86 deg/s^2), alpha4 = 0.079 rad/s^2 (4.51 deg/s^2),
 alpha5 = 0.128 rad/s^2 (7.32 deg/s^2), alpha6 = 0.019 rad/s^2 (1.10 deg/s^2)
Solution #3:
 alpha3 = 0.023 rad/s^2 (1.34 deg/s^2), alpha4 = -0.006 rad/s^2 (-0.32 deg/s^2),
```

```
 alpha5 = 0.069 rad/s^2 (3.95 deg/s^2), alpha6 = 0.026 rad/s^2 (1.48 deg/s^2)
Solution #4:
 alpha3 = 0.023 rad/s^2 (1.34 deg/s^2), alpha4 = -0.006 rad/s^2 (-0.32 deg/s^2),
 alpha5 = 0.016 rad/s^2 (0.93 deg/s^2), alpha6 = 0.059 rad/s^2 (3.40 deg/s^2)
```

CWattSixbarI::angularPos

Synopsis
#include <sixbar.h>
void angularPos(double *theta_1[1:6]*, **double** *theta_2[1:6]*, **double** *theta_3[1:6]*, **double** *theta_4[1:6]*);

Purpose
Given the angle of the input link, calculate the angle of the other links.

Parameters

theta_1 A double array of size 6 for the first solution.

theta_2 A double array of size 6 for the second solution.

theta_3 A double array of size 6 for the third solution.

theta_4 A double array of size 6 for the fourth solution.

Return Value
None.

Description
Given the angular position of one link of a sixbar linkage, this function computes the angular positions of the remaining moving links. *theta_1* is a one-dimensional array of size 6 which stores the first solution of angular position. *theta_2* is a one-dimensional array of size 6 which stores the second solution of angular position. *theta_3* is a one-dimensional array of size 6 which stores the third solution of angular position. *theta_4* is a one-dimensional array of size 6 which stores the fourth solution of angular position.

Example
A Watt (I) sixbar linkage has link lengths $r_1 = 8m, r_2 = 4m, r_3 = 10m, r_4 = 7m, r_5 = 7m, r_6 = 8m$, $r'_3 = 4m, \beta_3 = 30^o, r''_4 = 9m, \beta_4 = 50^o$, and $\theta_1 = 10^o$. Given the angle θ_2, calculate the angular positions of the other links for each circuit.

```
#include<stdio.h>
#include<sixbar.h>

int main()
{
   double r[1:6];
   double rP3, beta3,
          rPP4, beta4;
   double theta[1:4][1:6];
   double theta1 = M_DEG2RAD(10), theta2 = M_DEG2RAD(25);
   CWattSixbarI wattI;
   int i;
```

```
   /* Define Watt (I) Sixbar linkage. */
   r[1] = 8;  r[2] = 4;
   r[3] = 10;  r[4] = 7;
   r[5] = 7;   r[6] = 8;
   rP3 = 4;  beta3 = M_DEG2RAD(30);
   rPP4 = 9;  beta4 = M_DEG2RAD(50);
   for(i = 1; i <= 4; i++)
   {
      theta[i][1] = theta1;
      theta[i][2] = theta2;
   }

   /* Perform analysis. */
   wattI.setLinks(r, theta1, rP3, beta3, rPP4, beta4);
   wattI.angularPos(theta[1], theta[2], theta[3], theta[4]);

   /* Display results. */
   for(i = 1; i <= 4; i++)
   {
      printf("Solution #%d:\n", i);
      printf(" theta3 = %.3f radians (%.2f degrees), theta4 = %.3f radians (%.2f degrees),\n",
             theta[i][3], M_RAD2DEG(theta[i][3]),
             theta[i][4], M_RAD2DEG(theta[i][4]));
      printf(" theta5 = %.3f radians (%.2f degrees), theta6 = %.3f radians (%.2f degrees)\n",
             theta[i][5], M_RAD2DEG(theta[i][5]),
             theta[i][6], M_RAD2DEG(theta[i][6]));
   }

   return 0;
}
```

Output

```
Solution #1:
 theta3 = 0.554 radians (31.73 degrees), theta4 = 0.918 radians (52.60 degrees),
 theta5 = 0.421 radians (24.11 degrees), theta6 = 2.240 radians (128.32 degrees)
Solution #2:
 theta3 = 0.554 radians (31.73 degrees), theta4 = 0.918 radians (52.60 degrees),
 theta5 = -1.006 radians (-57.62 degrees), theta6 = -2.824 radians (-161.83 degrees)
Solution #3:
 theta3 = -0.695 radians (-39.83 degrees), theta4 = -1.059 radians (-60.70 degrees),
 theta5 = -0.848 radians (-48.59 degrees), theta6 = 0.356 radians (20.39 degrees)
Solution #4:
 theta3 = -0.695 radians (-39.83 degrees), theta4 = -1.059 radians (-60.70 degrees),
 theta5 = -2.979 radians (-170.70 degrees), theta6 = 2.100 radians (120.32 degrees)
```

CWattSixbarI::angularVel

Synopsis
#include <sixbar.h>
void angularVel(double *theta[1:6],* **double** *omega[1:6]*);

Purpose
Given the angular velocity of one link, calculate the angular velocity of the other links.

Parameters

theta A double array used for the input angle of links.

omega A double array used for the angular velocities of links.

Return Value
None.

Description
Given the angular velocity of the input link, this function calculates the angular velocities of the remaining moving links of the Watt (I) sixbar. *theta* is an array for link positions. *omega* is an array for angular velocity of links.

Example
A Watt (I) sixbar linkage has link lengths $r_1 = 8m, r_2 = 4m, r_3 = 10m, r_4 = 7m, r_5 = 7m, r_6 = 8m$, $r_3' = 4m, \beta_3 = 30^o, r_4'' = 9m, \beta_4 = 50^o$, and angle $\theta_1 = 10^o$. Given the angle θ_2 and the angular velocity ω_2, determine the angular velocities of the other links for each circuit.

```
#include<stdio.h>
#include<sixbar.h>

int main()
{
   double r[1:6];
   double rP3, beta3,
          rPP4, beta4;
   double theta[1:4][1:6], omega[1:4][1:6];
   double theta1 = M_DEG2RAD(10), theta2 = M_DEG2RAD(25);
   double omega2 = M_DEG2RAD(10);
   CWattSixbarI wattI;
   int i;

   /* Define Watt (I) Sixbar linkage. */
   r[1] = 8;  r[2] = 4;
   r[3] = 10;  r[4] = 7;
   r[5] = 7;   r[6] = 8;
   rP3 = 4;  beta3 = M_DEG2RAD(30);
   rPP4 = 9;  beta4 = M_DEG2RAD(50);
   for(i = 1; i <= 4; i++)
   {
      theta[i][1] = theta1;
      theta[i][2] = theta2;
      omega[i][2] = omega2;
   }

   /* Perform analysis. */
   wattI.setLinks(r, theta1, rP3, beta3, rPP4, beta4);
   wattI.angularPos(theta[1], theta[2], theta[3], theta[4]);
   wattI.angularVel(theta[1], omega[1]);
   wattI.angularVel(theta[2], omega[2]);
   wattI.angularVel(theta[3], omega[3]);
   wattI.angularVel(theta[4], omega[4]);

   /* Display results. */
```

```
   for(i = 1; i <= 4; i++)
   {
      printf("Solution #%d:\n", i);
      printf("\t omega3 = %.3f rad/s (%.2f deg/s), omega4 = %.3f rad/s (%.2f deg/s),\n",
             omega[i][3], M_RAD2DEG(omega[i][3]),
             omega[i][4], M_RAD2DEG(omega[i][4]));
      printf("\t omega5 = %.3f rad/s (%.2f deg/s), omega6 = %.3f rad/s (%.2f deg/s)\n",
             omega[i][5], M_RAD2DEG(omega[i][5]),
             omega[i][6], M_RAD2DEG(omega[i][6]));
   }

   return 0;
}
```

Output

```
Solution #1:
 omega3 = -0.091 rad/s (-5.20 deg/s), omega4 = -0.033 rad/s (-1.88 deg/s),
 omega5 = -0.149 rad/s (-8.52 deg/s), omega6 = -0.066 rad/s (-3.77 deg/s)
Solution #2:
 omega3 = -0.091 rad/s (-5.20 deg/s), omega4 = -0.033 rad/s (-1.88 deg/s),
 omega5 = -0.057 rad/s (-3.27 deg/s), omega6 = -0.140 rad/s (-8.02 deg/s)
Solution #3:
 omega3 = -0.195 rad/s (-11.20 deg/s), omega4 = -0.253 rad/s (-14.52 deg/s),
 omega5 = -0.134 rad/s (-7.68 deg/s), omega6 = -0.184 rad/s (-10.55 deg/s)
Solution #4:
 omega3 = -0.195 rad/s (-11.20 deg/s), omega4 = -0.253 rad/s (-14.52 deg/s),
 omega5 = -0.194 rad/s (-11.14 deg/s), omega6 = -0.144 rad/s (-8.27 deg/s)
```

CWattSixbarI::animation

Synopsis
#include <sixbar.h>
int animation(int *branchnum*, ... /* [**int** *animationtype*, **string_t** *datafilename*] */);

Syntax
animation(*branchnum*)
animation(*branchnum*, *animationtype*)
animation(*branchnum*, *animationtype*, *datafilename*)

Purpose
An animation of a Watt (I) sixbar mechanism.

Parameters

branchnum an integer used for indicating which branch will be drawn.

it animationtype an optional parameter to specify the output type of the animation.

datafilename an optional parameter to specify the output file name.

Return Value
This function returns 0 on success and -1 on failure.

Description

This function simulates the motion of Watt (I) sixbar mechanism. *branchnum* is an integer number which indicates the branch you want to draw. *animationtype* is an optional parameter used to specify how the animation should be outputted. *animationtype* can be either of the following macros:
QANIMATE_OUTPUTTYPE_DISPLAY, QANIMATE_OUTPUTTYPE_FILE,
QANIMATE_OUTPUTTYPE_STREAM. QANIMATE_OUTPUTTYPE_DISPLAY displays an animation on the screen. QANIMATE_OUTPUTTYPE_FILE writes the animation data onto a file.
QANIMATE_OUTPUTTYPE_STREAM outputs the animation to the standard out. *datafilename* is an optional parameter to specify the output file name.

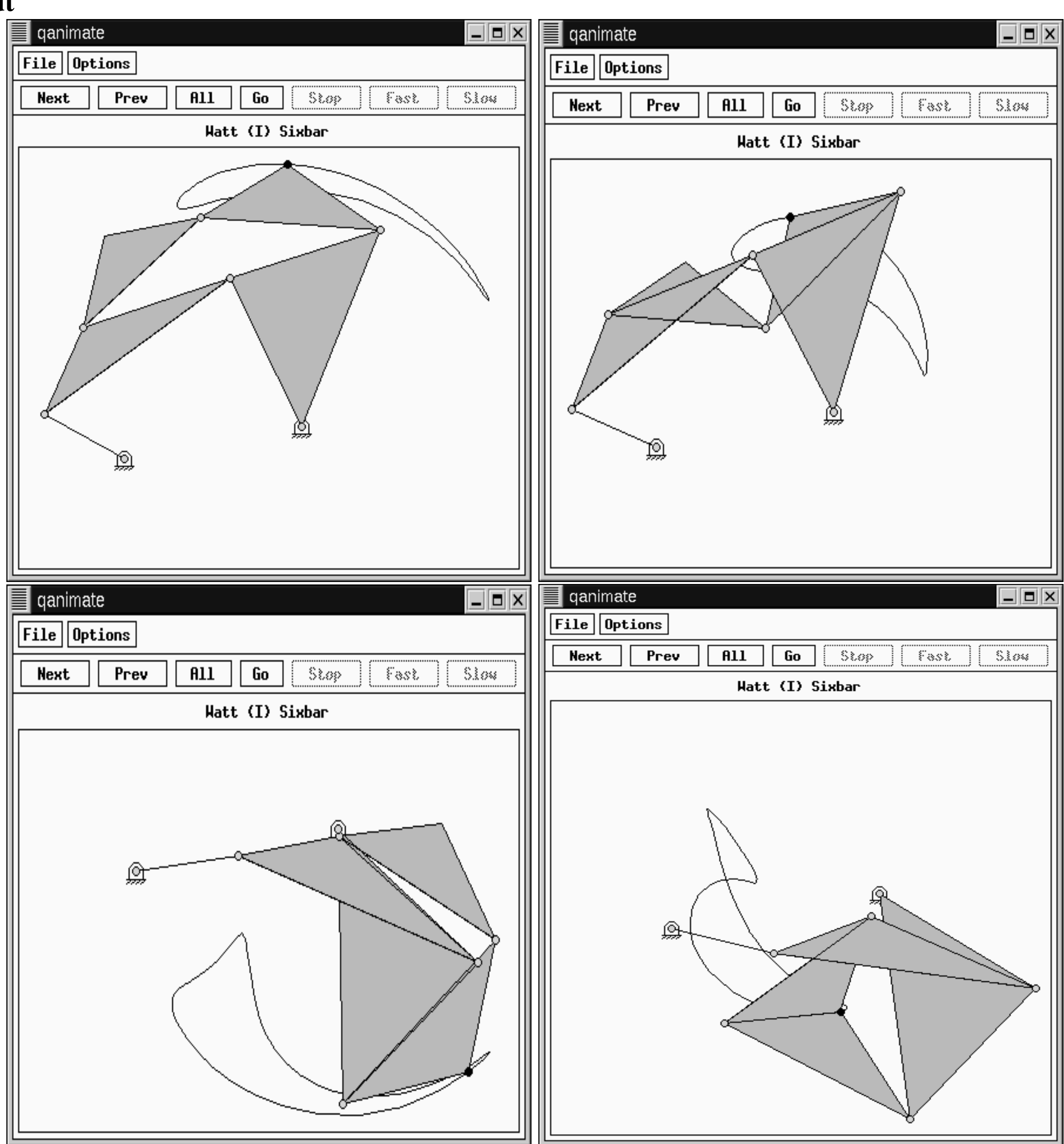
Example 1

For a Watt (I) sixbar linkage with link lengths $r_1 = 8m, r_2 = 4m, r_3 = 10m, r_4 = 7m, r_5 = 7m, r_6 = 8m, r'_3 = 4m, r''_4 = 9m, r'_5 = 4m, r''_6 = 5m$, and angles $\beta_3 = 30^o$, $\beta_4 = 50^o$, $\beta_5 = 35^o$, $\beta_6 = 30^o$, $\theta_1 = 10^o$, simulate the motion of the sixbar linkage. Also, trace the curve generated by the motion of the coupler on link 6.

```
#include<stdio.h>
#include<sixbar.h>

int main()
{
   double r[1:6];
   double rP3, beta3,
          rPP4, beta4,
          rP5, beta5,
          rPP6, beta6;
   double theta[1:4][1:6];
   double theta1 = M_DEG2RAD(10), theta2 = M_DEG2RAD(25);
   double complex P1[1:4], P2[1:4];
   CWattSixbarI wattI;
   int i;

   /* Define Watt (I) Sixbar linkage. */
   r[1] = 8;  r[2] = 4;
   r[3] = 10;  r[4] = 7;
   r[5] = 7;   r[6] = 8;
   rP3 = 4;  beta3 = M_DEG2RAD(30);
   rPP4 = 9;  beta4 = M_DEG2RAD(50);
   rP5 = 4;  beta5 = M_DEG2RAD(35);
   rPP6 = 5;  beta6 = M_DEG2RAD(30);
   for(i = 1; i <= 4; i++)
   {
      theta[i][1] = theta1;
      theta[i][2] = theta2;
   }

   /* Perform analysis. */
   wattI.setLinks(r, theta1, rP3, beta3, rPP4, beta4);
   wattI.setCouplerPoint(COUPLER_LINK5, rP5, beta5, TRACE_OFF);
   wattI.setCouplerPoint(COUPLER_LINK6, rPP6, beta6, TRACE_ON);
   wattI.animation(1);
   wattI.animation(2, QANIMATE_OUTPUTTYPE_FILE, "tempdata");
   wattI.animation(3);
   wattI.animation(4);
```

```
    return 0;
}
```

Output



CWattSixbarI::couplerPointAccel

Synopsis

#include <sixbar.h>

double complex couplerPointAccel(int *couplerLink*, **double** *theta[1:6]*, **double** *omega[1:6]*, **double** *alpha[1:6]*);

Purpose

Calculate the acceleration of the coupler point.

Parameters

couplerLink An *int* value specifying a macro to indicate which link the coupler is attached to.

theta An array of type *double* used to store the angular positions of each link.

omega An array of type *double* used to store the angular velocity of each link.

alpha An array of type *double* used to store the angular acceleration of each link.

Return Value
This function returns the acceleration of the coupler point.

Description
This function calculates the acceleration of the coupler point. The return value is a double complex, which indicates the vector of the acceleration of the coupler point.

Example
A Watt (I) sixbar linkage has link lengths $r_1 = 8m, r_2 = 4m, r_3 = 10m, r_4 = 7m, r_5 = 7m, r_6 = 8m, r'_3 = 4m, r''_4 = 9m, r'_5 = 4m, r''_6 = 5m$, and angles $\beta_3 = 30^o$, $\beta_4 = 50^o$, $\beta_5 = 35^o$, $\beta_6 = 30^o$, $\theta_1 = 10^o$. Given the angle θ_2, angular velocity ω_2 and angular acceleration α_2, calculate the acceleration of coupler points for each circuit.

```
#include<stdio.h>
#include<sixbar.h>

int main()
{
   double r[1:6];
   double rP3, beta3,
          rPP4, beta4,
          rP5, beta5,
          rPP6, beta6;
   double theta[1:4][1:6], omega[1:4][1:6], alpha[1:4][1:6];
   double theta1 = M_DEG2RAD(10), theta2 = M_DEG2RAD(25);
   double omega2 = M_DEG2RAD(10), alpha2 = 0;
   double complex Ap1[1:4], Ap2[1:4];
   CWattSixbarI wattI;
   int i;

   /* Define Watt (I) Sixbar linkage. */
   r[1] = 8;  r[2] = 4;
   r[3] = 10;  r[4] = 7;
   r[5] = 7;   r[6] = 8;
   rP3 = 4;  beta3 = M_DEG2RAD(30);
   rPP4 = 9;  beta4 = M_DEG2RAD(50);
   rP5 = 4;  beta5 = M_DEG2RAD(35);
   rPP6 = 5;  beta6 = M_DEG2RAD(30);
   for(i = 1; i <= 4; i++)
   {
      theta[i][1] = theta1;
      theta[i][2] = theta2;
      omega[i][2] = omega2;
   }
```

```
    /* Perform analysis. */
    wattI.setLinks(r, theta1, rP3, beta3, rPP4, beta4);
    wattI.setCouplerPoint(COUPLER_LINK5, rP5, beta5);
    wattI.setCouplerPoint(COUPLER_LINK6, rPP6, beta6);
    wattI.angularPos(theta[1], theta[2], theta[3], theta[4]);
    wattI.angularVel(theta[1], omega[1]);
    wattI.angularVel(theta[2], omega[2]);
    wattI.angularVel(theta[3], omega[3]);
    wattI.angularVel(theta[4], omega[4]);
    wattI.angularAccel(theta[1], omega[1], alpha[1]);
    wattI.angularAccel(theta[2], omega[2], alpha[2]);
    wattI.angularAccel(theta[3], omega[3], alpha[3]);
    wattI.angularAccel(theta[4], omega[4], alpha[4]);
    Ap1[1] = wattI.couplerPointAccel(COUPLER_LINK5, theta[1], omega[1], alpha[1]);
    Ap1[2] = wattI.couplerPointAccel(COUPLER_LINK5, theta[2], omega[2], alpha[2]);
    Ap1[3] = wattI.couplerPointAccel(COUPLER_LINK5, theta[3], omega[3], alpha[3]);
    Ap1[4] = wattI.couplerPointAccel(COUPLER_LINK5, theta[4], omega[4], alpha[4]);
    Ap2[1] = wattI.couplerPointAccel(COUPLER_LINK6, theta[1], omega[1], alpha[1]);
    Ap2[2] = wattI.couplerPointAccel(COUPLER_LINK6, theta[2], omega[2], alpha[2]);
    Ap2[3] = wattI.couplerPointAccel(COUPLER_LINK6, theta[3], omega[3], alpha[3]);
    Ap2[4] = wattI.couplerPointAccel(COUPLER_LINK6, theta[4], omega[4], alpha[4]);

    /* Display results. */
    for(i = 1; i <= 4; i++)
    {
       printf("Solution #%d:\n", i);
       printf("\t Ap1 = %.3f, Ap2 = %.3f\n", Ap1[i], Ap2[i]);
    }

    return 0;
}
```

Output

```
Solution #1:
 Ap1 = complex(-0.382,0.063), Ap2 = complex(-0.616,0.602)
Solution #2:
 Ap1 = complex(0.210,0.003), Ap2 = complex(0.034,0.593)
Solution #3:
 Ap1 = complex(0.021,0.169), Ap2 = complex(0.013,0.714)
Solution #4:
 Ap1 = complex(-0.137,-0.057), Ap2 = complex(-0.138,0.452)
```

CWattSixbarI::couplerPointPos

Synopsis

#include <sixbar.h>
void couplerPointPos(int *couplerLink*, **double** *theta2*, **double complex** *P[1:4]*);

Purpose

Calculate the position of the coupler point.

Parameters

couplerLink An *int* value specifying a macro to indicate which link the coupler is attached to.

theta2 A double number used for the input angle of link.

P A double complex array for the four solutions of coupler point.

Return Value
None.

Description
This function calculates the position of the coupler point. *theta2* is the input angle. *P[1:4]* is the four solutions of the coupler point position, respectively, which is a complex number indicating the vector of the coupler point.

Example
A Watt (I) sixbar linkage has link lengths $r_1 = 8m, r_2 = 4m, r_3 = 10m, r_4 = 7m, r_5 = 7m, r_6 = 8m, r'_3 = 4m, r''_4 = 9m, r'_5 = 4m, r''_6 = 5m$, and angles $\beta_3 = 30^o$, $\beta_4 = 50^o$, $\beta_5 = 35^o$, $\beta_6 = 30^o$, $\theta_1 = 10^o$. Given the angle θ_2, calculate the position of the coupler points for each circuit.

```
#include<stdio.h>
#include<sixbar.h>

int main()
{
   double r[1:6];
   double rP3, beta3,
          rPP4, beta4,
          rP5, beta5,
          rPP6, beta6;
   double theta[1:4][1:6];
   double theta1 = M_DEG2RAD(10), theta2 = M_DEG2RAD(25);
   double complex P1[1:4], P2[1:4];
   CWattSixbarI wattI;
   int i;

   /* Define Watt (I) Sixbar linkage. */
   r[1] = 8;  r[2] = 4;
   r[3] = 10;  r[4] = 7;
   r[5] = 7;   r[6] = 8;
   rP3 = 4;  beta3 = M_DEG2RAD(30);
   rPP4 = 9;  beta4 = M_DEG2RAD(50);
   rP5 = 4;  beta5 = M_DEG2RAD(35);
   rPP6 = 5;  beta6 = M_DEG2RAD(30);
   for(i = 1; i <= 4; i++)
   {
      theta[i][1] = theta1;
      theta[i][2] = theta2;
   }

   /* Perform analysis. */
   wattI.setLinks(r, theta1, rP3, beta3, rPP4, beta4);
   wattI.setCouplerPoint(COUPLER_LINK5, rP5, beta5);
   wattI.setCouplerPoint(COUPLER_LINK6, rPP6, beta6);
   wattI.angularPos(theta[1], theta[2], theta[3], theta[4]);
   wattI.couplerPointPos(COUPLER_LINK5, theta2, P1);
   wattI.couplerPointPos(COUPLER_LINK6, theta2, P2);
```

```
   /* Display results. */
   for(i = 1; i <= 4; i++)
   {
      printf("Solution #%d:\n", i);
      printf("\t P1 = %.3f, P2 = %.3f\n", P1[i], P2[i]);
   }

   return 0;
}
```

Output

```
Solution #1:
 P1 = complex(7.573,8.646), P2 = complex(16.146,6.744)
Solution #2:
 P1 = complex(9.212,3.675), P2 = complex(11.975,2.822)
Solution #3:
 P1 = complex(11.454,0.067), P2 = complex(9.627,-7.865)
Solution #4:
 P1 = complex(4.704,-1.786), P2 = complex(4.669,-2.030)
```

CWattSixbarI::couplerPointVel

Synopsis
#include <sixbar.h>
double complex couplerPointVel(int *couplerLink*, **double** *theta[1:6]*, **double** *omega[1:6]*);

Purpose
Calculate the velocity of the coupler point.

Parameters

couplerLink An *int* value specifying a macro to indicate whick link the coupler is attached to.

theta An array of type *double* used to store the angular positions of each link.

omega An array of type *double* used to store the angular velocities of each link.

Return Value
This function returns the velocity of the coupler point.

Description
This function calculates the velocity of the coupler point. The return value is a double complex, which indicates the vector of the velocity of the coupler point.

Example
A Watt (I) sixbar linkage has link lengths $r_1 = 8m, r_2 = 4m, r_3 = 10m, r_4 = 7m, r_5 = 7m, r_6 = 8m, r_3' = 4m, r_4'' = 9m, r_5' = 4m, r_6'' = 5m$, and angles $\beta_3 = 30^o$, $\beta_4 = 50^o$, $\beta_5 = 35^o$, $\beta_6 = 30^o$, $\theta_1 = 10^o$. Given the angle θ_2 and angular velocity ω_2, calculate the velocity of coupler points for each circuit.

```
#include<stdio.h>
#include<sixbar.h>

int main()
{
   double r[1:6];
   double rP3, beta3,
          rPP4, beta4,
          rP5, beta5,
          rPP6, beta6;
   double theta[1:4][1:6], omega[1:4][1:6];
   double theta1 = M_DEG2RAD(10), theta2 = M_DEG2RAD(25);
   double omega2 = M_DEG2RAD(10);
   double complex Vp1[1:4], Vp2[1:4];
   CWattSixbarI wattI;
   int i;

   /* Define Watt (I) Sixbar linkage. */
   r[1] = 8;  r[2] = 4;
   r[3] = 10;  r[4] = 7;
   r[5] = 7;   r[6] = 8;
   rP3 = 4;  beta3 = M_DEG2RAD(30);
   rPP4 = 9;  beta4 = M_DEG2RAD(50);
   rP5 = 4;  beta5 = M_DEG2RAD(35);
   rPP6 = 5;  beta6 = M_DEG2RAD(30);
   for(i = 1; i <= 4; i++)
   {
      theta[i][1] = theta1;
      theta[i][2] = theta2;
      omega[i][2] = omega2;
   }

   /* Perform analysis. */
   wattI.setLinks(r, theta1, rP3, beta3, rPP4, beta4);
   wattI.setCouplerPoint(COUPLER_LINK5, rP5, beta5);
   wattI.setCouplerPoint(COUPLER_LINK6, rPP6, beta6);
   wattI.angularPos(theta[1], theta[2], theta[3], theta[4]);
   wattI.angularVel(theta[1], omega[1]);
   wattI.angularVel(theta[2], omega[2]);
   wattI.angularVel(theta[3], omega[3]);
   wattI.angularVel(theta[4], omega[4]);
   Vp1[1] = wattI.couplerPointVel(COUPLER_LINK5, theta[1], omega[1]);
   Vp1[2] = wattI.couplerPointVel(COUPLER_LINK5, theta[2], omega[2]);
   Vp1[3] = wattI.couplerPointVel(COUPLER_LINK5, theta[3], omega[3]);
   Vp1[4] = wattI.couplerPointVel(COUPLER_LINK5, theta[4], omega[4]);
   Vp2[1] = wattI.couplerPointVel(COUPLER_LINK6, theta[1], omega[1]);
   Vp2[2] = wattI.couplerPointVel(COUPLER_LINK6, theta[2], omega[2]);
   Vp2[3] = wattI.couplerPointVel(COUPLER_LINK6, theta[3], omega[3]);
   Vp2[4] = wattI.couplerPointVel(COUPLER_LINK6, theta[4], omega[4]);

   /* Display results. */
   for(i = 1; i <= 4; i++)
   {
      printf("Solution #%d:\n", i);
      printf("\t Vp1 = %.3f, Vp2 = %.3f\n", Vp1[i], Vp2[i]);
   }

   return 0;
}
```

Output

```
Solution #1:
 Vp1 = complex(-0.113,-0.165), Vp2 = complex(0.339,-0.247)
Solution #2:
 Vp1 = complex(-0.691,0.321), Vp2 = complex(0.157,0.390)
Solution #3:
 Vp1 = complex(-1.077,-0.168), Vp2 = complex(-2.287,-0.101)
Solution #4:
 Vp1 = complex(-0.453,1.938), Vp2 = complex(-1.412,0.810)
```

CWattSixbarI::displayPosition

Synopsis
#include <sixbar.h>
int displayPosition(double *theta2*, **double** *theta3*, **double** *theta4*, **double** *theta5*, **double** *theta6*, ... /* [**int** *outputtype* [, [**char** * *filename*]] */);

Syntax
displayPosition(*theta2*, *theta3*, *theta4*, *theta5*, *theta6*)
displayPosition(*theta2*, *theta3*, *theta4*, *theta5*, *theta6*, *outputtype*)
displayPosition(*theta2*, *theta3*, *theta4*, *theta5*, *theta6*, *outputtype*, *filename*)

Purpose
Given θ_2, θ_3, θ_4, θ_5, and θ_6, display the current position of the Watt (I) sixbar linkage.

Parameters

theta2 θ_2 angle.

theta3 θ_3 angle.

theta4 θ_4 angle.

theta5 θ_5 angle.

theta6 θ_6 angle.

outputtype an optional argument for the output type.

filename an optional argument for the output file name.

Return Value
This function returns 0 on success and -1 on failure.

Description
Given θ_2, θ_3, θ_4, θ_5, and θ_6 display the current position of the Watt (I) sixbar linkage. `outputtype` is an optional parameter used to specify how the output should be handled. It may be one of the following macros: QANIMATE_OUTPUTTYPE_DISPLAY, QANIMATE_OUTPUTTYPE_FILE, QANIMATE_OUTPUTTYPE_STREAM. QANIMATE_OUTPUTTYPE_DISPLAY outputs the figure to the computer terminal. QANIMATE_OUTPUTTYPE_FILE writes the qanimate data onto a file. QANIMATE_OUTPUTTYPE_STREAM outputs the qanimate data to the standard out stream. `filename`

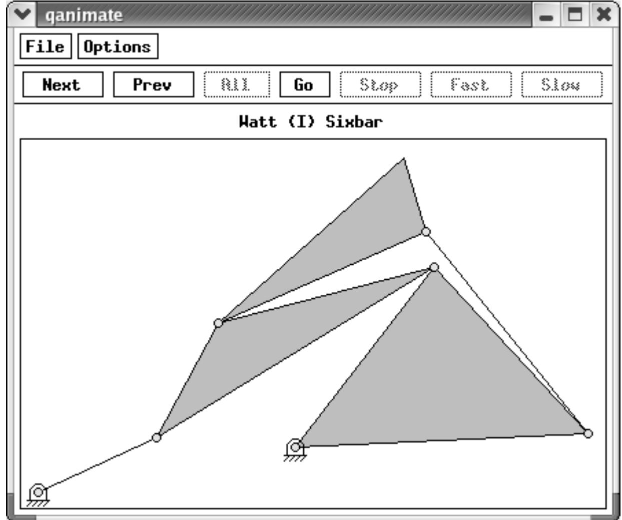
is an optional parameter to specify the output file name.

Example

A Watt (I) sixbar linkage has link lengths $r_1 = 8m, r_2 = 4m, r_3 = 10m, r_4 = 7m, r_5 = 7m, r_6 = 8m$, $\theta_1 = 10^o$, $r'_3 = 4m$, $\beta 3 = 30^o$, $r''_4 = 9m$, $beta_4 = 50^o$, and coupler properties $r_p = 4m$ and $\beta = 35^o$ attached to link 5. Given the angle θ_2, display the Watt (I) sixbar linkage in its current position for the first branch.

```
#include<stdio.h>
#include<sixbar.h>

int main()
{
   double r[1:6];
   double rP3, beta3,
          rPP4, beta4,
          rP5, beta5;
   double theta[1:4][1:6];
   double theta1 = M_DEG2RAD(10), theta2 = M_DEG2RAD(25);
   CWattSixbarI wattI;
   int i;

   /* Define Watt (I) Sixbar linkage. */
   r[1] = 8;  r[2] = 4;
   r[3] = 10;  r[4] = 7;
   r[5] = 7;   r[6] = 8;
   rP3 = 4;  beta3 = M_DEG2RAD(30);
   rPP4 = 9;  beta4 = M_DEG2RAD(50);
   rP5 = 4;  beta5 = M_DEG2RAD(35);
   for(i = 1; i <= 4; i++)
   {
      theta[i][1] = theta1;
      theta[i][2] = theta2;
   }

   /* Perform analysis. */
   wattI.setLinks(r, theta1, rP3, beta3, rPP4, beta4);
   wattI.setCouplerPoint(COUPLER_LINK5, rP5, beta5);
   wattI.angularPos(theta[1], theta[2], theta[3], theta[4]);
   wattI.displayPosition(theta[1][2], theta[1][3], theta[1][4], theta[1][5], theta[1][6]);

   return 0;
}
```

Output



CWattSixbarI::setCouplerPoint

Synopsis
#include <sixbar.h>
void setCouplerPoint(int *couplerLink*, **double** *rp, beta*, ... /* [**int** *trace*] */);

Syntax
setCouplerPoint(*couplerLink*, *rp*, *beta*)
setCouplerPoint(*couplerLink*, *rp*, *beta*, *trace*)

Purpose
Set parameters for the coupler point.

Parameters

couplerLink An *int* value specifying a macro to indicate which link the coupler is attached to.

rp A double number used for the link length connected to the coupler point.

beta A double number specifying the angular position of the coupler point relative to the link it is attached to.

it trace An optional parameter of **int** type specifying either macro TRACE_OFF or TRACE_ON to indicate whether the coupler curve should be traced during animation.

Return Value
None.

Description
This function sets the parameters of the coupler point on the link specified by *couplerLink*.

Example
see **CWattSixbarI::couplerPointAccel**().

CWattSixbarI::setAngularVel

Synopsis
#include <sixbar.h>
void setAngularVel(double *omega2*);

Purpose
Set the constant angular velocity of link2.

Parameters

omega2 A double number used for the constant input angular velocity of link2.

Return Value
None.

Description
This function sets the constant angular velocity of link2.

Example
see **CWattSixbarI::angularVel**().

CWattSixbarI::setLinks

Synopsis
#include <sixbar.h>
void setLinks(double *r[1:6]*, **double** *theta1, rP3, beta3, rPP4, beta4*);

Purpose
Set the lengths of links.

Parameters

r An array of double numbers used for the lengths of links.

theta1 A double number for the angle between link1 and the horizontal.

rP3 A double number for length r_3' of link 3.

beta3 A double number for the angle between r_3' and r_3.

rPP4 A double number for length r_4'' of link 4.

beta4 A double number for the angle between r_4'' and r_4.

Return Value
None.

Description
This function sets the lengths of the links and the known angles θ_1, β_3 and β_4.

Example
see **CWattSixbarI::angularPos**().

CWattSixbarI::setNumPoints

Synopsis
#include <sixbar.h>
void setNumPoints(int *numpoints*);

Purpose
Set number of points for the animation.

Parameters

numpoints An integer number used for the number of points.

Return Value
None.

Description
This function sets the number of points for the animation.

Example
see **CWattSixbarI::animation**().

CWattSixbarI::uscUnit

Synopsis
#include <sixbar.h>
void uscUnit(bool *unit*);

Purpose
Specify the use of SI or US Customary units in analysis.

Parameters

unit A boolean argument, where `true` indicates US Customary units are desired and `false` indicates SI units.

Return Value
None.

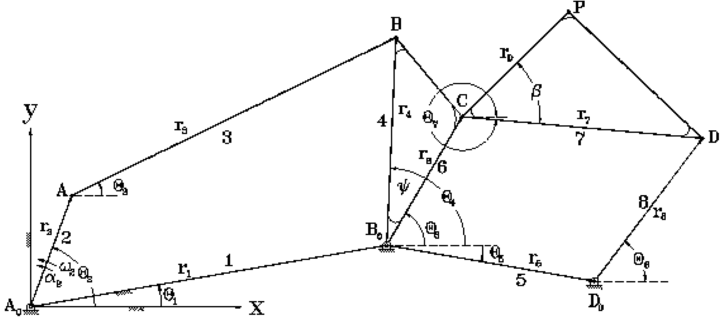
Description
This function specifies the whether SI or US Customary units are used.If `unit = true`, then US Customary units are used; otherwise, SI units are used. By default, SI units are assumed. This member function shall be called prior any other member function calls.

Appendix G

Class CWattSixbarII

CWattSixbarII

The header file **sixbar.h** includes header file **linkage.h**. The header file **sixbar.h** also contains a declaration of class **CWattSixbarII**. The **CWattSixbarII** class provides a means to analyze a Watt (II) sixbar linkage within a Ch language environment.



Public Data
None.

Public Member Functions

Functions	Descriptions
angularAccel	Given the angular acceleration of one link, calculate the angular acceleration of other links.
angularPos	Given the angle of one link, caculate the angle of other links.
angularVel	Given the angular velocity of one link, calculate the angular velocity of other links.
animation	Fourbar linkage animation.
couplerPointAccel	Calculate the coupler point acceleration.
couplerPointPos	Calculate the position of the coupler point.
couplerPointVel	Calculate the coupler point velocity.
displayPosition	Display the position of the Watt (II) sixbar linkage.
getIORanges	Calculate all the possible input/output ranges for the linkage.

setCouplerPoint	Set parameters for the coupler point.
setAngularVel	Set constant angular velocity of linkage 2.
setLinks	Set lengths of links.
setNumPoints	Set number of points for animation.
uscUnit	Specify the use of SI or US Customary units.

See Also

CWattSixbarII::angularAccel

Synopsis
#include <sixbar.h>
void angularAccel(double *theta[1:6]*, **double** *omega[1:6]*, **double** *alpha[1:6]*);

Purpose
Given the angular acceleration of the input link, calculate the angular acceleration of the other links.

Parameters

theta An array of double data type with angles of links.

omega An array of double data type with angular velocities of links.

alpha An array of double data type with angular accelerations of links.

Return Value
None.

Description
Given the angular acceleration of the input link, this function calculates the angular acceleration of the remaining moving links of the sixbar. *theta* is a one-dimensional array of size 6 which stores the angle of each link. *omega* is a one-dimensional array of size 6 which stores the angular velocity of each link. *alpha* is a one-dimensional array of size 6 which stores the angular acceleration of each link. The result of caculation is stored in array *alpha*.

Example
A Watt (II) sixbar linkage has parameters $r_1 = 12m, r_2 = 4m, r_3 = 12m, r_4 = 7m, r_5 = 7m, r_6 = 5m, r_7 = 8m, r_8 = 6m$, $\theta_1 = 10^o$, $\theta_5 = -10^o$, and $\psi = 30^o$. Given the angle θ_2, the angular velocity ω_2 and the angular acceleration α_2, calculate the angular acceleration of the other links.

```
/************************************************
 *  This example is for calculating the angular  *
 *  acceleration of various links of a           *
 *  Watt (II) sixbar linkage.                    *
 ************************************************/
#include <sixbar.h>

int main() {
    CWattSixbarII wattbar;
    double r[1:8], theta[1:8],theta_2[1:8],theta_3[1:8],theta_4[1:8];
```

```
    double omega[1:8],omega_2[1:8],omega_3[1:8],omega_4[1:8];
    double alpha[1:8],alpha_2[1:8],alpha_3[1:8],alpha_4[1:8];
    double theta1, theta5;
    double psi;
    int i;

    /* default specification of the four-bar linkage */
    r[1] = 12; r[2] = 4; r[3] = 12; r[4] = 7;
    r[5] = 7; r[6] = 5; r[7] = 8; r[8] =6;
    theta1 = 10*M_PI/180;
    theta5 = -10*M_PI/180;
    theta[2]=70*M_PI/180;
    omega[2]=10*M_PI/180;  /*  rad/sec  */
    alpha[2]=0;            /*  rad/sec*sec  */
    psi = 30*M_PI/180;


    theta_2[2] = theta_3[2] = theta_4[2] = theta[2];
    omega_2[2] = omega_3[2] = omega_4[2] = omega[2];
    alpha_2[2] = alpha_3[2] = alpha_4[2] = alpha[2];
    wattbar.setLinks(r, theta1, theta5, psi);
    wattbar.angularPos(theta, theta_2,theta_3,theta_4);
    wattbar.angularVel(theta, omega);
    wattbar.angularAccel(theta, omega, alpha);
    printf("\n Circuit 1: Angular Accelerations\n\n");
    for(i=2; i<=8; i++) {
       if(i!=5)
           printf("alpha[%d]=%6.3f\n", i, alpha[i]);
    }
    wattbar.angularVel(theta_2, omega_2);
    wattbar.angularAccel(theta_2, omega_2, alpha_2);
    printf("\n Circuit 2: Angular Accelerations\n\n");
    for(i=2; i<=8; i++) {
       if(i!=5)
           printf("alpha[%d]=%6.3f\n", i, alpha_2[i]);
    }
    wattbar.angularVel(theta_3, omega_3);
    wattbar.angularAccel(theta_3, omega_3, alpha_3);
    printf("\n Circuit 3: Angular Accelerations\n\n");
    for(i=2; i<=8; i++) {
       if(i!=5)
           printf("alpha[%d]=%6.3f\n", i, alpha_3[i]);
    }
    wattbar.angularVel(theta_4, omega_4);
    wattbar.angularAccel(theta_4, omega_4, alpha_4);
    printf("\n Circuit 4: Angular Accelerations\n\n");
    for(i=2; i<=8; i++) {
       if(i!=5)
           printf("alpha[%d]=%6.3f\n", i, alpha_4[i]);
    }
}
```

Output

```
 Circuit 1: Angular Accelerations
```

```
alpha[2]= 0.000
alpha[3]= 0.007
alpha[4]= 0.012
alpha[6]= 0.012
alpha[7]= 0.001
alpha[8]= 0.010

 Circuit 2: Angular Accelerations

alpha[2]= 0.000
alpha[3]= 0.007
alpha[4]= 0.012
alpha[6]= 0.012
alpha[7]= 0.009
alpha[8]=-0.000

 Circuit 3: Angular Accelerations

alpha[2]= 0.000
alpha[3]= 0.019
alpha[4]= 0.014
alpha[6]= 0.014
alpha[7]= 0.009
alpha[8]=-0.003

 Circuit 4: Angular Accelerations

alpha[2]= 0.000
alpha[3]= 0.019
alpha[4]= 0.014
alpha[6]= 0.014
alpha[7]= 0.001
alpha[8]= 0.013
```

CWattSixbarII::angularPos

Synopsis
#include <sixbar.h>
void angularPos(double *theta_1[1:6]*, **double** *theta_2[1:6]*, **double** *theta_3[1:6]*, **double** *theta_4[1:6]*);

Purpose
Given the angle of the input link, calculate the angle of the other links.

Parameters

theta_1 A double array of size 6 for the first solution.

theta_2 A double array of size 6 for the second solution.

theta_3 A double array of size 6 for the third solution.

theta_4 A double array of size 6 for the fourth solution.

Return Value
None.

Description

Given the angular position of one link of a sixbar linkage, this function computes the angular positions of the remaining moving links. *theta_1* is a one-dimensional array of size 6 which stores the first solution of angular position. *theta_2* is a one-dimensional array of size 6 which stores the second solution of angular position. *theta_3* is a one-dimensional array of size 6 which stores the third solution of angular position. *theta_4* is a one-dimensional array of size 6 which stores the fourth solution of angular position.

Example

A Watt (II) sixbar linkage has link lengths $r_1 = 12m, r_2 = 4m, r_3 = 12m, r_4 = 7m, r_5 = 7m, r_6 = 5m, r_7 = 8m, r_8 = 6m$, $\theta_1 = 10^o$, $\theta_5 = -10^o$, and $\psi = 30^o$. Given the angle θ_2, calculate the angular positions of the other links for each circuit.

```
/*************************************************
 *  This example is for calculating the angular  *
 *  position of various links of a              *
 *  Watt (II) sixbar linkage.                   *
 *************************************************/

#include <sixbar.h>

int main() {
  CWattSixbarII wattbar;
  double r[1:8];
  double theta1 = 10*M_PI/180;
  double theta5 = -10*M_PI/180;
  double rp = 5, beta = 45*M_PI/180;
  double psi = 30*M_PI/180;
  double theta_1[1:8], theta_2[1:8], theta_3[1:8], theta_4[1:8];
  double omega_1[1:8], omega_2[1:8], omega_3[1:8], omega_4[1:8];
  double alpha_1[1:8], alpha_2[1:8], alpha_3[1:8], alpha_4[1:8];
  double theta2 = 70*M_PI/180;
  double complex p[1:4]; // two solution of coupler point position
  int i;
  double complex vp1, vp2, vp3, vp4;
  double complex ap1, ap2, ap3, ap4;

  r[1] = 12, r[2] = 4, r[3] = 12.0, r[4] = 7.0;
  r[5] = 7,  r[6] = 5.0, r[7] = 8.0, r[8] = 6.0;
  theta_1[1] = theta1;
  theta_1[2] = theta2; // theta2

  wattbar.setLinks(r, theta1, theta5, psi);
  wattbar.angularPos(theta_1, theta_2, theta_3, theta_4);

  /**** print solutions ****/
  printf("\n Circuit 1: Angular Position\n\n");
  for(i=2; i<=8;i++)
    if(i!=5)
      printf("theta%d = %6.3f\n", i, theta_1[i]);
  printf("\n Circuit 2: Angular Position\n\n");
  for(i=2; i<=8;i++)
    if(i!=5)
      printf("theta%d = %6.3f\n", i, theta_2[i]);
  printf("\n Circuit 3: Angular Position\n\n");
  for(i=2; i<=8;i++)
    if(i!=5)
      printf("theta%d = %6.3f\n", i, theta_3[i]);
```

```
  printf("\n Circuit 4: Angular Position\n\n");
  for(i=2; i<=8;i++)
    if(i!=5)
      printf("theta%d = %6.3f\n", i, theta_4[i]);

  return 0;
}
```

Output

```

 Circuit 1: Angular Position

theta2 =  1.222
theta3 =  0.459
theta4 =  1.527
theta6 =  1.003
theta7 = -0.094
theta8 =  0.894

 Circuit 2: Angular Position

theta2 =  1.222
theta3 =  0.459
theta4 =  1.527
theta6 =  1.003
theta7 = -1.729
theta8 = -2.718

 Circuit 3: Angular Position

theta2 =  1.222
theta3 = -0.777
theta4 = -1.845
theta6 = -2.368
theta7 =  0.800
theta8 =  2.526

 Circuit 4: Angular Position

theta2 =  1.222
theta3 = -0.777
theta4 = -1.845
theta6 = -2.368
theta7 = -0.372
theta8 = -2.098
```

CWattSixbarII::angularVel

Synopsis
#include <sixbar.h>
void angularVel(double *theta[1:6],* **double** *omega[1:6]*);

Purpose
Given the angular velocity of one link, calculate the angular velocity of the other links.

Parameters

theta A double array used for the input angle of links.

omega A double array used for the angular velocities of links.

Return Value
None.

Description
Given the angular velocity of the input link, this function calculates the angular velocities of the remaining moving links of the Watt (II) sixbar. *theta* is an array for link positions. *omega* is an array for angular velocity of links.

Example
A Watt (II) sixbar linkage has link lengths $r_1 = 12m, r_2 = 4m, r_3 = 12m, r_4 = 7m, r_5 = 7m, r_6 = 5m, r_7 = 8m, r_8 = 6m$, and angles $\theta_1 = 10^o$, $\theta_5 = -10^o$, and $\psi = 30^o$. Given the angle θ_2 and the angular velocity ω_2, determine the angular velocities of the other links for each circuit.

```
/*************************************************
 *  This example is for calculating the angular  *
 *  velocity of various links of a               *
 *  Watt (II) sixbar linkage.                    *
 *************************************************/

#include <sixbar.h>

int main() {
    CWattSixbarII wattbar;
    double r[1:8];
    double theta1 = 10*M_PI/180;
    double theta5 = -10*M_PI/180;
    double psi = 30*M_PI/180;
    double theta[1:8], omega[1:8];
    double theta2[1:8], theta3[1:8], theta4[1:8];
    double omega2[1:8], omega3[1:8], omega4[1:8];
    int i;

    /* default specification of the four-bar linkage */
    r[1] = 12; r[2] = 4; r[3] = 12; r[4] = 7;
    r[5] = 7, r[6] = 5; r[7] = 8; r[8] = 6;
    theta[2]=70*M_PI/180;
    theta2[2] = theta3[2] = theta4[2] = theta[2];
    omega[2]=10*M_PI/180;  /*  rad/sec  */
    omega2[2] = omega3[2] = omega4[2] = omega[2];
    wattbar.setLinks(r, theta1, theta5, psi);
    wattbar.angularPos(theta, theta2, theta3, theta4);
    wattbar.angularVel(theta, omega);
    printf("\n Circuit 1: Angular Velocity\n\n");
    for(i=2;i<=8;i++) {
        if(i!=5)
           printf("omega[%d] = %6.3f\n", i, omega[i]);
    }
    wattbar.angularVel(theta2, omega2);
    printf("\n Circuit 2: Angular Velocity\n\n");
```

```
    for(i=2;i<=8;i++) {
        if(i!=5)
           printf("omega[%d] = %6.3f\n", i, omega2[i]);
    }
    wattbar.angularVel(theta3, omega3);
    printf("\n Circuit 3: Angular Velocity\n\n");
    for(i=2;i<=8;i++) {
        if(i!=5)
           printf("omega[%d] = %6.3f\n", i, omega3[i]);
    }
    wattbar.angularVel(theta4, omega4);
    printf("\n Circuit 4: Angular Velocity\n\n");
    for(i=2;i<=8;i++) {
        if(i!=5)
           printf("omega[%d] = %6.3f\n", i, omega4[i]);
    }
}
```

Output

```


 Circuit 1: Angular Velocity

omega[2] =  0.175
omega[3] = -0.020
omega[4] =  0.079
omega[6] =  0.079
omega[7] =  0.006
omega[8] =  0.070

 Circuit 2: Angular Velocity

omega[2] =  0.175
omega[3] = -0.020
omega[4] =  0.079
omega[6] =  0.079
omega[7] =  0.032
omega[8] = -0.031

 Circuit 3: Angular Velocity

omega[2] =  0.175
omega[3] = -0.005
omega[4] = -0.104
omega[6] = -0.104
omega[7] = -0.064
omega[8] = -0.002

 Circuit 4: Angular Velocity

omega[2] =  0.175
omega[3] = -0.005
omega[4] = -0.104
omega[6] = -0.104
omega[7] = -0.017
omega[8] = -0.080
```

CWattSixbarII::animation

Synopsis
#include <sixbar.h>
int animation(int *branchnum*, ... /* [**int** *animationtype*, **string_t** *datafilename*] */);

Syntax
animation(*branchnum*)
animation(*branchnum*, *animationtype*)
animation(*branchnum*, *animationtype*, *datafilename*)

Purpose
An animation of a Watt (II) sixbar mechanism.

Parameters

branchnum an integer used for indicating which branch will be drawn.

it animationtype an optional parameter to specify the output type of the animation.

datafilename an optional parameter to specify the file name of output.

Return Value
This function returns 0 on success and -1 on failure.

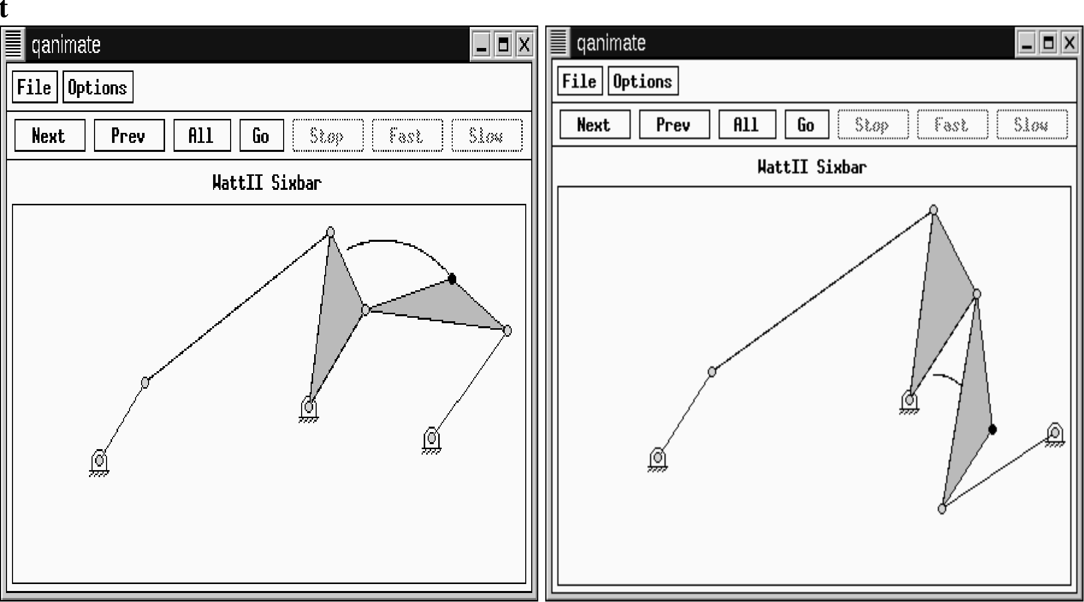
Description
This function simulates the motion of Watt (II) sixbar mechanism. *branchnum* is an integer number which indicates the branch you want to draw. *animationtype* is an optional parameter used to specify how the animation should be outputted. *animationtype* can be either of the following macros:
QANIMATE_OUTPUTTYPE_DISPLAY, QANIMATE_OUTPUTTYPE_FILE,
QANIMATE_OUTPUTTYPE_STREAM. QANIMATE_OUTPUTTYPE_DISPLAY displays an animation on the screen. QANIMATE_OUTPUTTYPE_FILE writes the animation data onto a file.
QANIMATE_OUTPUTTYPE_STREAM outputs the animation to the standard out. *datafilename* is an optional parameter to specify the file name of output if you want to output the data to a file.

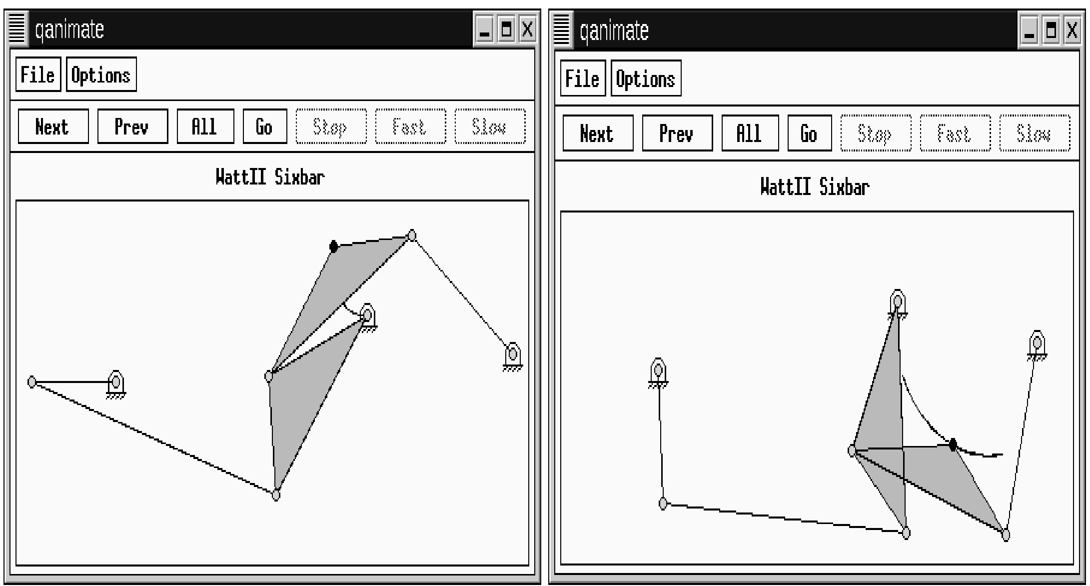
Example 1
For a Watt (II) sixbar linkage with parameters $r_1 = 12m, r_2 = 4m, r_3 = 12m, r_4 = 7m, r_5 = 7m, r_6 = 5m, r_7 = 8m, r_8 = 6m$, $\theta_1 = 10^o$, $\omega_2 = 10deg/sec$, and $\alpha_2 = 0$, simulate the motion of the sixbar linkage. Also, trace the motion generated by the coupler point attached to link 5 with parameters $r_p = 5m$ and $\beta = 20^o$.

```
/**************************************************
 *  This example is for simulating the motion of   *
 *  a Watt (II) sixbar linkage.                    *
 **************************************************/

#include <sixbar.h>

int main() {
    /* default specification of the four-bar linkage */
    double r[1:8];
    r[1]=12; r[2]=4; r[3]=12; r[4]=7; r[5] =7; r[6]=5; r[7]=8; r[8]=6;//cranker-rocker
```

```
    double theta1 = 10*M_PI/180, theta5 = -10*M_PI/180;
    double psi = 30*M_PI/180;
    double rp = 5, beta = 20*M_PI/180;
    double omega2=10*M_PI/180;  /*  rad/sec  */
    double alpha2=0;            /*  rad/sec*sec  */
    int numpoints =50;
    CWattSixbarII WattSixbar;

    WattSixbar.setLinks(r, theta1, theta5, psi);
    WattSixbar.setCouplerPoint(COUPLER_LINK7, rp, beta, TRACE_ON);
    WattSixbar.setNumPoints(numpoints);
    WattSixbar.animation(1);
    WattSixbar.animation(2);
    WattSixbar.animation(3);
    WattSixbar.animation(4);

    return 0;
}
```

Output





CWattSixbarII::couplerPointAccel

Synopsis
#include <sixbar.h>
double complex couplerPointAccel(int *couplerLink*, **double** *theta[1:]*, **double** *omega[1:]*, **double** *alpha[1:]*);

Purpose
Calculate the acceleration of the coupler point.

Parameters

couplerLink An *int* value specifying a macro to indicate which link the coupler is attached to.

theta An array of *double* type used to store the angular position values of each link.

omega An array of *double* type used to store the angular velocity values of each link.

alpha An array of *double* type used to store the angular acceleration values of each link.

Return Value
This function returns the acceleration of the coupler point.

Description
This function calculates the acceleration of the coupler point. The return value is a double complex, which indicates the vector of the acceleration of the coupler point.

Example
A Watt (II) sixbar linkage has link lengths $r_1 = 12m, r_2 = 4m, r_3 = 12m, r_4 = 7m, r_5 = 7m, r_6 = 5m, r_7 = 8m, r_8 = 6m, r_p = 5m$, and angles $\beta = 45^o$, $\theta_1 = 10^o$, $\theta_5 = -10^o$, and $\psi = 30^o$. Given the angle θ_2, angular velocity ω_2 and angular acceleration α_2, calculate the position of coupler point for

each circuit given that the coupler is attached to link 7. Also determine the velocity and acceleration of the coupler point.

```
/*************************************************
 *  This example is for calculating the coupler   *
 *  point position, velocity, and acceleration.   *
 *************************************************/

#include <sixbar.h>

int main() {
  CWattSixbarII wattbar;
  double r[1:8];
  double theta1 = 10*M_PI/180;
  double theta5 = -10*M_PI/180;
  double rp = 5, beta = 45*M_PI/180;
  double psi = 30*M_PI/180;
  double theta_1[1:8], theta_2[1:8], theta_3[1:8], theta_4[1:8];
  double omega_1[1:8], omega_2[1:8], omega_3[1:8], omega_4[1:8];
  double alpha_1[1:8], alpha_2[1:8], alpha_3[1:8], alpha_4[1:8];
  double theta2 = 70*M_PI/180;
  double complex p[1:4]; // four solution of coupler point position
  int i;
  double complex vp[1:4];
  double complex ap[1:4];

  r[1] = 12, r[2] = 4, r[3] = 12.0, r[4] = 7.0;
  r[5] = 7,  r[6] = 5.0, r[7] = 8.0, r[8] = 6.0;
  omega_1[2]=10*M_PI/180;  /*  rad/sec  */
  alpha_1[2]=0;  /*  rad/sec*sec  */

  theta_1[1] = theta1;
  theta_1[2] = theta2; // theta2
  omega_2[2] = omega_3[2] = omega_4[2] = omega_1[2];
  alpha_2[2] = alpha_3[2] = alpha_4[2] = alpha_1[2];

  wattbar.setLinks(r, theta1, theta5, psi);
  wattbar.setCouplerPoint(COUPLER_LINK7, rp, beta);

  wattbar.angularPos(theta_1, theta_2, theta_3, theta_4);
  wattbar.couplerPointPos(COUPLER_LINK7, theta_1[2], p);

  /* first solution */
  wattbar.angularVel(theta_1, omega_1);
  wattbar.angularAccel(theta_1, omega_1, alpha_1);

  /* coupler point velocity */
  vp[1] = wattbar.couplerPointVel(COUPLER_LINK7, theta_1, omega_1);

  /* coupler point acceleration */
  ap[1] = wattbar.couplerPointAccel(COUPLER_LINK7, theta_1, omega_1, alpha_1);

  /* second solution */
  wattbar.angularVel(theta_2, omega_2);
  wattbar.angularAccel(theta_2, omega_2, alpha_2);

  /* coupler point velocity */
  vp[2] = wattbar.couplerPointVel(COUPLER_LINK7, theta_2, omega_2);
```

```
  /* coupler point acceleration */
  ap[2] = wattbar.couplerPointAccel(COUPLER_LINK7, theta_2, omega_2, alpha_2);

  /* third solution */
  wattbar.angularVel(theta_3, omega_3);
  wattbar.angularAccel(theta_3, omega_3, alpha_3);

  /* coupler point velocity */
  vp[3] = wattbar.couplerPointVel(COUPLER_LINK7, theta_3, omega_3);

  /* coupler point acceleration */
  ap[3] = wattbar.couplerPointAccel(COUPLER_LINK7, theta_3, omega_3, alpha_3);

  /* fouth solution */
  wattbar.angularVel(theta_4, omega_4);
  wattbar.angularAccel(theta_4, omega_4, alpha_4);

  /* coupler point velocity */
  vp[4] = wattbar.couplerPointVel(COUPLER_LINK7, theta_4, omega_4);

  /* coupler point acceleration */
  ap[4] = wattbar.couplerPointAccel(COUPLER_LINK7, theta_4, omega_4, alpha_4);

  /* print solutions */
  for (i=1; i<=4; i++) {
    printf("P[%d] = %.3f\n", i, p[i]);
    printf("Vp[%d] = %.3f\n", i, vp[i]);
    printf("Ap[%d] = %.3f\n", i, ap[i]);
  }

  return 0;
}
```

Output

```
P[1] = complex(18.359,9.486)
Vp[1] = complex(-0.352,0.236)
Ap[1] = complex(-0.070,0.010)
P[2] = complex(17.439,2.250)
Vp[2] = complex(-0.201,0.306)
Ap[2] = complex(-0.033,0.036)
P[3] = complex(8.165,3.591)
Vp[3] = complex(-0.040,0.375)
Ap[3] = complex(0.041,-0.034)
P[4] = complex(12.818,0.600)
Vp[4] = complex(-0.327,0.291)
Ap[4] = complex(0.085,-0.010)
```

CWattSixbarII::couplerPointPos

Synopsis

#include <sixbar.h>
void couplerPointPos(int *couplerLink*, **double** *theta2*, **double complex** *p[1:4]*);

Purpose

Calculate the position of the coupler point.

Parameters

couplerLink An *int* value specifying a macro to indicate which link the coupler is attached to.

theta2 A double number used for the input angle of link.

p A double complex array for the four solutions of coupler point.

Return Value
None.

Description
This function calculates the position of the coupler point. *couplerLink* is a macro specifying the link that the coupler is attached to. *theta2* is the input angle. *p[1:4]* is the four solutions of the coupler point position, respectively, which is a complex number indicating the vector of the coupler point.

Example
see **CWattSixbarII::couplerPointAccel**().

CWattSixbarII::couplerPointVel

Synopsis
#include <sixbar.h>
double complex couplerPointVel(int *couplerLink*, **double** *theta[1:]*, **double** *omega[1:]*);

Purpose
Calculate the velocity of the coupler point.

Parameters

couplerLink An *int* value specifying a macro to indicate which link the coupler is attached to.

theta An array of *double* type used to store the angular position values of each link.

omega An array of *double* type used to store the angular velocity values of each link.

Return Value
This function returns the velocity of the coupler point.

Description
This function calculates the velocity of the coupler point. The return value is a double complex, which indicates the vector of the velocity of the coupler point.

Example
see **CWattSixbarII::couplerPointAccel**().

CWattSixbarII::displayPosition

Synopsis
#include <sixbar.h>

int displayPosition(double *theta2*, **double** *theta3*, **double** *theta4*, **double** *theta7*, **double** *theta8*, ... /* [**int** *outputtype* [, [**char *** *filename*]] */);

Syntax
displayPosition(*theta2*, *theta3*, *theta4*, *theta7*, *theta8*)
displayPosition(*theta2*, *theta3*, *theta4*, *theta7*, *theta8*, *outputtype*)
displayPosition(*theta2*, *theta3*, *theta4*, *theta7*, *theta8*, *outputtype*, *filename*)

Purpose
Given θ_2, θ_3, θ_4, θ_7, and θ_8, display the current position of the Watt (II) sixbar linkage.

Parameters

theta2 θ_2 angle.

theta3 θ_3 angle.

theta4 θ_4 angle.

theta7 θ_7 angle.

theta8 θ_8 angle.

outputtype an optional argument for the output type.

filename an optional argument for the output file name.

Return Value
This function returns 0 on success and -1 on failure.

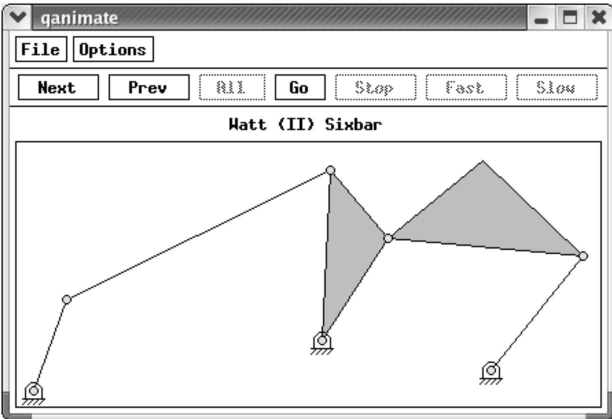
Description
Given θ_2, θ_3, θ_4, θ_7, and θ_8 display the current position of the Watt (II) sixbar linkage. `outputtype` is an optional parameter used to specify how the output should be handled. It may be one of the following macros: QANIMATE_OUTPUTTYPE_DISPLAY, QANIMATE_OUTPUTTYPE_FILE, QANIMATE_OUTPUTTYPE_STREAM. QANIMATE_OUTPUTTYPE_DISPLAY outputs the figure to the computer terminal. QANIMATE_OUTPUTTYPE_FILE writes the qanimate data onto a file. QANIMATE_OUTPUTTYPE_STREAM outputs the qanimate data to the standard out stream. `filename` is an optional parameter to specify the output file name.

Example
A Watt (II) sixbar linkage has link lengths $r_1 = 12m, r_2 = 4m, r_3 = 12m, r_4 = 7m, r_5 = 7m, r_6 = 5m, r_7 = 8m, r_8 = 6m$, $\theta_1 = 10^o$, $\theta_5 = -10^o$, $\psi = 30^o$, and coupler properties $r_p = 5m$ and $\beta = 45^o$. Given the angle θ_2, display the Watt (II) sixbar linkage in its current position for the first branch.

```
/************************************************
 *  This example is for displaying the position   *
 *  of the Watt (II) sixbar linkage.              *
 ************************************************/

#include </usr/ch/toolkit/include/sixbar.h>
```

```
int main() {
  CWattSixbarII wattbar;
  double r[1:8];
  double theta1 = 10*M_PI/180;
  double theta5 = -10*M_PI/180;
  double rp = 5, beta = 45*M_PI/180;
  double psi = 30*M_PI/180;
  double theta_1[1:8], theta_2[1:8], theta_3[1:8], theta_4[1:8];
  double theta2 = 70*M_PI/180;
  double complex p[1:4]; // four solution of coupler point position
  int i;
  double complex vp[1:4];
  double complex ap[1:4];

  r[1] = 12, r[2] = 4, r[3] = 12, r[4] = 7;
  r[5] = 7,  r[6] = 5, r[7] = 8, r[8] = 6;

  theta_1[1] = theta1;
  theta_1[2] = theta2; // theta2

  wattbar.uscUnit(1);
  wattbar.setLinks(r, theta1, theta5, psi);
  wattbar.setCouplerPoint(COUPLER_LINK7, rp, beta);
  wattbar.angularPos(theta_1, theta_2, theta_3, theta_4);
  wattbar.displayPosition(theta_1[2], theta_1[3], theta_1[4], theta_1[7], theta_1[8]);

  return 0;
}
```

Output



CWattSixbarII::getIORanges

Synopsis
#include <sixbar.h>
int getIORanges(double *in[1:][:]*, **double** *end[1:][:]*);

Purpose
Calculate all the possible input/output ranges of the mechanism.

Parameters

in An array that holds the possible input ranges of the linkage.

out An array that holds the possible output ranges of the linkage.

Return Value
This function returns the number of possible input/output ranges for the mechanism.

Description
This function calculates the various possible input/output ranges of a Watt (II) Sixbar linkage. The return value is an `int`, which indicates the number of possible ranges for the sixbar linkage.

Example
A Watt (II) sixbar linkage has link lengths $r_1 = 12m, r_2 = 4m, r_3 = 12m, r_4 = 7m, r_5 = 7m, r_6 = 5m, r_7 = 8m, r_8 = 6m$, $\theta_1 = 10^o$, $\theta_5 = -10^o$, and $\psi = 30^o$. Calculate the input/output ranges for each circuit.

```
/* Determine the input and output ranges of the Watt (II)
   sixbar linkage. */
#include<math.h>
#include<stdio.h>
#include <sixbar.h>

int main()
{
  double r[1:8], theta[1:8], psi, beta, rp;
  double input[5][2], output[5][2];
  int i, num_range;
  CWattSixbarII wattbar;

  /* specifications of the first four-bar linkage */
  r[1] = 12; r[2] = 4; r[3] = 12; r[4] = 7;
  theta[1] = 10*M_PI/180;

  /* specifications of the second four-bar linkage */
  r[5] = 7; r[6] = 5; r[7] = 8; r[8] = 6;
  theta[5] = -10*M_PI/180;

  /* adjoining angle */
  psi = 30*M_PI/180;

  /* input values */
  theta[2] = 70*M_PI/180;

  /* Setup Watt (II) sixbar linkage. */
  wattbar.setLinks(r, theta[1], theta[5], psi);
  wattbar.getIORanges(r, theta, psi, input, output);

  return 0;
}
```

Output

```
 0.591 <= theta2 <=  3.911,  0.746 <= theta8 <=  1.808
 0.593 <= theta2 <=  3.911, -2.640 <= theta8 <= -2.873
 2.724 <= theta2 <=  6.040,  2.485 <= theta8 <=  2.430
 2.718 <= theta2 <=  6.040, -2.402 <= theta8 <= -1.405
```

CWattSixbarII::setCouplerPoint

Synopsis
#include <sixbar.h>
void setCouplerPoint(int *couplerLink*, **double** *rp, beta*, ... /* [**int** trace] */);

Syntax
setCouplerPoint(*couplerLink*, *rp*, *beta*)
setCouplerPoint(*couplerLink*, *rp*, *beta*, *trace*)

Purpose
Set parameters for the coupler point.

Parameters

couplerLink An *int* value specifying a macro to indicate which link the coupler is attached to.

rp A double number used for the link length connected to the coupler point.

beta A double number specifying the angular position of the coupler point relative to the link it is attached to.

it trace An optional parameter of **int** type specifying either macro TRACE_OFF or TRACE_ON to indicate whether the coupler curve should be traced during animation.

Return Value
None.

Description
This function sets the parameters of the coupler point.

Example
see **CWattSixbarII::couplerPointAccel**().

CWattSixbarII::setAngularVel

Synopsis
#include <sixbar.h>
void setAngularVel(double *omega2*);

Purpose
Set the constant angular velocity of link2.

Parameters

omega2 A double number used for the constant angular velocity of link2.

Return Value
None.

Description
This function sets the constant angular velocity of link2.

Example
see **CWattSixbarII**::**angularVel**().

CWattSixbarII::setLinks

Synopsis
#include <sixbar.h>
int setLinks(**double** *r[1:8],* **double** *theta1, theta5, psi*);

Purpose
Set the lengths of links.

Parameters

r[1:8] A double number used for the lengths of links.

theta1 A double number for the angle between link1 and the horizontal.

theta5 A double number for the angle between link5 and the horizontal.

psi A double number for the included angle between link4 and link6.

Return Value

Description
This function sets the lengths of the links and the known angles θ_1, θ_5 and ψ.

Example
see **CWattSixbarII**::**angularPos**().

CWattSixbarII::setNumPoints

Synopsis
#include <sixbar.h>
void setNumPoints(**int** *numpoints*);

Purpose
Set number of points for the animation.

Parameters

numpoints An integer number used for the number of points.

Return Value
None.

Description
This function sets the number of points for the animation.

Example
see **CWattSixbarII**::**animation**().

CWattSixbarII::uscUnit

Synopsis
#include <sixbar.h>
void uscUnit(**bool** *unit*);

Purpose
Specify the use of SI or US Customary units in analysis.

Parameters

unit A boolean argument, where `true` indicates US Customary units are desired and `false` indicates SI units.

Return Value
None.

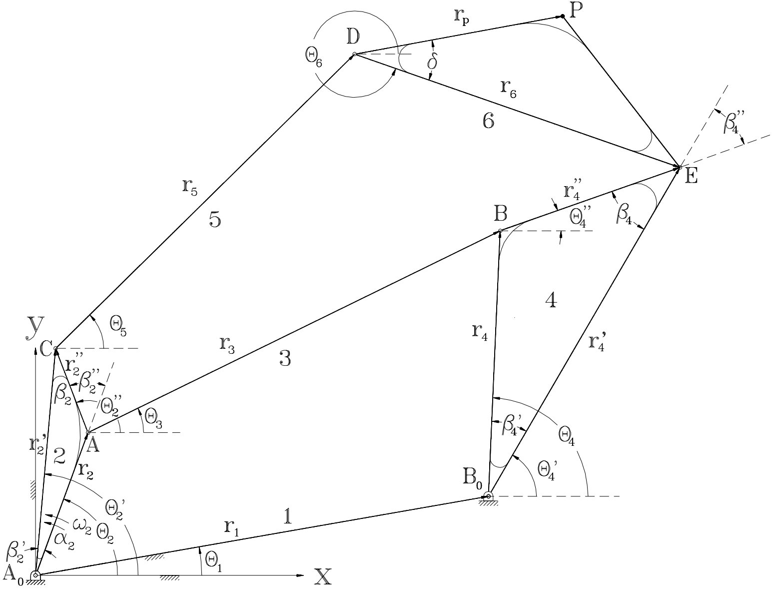
Description
This function specifies the whether SI or US Customary units are used.If `unit = true`, then US Customary units are used; otherwise, SI units are used. By default, SI units are assumed. This member function shall be called prior any other member function calls.

Appendix H

Class CStevSixbarI

CStevSixbarI

The header file **sixbar.h** includes header file **linkage.h**. The header file **sixbar.h** also contains a declaration of class **CStevSixbarI**. The **CStevSixbarI** class provides a means to analyze Stephenson (I) Sixbar linkage within a Ch language environment.



Public Data

None.

Public Member Functions

Functions	Descriptions
angularAccel	Given the angular acceleration of one link, calculate the angular acceleration of other links.
angularPos	Given the angle of one link, calculate the angle of other links.
angularVel	Given the angular velocity of one link, calculate the angular velocity of other links.

animation	Stephenson (I) linkage animation.
couplerPointAccel	Calculate the coupler point acceleration.
couplerPointPos	Calculate the position of the coupler point.
couplerPointVel	Calculate the coupler point velocity.
displayPosition	Display the position of the Stephenson (I) sixbar mechanism.
setCouplerPoint	Set parameters for the coupler point.
setAngularVel	Set constant angular velocity of linkage 2.
setLinks	Set lengths of links.
setNumPoints	Set number of points for animation.
uscUnit	Specify the use of SI or US Customary units.

See Also

CStevSixbarI::angularAccel

Synopsis
#include <sixbar.h>
void angularAccel(double *theta[1:6]*, **double** *omega[1:6]*, **double** *alpha[1:6]*);

Purpose
Given the angular acceleration of input link, calculate the angular acceleration of other links.

Parameters

theta An array of double data type with angles of links.

omega An array of double data type with angular velocities of links.

alpha An array of double data type with angular accelerations of links.

Return Value
None.

Description
Given the angular acceleration of input link, this function calculates the angular acceleration of the remaining moving links of the Stephenson (I) sixbar. *theta* is a one-dimensional array of size 6 which stores the angle of each link. *omega* is a one-dimensional array of size 6 which stores the angular velocity of each link. *alpha* is a one-dimensional array of size 6 which stores the angular acceleration of each link. The result of calculation is stored in array *alpha*.

Example
A Stephenson (I) sixbar linkage has parameters $r_1 = 12m, r_2 = 4m, r_3 = 12m, r_4 = 7m, r_5 = 11m, r_6 = 9m$, $rp2 = 6m$, $rp4 = 10m$, $\theta_1 = 10^o$, $\theta_2 = 70^o$, $\beta_2' = 15^o$, $\beta_4' = 30^o$, $rp = 5m$, and $\delta = 30^o$. Given the angle θ_2, the angular velocity ω_2 and the angular acceleration α_2, calculate the angular acceleration of the other links.

```
/************************************************
```

```
 *  This example is for calculating the angular  *
 *  accelerations of link3, link4, link5, and    *
 *  link6.                                       *
 ************************************************/
#include<stdio.h>
#include<sixbar.h>

int main()
{
   double r[1:6];
   double theta_1[1:6], theta_2[1:6], theta_3[1:6], theta_4[1:6];
   double omega_1[1:6], omega_2[1:6], omega_3[1:6], omega_4[1:6];
   double alpha_1[1:6], alpha_2[1:6], alpha_3[1:6], alpha_4[1:6];
   double theta1, theta2, omega2, alpha2;
   double rp2, rp4;
   double betaP2, betaP4, rp, delta;
   CStevSixbarI stevbar;
   int i;

   /* Specifications for the Stephenson I sixbar linkage. */
   r[1] = 12;  r[2] = 4;  r[3] = 12;  r[4] = 7;  r[5] = 11;  r[6] = 9;
   rp2 = 6; rp4 = 10;
   theta1 = 10*M_PI/180;  theta2 = 70*M_PI/180;
   betaP2 = 15*M_PI/180;  betaP4 = 30*M_PI/180;
   rp = 5;  delta = 30*M_PI/180;
   omega2 = 10*M_PI/180; /* rad/sec */
   alpha2 = 8*M_PI/180; /* rad/sec^2 */

   theta_1[1] = theta1;  theta_1[2] = theta2;
   theta_2[1] = theta1;  theta_2[2] = theta2;
   theta_3[1] = theta1;  theta_3[2] = theta2;
   theta_4[1] = theta1;  theta_4[2] = theta2;
   omega_1[2] = omega2;  alpha_1[2] = alpha2;
   omega_2[2] = omega2;  alpha_2[2] = alpha2;
   omega_3[2] = omega2;  alpha_3[2] = alpha2;
   omega_4[2] = omega2;  alpha_4[2] = alpha2;

   /* Perform analysis. */
   stevbar.setLinks(r, rp2, rp4, betaP2, betaP4, theta1);
   stevbar.angularPos(theta_1, theta_2, theta_3, theta_4);
   stevbar.angularVel(theta_1, omega_1);
   stevbar.angularVel(theta_2, omega_2);
   stevbar.angularVel(theta_3, omega_3);
   stevbar.angularVel(theta_4, omega_4);
   stevbar.angularAccel(theta_1, omega_1, alpha_1);
   stevbar.angularAccel(theta_2, omega_2, alpha_2);
   stevbar.angularAccel(theta_3, omega_3, alpha_3);
   stevbar.angularAccel(theta_4, omega_4, alpha_4);

   /* Display the results. */
   printf("1st Solution Set:\n");
   printf("\talpha3 = %.3f(%.2f), alpha4 = %.3f(%.2f),\n",
          alpha_1[3], alpha_1[3]*(180/M_PI), alpha_1[4], alpha_1[4]*(180/M_PI));
   printf("\talpha5 = %.3f(%.2f), alpha6 = %.3f(%.2f),\n\n",
          alpha_1[5], alpha_1[5]*(180/M_PI), alpha_1[6], alpha_1[6]*(180/M_PI));
   printf("2nd Solution Set:\n");
   printf("\talpha3 = %.3f(%.2f), alpha4 = %.3f(%.2f),\n",
          alpha_2[3], alpha_2[3]*(180/M_PI), alpha_2[4], alpha_2[4]*(180/M_PI));
   printf("\talpha5 = %.3f(%.2f), alpha6 = %.3f(%.2f),\n\n",
```

```
          alpha_2[5], alpha_2[5]*(180/M_PI), alpha_2[6], alpha_2[6]*(180/M_PI));
    printf("3rd Solution Set:\n");
    printf("\talpha3 = %.3f(%.2f), alpha4 = %.3f(%.2f),\n",
          alpha_3[3], alpha_3[3]*(180/M_PI), alpha_3[4], alpha_3[4]*(180/M_PI));
    printf("\talpha5 = %.3f(%.2f), alpha6 = %.3f(%.2f),\n\n",
          alpha_3[5], alpha_3[5]*(180/M_PI), alpha_3[6], alpha_3[6]*(180/M_PI));
    printf("4th Solution Set:\n");
    printf("\talpha3 = %.3f(%.2f), alpha4 = %.3f(%.2f),\n",
          alpha_4[3], alpha_4[3]*(180/M_PI), alpha_4[4], alpha_4[4]*(180/M_PI));
    printf("\talpha5 = %.3f(%.2f), alpha6 = %.3f(%.2f),\n\n",
          alpha_4[5], alpha_4[5]*(180/M_PI), alpha_4[6], alpha_4[6]*(180/M_PI));

    return 0;
}
```

Output

```
1st Solution Set:
alpha3 = -0.009(-0.50), alpha4 = 0.075(4.28),
alpha5 = 0.052(2.96), alpha6 = -0.016(-0.92),

2nd Solution Set:
alpha3 = -0.009(-0.50), alpha4 = 0.075(4.28),
alpha5 = -0.008(-0.44), alpha6 = 0.060(3.45),

3rd Solution Set:
alpha3 = 0.015(0.84), alpha4 = -0.069(-3.94),
alpha5 = 0.042(2.40), alpha6 = 0.097(5.54),

4th Solution Set:
alpha3 = 0.015(0.84), alpha4 = -0.069(-3.94),
alpha5 = 0.079(4.52), alpha6 = 0.024(1.39),
```

CStevSixbarI::angularPos

Synopsis

#include <gearedfivebar.h>
void angularPos(double *theta_1[1:6]*, **double** *theta_2[1:6]*, **double** *theta_3[1:6]*, **double** *theta_4[1:6]*);

Purpose

Given the angle of input link, calculate the angle of other links.

Parameters

theta_1 A double array with dimension size of 6 for the first solution.

theta_2 A double array with dimension size of 6 for the second solution.

theta_3 A double array with dimension size of 6 for the third solution.

theta_4 A double array with dimension size of 6 for the fourth solution.

Return Value

None.

Description

Given the angular position of the input link of a Stephen (I) linkage, this function computes the angular positions of the remaining moving links. *theta_1* is a one-dimensional array of size 6 which stores the first solution of angular. *theta_2* is a one-dimensional array of size 6 which stores the second solution of angular. *theta_3* is a one-dimensional array of size 6 which stores the third solution of angular. *theta_4* is a one-dimensional array of size 6 which stores the fourth solution of angular.

Example

A Stephenson (I) Sixbar linkage has parameters $r_1 = 12m, r_2 = 4m, r_3 = 12m, r_4 = 7m, r_5 = 11m, r_6 = 9m, rp2 = 6m, rp4 = 10m, \theta_1 = 10^o, \theta_2 = 70^o, \beta_2' = 15^o, \beta_4' = 30^o, rp = 5m, and \delta = 30^o$. Given the angle θ_2, calculate the angular positions of the other links.

```
/************************************************
 *  This example is for calculating the angular  *
 *  positions of link3, link4, link5, and        *
 *  link6.                                       *
 ************************************************/
#include<stdio.h>
#include<sixbar.h>

int main()
{
   double r[1:6];
   double theta_1[1:6], theta_2[1:6], theta_3[1:6], theta_4[1:6];
   double theta1, theta2;
   double rp2, rp4;
   double betaP2, betaP4, rp, delta;
   CStevSixbarI stevbar;
   int i;

   /* Specifications for the Stephenson I sixbar linkage. */
   r[1] = 12;  r[2] = 4;  r[3] = 12;  r[4] = 7;  r[5] = 11;  r[6] = 9;
   rp2 = 6; rp4 = 10;
   theta1 = 10*M_PI/180;  theta2 = 70*M_PI/180;
   betaP2 = 15*M_PI/180;  betaP4 = 30*M_PI/180;
   rp = 5;  delta = 30*M_PI/180;

   theta_1[1] = theta1;  theta_1[2] = theta2;
   theta_2[1] = theta1;  theta_2[2] = theta2;
   theta_3[1] = theta1;  theta_3[2] = theta2;
   theta_4[1] = theta1;  theta_4[2] = theta2;

   /* Perform analysis. */
   stevbar.setLinks(r, rp2, rp4, betaP2, betaP4, theta1);
   stevbar.angularPos(theta_1, theta_2, theta_3, theta_4);

   /* Display the results. */
   printf("1st Solution Set:\n");
   printf("\ttheta3 = %.3f(%.2f), theta4 = %.3f(%.2f),\n",
          theta_1[3], theta_1[3]*(180/M_PI), theta_1[4], theta_1[4]*(180/M_PI));
   printf("\ttheta5 = %.3f(%.2f), theta6 = %.3f(%.2f),\n\n",
          theta_1[5], theta_1[5]*(180/M_PI), theta_1[6], theta_1[6]*(180/M_PI));
   printf("2nd Solution Set:\n");
   printf("\ttheta3 = %.3f(%.2f), theta4 = %.3f(%.2f),\n",
          theta_2[3], theta_2[3]*(180/M_PI), theta_2[4], theta_2[4]*(180/M_PI));
   printf("\ttheta5 = %.3f(%.2f), theta6 = %.3f(%.2f),\n\n",
          theta_2[5], theta_2[5]*(180/M_PI), theta_2[6], theta_2[6]*(180/M_PI));
```

```
   printf("3rd Solution Set:\n");
   printf("\ttheta3 = %.3f(%.2f), theta4 = %.3f(%.2f),\n",
          theta_3[3], theta_3[3]*(180/M_PI), theta_3[4], theta_3[4]*(180/M_PI));
   printf("\ttheta5 = %.3f(%.2f), theta6 = %.3f(%.2f),\n\n",
          theta_3[5], theta_3[5]*(180/M_PI), theta_3[6], theta_3[6]*(180/M_PI));
   printf("4th Solution Set:\n");
   printf("\ttheta3 = %.3f(%.2f), theta4 = %.3f(%.2f),\n",
          theta_4[3], theta_4[3]*(180/M_PI), theta_4[4], theta_4[4]*(180/M_PI));
   printf("\ttheta5 = %.3f(%.2f), theta6 = %.3f(%.2f),\n\n",
          theta_4[5], theta_4[5]*(180/M_PI), theta_4[6], theta_4[6]*(180/M_PI));

   return 0;
}
```

Output

```
1st Solution Set:
theta3 = 0.459(26.31), theta4 = 1.527(87.48),
theta5 = -0.206(-11.82), theta6 = 0.855(49.01),

2nd Solution Set:
theta3 = 0.459(26.31), theta4 = 1.527(87.48),
theta5 = 0.738(42.29), theta6 = -0.324(-18.54),

3rd Solution Set:
theta3 = -0.777(-44.52), theta4 = -1.845(-105.70),
theta5 = -2.023(-115.92), theta6 = -0.110(-6.28),

4th Solution Set:
theta3 = -0.777(-44.52), theta4 = -1.845(-105.70),
theta5 = -0.391(-22.43), theta6 = -2.305(-132.06),
```

CStevSixbarI::angularVel

Synopsis
#include <sixbar.h>
void angularVel(double *theta[1:6],* **double** *omega[1:6]*);

Purpose
Given the angular velocity of one link, calculate the angular velocities of other links.

Parameters

theta A double array used for the input angle of links.

omega A double array used for the angular velocities of links.

Return Value
None.

Description
Given the angular velocity of the input link, this function calculates the angular velocities of the remaining links of the Stephenson (I) linkage. *theta* is an array for link positions. *omega* is an array for angular velocity

of links.

Example

A Stephenson (I) Sixbar linkage has link lengths $r_1 = 12m, r_2 = 4m, r_3 = 12m, r_4 = 7m, r_5 = 11m, r_6 = 9m, rp2 = 6m, rp4 = 10m, \theta_1 = 10^o, \theta_2 = 70^o, \beta_2' = 15^o, \beta_4' = 30^o, rp = 5m, and \delta = 30^o$. Given the angle θ_2 and the angular velocity ω_2, determine the angular velocities of the other links.

```
/************************************************
 *  This example is for calculating the angular  *
 *  velocities of link3, link4, link5, and       *
 *  link6.                                       *
 ************************************************/
#include<stdio.h>
#include<sixbar.h>

int main()
{
   double r[1:6];
   double theta_1[1:6], theta_2[1:6], theta_3[1:6], theta_4[1:6];
   double omega_1[1:6], omega_2[1:6], omega_3[1:6], omega_4[1:6];
   double theta1, theta2, omega2;
   double rp2, rp4;
   double betaP2, betaP4, rp, delta;
   CStevSixbarI stevbar;
   int i;

   /* Specifications for the Stephenson I sixbar linkage. */
   r[1] = 12;  r[2] = 4;  r[3] = 12;  r[4] = 7;  r[5] = 11;  r[6] = 9;
   rp2 = 6; rp4 = 10;
   theta1 = 10*M_PI/180;  theta2 = 70*M_PI/180;
   betaP2 = 15*M_PI/180;  betaP4 = 30*M_PI/180;
   rp = 5;  delta = 30*M_PI/180;
   omega2 = 10*M_PI/180; /* rad/sec */

   theta_1[1] = theta1;  theta_1[2] = theta2;
   theta_2[1] = theta1;  theta_2[2] = theta2;
   theta_3[1] = theta1;  theta_3[2] = theta2;
   theta_4[1] = theta1;  theta_4[2] = theta2;
   omega_1[2] = omega2;
   omega_2[2] = omega2;
   omega_3[2] = omega2;
   omega_4[2] = omega2;

   /* Perform analysis. */
   stevbar.setLinks(r, rp2, rp4, betaP2, betaP4, theta1);
   stevbar.angularPos(theta_1, theta_2, theta_3, theta_4);
   stevbar.angularVel(theta_1, omega_1);
   stevbar.angularVel(theta_2, omega_2);
   stevbar.angularVel(theta_3, omega_3);
   stevbar.angularVel(theta_4, omega_4);

   /* Display the results. */
   printf("1st Solution Set:\n");
   printf("\tomega3 = %.3f(%.2f), omega4 = %.3f(%.2f),\n",
          omega_1[3], omega_1[3]*(180/M_PI), omega_1[4], omega_1[4]*(180/M_PI));
   printf("\tomega5 = %.3f(%.2f), omega6 = %.3f(%.2f),\n\n",
          omega_1[5], omega_1[5]*(180/M_PI), omega_1[6], omega_1[6]*(180/M_PI));
   printf("2nd Solution Set:\n");
```

```
   printf("\tomega3 = %.3f(%.2f), omega4 = %.3f(%.2f),\n",
          omega_2[3], omega_2[3]*(180/M_PI), omega_2[4], omega_2[4]*(180/M_PI));
   printf("\tomega5 = %.3f(%.2f), omega6 = %.3f(%.2f),\n\n",
          omega_2[5], omega_2[5]*(180/M_PI), omega_2[6], omega_2[6]*(180/M_PI));
   printf("3rd Solution Set:\n");
   printf("\tomega3 = %.3f(%.2f), omega4 = %.3f(%.2f),\n",
          omega_3[3], omega_3[3]*(180/M_PI), omega_3[4], omega_3[4]*(180/M_PI));
   printf("\tomega5 = %.3f(%.2f), omega6 = %.3f(%.2f),\n\n",
          omega_3[5], omega_3[5]*(180/M_PI), omega_3[6], omega_3[6]*(180/M_PI));
   printf("4th Solution Set:\n");
   printf("\tomega3 = %.3f(%.2f), omega4 = %.3f(%.2f),\n",
          omega_4[3], omega_4[3]*(180/M_PI), omega_4[4], omega_4[4]*(180/M_PI));
   printf("\tomega5 = %.3f(%.2f), omega6 = %.3f(%.2f),\n\n",
          omega_4[5], omega_4[5]*(180/M_PI), omega_4[6], omega_4[6]*(180/M_PI));

   return 0;
}
```

Output

```
1st Solution Set:
omega3 = -0.020(-1.14), omega4 = 0.079(4.51),
omega5 = 0.052(2.98), omega6 = -0.039(-2.22),

2nd Solution Set:
omega3 = -0.020(-1.14), omega4 = 0.079(4.51),
omega5 = -0.027(-1.52), omega6 = 0.064(3.68),

3rd Solution Set:
omega3 = -0.005(-0.29), omega4 = -0.104(-5.93),
omega5 = 0.024(1.36), omega6 = 0.085(4.90),

4th Solution Set:
omega3 = -0.005(-0.29), omega4 = -0.104(-5.93),
omega5 = 0.067(3.85), omega6 = 0.006(0.32),
```

CStevSixbarI::animation

Synopsis
#include <sixbar.h>
int animation(**int** *branchnum*, ... /* [**int** *animationtype*, **string_t** *datafilename*] */);

Syntax
animation(*branchnum*)
animation(*branchnum*), *animationtype*)
animation(*branchnum*, *animationtype*, *datafilename*)

Purpose
An animation of a Stephenson (I) Sixbar mechanism.

Parameters

branchnum an integer used for indicating which branch will be drawn.

it animationtype an optional parameter to specify the output type of the animation.

datafilename an optional parameter to specify the filename of output.

Return Value
This function returns 0 on success and -1 on failure.

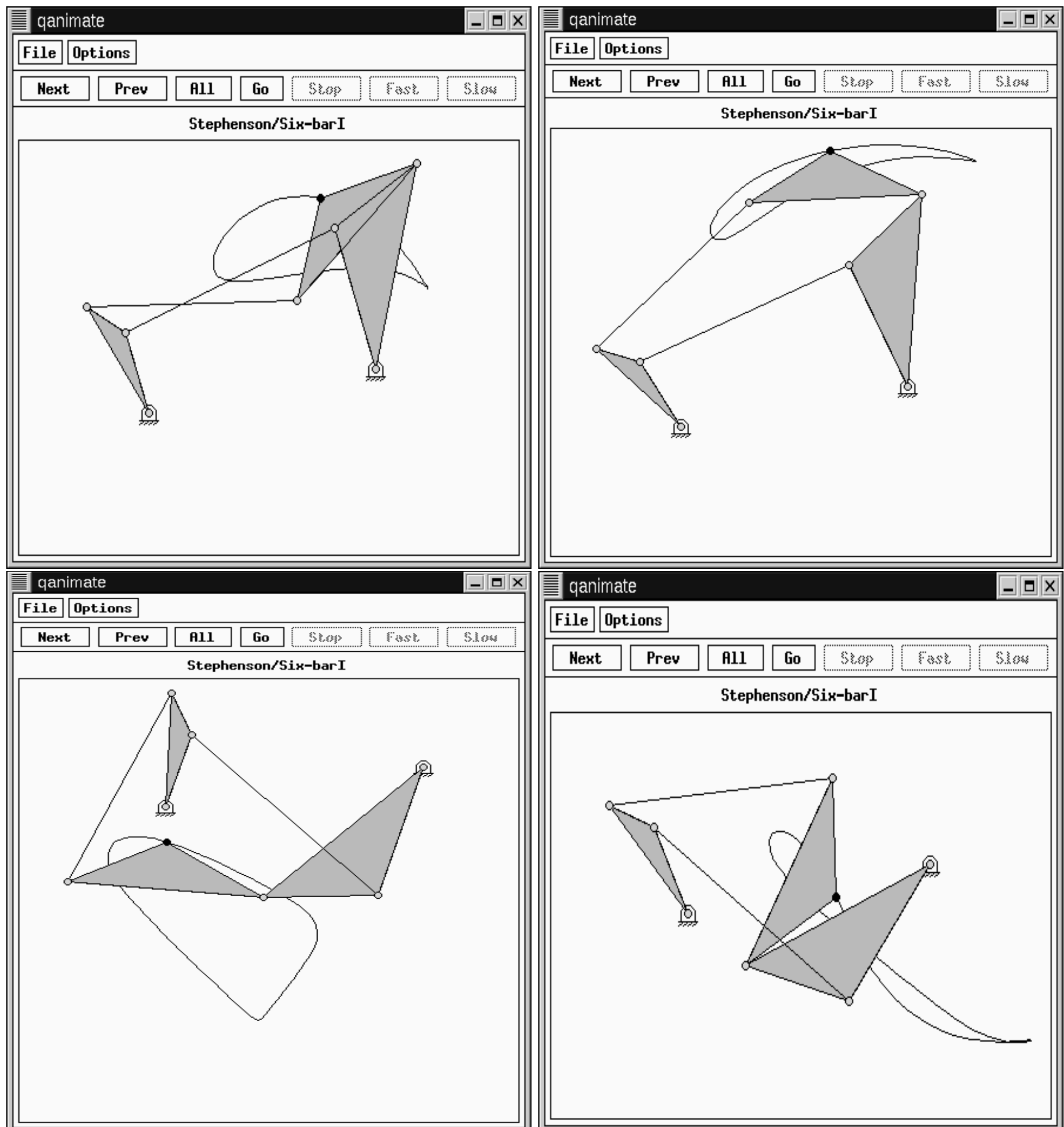
Description
This function simulates the motion of a Stevson Sixbar mechanism. *branchnum* is an integer number which indicates the branch you want to draw. *animationtype* is an optional parameter used to specify how the animation should be outputted. *animationtype* can be either of the following macros:
QANIMATE_OUTPUTTYPE_DISPLAY, QANIMATE_OUTPUTTYPE_FILE,
QANIMATE_OUTPUTTYPE_STREAM. QANIMATE_OUTPUTTYPE_DISPLAY displays an animation on the screen. QANIMATE_OUTPUTTYPE_FILE writes the animation data onto a file.
QANIMATE_OUTPUTTYPE_STREAM outputs the animation to the standard out. *datafilename* is an optional parameter to specify the file name of output if you want to output the data to a file.

Example
For a Stephenson Sixbar linkage with parameters $r_1 = 12m, r_2 = 4m, r_3 = 12m, r_4 = 7m, r_5 = 11m, r_6 = 9m, rp2 = 6m, rp4 = 10m, \theta_1 = 10^o$, $\omega_2 = 5rad/sec$, and $\alpha_2 = 0$, simulate the motion of the sixbar linkage. Also trace the curved generated by the motion of the coupler attached to link 6 with parameters $r_p = 5m$ and $\delta = 30^o$.

```
/***********************************************
 *  This example is for simulating a Stephenson  *
 *  (I) sixbar linkage.                          *
 ***********************************************/
#include<stdio.h>
#include<sixbar.h>

int main()
{
   double r[1:6];
   double theta_1[1:6], theta_2[1:6], theta_3[1:6], theta_4[1:6];
   double theta1, theta2;
   double omega2, alpha2;
   double rp2, rp4;
   double betaP2, betaP4, rp, delta;
   CStevSixbarI stevbar;
   int i;

   /* Specifications for the Stephenson I sixbar linkage. */
   r[1] = 12;  r[2] = 4;  r[3] = 12;  r[4] = 7;  r[5] = 11;  r[6] = 9;
   rp2 = 6; rp4 = 10;
   theta1 = 10*M_PI/180;  theta2 = 70*M_PI/180;
   betaP2 = 15*M_PI/180;  betaP4 = 30*M_PI/180;
   rp = 5;  delta = 30*M_PI/180;
   omega2 = 5;  alpha2 = 0;

   theta_1[1] = theta1;  theta_1[2] = theta2;
   theta_2[1] = theta1;  theta_2[2] = theta2;
   theta_3[1] = theta1;  theta_3[2] = theta2;
   theta_4[1] = theta1;  theta_4[2] = theta2;

   /* Perform analysis. */
```

```
    stevbar.setLinks(r, rp2, rp4, betaP2, betaP4, theta1);
    stevbar.setCouplerPoint(COUPLER_LINK6, rp, delta, TRACE_ON);
    stevbar.setNumPoints(50);
    stevbar.angularPos(theta_1, theta_2, theta_3, theta_4);
    stevbar.animation(1);
    stevbar.animation(2);
    stevbar.animation(3);
    stevbar.animation(4);

    return 0;
}
```

Output



CStevSixbarI::couplerPointAccel

Synopsis

#include <sixbar.h>
double complex couplerPointAccel(int *couplerLink*, **double** *theta[1:]*, **double** *omega[1:]*, **double** *alpha[1:]*);

Purpose
Calculate the acceleration of the coupler point.

Parameters

couplerLink An *int* value specifying a macro to indicate which link the coupler is attached to.

theta An array of *double* type used to store the angular position values of each link.

omega An array of *double* type used to store the angular velocity values of each link.

alpha An array of *double* type used to store the angular acceleration values of each link.

Return Value
The acceleration of the coupler point.

Description
This function calculates the coupler point acceleration. The return value is of type `double complex`.

Example
A Stephenson (I) Sixbar linkage has link lengths $r_1 = 12m, r_2 = 4m, r_3 = 12m, r_4 = 7m, r_5 = 11m, r_6 = 9m, rp2 = 6m, rp4 = 10m$, $\theta_1 = 10^o$, $\beta_2' = 15^o$, $\beta_4' = 30^o$, rp=5m, and $\delta = 30^o$. Given the angle θ_2, angular velocity ω_2 and angular acceleration α_2, calculate the coupler point acceleration for each circuit, respectively, if the coupler is attached to link 6.

```
/************************************************
 *  This example is for calculating the coupler  *
 *  point acceleration.                          *
 ************************************************/
#include<stdio.h>
#include<sixbar.h>

int main()
{
   double r[1:6];
   double theta_1[1:6], theta_2[1:6], theta_3[1:6], theta_4[1:6];
   double omega_1[1:6], omega_2[1:6], omega_3[1:6], omega_4[1:6];
   double alpha_1[1:6], alpha_2[1:6], alpha_3[1:6], alpha_4[1:6];
   double theta1, theta2, omega2, alpha2;
   double rp2, rp4;
   double betaP2, betaP4, rp, delta;
   double complex Ap[1:4];
   CStevSixbarI stevbar;
   int i;

   /* Specifications for the Stephenson I sixbar linkage. */
   r[1] = 12;  r[2] = 4;  r[3] = 12;  r[4] = 7;  r[5] = 11;  r[6] = 9;
   rp2 = 6; rp4 = 10;
   theta1 = 10*M_PI/180;  theta2 = 70*M_PI/180;
   betaP2 = 15*M_PI/180;  betaP4 = 30*M_PI/180;
   rp = 5;  delta = 30*M_PI/180;
```

```
    omega2 = 10*M_PI/180; /* rad/sec */
    alpha2 = 8*M_PI/180; /* rad/sec^2 */

    theta_1[1] = theta1;  theta_1[2] = theta2;
    theta_2[1] = theta1;  theta_2[2] = theta2;
    theta_3[1] = theta1;  theta_3[2] = theta2;
    theta_4[1] = theta1;  theta_4[2] = theta2;
    omega_1[2] = omega2;  alpha_1[2] = alpha2;
    omega_2[2] = omega2;  alpha_2[2] = alpha2;
    omega_3[2] = omega2;  alpha_3[2] = alpha2;
    omega_4[2] = omega2;  alpha_4[2] = alpha2;

    /* Perform analysis. */
    stevbar.setLinks(r, rp2, rp4, betaP2, betaP4, theta1);
    stevbar.setCouplerPoint(COUPLER_LINK6, rp, delta);
    stevbar.angularPos(theta_1, theta_2, theta_3, theta_4);
    stevbar.angularVel(theta_1, omega_1);
    stevbar.angularVel(theta_2, omega_2);
    stevbar.angularVel(theta_3, omega_3);
    stevbar.angularVel(theta_4, omega_4);
    stevbar.angularAccel(theta_1, omega_1, alpha_1);
    stevbar.angularAccel(theta_2, omega_2, alpha_2);
    stevbar.angularAccel(theta_3, omega_3, alpha_3);
    stevbar.angularAccel(theta_4, omega_4, alpha_4);
    Ap[1] = stevbar.couplerPointAccel(COUPLER_LINK6, theta_1, omega_1, alpha_1);
    Ap[2] = stevbar.couplerPointAccel(COUPLER_LINK6, theta_2, omega_2, alpha_2);
    Ap[3] = stevbar.couplerPointAccel(COUPLER_LINK6, theta_3, omega_3, alpha_3);
    Ap[4] = stevbar.couplerPointAccel(COUPLER_LINK6, theta_4, omega_4, alpha_4);

    /* Display the results. */
    printf("1st Solution Set:\n");
    printf("\t Ap = %.3f\n", Ap[1]);
    printf("2nd Solution Set:\n");
    printf("\t Ap = %.3f\n", Ap[2]);
    printf("3rd Solution Set:\n");
    printf("\t Ap = %.3f\n", Ap[3]);
    printf("4th Solution Set:\n");
    printf("\t Ap = %.3f\n", Ap[4]);

    return 0;
}
```

Output

```
1st Solution Set:
 Ap = complex(-0.686,0.431)
2nd Solution Set:
 Ap = complex(-0.880,0.115)
3rd Solution Set:
 Ap = complex(-0.661,0.123)
4th Solution Set:
 Ap = complex(-0.447,0.687)
```

CStevSixbarI::couplerPointPos

Synopsis
#include <sixbar.h>

void couplerPointPos(**int** *couplerLink*, **double** *theta2*, **double complex** &*p[1:4]*);

Purpose
Calculate the position of the coupler point.

Parameters

couplerLink An *int* value specifying a macro to indicate which link the coupler is attached to.

theta2 A double number used for the input angle of link.

p[1:4] A double complex array with size 4 for the solutions of the coupler point.

Return Value
None.

Description
This function calculates the position of the coupler point. *couplerLink* is a macro specifies the link with the coupler. *theta2* is the input angle. *p[1:4]* are the four solutions of the coupler point position, respectively, each of which is a complex number indicates the vector of the coupler point.

Example
A Stephenson (I) Sixbar linkage has link lengths $r_1 = 12m, r_2 = 4m, r_3 = 12m, r_4 = 7m, r_5 = 11m, r_6 = 9m, rp2 = 6m, rp4 = 10m$, $\theta_1 = 10^o$, $\beta_2' = 15^o$, $\beta_4' = 30^o$, rp=5m, and $\delta = 30^o$. Given the angle θ_2 and the fact that the coupler is attached to link 6, calculate the position of the coupler point for each circuit, respectively.

```
/************************************************
 *  This example is for calculating the position  *
 *  of the coupler point.                         *
 ************************************************/
#include<stdio.h>
#include<sixbar.h>

int main()
{
   double r[1:6];
   double theta_1[1:6], theta_2[1:6], theta_3[1:6], theta_4[1:6];
   double theta1, theta2;
   double rp2, rp4;
   double betaP2, betaP4, rp, delta;
   double complex P[1:4];
   CStevSixbarI stevbar;
   int i;

   /* Specifications for the Stephenson I sixbar linkage. */
   r[1] = 12;  r[2] = 4;  r[3] = 12;  r[4] = 7;  r[5] = 11;  r[6] = 9;
   rp2 = 6; rp4 = 10;
   theta1 = 10*M_PI/180;  theta2 = 70*M_PI/180;
   betaP2 = 15*M_PI/180;  betaP4 = 30*M_PI/180;
   rp = 5;  delta = 30*M_PI/180;

   theta_1[1] = theta1;  theta_1[2] = theta2;
   theta_2[1] = theta1;  theta_2[2] = theta2;
```

```
   theta_3[1] = theta1;  theta_3[2] = theta2;
   theta_4[1] = theta1;  theta_4[2] = theta2;

   /* Perform analysis. */
   stevbar.setLinks(r, rp2, rp4, betaP2, betaP4, theta1);
   stevbar.setCouplerPoint(COUPLER_LINK6, rp, delta);
   stevbar.angularPos(theta_1, theta_2, theta_3, theta_4);
   stevbar.couplerPointPos(COUPLER_LINK6, theta2, P);

   /* Display the results. */
   printf("1st Solution Set:\n");
   printf("\t P1 = %.3f\n", P[1]);
   printf("2nd Solution Set:\n");
   printf("\t P2 = %.3f\n", P[2]);
   printf("3rd Solution Set:\n");
   printf("\t P3 = %.3f\n", P[3]);
   printf("4th Solution Set:\n");
   printf("\t P4 = %.3f\n", P[4]);

   return 0;
}
```

Output

```
1st Solution Set:
 P1 = complex(12.243,8.631)
2nd Solution Set:
 P2 = complex(13.561,14.371)
3rd Solution Set:
 P3 = complex(0.293,-1.905)
4th Solution Set:
 P4 = complex(9.646,-3.109)
```

CStevSixbarI::couplerPointVel

Synopsis

#include <sixbar.h>
double complex couplerPointVel(int *couplerLink*, **double** *theta[1:]*, **double** *omega[1:]*);

Purpose

Calculate the velocity of the coupler point.

Parameters

couplerLink An *int* value specifying a macro to indicate which link the coupler is attached to.

theta An array of *double* type used to store the angular position values of each link.

omega An array of *double* type used to store the angular velocity values of each link.

Return Value

The velocity of the coupler point.

Description

This function calculates the coupler point velocity. The return value is of type `double complex`.

Example

A Stephenson (I) Sixbar linkage has link lengths $r_1 = 12m, r_2 = 4m, r_3 = 12m, r_4 = 7m, r_5 = 11m, r_6 = 9m, rp2 = 6m, rp4 = 10m$, $\theta_1 = 10^o$, $\beta_2' = 15^o$, $\beta_4' = 30^o$, rp=5m, and $\delta = 30^o$. Given the angle θ_2 and angular velocity ω_2, calculate the coupler point velocity for each respective circuit if the coupler is attached to link 6.

```
/************************************************
 *  This example is for calculating the coupler  *
 *  point velocity.                              *
 ************************************************/
#include<stdio.h>
#include<sixbar.h>

int main()
{
   double r[1:6];
   double theta_1[1:6], theta_2[1:6], theta_3[1:6], theta_4[1:6];
   double omega_1[1:6], omega_2[1:6], omega_3[1:6], omega_4[1:6];
   double theta1, theta2, omega2;
   double rp2, rp4;
   double betaP2, betaP4, rp, delta;
   double complex Vp[1:4];
   CStevSixbarI stevbar;
   int i;

   /* Specifications for the Stephenson I sixbar linkage. */
   r[1] = 12;  r[2] = 4;  r[3] = 12;  r[4] = 7;  r[5] = 11;  r[6] = 9;
   rp2 = 6; rp4 = 10;
   theta1 = 10*M_PI/180;  theta2 = 70*M_PI/180;
   betaP2 = 15*M_PI/180;  betaP4 = 30*M_PI/180;
   rp = 5;  delta = 30*M_PI/180;
   omega2 = 10*M_PI/180; /* rad/sec */

   theta_1[1] = theta1;  theta_1[2] = theta2;
   theta_2[1] = theta1;  theta_2[2] = theta2;
   theta_3[1] = theta1;  theta_3[2] = theta2;
   theta_4[1] = theta1;  theta_4[2] = theta2;
   omega_1[2] = omega2;
   omega_2[2] = omega2;
   omega_3[2] = omega2;
   omega_4[2] = omega2;

   /* Perform analysis. */
   stevbar.setLinks(r, rp2, rp4, betaP2, betaP4, theta1);
   stevbar.setCouplerPoint(COUPLER_LINK6, rp, delta);
   stevbar.angularPos(theta_1, theta_2, theta_3, theta_4);
   stevbar.angularVel(theta_1, omega_1);
   stevbar.angularVel(theta_2, omega_2);
   stevbar.angularVel(theta_3, omega_3);
   stevbar.angularVel(theta_4, omega_4);
   Vp[1] = stevbar.couplerPointVel(COUPLER_LINK6, theta_1, omega_1);
   Vp[2] = stevbar.couplerPointVel(COUPLER_LINK6, theta_2, omega_2);
   Vp[3] = stevbar.couplerPointVel(COUPLER_LINK6, theta_3, omega_3);
   Vp[4] = stevbar.couplerPointVel(COUPLER_LINK6, theta_4, omega_4);
```

```
   /* Display the results. */
   printf("1st Solution Set:\n");
   printf("\t Vp = %.3f\n", Vp[1]);
   printf("2nd Solution Set:\n");
   printf("\t Vp = %.3f\n", Vp[2]);
   printf("3rd Solution Set:\n");
   printf("\t Vp = %.3f\n", Vp[3]);
   printf("4th Solution Set:\n");
   printf("\t Vp = %.3f\n", Vp[4]);

   return 0;
}
```

Output

```
1st Solution Set:
 Vp = complex(-0.736,0.614)
2nd Solution Set:
 Vp = complex(-0.910,0.190)
3rd Solution Set:
 Vp = complex(-0.979,0.368)
4th Solution Set:
 Vp = complex(-0.733,0.769)
```

CStevSixbarI::displayPosition

Synopsis
#include <sixbar.h>
int displayPosition(double *theta2*, **double** *theta3*, **double** *theta4*, **double** *theta5*, **double** *theta6*, ... /* [**int** *outputtype* [, [**char *** *filename*]] */);

Syntax
displayPosition(*theta2*, *theta3*, *theta4*, *theta5*, *theta6*)
displayPosition(*theta2*, *theta3*, *theta4*, *theta5*, *theta6*, *outputtype*)
displayPosition(*theta2*, *theta3*, *theta4*, *theta5*, *theta6*, *outputtype*, *filename*)

Purpose
Given θ_2, θ_3, θ_4, θ_5 and θ_6, display the current position of the Stephenson (I) sixbar linkage.

Parameters

theta2 θ_2 angle.

theta3 θ_3 angle.

theta4 θ_4 angle.

theta5 θ_5 angle.

theta6 θ_6 angle.

outputtype an optional argument for the output type.

filename an optional argument for the output file name.

Return Value

This function returns 0 on success and -1 on failure.

Description

Given θ_2, θ_3, θ_4, θ_5, and θ_6 display the current position of the Stephenson (I) sixbar linkage. `outputtype` is an optional parameter used to specify how the output should be handled. It may be one of the following macros: QANIMATE_OUTPUTTYPE_DISPLAY, QANIMATE_OUTPUTTYPE_FILE, QANIMATE_OUTPUTTYPE_STREAM. QANIMATE_OUTPUTTYPE_DISPLAY outputs the figure to the computer terminal. QANIMATE_OUTPUTTYPE_FILE writes the qanimate data onto a file. QANIMATE_OUTPUTTYPE_STREAM outputs the qanimate data to the standard out stream. `filename` is an optional parameter to specify the output file name.

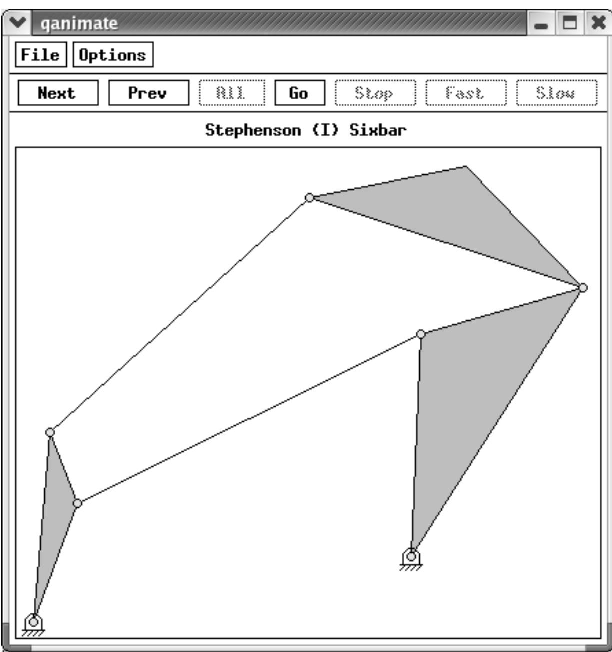
Example

A Stephenson (I) sixbar linkage has link lengths $r_1 = 12m, r_2 = 4m, r_3 = 12m, r_4 = 7m, r_5 = 6m, r_6 = 9m$, $\theta_1 = 10^o$, $r'_2 = 6m$, $\beta'_2 = 15^o$, $r'_4 = 10m$, $\beta'_4 = 30^o$, and coupler properties $r_p = 5m$ and $\delta = 30^o$ attached to link 6. Given the angle θ_2, display the Stephenson (I) sixbar linkage in its current position for one branch.

```
/*************************************************
 *  This example is for displaying the position   *
 *  of the Stephenson (I) sixbar linkage.         *
 *************************************************/
#include<stdio.h>
#include<sixbar.h>

int main()
{
   double r[1:6];
   double theta_1[1:6], theta_2[1:6], theta_3[1:6], theta_4[1:6];
   double theta1, theta2;
   double rp2, rp4;
   double betaP2, betaP4, rp, delta;
   double complex P[1:4];
   CStevSixbarI stevbar;
   int i;

   /* Specifications for the Stephenson I sixbar linkage. */
   r[1] = 12;  r[2] = 4;  r[3] = 12;  r[4] = 7;  r[5] = 11;  r[6] = 9;
   rp2 = 6; rp4 = 10;
   theta1 = 10*M_PI/180;  theta2 = 70*M_PI/180;
   betaP2 = 15*M_PI/180;  betaP4 = 30*M_PI/180;
   rp = 5;  delta = 30*M_PI/180;

   theta_1[1] = theta1;  theta_1[2] = theta2;
   theta_2[1] = theta1;  theta_2[2] = theta2;
   theta_3[1] = theta1;  theta_3[2] = theta2;
   theta_4[1] = theta1;  theta_4[2] = theta2;

   /* Perform analysis. */
   stevbar.setLinks(r, rp2, rp4, betaP2, betaP4, theta1);
   stevbar.setCouplerPoint(COUPLER_LINK6, rp, delta);
   stevbar.angularPos(theta_1, theta_2, theta_3, theta_4);
   stevbar.displayPosition(theta_2[2], theta_2[3], theta_2[4], theta_2[5], theta_2[6]);
```

```
    return 0;
}
```

Output



CStevSixbarI::setCouplerPoint

Synopsis
#include <sixbar.h>
void setCouplerPoint(int *couplerLink*, **double** *rp, delta*, ... [**int** *trace*] */);

Syntax
setCouplerPoint(*couplerLink*, *rp*, *beta*)
setCouplerPoint(*couplerLink*, *rp*, *beta*, *trace*)

Purpose
Set parameters for the coupler point.

Parameters

couplerLink An *int* value specifying a macro to indicate which link the coupler is attached to.

rp A double number used for the link length connected to the coupler point.

delta A double number specifying the angular position of the coupler point relative to the link it is attached to.

it trace An optional parameter of **int** type specifying either macro TRACE_OFF or TRACE_ON to indicate whether the coupler curve should be traced during animation.

Return Value
None.

Description
This function sets the parameters for the coupler point.

Example
see **CStevSixbarI::couplerPointPos**().

CStevSixbarI::setAngularVel

Synopsis
#include <sixbar.h>
void setAngularVel(double *omega2*);

Purpose
Set the constant angular velocity of link2.

Parameters

omega2 A double number used for the constant angular velocity of link2.

Return Value
None.

Description
This function sets the constant angular velocity of link2.

Example
see **CStevSixbarI::angularVel**().

CStevSixbarI::setLinks

Synopsis
#include <sixbar.h>
void setLinks(double *r[1:6], rp2, rp4, betaP2, betaP4, theta1*);

Purpose
Set the lengths of the links.

Parameters

r[6] A double array used for the lengths of the links.

theta1 A double number for the angle between link1 and horizontal.

rp2 A double number for the length of r'_2.

rp4 A double number for the length of r'_4.

betaP2 A double number for the angle between link r'_2 and horizontal.

betaP4 A double number for the angle between link r'_4 and horizontal.

Return Value

Description
This function sets the lengths of the links, including r'_2 and r'_4, and angles θ_1, β'_2 and β'_4.

Example
see **CStevSixbarI::angularVel()**.

CStevSixbarI::setNumPoints

Synopsis
#include <sixbar.h>
void setNumPoints(int *numpoints*);

Purpose
Set number of points for animation.

Parameters

numpoints An integer number used for the number of points.

Return Value
None.

Description
This function sets number of points for animation.

Example
see **CStevSixbarI::animation()**.

CStevSixbarI::uscUnit

Synopsis
#include <sixbar.h>
void uscUnit(bool *unit*);

Purpose
Specify the use of SI or US Customary units in analysis.

Parameters

unit A boolean argument, where `true` indicates US Customary units are desired and `false` indicates SI units.

Return Value
None.

Description

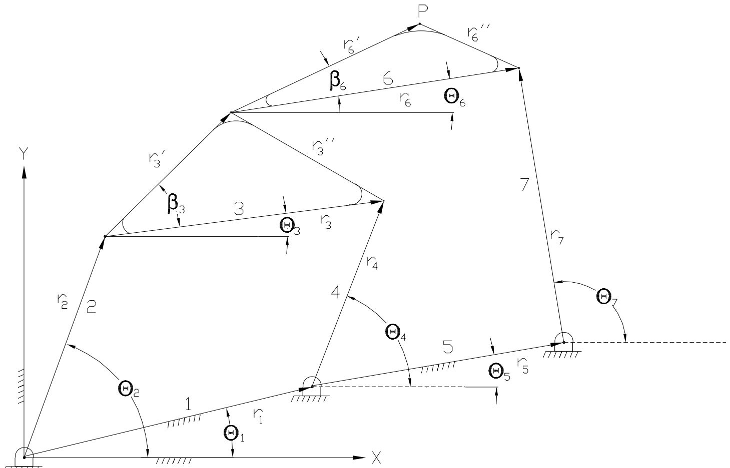
This function specifies the whether SI or US Customary units are used.If `unit = true`, then US Customary units are used; otherwise, SI units are used. By default, SI units are assumed. This member function shall be called prior any other member function calls.

Appendix I

Class CStevSixbarIII

CStevSixbarIII

The header file **sixbar.h** includes header file **linkage.h**. The header file **sixbar.h** also contains a declaration of class **CStevSixbarIII**. The **CStevSixbarIII** class provides a means to analyze Stephenson (III) Sixbar linkage within a Ch language environment.



Public Data
None.

Public Member Functions

Functions	Descriptions
angularAccel	Given the angular acceleration of one link, calculate the angular acceleration of other links.
angularPos	Given the angle of one link, calculate the angle of other links.
angularVel	Given the angular velocity of one link, calculate the angular velocity of other links.
animation	Stephenson (III) linkage animation.

couplerPointAccel	Calculate the coupler point acceleration.
couplerPointPos	Calculate the position of the coupler point.
couplerPointVel	Calculate the coupler point velocity.
displayPosition	Display the position of the Stephenson (III) linkage.
setCouplerPoint	Set parameters for the coupler point.
setAngularVel	Set constant angular velocity of linkage 2.
setLinks	Set lengths of links.
setNumPoints	Set number of points for animation.
uscUnit	Specify the use of SI or US Customary units.

See Also

CStevSixbarIII::angularAccel

Synopsis
#include <sixbar.h>
void angularAccel(double *theta[1:7]*, **double** *omega[1:7]*, **double** *alpha[1:7]*);

Purpose
Given the angular acceleration of the input link, calculate the angular acceleration of other links.

Parameters

theta An array of double data type with angles of links.

omega An array of double data type with angular velocities of links.

alpha An array of double data type with angular accelerations of links.

Return Value
None.

Description
Given the angular acceleration of the input link, this function calculates the angular acceleration of the remaining moving links of the Stephenson (III) sixbar. *theta* is a one-dimensional array of size 7 which stores the angle of each link. *omega* is a one-dimensional array of size 7 which stores the angular velocity of each link. *alpha* is a one-dimensional array of size 7 which stores the angular acceleration of each link. The results of the calculation are stored in array *alpha*.

Example
A Stephenson (III) sixbar linkage has parameters $r_1 = 9m, r_2 = 4m, r_3 = 10m, r_4 = 6m, r_5 = 8m, r_6 = 9m, r_7 = 12m$, $r_3' = 3m$, $\beta_3 = 20^o$, $\theta_1 = 0^o$, $\theta_5 = 15^o$, $\theta_2 = 25^o$, $\beta = 30^o$, and $rp = 5m$. Given the angle θ_2, the angular velocity ω_2 and the angular acceleration α_2, calculate the angular acceleration of the other links.

```
#include<stdio.h>
#include<sixbar.h>
```

```
int main()
{
   double r[1:7];
   double rP3, beta3,
          rp, beta;
   double theta[1:4][1:7], omega[1:4][1:7], alpha[1:4][1:7];
   double theta1 = 0, theta5 = M_DEG2RAD(15), theta2 = M_DEG2RAD(25);
   double omega2 = M_DEG2RAD(10), alpha2 = 0;
   CStevSixbarIII stevIII;
   int i;

   /* Define Stephenson (III) Sixbar linkage. */
   r[1] = 9;  r[2] = 4;
   r[3] = 10;  r[4] = 6;
   r[5] = 8;   r[6] = 9;
   r[7] = 12;
   rP3 = 3;  beta3 = M_DEG2RAD(20);
   rp = 5;  beta = M_DEG2RAD(30);
   for(i = 1; i <= 4; i++)
   {
      theta[i][1] = theta1;
      theta[i][2] = theta2;
      theta[i][5] = theta5;
      omega[i][2] = omega2;
   }

   /* Perform analysis. */
   stevIII.setLinks(r, rP3, beta3, theta1, theta5);
   stevIII.setCouplerPoint(rp, beta);
   stevIII.angularPos(theta[1], theta[2], theta[3], theta[4]);
   stevIII.angularVel(theta[1], omega[1]);
   stevIII.angularVel(theta[2], omega[2]);
   stevIII.angularVel(theta[3], omega[3]);
   stevIII.angularVel(theta[4], omega[4]);
   stevIII.angularAccel(theta[1], omega[1], alpha[1]);
   stevIII.angularAccel(theta[2], omega[2], alpha[2]);
   stevIII.angularAccel(theta[3], omega[3], alpha[3]);
   stevIII.angularAccel(theta[4], omega[4], alpha[4]);

   /* Display results. */
   for(i = 1; i <= 4; i++)
   {
      printf("Solution #%d:\n", i);
      printf("\t alpha3 = %.3f rad/s^2 (%.2f deg/s^2), alpha4 = %.3f rad/s^2 (%.2f deg/s^2),\n",
             alpha[i][3], M_RAD2DEG(alpha[i][3]),
             alpha[i][4], M_RAD2DEG(alpha[i][4]));
      printf("\t alpha6 = %.3f rad/s^2 (%.2f deg/s^2), alpha7 = %.3f rad/s^2 (%.2f deg/s^2)\n",
             alpha[i][6], M_RAD2DEG(alpha[i][6]),
             alpha[i][7], M_RAD2DEG(alpha[i][7]));
   }

   return 0;
}
```

Output

```
Solution #1:
 alpha3 = 0.027 rad/s^2 (1.55 deg/s^2), alpha4 = 0.047 rad/s^2 (2.72 deg/s^2),
```

```
 alpha6 = -0.017 rad/s^2 (-0.95 deg/s^2), alpha7 = 0.006 rad/s^2 (0.33 deg/s^2)
Solution #2:
 alpha3 = 0.027 rad/s^2 (1.55 deg/s^2), alpha4 = 0.047 rad/s^2 (2.72 deg/s^2),
 alpha6 = 0.018 rad/s^2 (1.03 deg/s^2), alpha7 = -0.004 rad/s^2 (-0.25 deg/s^2)
Solution #3:
 alpha3 = -0.027 rad/s^2 (-1.54 deg/s^2), alpha4 = -0.074 rad/s^2 (-4.25 deg/s^2),
 alpha6 = -0.014 rad/s^2 (-0.78 deg/s^2), alpha7 = 0.012 rad/s^2 (0.70 deg/s^2)
Solution #4:
 alpha3 = -0.027 rad/s^2 (-1.54 deg/s^2), alpha4 = -0.074 rad/s^2 (-4.25 deg/s^2),
 alpha6 = 0.026 rad/s^2 (1.52 deg/s^2), alpha7 = 0.001 rad/s^2 (0.03 deg/s^2)
```

CStevSixbarIII::angularPos

Synopsis
#include <sixbar.h>
void angularPos(double *theta_1[1:7]*, **double** *theta_2[1:7]*, **double** *theta_3[1:7]*, **double** *theta_4[1:7]*);

Purpose
Given the angle of input link, calculate the angle of other links.

Parameters

theta_1 A double array with dimension size of 7 for the first solution.

theta_2 A double array with dimension size of 7 for the second solution.

theta_3 A double array with dimension size of 7 for the third solution.

theta_4 A double array with dimension size of 7 for the fourth solution.

Return Value
None.

Description
Given the angular position of the input link, this function computes the angular positions of the remaining moving links. *theta_1* is a one-dimensional array of size 7 which stores the first solution of angular. *theta_2* is a one-dimensional array of size 7 which stores the second solution of angular. *theta_3* is a one-dimensional array of size 7 which stores the third solution of angular. *theta_4* is a one-dimensional array of size 7 which stores the fourth solution of angular.

Example
A Stephenson (III) Sixbar linkage has parameters $r_1 = 9m, r_2 = 4m, r_3 = 10m, r_4 = 6m, r_5 = 8m, r_6 = 9m, r_7 = 12m, r'_3 = 3m, \theta_1 = 0^o, \theta_5 = 15^o, \theta_2 = 15^o, \beta_3 = 20^o, rp = 5m, and \beta = 30^o$. Given the angle θ_2, calculate the angular positions of the other links.

```
#include<stdio.h>
#include<sixbar.h>

int main()
{
   double r[1:7];
   double rP3, beta3,
```

```
          rp, beta;
   double theta[1:4][1:7];
   double theta1 = 0, theta5 = M_DEG2RAD(15), theta2 = M_DEG2RAD(25);
   CStevSixbarIII stevIII;
   int i;

   /* Define Stephenson (III) Sixbar linkage. */
   r[1] = 9;  r[2] = 4;
   r[3] = 10;  r[4] = 6;
   r[5] = 8;   r[6] = 9;
   r[7] = 12;
   rP3 = 3;  beta3 = M_DEG2RAD(20);
   rp = 5;  beta = M_DEG2RAD(30);
   for(i = 1; i <= 4; i++)
   {
      theta[i][1] = theta1;
      theta[i][2] = theta2;
      theta[i][5] = theta5;
   }

   /* Perform analysis. */
   stevIII.setLinks(r, rP3, beta3, theta1, theta5);
   stevIII.setCouplerPoint(rp, beta);
   stevIII.angularPos(theta[1], theta[2], theta[3], theta[4]);

   /* Display results. */
   for(i = 1; i <= 4; i++)
   {
      printf("Solution #%d:\n", i);
      printf("\t theta3 = %.3f radians (%.2f degrees), theta4 = %.3f radians (%.2f degrees),\n",
             theta[i][3], M_RAD2DEG(theta[i][3]),
             theta[i][4], M_RAD2DEG(theta[i][4]));
      printf("\t theta6 = %.3f radians (%.2f degrees), theta7 = %.3f radians (%.2f degrees)\n",
             theta[i][6], M_RAD2DEG(theta[i][6]),
             theta[i][7], M_RAD2DEG(theta[i][7]));
   }

   return 0;
}
```

Output

```
Solution #1:
 theta3 = 0.251 radians (14.37 degrees), theta4 = 0.769 radians (44.05 degrees),
 theta6 = 1.177 radians (67.41 degrees), theta7 = 2.211 radians (126.69 degrees)
Solution #2:
 theta3 = 0.251 radians (14.37 degrees), theta4 = 0.769 radians (44.05 degrees),
 theta6 = -1.423 radians (-81.51 degrees), theta7 = -2.457 radians (-140.78 degrees)
Solution #3:
 theta3 = -0.860 radians (-49.28 degrees), theta4 = -1.378 radians (-78.97 degrees),
 theta6 = 1.479 radians (84.76 degrees), theta7 = 2.507 radians (143.64 degrees)
Solution #4:
 theta3 = -0.860 radians (-49.28 degrees), theta4 = -1.378 radians (-78.97 degrees),
 theta6 = -1.130 radians (-64.77 degrees), theta7 = -2.158 radians (-123.65 degrees)
```

CStevSixbarIII::angularVel

Synopsis
#include <sixbar.h>
void angularVel(double *theta[1:7],* **double** *omega[1:7]*);

Purpose
Given the angular velocity of the input link, calculate the angular velocity of other links.

Parameters

theta A double array used for the input angle of links.

omega A double array used for the angular velocities of links.

Return Value
None.

Description
Given the angular velocity of the input link, this function calculates the angular velocities of the remaining moving links of the sixbar. *theta* is an array for link positions. *omega* is an array for angular velocity of links.

Example
A Stephenson (III) Sixbar linkage has link lengths $r_1 = 9m, r_2 = 4m, r_3 = 10m, r_4 = 6m, r_5 = 8m, r_6 = 9m, r_7 = 12m, r'_3 = 3m, \theta_1 = 0^o, \theta_5 = 15^o, \theta_2 = 25^o, \beta_3 = 20^o, rp = 5m, and \beta = 30^o$. Given the angle θ_2 and the angular velocity ω_2, determine the angular velocities of the other links.

```
#include<stdio.h>
#include<sixbar.h>

int main()
{
   double r[1:7];
   double rP3, beta3,
          rp, beta;
   double theta[1:4][1:7], omega[1:4][1:7];
   double theta1 = 0, theta5 = M_DEG2RAD(15), theta2 = M_DEG2RAD(25);
   double omega2 = M_DEG2RAD(10);
   CStevSixbarIII stevIII;
   int i;

   /* Define Stephenson (III) Sixbar linkage. */
   r[1] = 9;  r[2] = 4;
   r[3] = 10;  r[4] = 6;
   r[5] = 8;   r[6] = 9;
   r[7] = 12;
   rP3 = 3;  beta3 = M_DEG2RAD(20);
   rp = 5;  beta = M_DEG2RAD(30);
   for(i = 1; i <= 4; i++)
   {
      theta[i][1] = theta1;
      theta[i][2] = theta2;
      theta[i][5] = theta5;
      omega[i][2] = omega2;
   }

   /* Perform analysis. */
```

```
   stevIII.setLinks(r, rP3, beta3, theta1, theta5);
   stevIII.setCouplerPoint(rp, beta);
   stevIII.angularPos(theta[1], theta[2], theta[3], theta[4]);
   stevIII.angularVel(theta[1], omega[1]);
   stevIII.angularVel(theta[2], omega[2]);
   stevIII.angularVel(theta[3], omega[3]);
   stevIII.angularVel(theta[4], omega[4]);

   /* Display results. */
   for(i = 1; i <= 4; i++)
   {
      printf("Solution #%d:\n", i);
      printf("\t omega3 = %.3f rad/s (%.2f deg/s), omega4 = %.3f rad/s (%.2f deg/s),\n",
             omega[i][3], M_RAD2DEG(omega[i][3]),
             omega[i][4], M_RAD2DEG(omega[i][4]));
      printf("\t omega6 = %.3f rad/s (%.2f deg/s), omega7 = %.3f rad/s (%.2f deg/s)\n",
             omega[i][6], M_RAD2DEG(omega[i][6]),
             omega[i][7], M_RAD2DEG(omega[i][7]));
   }

   return 0;
}
```

Output

```
Solution #1:
 omega3 = -0.046 rad/s (-2.64 deg/s), omega4 = 0.043 rad/s (2.48 deg/s),
 omega6 = -0.055 rad/s (-3.12 deg/s), omega7 = -0.055 rad/s (-3.16 deg/s)
Solution #2:
 omega3 = -0.046 rad/s (-2.64 deg/s), omega4 = 0.043 rad/s (2.48 deg/s),
 omega6 = -0.056 rad/s (-3.18 deg/s), omega7 = -0.055 rad/s (-3.15 deg/s)
Solution #3:
 omega3 = -0.137 rad/s (-7.84 deg/s), omega4 = -0.226 rad/s (-12.96 deg/s),
 omega6 = -0.047 rad/s (-2.70 deg/s), omega7 = -0.031 rad/s (-1.79 deg/s)
Solution #4:
 omega3 = -0.137 rad/s (-7.84 deg/s), omega4 = -0.226 rad/s (-12.96 deg/s),
 omega6 = -0.022 rad/s (-1.28 deg/s), omega7 = -0.038 rad/s (-2.19 deg/s)
```

CStevSixbarIII::animation

Synopsis
#include <sixbar.h>
void animation(int *branchnum*, ... /* [**int** *animationtype*, **string_t** *datafilename*] */);

Syntax
animation(*branchnum*)
animation(*branchnum*, *datafilename*)

Purpose
An animation of a Stephenson (III) Sixbar mechanism.

Parameters

branchnum an integer used for indicating which branch will be drawn.

it animationtype an optional parameter to specify the output type of the animation.

datafilename an optional parameter to specify the output file name.

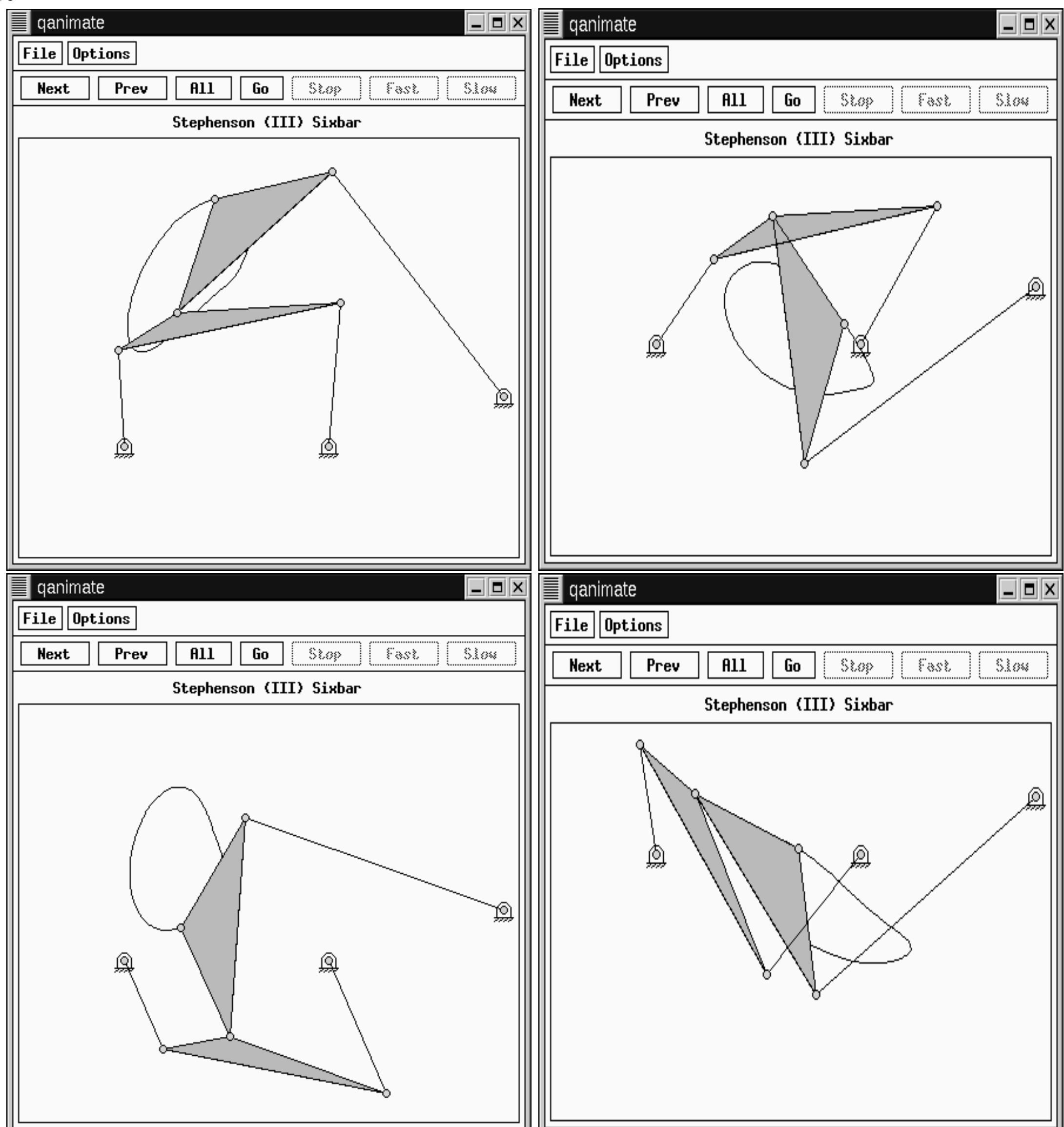
Return Value
None.

Description
This function simulates the motion of a Stephenson (III) Sixbar mechanism. *branchnum* is an integer number which indicates the branch you want to draw. *animationtype* is an optional parameter used to specify how the animation should be outputted. *animationtype* can be either of the following macros: QANIMATE_OUTPUTTYPE_DISPLAY, QANIMATE_OUTPUTTYPE_FILE, QANIMATE_OUTPUTTYPE_STREAM. QANIMATE_OUTPUTTYPE_DISPLAY displays an animation on the screen. QANIMATE_OUTPUTTYPE_FILE writes the animation data onto a file. QANIMATE_OUTPUTTYPE_STREAM outputs the animation to the standard out. *datafilename* is an optional parameter to specify the output file name.

Example
For a Stephenson (III) Sixbar linkage with parameters $r_1 = 9m, r_2 = 4m, r_3 = 10m, r_4 = 6m, r_5 = 8m, r_6 = 9m, r_7 = 12m, r'_3 = 6m, \beta_3 = 20^o, \theta_1 = 0^o, \theta_5 = 15^o$, $\omega_2 = 5rad/sec$, and $\alpha_2 = 0$, simulate the motion of the sixbar linkage.

```
#include<stdio.h>
#include<sixbar.h>

#define NUMPOINTS 50

int main()
{
   double r[1:7];
   double rP3, beta3,
          rp, beta;
   double theta1 = 0, theta5 = M_DEG2RAD(15);
   CStevSixbarIII stevIII;
   int i;

   /* Define Stephenson (III) Sixbar linkage. */
   r[1] = 9;  r[2] = 4;
   r[3] = 10;  r[4] = 6;
   r[5] = 8;   r[6] = 9;
   r[7] = 12;
   rP3 = 3;  beta3 = M_DEG2RAD(20);
   rp = 5;  beta = M_DEG2RAD(30);

   /* Perform analysis. */
   stevIII.setLinks(r, rP3, beta3, theta1, theta5);
   stevIII.setCouplerPoint(rp, beta, TRACE_ON);
   stevIII.setNumPoints(NUMPOINTS);
   stevIII.animation(1);
   stevIII.animation(2, QANIMATE_OUTPUTTYPE_FILE, "tempdata");
   stevIII.animation(3);
   stevIII.animation(4);

   return 0;
}
```

Output



CStevSixbarIII::couplerPointAccel

Synopsis
#include <sixbar.h>
double complex couplerPointAccel(double *theta[1:7]*, **double** *omega[1:7]*, **double** *alpha[1:7]*);

Purpose
Calculate the acceleration of the coupler point.

Parameters

theta An array of type *double* used to store the angular positions of the links.

omega An array of type *double* used to store the angular velocities of the links.

alpha An array of type *double* used to store the angular accelerations of the links.

Return Value

The acceleration of the coupler point.

Description

This function calculates the coupler point acceleration. The return value is of type `double complex`.

Example

A Stephenson (III) Sixbar linkage has link lengths $r_1 = 9m, r_2 = 4m, r_3 = 10m, r_4 = 6m, r_5 = 8m, r_6 = 9m, r_7 = 12m, r_3' = 3m$, $\theta_1 = 0^o$, $\theta_5 = 15^o$, $\beta_3 = 20^o$, rp=5m, and $\beta = 30^o$. Given the angle θ_2, angular velocity ω_2 and angular acceleration α_2, calculate the coupler point acceleration for each circuit, respectively.

```
#include<stdio.h>
#include<sixbar.h>

int main()
{
   double r[1:7];
   double rP3, beta3,
          rp, beta;
   double theta[1:4][1:7], omega[1:4][1:7], alpha[1:4][1:7];
   double theta1 = 0, theta5 = M_DEG2RAD(15), theta2 = M_DEG2RAD(25);
   double omega2 = M_DEG2RAD(10), alpha2 = 0;
   double complex Ap[1:4];
   CStevSixbarIII stevIII;
   int i;

   /* Define Stephenson (III) Sixbar linkage. */
   r[1] = 9;  r[2] = 4;
   r[3] = 10;  r[4] = 6;
   r[5] = 8;   r[6] = 9;
   r[7] = 12;
   rP3 = 3;  beta3 = M_DEG2RAD(20);
   rp = 5;  beta = M_DEG2RAD(30);
   for(i = 1; i <= 4; i++)
   {
      theta[i][1] = theta1;
      theta[i][2] = theta2;
      theta[i][5] = theta5;
      omega[i][2] = omega2;
   }

   /* Perform analysis. */
   stevIII.setLinks(r, rP3, beta3, theta1, theta5);
   stevIII.setCouplerPoint(rp, beta);
   stevIII.angularPos(theta[1], theta[2], theta[3], theta[4]);
   stevIII.angularVel(theta[1], omega[1]);
   stevIII.angularVel(theta[2], omega[2]);
   stevIII.angularVel(theta[3], omega[3]);
   stevIII.angularVel(theta[4], omega[4]);
   stevIII.angularAccel(theta[1], omega[1], alpha[1]);
   stevIII.angularAccel(theta[2], omega[2], alpha[2]);
   stevIII.angularAccel(theta[3], omega[3], alpha[3]);
   stevIII.angularAccel(theta[4], omega[4], alpha[4]);
   Ap[1] = stevIII.couplerPointAccel(theta[1], omega[1], alpha[1]);
```

```
   Ap[2] = stevIII.couplerPointAccel(theta[2], omega[2], alpha[2]);
   Ap[3] = stevIII.couplerPointAccel(theta[3], omega[3], alpha[3]);
   Ap[4] = stevIII.couplerPointAccel(theta[4], omega[4], alpha[4]);

   /* Display results. */
   for(i = 1; i <= 4; i++)
   {
      printf("Solution #%d:\n", i);
      printf("\t Ap = %.3f\n", Ap[i]);
   }

   return 0;
}
```

Output

```
Solution #1:
 Ap = complex(-0.077,0.008)
Solution #2:
 Ap = complex(-0.100,0.080)
Solution #3:
 Ap = complex(-0.132,-0.076)
Solution #4:
 Ap = complex(-0.125,0.016)
```

CStevSixbarIII::couplerPointPos

Synopsis
#include <sixbar.h>
void couplerPointPos(double *theta2*, **double complex** &*P[1:4]*);

Purpose
Calculate the position of the coupler point.

Parameters

theta2 A double number used for the input angle of link.

P[1:4] A double complex array with size 4 for the solutions of the coupler point.

Return Value
None.

Description
This function calculates the position of the coupler point. *theta2* is the input angle. *P[1:4]* are the four solutions of the coupler point position, respectively, each of which is a complex number indicates the vector of the coupler point.

Example
A Stephenson (III) Sixbar linkage has link lengths $r_1 = 9m, r_2 = 4m, r_3 = 10m, r_4 = 6m, r_5 = 8m, r_6 = 9m, r_7 = 11m, r_3' = 3m$, $\theta_1 = 0^o$, $\theta_5 = 15^o$, $\beta_3 = 20^o$, rp=5m, and $\beta = 30^o$. Given the angle θ_2, calculate the position of the coupler point for each circuit, respectively.

```
#include<stdio.h>
#include<sixbar.h>

int main()
{
   double r[1:7];
   double rP3, beta3,
          rp, beta;
   double theta[1:4][1:7];
   double theta1 = 0, theta5 = M_DEG2RAD(15), theta2 = M_DEG2RAD(25);
   double complex P[1:4];
   CStevSixbarIII stevIII;
   int i;

   /* Define Stephenson (III) Sixbar linkage. */
   r[1] = 9;  r[2] = 4;
   r[3] = 10;  r[4] = 6;
   r[5] = 8;   r[6] = 9;
   r[7] = 12;
   rP3 = 3;  beta3 = M_DEG2RAD(20);
   rp = 5;  beta = M_DEG2RAD(30);
   for(i = 1; i <= 4; i++)
   {
      theta[i][1] = theta1;
      theta[i][2] = theta2;
      theta[i][5] = theta5;
   }

   /* Perform analysis. */
   stevIII.setLinks(r, rP3, beta3, theta1, theta5);
   stevIII.setCouplerPoint(rp, beta);
   stevIII.angularPos(theta[1], theta[2], theta[3], theta[4]);
   stevIII.couplerPointPos(theta2, P);

   /* Display results. */
   for(i = 1; i <= 4; i++)
   {
      printf("Solution #%d:\n", i);
      printf("\t P = %.3f\n", P[i]);
   }

   return 0;
}
```

Output

```
Solution #1:
 P = complex(6.655,8.353)
Solution #2:
 P = complex(8.740,-0.863)
Solution #3:
 P = complex(5.117,5.095)
Solution #4:
 P = complex(9.754,-3.335)
```

CStevSixbarIII::couplerPointVel

Synopsis
#include <sixbar.h>
double complex couplerPointVel(double *theta[1:7]*, **double** *omega[1:7]*);

Purpose
Calculate the velocity of the coupler point.

Parameters

theta An array of type *double* that stores the angular positions of the links.

omega An array of type double that stores the angular velocities of the links .

Return Value
The velocity of the coupler point.

Description
This function calculates the coupler point velocity. The return value is of type `double complex`.

Example
A Stephenson (III) Sixbar linkage has link lengths $r_1 = 9m, r_2 = 4m, r_3 = 10m, r_4 = 6m, r_5 = 8m, r_6 = 9m, r_7 = 12m, r_3' = 3m$, $\theta_1 = 0^o$, $\theta_5 = 15^o$, $\beta_3 = 20^o$, rp=5m, and $\beta = 30^o$. Given the angle θ_2 and angular velocity ω_2, calculate the coupler point velocity for each respective circuit.

```
#include<stdio.h>
#include<sixbar.h>

int main()
{
   double r[1:7];
   double rP3, beta3,
          rp, beta;
   double theta[1:4][1:7], omega[1:4][1:7];
   double theta1 = 0, theta5 = M_DEG2RAD(15), theta2 = M_DEG2RAD(25);
   double omega2 = M_DEG2RAD(10);
   double complex Vp[1:4];
   CStevSixbarIII stevIII;
   int i;

   /* Define Stephenson (III) Sixbar linkage. */
   r[1] = 9;  r[2] = 4;
   r[3] = 10;  r[4] = 6;
   r[5] = 8;   r[6] = 9;
   r[7] = 12;
   rP3 = 3;  beta3 = M_DEG2RAD(20);
   rp = 5;  beta = M_DEG2RAD(30);
   for(i = 1; i <= 4; i++)
   {
      theta[i][1] = theta1;
      theta[i][2] = theta2;
      theta[i][5] = theta5;
      omega[i][2] = omega2;
   }

   /* Perform analysis. */
```

```
   stevIII.setLinks(r, rP3, beta3, theta1, theta5);
   stevIII.setCouplerPoint(rp, beta);
   stevIII.angularPos(theta[1], theta[2], theta[3], theta[4]);
   stevIII.angularVel(theta[1], omega[1]);
   stevIII.angularVel(theta[2], omega[2]);
   stevIII.angularVel(theta[3], omega[3]);
   stevIII.angularVel(theta[4], omega[4]);
   Vp[1] = stevIII.couplerPointVel(theta[1], omega[1]);
   Vp[2] = stevIII.couplerPointVel(theta[2], omega[2]);
   Vp[3] = stevIII.couplerPointVel(theta[3], omega[3]);
   Vp[4] = stevIII.couplerPointVel(theta[4], omega[4]);

   /* Display results. */
   for(i = 1; i <= 4; i++)
   {
      printf("Solution #%d:\n", i);
      printf("\t Vp = %.3f\n", Vp[i]);
   }

   return 0;
}
```

Output

```
Solution #1:
 Vp = complex(0.053,0.554)
Solution #2:
 Vp = complex(-0.435,0.346)
Solution #3:
 Vp = complex(-0.282,0.373)
Solution #4:
 Vp = complex(-0.559,0.183)
```

CStevSixbarIII::displayPosition

Synopsis

#include <sixbar.h>

int displayPosition(double *theta2*, **double** *theta3*, **double** *theta4*, **double** *theta6*, **double** *theta7*, ... /* [**int** *outputtype* [, [**char *** *filename*]] */);

Syntax

displayPosition(*theta2*, *theta3*, *theta4*, *theta6*, *theta7*)

displayPosition(*theta2*, *theta3*, *theta4*, *theta6*, *theta7*, *outputtype*)

displayPosition(*theta2*, *theta3*, *theta4*, *theta6*, *theta7*, *outputtype*, *filename*)

Purpose

Given θ_2, θ_3, θ_4, θ_6, and θ_7, display the current position of the Stephenson (III) sixbar linkage.

Parameters

theta2 θ_2 angle.

theta3 θ_3 angle.

theta4 θ_4 angle.

theta6 θ_6 angle.

theta7 θ_7 angle.

outputtype an optional argument for the output type.

filename an optional argument for the output file name.

Return Value
This function returns 0 on success and -1 on failure.

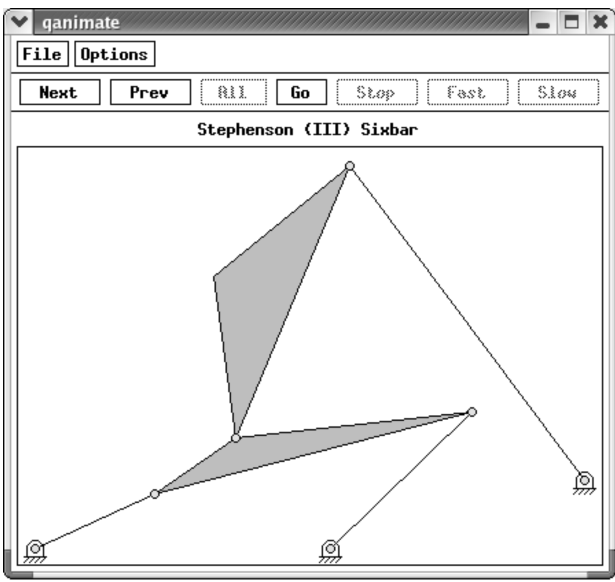
Description
Given θ_2, θ_3, θ_4, θ_6, and θ_7, display the current position of the Stephenson (II) sixbar linkage. `outputtype` is an optional parameter used to specify how the output should be handled. It may be one of the following macros: QANIMATE_OUTPUTTYPE_DISPLAY, QANIMATE_OUTPUTTYPE_FILE, QANIMATE_OUTPUTTYPE_STREAM. QANIMATE_OUTPUTTYPE_DISPLAY outputs the figure to the computer terminal. QANIMATE_OUTPUTTYPE_FILE writes the qanimate data onto a file. QANIMATE_OUTPUTTYPE_STREAM outputs the qanimate data to the standard out stream. `filename` is an optional parameter to specify the output file name.

Example
A Stephenson (III) sixbar linkage has link lengths $r_1 = 9m, r_2 = 4m, r_3 = 10m, r_4 = 6m, r_5 = 8m, r_6 = 9m, r_7 = 12m$, $\theta_1 = 10^o$, $r'_3 = 3m$, $\beta_3 = 20^o$, and coupler properties $r_p = 5m$ and $\beta = 30^o$. Given the angle θ_2, display the Stephenson (III) sixbar linkage in its current position for one branch.

```
#include<stdio.h>
#include<sixbar.h>

int main()
{
   double r[1:7];
   double rP3, beta3,
          rp, beta;
   double theta[1:4][1:7];
   double theta1 = 0, theta5 = M_DEG2RAD(15), theta2 = M_DEG2RAD(25);
   double complex P[1:4];
   CStevSixbarIII stevIII;
   int i;

   /* Define Stephenson (III) Sixbar linkage. */
   r[1] = 9;  r[2] = 4;
   r[3] = 10;  r[4] = 6;
   r[5] = 8;   r[6] = 9;
   r[7] = 12;
   rP3 = 3;  beta3 = M_DEG2RAD(20);
   rp = 5;  beta = M_DEG2RAD(30);
   for(i = 1; i <= 4; i++)
   {
      theta[i][1] = theta1;
      theta[i][2] = theta2;
      theta[i][5] = theta5;
   }
```

```
    /* Perform analysis. */
    stevIII.setLinks(r, rP3, beta3, theta1, theta5);
    stevIII.setCouplerPoint(rp, beta);
    stevIII.angularPos(theta[1], theta[2], theta[3], theta[4]);
    stevIII.displayPosition(theta[1][2], theta[1][3], theta[1][4], theta[1][6], theta[1][7]);

    return 0;
}
```

Output



CStevSixbarIII::setCouplerPoint

Synopsis
#include <sixbar.h>
void setCouplerPoint(double *rp, beta, ... /* [***int** *trace] */***);**

Syntax
setCouplerPoint(*couplerLink*, *rp*, *beta*)
setCouplerPoint(*couplerLink*, *rp*, *beta*, *trace*)

Purpose
Set parameters for the coupler point.

Parameters

rp A double number used for rp.

beta A double number for beta.

it trace An optional parameter of **int** type specifying either macro TRACE_OFF or TRACE_ON to indicate whether the coupler curve should be traced during animation.

Return Value
None.

Description
This function sets the parameters for the coupler point.

Example
see **CStevSixbarIII::couplerPointPos**().

CStevSixbarIII::setAngularVel

Synopsis
#include <sixbar.h>
void setAngularVel(double *omega2*);

Purpose
Set the constant angular velocity of link2.

Parameters

omega2 A double number used for the constant input angular velocity of link2.

Return Value
None.

Description
This function sets the constant angular velocity of link2.

Example
see **CStevSixbarIII::angularVel**().

CStevSixbarIII::setLinks

Synopsis
#include <sixbar.h>
void setLinks(double *r[1:7], rP3, beta3, theta1, theta5*);

Purpose
Set the lengths of the links.

Parameters

r A double array used for the lengths of the links.

rP3 A double number for the length of r_3'.

beta3 A double number for the angle between link r_3' and horizontal.

theta1 A double number for the angle between link1 and horizontal.

theta5 A double number for the angle between link5 and horizontal.

Return Value

Description
This function sets the lengths of the links, including r'_3 and angles θ_1, θ_5 and β_3.

Example
see **CStevSixbarIII::angularVel**().

CStevSixbarIII::setNumPoints

Synopsis
#include <sixbar.h>
void setNumPoints(int *numpoints*);

Purpose
Set number of points for animation.

Parameters

numpoints An integer number used for the number of points.

Return Value
None.

Description
This function sets number of points for animation.

Example
see **CStevSixbarIII::animation**().

CStevSixbarIII::uscUnit

Synopsis
#include <sixbar.h>
void uscUnit(bool *unit*);

Purpose
Specify the use of SI or US Customary units in analysis.

Parameters

unit A boolean argument, where `true` indicates US Customary units are desired and `false` indicates SI units.

Return Value
None.

Description

This function specifies the whether SI or US Customary units are used.If `unit = true`, then US Customary units are used; otherwise, SI units are used. By default, SI units are assumed. This member function shall be called prior any other member function calls.

Appendix J

Class CCam

CCam

CCam

The header file **cam.h** includes header file **linkage.h**. The header file **cam.h** also contains a declaration of class **CCam**. The **CCam** class can be used to design cams with translating or oscillating, flat-faced or roller followers, with either harmonic or cycloidal follower motion. The follower position, follower velocity, follower acceleration, and transmission angle for the system cam be plotted or returned to the user directly. In addition the cam/follower system can be animated and CNC code for manufacturing the cam can be generated and saved to a file.

Public Data
None.

Public Member Functions

Function	Description
CCam()	Class Constructor. Creates and initializes new instances of the class.
~**CCam**()	Class destructor. Frees memory associated with an instance of the class.
addSection()	Add a cam section to a previously declared instance of the cam class.
angularVel()	Set the cam angular velocity.
animation()	Display an animation of the cam.
baseRadius()	Set the cam base radius.
CNCCode()	Set the filename for CNC code output.
cutDepth()	Set the cut depth for CNC code generation.
cutter()	Set the cutter parameters for CNC code generation.
cutterOffset()	Set the cutter home position offset for CNC code generation.
deleteCam()	Remove data from a previously used instance of the **CCam** class.
feedrate()	Set the feedrate for CNC code generation.
followerType()	Set the cam follower type.
getCamAngle()	Get the cam angular position data.
getCamProfile()	Get the cam profile data.

getFollowerAccel()	Get the cam follower acceleration data.
getFollowerPos()	Get the cam follower position data.
getFollowerVel()	Get the cam follower velocity data.
getTransAngle()	Get the cam transmission angle data.
makeCam()	Generate the cam data and write CNC code to a file.
plotCamProfile()	Plot the cam profile data.
plotFollowerAccel()	Plot the cam follower acceleration vs. the cam angular position.
plotFollowerPos()	Plot the cam follower position vs. the cam angular position.
plotFollowerVel()	Plot the cam follower velocity vs. the cam angular position.
plotTransAngle()	Plot the cam transmission angle vs. the cam angular position.
spindleSpeed()	Set the spindle speed for CNC code generation.

Constants

Macro	Description
CAM_DURATION_FILL	The current cam section is used to fill the remaining duration up to 360 degrees. The *lift* parameter is ignored and the appropriate value is calculated internally to give a continuous profile.
CAM_FOLLOWER_OSC_FLAT	The cam has an oscillating flat-faced follower.
CAM_FOLLOWER_OSC_ROLL	The cam has an oscillating roller follower.
CAM_FOLLOWER_TRANS_FLAT	The cam has a translating flat-faced follower.
CAM_FOLLOWER_TRANS_ROLL	The cam has a translating roller follower.
CAM_MOTION_CYCLOIDAL	Cam displacement profile for the section is cycloidal.
CAM_MOTION_HARMONIC	Cam displacement profile for the section is harmonic.

References

Erdman, A. G. and Sandor, G. N., 1997, *Mechanism Design Analysis and Synthesis*, Vol. 1, 3rd ed., Prentice Hall, Englewood Cliffs, NJ.

CCam::addSection

Synopsis
#include <cam.h>
int addSection(double *duration*, **double** *displacement*, **int** *motion_type*);

Purpose
Add a cam section to a previously declared instance of the cam class.

Return Value
This function returns 0 on success and -1 on failure.

Parameters

duration The angular duration of the cam section in degree.

displacement The change in position of the cam follower. Positive away from the center of the cam, negative toward the center of the cam. Measured in inches for translating followers and in degrees for oscillating followers.

motion_type Valid values are:

CAM_MOTION_CYCLOIDAL Cam displacement profile for the section is cycloidal.

CAM_MOTION_HARMONIC Cam displacement profile for the section is harmonic.

Description
This function is called to add a section to a previously declared instance of the **CCam** class. The *duration* can be any positive number less than or equal to 360 or can be set to **CAM_DURATION_FILL**.
CAM_DURATION_FILL gives the current section a duration equal to 360 minus the sum of the previous section durations and should be used to specify the last section of a cam. The *displacement* is the change in the follower position over the *duration* of the section. this quantity is in inch for translating followers and in degree for oscillating followers. There are no restrictions placed on the value of this parameter, however an invalid value will result in an invalid cam profile. The *displacement* is selected automatically to give a continuous cam profile when *duration* is **CAM_DURATION_FILL**. The *motion_type* describes the shape of the displacement profile for the cam section. If *displacement* is zero, this parameter has no effect.

Algorithm
See Algorithm section in **CCam::makeCam**().

Example
See example in **CCam::plotCamProfile**().

CCam::angularVel

Synopsis
#include <cam.h>

void angularVel(double *omega*);

Purpose
Set the cam's angular velocity.

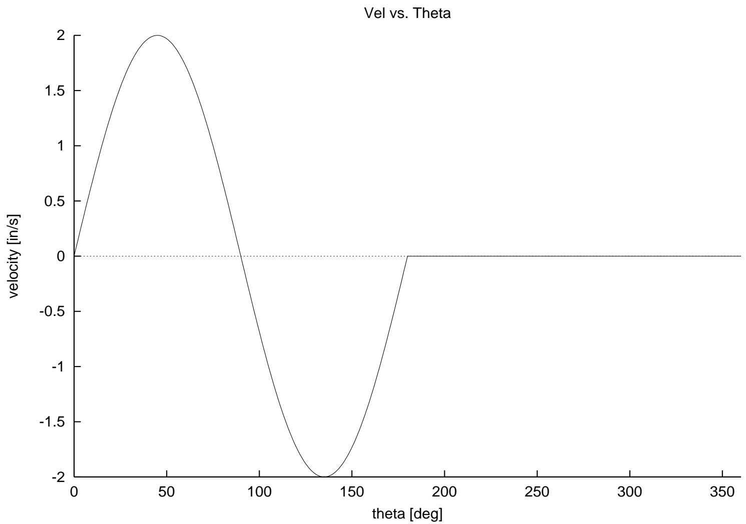
Return Value
None.

Parameters
omega The angular velocity of the cam in rad/s. Omega is positive for clockwise rotation of the cam.

Description
This function sets the angular velocity of the cam. The angular velocity, *omega*, is specified in rad/s. A positive value indicates that the cam is rotated clockwise. A negative value indicates counter-clockwise rotation. By default, the angular velocity is 1 rad/s.

Example
```
#include <cam.h>

int main() {
    class CCam cam;
    int steps = 360;

    cam.followerType(CAM_FOLLOWER_TRANS_FLAT, 0);
    cam.baseRadius(5);
    cam.angularVel(2);
    cam.addSection(90,  1, CAM_MOTION_HARMONIC);
    cam.addSection(90, -1, CAM_MOTION_HARMONIC);
    cam.addSection(CAM_DURATION_FILL, 0, CAM_MOTION_HARMONIC);
    cam.makeCam(steps);
    cam.plotFollowerVel(NULL);
}
```

Output



CCam::animation

Synopsis
#include <cam.h>
int animation(int *numframe*, ... /* [**int** outputtype, [**string_t** *filename*]] */);

Syntax
animation(*numframe*)
animation(*numframe*, *outputtype*)
animation(*numframe*, *outputtype*, *filename*)

Purpose
Display an animation of the cam.
Return Value
This function returns 0 on success and -1 on failure.

Parameters
numframe The number of different cam positions (frames) used in the animation.

outputtype An optional parameter to specify the output type of the animation.

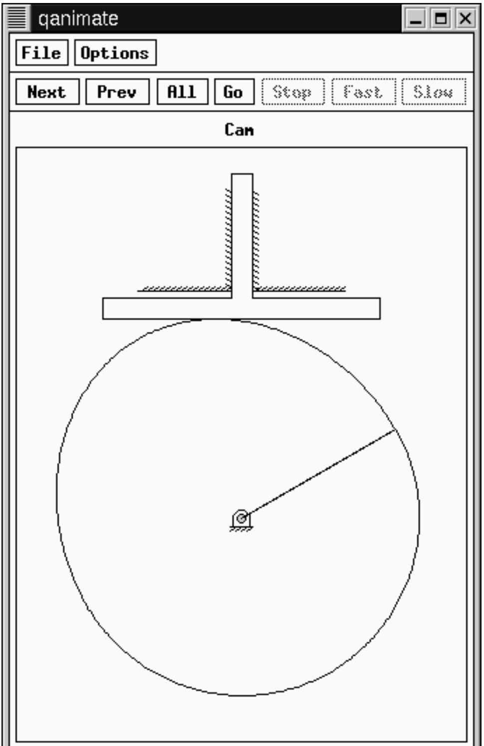
filename An optional parameter to specify the output file name.

Description
This function uses the **Quick Animation** program to display an animation of the cam. The value of *numframe* specifies the number of different cam positions (frames) used in the animation. Argument *outputtype* can be either of the following macros: `QANIMATE_OUTPUTTYPE_DISPLAY`, `QANIMATE_OUTPUTTYPE_STREAM`, and `QANIMATE_OUTPUTTYPE_FILE`. `QANIMATE_OUTPUTTYPE_DISPLAY` outputs the animation onto the computer terminal, `QANIMATE_OUTPUTTYPE_STREAM` outputs the animation to the standard output, and `QANIMATE_OUTPUTTYPE_FILE` writes the animation data onto a file. Optional argument `filename` is used to specify the file name to store the animation data. This function should only be called after **CCam::makeCam**().

Example

```
#include <cam.h>

int main() {

    class CCam cam;
    int steps = 360;

    cam.followerType(CAM_FOLLOWER_TRANS_FLAT);
    cam.baseRadius(5);
    cam.addSection(90,  1, CAM_MOTION_HARMONIC);
    cam.addSection(90, -1, CAM_MOTION_HARMONIC);
    cam.addSection(CAM_DURATION_FILL, 0, CAM_MOTION_HARMONIC);
    cam.makeCam(steps);
    cam.animation(12);
}
```

Output



CCam::baseRadius

Synopsis
#include <**cam.h**>
void baseRadius(**double** *base_radius*);

Purpose
Set the cam base radius.

Return Value
None.

Parameters
base_radius The base radius of the cam.

Description
The base radius of the cam is the initial radius of the cam before any calls to **CCam**::**addSection**(). By default, the base radius is 4.0 inches.

Example
See example in **CCam**::**plotCamProfile**().

See Also
CCam::addSection();

CCam::CNCCode

Synopsis
#include <cam.h>
void CNCCode(**string_t** *filename*);

Purpose
Set the filename for CNC code output.

Return Value
None.

Parameters
filename The name of the CNC code file to be created.

Description
This function sets the name of the CNC code file to be created. The file is created at the point when **CCam::makeCam**() is called.

Example

```
#include <cam.h>

int main() {
    class CCam cam;
    int steps = 360;

    cam.followerType(CAM_FOLLOWER_TRANS_FLAT, 0);
    cam.cutter(.25, 2, 2);
    cam.cutDepth(1.75);
    cam.feedrate(10.5);
    cam.spindleSpeed(2000);
    cam.cutterOffset(1.0, 1.0, 0);
    cam.addSection(90,  .5, CAM_MOTION_HARMONIC);
    cam.addSection(90, -.5, CAM_MOTION_HARMONIC);
    cam.addSection(CAM_DURATION_FILL, 0, CAM_MOTION_HARMONIC);
    cam.CNCCode("../output/CNCCode.txt");
    cam.makeCam(steps);
}
```

Output

See Also
CCam::makeCam();

References
Vickers, G. W., Ly, M. H. and Oetter, R. G., 1990, *Numerically Controlled Machine Tools*, Ellis Horwood, New York, NY.

CCam::cutDepth

Synopsis
#include <cam.h>
void cutDepth(double *cut_depth*);

Purpose
Set the cut depth for CNC code generation.

Return Value
None.

Parameters
cut_depth The depth of the cutter during cam manufacturing.

Description
This function sets the cutter depth to be used during cam manufacturing and is used during the creation of the CNC code. This parameter does not effect the shape of the cam profile and only needs to be set if CNC code will be generated. By default, the cut depth is 0.75 inch.

Example
See example in **CCam::CNCCode**().

See Also
CCam::CNCCode(), **CCam::cutter**(), **CCam::cutterOffset**(), **CCam::feedrate**(), and **CCam::spindleSpeed**().

CCam::cutter

Synopsis
#include <cam.h>
void cutter(double *cutter_radius*, **double** *cutter_length*, **int** *tool_num*);

Purpose
Set the cutter parameters for CNC code generation.

Return Value
None.

Parameters
cutter_radius The radius (in inches) of the cutter used for manufacturing of the cam.

cutter_length The length (in inches) of the cutter used for manufacturing the cam.

tool_num The tool number of the cutter used.

Description
This function sets the parameters for the cutter used to generate CNC code. Figure 6.5 illustrates the cutter parameters. The *cutter_length* should be less than the *depth* set in **CCam::cutDepth**(). The CNC machine to be used supports multiple tools, *tool_num* can be used to specify the correct cutter. These parameters do

not effect the shape of the cam profile and only need to be set if CNC code is to be created. By default the cutter radius is 0.125 in., the cutter length is 1.0 in., and the tool number is 1.

Example
See example in **CCam::CNCCode**().

See Also
CCam::CNCCode(), **CCam::cutDepth**(), **CCam::cutterOffset**(), **CCam::feedrate**(), and **CCam::spindleSpeed**().

CCam::cutterOffset

Synopsis
#include <cam.h>
void cutterOffset(**double** *x_offset*, **double** *y_offset*, **double** *z_offset*);

Purpose
Set the cutter home position offset for CNC code generation.

Return Value
None.

Parameters

x_offset The x coordinate of the cutter home position offset.

y_offset The y coordinate of the cutter home position offset.

z_offset The z coordinate of the cutter home position offset.

Description
The CNC home position offset may be changed if the CNC home position does not coincide with the desired location of the cam center. As shown in Figure 6.6, the home position offset is measured from the old home position to the new home position. It is important that these parameters be chosen properly, incorrect selection can cause damage to the tools and CNC machine. These parameters do not effect the shape of the cam profile and only need to be set if CNC code is to be created. By default, all offsets are zero.

Example
See example in **CCam::CNCCode**().

See Also
CCam::CNCCode(), **CCam::cutDepth**(), **CCam::cutter**(), **CCam::feedrate**(), and **CCam::spindleSpeed**().

CCam::deleteCam

Synopsis
#include <cam.h>
void deleteCam();

Purpose
Remove data from a previously used instance of the **CCam** class.

Return Value
None.

Parameters
None.

Description
This function frees all memory associated with previously allocated cam sections and data arrays. This function allows for reuse of a single instance of the **CCam** class to create multiple cams.

CCam::feedrate

Synopsis
#include <cam.h>
void feedrate(double *feedrate***);**

Purpose
Set the feedrate for CNC code generation.

Return Value
None.

Parameters
feedrate The feedrate in inches per minute for machining.

Description
The *feedrate*, in inches per minute, is the rate at which the workpiece is moved during machining. This parameter does not effect the shape of the cam profile and only needs to be set if CNC code is to be created. By default, the feedrate is 18.3 in/min.

Example
See example in **CCam::CNCCode()**.

See Also
CCam::CNCCode(), **CCam::cutDepth()**, **CCam::cutter()**, **CCam::cutterOffset()**, and **CCam::spindleSpeed()**.

CCam::followerType

Synopsis
#include <cam.h>
int followerType(int *follower_type*, ... /* [**double** *e*], [**double** *e*, **double** *rf*], [**double** *m*, **double** *f*],
[**double** *m*, **double** *A*, **double** *rf*] */);

Syntax
int followerType(CAM_FOLLOWER_TRANS_FLAT, double *e*)
int followerType(CAM_FOLLOWER_TRANS_ROLL, double *e*, **double** *rf*
int followerType(CAM_FOLLOWER_OSC_FLAT, double *m*, **double** *f*)
int followerType(CAM_FOLLOWER_OSC_ROLL, double *m*, **double** *A*, **double** *rf*)

Purpose
Set the cam follower type.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
e For a translating follower, the distance from the cam center to the line of follower motion, as shown in Figures 6.1 and 6.2. By default, *e* is zero.

rf The roller radius for a roller follower, as shown in Figures 6.2 and 6.4. By default, *rf* is 1.0 inch.

f For a flat-faced oscillating follower, the follower face offset measured from the follower pivot point, as showin in Figure 6.3. By default, *f* is 4.0 inches.

A For an oscillating roller follower, the length of the arm connecting the pivot point and roller center, as shown in Figure 6.4. By default, *A* is 10.0 inches.

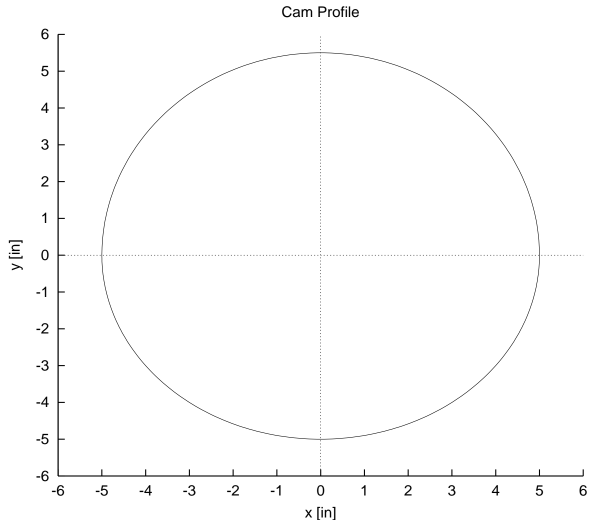
m For oscillating followers, the distance between the cam center and the follower pivot point, as shown in Figures 6.3 and 6.4. By default, *m* is 15.0 inches.

Description
This function is used to select the follower type and set the necessary related parameters. For a translating flat-faced follower, *e* is positive for an offset to the right of the cam. For a translating roller follower, *e* is positive for offsets to the left of the cam. For oscillating followers, *m* is positive when the follower pivot point is to the right of the cam. All dimensions are in inches. By default, the follower type is **CAM_FOLLOWER_TRANS_FLAT**.

Example 1

```
#include <cam.h>

int main() {

    class CCam cam;
    int steps = 360;

    cam.followerType(CAM_FOLLOWER_TRANS_FLAT);
    cam.addSection(90,  .5, CAM_MOTION_HARMONIC);
    cam.addSection(90, -.5, CAM_MOTION_HARMONIC);
    cam.addSection(CAM_DURATION_FILL, 0, CAM_MOTION_HARMONIC);
    cam.makeCam(steps);
    cam.plotCamProfile(NULL);
}
```

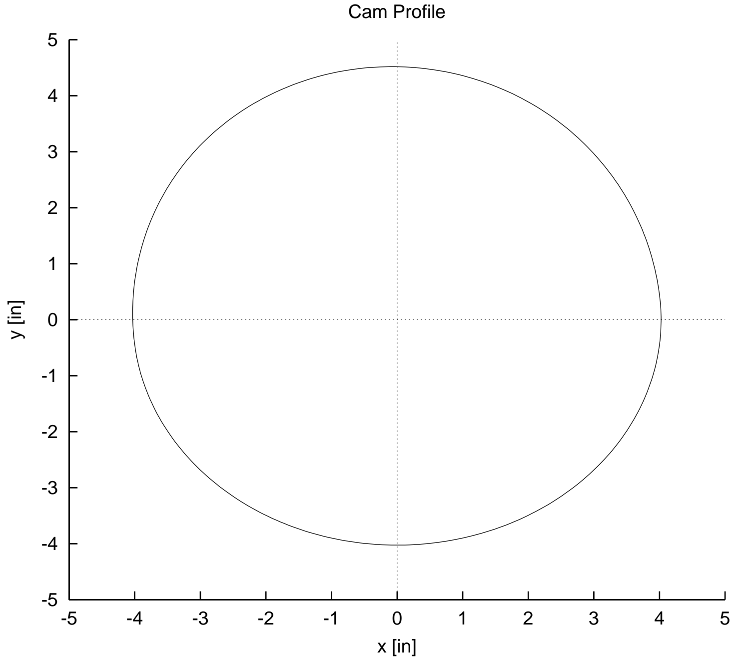
Output



Example 2

```
#include <cam.h>

int main() {

    class CCam cam;
    int steps = 360;

    cam.followerType(CAM_FOLLOWER_TRANS_ROLL, .5, 1);
    cam.addSection(90,  .5, CAM_MOTION_HARMONIC);
    cam.addSection(90, -.5, CAM_MOTION_HARMONIC);
    cam.addSection(CAM_DURATION_FILL, 0, CAM_MOTION_HARMONIC);
    cam.makeCam(steps);
    cam.plotCamProfile(NULL);

}
```

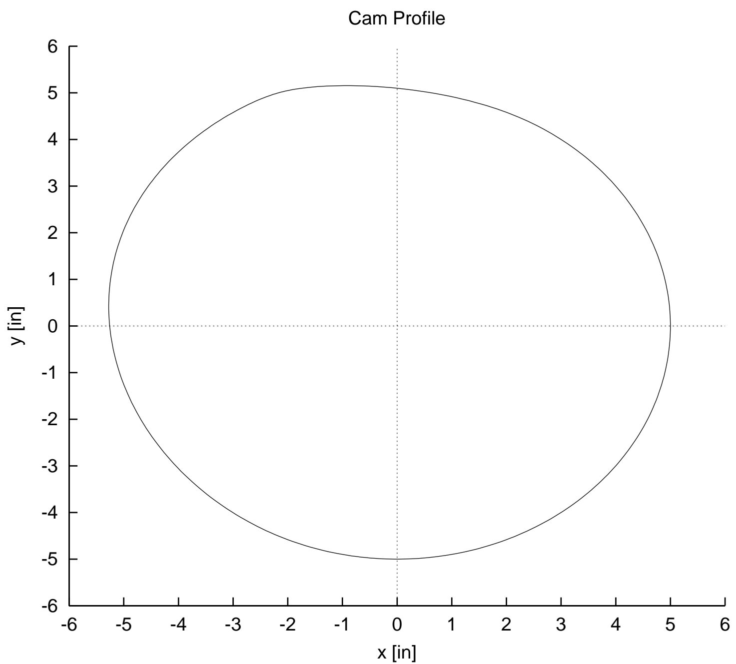
Output



Example 3

```
#include <cam.h>

int main() {

    class CCam cam;
    int steps = 360;

    cam.followerType(CAM_FOLLOWER_OSC_FLAT, 10, 1);
    cam.baseRadius(5);
    cam.addSection(60, 3, CAM_MOTION_CYCLOIDAL);
    cam.addSection(120, -3, CAM_MOTION_CYCLOIDAL);
    cam.addSection(CAM_DURATION_FILL, 0, CAM_MOTION_HARMONIC);
    cam.makeCam(steps);
    cam.plotCamProfile(NULL);
}
```

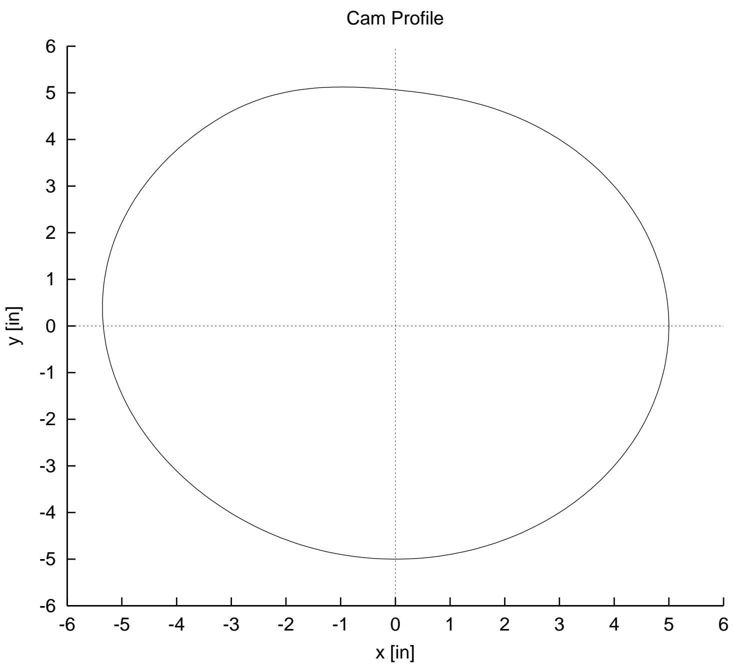
Output



Example 4

```
#include <cam.h>

int main() {

    class CCam cam;
    int steps = 360;

    cam.followerType(CAM_FOLLOWER_OSC_ROLL, 10, 10, 1);
    cam.baseRadius(5);
    cam.addSection(60, 3, CAM_MOTION_CYCLOIDAL);
    cam.addSection(120, -3, CAM_MOTION_CYCLOIDAL);
    cam.addSection(CAM_DURATION_FILL, 0, CAM_MOTION_HARMONIC);
    cam.makeCam(steps);
    cam.plotCamProfile(NULL);
}
```

Output



CCam::getCamAngle

Synopsis
#include <cam.h>
int getCamAngle(**double** *angle*[:]);

Purpose
Get the cam angular position data.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
angle A double array of size *steps*+1 for the cam angular position data.

Description
This function copies an internal array containing the angular position of the cam into *angle*. The *angle* array should be of size *steps*+1, where *steps* is the number of steps specified in **CCam**::**makeCam**(). This function should only be called after **CCam**::**makeCam**().

Example

```
#include <cam.h>

int main() {
    class CCam cam;
    int steps = 360;
    double angle[steps+1];
```

```
    cam.followerType(CAM_FOLLOWER_TRANS_FLAT, 0);
    cam.addSection(90,  1, CAM_MOTION_HARMONIC);
    cam.addSection(90, -1, CAM_MOTION_HARMONIC);
    cam.addSection(CAM_DURATION_FILL, 0, CAM_MOTION_HARMONIC);
    cam.makeCam(steps);
    cam.getCamAngle(angle);
}
```

See Also
CCam::**getCamProfile**(), **CCam**::getFollowerAccel(), **CCam**::**getFollowerPos**(), **CCam**::**getFollowerVel**, **CCam**::getTransAngle(), **CCam**::**makeCam**().

CCam::getCamProfile

Synopsis
#include <cam.h>
int getCamProfile(double *x*[:], **double** *y*[:]);

Purpose
Get the cam profile data.

Return Value
This function returns 0 on success and -1 on failure.

Parameters

x A double array of size *steps*+1 for the x coordinates of the cam profile.

y A double array of size *steps*+1 for the y coordinates of the cam profile.

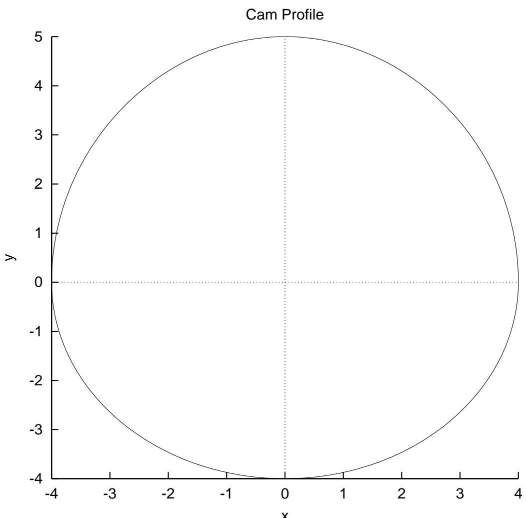
Description
This function copies internal arrays containing the cam profile into *x* and *y*. The *x* and *y* arrays should be of size *steps*+1, where *steps* is the number of steps specified in **CCam**::**makeCam**(). This function should only be called after **CCam**::**makeCam**().

Example

```
#include <cam.h>

int main() {
    class CCam cam;
    class CPlot plot;
    int steps = 360;
    double x[steps+1], y[steps+1];

    cam.followerType(CAM_FOLLOWER_TRANS_FLAT, 0);
    cam.addSection(90,  1, CAM_MOTION_HARMONIC);
    cam.addSection(90, -1, CAM_MOTION_HARMONIC);
    cam.addSection(CAM_DURATION_FILL, 0, CAM_MOTION_HARMONIC);
    cam.makeCam(steps);
    cam.getCamProfile(x, y);
    plotxy(x, y, "Cam Profile", "x", "y", &plot);
    plot.sizeRatio(-1);
```

```
    plot.plotting();
}
```

Output



See Also
CCam::makeCam().

CCam::getFollowerAccel

Synopsis
#include <cam.h>
int getFollowerAccel(double *accel*[:]);

Purpose
Get the cam follower acceleration data.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
accel A double array of size *steps*+1 for the cam acceleration data.

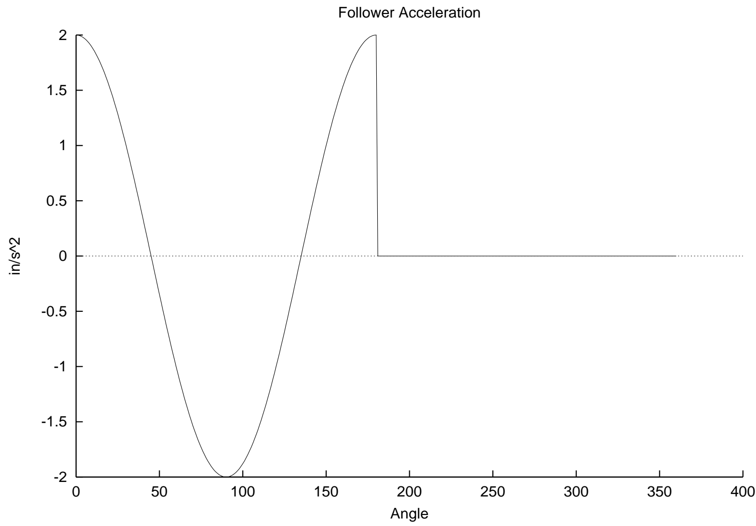
Description
This function copies an internal array containing the follower acceleration data into *accel*. The *accel* array should be of size *steps*+1, where *steps* is the number of steps specified in **CCam::makeCam**(). This function should only be called after **CCam::makeCam**().

Example

```
#include <cam.h>
```

```
int main() {
    class CCam cam;
    class CPlot plot;
    int steps = 360;
    double angle[steps+1], accel[steps+1];

    cam.followerType(CAM_FOLLOWER_TRANS_FLAT, 0);
    cam.addSection(90,  1, CAM_MOTION_HARMONIC);
    cam.addSection(90, -1, CAM_MOTION_HARMONIC);
    cam.addSection(CAM_DURATION_FILL, 0, CAM_MOTION_HARMONIC);
    cam.makeCam(steps);
    cam.getCamAngle(angle);
    cam.getFollowerAccel(accel);
    plotxy(angle, accel, "Follower Acceleration", "Angle", "in/s^2", &plot);
    plot.plotting();
}
```

Output



See Also
CCam::getCamAngle(), **CCam::getCamProfile**(), **CCam::getFollowerPos**(), **CCam::getFollowerVel**, **CCam::getTransA**
CCam::makeCam().

CCam::getFollowerPos

Synopsis
#include <cam.h>
int getFollowerPos(double *pos*[:]);

Purpose
Get the cam follower position data.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

pos A double array of size *steps*+1 for the follower position data.

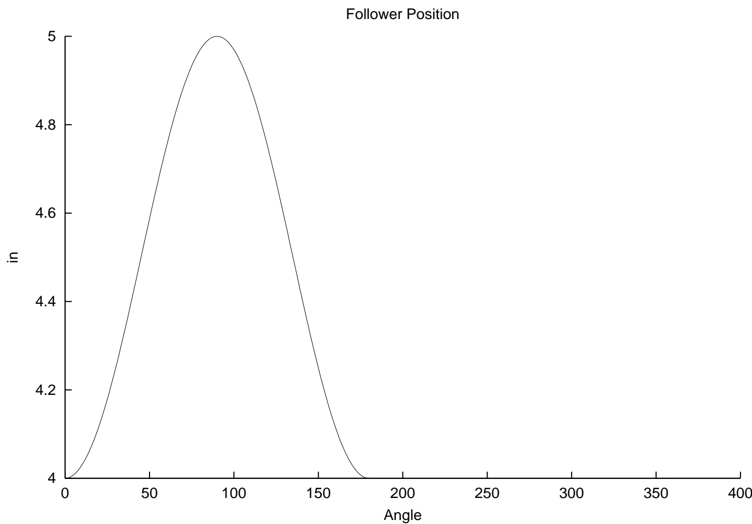
Description

This function copies an internal array containing the follower position data into *pos*. The *pos* array should be of size *steps*+1, where *steps* is the number of steps specified in **CCam::makeCam**(). This function should only be called after **CCam::makeCam**().

Example

```
#include <cam.h>

int main() {
    class CCam cam;
    class CPlot plot;
    int steps = 360;
    double angle[steps+1], pos[steps+1];

    cam.followerType(CAM_FOLLOWER_TRANS_FLAT, 0);
    cam.addSection(90,  1, CAM_MOTION_HARMONIC);
    cam.addSection(90, -1, CAM_MOTION_HARMONIC);
    cam.addSection(CAM_DURATION_FILL, 0, CAM_MOTION_HARMONIC);
    cam.makeCam(steps);
    cam.getCamAngle(angle);
    cam.getFollowerPos(pos);
    plotxy(angle, pos, "Follower Position", "Angle", "in", &plot);
    plot.sizeRatio(-1);
    plot.plotting();
}
```

Output



See Also

CCam::getCamAngle(), **CCam::getCamProfile**(), **CCam::getFollowerAccel**(), **getFollowerVel**(),

CCam::getTransAngle(), CCam::**makeCam**().

CCam::getFollowerVel

Synopsis
#include <cam.h>
int getFollowerVel(double *vel*[:]);

Purpose
Get the cam follower velocity data.

Return Value
This function returns 0 on success and -1 on failure.

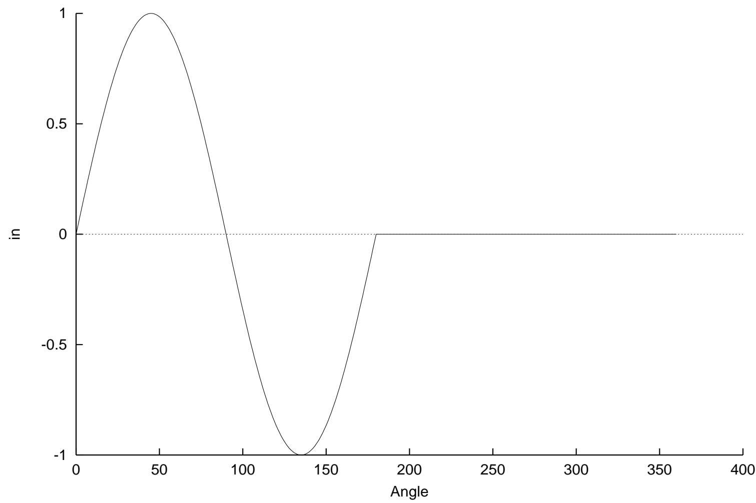
Parameters
vel A double array of size *steps*+1 for the follower velocity data.

Description
This function copies an internal array containing the follower velocity data into *vel*. The *vel* array should be of size *steps*+1, where *steps* is the number of steps specified in **CCam::makeCam**(). This function should only be called after **CCam::makeCam**().

Example

```
#include <cam.h>

int main() {
    class CCam cam;
    class CPlot plot;
    int steps = 360;
    double angle[steps+1], vel[steps+1];

    cam.followerType(CAM_FOLLOWER_TRANS_FLAT, 0);
    cam.addSection(90,  1, CAM_MOTION_HARMONIC);
    cam.addSection(90, -1, CAM_MOTION_HARMONIC);
    cam.addSection(CAM_DURATION_FILL, 0, CAM_MOTION_HARMONIC);
    cam.makeCam(steps);
    cam.getCamAngle(angle);
    cam.getFollowerVel(vel);
    plotxy(angle, vel, "Follower Velocity", "Angle", "in", &plot);
    plot.sizeRatio(-1);
    plot.plotting();
}
```

Output



See Also
CCam::getCamAngle(), **CCam::getCamProfile**(), **CCam::getFollowerAccel**(), **CCam::getFollowerPos**(), **CCam::getTransAngle**(), **CCam::makeCam**().

CCam::makeCam

Synopsis
#include <cam.h>
int makeCam(int *steps*)

Purpose
Generate the cam data and write CNC code to a file.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
steps The number of steps used in calculating the cam profile and other results.

Description
This function is called to perform the actual calculations necessary to generate the results. It should be called after all other member functions have been called except for **CCam::animation**(), **CCam::getCamAngle**(), **CCam::getCamProfile**(), **CCam::getFollowerAccel**(), **CCam::getFollowerPos**(), **CCam::getFollowerVel**(), **CCam::getTransAngle**(), **CCam::plotCamProfile**(), **CCam::plotFollowerAccel**(), **CCam::plotFollowerPos**(), **CCam::plotFollowerVel**(), and **CCam::plotTransAngle**(). The cam is divided into *steps* segments, giving *steps*+1 data points for the range of cam positions between 0 and 360 deg., inclusive. The results are stored in internal arrays for later plotting or access by the user. If a CNC file name was specified using **CCam::CNCCode**(), CNC code is written to the specified file.

Algorithm

When the **CCam** class is ready to calculate the cam profile, a loop is started. The variable ϕ starts at $0°$ and each time through the loop it is increased by $360°/$*steps*. The loop ends just before ϕ exceeds the number of degrees specified by the *duration* for the section.

Each time through the loop (for each value of phi), the position, velocity, and acceleration of the follower is calculated based on the type of motion specified by the user (harmonic or cycloidal). This loop is repeated for each section of the cam. The equations used in the calculations are shown below where L is the Lift over the duration of the section (in inches for a translating follower and radians for an oscillating follower), β is the section duration (in radians) and ϕ is the cam position relative to the beginning of the section (in radians).

The equations for harmonic motion are: (Erdman and Sandor, pp. 371-2)

$$\begin{aligned} y &= \frac{L}{2}\left[1 - \cos\left(\frac{\pi\phi}{\beta}\right)\right] \\ v &= \frac{\pi L}{2\beta}\omega \sin\left(\frac{\pi\phi}{\beta}\right) \\ a &= \frac{\pi^2 L}{2\beta^2}\omega^2 \cos\left(\frac{\pi\phi}{\beta}\right) \end{aligned}$$

The equations for cycloidal motion are: (Erdman and Sandor, pp. 372-3)

$$\begin{aligned} y &= L\left[\frac{\phi}{\beta} - \frac{1}{2\pi}\sin\left(\frac{2\pi\phi}{\beta}\right)\right] \\ v &= \frac{L}{\beta}\omega\left[1 - \cos\left(\frac{2\pi\phi}{\beta}\right)\right] \\ a &= \frac{2\pi L}{\beta^2}\omega^2 \sin\left(\frac{2\pi\phi}{\beta}\right) \end{aligned}$$

For translating followers the above equations produce the linear position, velocity and acceleration of the follower. For oscillating followers, the equations produce the angular position, angular velocity and angular acceleration of the follower. Once the position, velocity, and acceleration are calculated for a given value of ϕ, the coordinates of the cam profile and cutter center are calculated.

For a translating flat-faced follower, the angle between the direction of follower translation and the contact point of the cam and follower is: (Erdman and Sandor, pp. 392)

$$\begin{aligned} \theta &= \arctan\left(\frac{1}{L}\frac{dL}{d\phi}\right) \\ &= \arctan\left(\frac{v}{L\omega}\right) \end{aligned}$$

The distance between the cam center and the point of contact is:

$$R = \frac{L}{\cos\theta}$$

The coordinates of the cam profile with respect to the reference radial is:

$$x_{cam} = R\cos(\phi_{total} + \theta)$$
$$y_{cam} = R\sin(\phi_{total} + \theta)$$

The angle between the direction of follower translation and the center of the cutter is:

$$\gamma = \arctan\left(\frac{R\sin\theta}{L + r_c}\right)$$

The distance between the cam center and cutter center is:

$$c = \frac{L + r_c}{\cos\gamma}$$

The coordinates of the cutter center with respect to the reference radial is:

$$x_{cutter} = c\cos(\gamma + \phi_{total})$$
$$y_{cutter} = c\sin(\gamma + \phi_{total})$$

For an oscillating flat-faced follower, the angel between the normal to the follower face and the point of contact is: (Erdman and Sandor, pp. 394)

$$\begin{aligned}\theta &= \arctan\left[\left(\frac{(d\zeta/d\phi)}{1 - (d\zeta/d\phi)}\right)\frac{m\cos\zeta}{f - m\sin\zeta}\right] \\ &= \arctan\left[\left(\frac{(v/\omega)}{(1 - v/\omega)}\right)\frac{m\cos\zeta}{f - m\sin\zeta}\right]\end{aligned}$$

The distance between the cam center and the contact point is:

$$R = \frac{f + m\sin\zeta}{\cos\theta}$$

The coordinates of the cam profile with respect to the reference radial is:

$$x_{cam} = R\cos\left(\phi_{total} + \theta + \frac{\pi}{2} + \zeta\right)$$
$$y_{cam} = R\sin\left(\phi_{total} + \theta + \frac{\pi}{2} + \zeta\right)$$

The coordinates of the cutter center with respect to the reference radial is:

$$x_{cutter} = \sqrt{c_x^2 + c_y^2}\cos\left(\phi_{total} + \theta + \frac{\pi}{2} + \zeta - \arctan\frac{c_x}{c_y}\right)$$
$$y_{cutter} = \sqrt{c_x^2 + c_y^2}\sin\left(\phi_{total} + \theta + \frac{\pi}{2} + \zeta - \arctan\frac{c_x}{c_y}\right)$$

where

$$\begin{aligned} c_x &= R + r_c \cos\theta \\ c_y &= r_c \sin\theta \end{aligned}$$

For a translating roller follower the angle between the contact point of the cam and follower and the cam center is:

$$\begin{aligned} \alpha &= \arctan\left[\frac{L(dL/d\phi)}{m^2 + L^2 - m(dL/d\phi)}\right] \\ &= \arctan\left[\frac{L(v/\omega)}{m^2 + L^2 - m(v/\omega)}\right] \end{aligned}$$

The pressure angle is:

$$\delta = \alpha - \psi$$

where

$$\psi = \arctan(\frac{m}{L})$$

The distance between the cam and follower centers is

$$F = \sqrt{L^2 + m^2}$$

The coordinates of the cam profile with respect to the reference radial are:

$$\begin{aligned} x_{cam} &= \sqrt{{R_x}^2 + {R_y}^2} \cos(\phi_{total} + \psi + \arctan(R_y/R_x)) \\ y_{cam} &= \sqrt{{R_x}^2 + {R_y}^2} \sin(\phi_{total} + \psi + \arctan(R_y/R_x)) \end{aligned}$$

where

$$\begin{aligned} R_x &= F - r_f \cos\alpha \\ R_y &= r_f \sin\alpha \end{aligned}$$

The coordinates of the cutter location with respect to the reference radial are:

$$\begin{aligned} x_{cutter} &= \sqrt{c_x^2 + c_y^2} \cos(\phi_{total} + \psi + \arctan(c_y/c_x)) \\ y_{cutter} &= \sqrt{c_x^2 + c_y^2} \sin(\phi_{total} + \psi + \arctan(c_y/c_x)) \end{aligned}$$

where

$$
\begin{aligned}
c_x &= F + (r_c - r_f)\cos\alpha \\
c_y &= (r_f - r_c)\sin\alpha
\end{aligned}
$$

For an oscillating roller follower, angle between the follower center and the cam/follower contact point is

$$
\begin{aligned}
\alpha &= \arctan\left[\frac{(A\sin\gamma)\,(d\zeta/d\phi)}{L-(A\cos\gamma)\,(d\zeta/d\phi)}\right] \\
&= \arctan\left[\frac{(A\sin\gamma)\,(v/\omega)}{L-(A\cos\gamma)\,(v/\omega)}\right]
\end{aligned}
$$

where

$$
\begin{aligned}
L &= (A^2 + m^2 - 2Am\cos(\zeta)) \\
\gamma &= \arcsin(m\sin(\zeta)/L)
\end{aligned}
$$

The initial follower angle is

$$
\zeta_0 = \cos^{-1}((A^2 + m^2 - (r_b + r_f)^2)/(2mA)
$$

The pressure angle is

$$
\delta = \gamma + \alpha + -\frac{\pi}{2}
$$

The coordinates of the cam profile with respect to the reference radial are:

$$
\begin{aligned}
x_{cam} &= \sqrt{R_x^2 + R_y^2}\cos[\phi_{total} + \pi - \gamma - \zeta + \arctan(\frac{R_y}{R_x})] \\
y_{cam} &= \sqrt{R_x^2 + R_y^2}\sin[\phi_{total} + \pi - \gamma - \zeta + \arctan(\frac{R_y}{R_x})]
\end{aligned}
$$

where

$$
\begin{aligned}
R_x &= L - r_f\cos\alpha \\
R_y &= r_f\sin\alpha
\end{aligned}
$$

The coordinates of the cutter center with respect to the reference radial are:

$$
\begin{aligned}
x_{cutter} &= \sqrt{c_x^2 + c_y^2}\cos[\phi_{total} + \pi - \gamma - \zeta + \arctan(\frac{c_y}{c_x})] \\
y_{cutter} &= \sqrt{c_x^2 + c_y^2}\sin[\phi_{total} + \pi - \gamma - \zeta + \arctan(\frac{c_y}{c_x})]
\end{aligned}
$$

$$
\begin{aligned}
c_x &= L + (r_c - r_f)\cos\alpha \\
c_y &= (r_f - r_c)\sin\alpha
\end{aligned}
$$

References

Erdman, A. G. and Sandor, G. N., 1997, *Mechanism Design Analysis and Synthesis*, Vol. 1, 3rd ed., Prentice Hall, Englewood Cliffs, NJ.

CCam::plotCamProfile

Synopsis
#include <cam.h>
int plotCamProfile(**CPlot** **plot*);

Syntax
plotCamProfile(*&plot*)

Purpose
Plot the cam profile data.

Return Value
This function returns 0 on success and -1 on failure.

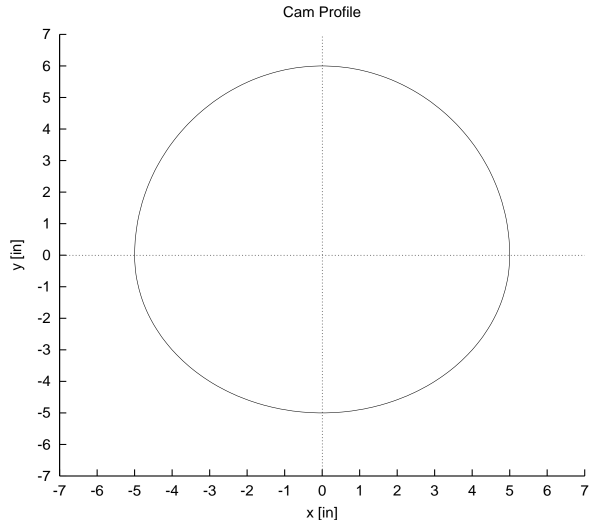
Parameters
&plot A pointer to a CPlot class variable for formatting the plot of the cam profile.

Description
Plot the cam profile. This function should only be called after **CCam::makeCam**().

Example

```
#include <cam.h>

int main() {
    class CCam cam;
    class CPlot plot;
    int steps = 360;

    cam.followerType(CAM_FOLLOWER_TRANS_FLAT, 0);
    cam.baseRadius(5);
    cam.addSection(90,  1, CAM_MOTION_HARMONIC);
    cam.addSection(90, -1, CAM_MOTION_HARMONIC);
    cam.addSection(CAM_DURATION_FILL, 0, CAM_MOTION_HARMONIC);
    cam.makeCam(steps);
    cam.plotCamProfile(&plot);
}
```

Output



See Also
CCam::**makeCam**().

CCam::plotFollowerAccel

Synopsis
#include <cam.h>
int plotFollowerAccel(**CPlot** **plot*);

Syntax
plotFollowerAccel(*&plot*)

Purpose
Plot the cam follower acceleration vs. the cam angular position.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
&plot A pointer to a CPlot class variable for formatting the plot of the cam follower acceleration.

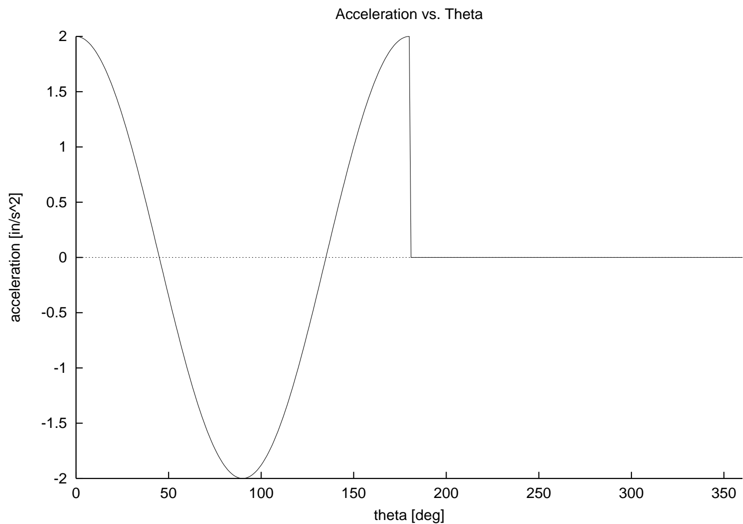
Description
Plot the cam follower acceleration. This function should only be called after **CCam**::**makeCam**().

Example

```
#include <cam.h>

int main() {
    class CCam cam;
    class CPlot plot;
    int steps = 360;

    cam.followerType(CAM_FOLLOWER_TRANS_FLAT, 0);
```

```
    cam.baseRadius(5);
    cam.addSection(90,  1, CAM_MOTION_HARMONIC);
    cam.addSection(90, -1, CAM_MOTION_HARMONIC);
    cam.addSection(CAM_DURATION_FILL, 0, CAM_MOTION_HARMONIC);
    cam.makeCam(steps);
    cam.plotFollowerAccel(&plot);
}
```

Output



See Also
CCam::**makeCam**().

CCam::plotFollowerPos

Synopsis
#include <**cam.h**>
int plotFollowerPos(**CPlot** **plot*);

Syntax
plotFollowerPos(*&plot*)

Purpose
Plot the cam follower position vs. the cam angular position.

Return Value
This function returns 0 on success and -1 on failure.

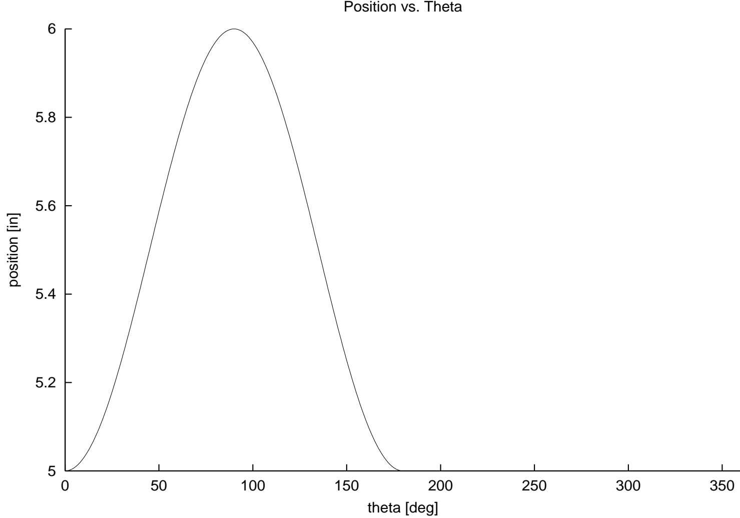
Parameters
&plot A pointer to a CPlot class variable for formatting the plot of the cam follower position.

Description
Plot the cam follower position. This function should only be called after **CCam**::**makeCam**().

Example

```
#include <cam.h>

int main() {
    class CCam cam;
    class CPlot plot;
    int steps = 360;

    cam.followerType(CAM_FOLLOWER_TRANS_FLAT, 0);
    cam.baseRadius(5);
    cam.addSection(90,  1, CAM_MOTION_HARMONIC);
    cam.addSection(90, -1, CAM_MOTION_HARMONIC);
    cam.addSection(CAM_DURATION_FILL, 0, CAM_MOTION_HARMONIC);
    cam.makeCam(steps);
    cam.plotFollowerPos(&plot);
}
```

Output



See Also

CCam::makeCam().

CCam::plotFollowerVel

Synopsis

#include <cam.h>
int plotFollowerVel(CPlot **plot*);

Syntax

plotFollowerVel(*&plot*)

Purpose

Plot the cam follower velocity vs. the cam angular position.

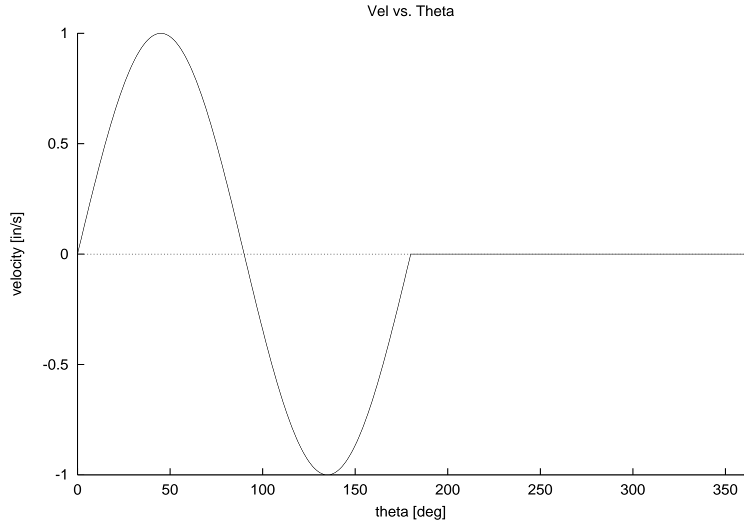
Return Value
This function returns 0 on success and -1 on failure.

Parameters
&plot A pointer to a CPlot class variable for formatting the plot of the cam follower velocity.

Description
Plot the cam follower velocity. This function should only be called after **CCam**::**makeCam**().

Example
```
#include <cam.h>

int main() {
    class CCam cam;
    class CPlot plot;
    int steps = 360;

    cam.followerType(CAM_FOLLOWER_TRANS_FLAT, 0);
    cam.baseRadius(5);
    cam.addSection(90,  1, CAM_MOTION_HARMONIC);
    cam.addSection(90, -1, CAM_MOTION_HARMONIC);
    cam.addSection(CAM_DURATION_FILL, 0, CAM_MOTION_HARMONIC);
    cam.makeCam(steps);
    cam.plotFollowerVel(&plot);
}
```

Output



See Also
CCam::**makeCam**().

CCam::plotTransAngle

Synopsis
#include <cam.h>

int plotTransAngle(CPlot **plot*);

Syntax
plotTransAngle(*&plot*)

Purpose
Plot the cam transmission angle vs. the cam angular position.

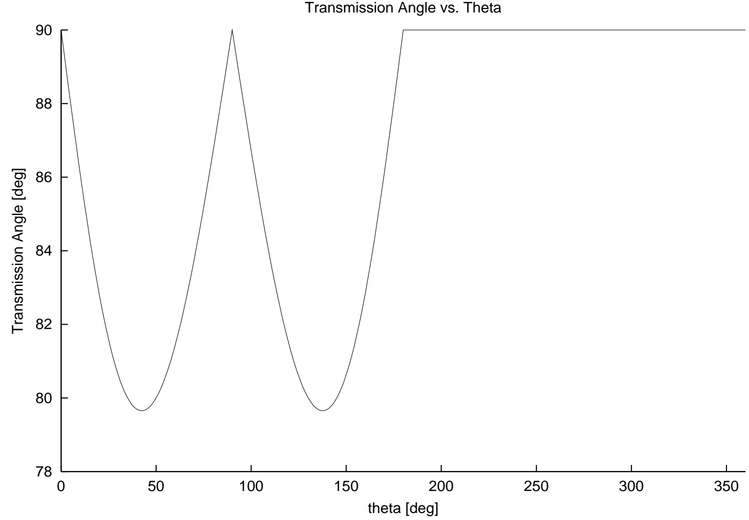
Return Value
This function returns 0 on success and -1 on failure.

Parameters
&plot A pointer to a CPlot class variable for formatting the plot of the cam transmission angle.

Description
Plot the cam transmission angle. This function should only be called after **CCam::makeCam**().

Example
```
#include <cam.h>

int main() {
    class CCam cam;
    class CPlot plot;
    int steps = 360;

    cam.followerType(CAM_FOLLOWER_TRANS_FLAT, 0);
    cam.baseRadius(5);
    cam.addSection(90,  1, CAM_MOTION_HARMONIC);
    cam.addSection(90, -1, CAM_MOTION_HARMONIC);
    cam.addSection(CAM_DURATION_FILL, 0, CAM_MOTION_HARMONIC);
    cam.makeCam(steps);
    cam.plotTransAngle(&plot);
}
```

Output



See Also
CCam::**makeCam**().

CCam::spindleSpeed

Synopsis
#include <cam.h>
void spindleSpeed(**double** *spindle_speed*);

Purpose
Set the spindle speed for CNC code generation.

Return Value
None.

Parameters
spindle_speed The spindle speed for the cutter in RPM (revolution per minute).

Description
The *spindle_speed* is the rotational speed of the cutter in RPM. This parameter does not effect the shape of the cam profile and only needs to be set if CNC code is to be created. By default, the spindle speed is 3036 RPM.

Example
See example in **CCam**::**CNCCode**().

See Also
CCam::**CNCCode**(), **CCam**::**cutDepth**(), **CCam**::**cutter**(), **CCam**::**cutterOffset**(), and **CCam**::**feedrate**().

cam.ch

Synopsis
cam.ch [-**acceleration**]
[-**animate**]
[-**base** *radius*]
[-**cncfile** *filename*]
[-**cutter** *radius length num*]
[-**depth** *depth*]
[-**feedrate** *rate*]
[-**follower** *follower_type* [*e*] [*e rf*] [*m f*] [*m A rf*]]
[-**offset** *x y z*]
[-**omega** *omega*]
[-**output** *display*]
[-**position**]
[-**profile**]
-**steps** *num*
[-**section** *duration displacement motion_type*]
[-**spindle** *speed*]
[-**transangle**]
[-**velocity**]

Syntax
As many of the optional arguments as desired can be used. The minimum number of arguments required to produce meaningful results are **steps**, two **sections**, and at least one of the output options (-**acceleration**, -**animate**, -**position**, -**profile**, -**transangle**, -**velocity**).

Purpose
This program provides a command line interface to the design capabilities of the **CCam** class.

Return Value
This program returns 0 on successful completion, or -1 if there is an error.

Arguments
steps The number of steps used in calculating the cam. See **CCam**::**makeCam**().

acceleration Plot the follower acceleration. See **CCam**::**plotFollowerAccel**().

animate Animate the cam using **xlinkage**. See **CCam**::**animation**().

base The base radius of the cam. See **CCam**::**baseRadius**.

cncfile The name of a file for CNC code output. See **CCam**::**CNCCode**().

cutter The cutter parameters for CNC code generation. See **CCam**::**cutter**().

depth The cutter depth for CNC code generation. See **CCam**::**cutDepth**().

feedrate The feedrate for CNC code generation. See **CCam**::feedrate.

follower The follower type and related parameters. See **CCam**::**followerType**().

offset The cutter home position offset CNC code generation. See **CCam**::cutterOffset().

omega The angular velocity of the cam. See **CCam**::**angularVel**().

output The output type for plots and animation. This can be **CAM_OUTPUT_DISPLAY** or **CAM_OUTPUT_STREAM**. File output is not supported.

position Plot the follower position. See **CCam**::**plotFollowerPos**().

profile Plot the cam profile. See **CCam**::**plotCamProfile**().

section The parameters for a new cam section. See **CCam**::**addSection**().

spindle The cutter spindle speed for CNC code generation. See **CCam**::**spindleSpeed**().

transangle Plot the transmission angle. See **CCam**::**plotTransAngle**().

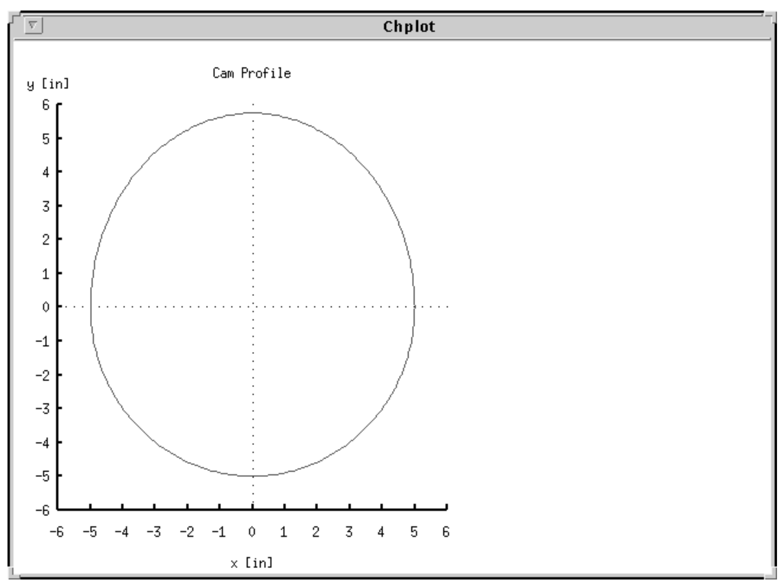
velocity Plot the follower velocity. See **CCam**::**plotFollowerVel**().

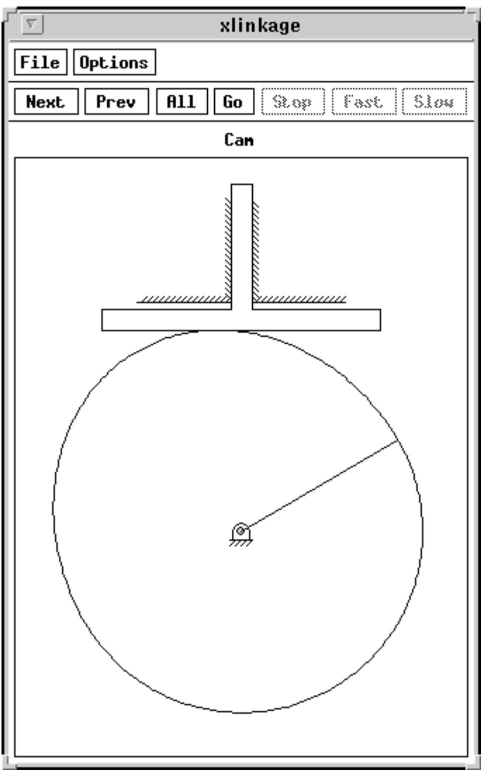
Description
The program functions as a command line interface to the design capabilities of the **CCam** class. It was developed primarily for use internally by the Web-based cam design tools.

Example 1

```
cam.ch -section 90 .75 CAM_MOTION_HARMONIC -section 90 -.75
CAM_MOTION_HARMONIC -section CAM_DURATION_FILL 0 CAM_MOTION_HARMONIC
-steps 300 -profile -animate 30
```

Output

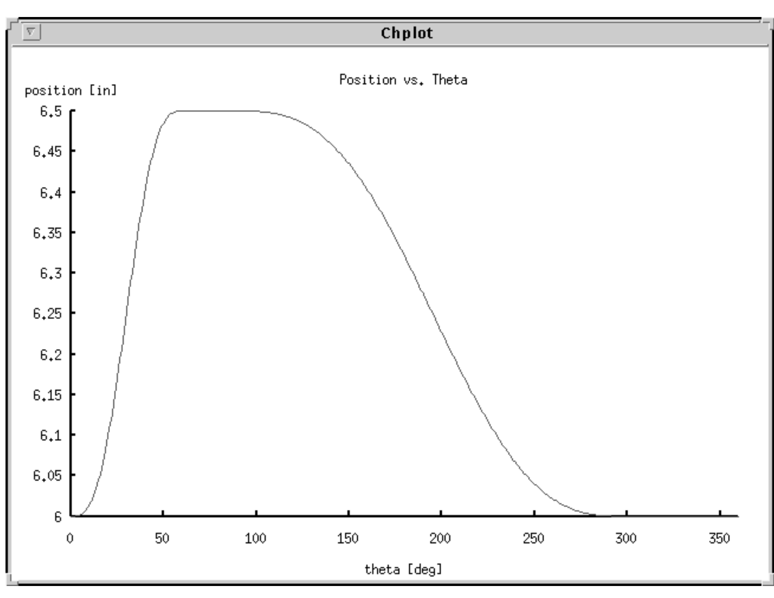




Example 2

```
cam.ch -section 60 .5 CAM_MOTION_CYCLOIDAL -section 30 0
CAM_MOTION_CYCLOIDAL -section 210 -.5 CAM_MOTION_CYCLOIDAL -section
CAM_DURATION_FILL 0 CAM_MOTION_HARMONIC -follower CAM_FOLLOWER_TRANS_ROLL
1 1.5 -base 4.5 -steps 500 -position
```

Output



See Also
CCam class.

Appendix K

Solving Complex Equations

A complex equation can be expressed in a general polar form of

$$R_1 e^{i\phi_1} + R_2 e^{i\phi_2} = z_3 \tag{K.1}$$

where z_3 can be expressed in either Cartesian coordinates $x_3 + iy_3$ as `complex(x3, y3)`, or polar coordinates $R_3 e^{i\phi_3}$ as `polar(R3, phi3)`. Because a complex equation can be partitioned into real and imaginary parts, two unknowns out of four parameters R_1, ϕ_1, R_2, and ϕ_2 can be solved in this equation.

To solve for these two unknowns, we first decompose equation (K.1) into the following real and imaginary parts.

$$R_1 \cos\phi_1 + R_2 \cos\phi_2 = x_3 \tag{K.2}$$

$$R_1 \sin\phi_1 + R_2 \sin\phi_2 = y_3 \tag{K.3}$$

In equation (K.1), parameters R_1, ϕ_1, R_2, and ϕ_2 are in positions 1, 2, 3, and 4, respectively. Depending on the positions of unknowns, we can solve equation (K.1) in the following six cases.

Case 1: $n_1 = 1, n_2 = 2$, Solve for R_1 and ϕ_1, given R_2, ϕ_2, R_3, ϕ_3 or R_2, ϕ_2, x_3, y_3. Where, n_1 and n_2 are the first and second positions of two unknowns on the left hand side of equation (K.1), respectively.

From equations (K.2) and equations (K.3), we get

$$R_1 \cos\phi_1 = x_3 - R_2 \cos\phi_2 = a \tag{K.4}$$

$$R_1 \sin\phi_1 = y_3 - R_2 \sin\phi_2 = b \tag{K.5}$$

R_1 and ϕ_1 can be calculated as

$$R_1 = \sqrt{a^2 + b^2} \tag{K.6}$$

$$\phi_1 = \operatorname{atan2}(b, a) \tag{K.7}$$

Case 2: $n_1 = 1, n_2 = 3$, Given ϕ_1 and ϕ_2, R_1 and R_2 are solved.

Multiplying equation (K.1) by $e^{-i\phi_2}$ and $e^{-i\phi_1}$ gives

$$R_1 e^{i(\phi_1 - \phi_2)} + R_2 = R_3 e^{i(\phi_3 - \phi_2)} \tag{K.8}$$

$$R_1 + R_2 e^{i(\phi_2 - \phi_1)} = R_3 e^{i(\phi_3 - \phi_1)} \tag{K.9}$$

Imaginary parts of equation (K.7) and equation (K.8) are

$$R_1 \sin(\phi_1 - \phi_2) = R_3 \sin(\phi_3 - \phi_2) \tag{K.10}$$

$$R_2 \sin(\phi_2 - \phi_1) = R_3 \sin(\phi_3 - \phi_1) \tag{K.11}$$

From equation (K.10) and equation (K.11) we get

$$R_1 = R_3 \frac{\sin(\phi_3 - \phi_2)}{\sin(\phi_1 - \phi_2)} \tag{K.12}$$

$$R_2 = R_3 \frac{\sin(\phi_3 - \phi_1)}{\sin(\phi_2 - \phi_1)} \tag{K.13}$$

Case 3: $n_1 = 1, n_2 = 4$, Given ϕ_1 and R_2, R_1 and ϕ_2 are solved.

From equation (K.2) we have

$$R_1 = \frac{x_3 - R_2 \cos\phi_2}{\cos\phi_1} \tag{K.14}$$

substitute equation (K.14) into equation (K.3) we get

$$(x_3 - R_2 \cos\phi_2) \sin\phi_1 + R_2 \sin\phi_2 \cos\phi_1 = y_3 \cos\phi_1 \tag{K.15}$$

equation (K.15) can be simplified as

$$\sin(\phi_2 - \phi_1) = \frac{y_3 \cos\phi_1 - x_3 \sin\phi_1}{R_2} = a \tag{K.16}$$

then

$$\phi_2 = \phi_1 + \sin^{-1}(a) \phi_2 = \phi_1 + \pi - \sin^{-1}(a) \tag{K.17}$$

If $\cos\phi_1$ is larger than machine epsilon,ε, R_1 can be obatined using equation (K.14). Otherwise R_1 will be obtained using equation (K.3)
if ($|\cos\phi_1)| >$ FLT_EPSILON

$$R_1 = \frac{x_3 - R_2 \cos\phi_2}{\cos\phi_1} \tag{K.18}$$

else

$$R_1 = \frac{y_3 - R_2 \sin\phi_2}{\sin\phi_1} \tag{K.19}$$

Case 4: $n_1 = 2, n_2 = 4$, given R_1 and R_2, ϕ_1 and ϕ_2 in equation (K.1) can be solved.

From equation (K.1), we get

$$\cos\phi_1 = \frac{x_3 - R_2 \cos\phi_2}{R_1}, \quad \sin\phi_1 = \frac{y_3 - R_2 \sin\phi_2}{R_1} \tag{K.20}$$

Substituting these results into the identity equation $\sin^2\phi_1 + \cos^2\phi_1 = 1$ and simplifying the resultant equation, we get

$$y_3 \sin\phi_2 + x_3 \cos\phi_2 = \frac{x_3^2 + y_3^2 + R_2^2 - R_1^2}{2R_2} \tag{K.21}$$

from equation (K.21), we can derive the formulas for ϕ_1 and ϕ_2 as shown in Figure K.1.

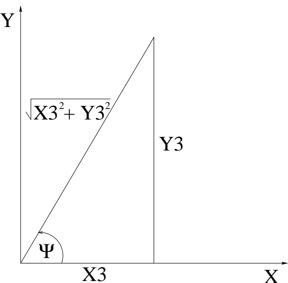


Figure K.1: Figure for solving the complex equation in case 5.

$$\tan\psi = \frac{y_3}{x_3}, \cos\psi = \frac{x_3}{\sqrt{x_3^2+y_3^2}}, \sin\psi = \frac{y_3}{\sqrt{x_3^2+y_3^2}}$$

Equation (K.21) becomes,

$$\frac{y_3}{\sqrt{x_3^2+y_3^2}}\sin\phi_2 + \frac{x_3}{\sqrt{x_3^2+y_3^2}}\cos\phi_2 = \frac{x_3^2+y_3^2+R_2^2-R_1^2}{2R_2\sqrt{x_3^2+y_3^2}} \tag{K.22}$$

$$a = \sin\psi\sin\phi_2 + \cos\psi\cos\phi_2$$

Let,

$$a = \frac{x_3^2+y_3^2+R_2^2-R_1^2}{2R_2\sqrt{x_3^2+y_3^2}} \tag{K.23}$$

Becomes,

$$\cos(\phi_2 - \psi) = a \tag{K.24}$$

$$\phi_2 = \psi \pm \cos^{-1}(a) \tag{K.25}$$

$$= \text{atan2}(y_3, x_3) \pm \text{acos}\left(\frac{x_3^2+y_3^2+R_2^2-R_1^2}{2R_2\sqrt{x_3^2+y_3^2}}\right) \tag{K.26}$$

ϕ_1 can be obtained using equation (K.20)

$$\phi_1 = \text{atan2}(\sin\phi_1, \cos\phi_1) \tag{K.27}$$

or use identity equation,

$$\tan(\frac{\phi}{2}) = \frac{1-\cos\phi}{\sin\phi} \text{ or } \tan(\frac{\phi}{2}) = \frac{\sin\phi}{1+\cos\phi} \tag{K.28}$$

$$\phi = 2\tan^{-1}(\frac{1-\cos\phi}{\sin\phi}) \text{ or } \phi = 2\tan^{-1}(\frac{\sin\phi}{1+\cos\phi}) \tag{K.29}$$

Case 5: $n_1 = 2, n_2 = 3$, given R_1 and ϕ_2, ϕ_1 and R_2 are solved. This case is similar to Case 3.

Case 6: $n_1 = 3, n_2 = 4$, given R_1 and ϕ_1, R_2 and ϕ_2 are solved. This case is similar to Case 1.

Index

addSection, 442, *see* CCam

angularAccel, *see* CFourbar, *see* CCrankSlider, *see* CGearedFivebar, *see* CFourbarSlider, *see* CWattSixbarI, *see* CWattSixbarII, *see* CStevSixbarI, *see* CStevSixbarIII

angularAccels, *see* CFourbar

angularPos, *see* CFourbar, *see* CCrankSlider, *see* CGearedFivebar, *see* CFourbarSlider, *see* CWattSixbarI, *see* CWattSixbarII, *see* CStevSixbarI, *see* CStevSixbarIII

angularPoss, *see* CFourbar

angularVel, *see* CFourbar, *see* CCrankSlider, *see* CGearedFivebar, *see* CFourbarSlider, *see* CWattSixbarI, *see* CWattSixbarII, *see* CStevSixbarI, *see* CStevSixbarIII, 442, *see* CCam

angularVels, *see* CFourbar

animation, 2, *see* CFourbar, *see* CCrankSlider, *see* CGearedFivebar, *see* CFourbarSlider, *see* CWattSixbarI, *see* CWattSixbarII, *see* CStevSixbarI, *see* CStevSixbarIII, 442, *see* CCam

baseRadius, 442, *see* CCam

C/C++, 2

cam.ch, **474**

CAM_DURATION_FILL, **443**, 444

CAM_FOLLOWER_OSC_FLAT, **443**

CAM_FOLLOWER_OSC_ROLL, **443**

CAM_FOLLOWER_TRANS_FLAT, **443**

CAM_FOLLOWER_TRANS_ROLL, **443**

CAM_MOTION_CYCLOIDAL, **443**, 444

CAM_MOTION_HARMONIC, **443**, 444

CCam, **442**, 442
- ~CCam, 442
- addSection, 442, **444**
- angularVel, 442, **444**
- animation, 442, **446**
- baseRadius, 442, **447**
- CCam, 442
- CNCCode, 442, **448**
- cutDepth, 442, **449**
- cutter, 442, **449**
- cutterOffset, 442, **450**
- deleteCam, 442, **450**
- feedrate, 442, **451**
- followerType, 442, **451**
- getCamAngle, 442, **456**
- getCamProfile, 442, **457**
- getFollowerAccel, 443, **458**
- getFollowerPos, 443, **459**
- getFollowerVel, 443, **461**
- getTransAngle, 443
- makeCam, 443, **462**
- plotCamProfile, 443, **467**
- plotFollowerAccel, 443, **468**
- plotFollowerPos, 443, **469**
- plotFollowerVel, 443, **470**
- plotTransAngle, 443, **471**
- spindleSpeed, 443, **473**

CCrankSlider, 287
- angularAccel, 288
- angularPos, 290
- angularVel, 291
- animation, 292
- couplerCurve, 293
- couplerPointAccel, 295
- couplerPointPos, 296
- couplerPointVel, 297
- displayPosition, 298
- forceTorque, 299
- forceTorques, 301
- getJointLimits, 304
- plotCouplerCurve, 305
- plotForceTorques, 306
- setAngularVel, 309
- setCouplerPoint, 308
- setGravityCenter, 308
- setInertia, 309
- setLinks, 310
- setMass, 310

setNumPoints, 311
sliderAccel, 311
sliderPos, 313
sliderVel, 314
transAngle, 315
uscUnit, 316
CFourbar, **233**
angularAccel, 235
angularAccels, 237
angularPos, 238
angularPoss, 239
angularVel, 241
angularVels, 242
animation, 244
couplerCurve, 252
couplerPointAccel, 254
couplerPointPos, 256
couplerPointVel, 257, 351
displayPosition, 259
displayPositions, 260
forceTorque, 261
forceTorques, 263
getAngle, 266
getJointLimits, 266
grashof, 268
plotAngularAccels, 269
plotAngularPoss, 270
plotAngularVels, 272
plotCouplerCurve, 273
plotForceTorques, 274
plotTransAngles, 275
printJointLimits, 276
setAngularVel, 277
setCouplerPoint, 278
setGravityCenter, 279
setInertia, 279
setLinks, 280
setMass, 280
setNumPoints, 281
synthesis, 281
transAngle, 283
transAngles, 284
uscUnit, 285
CFourbarSlider, **338**
angularAccel, 339
angularPos, 341
angularVel, 343
animation, 345
couplerPointAccel, 347
couplerPointPos, 349
displayPosition, 353
setAngularVel, 355
setCouplerPoint, 354
setLinks, 356
setNumPoints, 356
sliderAccel, 357
sliderPos, 358
sliderVel, 360
uscUnit, 362
CGearedFivebar, **317**
angularAccel, 318
angularPos, 320
angularVel, 322
animation, 323
couplerCurve, 325
couplerPointAccel, 327
couplerPointPos, 329
couplerPointVel, 330
displayPosition, 331
plotCouplerCurve, 333
setAngularVel, 336
setCouplerPoint, 335
setLinks, 336
setNumPoints, 337
uscUnit, 337
CGI, 217
CGI Programming, 213
CNCCode, 442, *see* CCam
complexsolve(), 5
copyright, ii
COUPLER_LINK3, 232
COUPLER_LINK4, 232
COUPLER_LINK5, 232
COUPLER_LINK6, 232
COUPLER_LINK7, 232
couplerCurve, *see* CFourbar, *see* CCrankSlider, *see* CGearedFivebar
couplerPointAccel, *see* CFourbar, *see* CCrankSlider, *see* CGearedFivebar, *see* CFourbarSlider, *see* CWattSixbarI, *see* CWattSixbarII, *see* CStevSixbarI, *see* CStevSixbarIII
couplerPointPos, *see* CFourbar, *see* CCrankSlider, *see* CGearedFivebar, *see* CFourbarSlider, *see* CWattSixbarI, *see* CWattSixbarII, *see* CStevSixbarI, *see* CStevSixbarIII
couplerPointVel, *see* CFourbar, *see* CCrankSlider,

see CGearedFivebar, *see* CFourbarSlider, *see* CWattSixbarI, *see* CWattSixbarII, *see* CStevSixbarI, *see* CStevSixbarIII
CStevSixbarI, **402**
angularAccel, 403
angularPos, 405
angularVel, 407
animation, 409
couplerPointAccel, 411
couplerPointPos, 413
couplerPointVel, 415
displayPosition, 417
setAngularVel, 420
setCouplerPoint, 419
setLinks, 420
setNumPoints, 421
uscUnit, 421
CStevSixbarIII, **423**
angularAccel, 424
angularPos, 426
angularVel, 427
animation, 429
couplerPointAccel, 431
couplerPointPos, 433
couplerPointVel, 434
displayPosition, 436
setAngularVel, 439
setCouplerPoint, 438
setLinks, 439
setNumPoints, 440
uscUnit, 440
cutDepth, 442, *see* CCam
cutter, 442, *see* CCam
cutterOffset, 442, *see* CCam
CWattSixbarI, **363**
angularAccel, 364
angularPos, 366
angularVel, 367
animation, 369
couplerPointAccel, 371
couplerPointPos, 373
couplerPointVel, 375
displayPosition, 377
setAngularVel, 380
setCouplerPoint, 379
setLinks, 380
setNumPoints, 381
uscUnit, 381
CWattSixbarII, **382**
angularAccel, 383
angularPos, 385
angularVel, 387
animation, 390
couplerPointAccel, 392
couplerPointPos, 394
couplerPointVel, 395
displayPosition, 395
getIORanges, 397
setAngularVel, 399
setCouplerPoint, 399
setLinks, 400
setNumPoints, 400
uscUnit, 401

deleteCam, 442, *see* CCam
displayPosition, *see* CFourbar, *see* CCrankSlider, *see* CGearedFivebar, *see* CFourbarSlider, *see* CWattSixbarI, *see* CWattSixbarII, *see* CStevSixbarI, *see* CStevSixbarIII
displayPositions, *see* CFourbar

embeddable, 2

features, 1
feedrate, 442, *see* CCam
followerType, 442, *see* CCam
forceTorque, *see* CFourbar, *see* CCrankSlider
forceTorques, *see* CFourbar, *see* CCrankSlider
FOURBAR_CRANKCRANK, 235
FOURBAR_CRANKROCKER, 234
FOURBAR_INVALID, 234
FOURBAR_INWARDINWARD, 235
FOURBAR_INWARDOUTWARD, 235
FOURBAR_LINK1, 234
FOURBAR_LINK2, 234
FOURBAR_LINK3, 234
FOURBAR_LINK4, 234
FOURBAR_OUTWARDINWARD, 235
FOURBAR_OUTWARDOUTWARD, 235
FOURBAR_ROCKERCRANK, 235
FOURBAR_ROCKERROCKER, 235

getAngle, *see* CFourbar
getCamAngle, 442, *see* CCam
getCamProfile, 442, *see* CCam
getFollowerAccel, 443, *see* CCam
getFollowerPos, 443, *see* CCam

getFollowerVel, 443, *see* CCam
getIORanges, *see* CWattSixbarII
getJointLimits, *see* CFourbar, *see* CCrankSlider
getTransAngle, 443
grashof, *see* CFourbar

HTML, 214
  tags, 217

linkage, **6**, **231**

M_DEG2RAD(), 231
M_FT2M, 231
M_LB2N, 231
M_LBFT2NM, 231
M_LBFTSS2KGMM, 231
M_RAD2DEG(), 231
M_SLUG2KG, 231
makeCam, 443, *see* CCam

object-oriented, 2

plotAngularAccels, *see* CFourbar
plotAngularPoss, *see* CFourbar
plotAngularVels, *see* CFourbar
plotCamProfile, 443, *see* CCam
plotCouplerCurve, *see* CFourbar, *see* CCrankSlider, *see* CGearedFivebar
plotFollowerAccel, 443, *see* CCam
plotFollowerPos, 443, *see* CCam
plotFollowerVel, 443, *see* CCam
plotForceTorques, *see* CFourbar, *see* CCrankSlider
plotting, 2
plotTransAngle, 443, *see* CCam
plotTransAngles, *see* CFourbar
printJointLimits, *see* CFourbar

qanimate, **189**
QANIMATE_OUTPUTTYPE_DISPLAY, 231
QANIMATE_OUTPUTTYPE_FILE, 232
QANIMATE_OUTPUTTYPE_STREAM, 232

setAngularVel, *see* CFourbar, *see* CCrankSlider, *see* CGearedFivebar, *see* CFourbarSlider, *see* CWattSixbarI, *see* CWattSixbarII, *see* CStevSixbarI, *see* CStevSixbarIII
setCouplerPoint, *see* CFourbar, *see* CCrankSlider, *see* CGearedFivebar, *see* CFourbarSlider, *see* CWattSixbarI, *see* CWattSixbarII, *see* CStevSixbarI, *see* CStevSixbarIII
setGravityCenter, *see* CFourbar, *see* CCrankSlider
setInertia, *see* CFourbar, *see* CCrankSlider
setLinks, *see* CFourbar, *see* CCrankSlider, *see* CGeared-Fivebar, *see* CFourbarSlider, *see* CWattSixbarI, *see* CWattSixbarII, *see* CStevSixbarI, *see* CStevSixbarIII
setMass, *see* CFourbar, *see* CCrankSlider
setNumPoints, *see* CFourbar, *see* CCrankSlider, *see* CGearedFivebar, *see* CFourbarSlider, *see* CWattSixbarI, *see* CWattSixbarII, *see* CStevSixbarI, *see* CStevSixbarIII
sliderAccel, *see* CCrankSlider
  seeCFourbarSlider, 357
sliderPos, *see* CCrankSlider
  seeCFourbarSlider, 358
sliderVel, *see* CCrankSlider
  seeCFourbarSlider, 360
spindleSpeed, 443, *see* CCam
synthesis, *see* CFourbar

transAngle, *see* CFourbar, *see* CCrankSlider
transAngles, *see* CFourbar

unwrap(), 12
uscUnit, *see* CFourbar, *see* CCrankSlider, *see* CGeared-Fivebar, *see* CFourbarSlider, *see* CWattSixbarI, *see* CWattSixbarII, *see* CStevSixbarI, *see* CStevSixbarIII