



# TABLE OF CONTENTS

CHAPTER	PAGE
<b>1 OVERVIEW</b> . . . . .	5
1.1 Capabilities of <i>Robotica</i> . . . . .	5
1.2 Requirements . . . . .	6
<b>2 BASIC CONCEPTS</b> . . . . .	7
2.1 Loading <i>Robotica</i> . . . . .	7
2.2 Lists in Mathematica . . . . .	7
2.3 Input File Format . . . . .	8
2.4 <i>Robotica</i> Notation . . . . .	11
2.5 The Help Command . . . . .	11
<b>3 FORWARD KINEMATICS</b> . . . . .	13
3.1 Loading a Robot . . . . .	13
3.2 Generating the Equations . . . . .	14
3.3 Printing Results . . . . .	15
3.4 Summary . . . . .	18
<b>4 ARM GRAPHICS</b> . . . . .	19
4.1 Arm Construction . . . . .	21
4.2 Design Guidelines for Joints . . . . .	22
4.3 An Example . . . . .	23
4.4 Planar Arm Interjoint Shapes . . . . .	24
4.5 Animation and other Options . . . . .	27
<b>5 DYNAMICS</b> . . . . .	28
5.1 Loading a Robot . . . . .	28

5.2	Generating the Equations . . . . .	29
5.3	Printing Results . . . . .	30
5.4	Summary . . . . .	31
6	<b>DYNAMICS RESPONSE COMPUTATION . . . . .</b>	<b>35</b>
6.1	Defining the Torque Vector . . . . .	35
6.2	Calling Response . . . . .	36
6.3	Saving Results . . . . .	37
6.4	Viewing Results . . . . .	38
6.5	Two-Link Example . . . . .	39
7	<b>MANIPULABILITY . . . . .</b>	<b>44</b>
7.1	Generating the Ellipsoids . . . . .	44
7.2	Options . . . . .	45
7.3	Examples . . . . .	47
8	<b>EXTERNAL SIMULATION COMPATIBILITY . . . . .</b>	<b>52</b>
8.1	Loading Simulation Data . . . . .	52
8.2	Simulation Input File Format . . . . .	54
8.3	Plotting Simulation Data . . . . .	54
8.4	Examples . . . . .	55
9	<b>ROBOTICA FRONT END . . . . .</b>	<b>60</b>
9.1	The Structure of RFE . . . . .	60
9.2	Menu Bar . . . . .	60
	9.2.1 File . . . . .	60
	9.2.2 Packages . . . . .	61
9.3	Mathematica Output Window . . . . .	62
9.4	Command Input Line . . . . .	63
9.5	<i>Robotica</i> Function Selection Menu . . . . .	63
10	<b>COMMAND REFERENCE . . . . .</b>	<b>66</b>
	APrint . . . . .	66

ClearLinkShape . . . . .	66
ClearPrisJoint . . . . .	67
ClearRevJoint . . . . .	67
CPrint . . . . .	67
DataFile . . . . .	68
ELDynamics . . . . .	68
EPrint . . . . .	68
FKin . . . . .	69
GetInputTau . . . . .	69
LoadAnim . . . . .	69
LinkShape . . . . .	69
MPrint . . . . .	70
Planar . . . . .	70
PrintInputData . . . . .	71
PrisJoint . . . . .	71
RElp . . . . .	71
Response . . . . .	74
RevJoint . . . . .	74
SDynamics . . . . .	75
SaveAnim . . . . .	75
SaveResponse . . . . .	75
SeqShowRobot . . . . .	76
SetRanges . . . . .	78
ShowAnim . . . . .	78
ShowRobot . . . . .	78
SimDrive . . . . .	80
SimplifyExpression . . . . .	82
SimplifyTrigNotation . . . . .	82
SimPlot . . . . .	82
TPrint . . . . .	83

<b>A KINEMATICS AND DYNAMICS EQUATIONS</b> . . . . .	85
<b>REFERENCES</b> . . . . .	89

# LIST OF TABLES

Table	Page
2.1 <i>Robotica</i> notation. . . . .	12

# LIST OF FIGURES

Figure	Page
2.1 The c.o.m. vectors for a two-link arm. . . . .	11
3.1 Successful read operation. . . . .	14
3.2 Results of FKin[] on input data. . . . .	15
3.3 MPrint[] usage. . . . .	15
3.4 Results of SimplifyTrigNotation[]. . . . .	16
3.5 The Jacobian for the two degree of freedom robot. . . . .	16
3.6 Output from EPrint[]. . . . .	17
4.1 Standard revolute joint. . . . .	19
4.2 Standard prismatic joint. . . . .	20
4.3 Stanford arm. . . . .	20
4.4 New prismatic joint. . . . .	24
4.5 Stanford arm with new joint shapes. . . . .	25
4.6 Arm with standard shapes. . . . .	26
4.7 Arm with new link shape. . . . .	26
5.1 Successful read operation. . . . .	29
5.2 Results of ELDynamics[]. . . . .	30
5.3 The elements of the mass matrix. . . . .	31
5.4 Results of running Simplify TrigNotation[]. . . . .	32
5.5 Examining the Christoffel symbols. . . . .	33
5.6 The Jvc matrices. . . . .	33
5.7 $G$ , the gravity vector. . . . .	34
6.1 Reading the input torque vector. . . . .	36
6.2 Joint parameter q1 tracking a Sine wave. . . . .	39

6.3	Circular motion control definitions. . . . .	41
6.4	Joint variables as a function of time. . . . .	42
6.5	Arm location from circular controller. . . . .	42
6.6	End effector location from circular controller. . . . .	43
7.1	Planar manipulability example. . . . .	48
7.2	Scaling the ellipsoids. . . . .	49
7.3	Three-link example. . . . .	50
7.4	Three-link nonplanar example. . . . .	50
7.5	A change of ViewPoint. . . . .	51
8.1	Loading a SIMNON file. . . . .	55
8.2	Two link planar arm driven with dataset. . . . .	56
8.3	Plot of the end effector location. . . . .	57
8.4	Time, $t$ , versus $q_1$ and $q_2$ . . . . .	58
8.5	Joint parameter $q_1$ versus $q_2$ . . . . .	59
9.1	<i>Robotica</i> Front End. . . . .	61
9.2	Output in the Mathematica output window. . . . .	62
9.3	DataFile option screen. . . . .	64
9.4	SimDrive option screen. . . . .	64
A.1	Denavit-Hartenburg frame assignment [3]. . . . .	85



# CHAPTER 1

## OVERVIEW

### 1.1 Capabilities of *Robotica*

*Robotica*<sup>1</sup> is a collection of useful robotics problem solving functions encapsulated in a Mathematica package. Utilizing Mathematica's computational features allows results to be generated in a purely symbolic form. The benefit of the symbolic representation is that the problem is solved for all possible input parameter values at once, and the user can then substitute and experiment with actual numbers to discover the effects of various parameters.

*Robotica* requires input in the form of a table of Denavit-Hartenberg parameters describing the robot to be analyzed. Once the table has been entered, *Robotica* can generate the forward kinematics for the robot. The  $A$  and  $T$  matrices as well as the velocity Jacobian,  $J$ , are generated. Of course, it is possible to display and save to an external file all of the data generated. If the dynamics equations of the robot are also to be generated, the input must include the dynamics description data as detailed in Chapter 2.

Once the forward kinematics are produced, Euler-Lagrange dynamics equations can be calculated. The inertia matrix, Coriolis and centrifugal terms, Christoffel symbols and gravity vectors are all available to the user once the dynamics routines have run.

Utilizing the forward kinematics results, *Robotica* can calculate the manipulability ellipsoids when supplied with a range of joint variable values. It is possible to generate and save a list of manipulability measures as well as display the ellipsoids with the robot on the screen.

---

<sup>1</sup>*Robotica*<sup>TM</sup> is a trademark of The Board of Trustees of the University of Illinois.

In addition, *Robotica* has the capability of reading external simulation (e.g., SIMNON [1]) output files and displaying the motion of the robot when subjected to the sequence of joint variable changes described in the file. This requires that the robot has been input as a table of Denavit-Hartenburg parameters, and that the forward kinematics routines have been executed.

*Robotica* contains several functions that can be used to draw the robot in a specific configuration, or show the robot moving through a range of joint parameter values. Most of the graphics output can be animated if the Animation.m package is loaded, this includes the graphics produced with SimDrive[], RElp[], ShowRobot[], and SeqShowRobot[]. The animations can be saved and later restored and viewed again. See the functions SaveAnim[], LoadAnim[], ShowAnim[], and SetRanges[] in Appendix A for more details.

To simplify interaction with *Robotica*, an X-Windows based interface has been designed. The interface insulates the user from the inconvenient textual interface Mathematica provides (see Chapter 9 for details.)

## 1.2 Requirements

Running *Robotica* directly through Mathematica requires Mathematica release 2.0 or better, while running *Robotica* through the X-Windows front end requires Mathematica 2.1 or better. Much of *Robotica* deals with text only input and output; however, there are functions which produce graphical output which requires a machine capable of producing the displays.

# CHAPTER 2

## BASIC CONCEPTS

### 2.1 Loading *Robotica*

Once Mathematica has been started (check that the version is 2.0 or greater), the *Robotica* package can be loaded with the following command:

```
In[1] := << robotica.m
```

At this point, all of the functions provided by the package are available to the user (although some may require that other actions have been performed first, e.g., running `FKin[]` before `ELDynamics[]`). Mathematica is case sensitive, thus it is important to type the function names exactly as they appear in the text, e.g., `FKin[]` is entered as capital *F*, capital *K*, lower case *i*, lower case *n*, open `[`, close `]`.

### 2.2 Lists in Mathematica

There are several functions in *Robotica* that take lists as input parameters. There are a few basic rules governing lists that should be understood when specifying them to functions.

First, a list begins with an opening `{`, ends with a closing `}`, and contains elements separated by commas (if there is more than one). For example, `{trace}` is a list containing one element called *trace* and `{trace, print}` is a list containing the two elements *trace*, and *print*.

In addition, Lists can include just about anything as elements: symbols, numbers, even other lists. For example, `{{q1, 0, 3.1415}, 6}`, is a list containing another list,

$\{q1, 0, 3.1415\}$ , and the number  $6$ . This inner list contains three elements,  $q1$ ,  $\theta$ , and  $3.1415$ .

If there is a list that is used frequently, it is sometimes more convenient to assign some variable name to it than to retype it every time it is used. For example:

```
In[5] := simlist = {d1, q2}
```

```
In[6] := SimPlot[t, simlist]
```

Whenever Mathematica sees *simlist*, it will substitute the list  $\{d1, q2\}$ , which can save much unnecessary user input. See [2] for more information about lists, and Mathematica in general.

## 2.3 Input File Format

A convenient way to specify input data to *Robotica* is through a data file. Denavit-Hartenberg parameters and dynamics data for the robot can be given in such a file. A representative input data file is shown below (there are line numbers at the beginning of each line which are for reference only — they should not actually be present in a datafile:

```
[01] A Robotica input data file for  
[02] a two degree of freedom planar robot  
[03] -----  
[04] DOF = 2  
[05] The Denavit-Hartenberg parameters:  
[06] joint1 = revolute  
[07] a1 = a1  
[08] alpha1 = 0  
[09] d1 = 0  
[10] theta1 = q1  
[11] joint2 = revolute  
[12] a2 = a2  
[13] alpha2 = 0  
[14] d2 = 0
```

```

[15] theta2 = q2
[16]
[17] The dynamics information:
[18]
[19] DYNAMICS
[20]
[21] gravity vector = {0,g,0}
[22] mass1 = m1
[23] center of mass = {-(1/2) a1, 0,0}
[24] inertia matrix = {0,0,0,0,0,I1}
[25] mass2 = m2
[26] center of mass = {-(1/2) a2, 0, 0}
[27] inertia matrix = {0,0,0,0,0,I2}

```

In this example, lines [01] through [03] are a general comment, and are ignored. In general, there can be any number of such comment lines prefacing the data.

Line [04] tells *Robotica* that Denavit-Hartenberg parameters are coming. The keyword *DOF* is used by *Robotica* to decide when to start reading the parameters; *DOF* should be followed by an equals sign (=) then the number of links in the robot.

Line [05] is provided as a comment line, it is skipped over when reading the data. If no comment is wanted, leave this line blank.

Lines [06] through [10] contain the Denavit-Hartenberg parameters for the first link. The line `joint1 = revolute` tells *Robotica* that the first link is a revolute type. Two types of links are supported by *Robotica*: revolute and prismatic (sliding). If the link had been prismatic, line [06] would have read `joint1 = prismatic`. Line [07] specifies the *a* parameter for the link, line [08] gives the *alpha* parameter, line [09] supplies the *d* parameter, and line [10] specifies the *theta* variable. The parameters must be given in that order: *a*, *alpha*, *d*, and *theta*. The format for each line is: `label = parameter`. The label is ignored, and the parameter is assigned to its respective Denavit-Hartenberg parameter for that link. There must be exactly one space or tab after the equals sign.

Lines [11] through [15] give the corresponding data for link two. The comments in the above paragraph apply. There should be no space between consecutive data segments for neighboring links.

Lines [16] through [18] are a general comment, and are ignored by *Robotica*. In general, there can be any number of such comment lines as a preface to the dynamics data.

Line [19] contains the required keyword DYNAMICS to indicate to *Robotica* that dynamics input data are to follow.

Line [20] is provided as a comment line, it is skipped over when reading the data. If no comment is wanted, leave this line blank.

Line [21] gives the gravity vector in standard x,y,z form. This vector is understood by *Robotica* to be referenced to the base frame.

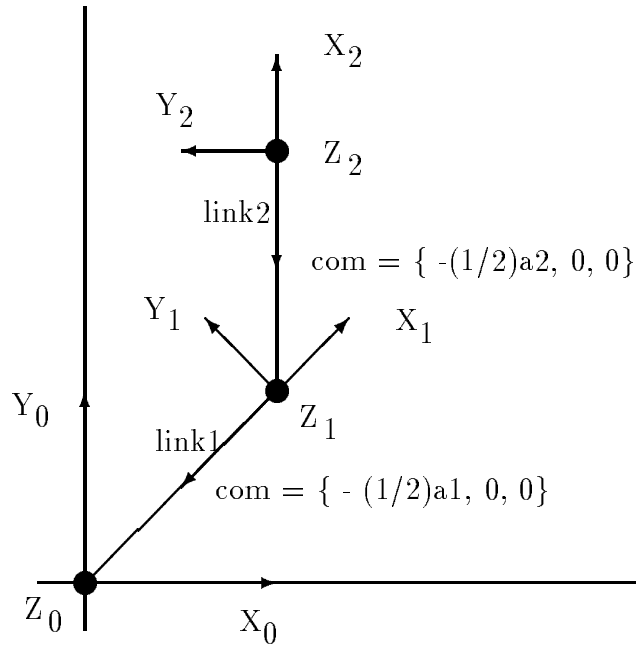
Lines [22] through [24] give the dynamics data for the first link. In line [22],  $mass1 = m1$  assigns the variable  $m1$  to be the mass of the first link. Line [23] supplies the location of the center of mass for link one in terms of the coordinate frame for link one. The format is a list of x, y, and z offsets. In Mathematica, a list is surrounded by braces, thus the input here is  $-(1/2) a1, 0, 0$  which means that the center of mass for this link is back  $(1/2)$  of the length of this link in the x-direction, 0 in the y-direction, and 0 in the z-direction (in terms of the coordinate frame for link one). Figure 2.1 shows the c.o.m. setup for a two-link arm.

Line [24] gives the six unique components of the symmetric inertia tensor, in list format. These are assigned as follows:

$$\{e1, e2, e3, e4, e5, e6\} ==> \begin{vmatrix} e1 & e2 & e3 \\ e2 & e4 & e5 \\ e3 & e5 & e6 \end{vmatrix}$$

The inertia tensor for link  $i$  is a  $3 \times 3$  matrix computed in the coordinate frame attached to link  $i$ . See Equation [6.2.19] in [3] for complete information. The format for each line is:  $label = parameter$ . The label is ignored, and the parameter is assigned to its respective Denavit-Hartenberg parameter for that link.

Lines [25] through [27] give the dynamics information for the next link. The comments of the above paragraphs apply to these lines.



**Figure 2.1:** The c.o.m. vectors for a two-link arm.

Of course, if the generation of forward kinematics equations are all that *Robotica* will be used for, the dynamics portion of the file need not exist. Note also that for each degree of freedom (specified by  $\text{DOF} = n$ ) there must be a set of Denavit-Hartenberg parameters, and if dynamics data are given, there must also be a set of data for each degree of freedom.

## 2.4 *Robotica* Notation

The notation that *Robotica* uses to store the various quantities and parameters is summarized in Table 2.1. In general, the names *Robotica* uses are as close as possible to standard mathematical notation. To access the variable in *Robotica*, one merely types the correct *Robotica* notation at the prompt.

## 2.5 The Help Command

Mathematica provides a built-in help command, `?`. It may be used to request information about any Mathematica command or function as well as all *Robotica* functions

available to the user. For example, to obtain information about the RElp[] function, type: ? RElp[], and Mathematica will respond with a brief help message about the function.

**Table 2.1:** *Robotica* notation.

Notation	Corresponding <i>Robotica</i> Notation	Notation	Corresponding <i>Robotica</i> Notation
$a_i$	a[i]	$g_i$	gravity[i]
$d_i$	d[i]	$m_i$	mass[i]
$\alpha_i$	alpha[i]	$com_i$	com[i]
$\theta_i$	theta[i]	$I_i$	inertia[i]
$A_i$	A[i]	$Jc_i$	Jc[i]
$T_j^k$	T[j,k]	$Jvc_i$	Jvc[i]
$J$	J	$Jwc_i$	Jwc[i]
$J_i$	Jvw[i]	$c_{ijk}$	c[[i,j,k]]
$z_i$	z[i]	$Cos(q_1)$	C1
$o_i$	o[i]	$Sin(q_1)$	S1
$M$	M	$Cos(q_1 + q_2)$	C12
$C$	CM	$Sin(q_1 + q_2)$	S12
$g$	G	$Cos(q_1 - q_2)$	C1-2
		$Sin(q_1 - q_2)$	S1-2
		$Cos(q_1 + q_2 + q_3)$	C123
		$Sin(q_1 + q_2 + q_3)$	S123
		$Cos(q_1 + q_2 - q_3)$	C12-3
		$Sin(q_1 + q_2 - q_3)$	S12-3



# CHAPTER 3

## FORWARD KINEMATICS

One of the primary features of *Robotica* is its ability to generate the forward kinematics equations for any robot described by Denavit-Hartenberg parameters. The mathematics behind this process is described in Appendix B. Essentially, *Robotica* will compute the A matrices, T matrices, and Jacobian (see [3]) for the arm.

### 3.1 Loading a Robot

The first step in the process of generating the equations is to load the data file. The *Robotica* command “DataFile[]” is used for this purpose; it can be used in two ways: DataFile[“filename”] or DataFile[].

The first method specifies the filename from which to load the data; it must be a quoted string. For example, DataFile[“twodof”] loads the file “twodof” if it exists in the file system. Entering DataFile[] without the filename parameter will cause *Robotica* to prompt for the filename. Simply type the filename without quotes in response to the query and *Robotica* will load the file. The following lines show the two possibilities:

```
In[2] := DataFile[“twodof”]
```

or

```
In[2] := DataFile[]
```

```
Enter Data File Name : twodof
```

If the data file is in the correct format, and DataFile[] successfully reads the parameters, a table will be displayed showing what *Robotica* has stored internally from the file. If the file does not exist, or is incorrectly formatted, a message will be displayed to that effect,

and *Robotica* will not store anything. Figure 3.1 shows what a successful read operation looks like.

```
In[2]:= Datafile[]
Enter data file name: newtwo
State Reset...

Kinematics Input Data
-----
Joint  Type      a   alpha d  theta
1      revolute  a1  0     0  q1
2      revolute  a2  0     0  q2

Dynamics Input Data
-----
Gravity vector: [0,g,0]

Link  mass  com vector
1     m1   [-a1/2, 0, 0]
2     m2   [-a2/2, 0, 0]

Inertia[1]=  $\begin{vmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & I1 \end{vmatrix}$ 

Inertia[2]=  $\begin{vmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & I2 \end{vmatrix}$ 
```

**Figure 3.1:** Successful read operation.

## 3.2 Generating the Equations

Once the data file has been loaded, it is very simple to generate the forward kinematics equations; the command `FKin[]` is used for this purpose. `FKin[]` takes no parameters, and displays a report of the various quantities computed. Figure 3.2 shows a sample run of `FKin[]` on the data file of Figure 3.1.

```

In[4]:=FKin[]

A Matrices Formed:

A[1]
A[2]

T Matrices Formed:

T[0,0]
T[0,1]
T[0,2]
T[1,2]

Jacobian Formed
Jacobian (6x2)

```

**Figure 3.2:** Results of FKin[] on input data.

### 3.3 Printing Results

After the forward kinematics equations have been generated, showing the results is accomplished through a set of display functions provided within *Robotica*. MPrint[] is such a function; it takes up to three parameters: the matrix to be displayed (required), a text label to print alongside the matrix (required), and a filename in quotes in which to save the matrix (optional). Figure 3.3 shows how MPrint[] is used to display the matrix A[2].

```

In[5]:= MPrint[A[2], "A2="]

Out[5]=
A2=

$$\begin{vmatrix} \cos[q_2] & -\sin[q_2] & 0 & a_2 \cos[q_2] \\ \sin[q_2] & \cos[q_2] & 0 & a_2 \sin[q_2] \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$


```

**Figure 3.3:** MPrint[] usage.

It is sometimes convenient to change the trigonometric notation to be somewhat more succinct. The function `SimplifyTrigNotation[]` takes no parameters, and modifies the display of `Sin[]` and `Cos[]` to an abbreviated form. Figure 3.4 shows the results of running `SimplifyTrigNotation[]`.

```
In[5]:= SimplifyTrigNotation[]
```

```
In[6]:= MPrint[A[2], "A2= "]
```

```
Out[7]= 
$$\text{A2} = \begin{vmatrix} C2 & -S2 & 0 & a2 & C2 \\ S2 & C2 & 0 & a2 & S2 \\ 0 & 0 & 1 & 0 & \\ 0 & 0 & 0 & 1 & \end{vmatrix}$$

```

**Figure 3.4:** Results of `SimplifyTrigNotation[]`.

Similarly, the Jacobian can be displayed with a simple call to `MPrint[]`; Figure 3.5 shows the results.

```
In[7]:= MPrint[J, "Jacobian="]
```

```
Out[7]= 
$$\text{Jacobian} = \begin{vmatrix} -(a1 S1) - a2 S12 & -(a2 S12) \\ a1 C1 + a2 C12 & a2 C12 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 1 & 1 \end{vmatrix}$$

```

**Figure 3.5:** The Jacobian for the two degree of freedom robot.

Another useful display function is `EPrint[]`; it prints the elements of the matrix one per line. `EPrint[]` takes either two parameters or three parameters: the matrix to be displayed (required), a text label to print alongside the elements (required), and a filename in quotes in which to save the matrix elements (optional). Figure 3.6 shows how `EPrint[]` is used to display the matrix `T[0,1]`.

```
In[8]:= MPrint[T[0,1], "T01 ="]
```

```
Out[8]= 
$$\begin{array}{l} \text{T01} = \left[ \begin{array}{cccc} \text{C1} & -\text{S1} & 0 & \text{a1 C1} \\ \text{S1} & \text{C1} & 0 & \text{a1 S1} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{array} \right] \end{array}$$

```

```
In[9]:= EPrint[T[0,1], "T01"]
```

```
T01[1,1] = C12
```

```
T01[2,1] = S12
```

```
T01[3,1] = 0
```

```
T01[4,1] = 0
```

```
T01[1,2] = -S1
```

```
T01[2,2] = C1
```

```
T01[3,2] = 0
```

```
T01[4,2] = 0
```

```
T01[1,3] = 0
```

```
T01[2,3] = 0
```

```
T01[3,3] = 1
```

```
T01[4,3] = 0
```

```
T01[1,4] = a1 C1
```

```
T01[2,4] = a1 S1
```

```
T01[3,4] = 0
```

```
T01[4,4] = 1
```

**Figure 3.6:** Output from EPrint[].

In addition to EPrint[] and MPrint[], there are several other functions useful for displaying information about the forward kinematics. TPrint[] prints all of the T matrices to a file if specified (e.g., TPrint["tmatrices"]) or to the screen if no parameter is given. APrint[] does the same thing for A matrices. PrintInputData[] takes no parameters and displays all of the information which *Robotica* knows about the current data set, previously read in with DataFile[].

### 3.4 Summary

In summary, there are only two steps required to generate the forward kinematics equations:

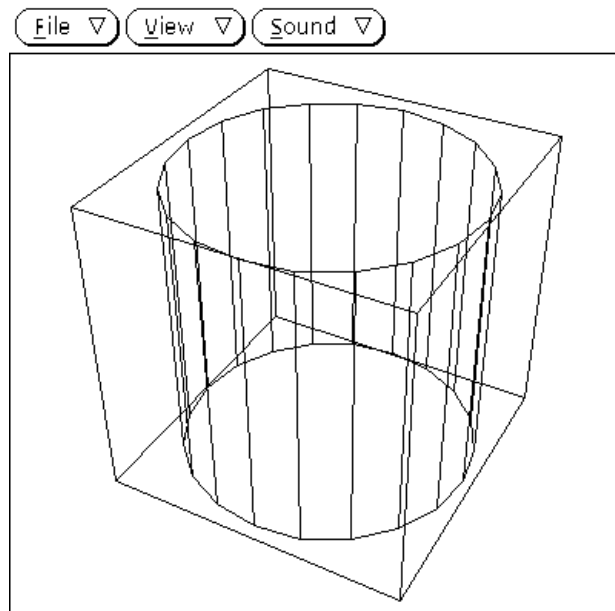
- 1) Run `DataFile[]` to load the set of Denavit-Hartenberg parameters.
- 2) Run `FKin[]` to make *Robotica* do the actual computation.

Once these steps are performed, any of the display functions can be invoked to show the results. Consult Table 2.1 for the notation *Robotica* uses to represent the various quantities.

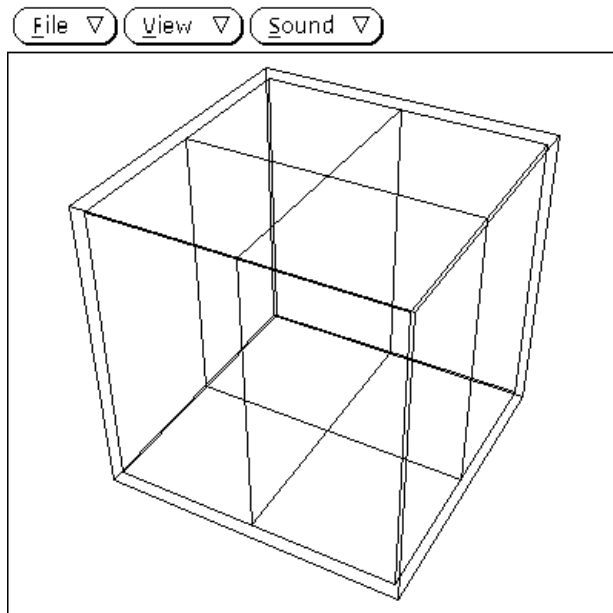
# CHAPTER 4

## ARM GRAPHICS

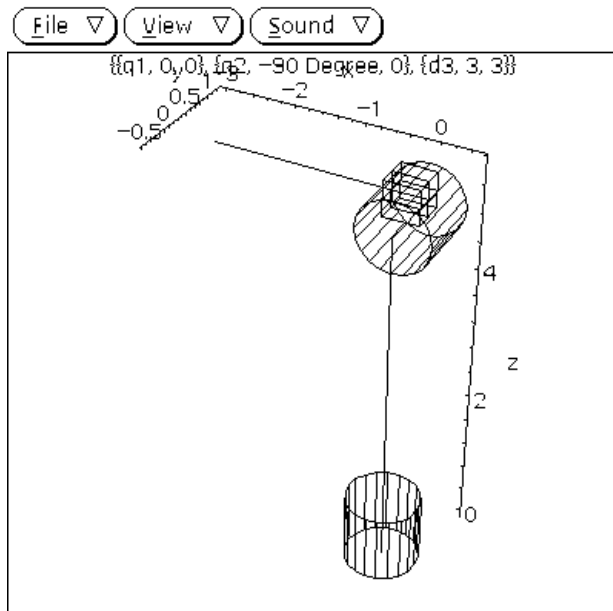
The graphics shapes used to draw robot arms are really quite simple, but still very flexible. Revolute joints and prismatic joints have separate models to represent them; shapes for revolute and prismatic joints can be loaded with the `RevJoint[]` and `PrisJoint[]` commands. If no filename is specified in these commands, then a standard shape for the joint is loaded. For revolute joints this shape is a cylinder (Figure 4.1), for prismatic joints, the shape is a rectangular column (Figure 4.2). The Stanford arm is shown in Figure 4.3 at the  $q_1=0$ ,  $q_2=-90$ Degree,  $d_3=3$  configuration with the standard joint shapes.



**Figure 4.1:** Standard revolute joint.



**Figure 4.2:** Standard prismatic joint.



**Figure 4.3:** Stanford arm.



## 4.1 Arm Construction

Any three-dimensional shape can be loaded from a file and used as a joint shape. The first thing to be aware of is that each shape must specify x-, y-, and z-coordinates so that the shape can be used for three-dimensional plots as well as planar plots. Of course, if the robot is planar in x-y, for example, then the z-coordinate could always be zero, but it still needs to be there. For example,

```
Line[{{0,0}, {2,2}}]
```

could not be used as part of the joint, but the following could:

```
Line[{{0,0,0}, {2,2,0}}]
```

For non-planar arms, joints are connected together with a straight line representing the link between joints. In the case of planar arms, an interjoint shape can be specified with the `LinkShape[]` command; see section 4.3 for more details.

For a revolute joint, the following steps are executed before the joint shape is placed in the output graphic:

- Step1:** The joint shape is rotated by the value of the joint variable. For example, if the second joint in the Stanford arm is at `30Degree`, then the joint shape is rotated about its z-axis `30Degree`.
- Step2:** The joint shape is rotated to the orientation of its frame relative to the base frame. Essentially, this aligns the shape with its own coordinate axes.
- Step3:** The joint is translated out to the proper location determined by the T matrices.

**Step4:** If the robot is planar, then the appropriate coordinate is dropped, and two-dimensional plots result when drawing the robot. For example, an x-y planar robot would have its z-coordinate dropped.

The steps for a prismatic joint are very similar:

**Step1:** The joint shape is rotated to the orientation of its frame relative to the base frame. Essentially, this aligns the shape with its own coordinate axes.

**Step2:** The joint is translated out to the proper location determined by the T matrices.

**Step3:** If the robot is planar, then the appropriate coordinate is dropped, and two-dimensional plots result when drawing the robot. For example, an x-y planar robot would have its z-coordinate dropped.

## 4.2 Design Guidelines for Joints

To design a joint graphic which will work with *Robotica*, the following points should be observed:

1) The joint should be laid out in the coordinate system of the base frame. This is why all of the rotations are done to align the shape with its local coordinate system.

2) A revolute joint probably should have its origin centered around (0,0,0) in the base frame so that the rotations applied will produce the expected results.

3) Any joint shape, even if planar, should include a value for all three coordinates (x,y,z). Thus the components of an x-y planar prismatic joint could look like

Line[{ {0,1,0}, {1,1,0}, ...}]

and not like

```
Line[{ {0,1}, {1,1}, ...}]
```

The packages `Shapes.m` and `Polyhedra.m` can provide building blocks for custom joint shapes. These two standard Mathematica packages contain functions for generating and manipulating many types of shapes such as cones, cylinders, and spheres. In addition, the packages include functions that can stretch, rotate, and translate shapes, as well as convert a solid to a wireframe. To create a custom joint with `Shapes.m` and `Polyhedra.m`, the functions would be used to generate a series of shapes which would be assigned to some variable in Mathematica. When the design is complete, the variable should be written to a file and loaded with `PrisJoint[]` or `RevJoint[]`.

### 4.3 An Example

Creating a new joint shape is relatively easy; solids from the package `Shapes.m` are the most useful is defining new joints. Assuming that the `Shapes.m` package has been loaded, the commands given below will create a new revolute shape that looks like a cylinder. This shape differs from the default revolute shape in that it is rendered as a solid with hidden surface elimination, which makes it more realistic looking. A solid generated with `Shapes.m` can be converted to wireframe with the `WireFrame[]` command; see the `Shapes.m` file for more information.

```
rev = Cylinder[];
rev = AffineShape[rev, .5, .5, .5]; (* make it half size *)
rev >> newrev (* save the shape in a file *)
```

Other shapes can be incorporated into the joint design as well. For example, adding a pyramid that points in the direction of motion for a prismatic joint is relatively straightforward. The following commands create a new prismatic joint with a square base and a pyramid whose apex indicates direction of motion:

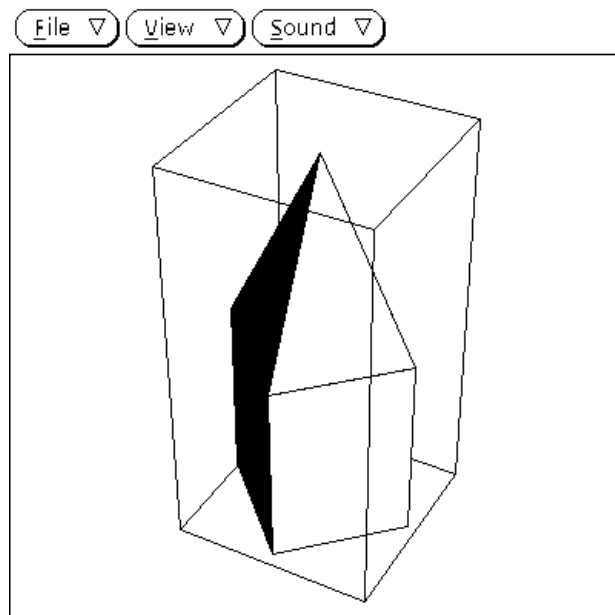
```
base = Cylinder[1,1,4] (* 4 sided cylinder is a cube *)
point = Cone[1,1,4] (*4 sided cone is a pyramid *)
```

```

base = AffineShape[base, .5, .5, .5]
point = AffineShape[point, .5, .5, .5]
point = TranslateShape[point, 0,0,1] (* put the pyramid on top of the cube *)
pris = base, point (* create a list with both shapes *)
pris >> newpris (* save the shape in a file *)

```

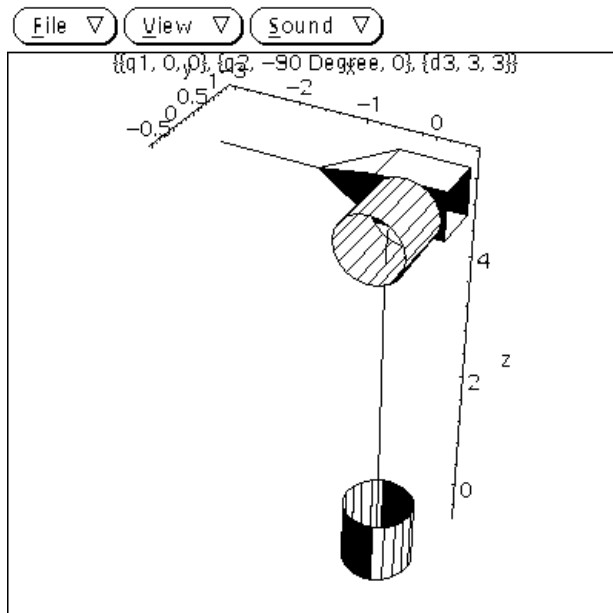
Figure 4.4 shows the new prismatic joint shape and Figure 4.5 shows how the new joints appear in a graphic. Of course, the commands `PrisJoint[]` and `RevJoint[]` must be used to load the new joint shapes before rendering.



**Figure 4.4:** New prismatic joint.

## 4.4 Planar Arm Interjoint Shapes

Planar arms can be drawn with link shapes connecting the joint shapes. The function `LinkShape[]` loads a shape definition from a file (or if no filename is given, uses a simple rectangular shape). The function `ClearLinkShape[]` causes *Robotica* to clear the link definition and use a line to represent links.

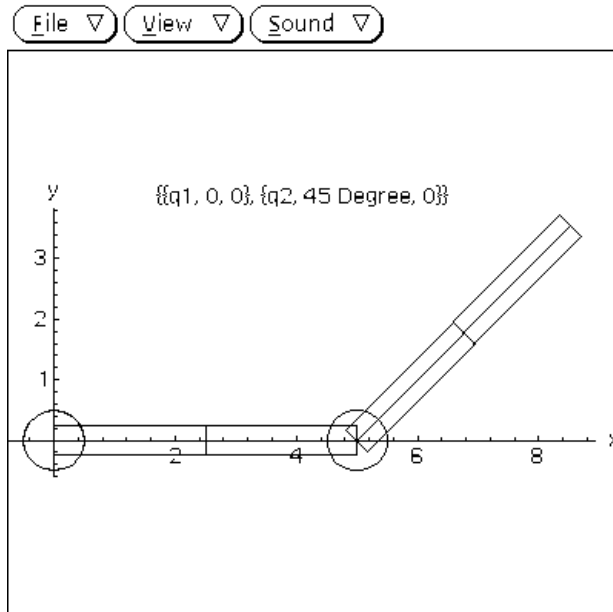


**Figure 4.5:** Stanford arm with new joint shapes.

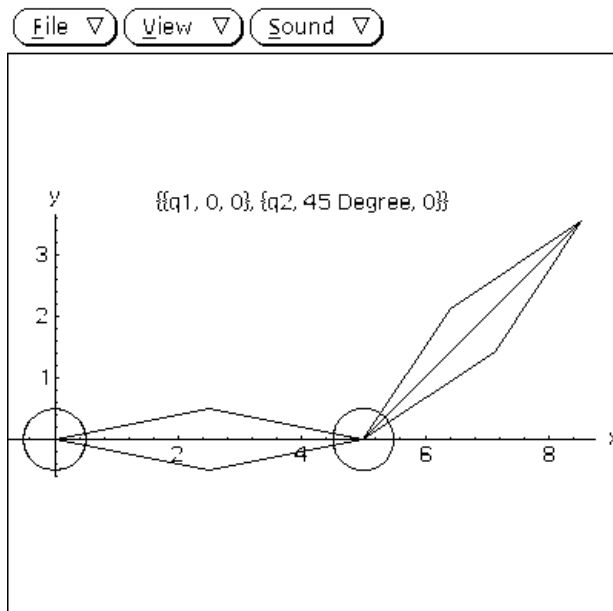
Figure 4.6 shows a two-link planar arm with the standard revolute joint shape and link shape. Figure 4.7 shows the same arm with a diamond link shape; this shape was stored in a file as:

```
Line[{{0,0,0}, {.5,.5,0}, {1,0,0}, {.5,-.5,0}, {0,0,0}}]
```

Much of the confusion in creating a link shape stems from the fact that the shape is scaled by either  $a_i$  in  $x$  or  $d_i$  in  $z$  before being placed in the graphic (where  $i$  is the joint number). If the  $a_i$  parameter is non-zero, then the link shape is scaled by  $a_i$  in  $x$ . This means that the link shape should begin at  $x=0$  and end at  $x=1$  so that after scaling, the link shape starts at  $x=0$  and ends at  $x=a_i$ . This was the case for the two link planar arm in Figure 4.6 and Figure 4.7. The situation is similar for a non-zero  $d_i$  except that the arm is scaled in  $z$ . These scalings must be done in order to insure that the links and joints appear connected on the graphic.



**Figure 4.6:** Arm with standard shapes.



**Figure 4.7:** Arm with new link shape.

## 4.5 Animation and other Options

To produce animations with *Robotica*, the package `Animation.m` must be loaded; this package is provided by Wolfram Research as part of the standard Mathematica system. Specifying the *animate* option to the various procedures will suppress the generation of the composite image and use each frame to generate a frame by frame animation. Writing all of the frames to the animation program is not a quick process; eventually, however, the animation output will stop flashing and the frames will flow smoothly together. Each frame of the animation must have consistent plot ranges so that the animation flows smoothly. The x, y, and z ranges are set with the `SetRanges[]` command which forces each part of the animation to include x, y, and z values as specified by the ranges. In addition, boxes can be drawn around each frame if the *frame* option is specified. Usually, however, adding the frame clutters the image too much to be useful.

## CHAPTER 5

### DYNAMICS

Another feature included in *Robotica* is the ability to compute the Euler-Lagrange dynamics description of a robot. *Robotica* computes the mass matrix, Christoffel symbols, C matrix, and the gravity vector when an input file with dynamics data is loaded and the appropriate commands are executed.

#### 5.1 Loading a Robot

The first step in the process of generating the equations is loading the data file representing the robot. The *Robotica* command “DataFile[]” is used for this purpose; it can be used in two ways: DataFile[“filename”] or DataFile[].

The first method specifies the filename from which to load the data; it must be a quoted string. For example, DataFile[“newtwo”] loads the file “newtwo” if it exists in the file system. Entering DataFile[] without the filename parameter will cause *Robotica* to prompt for the filename. Simply type the filename without quotes in response to the query and *Robotica* will load the file.

```
In[2] := DataFile[“newtwo”]
```

or

```
In[2] := DataFile[]
```

```
Enter Data File Name : newtwo
```

If the data file is in the correct format and DataFile[] successfully reads the parameters, a table will be displayed showing what *Robotica* has stored internally from the file. If the file does not exist or is incorrectly formatted, a message will be displayed to that effect, and *Robotica* will not store anything.



Figure 5.1 shows what a successful read operation looks like.

```
In[2]:= Datafile[]
Enter data file name: newtwo
State Reset...

Kinematics Input Data
-----
Joint  Type      a   alpha  d  theta
1      revolute  a1  0      0  q1
2      revolute  a2  0      0  q2

Dynamics Input Data
-----

Gravity vector: [0,g,0]

Link  mass  com vector
1      m1   [-a1/2, 0, 0]
2      m2   [-a2/2, 0, 0]

Inertia[1]=  $\begin{vmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & I1 \end{vmatrix}$ 

Inertia[2]=  $\begin{vmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & I2 \end{vmatrix}$ 
```

**Figure 5.1:** Successful read operation.

## 5.2 Generating the Equations

Once the data file has been loaded, a two-step process is required to generate Euler-Lagrange dynamics; first, `FKin[]` must be successfully run to compute the forward kinematics. See Chapter 3 for complete information on this process. The second step is to run `ELDynamics[]`, the command which will produce the dynamics information. `ELDy-`

namics[] does not take any parameters. Figure 5.2 shows the generation of dynamics equations with the input data from Figure 5.1.

```
In[5]:= ELDynamics[]  
  
Mass Matrix MU(2 x 2) Formed. No Trigonometric Simplification  
  
Christoffel Symbols Formed  
  
C Matrix CM(2 x 2) Formed  
  
Gravity Vector G(2 x 1) Formed
```

**Figure 5.2:** Results of ELDynamics[].

Due to its immense complexity, the mass matrix produced (MU) will not be simplified unless it is explicitly requested with SimplifyExpression[MU] or SDynamics[]. For arms with even a small number of links this process could take a very long time due to the symbolic nature of the quantities involved. If there are only a few elements of interest in the mass matrix, they can be simplified independently. For example, SimplifyExpression[MU[[1,1]]], reduces only the first element. The Christoffel symbols and C matrix will, in general, also need to be simplified. Running SDynamics[] will simplify, using SimplifyExpression[], all of the dynamics quantities generated and will also display which elements are currently going through the simplification process.

### 5.3 Printing Results

Once the dynamics equations have been generated, showing the results is accomplished through a set of display functions provided within *Robotica*. EPrint[] is one such function; it takes up to three parameters: the matrix to be displayed (required), a text label to print alongside the matrix (required), and a filename in quotes in which to save the matrix (optional). Figure 5.3 shows how EPrint[] is used to display the elements of the simplified mass matrix.

```
In[6]:= EPrint[M, "Mass matrix"]
```

$$\text{Mass matrix}[1,1] = I1 + I2 + \frac{a1^2 m1}{4} + a1 m2 + \frac{a2^2 m2}{4} + a1 a2 m2 \text{Cos}[q2]$$

$$\text{Mass matrix}[2,1] = I2 + \frac{a2^2 m2}{4} + \frac{a1 a2 m2 \text{Cos}[q2]}{2}$$

$$\text{Mass matrix}[1,2] = I2 + \frac{a2^2 m2}{4} + \frac{a1 a2 m2 \text{Cos}[q2]}{2}$$

$$\text{Mass matrix}[2,2] = I2 + \frac{a2^2 m2}{4}$$

**Figure 5.3:** The elements of the mass matrix.

It is sometimes convenient to change the trigonometric notation to be somewhat more succinct. The function `SimplifyTrigNotation[]` takes no parameters and modifies the display of `Sin[]` and `Cos[]` to an abbreviated form. Figure 5.4 shows the results of running `SimplifyTrigNotation[]`.

The Christoffel symbols are stored in a three-dimensional array called `c`. They can be examined by entering `c` with a subscript or by using the `CPrint[]` function. Figure 5.5 shows how to examine individual elements. Other quantities of interest that can be displayed include the Jacobians `Jvc` and `Jwc`. For example, Figure 5.6 shows the `Jvc` matrices for the two-link robot. In addition,  $G$ , the gravity vector, can be displayed with a simple call to `MPrint[]`. Figure 5.7 shows  $G$  for the two-link robot.

## 5.4 Summary

In summary, there are only three steps required to generate the Euler-Lagrange dynamics information:

- 1) Run `DataFile[]` to load the set of Denavit-Hartenburg parameters.
- 2) Run `FKin[]` to make *Robotica* do the forward kinematics.
- 3) Run `ELDynamics[]` to generate the dynamics information.

Once these steps are performed, any of the display functions can be invoked to show the results. `SimplifyExpression[]` can also be used at this point to simplify any required information.

```
In[7]:= SimplifyTrigNotation[]
```

```
In[8]:= EPrint[CM, "C Matrix"]
```

$$\text{C Matrix}[1,1] = \frac{-(a1 \ a2 \ m2 \ Dq2 \ S2)}{2}$$

$$\text{C Matrix}[2,1] = \frac{a1 \ a2 \ m2 \ Dq1 \ S2}{2}$$

$$\text{C Matrix}[1,2] = \frac{-(a1 \ a2 \ m2 \ Dq1 \ S2)}{2} - \frac{a1 \ a2 \ m2 \ Dq2 \ S2}{2}$$

$$\text{C Matrix}[2,2] = 0$$

**Figure 5.4:** Results of running `Simplify TrigNotation[]`.

In[10]:= c[[1,1,1]]

Out[10] = 0

In[11]:= c[[1,2,1]]

Out[11] =  $\frac{-(a1 a2 m2 S2)}{2}$

In[12]:= c[[1,1,2]]

Out[12]=  $\frac{a1 a2 m2 S2}{2}$

**Figure 5.5:** Examining the Christoffel symbols.

In[14]:= EPrint[Jvc[1], "Jvc1"]

Jvc1[1,1] =  $\frac{-(a1 S1)}{2}$

Jvc1[2,1] =  $\frac{a1 C1}{2}$

Jvc1[3,1] = 0

Jvc1[1,2] = 0

Jvc1[2,2] = 0

Jvc1[3,2] = 0

In[16]:= MPrint[Jvc[2], "Jvc2 = "]

Out[16]= 
$$\begin{array}{l} \text{Jvc2=} \\ \left| \begin{array}{cc} -(a1 S1) - \frac{a2 S12}{2} & -\frac{(a2 S12)}{2} \\ a1 C1 + \frac{a2 C12}{2} & \frac{a2 C12}{2} \\ 0 & 0 \end{array} \right| \end{array}$$

**Figure 5.6:** The Jvc matrices.

In[25]:= MPrint[G, "Gravity Vector g = "]

$$\text{Out[25] = Gravity Vector g = } \left| \begin{array}{l} \frac{a_1 g m_1 C_1}{2} + g m_2 \left( a_1 C_1 + \frac{a_2 C_{12}}{2} \right) \\ \frac{a_2 g m_2 C_{12}}{2} \end{array} \right|$$

**Figure 5.7:**  $G$ , the gravity vector.

## CHAPTER 6

### DYNAMICS RESPONSE COMPUTATION

Complementing the ability to compute the dynamics equations governing a robotic arm is the ability to solve the equations numerically for the time response of the joint variables, given an input torque. The torques are supplied to *Robotica* through an input file, which defines the elements of the torque vector, as well as other variables which may be used in the definition of the torque vector. The time response, once computed, can then be saved in SIMNON compatible format for later analysis with Simplot[], SimDrive[], or SIMNON itself. See SimDrive[] in the command reference for a description of the file format. Before any of these calculations can proceed, however, both ELDynamics[] and SDynamics[] must have been run.

#### 6.1 Defining the Torque Vector

The input torque vector represents  $\tau$  in  $M(q)\ddot{q}+C(q,\dot{q})\dot{q}+g(q) = \tau$ , where  $\tau = \begin{bmatrix} \tau_1 \\ \cdot \\ \cdot \\ \tau_n \end{bmatrix}$ ,

a vector containing a number of elements equal to the number of degrees of freedom in the robot. GetInputTau[] is the function responsible for reading a definition of the input torque; use a filename parameter to obtain an input vector from a file, or use no parameter to set the input torque vector to all zeros. For example,

```
GetInputTau["tau1"]
```

reads the torque definition from the file "tau1" and

GetInputTau[]

sets the torque vector to all zeros.

The input file format is rather flexible, as it can contain variable name and function definitions that can be used in later assignments within the file. Mathematica interprets all of the expressions found in the file, and keeps track of them so that they may be modified later. Note that definitions loaded from a file can interfere with user-defined variables; essentially, the result of reading the input file is the same as if each line had been entered directly to Mathematica. Consider the input file in Figure 6.1.

```
q1r = 3Sin[t]
q1dr = 3Cos[t]
q1ddr = -3Sin[t]

tau[1] = k1(q1r - q1[t]) + k2(q1dr - q1'[t]) + q1ddr
```

**Figure 6.1:** Reading the input torque vector.

The variables  $q1r$ ,  $q1dr$ , and  $q1ddr$  are defined in the first three lines, then later used in the definition of the first element of the tau vector,  $\text{tau}[1]$ . The only requirements on the format of the input file are that each element of the tau vector is assigned with the following semantics:  $\text{tau}[n] = \text{EXPRESSION}$ , and that there are  $n$  entries in tau, one for each degree of freedom. The example in Figure 5.1 defines the necessary element in the tau vector for a single link arm, it applies feedback to try to force the arm to track a Sine wave of amplitude 3. Note that the variables  $k1$  and  $k2$  are left undefined; these must be assigned values before the response function can be called. They are left as symbols in the input file to allow greater flexibility of parameter adjustment.

## 6.2 Calling Response

Assuming that the dynamics have been generated with `ELDynamics[]`, the response can be calculated with a call to `Response[]`. First, however, all symbolic entries must be assigned numerical values; this includes setting the inertias, link masses, etc. The following examples use the tau vector for the one degree of freedom revolute arm. There



are two parameters to response, each a list. The first list can contain two or three numbers: the first is the start time of the response, the second is the end time of the response, and the last is the maximum number of steps `Response[]` should take when calculating solutions. The start and stop times are required; the number of steps should be set to a large value if the total response time is long. The default number of steps is 500. One way to handle choosing the number of steps to specify is to assign it some arbitrary large value, e.g., 1000000, although the number of steps can be determined by running `Response[]`, then adjusting the steps to a greater value if the following message appears:

```
NDSolve::mxst: Maximum number of steps reached at the point xxx.
```

The second list contains the initial conditions used when solving the differential equations. In general, there should be  $2n$  initial conditions, where  $n$  is the number of degrees of freedom (since the equations involve up to second derivatives for each joint parameter). The following is a valid call to `Response[]`:

```
Response[{0,5}, {q1[0]==0, q1'[0]==0}]
```

This calculates the time response of the one link example over the time range 0 to 5 (using the default number of steps), with the input torque as defined in Figure 6.1. Notice the format of the initial condition assignment. It is important that double equal signs be used when setting these parameters; if a single equal sign is used, Mathematica assumes that an assignment is being made directly to the variables, rather than a conditional test.

### 6.3 Saving Results

After `Response[]` has been run and solutions have been generated, they may be saved to a file in SIMNON compatible format for analysis. Putting the results in a file is accomplished with the `SaveResponse[]` function. Either two or three parameters are given to `SaveResponse[]`; the first parameter is a string containing a file name in which to save the data; this parameter is required. The second parameter is a list containing the time

range from the solution to save; it is also required. The time range given must lie within the range initially solved for with `Response[]`, of course. The third parameter is optional, and specifies the size of time step to take when saving the data; it defaults to 0.1. A value for the joint variables will be stored starting at the start time, incremented by the step, and ending at the stop time. The following is a valid call to `SaveResponse[]`:

```
SaveResponse["sinres", {0,5}, 0.05]
```

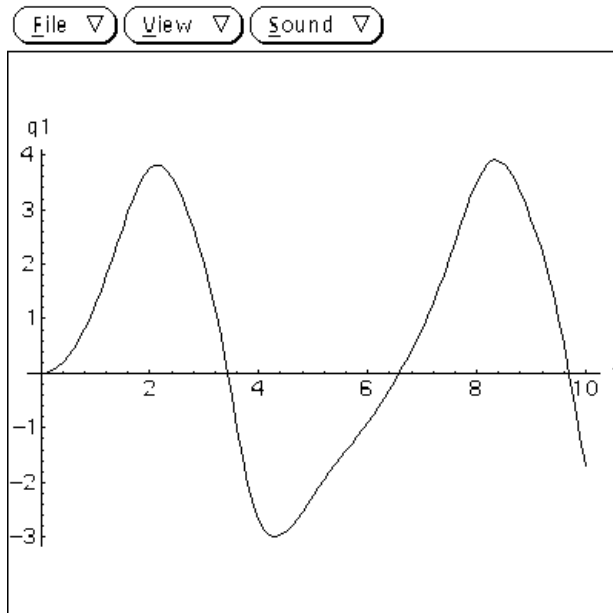
## 6.4 Viewing Results

Once the solutions have been saved, the functions `SimDrive[]` and `SimPlot[]` can be used to visualize the results. Assuming that we have generated the response and saved it to file "sinres" as described above, `SimDrive[]` can be called to load the data set into memory. `SimPlot[]` can then be used to display the joint trajectory. For example,

```
SimDrive["sinres"]  
SimPlot[t, {q1}]
```

results in the graph of `q1` shown in Figure 6.2. All of the options available from `SimDrive[]` can be specified, of course. For example, `SimDrive["$", {arm, animate}]` would show an animation of the arm responding to the input torque. Some of the more useful options include:

- trace:** Show a plot of the end effector location.
  
- arm:** Draw the arm graphics as the arm moves in response to the input data.
  
- animate:** Take all of the generated frames, and show them sequentially in movie fashion. The animation can be shown again with `ShowAnim[]`, or saved with `SaveAnim[]`.



**Figure 6.2:** Joint parameter  $q_1$  tracking a Sine wave.

## 6.5 Two-Link Example

Figure 6.3 shows the tau file definition of a controller that tries to make a two-link arm with revolute joints move in a circle centered about (2,2). The DH description is given in Figure 3.1. The input tau file defines the circle in Cartesian space in the first two lines, then proceeds to define quantities which are used as references in the control equations. The parameters  $om$ ,  $kp_1$ ,  $kp_2$ ,  $kd_1$ , and  $kd_2$  are set at the end of the file, although they could be assigned values at any time prior to the running of `Response[]`. In addition, the values  $a_1$ , and  $a_2$  were set to 3,  $m_1$  set to 10,  $m_2$  set to 5,  $g$  set to 9.8,  $I_1$  set to 0.833, and  $I_2$  set to 0.417.

Note that the definitions of `tau[1]` and `tau[2]` include quantities that are generated by the dynamics commands, namely, elements of the simplified mass matrix ( $M$ ), elements of the simplified  $C$  matrix ( $CM$ ), and elements of the gravity vector ( $G$ ).

The commands given to calculate the response over the time period 0,10 and save the response in a file called “circ1” are shown below.

```
In[25]:= Response[{0,10,100000}, {q1[0]==0, q2[0]==0, q1'[0]==0, q2'[0]==0}]
```

Solving equations...

Out[25]=

```
{{q1 -> InterpolatingFunction[{0,10}, <>], q2 -> InterpolatingFunction[{0,10}, <>]}}
```

```
In[26]:= SaveResponse["circ1", {0,10}]
```

Next, `SimDrive[]` was used to load the data; the following command was used:

```
In[30]:= SimDrive["circ1"]
```

With the data in memory, the following plots were created: the values of the joint variables, the location of the arm, and the trace of the end effector location. The commands used were:

`SimPlot[t, {q1,q2}]` to generate Figure 6.4,

`SimDrive["$", {arm}]`, to generate Figure 6.5, and

`SimDrive["$", {trace}]` to generate Figure 6.6.

```

x = 2+Cos[om*t]
y = 2+Sin[om*t]

vx=-om*Sin[om*t]
vy= om*Cos[om*t]

ax = -om * om * Cos[om*t]
ay = -om * om * Sin[om*t]

DD = (x*x + y*y -a1*a1 -a2*a2) / (2*a1*a2)

q1d = ArcTan[x,y] - ArcTan[a1+a2*Cos[q2d], a2*Sin[q2d]]
q2d = ArcCos[DD]

j11 = -a1*Sin[q1d] + a2*Cos[q1d+q2d]
j12 = -a2*Sin[q1d+q2d]
j21 = a1*Cos[q1d] + a2*Cos[q1d+q2d]
j22 = a2*Cos[q1d+q2d]
detj = a1*a2*Sin[q2d]

v1d = (1/detj) * (j22*vx - j12*vy)
v2d = (1/detj) * (-j21*vx + j11*vy)

jd11 = -a1*Cos[q1d] * v1d - a2*Cos[q1d+q2d] * (v1d+v2d)
jd12 = -a2*Cos[q1d+q2d] * (v1d+v2d)
jd21 = -a1*Sin[q1d] * v1d - a2*Sin[q1d+q2d] * (v1d+v2d)
jd22 = -a2*Sin[q1d+q2d] * (v1d+v2d)

bbx = ax-jd11*v1d - jd12*v2d
bby = ay-jd21*v1d - jd22*v2d

a1d = (1/detj) * (j22*bbx - j12*bby)
a2d = (1/detj) * (-j21*bbx + j11*bby)

aq1 = a1d + kp1 * (q1d -q1[t]) +kd1*(v1d - q1'[t])
aq2 = a2d + kp2 * (q2d - q2[t]) +kd2*(v2d - q2'[t])

tau[1] = M[[1,1]]*aq1 + M[[1,2]]*aq2 + CM[[1,1]]*q1'[t] + CM[[1,2]] * q2'[t]+ G[[1]]
tau[2] = M[[2,1]]*aq1 + M[[2,2]]*aq2 + CM[[2,1]]*q1'[t] + G[[2]]

om=1
kp1=400
kp2=400
kd1=40
kd2=40

```

**Figure 6.3:** Circular motion control definitions.

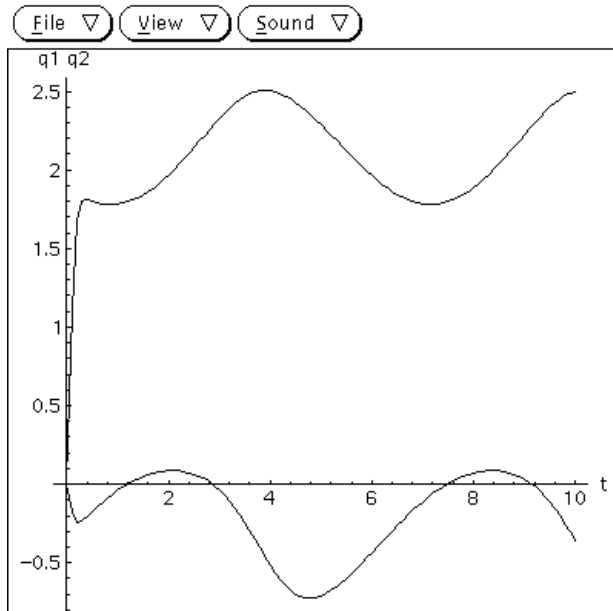


Figure 6.4: Joint variables as a function of time.

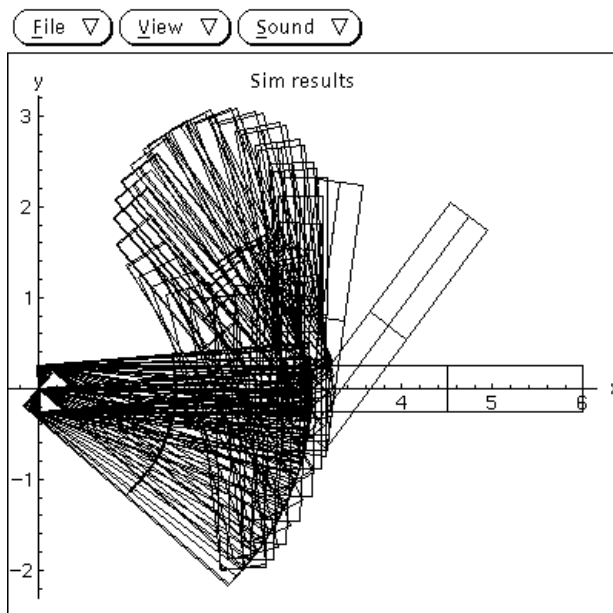
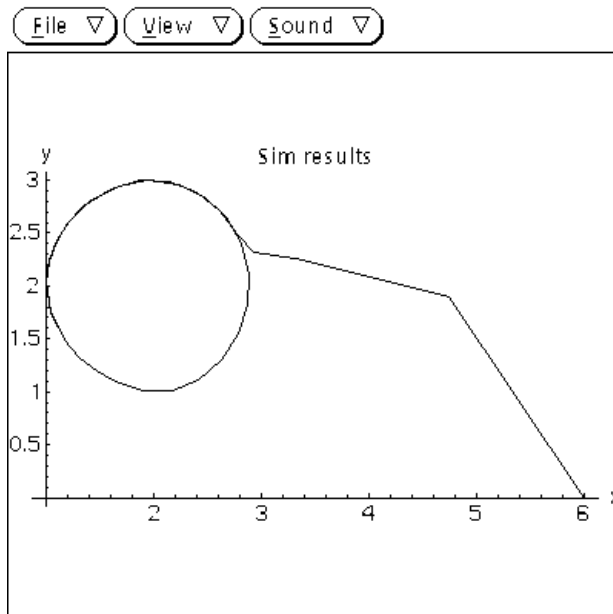


Figure 6.5: Arm location from circular controller.



**Figure 6.6:** End effector location from circular controller.

# CHAPTER 7

## MANIPULABILITY

Another unique feature of *Robotica* is its ability to compute and display the manipulability ellipsoids for a robot described by Denavit-Hartenburg input parameters. Complete information on manipulability can be found in [4]. The manipulability ellipsoids are related to how quickly the end effector can move in a given direction. For a planar arm, the ellipsoids are ellipses in a plane, and the distance from the center of the ellipse to the ellipse is one of the factors that determines how quickly the arm can move in that direction. The case is similar in three-dimensional space, except that the ellipse is an actual three-dimensional ellipsoid. Of course, the manipulability also depends on other factors such as the strength of the joint motors, etc. The manipulability measures that can be computed are valuable pieces of information to have when planning the workspace in which the robot will operate.

### 7.1 Generating the Ellipsoids

`RElp[]` is the function responsible for the calculation and display of the ellipsoids. The format of the `RElp[]` command is:

```
In[5] := RElp[vars, options]
```

This function calculates and displays manipulability ellipsoids for a robot when a range of input parameters is specified. First, of course, a robot must have been loaded with `DataFile[]`; in addition, `FKin[]` must have been run. (`RElp[]` makes heavy use of the Jacobian and `T` matrices calculated with `FKin[]`).

The first argument, called *vars* above, is a list containing a series of sublists. The sublists are of the following form: {variable, start, end}, and *vars* is a sequence of these:



`{{variable1,start,end}, {variable2,start,end},...}`. The sublists specify three things: first, a Denavit-Hartenberg parameter for the variable such as `d1`, `q2`, and `a1` (these are the variable entries in the input parameter file); second, the starting value for that variable; and finally, the ending value for that variable. For example, `{d1, 3, 8}` causes the Denavit-Hartenberg parameter `d1` to run from 3 to 8. The range from start to end is, by default, divided into five pieces, although this can be modified by appending the number of divisions to the end of the variable list. For example, to use eight steps, the variable list would look like this: `{ {q1, 3, 4}, ..., {dn, 0,3}, 8}`. It should be noted that angular parameters are specified in radians; however, using the built-in degrees to radians converter, `Degree`, a value can be given in degrees, e.g., `{q2, 30 Degree, 180 Degree}`. The converter can be placed anywhere after the value, and must be spelled with a capital “D.”

Not all of the input parameters that were specified in the input file have to be assigned a range; however, each must have some definite numerical value. If there is a Denavit-Hartenberg parameter that will have a constant value, it can be assigned with a normal Mathematica command, for example,

```
In[6] := d3 = 5
```

This definition can be cleared later with `Clear[d3]`, at which time the variable will be purely symbolic again. Variables should be cleared before loading a new robot as well.

The variable list and constant parameter assignments should leave no variables in the Jacobian or T matrices with purely symbolic values.

## 7.2 Options

The options are also given in the form of a list. The available options are:

**single:** `RElp[]` plots only the ellipsoid at the start values of the parameters in the sublists.

**frame:** Put a frame around graphics.

- animate:** Take all of the generated frames and show them sequentially in movie fashion. The animation can be shown again with `ShowAnim[]` or saved with `SaveAnim[]`.
- measures:** Print a set of manipulability measures for each ellipsoid (i.e., at each step of parameter values). The measures include the actual axes calculated, volume of the ellipsoid, eccentricity (closer to one means more spherical), minimum radius, and the geometric mean of the radii (a sphere with this radius has the same volume as the ellipsoid; see Chapter 4 of [4]).
- monly:** Causes `RElp[]` to print only manipulability measures; no graphics are generated.
- “file”:** Save the ellipsoid measures in a file called “file” if either *measures* or *monly* is specified in the option list. The filename must be a string (e.g., “measures”) which represents a valid filename on the system being used. If the file already exists, the ellipsoid measure information is appended to the end of the file; otherwise, a new file is created. Also, once a file has been specified, it becomes the default output file. To specify that output should go to the default file, use “\$” as the filename. If no filename is given, the ellipsoid measure information is displayed on the screen only.
- scale:** Specify this option to scale the ellipsoids such that the largest axis is always two units long. This helps to keep the plot from becoming too cluttered with overlapping ellipses.
- print:** If Mathematica has a `LaserPrint` function defined, it will be used to print the graph.

**xprint:** Using this option assumes that Mathematica is running under X-windows; including *xprint* in the options list will attempt to utilize the X-windows utilities *xpr* and *xwd* to allow the user to pick a graphics output window to save as a postscript compatible file. The output file will be called “xrelp.out.”

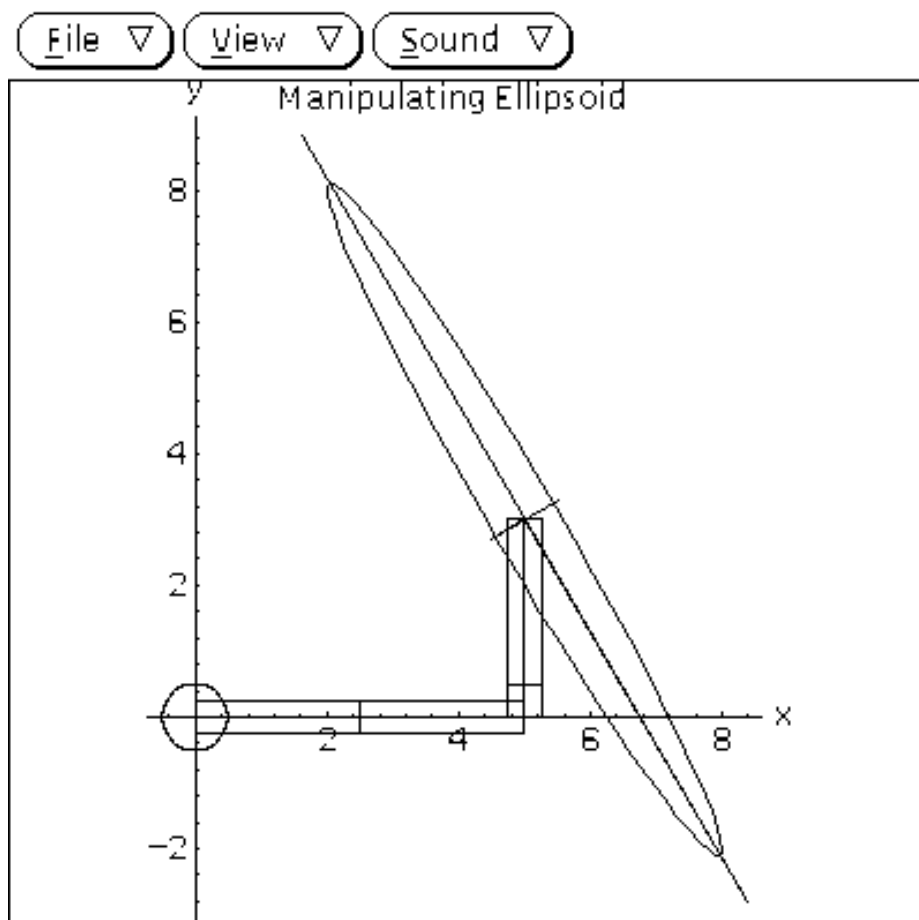
**mprint:** This uses the internal Mathematica command `Display[]` to convert the output graphic to a raw postscript file. Some extra processing may have to be done before the file is sent to a postscript output device. See [2] for details. The output file will be called “mrelp.out.”

Options must be in a list and can be given in any order. For example, `RElp[an, {scale, single}]` causes `RElp[]` to draw a single, scaled ellipsoid at the start values of the parameters in the list *an*.

If no arm graphics have been loaded, simple lines will be used to show the arm. See Chapter 4 for information on the use of more complicated arm graphic representations. Also, a variable called *manplot* is created to hold the current manipulability plot. It can be viewed at any time after a plot has been generated with `Show[manplot]`. Any of the various display options available from the Mathematica function `Show[]`, (e.g., `ViewPoint`) can be used when viewing the plot.

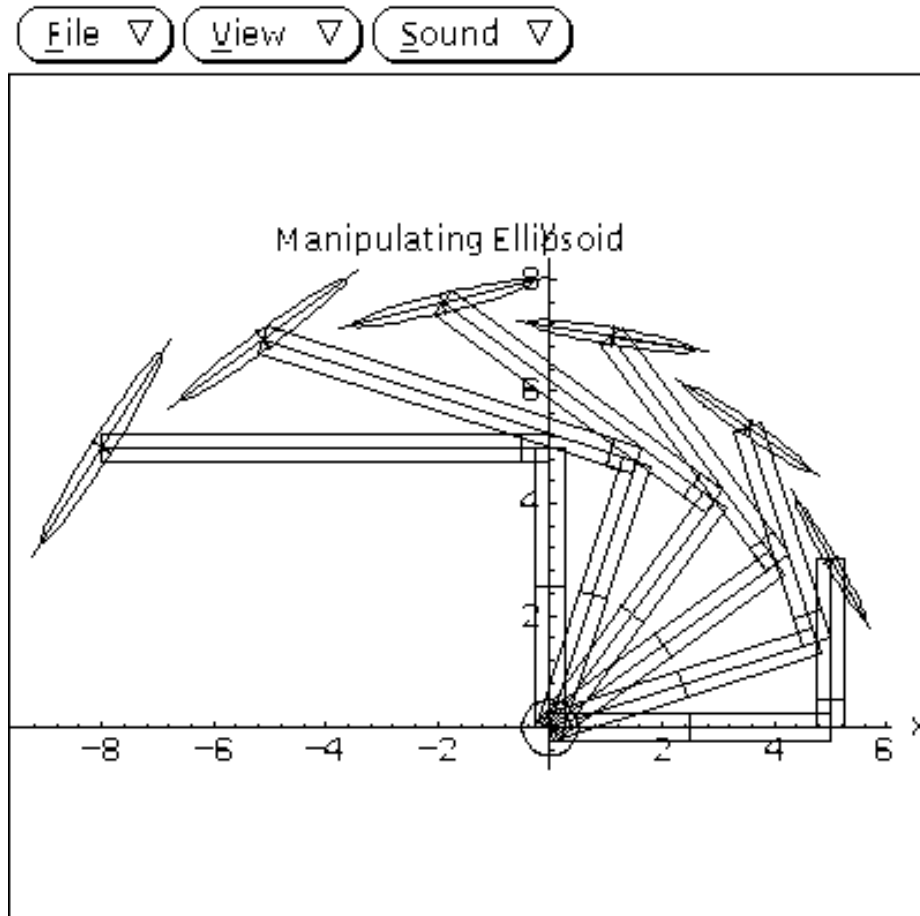
### 7.3 Examples

Figure 7.1 shows a sample of the output produced by `RElp[]` for a simple two-link revolute joint, prismatic joint arm. First, of course, the data file was loaded with `DataFile[]`. Next, a prismatic joint shape was loaded with `PrisJoint[]` and a revolute joint shape with `RevJoint[]`. The parameter `a1` was set to 5, and the command `RElp[{ {q1, 0, 90Degree}, {d2, 3, 8} }, {single}]` was used to generate the graph.



**Figure 7.1:** Planar manipulability example.

When many ellipsoids are drawn, they tend to obscure one another on the graph. Specifying the option *scale* normalizes the size of each ellipsoid such that the largest axis is 2 units long; this is useful for judging the relative amount of manipulability in each configuration. `RElp[{ {q1, 0, 90Degree}, {d2, 3, 8} }, {scale}]` generated the results shown in Figure 7.2.



**Figure 7.2:** Scaling the ellipsoids.

The input data file for a more complicated robot is shown in Figure 7.3. The manipulability for the arm was calculated with  $RElp[\{\{q1, 0, 180Degree\}, \{d3, 2, 8\}, 3\}]$ . The variable  $q2$  was fixed at  $50^\circ$ . Figure 7.4 shows the results.

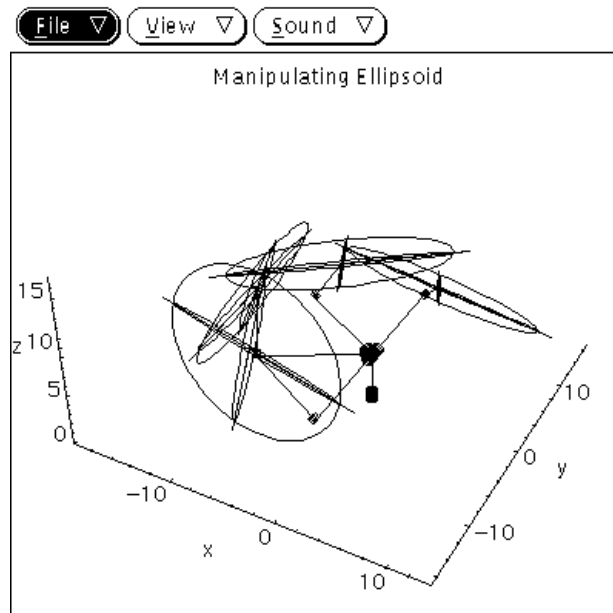
```

DOF      = 3

joint1   = revolute
a1       = 0
alpha1   = -Pi/2
d1       = 5
theta1   = q1
joint2   = revolute
a2       = 0
alpha2   = Pi/2
d2       = 10
theta2   = q2
joint3   = prismatic
a3       = 0
alpha3   = 0
d3       = d3
theta3   = -Pi/2

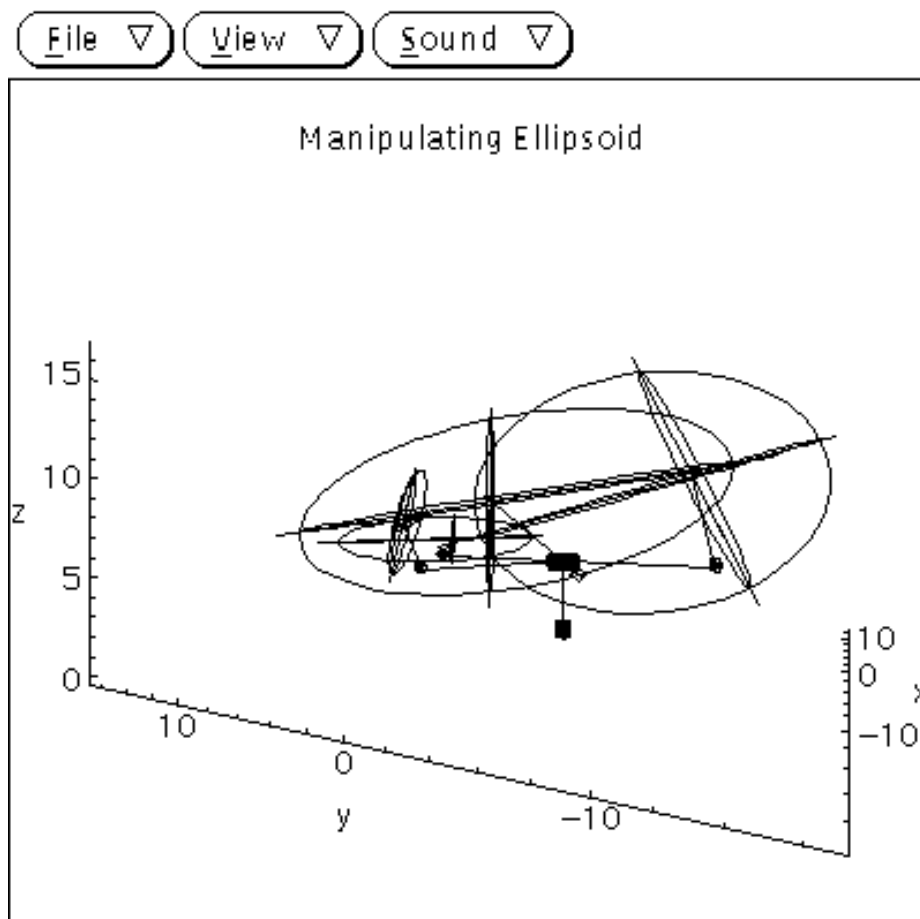
```

**Figure 7.3:** Three-link example.



**Figure 7.4:** Three-link nonplanar example.

With three-dimensional pictures, it can be difficult to see just what is happening as the robot moves. With the ViewPoint option, it is possible to view the plot from any point, allowing a more reasonable perspective to be presented. Figure 7.5 shows the results of moving the viewpoint with `Show[manplot, ViewPoint->{-1,-0.5,0}]`. Even with the ability to adjust the viewpoint, it is usually more informative to generate the numerical measures of manipulability rather than extracting this information from the graphic. See Chapter 4 of [4] for more information on manipulability.



**Figure 7.5:** A change of ViewPoint.

## CHAPTER 8

### EXTERNAL SIMULATION COMPATIBILITY

*Robotica* has the ability to read data from external simulations such as SIMNON [1] and generate graphical output showing the result of applying the joint motion data in the file to the current robot. In addition, the data in the simulation file can be displayed, plotted, and printed in numerous ways.

There are two functions provided to handle simulation compatibility. The first, `SimDrive[]`, begins by reading a simulation input file. It then graphically shows the results of applying the joint parameter values specified by the simulation file to the current robot. The second command, `SimPlot[]`, is used to plot the simulation input data in familiar x-y format by specifying the various quantities to be graphed.

#### 8.1 Loading Simulation Data

The *Robotica* function `SimDrive[]` is used to read the simulation output file – it must adhere to the format shown in Section 8.2. If necessary, a text editor can be used to modify the file so that it matches the required format. `SimDrive[]` requires that an input file of Denavit-Hartenberg parameters has been created for the robot that the simulation output will drive. This input file should be loaded with `DataFile[]` and the forward kinematics should then be generated with `FKin[]`. The format for the `SimDrive[]` command is:

```
SimDrive[filename, options]
```

The filename must be in quotes and designates the file to acquire input from. Once the data are read in, the data in each set are used to update robot arm parameters, and



the graphics are generated and output. The data and variable list is stored for further analysis. To indicate another plot with the same data, use “\$” as the filename.

The options for `SimDrive[]` are held in a list which may contain any of the following:

- trace:** Show a plot of the final coordinate frame location.
  
- arm:** Draw the arm graphics as the arm moves in response to the input data.
  
- frame:** Put a frame around graphics.
  
- animate:** Take all of the generated frames, and show them sequentially in movie fashion. The animation can be shown again with `ShowAnim[]`, or saved with `SaveAnim[]`.
  
- print:** If Mathematica has a `LaserPrint` function defined, it will be used to print the graph.
  
- xprint:** Using this option assumes that Mathematica is running under X-windows; including `xprint` in the options list will attempt to utilize the X-windows utilities `xpr` and `xwd` to allow the user to pick a graphics output window to save as a postscript compatible file. The output file will be called “`xsim.out.`”
  
- mprint:** This uses the internal Mathematica command `Display[]` to convert the output graphic to a raw postscript file. Some extra processing may need to be done before the file is sent to a postscript output device. See the Mathematica users’ manual for details. The output file will be called “`msim.out.`”

A new image of the robot is drawn whenever a specified sample time has elapsed (see Section 8.2, the default being .2 units. A different time can be specified by appending a number to the options list. For example, `SimDrive[“simnon”, {trace, .1}]` would update the plot whenever more than .1 time units elapsed.

If neither *arm* nor *trace* is a member of the options list, or an option list is not given at all, `SimDrive[]` will merely read the input data set and not produce any graphics; of course, the data set is now available to `SimPlot[]` for analysis.

## 8.2 Simulation Input File Format

The input file should be in the following format (the line numbers are for reference only, they should not actually exist in the input file).

```
[1] " t d1 q1
[2] "
[3] 0 1 0.02
[4] 1 1 0.04 ...
```

SIMNON directly supports this output format. Line [1] is a comment line containing the names of the variables for which data is listed in this file. In the above example, data will be specified for *t*, *d1*, and *q1*. It is always assumed that *t* will be the first of the input parameters; it is used in the determination of elapsed time mentioned above. The third line is just a comment line and is ignored; there can be any number of comment lines following the variable list.

The data are listed after the last comment line, with each line containing one set of input values. Thus, if there are three input variables, each line will contain three values. For example, Line [3] would assign 0 to *t*, 1 to *d1* and 0.02 to *q1*. The time parameter *t* is always assumed to be the first variable in the list.

## 8.3 Plotting Simulation Data

The other *Robotica* function dealing with external simulation data is called `SimPlot[]`. Its format is shown below:

```
SimPlot[indep, deplist, options]
```

`SimPlot[]` plots data loaded through the function `SimDrive[]`. The first parameter is the independent variable, i.e., the x-axis. The second parameter is a list of dependent

variables to be plotted against the independent variable. For example, `SimPlot[t, {q1}]` plots  $t$  vs.  $q1$ , and `SimPlot[t, {q1,q2}]` plots  $t$  vs.  $q1$  and  $q2$ .

The parameters *indep* and *deplist* are required, and *options* is optional. The options must be in a list, and can include *print*, *xprint*, and *mprint*. These functions as described in Section 7.1; the files produced, however, are called “*xsimplot.out*” for *xprint*, and “*msimplot.out*” for *mprint*.

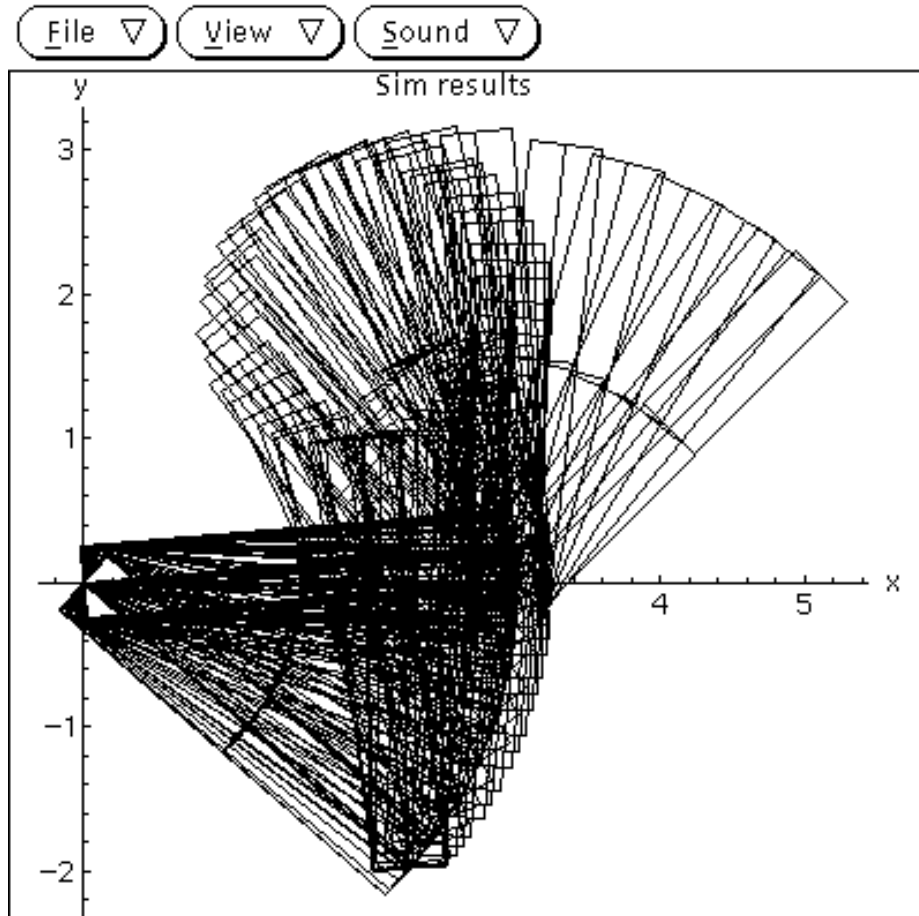
## 8.4 Examples

Assume SIMNON has been used to simulate the effect of a controller attempting to move a two degree of freedom planar arm in a circle. The joint variable data have been stored in the file “simnon.” Figure 8.1 shows how the file would be loaded.

```
In[4]:= SimDrive["Simnon"]
Got header...
Attempting to read values for these parameters:
t
q1
q2
q1d
q2d
Working...
Data file loaded...
```

**Figure 8.1:** Loading a SIMNON file.

Now, assuming that a two degree of freedom planar robot has been loaded, with joint variables matching those specified in the SIMNON data ( $q1$  and  $q2$  here), we can see the effects of the controller as specified by the data set. For this simulation,  $a1$  and  $a2$  were set at 5. For example, entering `SimDrive["$", {arm}]` results in the output shown in Figure 8.2.



**Figure 8.2:** Two link planar arm driven with dataset.

Note the use of “\$” to tell `SimDrive[]` to utilize the data currently loaded and the drawing of a revolute arm shape loaded with `RevLink[]`. Figure 8.3 shows a plot of the origin of the last coordinate frame, which makes the circular motion much easier to see. This plot was generated with `SimDrive["$", {trace}]`.

Now that the data are loaded, we can plot the information using `SimPlot[]`. Figure 8.4 shows an example of plotting time,  $t$ , against  $q1$  and  $q2$ . The command used to generate this graph was `SimPlot[t, {q1, q2}]`.

Finally, Figure 8.5 shows a plot of  $q1$  against  $q2$ , produced by `SimPlot[q1, {q2}]`.

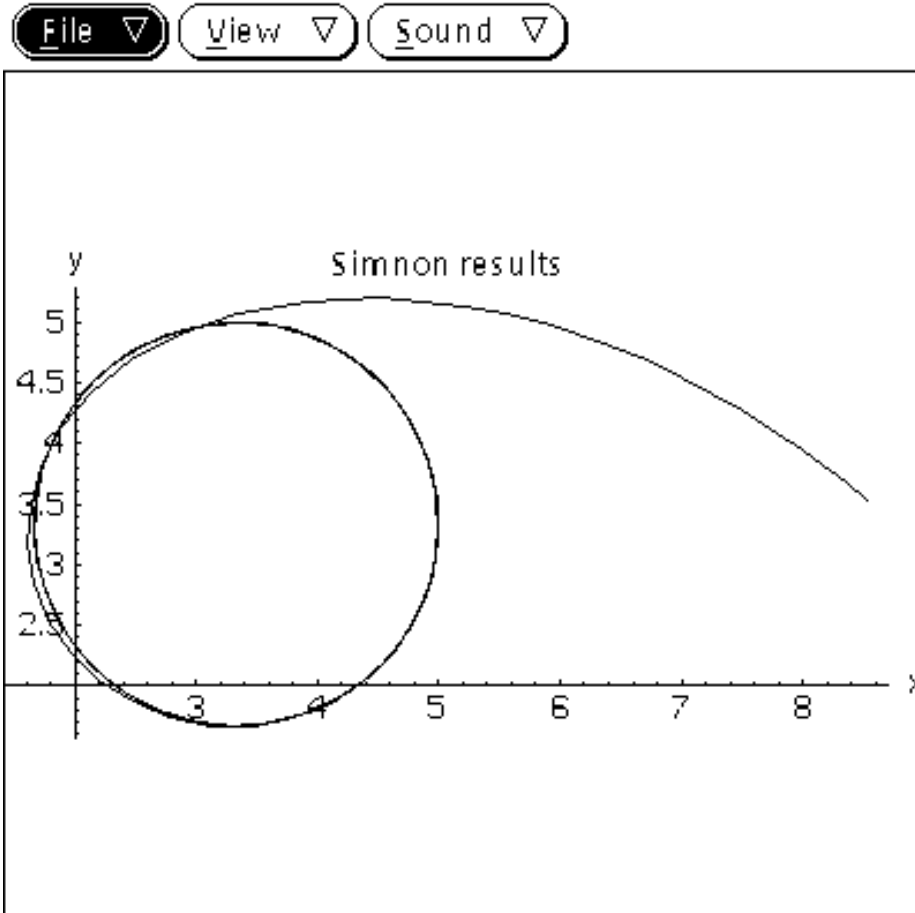
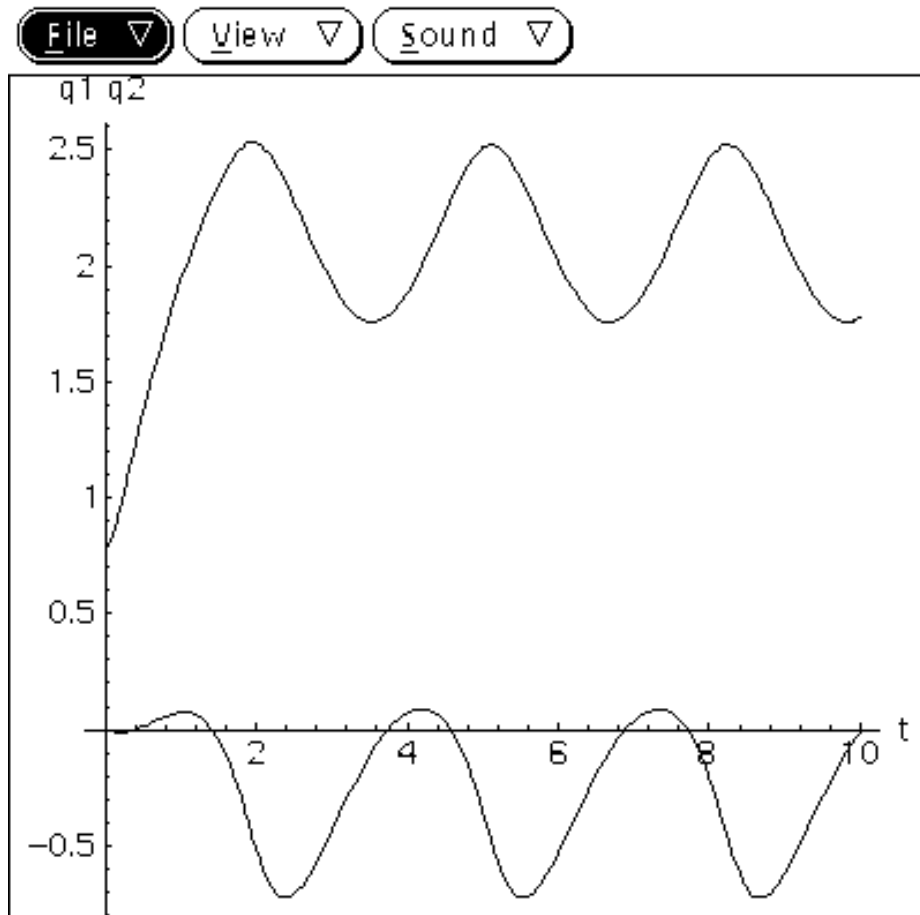


Figure 8.3: Plot of the end effector location.



**Figure 8.4:** Time,  $t$ , versus  $q_1$  and  $q_2$ .

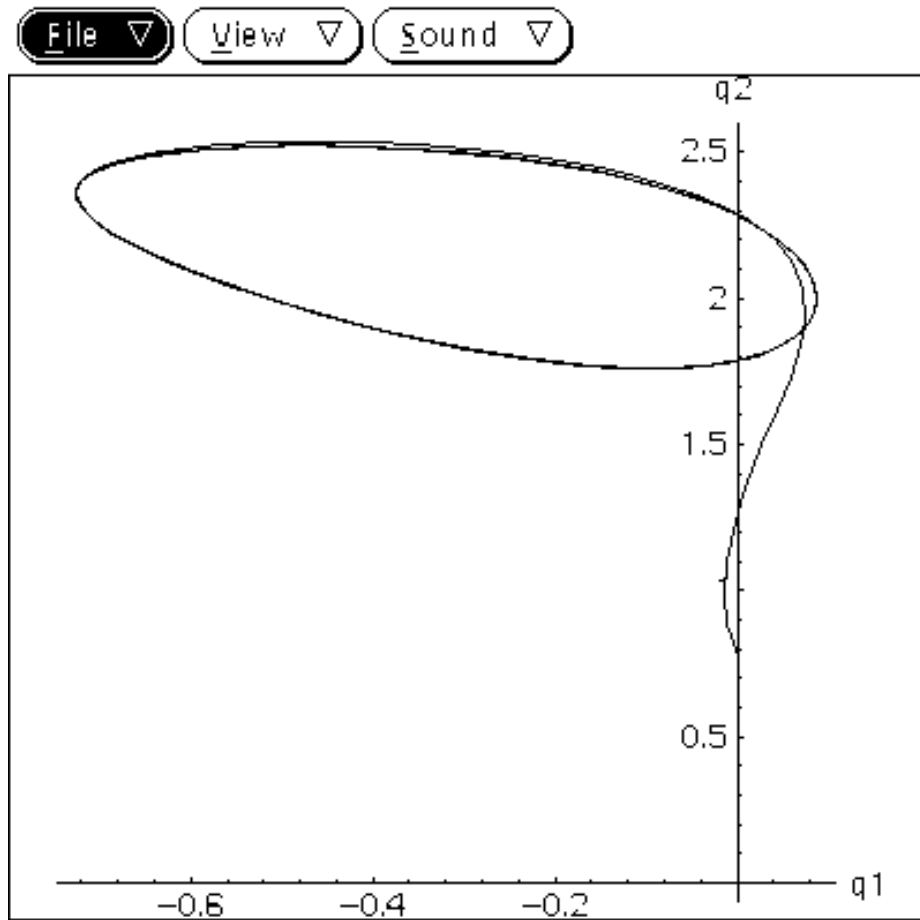


Figure 8.5: Joint parameter  $q_1$  versus  $q_2$ .

# CHAPTER 9

## ROBOTICA FRONT END

The *Robotica* front end (RFE) is a graphical user interface for *Robotica* that allows the user to access the capabilities of the package without having to deal directly with Mathematica. The front end insulates the user from the inconvenience and idiosyncrasies associated with typing commands directly and provides an environment far easier to use than the standard Mathematica textual interface. The interface was written in C using the Open Look widgets [5], [6].

### 9.1 The Structure of RFE

The front end (Figure 9.1) consists of four major elements: a menu bar, located across the top of the window (it contains the file, fonts, and packages menus); a Mathematica output window, located in the left half of the window; a function selection window, located in the right half of the window; and a command entry field, located below the Mathematica output window.

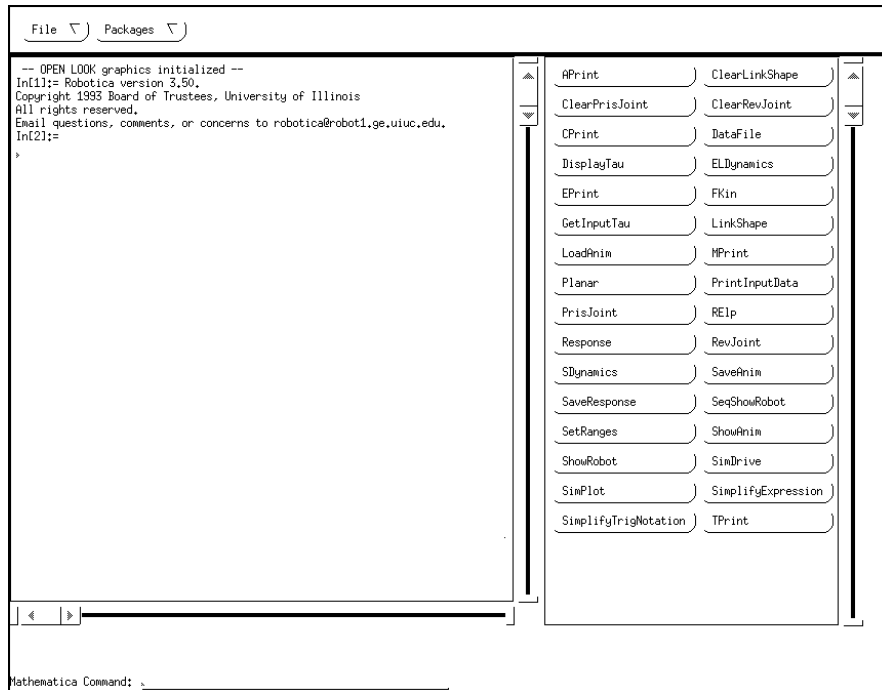
### 9.2 Menu Bar

The menu bar contains pulldown menus called *file*, *fonts*, and *packages*. The menus offer convenient access to features which are frequently needed.

#### 9.2.1 File

Activating the file menu displays the following choices: *Clear Text*, *Abort*, and *Quit*. Selecting *Clear Text* erases everything in the Mathematica output window and sets the





**Figure 9.1:** *Robotica* Front End.

scrollbar slider back to the top, ready for new input. Interrupting an operation can be done with the *Abort* selection. The currently executing Mathematica operation will be stopped, and Mathematica will be ready for new input after the abort sequence. The last operation available under *File* is *Quit*. Not surprisingly, this ends the session with RFE and returns the user to the environment that existed before RFE was run.

### 9.2.2 Packages

This menu contains a short list of Mathematica standard packages that contain functions which add flexibility to *Robotica*. The package *Animation.m*, when loaded, allows the use of the movie-like animation sequences that *RElp[]*, *ShowRobot[]*, etc. can generate. The other two packages, *Shapes.m* and *Polyhedra.m*, are primarily useful for designing new joint shapes. See Chapter 4 for further information on designing joints.

### 9.3 Mathematica Output Window

When Mathematica has an output ready for display, it sends the result to RFE, and RFE posts the output in the Mathematica output window. The window extends beyond the limits of the screen area allocated for it, and therefore scrollbars are provided to allow access to nondisplayed portions of the window. Figure 9.2 shows the output window after the `DataFile[]` and `FKin[]` commands have been executed. When the display fills, the vertical scrollbar is automatically moved down to keep the most recent information visible. Of course, the scrollbars can be moved to another position to display previous output. This window consists of read-only text; to send a command to Mathematica, use the command input line.

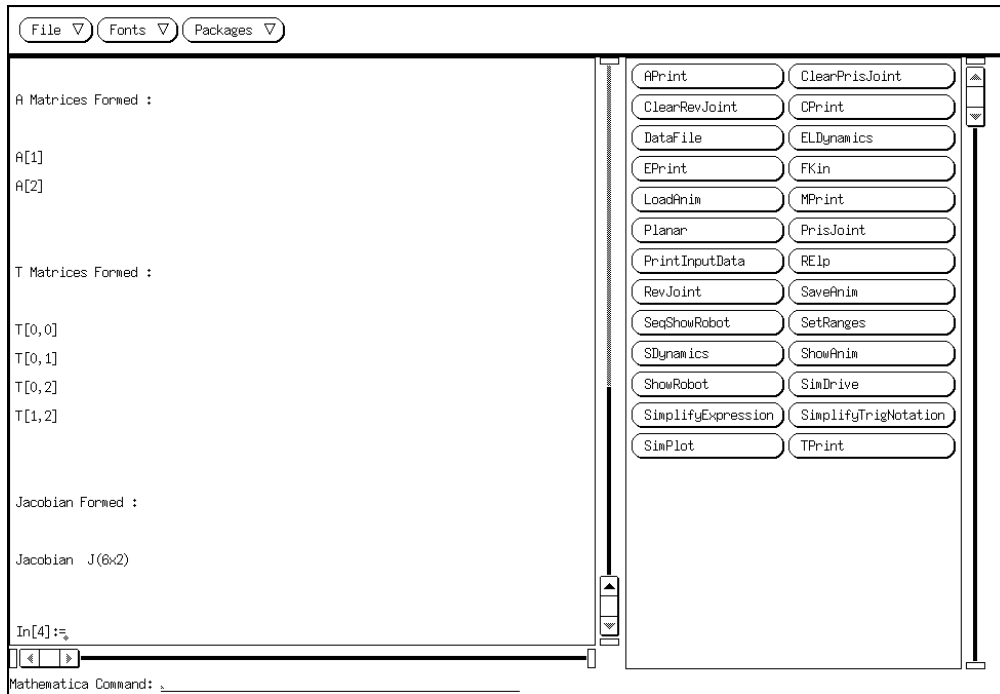


Figure 9.2: Output in the Mathematica output window.

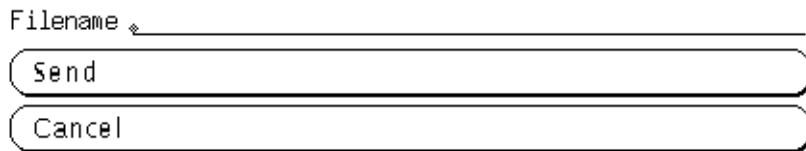
## 9.4 Command Input Line

While the function selection menu provides access to all of the *Robotica* routines, any function or Mathematica command can be entered on the command input line at the bottom of the screen. When a text string is entered on the line and the `return` key pressed, the command is sent to Mathematica, and the output is posted in the output window. The command input line allows the processing of any Mathematica command. For example, entering `Plot[Sin[x], {x, 0, Pi}]` will produce a plot of `Sin[x]` from 0 to `Pi`. In general, any Mathematica command can be executed in this way.

## 9.5 *Robotica* Function Selection Menu

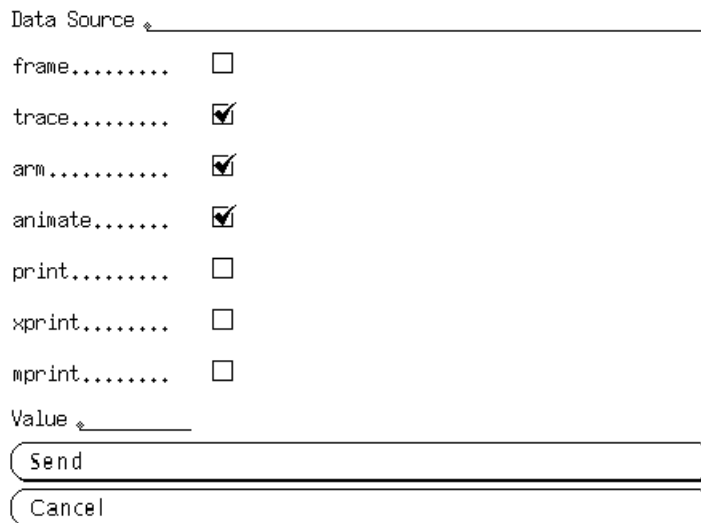
The right side of RFE consists of the *Robotica* Function Selection Menu. In the menu are buttons corresponding to all of the available *Robotica* functions. To execute a *Robotica* procedure, simply click on the button that contains the function name. For every function, a screen will pop up showing the available options as well as a “Send” button and a “Cancel” button. If the function takes no options (such as `FKin[]`), the pop-up screen contains only the “Send” and “Receive” buttons.

Figure 9.3 shows an example of the option screen for the `DataFile[]` function. Here, the only parameter is a filename; pressing “Send” transmits the function to *Robotica*, while pressing “Cancel” prevents the function from being executed. Each pop-up screen remembers what was entered the last time the function was executed, and sets the options to those values.



**Figure 9.3:** DataFile option screen.

Figure 9.4 shows the option screen for the `SimDrive[]` command. The data source field is for the file name from which data are loaded, and below that are listed the various options that can be used with `SimDrive[]`. Each option has a check box to its right to indicate whether or not that option is turned on. A check mark in the box indicates that the option will be turned on when the function is sent, while an empty box means the option is turned off. In Figure 9.4, the options `trace`, `arm`, and `animate` have been turned on. The Value field is a text entry field used to give a value for the step parameter described in the `SimDrive[]` section of Appendix A.



**Figure 9.4:** SimDrive option screen.

When entering parameters into the option screens, the delimiters that usually surround items are automatically added by RFE. For example, no quotation marks are needed around the filename parameter in the DataFile[] option screen. As another example, when specifying lists of parameters to RElp[] or ShowRobot[], only the individual elements inside the variable list have to have braces around them. The delimiters around the entire list are added automatically. In other words, entering {q1,0,90Degree}, {d3,2,4} on the variables line will be sent to *Robotica* as { {q1,0,90Degree}, {d3,2,4} }, which has had the brace delimiters added.

The function selection menu also contains a vertical scrollbar. At this point in time, however, all of the available functions fit on one screen, therefore this scrollbar always stays at the top. When additional functions are added, the scrollbar would be used to move up and down the list of functions, which would be larger than the screen display.

## CHAPTER 10

### COMMAND REFERENCE

#### **APrint[file]**

This function prints all the A matrices for the most recent run of FKIn[]. The file argument is optional and specifies a filename in which to store the matrices. The filename must be a string (e.g., “matrices”) which represents a valid filename on the system being used. If the file already exists, the A matrix information is appended to the end of the file; otherwise, a new file is created. Also, once a file has been specified, it becomes the default file for all of the printing functions. To specify that the output should go to the default file, use “\$” as the filename. If no filename is given, the A matrices are displayed on the screen only.

**Example:** APrint[“amatrix”]

#### **ClearLinkShape[]**

ClearLinkShape[] clears any definition for an interjoint shape. The result is that the package will use a simple line to show links until a new interlink shape is defined with LinkShape[]. ClearLinkShape[] takes no arguments.

**Example:** ClearLinkShape[]

## ClearPrisJoint[]

ClearPrisJoint clears any definition for a prismatic joint shape. The result is that the package will use a simple line to show prismatic joint connections until a new prismatic link shape is defined with PrisJoint[]. ClearPrisJoint[] takes no arguments.

**Example:** ClearPrisJoint[]

## ClearRevJoint[]

ClearRevJoint[] is similar to ClearPris except that it works on the revolute joint shape definition. The revolute joint shape is cleared, and the system will use a line to show revolute joint connections. A new revolute shape can be defined with RevJoint[]. ClearRevJoint[] function takes no arguments.

**Example:** ClearRev[]

## CPrint[label, filename]

CPrint[] prints, one per line, all of the elements of the C matrix (called CM in Robotica). The label parameter is required and is specified as a text string that will be printed in front of the individual elements. The filename is optional and specifies a filename in which to store the elements. The filename must be a string (e.g., “elements”) which represents a valid filename on the system used. If the file already exists, the matrix information is appended to the end of the file; otherwise, a new file is created. In addition, once a file has been specified, it becomes the default file for all of the printing functions. To specify that the output should go to the default file, use “\$” as the filename. If no filename is given, the elements of the matrix are displayed on the screen only.

**Example:** CPrint[“C symbols”]

### **DataFile[filename]**

DataFile[] loads the DH parameters from a file. The filename argument is optional, but, if included, must be a string containing the name of the source file. If the filename is not specified, the routine will query for the name which should be typed in without quotes.

**Example:** DataFile[“twodof”]

### **ELDynamics[]**

ELDynamics[] computes the dynamics information for the current robot data set. First, however, a data set must have been loaded with DataFile[], and the forward kinematics must have been computed with FKin[]. See Table 2.1 for a complete listing of the quantities generated by Robotica.

**Example:** ELDynamics[]

### **EPrint[matrix, label, filename]**

This function prints, one per line, all elements of the matrix or vector specified as the first parameter, which is required. The label argument, also required (although one could make it “” for no label), is a string that will be printed in front of individual matrix elements. The file argument is optional and specifies a filename in which to store the matrices. The filename must be a string (e.g., “elements”) which represents a valid filename on the system used. If the file already exists, the matrix information is appended to the end of the file; otherwise, a new file is created. Also, once a file has been specified, it becomes the default file for all of the printing functions. To specify that the output should go to the default file, use “\$” as the filename. If no filename is given, the elements of the matrix are displayed on the screen only.

**Example:** EPrint[T[0,2], “T02=”, “tfile”]



## **FKin[]**

FKin computes the forward kinematics for the current robot data set. It generates all of the A and T matrices as well as the Jacobian. See Table 2.1 for a complete listing of quantities generated by Robotica.

**Example:** FKin[]

## **GetInputTau[filename]**

GetInputTau[] loads an input vector for the Response[] function. The filename parameter is optional. If specified, a vector definition is loaded from that file, otherwise the torque vector is set to all zeros.

**Example:** GetInputTau["tau1"]

## **LoadAnim[filename]**

LoadAnim[] will load an animation previously saved with SaveAnim[]. Loading a large animation may take several minutes due to the amount of rendering information stored.

**Example:** LoadAnim["anim1"]

## **LinkShape[filename]**

This function loads an interjoint shape from a file to be used in planar arms. The argument to LinkShape[] is a string containing a filename from which to load the shape. If no filename is given, a standard rectangular column shape is loaded as the default link. Currently, shapes must be three dimensional, and can include sets of Line, Polygon, and

Point primitives (the shapes defined in the package `shapes.m` satisfy this requirement). See Chapter 4 for complete information about how arm graphics are handled.

**Example:** `LinkShape["link2"]`

### **MPrint[matrix, label, filename]**

This function prints, in matrix form, the matrix or vector specified as the first parameter, which is required. The label argument, also required (although one could make it "" for no label), is a string that will be printed in front of the matrix. The file argument is optional and specifies a filename in which to store the matrices. The filename must be a string (e.g., "matrix") which represents a valid filename on the system used. If the file already exists, the matrix information is appended to the end of the file; otherwise, a new file is created. Also, once a file has been specified, it becomes the default file for all the printing functions. To specify that the output should go to the default file, use "\$" as the filename. If no filename is given, the matrix is displayed on the screen only.

**Example:** `MPrint[A[2], "A2 = "]`

### **Planar[]**

Planar uses the Jacobian constructed with `FKin[]` and returns an integer which describes the space in which the robot moves. The return codes are:

- 0: The robot can not move in x, y, or z.
- 1: The robot moves only in x.
- 2: The robot moves only in y.
- 3: The robot moves in x and y, i.e., an x-y planar robot arm.
- 4: The robot moves only in z.
- 5: The robot moves in x and z, i.e., an x-z planar robot arm.
- 6: The robot moves in y and z, i.e., a y-z planar robot arm.

7: The robot moves in x, y, and z, i.e., a general motion in 3-space.

**Example:** `Planar[]`

### **PrintInputData[]**

This function prints the input data set to the screen. It does not take any parameters.

**Example:** `PrintInputData[]`

### **PrisJoint[filename]**

This function loads a prismatic joint shape from a file. The argument to `PrisJoint[]` is a string containing a filename from which to load the shape. If no filename is given, a standard rectangular column shape is loaded as the default prismatic joint. Currently, shapes must be three dimensional, and can include sets of `Line`, `Polygon`, and `Point` primitives (the shapes defined in the package `shapes.m` satisfy this requirement). See Chapter 4 for complete information about how arm graphics are handled.

**Example:** `PrisJoint[]`

### **RElp[vars, options]**

This function calculates and displays manipulability ellipsoids for a given robot over a range of input parameters. First, though, a robot must have been loaded with `DataFile[]`, and `FKin[]` must have been run. The first argument is a list containing a series of sublists. The sublists are of the following form: `{var, start, end}`, and `vars` is a sequence of these: `{ {var1,start,end}, {var2,start,end}, ... }`. The sublists specify a DH parameter such as `d1`, `q2`, or `a1` for the variable, and the starting value and ending value for that variable. For example, `{d1, 3, 8}` causes the DH parameter `d1` to vary from 3 to 8. The range

from start to end is, by default, divided into five pieces, although this can be changed by appending the number of divisions to the variable list. To use eight steps, the variable list would look like this:  $\{\{q1, 3, 4\}, \dots, \{dn, 0,3\}, 8\}$ . It should be noted that angular parameters are specified in radians.

Not all parameters need to be specified as a range. If there is a DH parameter that will have a constant value, it can be assigned with a normal Mathematica command (e.g.,  $d3=5$ ). The variable list and constant variables should leave no undefined variables in the Jacobian or T matrices.

The options are given in the form of a list as well. The available options are:

**single:** Plot only one ellipsoid at the start values of the parameters.

**measure:** Print a list of manipulability measures for each ellipsoid. The measures include volume, eccentricity, minimum radius, and geometric mean.

- monly:** Do not generate any graphics, only print manipulability information (or save it to a file).
- frame:** Put a frame around graphics.
- animate:** Take all of the generated frames and show them sequentially in movie fashion. The animation can be shown again with `ShowAnim[]`, or saved with `SaveAnim[]`.
- scale:** Specify this option to scale the ellipsoids such that the largest axis is always two units long. This helps to keep the plot from becoming too cluttered.
- print:** If Mathematica has a `LaserPrint` function defined, it will be used to print the graph.
- xprint:** Using this option assumes that Mathematica is running under X-windows; including `xprint` in the options list will attempt to utilize the X- windows utilities `xpr` and `xwd` to allow the user to pick a graphics output window to save as a postscript compatible file. The file will be called “`xrelp.out`.”
- mprint:** This uses the internal Mathematica command `Display[]` to convert the output graphic to a raw postscript file. Some extra processing may need to be done before the file is sent to a postscript output device. See the Mathematica users’ manual for details. The file will be called “`mrelp.out`.”

**“file”:** Save the ellipsoid measures in a file called “file” if either *measures* or *monly* is specified in the option list. If the file already exists, the information is appended to the end of the file; otherwise, a new file is created. Use “\$” as the filename to indicate that the output should go to the current default file.

Options must be in a list, and can be given in any order. For example, `RElp[an, {scale, single}]` causes `RElp[]` to draw a single, scaled ellipsoid at the start values of the parameters in the list *an*. If no arm graphics have been loaded, simple lines will be used to show the arm. The functions `RevJoint[]` and `PrisJoint[]` can be used to load the standard joint shapes to provide more detail.

**Example:** `RElp[{ {q1, 20Degree, 90Degree} , 8}, {monly, “measures”}]`

### **Response[time, init]**

`Response[]` calculates the time response from the dynamics equations, given an input torque vector (loaded with `GetInputTau[]`). Of course, `ELDynamics[]` must have been run, and all of the symbols in the equations must have been assigned numerical values. The first parameter to `Response[]` is a list containing a start and end time over which to solve. Additionally, a third number may be added to the end of the list; this number sets the maximum number of steps that the algorithms can take to solve the equations. The second parameter is a list containing a set of initial conditions that determines a unique solution to the equations. When successful, the output can be saved with `SaveResponse[]`. See Chapter 6 for complete details.

**Example:** `Response[{0,5}, {q1[0]==0, q1'[0]==0}]`

### **RevJoint[filename]**

This function loads a revolute joint shape from a file. The argument to `RevJoint[]` is a string containing a filename from which to load the shape. If no filename is given, a

standard circular column shape is loaded as the default revolute joint. Currently, shapes must be three-dimensional, and can include sets of Line, Polygon, and Point primitives (the shapes defined in shapes.m satisfy this requirement). See Chapter 4 for complete information on how link graphics are handled.

**Example:** RevJoint[“revolute”]

**SDynamics[]**

Since dynamics quantities are not simplified when generated, this function is provided to instruct Robotica to attempt to reduce the entire set of dynamics information. Depending on the number of links in the robot, this process could take a large amount of time. Specific pieces of information can be simplified with SimplifyExpression[] instead.

**Example:** SDynamics[]

**SaveAnim[filename]**

SaveAnim[] will save an animation generated with RElp[], SeqShowRobot[], etc. in the file specified. Files are usually quite large due to the amount of rendering information stored.

**Example:** SaveAnim[“anim1”]

**SaveResponse[file, time, inc]**

After a successful run of Response[], SaveResponse[] will store numerical values for the solution to the dynamics equations in the file specified as the first parameter. The second parameter is a list containing a start time and a stop time over which to save the solutions. The range specified must, of course, be included in the range solved for with Response[]. The third parameter gives the time increment to be used when saving the data; it defaults to 0.1. The first and second parameters are required, and the third

is optional. The output file is stored in a SIMNON compatible format; as a result, `SimDrive[]` and `SimPlot[]` can be used to plot and analyze the data.

**Example:** `SaveResponse["sinres", {0,5}, 0.01]`

### **SeqShowRobot[vars, options]**

This function displays the robot over a range of joint parameters; it differs from `ShowRobot[]` in that each joint specification in `vars` is executed sequentially rather than simultaneously (i.e., one joint at a time moves). First, though, a robot must have been loaded with `DataFile[]`, and `FKin[]` must have been run. The first argument is a list containing a series of sublists. The sublists are of the following form:  $\{\text{var}, \text{start}, \text{end}\}$ , and `vars` is a sequence of these:  $\{\{\text{var1}, \text{start}, \text{end}\}, \{\text{var2}, \text{start}, \text{end}\}, \dots\}$ . The sublists specify a DH parameter such as `d1`, `q2`, or `a1` for the variable, and the starting value and ending value for that variable. For example,  $\{\text{d1}, 3, 8\}$  causes the DH parameter `d1` to vary from 3 to 8. The range from `start` to `end` is, by default, divided into five pieces, although this can be changed by appending the number of divisions to the variable list. To use eight steps, the variable list would look like this:  $\{\{\text{q1}, 3, 4\}, \dots, \{\text{dn}, 0,3\}, 8\}$ . It should be noted that angular parameters are specified in radians.

Joint parameters that have a constant value should be assigned with a Mathematica command (e.g. `d3=5`) entered at the command line. The options are given in the form of a list as well. The available options are:



- single:** Plot only one arm at the start values of the parameters.
  
- frame:** Put a frame around each graphics frame.
  
- animate:** Take all of the generated frames, and show them sequentially in movie fashion. The animation can be shown again with `ShowAnim[]` or saved with `SaveAnim[]`.
  
- print:** If Mathematica has a `LaserPrint` function defined, it will be used to print the graph.
  
- xprint:** Using this option assumes that Mathematica is running under X- windows; including *xprint* in the options list will attempt to utilize the X-windows utilities `xpr` and `xwd` to allow the user to pick a graphics output window to save as a postscript compatible file. The output file will be called “xrobotplot.out.”
  
- mprint:** This uses the internal Mathematica command `Display[]` to convert the output graphic to a raw postscript file. Some extra processing may have to be done before the file is sent to a postscript output device. See the Mathematica users’ manual for details. The output file will be called “mrobotplot.out.”

In addition, a variable called *robotplot* is created to hold the contents of the current output of `SeqShowRobot`. It can be displayed at any time once generated with `Show[robotplot]`. Any of the various display options available from `Show[]` (e.g., `ViewPoint`) can be used to view the plot.

**Example:** `SeqShowRobot[{ {q1, 20Degree, 90Degree}, {d2, 3, 6}, {d2, 6, 3}, {q1, 90Degree, 20Degree} , 8]`

## **SetRanges[xrange, yrange, zrange]**

This function is used to set the range of display for the animation frames. To ensure that each frame's view is consistent with every other frame, the view limits are set to xrange, yrange, zrange. Each of the ranges should be a list containing a min, max pair. For example, `SetRanges[{-5,5}, {0,10}, {-10,0}]` sets the animation views to include x values from -5 to 5, y values from 0 to 10, and z values from -10 to 0.

**Example:** `SetRanges[{-5,5}, {2,4}, {0,9}]`

## **ShowAnim[]**

ShowAnim shows an animation sequence that was generated with `RElp[]`, `SimDrive[]`, etc. or loaded with `LoadAnim[]`. The package `Animation.m` is required for this to function to execute.

**Example:** `ShowAnim[]`

## **ShowRobot[vars, options]**

This function displays a robot over a range of joint parameters. First, though, a robot must have been loaded with `DataFile[]`, and `FKin[]` must have been run. The first argument is a list containing a series of sublists. The sublists are of the following form: `{var, start, end}`, and vars is a sequence of these: `{ {var1, start, end}, {var2, start, end}, ... }`. The sublists specify a DH parameter such as `d1`, `q2`, or `a1` for the variable, and the starting value and ending value for that variable. For example, `{d1, 3, 8}` causes the DH parameter `d1` to vary from 3 to 8. The range from start to end is, by default, divided into five pieces, although this can be changed by appending the number of divisions to the variable list. To use eight steps, the variable list would look like this: `{ {q1, 3, 4}, ..., {dn, 0,3}, 8}`. It should be noted that angular parameters are specified in radians.

Not all parameters need to be specified as a range. If there is a DH parameter that will have a constant value, it can be assigned with a normal Mathematica command (e.g.,

d3=5). The variable list and constant variables should leave no undefined variables in the T matrices.

The options are given in the form of a list as well. The available options are:

- single:** Plot only one arm at the start values of the parameters.
  
- frame:** Put a frame around each graphics frame.
  
- animate:** Take all of the generated frames and show them sequentially in movie fashion. The animation can be shown again with `ShowAnim[]` or saved with `SaveAnim[]`.
  
- print:** If Mathematica has a `LaserPrint` function defined, it will be used to print the graph.
  
- xprint:** Using this option assumes that Mathematica is running under X- windows; including `xprint` in the options list will attempt to utilize the X-windows utilities `xpr` and `xwd` to allow the user to pick a graphics output window to save as a postscript compatible file. The output file will be called “xrobotplot.out.”
  
- mprint:** This uses the internal Mathematica command `Display[]` to convert the output graphic to a raw postscript file. Some extra processing may have to be done before the file is sent to a postscript output device. See the Mathematica users’ manual for details. The output file will be called “mrobotplot.out.”

In addition, a variable called ‘robotplot’ is created to hold the contents of the current output of `ShowRobot`. It can displayed at any time once generated with `Show[robotplot]`. Any of the various display options available from `Show[]` (e.g., `ViewPoint`) can be used to view the plot.

**Example:** `ShowRobot[{ {q1, 20Degree, 90Degree} , 8}]`

## SimDrive[filename, options]

SimDrive[] is provided to show graphically the results of output created by external simulation programs such as SIMNON. The first step is to create an input file of DH parameters for the robot to display. Then, load the robot with DataFile[] and run FKin[]. The filename must be in quotes and specifies the file to take input from. The input file should be in the following format (which is supported directly by SIMNON):

```
" t d1 q1...  
0 1 0.02  
1 1 0.04
```

The first line is a comment line containing the variables (separated by spaces) for which data are listed in this file. There must be one space between the initial quotation mark and  $t$ . Time must always be the first variable, and there must be at least one other variable. Following the variable list there may be any number of comment lines beginning with a quotation mark. The data are listed after the last comment line, with each line containing one set of input values. The time parameter  $t$  is always assumed to be the first variable in the list. Once the data are read in, the graphics are generated, and output. The data and the variable list are stored for further analysis so that to replot with the same data, the filename can be specified with "\$." The options for SimDrive[] are put in a list which may contain any of the following:

- trace:** Show a plot of the end effector location.
  
- arm:** Draw the arm graphics as the arm moves in response to the input data.
  
- frame:** Put a frame around each graphics frame.
  
- animate:** Take all of the generated frames, and show them sequentially in movie fashion. The animation can be shown again with `ShowAnim[]`, or saved with `SaveAnim[]`.
  
- print:** If Mathematica has a `LaserPrint` function defined, it will be used to print the graph.
  
- xprint:** Using this option assumes that Mathematica is running under X-windows; including `xprint` in the options list will attempt to utilize the X-windows utilities `xpr` and `xwd` to allow the user to pick a graphics output window to save as a postscript compatible file. The output file will be called “`xsim.out.`”
  
- mprint:** This uses the internal Mathematica command `Display[]` to convert the output graphic to a raw postscript file. Some extra processing may have to be done before the file is sent to a postscript output device. See the Mathematica users’ manual for details. The output file will be called “`msim.out.`”

A new image of the robot is drawn whenever the end effector moves a certain absolute distance, which defaults to 0.2. A different distance can be specified by appending a number to the options list. For example, `SimDrive[“simnon”, {trace, 0.1}]` would update the plot whenever the end effector move more that 0.1 absolute units. The plot produced by `SimDrive[]` is saved in the variable `driveplot`.

**Example:** `SimDrive["simtwo", {trace}]`

### **SimplifyExpression[exp]**

`SimplifyExpression[]` applies various trigonometric identities and other simplifying reductions to an expression passed in as the argument. The length of time required to reduce the expression varies with the complexity of the expression.

**Example:** `SimplifyExpression[c[[1,1,1]]]`

### **SimplifyTrigNotation[]**

`SimplifyTrigNotation` takes no parameters. Its purpose is to shorten the notation used for the Sin and Cosine functions to match that found in most robotics texts. Enter `?SimplifyTrigNotation` to obtain the complete list of reductions.

**Example:** `SimplifyTrigNotation[]`

### **SimPlot[indep, deplist, options]**

`SimPlot[]` plots data loaded with `SimDrive[]`. The first parameter is the independent variable, i.e., the x-axis. The second parameter is a list of dependent variables to be plotted against the independent variable. For example, `SimPlot[t, {q1}]` plots  $t$  vs.  $q_1$ , and `SimPlot[t, {q1,q2}]` plots  $t$  vs.  $q_1$  and  $q_2$ . The following options are available for use with `SimPlot`; they should be put in the options list:

- print:** If Mathematica has a LaserPrint function defined, it will be used to print the graph.
- frame:** Put a frame around the plot.
- xprint:** Using this option assumes that Mathematica is running under X-windows; including *xprint* in the options list will attempt to utilize the X-windows utilities *xpr* and *xwd* to allow the user to pick a graphics output window to save as a postscript compatible file. The output file will be called “*xsimplot.out*.”
- mprint:** This uses the internal Mathematica command `Display[]` to convert the output graphic to a raw postscript file. Some extra processing may have to be done before the file is sent to a postscript output device. See the Mathematica users’ manual for details. The output file will be called “*msimplot.out*.”

In addition, appending a number to the end of the options list forces the plot to have an aspect ratio (height divided by width) equal to that number. The plot generated by `SimPlot[]` is saved in the variable *simplot*.

**Example:** `SimPlot[t, {q1, q1d}, {print, 1}]`

### **TPrint[filename]**

This function prints all of the T matrices for the most recent run of `FKin[]`. The file argument is optional and specifies a filename in which to store the matrices. The filename must be a string (e.g., “*Tmatrices*”) which represents a valid filename on the system used. If the file already exists, the T matrix information is appended to the end of the file; otherwise, a new file is created. Also, once a file has been specified, it becomes the default file for all the printing functions. To specify that the output should go to the default file, use “\$” as the filename. The file created is a simple text file and can be sent to any printer or edited with a standard text editor. If no filename is given, the T matrices are displayed on the screen only.

**Example:** TPrint[]

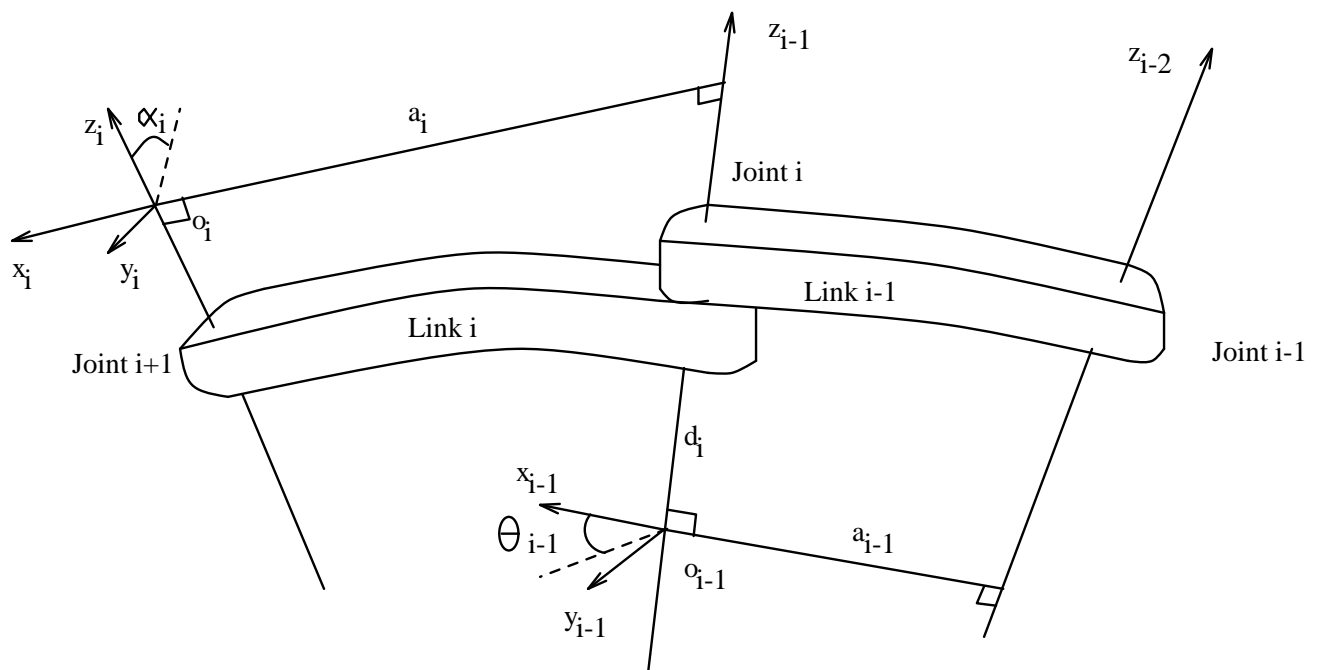


# APPENDIX A

## KINEMATICS AND DYNAMICS EQUATIONS

### Forward Kinematics Equations

Figure A.1 (this is Figure 3.4 in [3]) shows the Denavit-Hartenberg representation defined for successive links.



**Figure A.1:** Denavit-Hartenberg frame assignment [3].

There are four parameters of interest:

$a_i$  = the distance along  $x_i$  from  $o_i$  to the intersection of the  $x_i$  and  $z_{i-1}$  axes.

$d_i$  = the distance along  $z_{i-1}$  from  $o_{i-1}$  to the intersection of the  $x_i$  and  $x_{i-1}$  axes;  $d_i$  is the joint variable if joint  $i$  is prismatic.

$\alpha_i$  = the angle between  $z_{i-1}$  and  $z_i$  along  $x_i$ .

$\theta_i$  = the angle between  $x_{i-1}$  and  $x_i$  measured about  $z_{i-1}$ .  $\theta_i$  is the joint variable if joint  $i$  is revolute.

The homogeneous transformation matrix  $A_i$  is then given by

$$A_i = \begin{bmatrix} C_{\theta_i} & -S_{\theta_i}C_{\alpha_i} & S_{\theta_i}S_{\alpha_i} & a_iC_{\theta_i} \\ S_{\theta_i} & C_{\theta_i}C_{\alpha_i} & -C_{\theta_i}S_{\alpha_i} & a_iS_{\theta_i} \\ 0 & S_{\alpha_i} & C_{\alpha_i} & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where  $C_{\text{angle}} \equiv \text{Cos}(\text{angle})$ , and  $S_{\text{angle}} \equiv \text{Sin}(\text{angle})$ .

The matrix  $A_i$  can be written as

$$A_i = \begin{bmatrix} R_i & d_i \\ 0 & 1 \end{bmatrix}$$

where

$$R_i = \begin{bmatrix} C_{\theta_i} & -S_{\theta_i}C_{\alpha_i} & S_{\theta_i}S_{\alpha_i} \\ S_{\theta_i} & C_{\theta_i}C_{\alpha_i} & -C_{\theta_i}S_{\alpha_i} \\ 0 & -S_{\alpha_i} & C_{\alpha_i} \end{bmatrix}$$

and

$$d_i = \begin{bmatrix} a_iC_{\theta_i} \\ a_iS_{\theta_i} \\ d_i \end{bmatrix}$$

With these quantities defined, the  $T$  matrices are given by (for  $k > j$ )

$$T_j^k = A_{j+1} \dots A_k$$

$$T_j^k = \begin{bmatrix} R_j^k & d_j^k \\ 0 & 1 \end{bmatrix}$$

Once the  $A$  and  $T$  matrices are generated, the Jacobian,  $J$ , can be computed:

$$J = [J_1 \ J_2 \ \dots \ J_n]$$

where the  $i^{\text{th}}$  column  $J_i$  is given by

$$J_i = \begin{bmatrix} Jv_i \\ Jw_i \end{bmatrix} = \begin{bmatrix} z_{i-1} \times (o_n - o_{i-1}) \\ z_{i-1} \end{bmatrix}$$

if joint  $i$  is revolute, and

$$J_i = \begin{bmatrix} Jv_i \\ Jw_i \end{bmatrix} = \begin{bmatrix} z_{i-1} \\ 0 \end{bmatrix}$$

if joint  $i$  is prismatic.

Here,  $z_i$  is the unit vector along the  $z_i$  axis, but expressed in the orientation of the base frame (i.e.,  $z_{i-1} = R_0^{i-1}k$ , where  $k$  is the unit vector for  $z_i$  in the local coordinate frame).

The vector  $o_i$  is the vector  $d_o^i$  from the origin  $O_o$  to the origin  $O_i$ , for each  $i$ . The vector  $z_i$  is merely the first three elements in the third column of  $T_o^i$ , and  $o_i$  is given by the first three elements of the fourth column of  $T_o^i$ .

### Dynamics Equations

As detailed in Chapter 6 of [3], the Euler-Lagrange equations of motion for an  $n$  link rigid robot are given by

$$M(q)\ddot{q} + C(q, \dot{q})\dot{q} + g(q) = \tau$$

where  $q \in R^n$  is a vector of joint variables,  $M(q)$  is the  $n \times n$  inertia matrix,  $C(q, \dot{q})\dot{q}$  contains the Coriolis and centrifugal terms,  $g(q)$  contains the gravitational terms, and  $\tau$  is the input torque.

The inertia matrix is calculated as

$$M = \sum_{i=1}^n (m_i J_{v_{ci}}^T J_{v_{ci}} + J_{w_{ci}}^T R_i I_i R_i^T J_{w_{ci}})$$

where,  $m_i$  is the mass of link  $i$ ,  $I_i$  is the inertia tensor for link  $i$ ,  $R_i$  is the rotation matrix defined above, and  $J_{v_{ci}}$  and  $J_{w_{ci}}$  are given by

$$J_{ci} = \begin{bmatrix} J_{v_{ci}} \\ J_{w_{ci}} \end{bmatrix} = \begin{bmatrix} J_{v_{ci}}^1 & J_{v_{ci}}^2 & \dots & J_{v_{ci}}^i & 0 & \dots & 0 \\ J_{w_{ci}}^1 & J_{w_{ci}}^2 & \dots & J_{w_{ci}}^i & 0 & \dots & 0 \end{bmatrix},$$

where

$$\begin{bmatrix} J_{v_{ci}}^j \\ J_{w_{ci}}^j \end{bmatrix} = \begin{bmatrix} z_{j-1} \times (o_{ci} - o_{j-1}) \\ z_{j-1} \end{bmatrix}$$

if joint  $j$  is revolute ( $o_{ci}$  is the vector to the center of mass of link  $i$ ), and

$$\begin{bmatrix} J_{v_{ci}}^j \\ J_{w_{ci}}^j \end{bmatrix} = \begin{bmatrix} z_{j-1} \\ 0 \end{bmatrix}$$

if joint  $j$  is prismatic.

The  $C$  matrix is an  $n \times n$  matrix containing the Coriolis and centrifugal terms. The  $k, j^{th}$  element is calculated from

$$c_{kj} = \sum_{i=1}^n c_{ijk} \dot{q}_i$$

where the Christoffel symbols are given by

$$c_{ijk} = \frac{1}{2} \left\{ \frac{\partial d_{kj}}{\partial q_i} + \frac{\partial d_{ki}}{\partial q_j} - \frac{\partial d_{ij}}{\partial q_k} \right\}$$

where  $d_{ij}$  is the  $i, j^{th}$  element of  $M$ . The  $j^{th}$  element of the gravity vector is simply

$$g_j = \sum_{i=1}^n m_i g_i^T J_{v_{ci}}^j$$

The material in this appendix is largely taken from Chapter 3 of [3].

## REFERENCES

- [1] H. Elmqvist, *SIMNON, An Interactive Simulation Program for Nonlinear Systems, User's Manual*. Department of Automatic Control, Lund Institute of Technology, April 1975. Report 7502.
- [2] S. Wolfram, *Mathematica: A System for Doing Mathematics by Computer*. Addison-Wesley Publishing Company, Inc, 1991.
- [3] M. W. Spong and Vidyasagar, *Robot Dynamics and Control*. New York, NY: John Wiley & Sons, 1989.
- [4] T. Yoshikawa, *Foundations of Robotics*. The MIT Press, 1990.
- [5] D. A. Young, *The X Window System. Programming & Applications with Xt*. Prentice Hall, 1992.
- [6] Hewlett Packard, *Programming with the Xt Intrinsic*. Hewlett-Packard Company, 1989.