@

# Introduction to SWAT:
# The Scheme Widget Application Toolkit

Edition 1.1
November 1996

by Hal Abelson, Stephen Adams, Natalya Cohen and Jim Miller

This is Edition 1.1, last updated 12 November 1996, of *Introduction to Swat*

# 1 Introduction to Swat

Swat (Scheme Widget Application Toolkit) is a Scheme package for creating and controlling interactive graphical objects (called widgets). Examples of widgets are buttons, menus, and graphical and text objects with mouse-sensitive regions. Swat includes a rich collection of predefined widgets, because it incorporates Tk, an extensive graphical toolkit developed by John Ousterhout at UC Berkeley [ref??]. Ousterhout designed Tk to be used in conjunction with his string-based interpreted language Tcl. Swat controls Tk widgets from Scheme rather than Tcl, and the Swat commands for controlling Tk-derived widgets are in general straightforward reflections of the corresponding Tcl commands. Swat also provides facilities for creating new widgets (independently of Tk or Tcl) and for combining widgets to make composite objects.

Both Swat and Tk are built on top of the X Window System [ref??]. We intend (someday) to make a more general interface that can handle widgets in both X and Microsoft Windows (and perhaps others) but we haven't done this yet. The current version of Swat runs only in MIT Scheme, although we intend to port it to Scheme-to-C, and perhaps other Scheme implementations.

This introduction focuses on using Swat with the basic built-in widgets appropriated from Tk. The Swat Reference Manual [not written yet] provides information on more extensive use of the built-in widgets and on Swat's facilities for defining new widgets.

*Acknowledgements:* We are grateful to John Ousterhout and UC Berkeley for creating and distributing Tk and Tcl as publically accessible, unencumbered software so that we and others can build upon his work. Joel Bartlett of Digital Equipment Corporation provided helpful advice on the design of Swat. The EZD graphics system that Joel implemented for his Scheme-to-C system [ref??] also served as the initial model for Tk's canvas widget. Swat's model of boxes and glue (and the code that implements it) was provided by Bert Halstead, who developed it for his Stk system [ref??] at Digital's Cambridge Research Lab. Andy Payne, also of DEC CRL, gave us valuable help on Tcl. We're grateful to Erik Gallesio of the University of Nice for sharing with us the implementation of his STk system [ref??] (no relation to Halstead's) which, like Swat, provides a Scheme interface to Tk. And we would especially like to thank Chris Hanson for help with debugging and for enhancing MIT Scheme's multithreading and event processing facilities so we could use these in Swat.

## 1.1 Loading Swat

This pre-release of Swat is available only at MIT. To load Swat, type

```
(load-option 'swat).
```

Once you have Swat loaded you can type

```
(load "/usr/local/lib/mit-scheme/swat/pole-zero")
(define x (make-pole-zero))
```

to run the poles and zeros demonstration program described at the end of this document.

# 2 Working with simple widgets

Widgets are objects that have a graphical representation on the screen. There are various kinds of widgets: buttons, scales, scrollbars, and more complex widgets such as canvases and menus. Widgets may also initiate various actions in response to events. For example, a button can activate a designated Scheme procedure whenever you push it (i.e., whenever you move the mouse cursor over the button and press a mouse button).

This chapter introduces Swat by describing some of the simplest built-in widgets and the means for combining them to make more complex objects.

## 2.1 Buttons

Here's an example of how to create a button:

```
(define (button-demo)
  (let ((b (make-button
            `(-background yellow -foreground blue
              -activebackground red -text "Push me"))))
    (set-callback! b (lambda () (write-line "Ouch!")))
    (swat-open b)))
```

`Button-demo` begins by making a button with the command `make-button`, which takes as argument a list of *configuration options*. In this example, the button will have a yellow background that turns red when the mouse enters it (activebackground). The foreground (lettering) will be blue, and the label on the button will be "Push me". The Swat Reference Manual contains a complete list of applicable options for each type of widget.

Once we've made the button, we use `set-callback!` to assign a procedure to be activated when the button is pressed. For a button, the callback is a procedure of no arguments. In this demo the button will write "Ouch!" in the Scheme interaction buffer whenever it is pressed.

Finally, having made and configured the button, we make it appear on the screen with the `swat-open` command. This creates a new top-level window on the screen with the button in it. (The `swat-open` procedure returns as its value the symbol `OK`.) Notice that we didn't specify a size for the window. When Swat creates a window, it chooses the window's size to fit what is supposed to go in the window. In this case, the window size is determined by size of the text we specified for the button. If you enlarge the window manually with the mouse, you'll see the button expand to fill the window. You can resize the window with the mouse, but you cannot shrink it below the size at which it was initially created.

## 2.2 Configuring widgets incrementally

When we built the button in the example above, we specified the configuration options at creation time.

You can also configure widgets incrementally, as illustrated by the following alternative version of `button-demo`:

```
(define (button-demo)
  (let ((b (make-button)))
    (ask-widget b `(configure -foreground blue))
    (ask-widget b `(configure -background yellow -activebackground red))
    (ask-widget b `(configure -text "Push me"))
    (set-callback! b (lambda () (write-line "Ouch!")))
    (swat-open b)))
```

Here we've created the button without any specified information, and later used `ask-widget` to configure the button bit by bit. The `ask-widget` procedure issues commands to widgets. Here we've used the `configure` command. In general, each type of widget has its own set of applicable commands—these are listed in the Swat Reference Manual.

Configuring widgets in this way permits us to *change* widget attributes. Here's an example that shows how a button can be used to change the attributes of another button:

```
(define (two-buttons)
  (let ((b1 (make-button
             '(-background yellow -foreground blue
               -activebackground red -text "Push me")))
        (b2 (make-button
             '(-background yellow -foreground blue
               -activebackground red -text "Break it"))))
    (set-callback! b1 (lambda () (write-line "Ouch!")))
    (set-callback! b2
                   (lambda ()
                     (set-callback! b1 (lambda () '()))
                     (ask-widget b1 '(configure -text "Broken"))
                     (add-event-handler!
                      b1
                      "<Enter>"
                      (lambda () (write-line "go away")))))
    (swat-open b1 b2)))
```

Here two buttons appear, each in its own window. Button 1 works just as before—until you press button 2. Pressing button 2 changes button 1's text to read "Broken" and changes button 1's callback so that it no longer does anything when you press it. Note, from the final line of code, that `swat-open` can take multiple arguments. Each object will be opened in a separate top-level window.

Pressing button 2 also adds an *event handler* to button 1 so that, whenever the mouse enters button 1, it prints "go away" in the Scheme buffer. The `add-event-handler!` procedure arranges for a Scheme procedure to be activated upon various kinds of events associated with a widget—mouse entering, mouse leaving, particular button presses, key presses, and so on. The Swat Reference Manual gives a complete list of available triggering events.

## 2.3 Closing windows

You can get rid of widgets like buttons by using the mouse to close the window in which they appear.

Bear in mind, though, that as long as Scheme has a name for the object (referenced from the global environment) then that name will still refer to something, even if it refers to a widget whose window has been closed. For example, you create a button as follows:

```
(define b
  (make-button '(-background yellow -foreground blue
                 -activebackground red -text "Push me")))
```

Now you open `b` and then close the window with the mouse. The name `b` will still be bound to a widget, but you can't do much with it. For example, you can't open it again.[1]

[1] If you want to have a window that "comes and goes," you should be using `map` and `unmap`, not repeatedly trying to open and close the windows.

As an alternative to closing a window manually with the mouse, you can use the `swat-close` command. So in the example above, you could evaluate `(swat-close b)` to close the window, which will have exactly the same effect as closing it with the mouse.

## 2.4 Building on primitive widgets

Even with buttons as the only primitive widget, there are a lot of interactive objects you can create with Swat, because Swat is embedded in Scheme. The following procedure, for instance, creates an *n-position switch*. This is a button that cycles through a number of different states. In each state the button shows a different label and has a different associated procedure. Pressing the button moves the switch to the next state and activates the procedure for that state.

```
(define (make-switch color states)
  ;;states is a list ((text command) (text command))
  (let ((n (length states))
        (button (make-button))
        (current-state 0))
    (define (switch-to-state i)
      ((cadr (list-ref states current-state))) ;run the current command
      (set! current-state i)
      (ask-widget button
                  `(configure -text ,(car (list-ref states i)))))
    (ask-widget button `(configure -background ,color
                                   -text ,(car (car states))))
    (set-callback!
     button
     (lambda () (switch-to-state (modulo (+ current-state 1) n))))
    button))
```

The following demo constructs a three-position switch that cycles through printing 1, 2, and 3:

```
(define (switch-demo)
  (swat-open
   (make-switch "yellow"
                `(("One" ,(lambda () (write-line 1)))
                  ("Two" ,(lambda () (write-line 2)))
                  ("Three" ,(lambda () (write-line 3)))))
   '-title "switch"))
```

In a more realistic example, the action procedures might set the value of a variable, or switch a program object among a number of different modes.

Notice that `switch-demo` uses the `title` option for `swat-open` to make the window appear with the title `switch`. In general, if you follow one of the objects in the list of arguments to open by the symbol `-title`, followed by a string, then the object will be opened in a window whose title is the string. If you don't specify a title, the title of the window will be `SWAT`.

## 2.5 Combining widgets in h-boxes and v-boxes

In addition to giving you access to built-in widgets, Swat also provides means of combination for arranging widgets into more complex objects. An *h-box*, for example, is a horizontal row of objects; a *v-box* is a vertical column.

Here's a variation of the `two-buttons` procedure given above. The only difference is in the last line. Instead of opening the two buttons individually, we arrange the buttons in a vbox and open the vbox.

```
(define (vbox-demo)
  (let ((b1 (make-button
             `(-background yellow -foreground blue
               -activebackground red -text "Push me")))
        (b2 (make-button
             `(-background yellow -foreground blue
               -activebackground red -text "Break it"))))
    (set-callback! b1
                   (lambda () (write-line "Ouch!")))
    (set-callback! b2
                   (lambda ()
                     (set-callback! b1 (lambda () '()))
                     (ask-widget b1 '(configure -text "Broken"))
                     (add-event-handler!
                      b1
                      "<Enter>"
                      (lambda () (write-line "go away")))))
    (swat-open (make-vbox b1 b2) '-title "v-box")))
```

Now we get a single top-level window with the buttons arranged vertically. Note that the two buttons have the same width, and that this equality is maintained as you resize the window.

Boxes can be nested. (In 6.001 parlance, we'd say that Swat objects are *closed* under the means of combination provided by boxes.) The following example displays an hbox—on the left is a vbox of two buttons and on the right is a single button.

```
(define nested-boxes
  (swat-open
   (make-hbox (make-vbox (make-button '(-text "Button 1"))
                         (make-button '(-text "Button 2")))
              (make-button '(-text "Button 3")))))
```

In this example, button 3 will be as high as button 1 plus button 2. Swat includes an elaborate protocol for defining *glue* that determines how objects resize themselves to fill available space on the screen. The fact that buttons stretch to fill out the rows and columns of boxes is determined by the button widget's default glue behavior. The Swat Reference Manual describes how you can change the default glue behavior of widgets if you wish.

You can also add new elements to an existing box using the `add-child!` procedure. The following example creates the same object as `nested-boxes`; then it adds a new button to the first column:

```
(define column1
  (make-vbox (make-button '(-text "Button 1"))
             (make-button '(-text "Button 2"))))

(define nested-boxes-2
  (swat-open
   (make-hbox column1 (make-button '(-text "Button 3")))))

(add-child! column1 (make-button '(-text "Button 4")))
```

When you add button 4, you'll see buttons 1 and 2 become narrower, so that all three buttons can fit into the first column. The procedure `box-children` takes a box as argument and returns a list of the box's children.

Here's an example that creates a simple menu as a vbox of buttons. The procedure takes as argument a list of two-element lists `(label proc)` that give the label to appear on the button and the no-argument procedure to activate when the button is pressed:

```
(define (make-vbox-menu labels-procs)
  (let ((buttons
         (map (lambda (label-proc)
                (let ((b (make-button
                          `(-background yellow -foreground blue
                            -activebackground red
                            -text ,(car label-proc)))))
                  (set-callback! b (cadr label-proc))
                  b))
              labels-procs)))
    (swat-open (apply make-vbox buttons) '-title "menu")))

(define sample-vbox-menu
  (make-vbox-menu
   `(("One" ,(lambda () (write-line 1)))
     ("Two" ,(lambda () (write-line 2)))
     ("Three" ,(lambda () (write-line 3))))))
```

This is a very simple kind of menu. Swat also gives access to Tk's elaborate *menu* widget (see Section 3.2 [Menus], page 14), which provides pull-down and tear-off menus and submenus.

## 2.6 Array boxes

*Array boxes* provide a more elaborate control over arranging Swat widgets. Whereas h-boxes and v-boxes can combine objects separately into rows or columns, array boxes align their children widgets both horizontally and vertically.

An array box is created using a procedure `make-array-box`, which takes an unlimited number of arguments. Each argument is a list of objects which is to comprise a single row, and the number of items in a list specifies the number of columns in the box. Rows are listed top to bottom, and columns left to right. Here is a simple example of a two-by-two array box:

```
(define (array-box-demo)
  (let ((b1 (make-button '(-text "B1")))
        (b2 (make-button '(-text "B2")))
        (b3 (make-button '(-text "B3")))
        (b4 (make-button '(-text "B4"))))
    (swat-open (make-array-box (list b1 b2) (list b3 b4))
               '-title "array")))
```

This produces a box with `b1` and `b2` on the top, and `b3` and `b4` on the bottom. In this particular case, the same arrangement could have been achieved using simple h-boxes and v-boxes:

```
(define (array-box-simulation)
  (let ((b1 (make-button '(-text "B1")))
        (b2 (make-button '(-text "B2")))
        (b3 (make-button '(-text "B3")))
        (b4 (make-button '(-text "B4"))))
    (swat-open (make-vbox (make-hbox b1 b2)
                          (make-hbox b3 b4))
     '-title "fake-array")))
```

However, consider what would happen if we changed the width of one of the buttons, say `b2`, by making the text string longer. In `array-box-simulation`, `b1` and `b2` would fight for horizontal space independently of the other h-box, and since the widths of the buttons would differ, the columns would become misaligned. In an actual array box, on the other hand, the sizes of individual widgets interact so that columns and rows are aligned. Thus, the array box will adjust the width of its columns to make sure the alignment is kept intact. This is the main property of an array box. A more complicated example makes use of alignment in both directions, horizontal and vertical:

```
(define (array-box-demo2)
  (let ((b1 (make-button '(-text "B1")))
        (b2 (make-button '(-text "Button2" -height 3)))
        (b3 (make-button '(-text "Long Button3" -height 2)))
        (b4 (make-button '(-text "B4"))))
    (swat-open (make-array-box (list b1 b2) (list b3 b4))
               '-title "array2")))
```

In this case, the top row is taller than the bottom one (by virtue of `b2`'s height), and the left column is wider. Of course, array boxes don't have to be square, and they can have any number of rows and columns.

There are some interesting variations on the theme of packing objects into array boxes. For example, you can arrange for a child to occupy more than one consecutive location (or cell) in a row or column. The following test procedure creates an array box with three rows, the middle of which is occupied by a single button:

```
(define (array-box-variation)
  (let ((b1 (make-button '(-text "top left")))
        (b2 (make-button '(-text "top right")))
        (b3 (make-button '(-text "middle")))
        (b4 (make-button '(-text "bottom left")))
        (b5 (make-button '(-text "bottom right"))))
    (swat-open (make-array-box (list b1 b2)
                               (list b3 'left)
                               (list b4 b5))
               '-title "funky array")))
```

The symbol `left` is special: it indicates that the current cell should contain the widget that already occupies the cell to its left. Similarly, `up` can be used to refer to the cell above the current one. Another special symbol is `skip`; the cell specified by it will be skipped—it will not contain a widget at all. These special symbols, along with more extensive examples of their usage, are discussed in more detail in the Swat Reference Manual.

## 2.7 Entries

So far, all our examples have been based on button widgets. It's time to introduce a new widget. An *entry*, created with the `make-entry` procedure, is an area into which you can type a line of text.

Here's a simple example:

```
(define (entry-demo)
  (let ((clear-button (make-button `(-text clear)))
        (ok-button (make-button `(-text OK)))
        (e (make-entry `(-width 30))))
    (set-callback! clear-button
                   (lambda () (ask-widget e `(delete 0 end))))
    (set-callback! ok-button
                   (lambda ()
                     (let ((color (ask-widget e '(get))))
                       (ask-widget e `(configure -background ,color)))))
    (swat-open (make-vbox e (make-hbox clear-button ok-button)))))
```

If you type the name of a color into the entry and press `OK`, the entry will change to that color. Pressing the clear button clears the text in the entry. Notice the commands used here: `get` returns the contents of the entry (as a string); `delete` deletes text from the entry in a specified range. Swat provides access to Tk's elaborate set of commands for manipulating the contents of an entry. See the Swat manual for details.

When you type into an entry, you can use appropriate Emacs keys to edit what you type. For instance, rather than pressing `clear`, you can clear the entry by typing `CTRL-A CTRL-K`.

## 2.8 Active variables

The `entry-demo` above shows how you can make widgets interact by invoking appropriate Scheme procedures. In the example, the OK button has Scheme read the entry contents using `ask-widget` and then sets the entry color appropriately. But you can also arrange for widgets to interact in a more direct fashion by using *active variables*.[2]

An active variable is an object that has a value that can be read or set from Scheme. In addition, an active variable can be "linked" to attributes of widgets, such as the contents of an entry. When the variable is set the entry contents change automatically; and when the entry contents change, the value of the variable changes automatically.

For instance, we can link an active variable to the contents of an entry, or the label of a button, by declaring it as the widget's `textvariable` as in the following example:

```
(define (linked-entry-button)
  (let ((v (make-active-variable))
        (e (make-entry '(-width 30)))
        (b (make-button '(-width 30 -text "change me"))))
    (ask-widget b `(configure -textvariable ,v))
    (ask-widget e `(configure -textvariable ,v))
    (swat-open e '-title "Type Here" b '-title "Look Here")))
```

Here the button and the entry are both linked to the same active variable. As you type into the entry, the label on the button changes to match what you type.

You create an active variable using the procedure `make-active-variable`. You read and set active variables from Scheme using `active-variable-value` and `set-active-variable!`. In addition, you can set a callback for an active variable—a Scheme procedure of no arguments, that will be run whenever the variable's value changes.

Here's an example: a v-box of two entries. If you type a number into the top entry, the contents of the bottom entry will automatically be set to twice the value you type:

```
(define (linked-entries)
  (let ((v1 (make-active-variable))
        (v2 (make-active-variable)))
    (let ((e1 (make-entry `(-width 30 -background white
                            -textvariable ,v1)))
          (e2  (make-entry `(-width 30 -background yellow
                             -textvariable ,v2))))
      (set-callback!
       v1
       (lambda ()
         (let ((val (string->number (active-variable-value v1))))
           (if val (set-active-variable! v2 (* 2 val))))))
      (swat-open (make-vbox e1 e2)))))
```

The `linked-entries` example illustrates an important (and annoying!) aspect of programming with active variables: the values of active variables are always *strings*. You can use any value when you call `set-active-variable!` because Swat automatically converts the value to a string. But regardless of the value you set, the value returned by `active-variable-value` is always a string.

[2] Tk calls these simply "variables." The more appropriate name "active variable" was suggested by Michel Beaudouin-Lafon.

This is why, in the example above, the callback on `v1` explicitly converts the value of `v1` to a number before multiplying it by 2; but it does not explicitly convert the result to a string before setting `v2` to that value.

Active variables are powerful features. For instance, in the example above, you could produce an infinite loop by adding a callback on `v2` that changes the value of `v1`. Caveat hacker.

## 2.9 Recovering from errors

Swat allows you to run an interactive application in a window while still maintaining a standard Scheme read-eval-print (REP) loop. It does this by using MIT Scheme's thread package to process all of the graphical interaction concurrently with the REP loop. Bugs in programs written using Swat, therefore, tend to occur in this separate thread. When an error occurs in a thread that does not belong to the REP loop, the Scheme error system stops the thread and forces the REP loop thread to enter an error loop, with an error message such as

```
;The thread #[thread 1] signalled an error:
Unbound variable: gobble
```

This permits you to enter the debugger to analyze the error and make whatever changes you feel are appropriate. If the error can be corrected and the system can proceed, you can use the debugger commands `j`, `k`, or `z` or the `RESTART` procedure. Scheme will present a prompt to the effect that it is

```
restarting other thread; discard events in queue?
```

It is important that you respond with `no` to this question: these events are what allows Swat to proceed. If they are discarded, the event processing will become confused, and you will have to restart Scheme. Similarly, if you encounter a bug in your program that causes the `uitk-thread` to enter an error or loop and it can *not* be proceeded, you will have to kill Scheme and restart it.

# 3 Working with complex widgets

In addition to the simple widgets such as buttons and entries, Swat provides an interface to Tk's elaborate canvas, menu, and text widgets. Each of these is effectively a programming system in itself, with an extensive assortment of subwidgets and options. This manual gives only a brief introduction and a few examples.

## 3.1 Canvases

A canvas is a general-purpose surface that contains graphical *canvas items* such as rectangles, circles, text. Just as with widgets, you can add event handlers to canvas items to make them respond to mouse clicks, key presses, and so on.

In this section we'll illustrate some of these features by building up a canvas, bit by bit.

Start by creating a canvas:

```
(define c (make-canvas '(-width 400 -height 300 -background "light gray")))
(swat-open c)
```

As with the other widgets, the `make-canvas` command permits you to specify various configuration options. Here we've specified the width, the height, and the background color. The size specified here is the size with which the canvas is created. You can make the canvas larger by expanding the window with the mouse.

Now we'll add an item to the canvas. This will be a "pac-man" shape, formed by a 300-degree arc that starts at 0 degrees. The shape will be pink, and outlined in black curve of width 2. The bounding box for the arc will have its upper left corner at (200, 200) and its lower right corner at (250, 250):

```
(define george
  (make-arc-on-canvas c 200 200 250 250
                      `(-fill pink -outline black
                        -width 2 -start 0 -extent 300)))
```

There are similar `make` procedures for making other kinds of canvas items: lines, ovals, rectangles, text, bitmaps, etc. As with other widgets, each kind of item has a set of applicable configure options. Here, for example, we add some text to the canvas at (100, 100).

```
(define message
  (make-text-on-canvas c 100 100 '(-text "Hello there")))
```

Now we add event handlers to George to make him turn yellow whenever the mouse passes over him, and turn pink whenever the mouse leaves:

```
(add-event-handler! george
  "<Enter>"
  (lambda () (ask-widget george '(configure -fill yellow))))

(add-event-handler! george
  "<Leave>"
  (lambda () (ask-widget george '(configure -fill pink))))
```

We'll also arrange things so that whenever we press the mouse button over George, he will rotate by 30 degrees. Also, the text of the message will change to "OUCH!" in a larger font. The text will change back when the button is released.

```
(add-event-handler! george
  "<ButtonPress-1>"
  (lambda ()
    (ask-widget
     message
     `(configure -text "OUCH!"
                 -font "-Adobe-Helvetica-Bold-R-Normal--*-240-*"))
    (set! start (modulo (+ start 30) 360))
    (ask-widget george `(configure -start ,start))))

(add-event-handler! george
   "<ButtonRelease-1>"
   (lambda ()
     (ask-widget
      message
      '(configure -text "Hello there"
                  -font "-Adobe-Helvetica-Bold-R-Normal--*-120-*"))))
```

Notice that the rotation is accomplished by keeping a variable `start` that is used as the starting angle of the arc, and incrementing this by 30 on each button press.

Finally, we'll arrange things so that we can drag George around with the mouse. The idea is that we'll trigger on mouse motion when the mouse is over George and button 1 is pressed. We'll read the coordinates of the button press, compute the difference of these coordinates from the previous coordinates, and tell George to move by that amount. We assume that we've saved the last coordinates in `last-x` and `last-y`. The `move` command for a canvas item tells the item to move by a specific delta-x and delta-y.

```
(add-event-handler! george
  "<Button1-Motion>"
  (lambda (x y)
    (ask-widget george `(move ,(- x last-x) ,(- y last-y)))
    (set! last-x x)
    (set! last-y y))
  "%x" "%y")
```

Observe that the procedure triggered by `add-event-handler!` expects two arguments, `x` and `y`. The extra arguments to `add-event-handler!`—strings `%x` and `%y`—specify that, when the procedure is triggered, the inputs will be bound, respectively, to the x and y coordinates of the button-motion event. (There are various input specifiers you can use with `add-event-handler!` to access the coordinates of the event, the time of the event, and various other X-Windows-related information. See the Swat Manual for a complete list.) We also need to change the ButtonPress event handler to initialize `last-x` and `last-y`:

```
(add-event-handler! george
   "<ButtonPress-1>"
   (lambda (x y)
     (ask-widget
      message
      `(configure -text "OUCH!"
                  -font "-Adobe-Helvetica-Bold-R-Normal--*-240-*"))
     (set! start (modulo (+ start 30) 360))
     (ask-widget george `(configure -start ,start))
     (set! last-x x)
     (set! last-y y))
   "%x" "%y")
```

Here's the complete procedure:

```
(define (canvas-demo)
  (let ((c (make-canvas '(-width 400 -height 300 -background "light gray"))))
    (swat-open c)
    (let ((george
           (make-arc-on-canvas
            c 200 200 250 250
            `(-fill pink -outline black -width 2 -start 0 -extent 300)))
          (message (make-text-on-canvas c 100 100 '(-text "Hello there")))
          (last-x #F)
          (last-y #F)
          (start 0))
      (add-event-handler! george "<Enter>"
       (lambda () (ask-widget george '(configure -fill yellow))))
      (add-event-handler! george "<Leave>"
       (lambda () (ask-widget george '(configure -fill pink))))
      (add-event-handler! george "<ButtonPress-1>"
       (lambda (x y)
         (ask-widget message
          `(configure -text "OUCH!"
                      -font "-Adobe-Helvetica-Bold-R-Normal--*-240-*"))
         (set! start (modulo (+ start 30) 360))
         (ask-widget george `(configure -start ,start))
         (set! last-x x)
         (set! last-y y))
       "%x" "%y")
      (add-event-handler! george "<ButtonRelease-1>"
       (lambda ()
         (ask-widget message
          '(configure -text "Hello there"
                      -font "-Adobe-Helvetica-Bold-R-Normal--*-120-*"))))
      (add-event-handler! george "<Button1-Motion>"
       (lambda (x y)
         (ask-widget george `(move ,(- x last-x) ,(- y last-y)))
         (set! last-x x)
         (set! last-y y))
       "%x" "%y")
      )))
```

### 3.1.1 Grouping canvas items

Items on a canvas can be grouped together, with the entire set given a single name. This *canvas group* can then be used as the target of various commands. For example, if `group` is a canvas group, you can fill all of the group items with blue by issuing a single command, `(ask-widget group '(configure -fill blue))`. The procedure `make-canvas-item-group` takes as arguments a canvas and a list of canvas items and creates a canvas group. You can add new items to an existing group using the `add-to-canvas-item-group` command, which takes a group and a canvas item to be added to it. Groups can also be merged together into a new group with `(merge-canvas-item-groups canvas destructive? . groups)` (if `destructive?` is `true` the names for the old groups shouldn't be used after the command is executed).

Another important property of a group of canvas items is that event handlers can be attached to the entire group. This allows you to detect the cursor entering or leaving any one of the items, or a button being pressed or released while over any item, and so forth. In the pole-zero demonstration program (see Chapter 5 [Extended example], page 18) a group of canvas items is used to convert two small lines into a single item for making an "x"-shaped mark which moves as a unit. As another example, consider an addition to the `canvas-demo` procedure above:

```
(let ((group (make-canvas-item-group c (list george message))))
  (add-event-handler! group "<ButtonPress-2>"
   (lambda () (ask-widget group `(configure -fill violetred))))
  (add-event-handler! group "<ButtonRelease-2>"
   (lambda ()
     (ask-widget message `(configure -fill black))
     (ask-widget  george `(configure -fill pink)))))
```

If button 2 is pressed over either of the two canvas items, both will turn violetred. On the other hand, the two items are reconfigured separately on a button release.

### 3.1.2 Canvas with scrollbars

A scrollable canvas appears on the screen as a "viewport" into a larger canvas. You can shift the view of the canvas by moving horizontal and vertical scrollbars. To create a scrollable canvas, use the procedure `make-scrollable-canvas`, as in the following example:

```
(define (canvas-scrollbar-demo)
  (let ((sc (make-scrollable-canvas
             '(-width 500 -height 500
               -scrollregion (0 0 1000 1000)))))
    (swat-open sc)
    (ask-widget (scrollable-canvas-hscroll sc)
                `(configure -foreground "yellow"))
    (ask-widget (scrollable-canvas-vscroll sc)
                `(configure -foreground "yellow"))
    (make-rectangle-on-canvas (scrollable-canvas-canvas sc)
                              100 100 400 400
                              '(-fill red))
    sc))
```

Here the size of the canvas "viewport" on the screen is 500 by 500, but the actual canvas is size 1000 by 1000. (The `-scrollregion` attribute list specifies the left, top, right, and bottom coordinates of the canvas.)

A scrollable canvas is implemented as a vbox whose top row is an hbox—the canvas and the vertical scrollbar—and whose bottom row is the horizontal scrollbar. You can extract these individual pieces using the selectors `scrollable-canvas-canvas`, `scrollable-canvas-hscroll`, and `scrollable-canvas-vscroll`. Notice how the above demo example extracts the canvas to place a rectangle on it, and extracts the scrollbars to configure their colors.

## 3.2 Menus

Menus are used to select from among a set of choices. In Swat, you make menus (using `make-menu`) and add to them various kinds of buttons (using `(add-to-menu menu kind-of-entry . options)`) and sub-menus (using `(add-sub-menu menu sub-menu . options)`. Menus respond to various commands. The most important of these are the `post` and `unpost` commands. In response to a `post` command (which has two arguments: an `x` and `y` position) the menu appears on the screen at the given coordinates. In response to an `unpost` message the menu disappears. There are also special *menubuttons* created with `make-menubutton` that can be used to conveniently create pop-up menus. When pressed a menubutton sends a `post` message to its associated menu, calculating a "reasonable" value for the `x` and `y` positions based on the position of the menubutton. Menus and menubuttons have a variety of configuration options (including `background`, `label`, and so forth). See the Tk Manual for details.

There are five kinds of items that may appear in a menu. Four of these are created using `add-to-menu`: `'CHECKBUTTON`, `'COMMAND`, `'RADIOBUTTON`, and `'SEPARATOR`. Sub-menus (called "cascades" in the Tk documentation) are created with `add-sub-menu`. See the Tk manual for more information.

Here is a simple program that creates an application with one initial button, labelled "Go!." When pressed, the button posts a menu containing three items: a button labelled "Close," another button labelled "Two," and a sub-menu labelled "More..." When the mouse passes over the More button, the sub-menu (with "A," "B," "C," and "D" buttons) appears. The buttons have callbacks that produce output when they are pushed.

```
(define (menu-test)
  (let ((application (make-application "Menu Test")))
    (define start-button (make-button '(-text "Go!")))
    (define main (make-menu))
    (define sub (make-menu))
    (define button-1 (add-to-menu main 'command '-background "blue" '-label "Close"))
    (define button-2 (add-to-menu main 'command '-background "yellow" '-label "Two"))
    (define button-3 (add-sub-menu main sub '-background "blue" '-label "More..."))
    (define sub-1 (add-to-menu sub 'command '-background "blue" '-label "A"))
    (define sub-2 (add-to-menu sub 'command '-background "blue" '-label "B"))
    (define sub-3 (add-to-menu sub 'command '-background "blue" '-label "C"))
    (define sub-4 (add-to-menu sub 'command '-background "blue" '-label "D"))
    (ask-widget start-button
                '(configure -background yellow
                            -font
                            "-adobe-helvetica-bold-r-normal--34-240-100-100-p-182-iso8859-1"))
    (add-child! application start-button)
    (add-child! start-button main)
    (set-callback! button-1 (lambda () (ask-widget main '(unpost))))
    (set-callback! button-2 (lambda () (write-line 'Two)))
    (set-callback! button-3 (lambda () (write-line 'Three)))
    (set-callback! sub-1 (lambda () (write-line 'A)))
    (set-callback! sub-2 (lambda () (write-line 'B)))
    (set-callback! sub-3 (lambda () (write-line 'C)))
    (set-callback! sub-4 (lambda () (write-line 'D)))
    (set-callback! start-button (lambda () (ask-widget main '(post 10 20))))
    'OK))
```

## 3.3 Text

Swat supports text objects based on the Tk widgets of the same name. These are similar to the `entry` objects (see Section 2.7 [Entries], page 8), but a text object is much more elaborate. It has a buffer into which text can be entered; there are keyboard and programmatic ways to edit the text; the text can be read under program control; the attributes (color, font, highlighting) of characters in the buffer can be manipulated; and the cursor and mouse buttons can be detected and handled over regions of characters.

Text objects are created using `make-text`. They have an incredible variety of configuration options (see the Tk manual). Text objects support an extensive set of editting options, either from the keyboard or via Scheme `configure` commands. In addition, a text widget can have *marks* indicating specific points (between characters) in the buffer; the marks move as text is inserted or deleted from the buffer (Swat currently has no direct support for marks).

Text objects can also have *text tags* to mark ranges of characters. Once tagged, the group of characters can be treated as a unit for changing the font, highlighting, color, etc.—much as the canvas group inside a canvas. You can also associate Tk events with tagged characters, so that you can receive an alert when the mouse enters or exists a tagged group, or when the button is clicked, and so forth. They are also used in managing X window selections (cut and paste operations). A tag is created using `(make-text-tag text-widget index1 . index2)`, where `index1` and `index2` specify the characters at the beginning and the end of the character range. Different ways to specify the indices and other details are found in the Tk documentation.

As a convenience, it is also possible to make a text area with a vertical scrollbar in a single operation, by using `make-scrollable-text` instead of `make-text`. The behavior of a scrollable text is similar to that of a scrollable canvas. In this case, however, the scroll region cannot be specified; it is extended automatically as the text buffer increases in length. As with the scrollable canvas, there are selectors, `scrollable-text-text` and `scrollable-text-vscroll`, used to configure the text and the scrollbar widgets separately.

# 4 Beyond this introduction

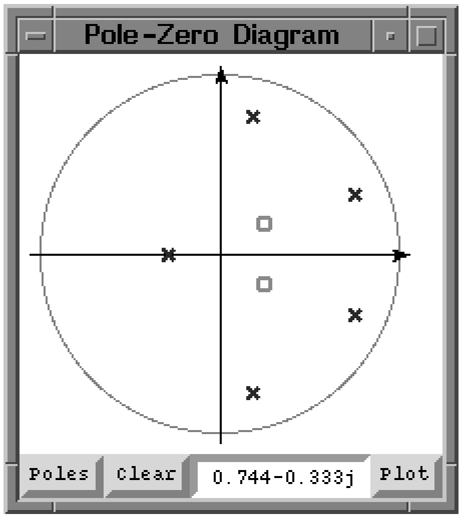
This section isn't written yet. It will be an overview of the following ideas. But the details of these are not yet frozen.

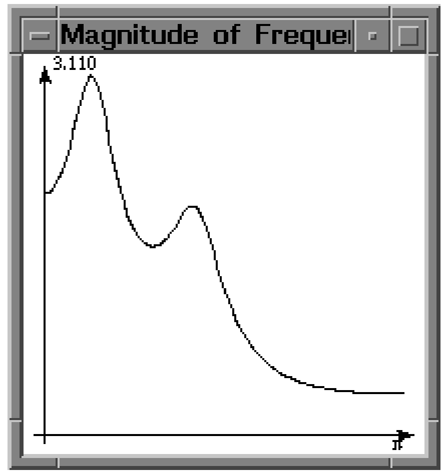
glue

making your own widgets

multiple applications for multiple display connections

# 5 Extended example: A digital filter plotter




This chapter describes an extended application using Swat—an interactive pole-zero plotter for digital filters. The program demonstrates many of the features discussed in this introduction. It also uses a few features that we haven't yet seen, such as the `on-death!` notification, and the `after-delay` mechanism.

For readers unfamiliar with digital filters, it's enough to know that a filter can be described by two sets of complex numbers, the *zeros* and the *poles*. The zeros and poles determine a function called the *frequency response*, defined for each point $z$ on the unit circle. Its value is the ratio $P/Q$, where $P$ is the product of the difference between $z$ and each of the zeros, and $Q$ is the product of the difference between $z$ and each of the poles:

$$F(z) = \frac{\prod_i z - z_i}{\prod_j z - p_j}$$

The frequency response is normally plotted as a function of a real number *w* versus the value of the frequency response at the point on the unit circle *z=exp(iw)*, where w is between 0 and *pi*.

For a so-called "filter with real coefficients" the zeros and poles come in conjugate pairs. That is, if $p$ is a pole, then either $p$ is a real number (i.e., it lies on the real axis in the complex plane), or else *p=a+bi* and its complex conjugate *a-bi* are both poles. Similarly, a zero $z$ is either real, or else $z$ and its complex conjugate are both zeros.

The pole-zero program displays a *diagram* window that shows the unit circle in the complex plane. By double clicking with the mouse, the user can add new zeros and poles, as specified by a switch below the diagram. Poles appear on the diagram as crosses (X's), and zeros appear as circles (O's). When the switch reads "Zeros," clicking on the diagram adds new zeros; when the switch reads "Poles," clicking on the diagram adds new poles. Clicking on the switch itself flips the switch between zeros and poles. Zeros and poles will always appear in conjugate pairs. Namely, if you click very close to the real axis, then a single zero or pole will appear. Otherwise, a zero (pole) will appear at the position you indicate, and another one will appear at the complex conjugate position.

If you (single) click on a zero or pole, you can drag it around with the mouse. If you move a zero or pole that is part of a conjugate pair, the other zero or pole will move simultaneously, so that the two elements of the pair are always complex conjugate values. As you move a zero or pole around, a *label* under the diagram displays the changing (real or complex number) coordinates.

There is also a *plot* button under the diagram. Pressing this makes a *graph* window appear, with a plot *w* of the magnitude of the frequency response at *z=exp(iw)*, for the given poles and zeros. Once the graph window is exposed, the frequency response will be automatically recomputed, and the plot updated, whenever poles and zeros are added or moved.

Finally, the entire pole-zero diagram is a message-receiving "object" that accepts the commands `add-zero` and `add-pole`, so that the diagram can be controlled by programs as well as by direct manipulation.

The remainder of this chapter is a commented listing of the Scheme program that implements all of this.[1]

## 5.1 Some constants and variables

The program uses the following global variables. In order to make this exposition simpler, we have defined these as global variables, which means that there can be only one pole-zero diagram at a time. It would be straightforward to restructure the program to maintain local state information for separate pole-zero diagram objects.

```
(define half-window-size 200)    ;size for diagram and plot windows
(define zero-size 5)
(define pole-size 4)
(define trim 10)                 ;size of border on windows
(define zero-color "violetred")
(define pole-color "blue")
(define canvas-color "white")
(define number-of-points 100)    ;number of points to plot
(define max-w  3.14159)          ;plotting range goes from 0 to pi
(define delayed-plot-interval 2) ;how long to wait before
                                 ;automatically generating a new plot

(define text-font "CourR12")
(define symbol-font "symb10")    ;font includes numbers and pi

(define location 'later)         ; active variable for coords
(define all-zeros '())           ; alist of zeros(objects)/location
(define all-poles '())           ; alist of poles(objects)/location

(define time-to-update-plot? #F) ;flag for automatic replotting
(define diagram-canvas #F)
(define graph-canvas #F)
(define shape-size #F)           ;size of shapes being drawn
                                 ;(zeros or poles)

(define single-maker #F)         ;drawing procedure
(define pair-maker #F)           ;drawing procedure
```

[1] This introduction is all you need to know about digital filters in order to understand this chapter, which is concerned with the Swat programming to create the interaction described here. The program was implemented by Natalya Cohen, based upon a somewhat simpler Tk demonstration program by Michel Beaudouin-Lafon.

## 5.2 The main routine

The pole-zero diagram is created by calling `make-pole-zero`. This creates the diagram window and displays it on the screen so that you can interact with it as described above. The result returned by `make-pole-zero` is a message-accepting object, which permits you to modify the same diagram under program control. For example:

```
(define pz (make-pole-zero))
((pz 'add-zero) .1+.3i)
(pz 'plot-response)
```

Of course, you can also interact with `pz` directly with the mouse. The principle that each important operation on a direct-manipulation object should also be accessible under program control is an important guideline for user interface design.

Here's the `make-pole-zero` routine. It creates and displays the diagram, then returns the message acceptor.

```
(define (make-pole-zero)
  (initialize-pole-zero-variables)
  (build-diagram-window)          ;create buttons and window
  (initialize-diagram)            ;draw initial diagram
  (switch-to-zeros)
  (lambda (message)
    (case message
      ((graph-canvas) graph-canvas)
      ((diagram-canvas) diagram-canvas)
      ((plot-response) (plot-response))
      ((add-zero)
       (lambda (z)
         (let ((xy (z->canvas-coords z)))
           (if (= (imag-part z) 0)
               (make-single-zero (car xy))
               (make-zero-pair (car xy) (cadr xy))))))
      ((add-pole)
       (lambda (p)
         (let ((xy (z->canvas-coords p)))
           (if (= (imag-part p) 0)
               (make-single-pole (car xy))
               (make-pole-pair (car xy) (cadr xy))))))
      (else "Unknown message -- MAKE-POLE-ZERO" message))))
```

Initializing variables is straightforward. Notice that `location` is initialized to an active variable whose initial value is the empty string.

```
(define (initialize-pole-zero-variables)
  (set! all-zeros '())
  (set! all-poles '())
  (set! diagram-canvas
        (make-canvas `(-width ,(* 2 half-window-size)
                       -height ,(* 2 half-window-size))))
  (set! graph-canvas #F)
  (set! location (make-active-variable))
  (set-active-variable! location ""))
```

The pole-zero diagram itself is a vbox consisting of a canvas called `diagram-canvas` above an hbox containing three buttons and a label widget. The `clear-button` will delete all the poles and zeros, the `make-button` is a switch that flips between creating zeros and creating poles, the `plot` button plots the frequency response, and the `coords-display` label will display the complex number coordinates of the current zero or pole (which will be the value of the active variable `location`).

The `build-diagram-window` procedure sets all this up, adding the appropriate callbacks to the buttons.

```
(define (build-diagram-window)
  (let ((maker-button
         (make-switch "yellow"
                      `(("Zeros" ,switch-to-zeros)
                        ("Poles" ,switch-to-poles))))
        (clear-button (make-button '(-text "Clear")))
        (coords-display (make-label))
        (plot-button (make-button '(-text "Plot"))))
    (ask-widget coords-display
                `(configure -width 13 -background ,canvas-color
                            -relief sunken -textvariable ,location
                            -font ,text-font))
    (for-each
     (lambda (b) (ask-widget b `(configure -background "yellow" -font ,text-font)))
     (list maker-button clear-button plot-button))
    (set-callback! clear-button clear-button-action)
    (set-callback! plot-button plot-button-action)

    (ask-widget diagram-canvas `(configure -background ,canvas-color))
    (on-death! diagram-canvas 'little-brother-canvas
               (lambda () (if graph-canvas (swat-close graph-canvas))))

    (swat-open
     (make-vbox diagram-canvas
                (make-hbox maker-button clear-button
                           coords-display plot-button))
     '-title "Pole-Zero Diagram")))
```

Observe the `on-death!` alert that is placed on the diagram canvas. This feature of Swat (which we haven't seen before) arranges for a no-argument procedure to be run when a widget is destroyed. In this case, the procedure arranges that, when the diagram canvas is destroyed (e.g., by running `swat-close` or closing the window with the mouse) the associated graph canvas window will also be closed. The symbol `little-brother-canvas` is a *reason* included as a comment on the notification.[2]

The action procedures for the `maker-button` switch set the procedures for making single poles/zeros and conjugate pole/zero pairs to poles or zeros, accordingly. They also set the shape size.

```
(define (switch-to-zeros)
  (set! shape-size zero-size)
  (set! single-maker make-single-zero)
  (set! pair-maker make-zero-pair))

(define (switch-to-poles)
  (set! shape-size pole-size)
  (set! single-maker make-single-pole)
  (set! pair-maker make-pole-pair))
```

The action placed on `clear` button resets the lists of poles and zeros, deletes these from the canvas, clears the `location` variable, and clears the graph canvas if necessary:

---

[2] Swat provides operations for manipulating alerts, based on reasons, but we aren't yet advertising the methods for doing this. For now, you're stuck with just having to include them without making much use of them.

```
(define (clear-button-action)
  (for-each (lambda (entry) (ask-widget (car entry) '(delete)))
           (append all-zeros all-poles))
  (if graph-canvas
      (begin (ask-widget graph-canvas '(delete all))
             (draw-axes graph-canvas)))
  (set-active-variable! location "")
  (set! all-zeros '())
  (set! all-poles '()))
```

The `plot` button creates the graph canvas (if necessary) and draws the plot. Notice the `on-death!` alert placed on the graph canvas, which, if the graph window is closed, sets a flag to inform the diagram window.

```
(define (plot-button-action)
  (if (not graph-canvas)
      (create-graph-window))
  (plot-response))

(define (create-graph-window)
  (set! graph-canvas
        (make-canvas `(-width ,(* 2 half-window-size)
                      -height ,(* 2 half-window-size))))
  (ask-widget graph-canvas `(configure -background ,canvas-color))
  (swat-open graph-canvas '-title "Magnitude of Frequency Response")
  (on-death! graph-canvas 'big-brother-canvas
             (lambda () (set! graph-canvas #F))))
```

### 5.2.1 Initializing the diagram

To actually draw the pole-zero diagram, we draw two axes and a circle to represent the unit circle in the complex plane. Notice the `arrow` configure option that is used to add arrows to the axes, which are created with the canvas-item maker `make-line-on-canvas` (which takes the canvas, and the starting and ending $x$ and $y$ coordinates as arguments). The circle is created with `make-oval-on-canvas`, which takes the canvas and the $x$ and $y$ coordinates of the upper left and lower right of the bounding box for the oval to be drawn.

```
(define (initialize-diagram)
  (let ((x-axis
         (make-line-on-canvas
          diagram-canvas
          trim half-window-size
          (- (* 2 half-window-size) trim) half-window-size))
        (y-axis
         (make-line-on-canvas
          diagram-canvas
          half-window-size trim
          half-window-size (- (* 2 half-window-size) trim)))
        (unit-circle
         (make-oval-on-canvas
          diagram-canvas
          (* 2 trim) (* 2 trim)
          (* 2 (- half-window-size trim))
          (* 2 (- half-window-size trim)))))
    (ask-widget x-axis '(configure -arrow last))
    (ask-widget y-axis '(configure -arrow first))
    (ask-widget unit-circle '(configure -outline gray))
    (add-event-handler!
     diagram-canvas "<Double-ButtonPress-1>"
     (lambda (x y)
       (set! time-to-update-plot? #F)
       (if (< (abs (- y half-window-size)) shape-size)
           (single-maker x)
           (pair-maker x y)))
     "%x" "%y")))
```

We add an event handler to the canvas, that triggers when button 1 is pressed twice (`Double-ButtonPress-1`). This checks to see whether the indicated location is close to the $x$-axis (i.e., whether to draw a single pole/zero or a conjugate pair of poles/zeros) and calls the appropriate procedure. The flag `time-to-update-plot` is used to control whether to automatically redraw the frequency response. (See the discussion of `maybe-update-plot` below.)

## 5.3 Creating zeros and poles

Poles and zeros are either created singly (when they are on the real axis), or else created in complex-conjugate pairs that move together. To make a single zero at the mouse-click location, we make a zero (a circle) at the location, set the variable `location` to the corresponding value on the real axis, add this to the list of zeros, install the motion handler, and, after a sufficient delay, replot the frequency response:

```
(define (make-single-zero x)
  (let ((z (make-zero (list x half-window-size))))
    (set-active-variable!
     location
     (real->string
      (canvas-coords->z (list x half-window-size))))
    (move-by-itself z)
    (maybe-update-plot)))
```

```
(define (make-zero xy)
  (let* ((x (car xy))
         (y (cadr xy))
         (zero
          (make-oval-on-canvas diagram-canvas
                               (- x zero-size) (- y zero-size)
                               (+ x zero-size) (+ y zero-size))))
    (set! all-zeros (cons (cons zero (canvas-coords->z xy)) all-zeros))
    (ask-widget zero `(configure -outline ,zero-color -fill ,canvas-color -width 2))
    zero))
```

The motion handler installs events triggered by pressing and releasing button 1 and for mouse motion while button 1 is depressed.

```
(define (move-by-itself obj)
  (let ((last-x 'later))
    (define (keep-track-of-coords x)
      (set! last-x x)           ;save previous position
      (let ((z (canvas-coords->z (list last-x half-window-size))))
        (set-active-variable! location (real->string z))))
    (add-event-handler! obj "<ButtonPress-1>"
     (lambda (x)
       (set! time-to-update-plot? #F)
       (keep-track-of-coords x))
     "%x")
    (add-event-handler! obj "<ButtonRelease-1>"
     (lambda ()
       (store-object-coords obj last-x)
       (maybe-update-plot)))
    (add-event-handler! obj "<Button1-Motion>"
     (lambda (x)        ;move by dx = x - previous x, dy = 0
       (ask-widget obj `(move ,(- x last-x) 0))
       (keep-track-of-coords x))
     "%x")))
```

The subprocedure `store-object-coords` updates the coordinate value associated with the given zero or pole in the association lists `all-zeros` and `all-poles`.

```
(define (store-object-coords obj last-x)
  (let ((entry
         (let ((zero (assq obj all-zeros)))
           (if zero zero (assq obj all-poles))))
        (z (real-part (canvas-coords->z (list last-x 0)))))
    (set-cdr! entry z)))
```

Poles are dealt with almost the same as zeros. The main difference to notice here is that a pole—which is an “x” shape—is represented as a *canvas item group* consisting of two lines.

```
(define (make-single-pole x)
  (let ((p (make-pole (list x half-window-size))))
    (set-active-variable!
     location
     (real->string
      (canvas-coords->z (list x half-window-size))))
    (move-by-itself p)
    (maybe-update-plot)))
```

```
(define (make-pole xy)
  (let* ((x (car xy))
         (y (cadr xy))
         (line1
          (make-line-on-canvas diagram-canvas
                               (- x pole-size) (- y pole-size)
                               (+ x pole-size) (+ y pole-size)))
         (line2
          (make-line-on-canvas diagram-canvas
                               (- x pole-size) (+ y pole-size)
                               (+ x pole-size) (- y pole-size)))
         (pole (make-canvas-item-group diagram-canvas (list line1 line2))))
    (set! all-poles (cons (cons pole (canvas-coords->z xy))
                          all-poles))
    (ask-widget pole `(configure -fill ,pole-color -width 2))
    pole))
```

### 5.3.1 Creating conjugate pairs

Clicking on the canvas away from the real axis creates a conjugate pair of zeros or poles. The procedures that create these are similar to the ones that create single zeros and poles. The major difference is that the motion handler procedure `move-with-conjugate-pair` arranges that when one object in the pair moves by *dx* and *dy*, the other object moves by *dx* and -*dy*.

```
(define (make-zero-pair x y)
  (let* ((zero (canvas-coords->z (list x y)))
         (other-pos (z->canvas-coords (conjugate zero)))
         (z1 (make-zero (list x y)))
         (z2 (make-zero other-pos)))
    (set-active-variable!
     location
     (complex->string
      (canvas-coords->z (list x y))))
    (move-with-conjugate-pair z1 z2)
    (move-with-conjugate-pair z2 z1)
    (maybe-update-plot)))


(define (make-pole-pair x y)
  (let* ((pole (canvas-coords->z (list x y)))
         (other-pos (z->canvas-coords (conjugate pole)))
         (p1 (make-pole (list x y)))
         (p2 (make-pole other-pos)))
    (set-active-variable!
     location
     (complex->string
      (canvas-coords->z (list x y))))
    (move-with-conjugate-pair p1 p2)
    (move-with-conjugate-pair p2 p1)
    (maybe-update-plot)))
```

```
(define (move-with-conjugate-pair obj1 obj2)
  (let ((last-x 'later)
        (last-y 'later))
    (define (keep-track-of-coords x y)
      (set! last-x x)
      (set! last-y y)
      (let ((z (canvas-coords->z (list last-x last-y))))
        (set-active-variable! location (complex->string z))))
    (add-event-handler! obj1 "<ButtonPress-1>"
      (lambda (x y)
        (set! time-to-update-plot? #F)
        (keep-track-of-coords x y))
      "%x" "%y")
    (add-event-handler! obj1 "<ButtonRelease-1>"
     (lambda ()
       (store-conjugate-pair-coords obj1 obj2 last-x last-y)
       (maybe-update-plot)))
    (add-event-handler! obj1 "<Button1-Motion>"
     (lambda (x y)
       (ask-widget obj1 `(move ,(- x last-x) ,(- y last-y)))
       (ask-widget obj2 `(move ,(- x last-x) ,(- last-y y)))
       (keep-track-of-coords x y))
     "%x" "%y")))

(define (store-conjugate-pair-coords obj1 obj2 last-x last-y)
  (let* ((zero-entry (assq obj1 all-zeros))
         (obj1-entry
          (if zero-entry zero-entry (assq obj1 all-poles)))
         (obj2-entry
          (if zero-entry
              (assq obj2 all-zeros)
              (assq obj2 all-poles)))
         (z (canvas-coords->z (list last-x last-y))))
    (set-cdr! obj1-entry z)
    (set-cdr! obj2-entry (conjugate z))))
```

## 5.4 Plotting the frequency response

`Plot-response` draws a plot (assuming that the graph window has been created).

```
(define (plot-response)
  (if graph-canvas
      (begin (ask-widget graph-canvas '(delete all))
             (draw-axes graph-canvas)
             (plot-magnitude graph-canvas))))
```

The graph axes are created as two lines on the canvas, and the *pi* symbol (to show the scale on the $x$-axis) is a text item. The `anchor e` option to `configure` specifies that the location given for the text should correspond the the rightmost ("east") point on the text.

```
(define (draw-axes graph-canvas)
  (let ((x-axis (make-line-on-canvas
                 graph-canvas
                 trim (* 2 (- half-window-size trim))
                 (- (* 2 half-window-size) trim)
                 (* 2 (- half-window-size trim))))
        (y-axis (make-line-on-canvas
                 graph-canvas
                 (* 2 trim) trim
                 (* 2 trim) (- (* 2 half-window-size) trim)))
        (pi (make-text-on-canvas
             graph-canvas
             (* 2 (- half-window-size trim)) (- (* 2 half-window-size) trim)
             '(-text "p"))))
    (ask-widget x-axis '(configure -arrow last))
    (ask-widget y-axis '(configure -arrow first))
    (ask-widget pi `(configure -anchor e -font ,symbol-font))))
```

Having drawn the axes, we now construct a "graph" of the magnitude of the frequency response, namely, a list of pairs *(x, f(x))* as *x* runs from zero to pi. The value of the corresponding *f(x)* is the magnitude of the frequency response at the point *exp(ix)*, which is a point on the unit circle.

```
(define (plot-magnitude graph-canvas)
  (let ((zero-locations (map cdr all-zeros))
        (pole-locations (map cdr all-poles)))
    (let ((fcn
           (lambda (x)
             (let ((jw (exp (* x +i))))
               (let ((numer
                      (apply * (map (lambda (z) (magnitude (- jw z)))
                                    zero-locations)))
                     (denom
                      (apply * (map (lambda (z) (magnitude (- jw z)))
                                    pole-locations))))
                 (if (< denom 1.e-10) 1.e5 (/ numer denom)))))))
      (plot-graph-on-canvas
       graph-canvas
       (let loop ((index 0) (points '()))
         (if (> index number-of-points)
             points
             (let ((w (* index (/ max-w number-of-points))))
               (loop (+ index 1)
                     (cons (cons w (fcn w)) points)))))))))
```

Once we have the graph (list of pairs) we plot it by drawing lines between consecutive points, with the *y*-axis autoscaled to the maximum value of the function. We also display the maximum value as a text item near the top of the *y*-axis. (Note the `anchor sw` (southwest) configure option.)

```
(define (plot-graph-on-canvas canvas graph)
  (let* ((maxval (apply max (map cdr graph)))
         (canvas-points
          (map (lambda (graph-point)
                 (magnitude-coords->canvas-coords graph-point maxval))
               graph)))
    (let loop ((rest-points (cdr canvas-points))
               (this-point (car canvas-points)))
      (if (null? rest-points)
          'done
          (let ((next-point (car rest-points)))
            (make-line-on-canvas canvas
                                 (car this-point)
                                 (cdr this-point)
                                 (car next-point)
                                 (cdr next-point))
            (loop (cdr rest-points) (car rest-points)))))
    (let ((maxval-display
           (make-text-on-canvas
            canvas (* 3 trim) (* 2 trim) `(-text ,(real->string maxval)))))
      (ask-widget maxval-display
                  `(configure -anchor sw -font ,symbol-font)))))
```

### 5.4.1 Delayed plotting

The procedure `maybe-update-plot`, which is called after poles and zeros are created or moved, arranges for the frequency response to be plotted after a delay of `delayed-plot-interval` seconds, assuming that no one has set the flag `time-to-update-plot` false in the meantime. Scheduling such delayed actions is accomplished in Swat with the procedure `after-delay`, which takes a time interval (in seconds) and a procedure of no arguments to be run after that delay. In this case, the action procedure checks the flag and calls `plot-response` if the flag is still true.

```
(define (maybe-update-plot)
  (set! time-to-update-plot? #T)
  (after-delay delayed-plot-interval
   (lambda ()
     (if time-to-update-plot?
         (plot-response)))))
```

The pole-zero procedures above set the flag to false and call `maybe-update-plot` when they are done. The effect is that the plot will be redrawn if the specified interval of time goes by without any more poles or zeros being created or moved.

## 5.5 Miscellaneous utilities

This completes the pole-zero program, except for a few miscellaneous utilities.

The following procedures convert between complex numbers and pixel coordinates on the pole-zero canvas and on the magnitude plot canvas:

```
(define (z->canvas-coords z)
  (let ((x (real-part z))
        (y (imag-part z)))
    (list
     (round->exact
      (+ (* x (- half-window-size (* 2 trim)))
         half-window-size))
     (round->exact
      (+ (* y (- (* 2 trim) half-window-size))
         half-window-size)))))


(define (canvas-coords->z xy)
  (let ((x (exact->inexact (car xy)))
        (y (exact->inexact (cadr xy))))
    (let ((real (/ (- x half-window-size)
                   (- half-window-size (* 2 trim))))
          (imag (/ (- y half-window-size)
                   (- (* 2 trim) half-window-size))))
      (+ real (* imag +i)))))


(define (magnitude-coords->canvas-coords xy max-mag)
  (let ((x (car xy))
        (y (cdr xy)))
    (cons (round->exact
           (+ (* x (/ (- (* 2 half-window-size) (* 4 trim)) max-w))
              (* 2 trim)))
          (round->exact
           (+ (* y (/ (- (* 4 trim) (* 2 half-window-size)) max-mag))
              (* 2 (- half-window-size trim)))))))
```

Finally, there are routines that convert real and complex numbers to strings (with three decimal places).

```
(define (complex->string z)
  (let* ((r (real-part z))
         (i (imag-part z))
         (rs (unsigned->string (abs r)))
         (is (unsigned->string (abs i)))
         (signed-r
          (if (< r 0)
              (string-append "-" rs)
              rs))
         (signed-i
          (if (< i 0)
              (string-append "-" is)
              (string-append "+" is))))
    (string-append signed-r signed-i "j")))


(define (real->string r)
  (let* ((rs (unsigned->string (abs r)))
         (signed-r
          (if (< r 0)
              (string-append "-" rs)
              rs)))
    signed-r))
```

```
(define (unsigned->string n)
  (let* ((int-part (floor n))
         (frac-part (- n int-part))
         (dec (floor->exact (* frac-part 1000)))
         (string-dec (number->string dec))
         (padded-string-dec
          (cond ((< dec 10) (string-append "00" string-dec))
                ((< dec 100) (string-append "0" string-dec))
                (else string-dec))))
    (string-append (number->string (floor->exact int-part))
                   "."
                   padded-string-dec)))
```

## 5.6 Using the diagram under program control

explain:

```
(define (add-butterworth-poles pole-zero-diagram n cutoff)
  (define pi (* (atan 1 1) 4))
  (define (make-index-list n)
    (define (loop k)
      (if (>= k (/ n 2))
          (list k)
          (cons k (loop (+ k 1)))))
    (loop 1))
  (let ((index-list (make-index-list n))
        (w (exp (/ (* +i pi) (* 2 n)))))
    (for-each (lambda (pole)
                ((pole-zero-diagram 'add-pole) pole))
              (map (lambda (s) (/ (+ 1 (* s cutoff)) (- 1 (* s cutoff))))
                   (map (lambda (k) (expt w (+ n (* 2 k) -1)))
                        index-list))))
  (let loop ((z 1))
    (if (> z n)
        'done
        (begin ((pole-zero-diagram 'add-zero) -1)
               (loop (1+ z)))))
  (pole-zero-diagram 'plot-response))
```

# Index

# Table of Contents