Chapter 3. ptcl: The Ptolemy Interpreter

Authors: *Joseph T. Buck*
Wan-Teh Chang
Edward A. Lee

Other Contributors: *Brian L. Evans*
Christopher Hylands

3.1 Introduction

There are two ways to use Ptolemy: as an *interpreter* and a *graphical* user interface. The Ptolemy Tcl interpreter `ptcl` conveniently operates on dumb terminals and other environments where graphical user interfaces may not be available, and is described in this chapter. The Ptolemy graphical user interface `pigi` is described in chapter 2. When `pigi` is run with the `-console` option, a ptcl window will appear. This combination allows the user to interact with Ptolemy using both graphical and textual commands. Invoking `tycho`, the Ptolemy syntax manager also brings up a ptcl interpreter window. To invoke `tycho` from `pigi`, move the mouse over a facet and type a `y`.

In Ptolemy 0.7, the `tysh` binary contains a prototype of a new interface to the kernel called pitcl. If you start `tycho` with the `-pigi`, `-ptrim`, or `-ptiny` options, then you will be running pitcl, not ptcl. Pitcl is not backward compatible with ptcl, and the pitcl interface is bound to change over time. See the Tycho documentation in `$TYCHO/typt/doc/internals/pticl.html` for further information.

The Ptolemy interpreter, `ptcl`, accepts input commands from the keyboard, or from a file, or some combination thereof. It allows the user to set up a new simulation by creating instances of blocks (stars, galaxies, or wormholes), connecting them together, setting the initial values of internal parameters and states, running the simulation, restarting it, etc. It allows simulations to be run in batch mode. We have used batch mode simulation to run regression tests that compare runs from different versions of Ptolemy.

`Ptcl` is based on John Ousterhout's Tcl (tool command language), which is an extensible interpreted language. All the commands of Tcl are available in `ptcl`. This interface is more convenient than the graphical interface when large complex universes are being created automatically by some other program. Some users also find it more convenient when using a symbolic debugger to debug a new piece of code linked to Ptolemy.

`Ptcl` extends the Tcl interpreter language by adding new commands. The underlying grammar and control structure of Tcl are not altered. Commands in Tcl have a simple syntax: a verb followed by arguments. This document will not explain Tcl; please refer to the manual entry at `$PTOLEMY/tcltk/itcl/html/tcl7.6/Tcl.n.html` which is included with the Ptolemy distribution. Two other excellent references on Tcl are books by Ousterhout [Ous94] and Welch [Wel95]. This chapter describes only the extensions to Tcl made by `ptcl`.

3.2 Getting started

Follow the instructions in the section “Setup” on page 2-1. Now type `ptcl` to invoke the Ptolemy interpreter. It is also possible to specify a file of interpreter commands as a command line argument. See “Loading commands from a file” on page 3-13.

3.3 Global information

The interpreter has a *known list* containing all the classes of stars and galaxies it currently knows about. New stars can be added to the known list at run time only by using the incremental linking facility, but this has restrictions (see the `link` command below). You can also make your own copy of the interpreter with your own stars linked in. Galaxies, however, are easy to add to the known list (see the `defgalaxy` command below).

The interpreter also has a *current galaxy*. Normally, this is the most recently defined universe, or the most recent universe specified with the `curuniverse` command. During the execution of a `defgalaxy` command, which defines a galaxy, the current galaxy is set to be the galaxy being defined. After the closing curly brace of the `defgalaxy` command, the current galaxy is reset to the previous current universe.

3.4 Commands for defining the simulation

This section describes commands to build simulations and add stars, galaxies, states, and the connections among them. The commands are summarized in tables 3-1, 3-2 and 3-3.

3.4.1 Creating and deleting universes

The command

```
univlist
```

will return the list of names of universes that currently exist. The command

```
newuniverse ?name? ?dom?
```

creates a new, empty universe named *`name`* (default “main”) and makes it the current universe with domain *`dom`* (default current domain). If there was previously a universe with this name, it is deleted. Whatever universe was previously the current universe is not affected, unless it was named *`name`*. To remove a universe, simply issue the command:

```
deluniverse ?name?
```

If no argument is given, this will delete the current universe. After this, the current universe will be “main.” To find out what the current universe is, issue the command:

```
curuniverse
```

With no arguments, this returns the name of the current universe. With one argument, as in:

```
curuniverse name
```

it will make the current universe name equal to that argument. A universe can be renamed using either syntax below:

```
renameuniv newname
```

`renameuniv` *`oldname newname`*

With one argument, `renameuniv` renames the current universe to `newname`. With two arguments, it renames the universe named `oldname` to `newname`. Note that any existing universe named `newname` is deleted.

3.4.2 Setting the domain

Ptolemy supports multiple simulation domains. Before creating a simulation environment and running it, it is necessary to establish the domain. The interpreter has a *current domain* which is initially the default domain `SDF`. The command

`domain` *`domain-name`*

changes the current domain; it is only legal when the current galaxy is empty. The argument must be the name of a known domain. The command

command	arguments	description	page
`alias`	*galport b1 p1*	Connect a galaxy port to a block port.	3-6
`animation`	`?on` | `off?`	Enable or disable printing of star names as they fire.	3-11
`busconnect`	*b1 p1 b2 p2 w ?delay?*	Form a bus connection of width *w* between two multi-portholes.	3-6
`cancelAction`	*action_handle*	Cancel an action previously registered using *registerAction.*	3-15
`cd`	*directory*	Change the current directory to the one given.	3-14
`connect`	*b1 p1 b2 p2 ?delay?*	Form a connection between two portholes.	3-5
`cont`	*?num?*	Continue executing the current universe *num* times (default: 1).	3-10
`curuniverse`	*?name?*	Print or set the name of the current universe.	3-2
`defgalaxy`	*name* { *body* }	Define a new galaxy class.	3-8
`delnode`	*name*	Delete the named node from the current galaxy.	3-12
`delstar`	*name*	Delete the named star from the current galaxy.	3-11
`deluniverse`	*?name?*	Delete the current or named universe.	3-2
`descriptor`	*?block?*	Return the descriptor of *block* (default: current galaxy).	3-9
`disconnect`	*b1 p1*	Remove the connection going to the specified port.	3-11
`domain`	*?name?*	Set the domain, or print the name of the current domain.	3-3
`domains`		List the known domains.	3-3
`exit`		Exit ptcl.	3-15
`halt`		Request that the current simulation stop.	3-10
`help`	*?command?*	Print a short description of *command*, or help on *help* if the argument is omitted.	3-15
`knownlist`	*?domain?*	List the known blocks of *domain* (default: current domain).	3-9

TABLE 3-1: First third of the summary of `ptcl` commands. Arguments are in *italic*; literals are in `Courier`; optional arguments are enclosed in question marks. A block name is indicated by *b1* or *b2* and a port name by *p1* or *p2*.

```
domain
```

returns the current domain. It is possible to create wormholes—interfaces between domains—by including a `domain` command inside a galaxy definition. The command

```
domains
```

lists the domains that are currently linked into the interpreter.

3.4.3 Creating instances of stars and galaxies

The first step in any simulation is to define the blocks (stars and galaxies) to be used in the simulation. The command

```
star name class
```

creates a new instance of a star or galaxy of class *class*, names it *`name`*, and inserts it into the current galaxy. Any states in the star (or galaxy) are created with their default values. While it is not enforced, the normal naming convention is that *`name`* begin with a lower case letter and *`class`* begin with an upper case letter (this makes it easy to distinguish instances of a class

command	arguments	description	page
`link`	*objfile*	Incrementally link *objfile* into ptcl.	3-14
`listobjs`	*class ?name?*	List states, ports, or multiports in the named block (default: current galaxy).	3-9
`matlab`	*command ?arg1? ?arg2?*	Manage a Matlab process and evaluate Matlab commands.	3-16
`mathematica`	*command ?arg1? ?arg2?*	Manage a Mathematica process and evaluate commands.	3-16
`multilink`	*linker_args code.o*	Link arbitrary code into the interpreter.	3-14
`newstate`	*name type value*	Define a state for the current galaxy with a default value.	3-6
`newuniverse`	*?name? ?domain?*	Create a new empty universe (defaults: “main” and the current domain).	3-2
`node`	*name*	Create a node for use by *nodeconnect.*	3-6
`nodeconnect`	*b1 p1 node ?delay?*	Connect a porthole to a specified node.	3-6
`numports`	*b1 p1 number*	Force a multiporthole to have a given number of portholes.	3-7
`permlink`	*linker_args code.o*	Link arbitrary code into the interpreter permanently.	3-14
`pragma`	*b1 b2 name value*	Set pragma *name* to *value* for block *b2* in parent *b1.*	3-12
`pragmaDefaults`	*target*	print default values of the pragmas for the target	3-12
`print`	*?b1?*	print a description of block *b1* (or the current galaxy)	3-9

TABLE 3-2: Second third of the summary of `ptcl` commands. Arguments are in *italic*; literals are in `Courier`; optional arguments are enclosed in question marks. A block name is indicated by *b1* or *b2* and a port name by *p1* or *p2*.

from the class itself).

3.4.4 Connecting stars and galaxies

The next step is to connect the blocks so that they can pass data among themselves using the `connect` command. This forms a connection between two stars (or galaxies) by connecting their portholes. A porthole is specified by giving the star (or galaxy) name followed by the port name within the star. The first porthole must be an output porthole and the second must be an input porthole. For example:

```
connect mystar output yourstar input
```

The connect command accepts an optional integer delay parameter. For example:

```
connect mystar output yourstar input 1
```

This specifies one delay on the connection. The delay parameter makes sense only for

command	arguments	description	page
`registerAction`	`pre	post` *command*	Register a Tcl command to be executed before or after stars fire.	3-15
`renameuniv`	*?oldname? newname*	Rename a universe (default: current universe).	3-2
`reset`	*?name?*	Empty a universe (default: "main").	3-11
`run`	*?num?*	Run the current universe *num* times (default: 1).	3-10
`schedtime`	`?actual?`	Print the normalized (default) or unnormalized current scheduler time.	3-11
`schedule`		Generate and print a schedule (only valid for some domains).	3-10
`seed`	*number*	Change or print the random number seed.	3-13
`setstate`	*b1 state_name value*	Change the state of a block to *value*.	3-7
`source`	*filename*	Read commands from the specified file.	3-13
`star`	*name class*	Create a named instance of a star from the given class.	3-4
`stoptime`		Return the stop time of the current run.	3-10
`statevalue`	*b1 name* `?current	initial?`	Print the current or initial value of state *name* in block *b1*.	3-15
`target`	*?newtarget?*	Change or display the name of the current target.	3-12
`targetparam`	*name ?value?*	Change or display the value of a target state.	3-12
`targets`	*?domain?*	List targets usable with *domain* (default: current domain).	3-12
`topblocks`	*?block_or_classname?*	List top-level blocks of the named block (default: current galaxy).	3-15
`univlist`		List the names of all defined universes.	3-2
`wrapup`		Invoke the wrapup method of all the blocks.	3-10

TABLE 3-3: Final third of the summary of `ptcl` commands. Arguments are in *italic*; literals are in `Courier`; optional arguments are enclosed in question marks. A block name is indicated by *b1* or *b2* and a port name by *p1* or *p2*.

domains that support it. The delay argument may be an integer expression with variables referring to galaxy parameters as well.

One or both of the portholes may really be a `MultiPortHole`. If so, the effect of doing the connect is to create a new porthole within the `MultiPortHole` and connect to that (see also the `numports` command).

3.4.5 Netlist-style connections

As an alternative to issuing connect commands (which specify point-to-point connections) you may specify connections in a netlist style. This syntax is used to connect an output to more than one input, for example (this is called *auto-forking*). Two commands are provided for this purpose. The `node` command creates a node:

```
node nodename
```

The `nodeconnect` command connects a porthole to a node:

```
nodeconnect starname portname nodename ?delay?
```

Any number of portholes may be connected to a node, but only one of them can be an output node.

3.4.6 Bus connections between MultiPortHoles

A pair of multiportholes can be connected with a bus connection, which means that each multiporthole has *N* portholes and they all connect in parallel to the corresponding port in the other multiporthole. The syntax for creating such connections is

```
busconnect srcstar srcport dststar dstport width ?delay?
```

Here *`width`* is an expression specifying the width of the bus (how many portholes in the multiportholes); and *`delay`* is an optional expression giving the delay on each connection. The other arguments are identical to those of the `connect` command.

3.4.7 Connecting internal galaxy stars and galaxies to the outside

When you define a new galaxy there are typically external connections to that galaxy that need to be connected through to internal blocks. The `alias` command is used to add a porthole to the current galaxy, and associate it with an input or output porthole of one of the contained stars within the galaxy. An example is:

```
alias galaxyin mystar starin
```

This also works if *`starin`* is a `MultiPortHole` (the galaxy will then appear to have a multiporthole as well).

3.4.8 Defining parameters and states for a galaxy

A *state* is a piece of data that is assigned to a galaxy and can be used to affect its behavior. Typically the value of a state is coupled to the state of blocks within the galaxy, allowing you to customize the behavior of blocks within the galaxy. A *parameter* is the initial value of a state. The `newstate` command adds a state to the current galaxy. The form of the command is

```
newstate state-name state-class default-value
```

The *`state-name`* argument is the name to be given to the state. The *`state-class`* argument is the type of state. All standard types are supported (see table 2-6 on page 2-33). The *default-value* argument is the default value to be given to the state if the user of the galaxy does not change it (using the `setstate` command described below). The *default-value* specifies the initial value of the state, and can be an arbitrary expression involving constant values and other state names; this expression is evaluated when the simulation starts. The following state names are predefined: `YES`, `NO`, `TRUE`, `FALSE`, `PI`. `YES` and `TRUE` have value 1; `NO` and `FALSE` have value 0; `PI` has the value 3.14159... Some examples are:

```
newstate count int 3
newstate level float 1.0
newstate title string "This is a title"
newstate myfreq float galaxyfreq
newstate angularFreq float "2*PI*freq"
```

The full syntax of state initial value strings depends on the type of state, and is explained in "Parameters and states" on page 2-14.

3.4.9 Setting the value of states

The `setstate` command is used to change the value of a state. It can be used in three contexts:

- Change the value of a state for a star within the current galaxy.
- Change the value of a state for a galaxy within the current galaxy.
- Change the value of a state within the current galaxy.

The latter would normally be used when you want to perform multiple simulations using different parameter values. The syntax for `setstate` is:

setstate *block-name state-name value*

Here,

- *block-name* is either the name of a star or a galaxy that is inside the current galaxy, and it is the block for which the value of the state is to be changed. It can also be `this`, which says to change a state belonging to the current galaxy itself.
- *state-name* is the name of a state which you wish to change.
- *value* is the new value for the state. The syntax for *value* is the same as described in the `newstate` command. However, the expression for *value* may refer to the name of one or more states in the current galaxy or an ancestor of the current galaxy.

An example of the use of `setstate` is given in the section describing `defgalaxy` below.

3.4.10 Setting the number of ports to a star

Some stars in Ptolemy are defined with an unspecified number of multiple ports. The number of connections is defined by the user of the star rather than the star itself. The `numports` command applies to stars that contain such `MultiPortHoles`; it causes a specified number of `PortHoles` to be created within the `MultiPortHole`. The syntax is

numports *star portname n*

where *star* is the name of a star within the current galaxy, *portname* is the name of a `Mul-`

`tiPortHole` in the star, and *`n`* is an integer, representing the number of `PortHoles` to be created. After the portholes are created, they may be referred to by appending `#i`, where *`i`* is an integer, to the multiporthole name, and enclosing the resulting name in quotes. The main reason for using this command is to allow the portholes to be connected in random order. Here is an example:

```
star summer Add
numports summer input 2
alias galInput summer "input#1"
connect foo output summer "input#2"
```

3.4.11 Defining new galaxies

The `defgalaxy` command allows the user to define a new class of galaxy. The syntax is

```
defgalaxy class-name {
      command
      command
      ...
}
```

Here *`class-name`* is the name of the galaxy type you are creating. While it is not required, we suggest that you have the name begin with a capital letter in accordance with our standard naming convention — class names begin with capital letters. The *`command`* lines may be any of the commands described above — `star`, `connect`, `busconnect`, `node`, `nodeconnect`, `numports`, `newstate`, `setstate`, or `alias`. The defined class is added to the known list, and you can then create instances of it and add them to other galaxies. An example is:

```
reset
domain SDF
defgalaxy SinGen {
      domain SDF
      # The frequency of the sine wave is a galaxy parameter
      newstate freq float "0.05"
      # Create a star instance of class "Ramp" named "ramp"
      star ramp Ramp
      # The ramp advances by 2*pi each sample
      setstate ramp step "6.283185307179586"
      # Multiply the ramp by a value, setting the frequency
      star gain Gain
      # The multiplier is set to "freq"
      setstate gain gain "freq"
      # Finally the sine generator
      star sin Sin
      connect ramp output gain input
      connect gain output sin input
      # The output of "sin" becomes the galaxy output
      alias output sin output
}
```

In this example, note the use of states to allow the frequency of the sine wave generator to be changed. For example, we could now run the sine generator, changing its frequency to “0.02”, with the interpreter input:

```
star generator SinGen
setstate generator freq "0.02"
star printer Printer
connect generator output printer input
run 100
```

You may include a `domain` command within a `defgalaxy` command. If the inside domain is different from the outside domain, this creates an object known as a `Wormhole`, which is an interface between two domains. An example of this appears in a later section.

3.5 Showing the current status

The following commands display information about the current state of the interpreter.

3.5.1 Displaying the known classes

The `knownlist` command returns a list of the classes of stars and galaxies on the known list that are usable in the current domain. The syntax is

```
knownlist
```

It is also possible to ask for a list of objects available in other domains; the command

```
knownlist DE
```

displays objects available in the `DE` (discrete event) domain.

3.5.2 Displaying information on a the current galaxy or other class

If invoked without an argument, the `print` command displays information on the current galaxy. If invoked with an argument, the argument is either the name of a star (or galaxy) contained in the current galaxy, or the name of a class on the known list, and information is shown about that star (or galaxy). The syntax is

```
print
print star-name
print star-class
```

The command

```
descriptor ?name?
```

will print a short description of a block in the current galaxy or on the known list, or of the current galaxy if *`name`* is omitted. The commands

```
listobjs states ?name?
listobjs ports ?name?
listobjs multiports ?name?
```

will list the names of the states, ports, or multiportholes associated with the named star or galaxy.

3.6 Running the simulation

Once a simulation has been constructed using the commands previously described (also see the `source` command in “Loading commands from a file” on page 3-13), use the commands in this section to run the simulation.

3.6.1 Creating a schedule

The `schedule` command generates and returns the schedule (the order in which stars are invoked). For domains such as DE, this command returns a not-implemented message (since there is no “compile time” DE schedule as there is for SDF). The syntax is:

```
schedule
```

3.6.2 Running the simulation

The `run` command generates the schedule and runs it *n* times, where *n* is the argument (the argument may be omitted; its default value is 1). For the DE interpreter, this command runs the simulation for *n* time units, and *n* may be a floating point number (default 1.0). If this command is repeated, the simulation is started from the beginning. If animation is enabled, the full name of each star will be printed to the standard output when the star fires. The syntax is:

```
run
run n
```

3.6.3 Continuing a simulation

The `cont` command continues the simulation for *n* additional steps, or time units. If the argument is omitted, the default value of the argument is the value of the last argument given to a `run` or `cont` command (1.0 if no argument was ever given). The syntax is

```
cont
cont n
```

3.6.4 Wrapping up a simulation

The `wrapup` command calls the wrapup method of the current target (which, as a rule, will call the `wrapup` method of each star), signaling the end of the simulation run. The syntax is

```
wrapup
```

3.6.5 Interrupting a simulation

The command

```
halt
```

requests a halt of the currently executing simulation. Note that the halt does not occur immediately. This merely registers the request with the scheduler. This is especially useful within Tcl stars.

3.6.6 Obtaining the stop time of the current run

The command

```
stoptime
```

returns the time until which the current simulation will run. Tcl/Tk stars can use this command in their setup or go methods to find out the stop time of the current run.

3.6.7 Obtaining time information from the scheduler

The command

```
schedtime
```

returns the current time from the top-level scheduler of the current universe. If the target has a parameter named "schedulePeriod", then the returned time is divided by this value. The command

```
schedtime actual
```

returns the scheduler time without dividing by "schedulePeriod."

In SDF, `schedtime actual` should return the number of iterations. In SDF, "schedulePeriod" is usually set to 0, since in SDF has no notion of time, and to a timed domain, such as DE, SDF universes appear to fire instantaneously.

3.6.8 Animating a simulation

The `animation` command can be used to display on the standard output the name of each star as it runs. The syntax

```
animation on
```

enables animation, while

```
animation off
```

disables it. The syntax

```
animation
```

simply tells you whether animation is enabled or disabled.

3.7 Undoing what you have done

The commands in this section remove part or all of the structure you have built with previous commands.

3.7.1 Resetting the interpreter

The `reset` command replaces the universe `main` or a named universe by an empty universe. Any `defgalaxy` definitions you have made are still remembered. The syntax is

```
reset
reset universe_name
```

3.7.2 Removing a star

The `delstar` command removes the named star from the current galaxy. The syntax is

```
delstar name
```

where *name* is the name of the star.

3.7.3 Removing a connection

The `disconnect` command reverses the effect of a previous `connect` or `nodeconnect` command. The syntax is

```
disconnect starname portname
```

where *`starname`* and *`portname`*, taken together, specify one of the two connected portholes. Note that you can disconnect by specifying either end of a porthole for a point-to-point connection.

3.7.4 Removing a node

The `delnode` command removes a node from the current galaxy. Syntax:

```
delnode node
```

3.8 Targets

Ptolemy uses a structure called a *target* to control the execution of a simulation, or, in code generation, to control code generation, compilation, and execution. There is always a target; by default (if you issue no target commands), your target will have the name `default-XXX`, where `XXX` is replaced by the name of the current domain. Alternative targets for simulation can be used to specify different behavior (for example, to use a different scheduler or to analyze a schematic rather than running a simulation). For code generation, the target contains information about the target of compilation, and has methods for downloading code and starting execution.

3.8.1 What targets are available?

The command

```
targets
```

returns the list of targets available for the current domain. The command

```
targets domain
```

returns the list of targets available for *domain*.

3.8.2 Changing the target

The command

```
target
```

displays the target for the current universe or current galaxy, together with its parameters. Specifying an argument, e.g.

```
target new-target-name
```

changes the target to *`new-target-name`*.

3.8.3 Changing target parameters

Target parameters may be queried or changed with the `targetparam` command. The syntax is

```
targetparam param-name ?new-value?
```

3.8.4 Pragmas

Ptolemy can use target pragmas as a generalization of the attribute mechanism to inform the target of the user's wishes. The Dynamic Dataflow (DDF) domain uses pragmas to specify the number of firings of a star required in one iteration. The C Code Generation (CGC) domain uses pragmas to identify any parameters that the user would like to change on the command line. See "Setting Parameters Using Command-line Arguments" on page 14-4.

`pragma` *`b1 b2 name value`*

Set pragma *`name`* to *`value`* for block *`b2`* in parent *`b1`*.

`pragmaDefaults` *`target`*

Print the default values of the pragmas for the target.

3.9 Miscellaneous commands

This section describes the remaining interpreter commands.

3.9.1 Loading commands from a file

For complicated simulations it is best to store your interpreter commands—at least those defining the simulation connectivity—in a file rather than typing them into the interpreter directly. This way you can run your favorite editor in one window and run the interpreter from another window, easily modifying the simulation and also keeping a permanent record. Two exceptions to this are changing states using the `setstate` command and running and continuing the simulation using `run` and `cont`—this is normally done interactively with the interpreter.

The `source` command reads interpreter commands from the named file, until the end of the file or a syntax error occurs. The "#" character indicates that the rest of the line is a comment. By convention, files meant to be read by the load command end in ".pt". Example:

```
source "testfile.pt"
```

The tilde notation for users' home directories is allowed; for example, if your installation of Ptolemy was made by creating a user `ptolemy` (see "Setup" on page 2-1), try

```
source "$PTOLEMY/demo/ptcl/sdf/basic/butterfly.pt"
```

It is also possible to specify a file to be loaded by the interpreter on the command line. If, when you start the interpreter you type

```
ptcl myCommands.pt
```

the interpreter will load the named file, execute its commands, and then quit. No command prompt will appear. The `source` command is actually built into Tcl itself, but it is described here nevertheless, for convenience.

3.9.2 Changing the seed of random number generation

The `seed` command changes the seed of the random number generation. The default value is 1. The syntax is

`seed` *`n`*

where *`n`* is an unsigned integer.

3.9.3 Changing the current directory

The `cd` command changes the current directory. For example,

```
cd "$PTOLEMY/demo/ptcl/sdf/basic"
source "butterfly.pt"
```

will load the same file as the example in the previous section. Again, we have assumed that your installation contains a user `ptolemy` (see "Setup" on page 2-1). To see what the interpreter's current directory is, you can type

```
pwd
```

3.9.4 Dynamically linking new stars

The interpreter has the ability to extend itself by linking in outside object files; the object files in question must define single stars (they will have the right format if they are produced from preprocessor input). Unlike `pigi`, the graphical interface, the interpreter will not automatically run the preprocessor and compiler; it expects to be given object files that have already been compiled. The syntax is

link *object-file-name*

Any star object files that are linked in this way must only call routines that are already statically or permanently linked into the interpreter. For that reason, it is possible that a star that can be linked into `pigi` might not be linkable into the interpreter, although this is rare. Specifically, `pigi` contains Tk, an X window toolkit based on Tcl, while `ptcl` does not. Hence, any star that uses Tk is excluded from `ptcl`.

Building object files for linking into Ptolemy can be tricky since the command line arguments to produce the object file depend on the operating system, the compiler and whether or not shared libraries are used. `$PTOLEMY/mk/userstars.mk` includes rules to build the proper object file for a star. See "Dynamic linking fails" on page A-30. for hints about fixing incremental linking problems.

It is also possible to link in several object files at once, or pull in functions from libraries by use of the `multilink` command. The syntax is

multilink *opt1 opt2 opt3 ...*

where the options may be the names of object files, linker options such as "-L" or "-l" switches, etc. These arguments are supplied to the Unix linker along with whatever options are needed to completely specify the incremental link.

When the above linker commands are used, the linked code has temporary status; symbols for it are not entered into the symbol table (meaning that the code cannot be linked against by future incremental links), and it can be replaced; for example, an error in the loaded modules could be corrected and the `link` or `multilink` command could be repeated. There is an alternative linking command that specifies that the new code is to be considered "permanent"; it causes a new symbol table to be produced for use in future links (See the ptlang `derivedFrom` item in the Ptolemy Programmers Manual for more information). Such code cannot be replaced, but it can be linked against by future incremental link commands. The syntax is

permlink *opt1 opt2 opt3 ...*

where the options are the same as for the `multilink` command.

3.9.5 Top-level blocks

The command

```
topblocks
```

returns the list of top-level blocks in the current galaxy or universe. With an argument,

```
topblocks block
```

it returns the list of top-level blocks in the named block.

3.9.6 Examining states

The `statevalue` command takes the form

```
statevalue block state
```

and returns the current value of the state *`state`* within the block *`block`*. The command takes an optional third argument, which may be either "`current`" to specify that the current value should be returned (the default), or "`initial`" to specify that the initial value (the parameter value) should be returned.

3.9.7 Giving up

The `exit` command exits the interpreter. The syntax is

```
exit
```

3.9.8 Getting help

The `help` command implements a simple help system describing the commands available and their syntax. It does not provide help with the standard Tcl functions. The syntax is

```
help topic
```

or

```
help ?
```

for a list of topics. If the argument is omitted, a short "help on help" is printed.

3.9.9 Registering actions

It is possible to associate a Tcl action with the firing of any star. The `registerAction` command does this. The syntax is

```
registerAction pre tcl_command
registerAction post tcl_command
```

The first argument specifies whether the action should occur before or after the firing of a star. The second argument is a string giving the first part of a tcl command. Before this command is invoked, the name of the star that triggered the action will be appended as an argument. For example:

```
registerAction pre puts
```

will result in the name of a star being printed on the standard output before it is fired. A typical "action" resulting from this command would be

```
puts universe_name.galaxy_name.star_name
```

The value returned by `registerAction` is an "action_handle", which must be used to cancel the action using `cancelAction`. The syntax is

```
set action_handle [registerAction pre tcl_command]
cancelAction action_handle
```

3.9.10 The Interface to Matlab and Mathematica

Ptcl can control Matlab [Han96] and Mathematica [Wol92] processes by means of the `matlab` and `mathematica` commands. The commands have a similar syntax:

```
matlab command ?arg1? ?arg2?
mathematica command ?arg1? ?arg2?
```

The `matlab` command controls the interaction with a shared Matlab process. The possible commands and arguments are:

command	arguments	description
`end`		terminate a session with Matlab
`eval`	*script*	evaluate a Matlab script and print the result
`get`	*name script*	evaluate a Matlab script and get the named Matlab matrix as Tcl lists of numbers
`getpairs`	*name script*	evaluate a Matlab script and get the named Matlab matrix as ordered pairs of numbers
`send`	*script*	evaluate a Matlab script and suppress the output
`set`	*name rows cols real imag*	set the named Matlab matrix with real and imaginary values
`start`		start a new Matlab session
`status`		return the status of the Tcl/Matlab connection (0 means connected, -1 means not initialized, and 1 means error)
`unset`	*name*	unset the named Matlab matrix

The `mathematica` command controls the interaction with a shared Mathematica process. The possible commands and arguments are

command	arguments	description
`end`		terminate a session with Mathematica
`eval`	*script*	evaluate a Mathematica script and print the result
`get`	*name script*	evaluate a Mathematica script and get the named Mathematica variable as a Tcl string
`send`	*script*	evaluate a Mathematica script and suppress the output
`start`		start a new Mathematica session
`status`		return the status of the Tcl/Mathematica connection (0 means connected, -1 means not initialized, and 1 means error)

To initiate a connection to a Matlab and Mathematica process, use

```
matlab start
mathematica start
```

To generate a simple plot of a straight line in Matlab and Mathematica, use

```
matlab send { plot([0 1 2 3])}
mathematica send { Plot[x, {x, 0, 3} ] }
```

The `send` command suppresses the output normally returned by interacting with the program using the command interface. The `eval` command, on the other hand, returns the dialog with the console interface:

```
mathematica eval { Plot[x, {x, 0, 3}] }
-Graphics-
```

To terminate the connection, use

```
matlab end
mathematica end
```

One can work with matrices as Tcl lists or in Matlab format. To create a new Matlab matrix x that has two rows and three columns:

```
matlab set x 2 3 "1 2 3 4 5 6" "1 1 1 1 1 1"
```

We can retrieve this Matlab matrix in the same format:

```
matlab get x
2 3 {1.0 2.0 3.0 4.0 5.0 6.0} {1.0 1.0 1.0 1.0 1.0 1.0}
```

We can also retrieve the matrix elements as a Tcl list of complex numbers in an ordered-pair format:

```
matlab getpairs x
(1.0,1.0) (2.0,1.0) (3.0,1.0) (4.0,1.0) (5.0,1.0) (6.0,1.0)
```

Now, matrices can be manipulated in both Tcl and Matlab.

Javier Contreras contributed the following example that creates a Tcl list, sends it to MATLAB as a 2x2 matrix, calculates the inverse in MATLAB and retrieves it back to Tcl as list and/or pairs.

```
ptcl> matlab start
ptcl> set a 1
1
ptcl> set b 2
2
ptcl> set c 3
3
ptcl> set d 4
4
ptcl> set e [expr "{$a $b $c $d}"]
1 2 3 4
ptcl> set f [expr "{$a $b $c $d}"]
1 2 3 4
ptcl> matlab set matrix $b $b $e $f
ptcl> matlab eval {matrix(1,1)}
>>
ans =

1.0000 + 1.0000i


ptcl> set inv_matrix [matlab get inverse {inverse = inv(matrix)}]
2 2 {-1.0 0.5 0.75 -0.25} {1.0 -0.5 -0.75 0.25}
ptcl> set inv_matrix [matlab getpairs inverse {inverse =
inv(matrix)}]
(-1.0,1.0) (0.5,-0.5) (0.75,-0.75) (-0.25,0.25)
ptcl> set new $inv_matrix
```

```
(-1.0,1.0) (0.5,-0.5) (0.75,-0.75) (-0.25,0.25)
ptcl> lindex $new 0
(-1.0,1.0)
ptcl> matlab unset matrix
ptcl> matlab eval {matrix(1,1)}
ptcl> matlab end
```

For other examples of the use of the matlab and mathematica Ptcl commands, see “Using Matlab and Mathematica to Compute Parameters” on page 2-18. These commands support the Matlab and Mathematica consoles in Tycho.

3.10 Limitations of the interpreter

There should be many more commands returning information on the simulation, to permit better exploitation of the full power of the Tcl language.

3.11 A wormhole example

Here is an example of a simulation that contains both an SDF portion and a DE portion. In this example, a Poisson process where particles have value 0.0 is sent into an SDF wormhole, where Gaussian noise is added to the samples. This demo shows how easy it is to use the SDF stars to perform computation on DE particles. The overall delay of the SDF wormhole is zero, so the result is simply Poisson arrivals of Gaussian noise samples.

A `Wormhole` has an *outer* domain and an *inner* domain. The outer domain is determined by the current domain at the time the user starts the `defgalaxy` command to create the wormhole. The inner domain is determined by the `domain` command that appears inside the galaxy definition.

```
reset
# create the wormhole
domain DE
defgalaxy wormBody {
      domain SDF
      star add Add; numports add input 2
      star IIDGaussian1 IIDGaussian
      alias out add output
      alias in add "input#1"
      connect IIDGaussian1 output add "input#2"
}
# Creating the main universe.
domain DE
star wormBody1 wormBody
star Poisson1 Poisson; star graf XMgraph
numports graf input 2
setstate graf title "Noisy Poisson Process"
setstate graf options "-P -0 original -1 noisy"
node node1
nodeconnect Poisson1 output node1
nodeconnect wormBody1 in node1
nodeconnect graf "input#1" node1
```

```
connect wormBody1 out graf "input#2"
run 40
wrapup
```

3.12 Some hints on advanced uses of ptcl with pigi

Although we have not had time to pursue it aggressively in this release, flexible control of Ptolemy simulations (e.g. executing a simulation many times with different parameter settings) is now possible. This can be done by using `ptcl` and `pigi` together.

Warning: This mechanism is still under development, so please note that what is described in this section is likely to change.

3.12.1 Ptcl as a simulation control language for pigi

If you start `pigi` with the `-console` option, then a console window will appear that will accept `ptcl` commands. To experiment with this, open the `sinMod` demo in the SDF basic demo palette, and execute the `pigi` command `compile-facet` (in the `Exec` submenu). This command reads the oct facet from disk, and constructs the Ptolemy data structures to represent it in memory. In your console window, you should see the prompt:

```
pigi>
```

Note what happens if you ask for the name of the current universe:

```
pigi> curuniverse
sinMod
pigi>
```

By compiling the facet, you have created a universe called `sinMod`, and made it the current universe. If you just started `pigi`, then this is one of only two universes in existence:

```
pigi> univlist
main sinMod
pigi>
```

The universe `main` is the default, empty universe that Ptolemy starts with. To verify the contents of the `sinMod` universe, use the `print` command:

```
pigi> print
GALAXY: sinMod
Descriptor: An interpreted galaxy
Contained blocks: singen2 modulator1 XMgraph.input=11
pigi>
```

You can execute this universe from the console window:

```
pigi> run 400
pigi> wrapup
```

Notice that you will not see any output until you invoke the wrapup command, since the `XMgraph` star creates the output plot in its wrapup method.

So far, you have not done anything you could not have done more directly using `pigi`. However, you can change the value of parameters from `ptcl`. To do this, you must first determine the name of the instance of the star or galaxy with the parameter you want to control. Place the mouse over the `singen` icon in the `sinMod` galaxy, and issue the pigi `show-name` (‘n’) command. Most likely, the name will be `singen2`, although it could be different on successive runs. This is an instance name generated automatically by `pigi`. Notice that it is the name shown by the print command above. Also, use the `edit-params` (‘e’) command over the `singen` icon to determine that `singen2` has a parameter named `frequency` with value `PI/100`. Now try the following commands:

```
pigi> setstate singen2 frequency PI/50
pigi> run 400
pigi> wrapup
```

Notice that the frequency of the modulating sinusoid is now twice as high as before.

Much more interestingly, you can now construct a series of runs using Tcl as a scripting language:

```
pigi> foreach i {0.25 0.5 0.75 1 1.25 1.5} {
pigi? setstate singen2 frequency $i*PI/100
pigi? setstate XMgraph.input=11 title \
pigi? "message frequency = [expr 0.01*$i]*PI"
pigi? run 400
pigi? wrapup
pigi? }
pigi>
```

This will invoke six runs, each with a different frequency parameter for the `singen` galaxy `singen2`. The `foreach` command is a standard Tcl command. Notice that in the third and fourth lines, we have also set the title parameter of the `XMgraph` star. This is advisable because otherwise it might be very difficult to tell which result corresponded to which run. Notice that the name of the `XMgraph` instance is “`XMgraph.input=11`”. It is a more complicated name because the icon is specialized to have only a single input port.

Using the full power of the Tcl language, the above mechanism can become extremely powerful. To use its full power, however, you will most likely want to construct your Tcl scripts in files. These files can even include the universe definition, as explained below, so you can create scripts that can be run under `ptcl` only, independent of `pigi`.

3.12.2 The pigi log file pigiLog.pt

In each `pigi` session, a log file named `pigiLog.pt` is generated in the user’s home directory. Every time an oct facet that represents a Ptolemy galaxy or universe is compiled, for example when running a simulation, the equivalent `ptcl` commands building the galaxy or universe are logged in pigiLog.pt. For example, if you followed the above procedure, opening the `sinMod` demo and issuing the `compile-facet` command, your `pigiLog.pt` file will contain something like the following:

```
reset
domain SDF
defgalaxy singen {
      domain SDF
      newstate sample_rate FLOAT "2*PI"
      newstate frequency FLOAT "PI/50"
      newstate phase_in_radians float 0.0
      star Ramp1 Ramp
```

```
        setstate Ramp1 step "2*PI*frequency/sample_rate"
        setstate Ramp1 value phase_in_radians
        star Sin1 Sin
        connect Ramp1 output Sin1 input
        alias out Sin1 output
    }
    defgalaxy modulator {
        domain SDF
        newstate freq FLOAT 0.062832
        star "Mpy.input=21" Mpy
        numports "Mpy.input=21" input 2
        star singen1 singen
        setstate singen1 sample_rate "2*PI"
        setstate singen1 frequency freq
        setstate singen1 phase_in_radians 0.0
        alias in "Mpy.input=21" "input#1"
        alias out "Mpy.input=21" output
        connect singen1 out "Mpy.input=21" "input#2"
    }
    newuniverse sinMod SDF
    target default-SDF
        targetparam logFile ""
        targetparam loopScheduler NO
        targetparam schedulePeriod 10000.0
        star singen2 singen
        setstate singen2 sample_rate "2*PI"
        setstate singen2 frequency "PI/100"
        setstate singen2 phase_in_radians 0.0
        star modulator1 modulator
        setstate modulator freq "0.2*PI"
        star "XMgraph.input=11" XMgraph
        numports "XMgraph.input=11" input 1
        setstate "XMgraph.input=11" title "A modulator demo"
        setstate "XMgraph.input=11" saveFile ""
        setstate "XMgraph.input=11" options "=800x400+0+0 -0 x"
        setstate "XMgraph.input=11" ignore 0
        setstate "XMgraph.input=11" xUnits 1.0
        setstate "XMgraph.input=11" xInit 0.0
        connect singen2 out modulator1 in
        connect modulator1 out "XMgraph.input=11" "input#1"
```

This is a `ptcl` definition of a universe that is equivalent to the oct facet. In normal usage, you may need to edit this file considerably to extract the portions you need, because all the galaxies and universes compiled in a `pigi` session are logged in the same log file. Also, as of this writing, the file does not necessarily get flushed after your compile-facet command completes, so the last few lines may not appear until more lines are written to the file, or you exit `pigi`.

Note that `pigi` compiles the sub-galaxies recursively before compiling the top-level universe. Therefore, the `ptcl` definitions are generated and logged in this recursive order. For instance, in the pigiLog.pt shown above, `ptcl` definitions of the `singen` and `modulator` galaxies appear before that of the `sinMod` universe. Also, if a galaxy has been compiled before, and thus is on the knownlist, its `ptcl` definition will not be generated and logged again when it is used in another universe.

One use of the `ptcl` definitions obtained from pigiLog.pt is to submit bug reports. It is the best way to describe in ASCII text the Ptolemy universe that causes problems.

3.12.3 Using pigiLog.pt to build scripts

If you restart `pigi`, run the `sinMod` demo in the SDF basic demo palette once, then quit `pigi`, then your `pigiLog.pt` file will be as above. Make a copy of `pigiLog.pt` and name it, say, `sinMod.pl`.

To run this simulation with different message waveform frequencies, you may do the following in `ptcl`, analogous to the above commands in `pigi`:

```
# build the sinMod universe
source sinMod.pl
foreach i {0.25 0.5 0.75 1 1.25 1.5} {
      # set parameter values
      setstate singen2 frequency $i*PI/100
      setstate XMgraph.input=11 title \
      "message frequency = [expr 0.01*$i]*PI"
      # execute it
      run 400
      wrapup
}
```

The combination of `ptcl` and `pigi` is very powerful. The above are just some hints on how they can be used together.

3.12.4 oct2ptcl

Kennard White’s program `oct2ptcl` can be used to convert Ptolemy facets to ptcl code. `Oct2ptcl` is not part of the default distribution, and it is not built automatically. You can find the `oct2ptcl` sources in the other.src tar file in `ptolemy/src/octtools/tkoct/oct2ptcl`. `oct2ptcl` is not formally part of Ptolemy, but some developers may find it useful.