Pedagogically Effective Programming Environment for Teaching Mechanism Design

Dr. Harry H. Cheng, Assistant Professor
Director of Integration Engineering Laboratory
Department of Mechanical & Aeronautical Engineering
and Graduate Group in Computer Science
University of California
Davis, CA 95616
Email: `hhcheng@ucdavis.edu`
World Wide Web: `http://iel.ucdavis.edu`

Abstract

The C^H programming environment is an open system. Users can enhance the system through its various user interfaces. C^H is specially designed for applications in mechanical systems engineering, although it is applicable to many other disciplines as well. C^H has been successfully used as a teaching and learning tool for an undergraduate course, Computer-Aided Mechanism Design, at the University of California, Davis in Fall 1993. In this paper we will present the C^H programming environment and programming features developed for teaching and student learning. We will describe how a teaching toolbox is developed and used for teaching mechanism design. Source codes in the teaching toolbox are available to students so that they can study the software implementation of algorithms and modify the codes to solve the similar problems. Although the developed teaching toolbox is specific for instruction on mechanism design, the C^H programming environment and ideas presented in this paper are general, and they are applicable to instructional improvement for a wide range of subjects in engineering.

Keywords:

C^H programming environment, computer-aided mechanism design, complex numbers, teaching toolbox.

1 Introduction

Mechanism design is an intriguing subject. It gives students some experience in and physical appreciation of mechanical design. Mechanism design is therefore an important course for undergraduate students, majoring in mechanical design, who prepare either for advanced graduate studies or for entering engineering practice at the bachelor's level. Some basic principles and concepts of the subject can be explained by simple illustrative problems with traditional graphic methods [1, 2]. However, in general, analytical and numerical methods with application of digital computers are needed in order to fully comprehend the subject matter and solve complicated problems. Because of the rapid advance of computer technologies, computational methods for analysis and design of multibody mechanical systems are becoming popular in engineering practice. When students enter engineering practice, they will learn how to use one or more commercial software packages such as ADAMS (Automated Dynamic Analysis of Mechanical Systems), DADS (Dynamic Analysis and

Design System), KINSYN (KINematic SYNthesis) [3], and LINCAGES (Linkage INteractive Computer Analysis and Graphical Enhanced Synthesis Package) [4, citeErdman3] to solve complicated engineering problems. Therefore, in addition to basic principles, it is increasingly important to teach numerical aspects of the subject in class and motivate students to use computational techniques to solve real-world engineering problems. These specialized software packages cannot best serve this purpose, because these packages are supplied as black boxes without source codes. Learning curves for some of these packages are quite steep because users are assumed to have full knowledge of the subject. Most important, these packages are dedicated to specific application areas and they are not suitable for teaching students general basic principles. Menu-driven educational software for animation and design of mechanisms has been developed for students and less experienced users. Although graphics oriented educational software such as the student version of LINCAGES [6, 7] and MechAnimator [8] are simpler than commercial software packages designed for experienced practicing engineers, the numerical aspects and software implementation of mechanism analysis and design cannot be easily appreciated through menu interfaces. The analytical sequences of the algorithm are not transparent to users in a menu-driven software package.

It is valuable for students to develop their analytical skill by translating mathematical formulas into a computer program and solving some practical design problems. However, if students are asked to write their own computer programs starting from scratch in FORTRAN, C, or any other language, it is too time consuming and students will spend most of their time debugging their computer programs rather than studing the subject. They may lose sight of the forest for the trees. Because of the lack of suitable programming environment and the constraint of a short period of instructional time, some instructors abandon the idea of requiring students to "code their own" [9]. Small programs such as `FOURBAR`, `FIVEBAR`, and `DYNAFOUR` are used by students to solve some simple design problems for reinforcing kinematic concepts [9, 10]. The source codes for these program modules are normally not available to students. Even if they were available, the complexity of these programs and large size of source codes, typically written in C or FORTRAN, cannot be easily comprehended by average students within a short period of time.

To teach mechanism design effectively, we have used a teaching toolbox to teach mechanism design. A toolbox is significantly different from specialized commercial software packages. Students can examine the available source code of a toolbox and modify it to solve similar problems. Through this learning-by-examples process, students will better understand the principles and numerical aspects of the subject. In addition, with this kind of toolbox, students can use its high-level building blocks to conveniently build their own software packages to solve complicated practical engineering analysis and design problems. This toolbox can be best implemented by the C^H programming language because it has been especially enhanced for this educational purpose [11-16]. C^H is designed to be a superset of ANSI C; it bridges the gap between C [17, 18] and FORTRAN [19, 20]. Data types such as complex and computational array not only simplify the analysis and design of multibody mechanical systems significantly, but also make the source code more readable. A C^H program for solving mechanism design problem is significantly smaller than that written in C or FORTRAN 77. Hence, students can easily modify the existing source code in a toolbox to solve different problems. Unlike C, C^H is strongly block-structured so that a function can not only call itself recursively, but can also be nested. Therefore, C^H is a modular language and is ideal for the creation of toolboxes for teaching and student learning.

Feedback from students is critical to the development of the C^H programming language, it has reshaped the language in many ways and has scaled up our research into the working technology. In this paper we will present the C^H programming environment and programming features developed for this educational purpose. We will describe how a teaching toolbox is developed and used for instructional improvement on mechanism design at the University of California, Davis.

2 C^H Programming Environment

The C^H programming environment can be introduced with a famous programming output statement

`hello, world`

that was popularized by Kernighan and Ritchie (1978). The level of difficulty in printinf this statement along with other criteria is often used to judge the simplicity and friendliness of a language. Users with previous C or FORTRAN experience may remember that print this statement, one has to go through compilation and link processes to get the executable object code first, then run the program to get the output. For a large program, the `make` utility may have to be used to maintain the program's integrity. However, these compilation and link processes are unnecessary for running a C^H program because C^H is interpretive, with a quick system response at its current implementation. As a specific example, assume that the machine name of a computer is `fool`; the prompt of the screen in the C-shell programming environment under the UNIX operating system is shown below:
fool%
The output from the system, as shown in this system prompt, is displayed in italics in this paper. To invoke the C^H programming environment, one types `ch` on the terminal keyboard. The screen will become:
fool>
This prompt indicates that the system is in the C^H programming environment and is ready to accept the user's terminal keyboard input. It is also possible to set C^H as the default shell so that, whenever the system is logged in, the C^H programming environment will be invoked automatically.

2.1 Command Mode

If the input typed in is syntactically correct, it will be executed successfully. Upon completion of the execution, the system prompt *fool>* appears again, otherwise it prints out the corresponding error messages to assist the user in debugging the program. At the system prompt *fool>*, any UNIX commands such as `cd, ls`, and `pwd` can be executed. In this scenario, C^H is used as a UNIX shell in the same manner as Bourne-shell, C-shell, or Korn-shell. Because C^H is a superset of ANSI C, it is more powerful than these conventional UNIX shells. For example, to get the output `hello, world`, one can type `"hello, world"`. Then, the screen will appear as follows:
fool> `"hello, world"`
hello, world
fool>
where the input typed in from the terminal is in the typewriter font. In C^H, if there is any output from the system resulting from the execution of a command, it will be printed out. In this case, `hello world` is the output from execution of the expression `"hello world"` that is a string value. If an expression is typed in, it will be evaluated by C^H and the result will be printed out immediately. For example, if the expression `1+3*2` is typed in, the output will be 7. If the input is `8`, the output is the same as the input of 8. Any valid C^H expressions can be evaluated in this command mode. Therefore, C^H can be used as a calculator by novice users.

The first lesson that a C programmer learned may be to use the standard I/O function **printf**() to get the output `hello, world`. Because C^H is a superset of C, the output can be obtained by the I/O function **printf**() as follows:
fool> `printf("hello, world")`
hello, world
fool>

2.2 Command Files

An ANSI C program can be executed without compilation in a C^H programming environment. The command-line argument interface in C^H is ANSI C compatible. C programs are called *command files* or simply *commands* in C^H. A command file must start with a comment to distinguish itself from other shell commands. In addition, it must have both read and execute permissions. In a C^H programming environment, a command file can be executed without compilation. For example, one can create a command file named `print` by a text editor. If the program `print` is as follows:

```
/* the first line must be a comment */
main()
{
  printf("hello, world\n");
}
```

One can type the command `print` to get the output of `hello, world` as follows:

fool> `print`
hello, world
fool>

To parse a C^H program without execution, one can use the built-in command `chh`. For example, if the program `print` is

```
/* print */
main()
{
  printf("hello, world\n";
}
```

the error of the program can be diagnosed by the command `chh` as follows:

fool> `chh print`
ERROR: missing)
ERROR: Syntax error at line 4
fool>
where the missing parenthesis for the function **printf**() at line 4 is detected by the system.

2.3 Script Files

Statements, functions, and commands can be grouped as a *script file* or *script* in C^H. Like a command file, a script file must start with a comment and have both read and execute permissions. For example, if the script file `test` contains the following statements:

```
/* test */
int i = 90;
/* copy test to test1 */
cp test test1
printf("i is equal to %d from the script file\n", i);
```

it can be executed interactively as follows:

fool> `test`
i is equal to 90 from the script file
fool>
Or, it can be executed in two separate steps as follows:

fool> `chh test`
fool> `run`
i is equal to 90 from the script file
fool>
where the command `chh test` parses the script file `test` first, and the built-in command `run` then executes the parsed program. After execution of the script file `test`, the program `test` will be copied to a new file named `test1` by the programming statement `cp test test1` in the program `test`.

2.4 Function Files

A $\mathrm{C^H}$ program can be divided into many separate files. Each file consists of many related functions at the top level that are accessible to any part of a program. A file that contains more than one function is usually suffixed with `.ch` to identify itself as part of a $\mathrm{C^H}$ program. Besides command files and script files, there are *function files* in $\mathrm{C^H}$. A *function file* in $\mathrm{C^H}$ is a program that contains only one function definition. A function file must be readable. The extension of a function file is specified by the system built-in string variable `extf`. The names of the function file and function definition inside the function file must be the same. The functions defined using function files are treated as if they were the system built-in functions in a $\mathrm{C^H}$ programming environment. For example, if the system variable `extf` is `".c .ff"` and the program `addition.c` contains the following statements:

```
/**** function file for adding two integers ****/
int addition(int a, b)
{
  int c;
  c = a + b;
  return c;
}
```

it can be invoked automatically to add two integers as shown in the following interactive execution session:

fool> `int i = 9;`
fool> `i = addition(3, i);`
fool> `i`
12
fool>
where the integer value 3 and integer variable i with the value of 9 are added together by the function `addition()` first, the result is then assigned to variable i. In this case, function `addition()` is treated as if it was a built-in function like **sin**() or **cos**().

Table 1: The complex operations

Definition	C$^{\rm H}$ Syntax	C$^{\rm H}$ Semantics
negation	−z	$-x - iy$
addition	z1 + z2	$(x_1 + x_2) + i(y_1 + y_2)$
subtraction	z1 − z2	$(x_1 - x_2) + i(y_1 - y_2)$
multiplication	z1 ∗ z2	$(x_1 * x_2 - y_1 * y_2) + i(y_1 * x_2 + x_1 * y_2)$
division	z1 / z2	$\frac{x_1 * x_2 + y_1 * y_2}{x_2^2 + y_2^2} + i\frac{y_1 * x_2 - x_1 * y_2}{x_2^2 + y_2^2}$
equal	z1 == z2	$x_1 == x_2$ and $y_1 == y_2$
not equal	z1 != z2	$x_1 != x_2$ or $y_1 != y_2$

3 Programming Features Related to the Teaching Toolbox

The ANSI C programming language is not the best language for beginners. Undergraduate students in mechanical engineering usually do not have extensive knowledge about computer hardware and software engineering. Most students taking a course on mechanism design are junior or senior students. They have taken a first course in computer programming using a simple programming language such as FORTRAN 77 or BASIC. Students intend to compare what they are learning with what they have learned before. Therefore, we have tried to merge C and FORTRAN 77 in the C$^{\rm H}$ programming language so that students can treat C$^{\rm H}$ as if it were a FORTRAN programming language. C$^{\rm H}$ is also similar to BASIC because of its interpretive nature. Hence, students can learn advanced programming features of C$^{\rm H}$ at a comfortable pace.

In addition, we have added many new programming features to C$^{\rm H}$. These programming features are not only useful for general-purpose programming but also important for development of teaching toolboxes. A *toolbox* is a directory that contains many relevant command files, script files, and function files described in the previous section. A C$^{\rm H}$ program for solving practical engineering problems can be easily developed by using modules from the toolboxes. The major programming features enhanced in C$^{\rm H}$ for development of teaching toolboxes for mechanical systems design are built-in complex and dual data types, polymorphism, pass-by-reference in functions, nested functions, and computational array [11-16]. These features are simple and easy to comprehend by students who have had a first course in computer programming.

Complex number, an extension of real number, has wide applications in science and engineering. It is useful for analysis and design of planar mechanisms [7, 10]. Because of its importance in scientific programming, numerically oriented programming languages and software packages usually provide complex number support in one way or another. For example, FORTRAN 77 [19], a language mainly for scientific computing, has provided complex data type since its earliest days. C, a modern language originally invented for UNIX system programming [17], does not have complex as a basic data type because numerically oriented scientific computing was not its original design goal. Computations involving complex numbers can be introduced as a data structure in C. However, programming with such a structure is somewhat clumsy, because for each operation a corresponding function has to be invoked.

To simplify the matter, complex is implemented as a built-in basic data type in C$^{\rm H}$. The arithmetic and relational operations for complex numbers are treated in the same manner as those for real numbers in C$^{\rm H}$. The negation of a complex number, and arithmetic and comparison operations for two complex numbers are defined in Table 1, where complex numbers z, z_1, and z_2 are defined as $x + iy, x_1 + iy_1$, and $x_2 + iy_2$, respectively. The built-in complex functions related to

Table 2: The syntax and semantics of built-in complex functions for planar mechanism design.

$\mathrm{C^H}$ Syntax	$\mathrm{C^H}$ Semantics
sizeof(z)	8
abs(z)	sqrt(x^2+y^2)
real(z)	x
imaginary(z)	y
complex(x, y)	$x+iy$
conjugate(z)	$x-iy$
arg(z)	Θ; $\Theta = \text{atan2}(y, x)$
polar($r, theta$)	$r\cos(theta) + ir\sin(theta)$
sqrt(z)	sqrt(sqrt(x^2+y^2))($\cos\frac{\Theta}{2} + i\sin\frac{\Theta}{2}$); $\Theta = \text{atan2}(y, x)$

planar mechanism design are listed in Table 2 along with their definitions. The input arguments of these functions can be complex numbers, variables, or expressions. There are more built-in complex functions available in $\mathrm{C^H}$. Further details about handling of complex numbers in $\mathrm{C^H}$ have been presented by Cheng [13, 15]. In the following sections, we will describe how complex numbers in $\mathrm{C^H}$ are used for analytical treatment of planar mechanism design.

4 Analysis of a Four-Bar Linkage

The simplest closed-loop linkage is the four-bar linkage, as shown in Figure 1. In this section we describe how programs in the teaching toolbox are used for analysis of the four-bar linkage. The ideas presented in this section, however, are applicable to more complicated planar mechanisms as well.

4.1 Position Analysis

For the four-bar linkage shown in Figure 1, the displacement analysis can be formulated by the following loop-closure equation

$$\mathbf{r}_2 + \mathbf{r}_3 = \mathbf{r}_1 + \mathbf{r}_4. \tag{1}$$

Using complex numbers, equation (1) becomes

$$r_2 e^{i\theta_2} + r_3 e^{i\theta_3} = r_1 e^{i\theta_1} + r_4 e^{i\theta_4}, \tag{2}$$

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) can be rearranged as

$$r_3 e^{i\theta_3} - r_4 e^{i\theta_4} = r_1 e^{i\theta_1} - r_2 e^{i\theta_2}. \tag{3}$$

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 (3) becomes the following general complex equation

$$R_1 e^{i\phi_1} + R_2 e^{i\phi_2} = z. \tag{4}$$

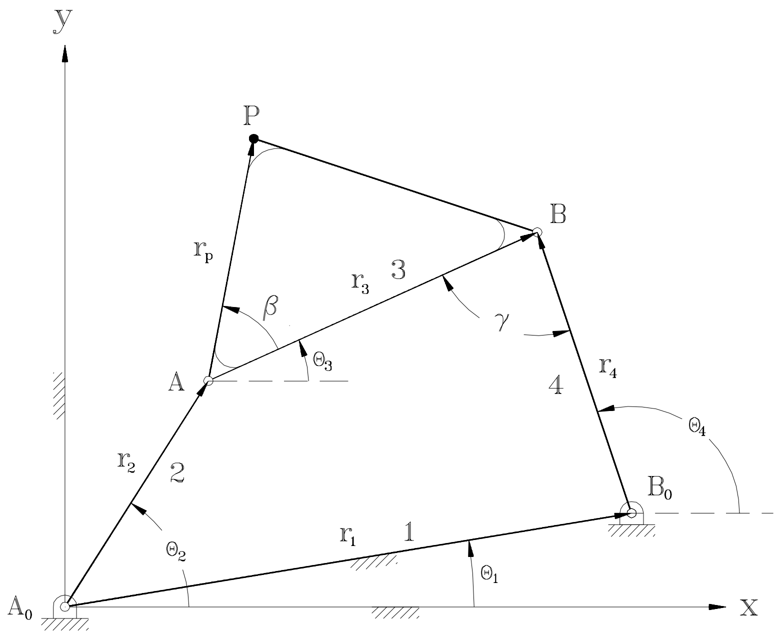

Figure 1: The four-bar linkage.

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 as follows. From equation (4), we get

$$\cos\phi_1 = \frac{x_3 - R_2\cos\phi_2}{R_1}, \quad \sin\phi_1 = \frac{y_3 - R_2\sin\phi_2}{R_1} \tag{5}$$

Substituting these results into the identity equation $\sin^2\phi_1 + \cos^2\phi_1 = 1$ and simplifying the resultant equation, we get

$$y_3\sin\phi_2 + x_3\cos\phi_2 = \frac{x_3^2 + y_3^2 + R_2^2 - R_1^2}{2R_2} \tag{6}$$

From this equation, we can derive the formulas for ϕ_1 and ϕ_2 as follows:

$$\phi_2 = \operatorname{atan2}(y_3, x_3) \pm \operatorname{acos}\left(\frac{x_3^2 + y_3^2 + R_2^2 - R_1^2}{2R_2\sqrt{x_3^2 + y_3^2}}\right) \tag{7}$$

$$\phi_1 = \operatorname{atan2}(\sin\phi_1, \cos\phi_1) \tag{8}$$

where $\sin\phi_1$ and $\cos\phi_1$ are computed using equation (5). The above derivation of analytical formulas (7) and (8) for ϕ_2 and ϕ_1 in equation (4) is quite simple in comparison with presentations in most contemporary mechanism design textbooks.

In general, there are two sets of solutions for θ_3 and θ_4 for a given θ_2 as shown in equation (7), which correspond to two different circuits or two geometric inversions of a circuit of the four-bar linkage [7]. Equation (4) is a general complex equation. Many analysis and design problems for planar mechanisms can be formulated in this form. Because it is a complex equation that 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 from this equation. To assist student learning, we have developed a function file `complexSolver.ff` in the teaching toolbox to solve this complex equation. The function prototype for the function `complexSolver()` is given below, Program 1: Program `fourbarTheta3Theta4P` for computing θ_3, θ_4 and position of the coupler point P of a four-bar linkage.

```
/* fourbarTheta3Theta4P */
main()
{
  float r[5], theta[5], rp, beta;
  int n1 = 2,  n2 = 4;               /* positions of two unknowns */
  complex firstz, secondz, z, P;
  /* specification of the four-bar linkage */
  r[1] = 12; r[2] = 4; r[3] = 12; r[4] = 7; rp = 5; beta = 20*PI/180;
  theta[1] = 10*PI/180; theta[2]=70*PI/180;
  z = polar(r[1], theta[1]) - polar(r[2], theta[2]);      /* z = r1-r2 */
  firstz = complexSolver(n1, n2, r[3], -r[4], z, secondz);/* find theta3, theta4 */
  /**** the first set of solutions ****/
  theta[3] = real(firstz); theta[4] = imaginary(firstz);
  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] = real(secondz); theta[4] = imaginary(secondz);
  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);
}
```

```
complex complexSolver(int n1, n2; float firstknown, secondknown;
                      complex z, &secondz)
```

where parameter `n1` is the position of the first of two unknowns to be obtained on the left-hand side of equation (4) , `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 (4), `secondknown` is the value of the second known, and `z` is the complex number on the right-hand side of equation (4). The return value from this routine is a complex number. The complex format is for convenience only. The real part of the return complex number is the first unknown whereas the imaginary part is the second unknown in equation (4). If either ϕ_1 or ϕ_2 is to be found, there are two sets of solutions for equation (4). In this case, the second set of solutions is passed to the calling function from `complexSolver()` through the complex parameter `secondz` by reference [11].

With this general complex equation solver, a 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 1, are given as follows: $r_1 = 12'', r_2 = 4'', r_3 = 12'', r_4 = 7''$. The phase angle for the ground link is $\theta_1 = 10°$, the coupler point P is defined by the distance $r_p = 5''$ 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°.

Problem 1 can be easily solved by using a $\mathrm{C^H}$ program named `fourbarTheta3Theta4P` shown in Program 1. The position of coupler point P shown in Figure 1 can be expressed in vector form using complex numbers as:

$$P = r_2 e^{i\theta_2} + r_p e^{i(\theta_3+\beta)} \tag{9}$$

A complex number $z = (x, y) = re^{i\theta}$ in C$^{\text{H}}$ can be constructed either by `complex(x,y)` or `polar(r,theta)`. Equation (9) can be translated into a C$^{\text{H}}$ programming statement

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

as shown in Program 1.

The four-bar linkage given in Problem 1 is a crank-rocker. There are two distinct circuits for each input link position. 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 1. Arrays in C$^{\text{H}}$ are fully ANSI C compatible, they are pointers. For the convenience of student learning and consistency with text description, we use arrays with index starting from 1, instead of 0, in the teaching toolbox. Computational arrays with adjustable initial index will be implemented in C$^{\text{H}}$ in the future. In Program 1, link parameters r_i and θ_i are treated as elements of `r[i]` and `theta[i]` of arrays `r[ ]` and `theta[]`, respectively. The output from Program 1 is as follows:

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

According to the IEEE 754 standard for binary floating-point arithmetic [21], any invalid solution in C$^{\text{H}}$ 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 $r_1 = 12'', r_2 = 12'', r_3 = 4'', r_4 = 7''$. The linkage then becomes a double-rocker. There are two circuits, each with two geometric 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 C$^{\text{H}}$ program. If the following programming statement

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

in Program 1 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)

A four-bar linkage may take the so-called crank-rocker, double-crank (drag-link), double-rocker, or triple-rocker [7]. 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. To assist student learning, we have developed a function file `grashof.ff` in the teaching toolbox. The function prototype for `grashof()` is as follows:

```
int grashof( float r[], theta1, inputlimits1[], inputlimits2[],
             outputlimits1[], outputlimits2[]; char *grashof_filestr)
```

where array `r[ ]` contains length of each link of the linkage, `theta1` stands for θ_1. Both array `r[ ]` and `theta1` are input to the function. The other parameters are related to the output from the function. Elements `inputlimit1[0]` and `inputlimit1[1]` of array `inputlimit1[ ]` contain the lower and upper limits of the input link 2 for the first branch of the linkage, elements `inputlimit2[0]` and `inputlimit2[1]` contain the lower and upper limits of the input link 2 for the second branch of the linkage, respectively. Similarly, elements `outputlimit1[0]` and `outputlimit1[1]` of array `inputlimit1[ ]` contain the lower and upper limits of the output link 4 for the first branch of the linkage, whereas `outputlimit1[0]` and `outputlimit1[1]` contain the lower and upper limits of the output link 4 for the second branch of the linkage, respectively. Parameter `grashof_filestr` can be used to direct the output from the function `grashof()` to either a file or the screen. The function returns the number of distinct input ranges for the given link dimensions. If four bars cannot form a valid linkage, the return value is 0; if the linkage is a rocker-crank or double-rocker, the return value is 2; otherwise, the return value is 1. How this function in the teaching toolbox has been used for mechanism design can be demonstrated by the following mechanism design problem.

> **Problem 2**: The link lengths of a four-bar linkage, as shown in Figure 1, are given as follows: $r_1 = 12'', r_2 = 4'', r_3 = 12'', r_4 = 7''$. The phase angle for link 1 is $\theta_1 = 10°$, the coupler point P is defined by distance $r_5 = 5''$ 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}.

We can enhance Program 1 to solve this mechanism design problem. Program 1 has been expanded in Program 2. The output at the top of Figure 2 is produced by the function `grashof()`. It indicates that the four-bar linkage is a crank-rocker with a complete 360° rotation for the input link 2. The output ranges $77.98° \leq \theta_4 \leq 150.16°$ and $315.44° \leq \theta_4 \leq 355.64°$ of the output link 4 for two distinct circuits are also printed out. If there are multiple input ranges, the function `couplerCurve()` will be called more than once to generate coupler curves for coupler point P. For Problem 2, the function `couplerCurve()` is called only once because the input range $0 \leq \theta_2 \leq 360°$ is the same for two distinct circuits, that is, the return value `num_range` of the function `grashof()` is 1. These input and output ranges are also saved in arrays `inputlimit1[ ]`, and `outputlimit1[ ]` and `outputlimit2[ ]`, respectively. The lower and upper input limits `inputlimit1[1]` and `inputlimit1[2]` for link 2 are passed to the function `couplerCurve()` for generating coupler curves. Similar to Program 1, angles θ_3 and θ_4 corresponding to θ_2 are calculated by the function `complexSolver()`. The coupler curve for point P is generated when θ_2 is rotated from θ_{2min} to θ_{2max} using a for-loop with an increment step size of 1°. The data for coupler curves of different circuits are saved in files `branch1.out` and `branch2.out` first. The coupler curves are then plotted by the plot routine `plotXYR()` using the data from these two files. The function `plotXYR()` is called from the function file `plotXYR.ff` in the teaching toolbox. The first argument of the function `plotXYR()` is the name of the file that contains data for plotting. The second argument is the title of the plot. The third and fourth arguments are labels for the x-axis and y-axis, respectively. The fifth, sixth, and seventh arguments are the inital value , final value, and incremental step size for x-axis, respectively. The last three arguments are for the y-axis.

4.2 Velocity and Acceleration Analysis

4.2.1 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 (3), we get the following velocity relation

$$\omega_3 r_3 e^{i\theta_3} - \omega_4 r_4 e^{i\theta_4} = -\omega_2 r_2 e^{i\theta_2} \tag{10}$$

for the four-bar linkage shown in Figure 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 `complexSolver()` to compute angular velocities ω_3 and ω_4 for coupler and output

Program 2: Program `fourbarCouplerCurve` for generating coupler curves of a four-bar linkage.

```
/* fourbarCouplerCurve */
main()
{
  float r[5], theta[5], rp, beta;         /* dimension shown in Figure 1 */
  float inputlimit1[2], inputlimit2[2];   /* theta2_min and theta2_max */
  float outputlimit1[2], outputlimit2[2]; /* theta4_min and theta4_max */
  int num_range;                          /* number of input ranges */
  void couplerCurve(float r[], theta[], rp, beta, theta2_min, theta2_max);
  /* specification of the four-bar linkage */
  r[1] = 12; r[2] = 4; r[3] = 12; r[4] = 7; rp = 5; beta = 20*PI/180;
  theta[1] = 10*PI/180;

  num_range = grashof(r,theta[1],inputlimit1,inputlimit2,
                      outputlimit1,outputlimit2,NULL);
  if(num_range >= 1)
    couplerCurve(r, theta, rp, beta, inputlimit1[0], inputlimit1[1]);
  if(num_range == 2)
    couplerCurve(r, theta, rp, beta, inputlimit2[0], inputlimit2[1]);
}

/**** this routine computes and plots coupler courves ****/
void couplerCurve(float r[], theta[], rp, beta, theta2_min, theta2_max)
{
  FILE *stream1, *stream2;
  int n1 = 2,  n2 = 4;
  complex firstz, secondz, z, P;

  stream1 = fopen("branch1.out", "wr");  /* open file branch1.out  */
  stream2 = fopen("branch2.out", "wr");  /* open file branch2.out  */
  /* loop for theta2, theta2_min < theta2 < theta2_max, with step size 1 deg */
  for(theta[2]=theta2_min; theta[2]<=theta2_max; theta[2] += PI/180)
  {
    z = polar(r[1], theta[1]) - polar(r[2], theta[2]);
    firstz = complexSolver(n1, n2, r[3], -r[4], z, secondz);
    /**** the first set of solutions ****/
    theta[3] = real(firstz);
    P = polar(r[2], theta[2]) + polar(rp, theta[3]+beta);
    fprintf(stream1, "%f %f\n", real(P), imaginary(P));
    /**** the second set of solutions ****/
    theta[3] = real(secondz);
    P = polar(r[2], theta[2]) + polar(rp, theta[3]+beta);
    fprintf(stream2, "%f %f\n", real(P), imaginary(P));
  }
  fclose(stream1);  fclose(stream2);          /* close files */
  plotXYR("branch1.out", "Coupler curve for the 1st branch",
         "Px (inches)", "Py (inches)", -3, 7, 2, 0, 10, 2);
  plotXYR("branch2.out", "Coupler curve for the 2nd branch",
         "Px (inches)", "Py (inches)", 0, 10, 2, -5, 5, 2);
  remove("branch1.out");
  remove("branch2.out");
}
```

```
Grashof type: Crank-Rocker
Input Characteristics: Input 360 degree rotation
Output Range:
        Branch:          1       2
                         (deg)   (deg)
        Lo limit:        77.98   315.44
        Hi limit:        150.16  355.64
```

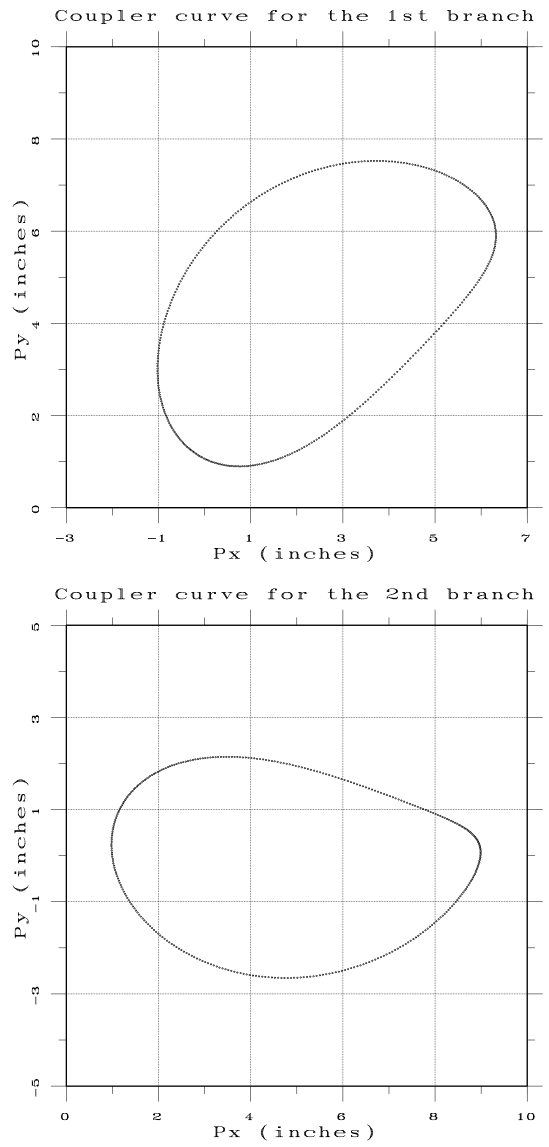

Figure 2: The output from Program 1.

links, respectively. We can also derive analytical solutions for ω_3 and ω_4. Multiplying equation (10) with $e^{-i\theta_4}$, equation (10) becomes [22]

$$\omega_3 r_3 e^{i(\theta_3-\theta_4)} - \omega_4 r_4 = -\omega_2 r_2 e^{i(\theta_2-\theta_4)} \tag{11}$$

The imaginary part of equation (11) gives

$$\omega_3 r_3 \sin(\theta_3 - \theta_4) = -\omega_2 r_2 \sin(\theta_2 - \theta_4) \tag{12}$$

Then

$$\omega_3 = -\frac{\omega_2 r_2 \sin(\theta_4 - \theta_2)}{r_3 \sin(\theta_4 - \theta_3)} \tag{13}$$

Computation of the angular velocity ω_3 can be programmed in $\mathrm{C^H}$ as function files or they can use a single line of code. For example, ω_3 can be calculated in $\mathrm{C^H}$ using a function `fourbarOmega3()` as follows:

```
float fourbarOmega3(float r[], theta[], omega[])
{
  float a, b;
  a = -r[2]*omega[2]*sin(theta[4]-theta[2]);
  b = r[3]*sin(theta[4]-theta[3]);
  return a/b;
}
```

Similarly, the following analytical expression for ω_4 can be derived by multiplying equation (10) with $e^{-i\theta_3}$,

$$\omega_4 = \frac{\omega_2 r_2 \sin(\theta_3 - \theta_2)}{r_4 \sin(\theta_3 - \theta_4)} \tag{14}$$

ω_4 can be calculated in $\mathrm{C^H}$ using a function called `fourbarOmega4()`.

The derivative of equation (9) gives the following analytical expression for the velocity of coupler point P.

$$V_p = i\omega_2 r_2 e^{i\theta_2} + i\omega_3 r_p e^{i(\theta_3+\beta)} \tag{15}$$

which can be translated into a $\mathrm{C^H}$ code fragment as

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

4.2.2 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, taking second derivative of the loop-closure equation (3), we get the following acceleration relation for the four-bar linkage shown in Figure 1.

$$i\alpha_3 r_3 e^{i\theta_3} - \omega_3^2 r_3 e^{i\theta_3} - i\alpha_4 r_4 e^{i\theta_4} + \omega_4^2 r_4 e^{i\theta_4} = i\alpha_2 r_2 e^{i\theta_2} + \omega_2^2 r_2 e^{i\theta_2} \tag{16}$$

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

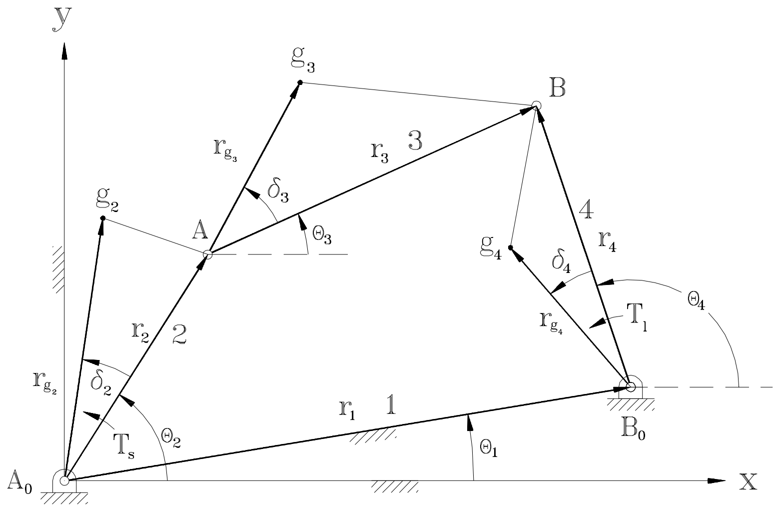

Figure 3: The four-bar linkage with offset gravity centers for moving links.

be derived:

$$\alpha_3 = \frac{-r_2\alpha_2 \sin(\theta_4-\theta_2) + r_2\omega_2^2\cos(\theta_4-\theta_2) + rr_3\omega_3^2\cos(\theta_4-\theta_3) - r_4\omega_4^2}{r_3\sin(\theta_4-\theta_3)} \tag{17}$$

$$\alpha_3 = \frac{r_2\alpha_2 \sin(\theta_3-\theta_2) - r_2\omega_2^2\cos(\theta_3-\theta_2) + r_4\omega_4^2\cos(\theta_3-\theta_4) - r_3\omega_3^2}{r_4\sin(\theta_3-\theta_4)} \tag{18}$$

Two functions `fourbarAlpha3()` and `fourbarAlpha4()` have been written for calculating α_3 and α_4, respectively. They are included in the teaching toolbox.

4.3 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. For this entry-level undergraduate mechanism design course, we choose to use matrix method [7]. To simplify the programming burden, we have implemented computational arrays in the C^H programming language. Computational arrays can be treated as single objects.

For the four-bar linkage shown in Figure 3, 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 4. 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

$$F_{12x} + F_{32x} + F_{g_2x} = 0 \tag{19}$$

$$-m_2 g + F_{12y} + F_{32y} + F_{g_2y} = 0 \tag{20}$$

$$T_s + (-\mathbf{r}_{g_2}) \times \mathbf{F}_{12} + (\mathbf{r}_2 - \mathbf{r}_{g_2}) \times \mathbf{F}_{32} + T_{g_2} = 0 \tag{21}$$

where $\mathbf{r}_{g_2} = r_2 e^{i(\theta_2+\delta_2)}$ 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.

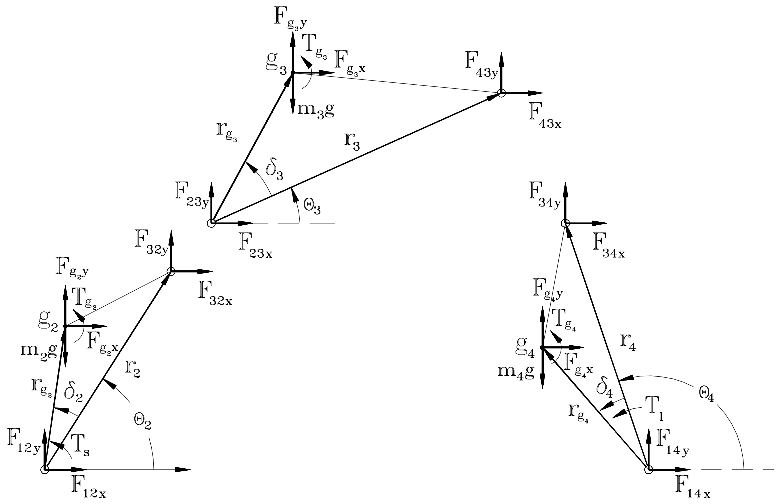

Figure 4: Free body diagrams for the moving links of the four-bar linkage.

For link 3, we get

$$
\begin{aligned}
F_{23x} + F_{43x} + F_{g_3x} &= 0 && (22)\\
-m_3 g + F_{23y} + F_{43y} + F_{g_3y} &= 0 && (23)\\
(-\mathbf{r}_{g_3}) \times \mathbf{F}_{23} + (\mathbf{r}_3 - \mathbf{r}_{g_3}) \times \mathbf{F}_{43} + T_{g_3} &= 0 && (24)
\end{aligned}
$$

where $\mathbf{r}_{g_3} = r_3 e^{i(\theta_3+\delta_3)}$ 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

$$
\begin{aligned}
F_{34x} + F_{14x} + F_{g_4x} &= 0 && (25)\\
-m_4 g + F_{34y} + F_{14y} + F_{g_4y} &= 0 && (26)\\
(-\mathbf{r}_{g_4}) \times \mathbf{F}_{14} + (\mathbf{r}_4 - \mathbf{r}_{g_4}) \times \mathbf{F}_{34} + T_{g_4} + T_l &= 0 && (27)
\end{aligned}
$$

where $\mathbf{r}_{g_4} = r_4 e^{i(\theta_4+\delta_4)}$ 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 (21), (24), and (27) can be expressed explicitly as

$$
\begin{aligned}
T_s - r_{g_2} - r_{g_2}\cos(\theta_2+\delta_2)F_{12y} + r_{g_2}\sin(\theta_2+\delta_2)F_{12x} & \\
+[r_2\cos\theta_2 - r_{g_2}\cos(\theta_2+\delta_2)]F_{32y} - [r_2\sin\theta_2 - r_{g_2}\cos(\theta_2+\delta_2)]F_{32x} + T_{g_2} &= 0 && (28)\\
-r_{g_3} - r_{g_3}\cos(\theta_3+\delta_3)F_{23y} + r_{g_3}\sin(\theta_3+\delta_3)F_{23x} & \\
+[r_3\cos\theta_3 - r_{g_3}\cos(\theta_3+\delta_3)]F_{43y} - [r_3\sin\theta_3 - r_{g_3}\cos(\theta_3+\delta_3)]F_{43x} + T_{g_3} &= 0 && (29)\\
-r_{g_4} - r_{g_4}\cos(\theta_4+\delta_4)F_{14y} + r_{g_4}\sin(\theta_4+\delta_4)F_{14x} & \\
+[r_4\cos\theta_4 - r_{g_4}\cos(\theta_4+\delta_4)]F_{34y} - [r_4\sin\theta_4 - r_{g_4}\cos(\theta_4+\delta_4)]F_{34x} + T_{g_4} &= 0 && (30)
\end{aligned}
$$

Note that $F_{ijx} = -F_{jix}$ and $F_{ijy} = -F_{jiy}$, equations (19-27) 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

$$\mathbf{A}\mathbf{x} = \mathbf{b} \qquad (31)$$

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, $\mathbf{b} = (F_{g_2x}, F_{g_2y} - m_2 g, T_{g_2}, F_{g_3x}, F_{g_3y} - m_3 g, T_{g_3}, F_{g_4x}, F_{g_4y} - m_4 g, T_{g_4} + T_l)^T$ is a vector that contains external load plus inertia forces and inertia torques, and $\mathbf{A}$ is a 9x9 square matrix formed using the angular position of each link and link parameters. What distinguish the above-derived equations (19-27) from those in Erdman and Sandor [7] 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 (31) has 9 unknowns, it should be solved numerically. This can be easily implemented in C$^{\mathrm{H}}$ by only a single line of code shown below,

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

A function file `fourbarForce.ff` has been written for the teaching toolbox. The program appears as follows:

```
void fourbarForce(float r[],theta[],rg[],delta[],omega[],alpha[],m[],ig[],tl;
                  array float X[:])
{
   array float A[9][9], B[9];
   /*  compute matrix A and vector b */
   ...
   X = inverse(A)*B;
}
```

Function `fourbarForce()` can calculate the joint forces and required input torque to achieve the desired motion of the four-bar linkage. The first eight input arguments of the function `fourbarForce()` are arrays, `r[i]` for link length r_i, `theta[i]` for joint angle θ_i, `rg[i]` and `delta[i]` for the position vector $r_i e^{i\delta_i}$ of the center of gravity, `omega[i]` for angular velocity ω_i, `alpha[i]` for angular acceleration α_i, `m[i]` for mass m_i, and `ig[i]` for mass moment of inertia I_{g_i}. `tl` is the external load T_l. The output `X` from the function `fourbarForce()` contains the joint forces and required input torque, which is passed to the calling function as an argument of computational array using an assumed-shape array [11, 20]. How this function in the teaching toolbox has been used in the class can be demonstrated by the following mechanism design problem given in [7].

Problem 3: Link parameters and inertia properties of a four-bar linkage, as shown in Figure 3, are given in the chart below.

Link	Length r (in)	Weight (lbf)	I_g (lbf in sec^2)	C. G. r_g (in)	δ
1	12	—	—	—	—
2	4	0.8	0.012	2	0
3	12	2.4	0.119	6	0
4	7	1.4	0.038	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.

Program 1 can be enhanced to solve Problem 3. Program 3 modified from Program 1 is simple and easy to understand. The angular positions, velocities, and accelerations for coupler and output

Program 3: Program `fourbarForceTorque` for computing joint reaction forces and required input torque.

```
/* fourbarForceTorque */
main()
{
  float r[5],theta[5],rg[5],delta[5],omega[5],
        alpha[5],m[5],ig[5],tl;
  array float X[9];
  int n1 = 2, n2 = 4;/* positions of two unknowns */
  complex firstz, secondz, z;
  float g=32.2*12;
  /* initialization of link parameters and
     inertia properties */
  r[1] = 12; r[2] = 4; r[3] = 12; r[4] = 7;
  theta[1] = 0; theta[2]=150*PI/180;
  rg[2] = 2; rg[3] = 6; rg[4] = 3.5;
  delta[2] = 0.0; delta[3] = 0; delta[4] = 0.0;
  omega[2] = 5;  alpha[2] = -5;
  m[2] = 0.8/g; m[3] = 2.4/g; m[4] = 1.4/g;
  ig[2] = 0.012; ig[3] = 0.119; ig[4] = 0.038; tl=0;

  /* z = r1-r2 */
  z = polar(r[1], theta[1]) - polar(r[2], theta[2]);
  /* find theta3, theta4 */
  firstz = complexSolver(n1,n2,r[3],-r[4],z,secondz);
  theta[3] = real(firstz);
  theta[4] = imaginary(firstz);
  /* find omega3 */
  omega[3] = fourbarOmega3(r, theta, omega);
  /* find omega4 */
  omega[4] = fourbarOmega4(r, theta, omega);
  /* find alpha3 */
  alpha[3] = fourbarAlpha3(r, theta, omega, alpha);
  /* find alpha4 */
  alpha[4] = fourbarAlpha4(r, theta, omega, alpha);
  /* find forces, torque */
  fourbarForce(r, theta, rg, delta, omega, alpha,
               m, ig, tl, X);
  printf("X = %.4f \n", X);
}
```

links in Program 3 are calculated as described in the previous section. The joint reaction forces and input torque are obtained by the function `fourbarForce()`. The output from Program 3 is given:

X = 1.7998 2.3564 1.6998 1.5902 1.1648 -0.7431 -1.0278 2.1635 -10.3952

If the gravitation forces in function `fourbarForce()` for moving links are ignored as in Erdman and Sandor [7], the output from Program 3 will become

X = 0.7040 0.0003 0.6040 0.0341 0.0690 0.1008 0.0680 -0.0805 -1.4276

5 Using Tool-Box for Mechanism Analysis and Design

The four-bar linkage is studied in great detail in the course of mechanism design as described in the previous section. Programs presented in the previous section are some of examples in the teaching toolbox; many more are available to students. These programs can be easily modified by students to solve various analysis and design problems of the four-bar linkage. For example, equation (31) in section 4.2.2 is formulated to find the required input torque to achieve the desired motion (position, velocity, and acceleration) under the external load T_l. It can also be reformulated from equations (19-27) to calculate the possible external load T_l with a given input torque T_s. In this case, Program 3 can still be used, but function `fourbarForce()` needs to be modified. As another example, students can combine Programs 2 and 3 to solve quite complicated analysis and design problems for the four-bar linkage. In fact, there is a program called `fourbarTorque` in the teaching toolbox. It can calculate the required input torque for the linkage specified in Problem 3. Unlike in Problem 3, the input link 2 is rotated at a constant angular velocity $\omega_2 = 5$ rad/sec in the program `fourbarTorque`. The output from the program `fourbarTorque` is shown in Figure 5. Students appreciate the fact that the input torque at different time stamp is important for model-based real-time control of the mechanical system [23]. The program `fourbarTorque` can be easily modified, with changing only one output statement, to obtain plots of joint reaction forces versus input angle θ_2. If joint reaction forces exceed the maximum allowable bearing forces, students can balance the linkage with counterweights to reduce the joint reaction forces. In many cases, the force-balanced linkage may also reduce the maximum required input torque.

As described in the previous section, Program 1 named `fourbarTheta3Theta4P` can print out the angular positions θ_3 and θ_4 as well as the position vector for coupler point P for a given four-bar linkage at a specified input angle for θ_2, Program 2 named `fourbarCouplerCurve` can plot the coupler curve as shown in Figure 2. The program `fourbarTorque` can plot the required input torque as shown in Figure 5. These three related programs can be used to form a single script file or a command file as described in section 2. For example, if the program named `fourbarScript` contains the following programming statements

```
/* fourbarScript */
fourbarTheta3Theta4P
fourbarCouplerCurve
fourbarTorque
```

these three programs can be executed by only running the program `fourbarScript` in a $\mathrm{C^H}$ programming environment. In this case, the program `fourbarScript` is treated as a UNIX shell script file.

Programs in the teaching toolbox can also be easily modified to solve other mechanisms. For example, displacement analysis of the slider-crank mechanism can be performed similarly using the function `complexSolver()`, as shown in the solution procedure for the following problem.

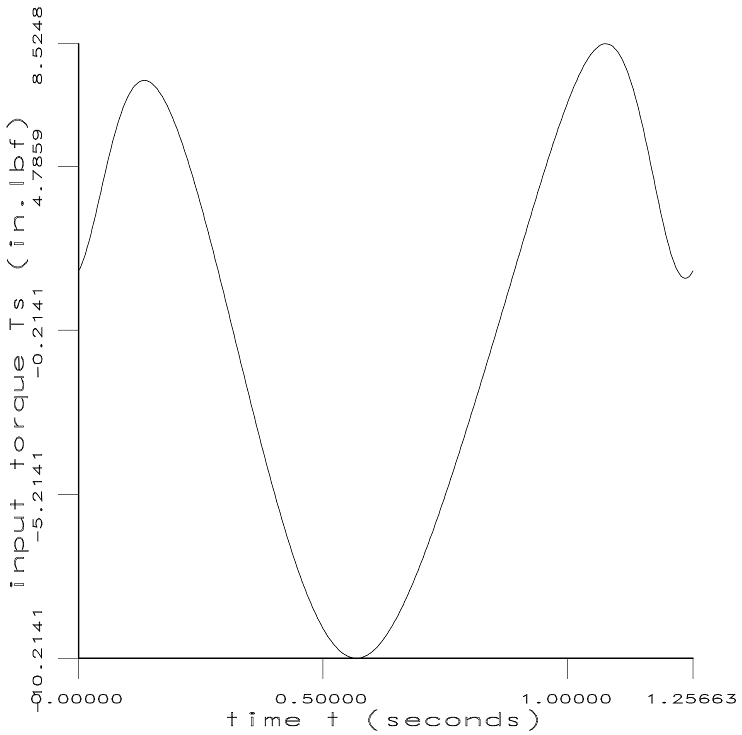

Figure 5: The required input torque versus time for the four-bar linkage.

Problem 4: For the slider-crank mechanism shown in Figure 6, if $r_2 = 1'', r_3 = 5'', r_p = 2''$, and $\beta = 30°$, find the displacement r_1 of the slider and position (x_p, y_p) of coupler point P when $\theta_2 = 45°$.

The loop-closure equation for the slider-crank mechanism shown in Figure 6 can be formulated as

$$r_1 e^{i\theta_1} = r_2 e^{i\theta_2} + r_3 e^{i\theta_3} \tag{32}$$

Equation (32) can be written in the standard form (4),

$$r_1 e^{i\theta_1} - r_3 e^{i\theta_3} = r_2 e^{i\theta_2} \tag{33}$$

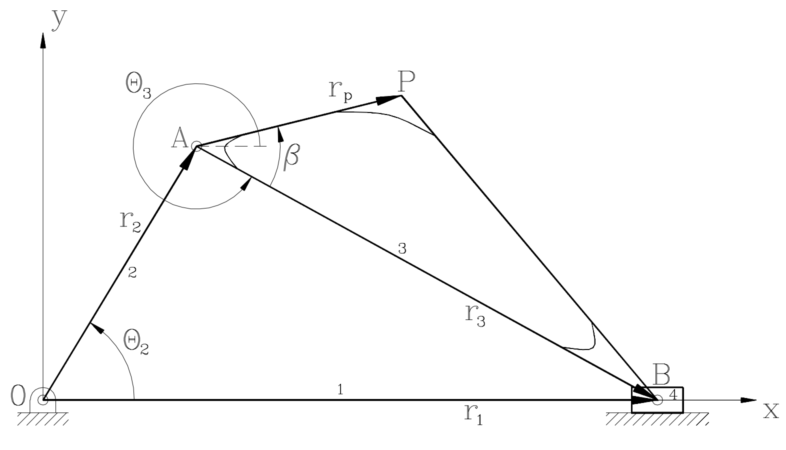

Figure 6: The slider-crank mechanism.

Program 4: Program `slidercrankR1Theta3P` for computing r_1, θ_3 and position of the coupler point P of a slider-crank mechanism.

```
/* slidercrankR1Theta3P */
main()
{
  float r[5], theta[5], rp, beta;
  int n1 = 1,  n2 = 4;              /* positions of two unknowns */
  complex firstz, secondz, P;
  r[2] = 1; r[3] = 5;  rp = 2; beta = 30*PI/180;
  theta[1] = 0; theta[2]=45*PI/180;
  firstz = complexSolver(n1, n2, theta[1], -r[3], polar(r[2], theta[2]), secondz);
  /**** the first set of solutions ****/
  r[1] = real(firstz);   theta[3] = imaginary(firstz);
  P = polar(r[2], theta[2]) + polar(rp, theta[3]+beta); /* P=r2+rp */
  printf("r1 = %6.3f, theta3 = %6.3f, P = %6.3f \n", r[1], theta[3], P);
  /**** the  second set of solutions ****/
  r[1] = real(secondz);   theta[3] = imaginary(secondz);
  P = polar(r[2], theta[2]) + polar(rp, theta[3]+beta); /* P=r2+rp */
  printf("r1 = %6.3f, theta3 = %6.3f, P = %6.3f \n", r[1], theta[3], P);
}
```

Unknown variables r_1 and θ_3 in equation (33) can be readily solved using function `complexSolver()`. Program 1 can be easily modified to solve Problem 4. The output of Program 4 that is modified from Program 1 is as follows:
r1 = 5.657, theta3 = -0.142, P = complex(2.563, 1.452)
r1 = -4.243, theta3 = -3.000, P = complex(-0.866,-0.528)

The offset slider-crank mechanism and its kinematic inversions such as rotating-slide, oscillating slide, and stationary slide can be treated similarly. Program 4 can be modified to handle these variations of the slider-crank mechanism with options provided by the user through the command-line interface of the main routine `main(int argc, char **argv)`. More complicated single-loops such as geared five-bar linkage can be analyzed in the same manner. The complexity of a mechanism increases when a four-bar linkage is combined with a slider-crank mechanism or its kinematic inversions. But it can be easily handled by a computer program. For example, if the crank in Figure 6 is attached to the output link of the four-bar linkage in Figure 1, the displacement for the slider can be formulated by two equations (3) and (33), which can be solved numerically with a program modified from Programs 1 and 4.

Analysis of more complicated mechanisms such as Watt six-bar linkage and Stephenson six-bar linkage can be performed in the same manner. For example, there are multiple loops for a Stephenson six-bar linkage, two loop-closure equations are required. In this case, the function `complexSolver()` will be called twice in order to solve the displacement analysis problem. Several programs for analysis and design of these popular mechanisms have been developed in the teaching toolbox for student learning. Students can easily modify the source codes to solve the various analysis and design problems.

6 Conclusions

The C^H programming environment is designed for both experienced and *novice* users. It has been designed as a superset of ANSI C with incorporation of programming features of FORTRAN and many other programming languages and software packages. The C^H programming environment and programming features related to teaching and student learning have been presented in this paper. To accelerate the process of developing C^H as a pedagogically effective programming environment for teaching and learning, a pilot teaching project — using C^H in an undergraduate course, Computer-Aided Mechanism Design — was conducted at the University of California, Davis in Fall 1993. At the end of the project, we conducted anonymous student evaluations in using the C^H programming language. In summary, students like the simplicity of the C^H programming language, its similarity to FORTRAN, built-in complex data type, computational arrays, a rich set of built-in polymorphic functions, and fast interactive system response without compilation [24]. Most students in the class even liked C^H better than C and FORTRAN, they wanted to continue to use C^H as their primary language for scientific programming. Considering the fact that this is the first time C^H was used in a classroom teaching, the future of developing C^H as a pedagogically effective teaching and learning tool is very promising.

Although the presentation about integration of C^H with mechanism design is specific, the C^H programming environment and ideas presented in the paper are general, and they are applicable to teaching of many other subjects in engineering as well. Currently, we are developing more teaching toolboxes for instructional improvement at both undergraduate and graduate levels.

7 Acknowledgments

This work was supported in part by the National Science Foundation as a Research Initiation Award under Grant DMII-9309207 and by the University of California, Davis through an undergraduate instructional improvement grant, a faculty research grant, and a junior faculty research fellowship.

8 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. Rubel, A. J., and Kaufman, R. E., KINSYS III: A New Human-Engineered System for Interactive Computer-Aided Design of Planar Linkages, *J. of Eng. for Industry*, Vol. 99, No. 2, 1977, pp. 440-448.
4. Erdman, A. G., and Gustafson, J. E., LINCAGES: Linkage INteractive Computer Analysis and Graphically Enhanced Synthesis Package, ASME paper, 77-DTC-5, 1977.
5. Erdman, A. G., and Riley, D. R, Computer-Aided Linkage Design Using the LINCAGES Package, ASME Paper No. 81-DET-121, 1981.
6. Riley, D. R., and Erdman, A. G., Computer Graphics and Computer-Aided Design in Mechanical Engineering at the University of Minnesota, *Computers and Education*, vol. 5, No. 4, 1981, pp. 229-243.
7. Erdman, A. G., and Sandor, G. N, Mechanism Design, Analysis and Synthesis, vol. 1, 2nd edition, Prentice Hall, Englewood Cliffs, New Jersey, 1991.
8. Timmins, S. J., Keefe, M., and Zimmerman, J. R., Educational Software for Mechanism Design, *CoED Journal*, Vol. 9, No. 2, 1989, pp. 47-52.

9. Norton, R. L., Teaching Kinematics with Microcomputer, *CoED Journal*, vol. 6, No.1, 1986, pp. 28-36.
10. Norton, R. L., *Design of Machinery*, McGraw-Hill, Inc., New York, 1992.
11. Cheng, H. H., *The* C^H *Programming Language*, revision 1.1, Department of Mechanical and Aeronautical Engineering, University of California, Davis, November 28, 1993.
12. Cheng, H. H., Scientific Computing in the C^H Programming Language, *Scientific Programming*, vol. 2, No. 3, Fall, 1993, pp. 49-75.
13. Cheng, H. H., Handling of Complex Numbers in the C^H Programming Language, *Scientific Programming*, vol. 2, No. 3, Fall, 1993, pp. 77-106.
14. Cheng, H. H., Programming with Dual Numbers and its Applications in Mechanism Design, *Engineering with Computers*, Vol. 10, No. 4, 1994, pp. 212-229.
15. Cheng, H. H., Numerical Computations in the C^H Programming Language with Applications in Mechanisms and Robotics, *Proc. of the 3rd National Conference on Applied Mechanisms and Robotics*, Cincinnati, OH, Nov. 8-10, 1993, vol. 1, pp. AMR-93-17-1 to pp. AMR-93-19-15.
16. Cheng, H. H., Computations of Dual Numbers in the Extended Finite Dual Plane, *Proc. of ASME Design Automation Conf.*, Albuquerque, NM, Sept. 19-22, 1993, vol. 2, pp. 73-80.
17. Kernighan, B. W. and Ritchie, D. M., *The C Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1st edition, 1978; 2nd edition, 1988.
18. ISO/IEC, *Programming Languages - C*, 9899:1990E, ISO, Geneva, Switzerland.
19. ANSI, *ANSI Standard X3.9-1978, Programming Language FORTRAN*, (revision of ANSI X2.9-1966), American National Standards Institute, Inc., NY, 1978.
20. ISO/IEC, *Information Technology, Programming Languages - FORTRAN*, 1539:1991E, ISO, Geneva, Switzerland.
21. IEEE, *ANSI/IEEE Standard 754-1985, IEEE Standard for Binary Floating-Point Arithmetic*, Institute of Electrical and Electronic Engineers, Inc., Piscataway, NJ, 1985.
22. Yang, A. T., Lecture Notes for EME152 Mechanism Design, Department of Mechanical Engineering, University of California, Davis, Fall, 1989.
23. Cheng, H. H., Real-Time Manipulation of a Hybrid Serial-and-Parallel-Driven Redundant Industrial Manipulator, *Proc. of American Control Conference*, vol. 2, pp. 1801-1805, San Francisco, CA, June 2-4, 1993; also *ASME Trans., J of Dynamic Systems, Measurement, and Control* (in press).
24. Cheng, H. H., Computer-Aided Analysis and Design of Multibody Mechanical Systems, Integration Engineering Laboratory, Department of Mechanical and Aeronautical Engineering, University of California, Davis, Technical Report TR-IEL-94-102, March 25, 1994.