CHAPTER 11

Graphs and charts

There was a time when the term *graphics* included graphs; this chapter reintroduces this meaning. Although graphs, histograms and pie charts may not be appropriate for all applications, they do provide a useful means of conveying a large amount of information to the viewer. Examples will include linegraphs, histograms, and pie charts to support classical graphical formats. More complex graph examples will include threshold alarms and indicators.

11.1 Simple graphs

Let's start by constructing a very simple graph, without trying to make a graph class or adding too many features, so we can see how easy it can be to add a graph to an application. We'll add more functionality later.

simpleplot.py

```
from Tkinter import *
root = Tk()
root.title('Simple Plot - Version 1')
```

```
canvas = Canvas(root, width=450, height=300, bg = 'white')
canvas.pack()

Button(root, text='Quit', command=root.quit).pack()

canvas.create_line(100,250,400,250, width=2)
canvas.create_line(100,250,100,50,  width=2)

for i in range(11):
   x = 100 + (i * 30)
   canvas.create_line(x,250,x,245, width=2)
   canvas.create_text(x,254, text='%d'% (10*i), anchor=N)

for i in range(6):
   y = 250 - (i * 40)
   canvas.create_line(100,y,105,y, width=2)
   canvas.create_text(96,y, text='%5.1f'% (50.*i), anchor=E)

for x,y in [(12, 56), (20, 94), (33, 98), (45, 120), (61, 180),
       (75, 160), (98, 223)]:
   x = 100 + 3*x
   y = 250 - (4*y)/5
   canvas.create_oval(x-6,y-6,x+6,y+6, width=1,
               outline='black', fill='SkyBlue2')

root.mainloop()
```

Draw axes

1

Draw y ticks

Draw data points

Code comments

1 Here we add the ticks and labels for the x-axis. Note that the values used are hard-coded—we have made little provision for reuse!

```
for i in range(11):
     x = 100 + (i * 30)
     canvas.create_line(x,250,x,245, width=2)
     canvas.create_text(x,254, text='%d'% (10*i), anchor=N)
```

Notice how we have set this up to increment *x* in units of 10.

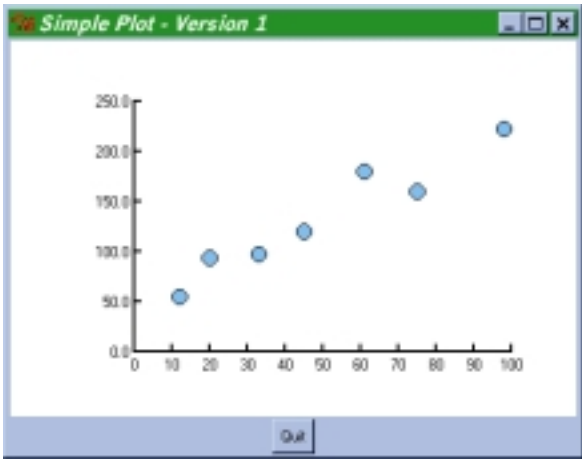


Figure 11.1 Simple two-dimensional graph

This small amount of code produces an effective graph with little effort as you can see in figure 11.1. We can improve this graph easily by adding lines connecting the dots as shown in figure 11.2.

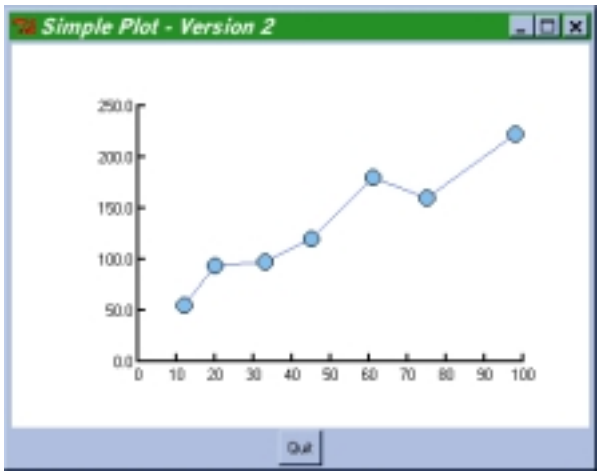


Figure 11.2 Adding lines to a simple graph

simpleplot2.py

```
scaled = []
for x,y in [(12, 56), (20, 94), (33, 98), (45, 120), (61, 180),
       (75, 160), (98, 223)]:
   scaled.append(100 + 3*x, 250 - (4*y)/5)

canvas.create_line(scaled, fill='royalblue')

for x,y in scaled:
   canvas.create_oval(x-6,y-6,x+6,y+6, width=1,
               outline='black', fill='SkyBlue2')
```

❶ ❷ ❸

Code comments

❶ So that we do not have to iterate through the data in a simple loop, we construct a list of x-y coordinates which may be used to construct the line (a list of coordinates may be input to the `create_line` method).

❷ We draw the line first. Remember that items drawn on a canvas are layered so we want the lines to appear under the blobs.

❸ Followed by the blobs.

Here come the Ginsu knives! We can add line smoothing at no extra charge! If we turn on smoothing we get cubic splines for free; this is illustrated in figure 11.3.

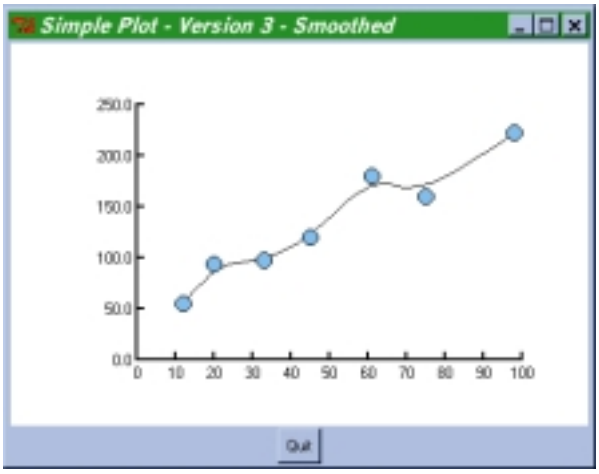


Figure 11.3 Smoothing the line

simpleplot3.py

```
canvas.create_line(scaled, fill='black', smooth=1)
```

I don't think that needs an explanation!

11.2 A graph widget

The previous examples illustrate that it is quite easy to produce simple graphs with a small amount of code. However, when it is necessary to display several graphs on the same axes, it is cumbersome to produce code that will be flexible enough to handle all situations. Some time ago Konrad Hinsen made an effective graph widget available to the Python community. The widget was intended to be used with NumPy.* With his permission, I have adapted it to make it usable with the standard Python distribution and I have extended it to support additional display formats. An example of the output is shown in figure 11.4. In the following code listing, I have removed some repetitive code. You will find the complete source code online.

plot.py

```
from Tkinter import *
from Canvas import Line, CanvasText
import string, math
from utils import *
from math import pi
```

* NumPy is Numeric Python, a specialized collection of additional modules to facilitate numeric computation where performance is needed.

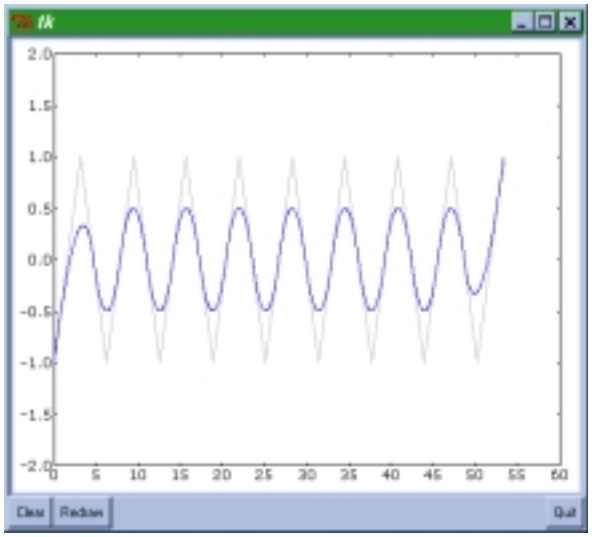
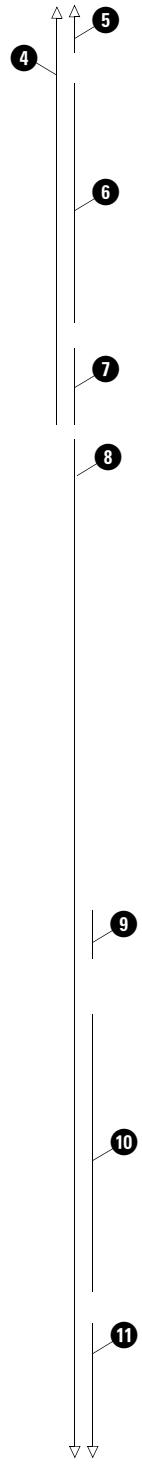


Figure 11.4 Simple graph widget: lines only

```
class GraphPoints:
   def __init__(self, points, attr):
        self.points = points
        self.scaled = self.points
        self.attributes = {}
        for name, value in self._attributes.items():
             try:
                  value = attr[name]
             except KeyError: pass
             self.attributes[name] = value

   def boundingBox(self):
        return minBound(self.points),  maxBound(self.points)

   def fitToScale(self, scale=(1,1), shift=(0,0)):
        self.scaled = []
        for x,y in self.points:
             self.scaled.append((scale[0]*x)+shift[0],\
                         (scale[1]*y)+shift[1])

class GraphLine(GraphPoints):
   def __init__(self, points, **attr):
        GraphPoints.__init__(self, points, attr)

   _attributes = {'color':  'black',
                  'width':        1,
                  'smooth':       0,
                  'splinesteps': 12}

   def draw(self, canvas):
        color  = self.attributes['color']
        width  = self.attributes['width']
        smooth = self.attributes['smooth']
```

```
        steps  = self.attributes['splinesteps']
        arguments = (canvas,)

        if smooth:
             for i in range(len(self.points)):
                  x1, y1 = self.scaled[i]
                  arguments = arguments + (x1, y1)
        else:
             for i in range(len(self.points)-1):
                  x1, y1 = self.scaled[i]
                  x2, y2 = self.scaled[i+1]
                  arguments = arguments + (x1, y1, x2, y2)

        apply(Line, arguments, {'fill': color, 'width': width,
                       'smooth': smooth,
                       'splinesteps': steps})

class GraphSymbols(GraphPoints):
   def __init__(self, points, **attr):
        GraphPoints.__init__(self, points, attr)

   _attributes = {'color': 'black',
                  'width': 1,
                  'fillcolor': 'black',
                  'size': 2,
                  'fillstyle': '',
                  'outline': 'black',
                  'marker': 'circle'}

   def draw(self, canvas):
        color  = self.attributes['color']
        size   = self.attributes['size']
        fillcolor = self.attributes['fillcolor']
        marker  = self.attributes['marker']
        fillstyle = self.attributes['fillstyle']
        self._drawmarkers(canvas, self.scaled, marker, color,
                    fillstyle, fillcolor, size)

   def _drawmarkers(self, c, coords, marker='circle',
             color='black', fillstyle='', fillcolor='',size=2):
        l = []
        f = eval('self._' +marker)
        for xc, yc in coords:
             id = f(c, xc, yc, outline=color, size=size,
                    fill=fillcolor, fillstyle=fillstyle)
             if type(id) is type(()):
                  for item in id: l.append(item)
             else:
                  l.append(id)
        return l

   def _circle(self, c, xc, yc, size=1, fill='',
             outline='black', fillstyle=''):
        id = c.create_oval(xc-0.5, yc-0.5, xc+0.5, yc+0.5,
                     fill=fill, outline=outline,
                     stipple=fillstyle)
```



```
        c.scale(id, xc, yc, size*5, size*5)
        return id

# --- Code Removed ----------------------------------------------------
```

Code comments

❶ The GraphPoints class defines the points and attributes of a single plot. As you will see later, the attributes that are processed by the constructor vary with the type of line style. Note that the `self._attributes` definitions are a requirement for subclasses.

❷ `boundingBox` returns the top-left and bottom-right coordinates by scanning the coordinates in the points data. The convenience functions are in utils.py.

❸ `fitToScale` modifies the coordinates so that they fit within the scale determined for all of the lines in the graph.

```
def fitToScale(self, scale=(1,1), shift=(0,0)):
     self.scaled = []
     for x,y in self.points:
          self.scaled.append((scale[0]*x)+shift[0],\
                             (scale[1]*y)+shift[1])
```

Note that we supply tuples for `scale` and `shift`. The first value is for *x* and the second is for *y*.

❹ The GraphLine class defines methods to draw lines from the available coordinates.

❺ The `draw` method first extracts the appropriate arguments from the attributes dictionary.

❻ Depending on whether we are doing smoothing, we supply start-end-coordinates for line segments (unsmoothed) or a sequence of coordinates (smoothed).

❼ We then `apply` the arguments and the keywords to the canvas `Line` method. Remember that the format of the `Line` arguments is really:

```
Line(*args, **keywords)
```

❽ `GraphSymbols` is similar to `GraphLine`, but it outputs a variety of filled shapes for each of the *x-y* coordinates.

❾ The `draw` method calls the appropriate marker routine through the generic `_drawmarkers` method:

```
self._drawmarkers(canvas, self.scaled, marker, color,
                  fillstyle, fillcolor, size)
```

❿ `_drawmarkers` evaluates the selected marker method, and then it builds a list of the symbols that are created.

```
f = eval('self._' +marker)
for xc, yc in coords:
     id = f(c, xc, yc, outline=color, size=size,
            fill=fillcolor, fillstyle=fillstyle)
```

⓫ I have included just one of the shapes that can be drawn by the graph widget. The full set are in the source code available online.

plot.py (continued)

```
   def _dot(self, c, xc, yc, ... ):
   def _square(self, c, xc, yc, ... ):
   def _triangle(self, c, xc, yc, ... ):
   def _triangle_down(self, c, xc, yc, ... ):
   def _cross(self, c, xc, yc, ... ):
   def _plus(self, c, xc, yc, ... ):

# --- Code Removed ----------------------------------------------------

class GraphObjects:
   def __init__(self, objects):
        self.objects = objects

   def boundingBox(self):
        c1, c2 = self.objects[0].boundingBox()
        for object in self.objects[1:]:
             c1o, c2o = object.boundingBox()
             c1 = minBound([c1, c1o])
             c2 = maxBound([c2, c2o])
        return c1, c2

   def fitToScale(self, scale=(1,1), shift=(0,0)):
        for object in self.objects:
             object.fitToScale(scale, shift)

   def draw(self, canvas):
        for object in self.objects:
             object.draw(canvas)

class GraphBase(Frame):
   def __init__(self, master, width, height,
                background='white', **kw):
        apply(Frame.__init__, (self, master), kw)
        self.canvas = Canvas(self, width=width, height=height,
                             background=background)
        self.canvas.pack(fill=BOTH, expand=YES)
        border_w = self.canvas.winfo_reqwidth() - \
                string.atoi(self.canvas.cget('width'))
        border_h = self.canvas.winfo_reqheight() - \
                string.atoi(self.canvas.cget('height'))
        self.border = (border_w, border_h)
        self.canvas.bind('<Configure>', self.configure)
        self.plotarea_size = [None, None]
        self._setsize()
        self.last_drawn = None
        self.font = ('Verdana', 10)

   def configure(self, event):
        new_width = event.width-self.border[0]
        new_height = event.height-self.border[1]
        width = string.atoi(self.canvas.cget('width'))
        height = string.atoi(self.canvas.cget('height'))
        if new_width == width and new_height == height:
```

12 13 14 15 16

```
            return
        self.canvas.configure(width=new_width, height=new_height)
        self._setsize()
        self.clear()
        self.replot()

    def bind(self, *args):
        apply(self.canvas.bind, args)

    def _setsize(self):
        self.width = string.atoi(self.canvas.cget('width'))
        self.height = string.atoi(self.canvas.cget('height'))
        self.plotarea_size[0] = 0.97 * self.width
        self.plotarea_size[1] = 0.97 * -self.height
        xo = 0.5*(self.width-self.plotarea_size[0])
        yo = self.height-0.5*(self.height+self.plotarea_size[1])
        self.plotarea_origin = (xo, yo)

    def draw(self, graphics, xaxis = None, yaxis = None):
        self.last_drawn = (graphics, xaxis, yaxis)
        p1, p2 = graphics.boundingBox()
        xaxis = self._axisInterval(xaxis, p1[0], p2[0])
        yaxis = self._axisInterval(yaxis, p1[1], p2[1])
        text_width = [0., 0.]
        text_height = [0., 0.]

        if xaxis is not None:
            p1 = xaxis[0], p1[1]
            p2 = xaxis[1], p2[1]
            xticks = self._ticks(xaxis[0], xaxis[1])
            bb = self._textBoundingBox(xticks[0][1])
            text_height[1] = bb[3]-bb[1]
            text_width[0] = 0.5*(bb[2]-bb[0])
            bb = self._textBoundingBox(xticks[-1][1])
            text_width[1] = 0.5*(bb[2]-bb[0])
        else:
            xticks = None
        if yaxis is not None:
            p1 = p1[0], yaxis[0]
            p2 = p2[0], yaxis[1]
            yticks = self._ticks(yaxis[0], yaxis[1])
            for y in yticks:
                bb = self._textBoundingBox(y[1])
                w = bb[2]-bb[0]
                text_width[0] = max(text_width[0], w)
                h = 0.5*(bb[3]-bb[1])
                text_height[0] = h
                text_height[1] = max(text_height[1], h)
        else:
            yticks = None
        text1 = [text_width[0], -text_height[1]]
        text2 = [text_width[1], -text_height[0]]
        scale = ((self.plotarea_size[0]-text1[0]-text2[0]) / \
              (p2[0]-p1[0]),
              (self.plotarea_size[1]-text1[1]-text2[1]) / \
```

```
            (p2[1]-p1[1]))
        shift = ((-p1[0]*scale[0]) + self.plotarea_origin[0] + \
            text1[0],
            (-p1[1]*scale[1]) + self.plotarea_origin[1] + \
            text1[1])
        self._drawAxes(self.canvas, xaxis, yaxis, p1, p2,
                    scale, shift, xticks, yticks)
        graphics.fitToScale(scale, shift)
        graphics.draw(self.canvas)

# --- Code Removed ----------------------------------------------------
```

Code comments (continued)

⓬ The GraphObjects class defines the collection of graph symbologies for each graph. In particular, it is responsible for determining the common bounding box for all of the lines.

⓭ `fitToScale` scales each of the lines to the calculated bounding box.

⓮ Finally, the `draw` method renders each of the graphs in the composite.

⓯ `GraphBase` is the base widget class which contains each of the composites. As you will see later, you may combine different arrangements of graph widgets to produce the desired effect.

⓰ An important feature of this widget is that it redraws whenever the parent container is resized. This allows the user to shrink and grow the display at will. We bind a `configure` event to the `configure` callback.

plot.py (continued)

```
        self.canvas.bind('<Configure>', self.configure)
if __name__ == '__main__':
   root = Tk()
   di = 5.*pi/5.
   data = []

   for i in range(18):                                                    ⓱
        data.append((float(i)*di,
                  (math.sin(float(i)*di)-math.cos(float(i)*di))))
   line  = GraphLine(data, color='gray', smooth=0)
   linea = GraphLine(data, color='blue', smooth=1, splinesteps=500)

   graphObject = GraphObjects([line, linea])                              ⓲

   graph  = GraphBase(root, 500, 400, relief=SUNKEN, border=2)
   graph.pack(side=TOP, fill=BOTH, expand=YES)                            ⓳

   graph.draw(graphObject, 'automatic', 'automatic')

   Button(root, text='Clear',  command=graph.clear).pack(side=LEFT)
   Button(root, text='Redraw', command=graph.replot).pack(side=LEFT)
   Button(root, text='Quit',   command=root.quit).pack(side=RIGHT)

   root.mainloop()
```

Code comments (continued)

⓱ Using the graph widget is quite easy. First, we create the line/curve that we wish to plot:

```
for i in range(18):
     data.append((float(i)*di,
                  (math.sin(float(i)*di)-math.cos(float(i)*di))))
line  = GraphLine(data, color='gray', smooth=0)
linea = GraphLine(data, color='blue', smooth=1, splinesteps=500)
```

⓲ Next we create the `GraphObject` which does the necessary scaling:

```
graphObject = GraphObjects([line, linea])
```

⓳ Finally, we create the graph widget and associate the `GraphObject` with it:

```
graph  = GraphBase(root, 500, 400, relief=SUNKEN, border=2)
graph.pack(side=TOP, fill=BOTH, expand=YES)
graph.draw(graphObject, 'automatic', 'automatic')
```

11.2.1 Adding bargraphs

Having developed the basic graph widget, it is easy to add new types of visuals. Bargraphs, sometimes called *histograms*, are a common way of presenting data, particularly when it is intended to portray the magnitude of the data, since the bars have actual volume as opposed to perceived volume under-the-curve. Figure 11.5 shows some typical bargraphs, in some cases combined with line graphs. Note that it is quite easy to set up multiple instances of the graph widget.

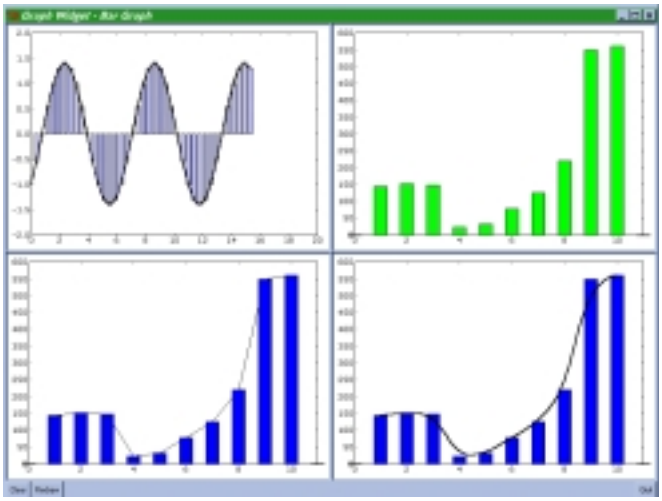


Figure 11.5 Adding bar graphs to the graph widget

plot2.py

```
from Tkinter import *
from Canvas import Line, CanvasText, Rectangle
```

```
class GraphPoints:

# --- Code Removed -------------------------------------------------------

def fitToScale(self, scale=(1,1), shift=(0,0)):
        self.scaled = []
        for x,y in self.points:
            self.scaled.append((scale[0]*x)+shift[0],\
                               (scale[1]*y)+shift[1])
        self.anchor = scale[1]*self.attributes.get('anchor', 0.0)\
                            + shift[1]

# --- Code Removed -------------------------------------------------------

class GraphBars(GraphPoints):
   def __init__(self, points, **attr):
        GraphPoints.__init__(self, points, attr)
        _attributes = {'color': 'black',
                       'width': 1,
                       'fillcolor': 'yellow',
                       'size': 3,
                       'fillstyle': '',
                       'outline': 'black'}

   def draw(self, canvas):
        color  = self.attributes['color']
        width  = self.attributes['width']
        fillstyle = self.attributes['fillstyle']
        outline  = self.attributes['outline']
        spread  = self.attributes['size']
        arguments = (canvas,)
        p1, p2  = self.boundingBox()
        for i in range(len(self.points)):
             x1, y1 = self.scaled[i]
             canvas.create_rectangle(x1-spread, y1, x1+spread,
                            self.anchor, fill=color,
                            width=width, outline=outline,
                            stipple=fillstyle)

# --- Code Removed -------------------------------------------------------

if __name__ == '__main__':
   root = Tk()
   root.title('Graph Widget - Bar Graph')

   di = 5.*pi/40.
   data = []
   for i in range(40):
        data.append((float(i)*di,
                  (math.sin(float(i)*di)-math.cos(float(i)*di))))
   line1  = GraphLine(data, color='black', width=2,
                smooth=1)
   line1a = GraphBars(data[1:], color='blue', fillstyle='gray25',
                anchor=0.0)

   line2 = GraphBars([(0,0),(1,145),(2,151),(3,147),(4,22),(5,31),
```

❶ ❷ ❸ ❹

```
                (6,77),(7,125),(8,220),(9,550),(10,560),(11,0)],
               color='green', size=10)

    line3 = GraphBars([(0,0),(1,145),(2,151),(3,147),(4,22),(5,31),
                (6,77),(7,125),(8,220),(9,550),(10,560),(11,0)],
               color='blue', size=10)
    line3a = GraphLine([(1,145),(2,151),(3,147),(4,22),(5,31),
                 (6,77),(7,125),(8,220),(9,550),(10,560)],
                color='black', width=1, smooth=0)

    line4 = GraphBars([(0,0),(1,145),(2,151),(3,147),(4,22),(5,31),
                (6,77),(7,125),(8,220),(9,550),(10,560),(11,0)],
               color='blue', size=10)
    line4a = GraphLine([(1,145),(2,151),(3,147),(4,22),(5,31),
                 (6,77),(7,125),(8,220),(9,550),(10,560)],
                color='black', width=2, smooth=1)

    graphObject  = GraphObjects([line1a, line1])
    graphObject2 = GraphObjects([line2])
    graphObject3 = GraphObjects([line3a, line3])
    graphObject4 = GraphObjects([line4, line4a])

    f1 = Frame(root)
    f2 = Frame(root)

    graph  = GraphBase(f1, 500, 350, relief=SUNKEN, border=2)
    graph.pack(side=LEFT, fill=BOTH, expand=YES)
    graph.draw(graphObject, 'automatic', 'automatic')

    graph2= GraphBase(f1, 500, 350, relief=SUNKEN, border=2)
    graph2.pack(side=LEFT, fill=BOTH, expand=YES)
    graph2.draw(graphObject2, 'automatic', 'automatic')

    graph3= GraphBase(f2, 500, 350, relief=SUNKEN, border=2)
    graph3.pack(side=LEFT, fill=BOTH, expand=YES)
    graph3.draw(graphObject3, 'automatic', 'automatic')

    graph4= GraphBase(f2, 500, 350, relief=SUNKEN, border=2)
    graph4.pack(side=LEFT, fill=BOTH, expand=YES)
    graph4.draw(graphObject4, 'automatic', 'automatic')

    f1.pack()
    f2.pack()

# --- Code Removed ------------------------------------------------------
```

Code comments

❶ There's not much to explain here; I think that the changes are fairly self-explanatory. However, `anchor` is worthy of a brief note. In the case of the sine/cosine curve, we want the bars to start on zero. This is the anchor value. If we don't set it, we'll draw from the x-axis regardless of its value.

```
self.anchor = scale[1]*self.attributes.get('anchor', 0.0) + shift[1]
```

❷ The bargraph has some slightly different options that need to be set.

❸ The bargraph simply draws a rectangle for the visual.

❹ Defining the data is similar to the method for lines. Note that I have omitted the first data point so that it does not overlay the y-axis:

```
line1a = GraphBars(data[1:], color='blue', fillstyle='gray25',
anchor=0.0)
```

11.2.2 Pie charts

As Emeril Lagasse* would say, "Let's kick it up a notch!" Bargraphs were easy to add, and adding pie charts is not much harder. Pie charts seem to have found a niche in management reports, since they convey certain types of information very well. As you will see in figure 11.6, I have added some small details to add a little extra punch. The first is to scale the pie chart if it is drawn in combination with another graph—this prevents the pie chart from getting in the way of the axes (I do not recommend trying to combine pie charts and bar graphs, however). Secondly, if the height and width of the pie chart are unequal, I add a little decoration to give a three-dimensional effect.

There is a problem with Tk release 8.0/8.1. A stipple is ignored for arc items, if present, when running under Windows; the figure was captured under UNIX. Here are the changes to create pie charts:

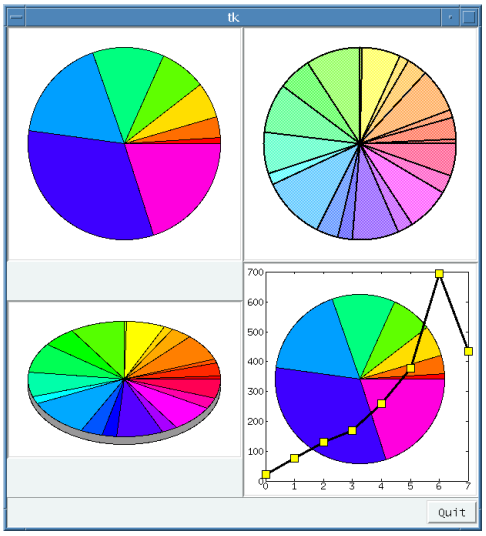


Figure 11.6 Adding pie charts to the graph widget

* Emeril Lagasse is a popular chef/proprietor of restaurants in New Orleans and Las Vegas in the USA. He is the exhuberant host of a regular cable-television cooking show. The audience join Emeril loudly in shouting "Bam! Let's kick it up a notch!" as he adds his own Essence to his creations.

plot3.py

```
# --- Code Removed -------------------------------------------------------

class GraphPie(GraphPoints):
   def __init__(self, points, **attr):
        GraphPoints.__init__(self, points, attr)

        _attributes = {'color': 'black',
                       'width': 1,
                       'fillcolor': 'yellow',
                       'size': 2,
                       'fillstyle': '',
                       'outline': 'black'}

   def draw(self, canvas, multi):
        width  = self.attributes['width']
        fillstyle = self.attributes['fillstyle']
        outline  = self.attributes['outline']
        colors  = Pmw.Color.spectrum(len(self.scaled))
        arguments = (canvas,)

        x1 = string.atoi(canvas.cget('width'))
        y1 = string.atoi(canvas.cget('height'))
        adj = 0
        if multi: adj = 15
        xy  = 25+adj, 25+adj, x1-25-adj, y1-25-adj
        xys = 25+adj, 25+adj+10, x1-25-adj, y1-25-adj+10
        tt = 0.0
        i = 0
        for point in self.points:
             tt = tt + point[1]
        start = 0.0
        if not x1 == y1:
             canvas.create_arc(xys, start=0.0, extent=359.99,
                         fill='gray60', outline=outline,
                         style='pieslice')

        for point in self.points:
             x1, y1 = point
             extent = (y1/tt)*360.0
             canvas.create_arc(xy, start=start, extent=extent,
                         fill=colors[i], width=width,
                         outline=outline, stipple=fillstyle,
                         style='pieslice')
             start = start + extent
             i = i+1

class GraphObjects:
   def __init__(self, objects):
        self.objects  = objects
        self.multiple = len(objects)-1

# --- Code Removed -------------------------------------------------------
```

❶ ❷ ❸ ❹ ❺

```
    def draw(self, canvas):
        for object in self.objects:
            object.draw(canvas, self.multiple)      ❻

# --- Code Removed ----------------------------------------------------

if __name__ == '__main__':
    root = Tk()
    root.title('Graph Widget - Piechart')

    pie1   = GraphPie([(0,21),(1,77),(2,129),(3,169),(4,260),(5,377),
                       (6,695),(7,434)])

    pie2   = GraphPie([(0,5),(1,22),(2,8),(3,45),(4,22),
                       (5,9),(6,40),(7,2),(8,56),(9,34),
                       (10,51),(11,43),(12,12),(13,65),(14,22),
                       (15,15),(16,48),(17,16),(18,45),(19,19),
                       (20,33)], fillstyle='gray50', width=2)

    pie3   = GraphPie([(0,5),(1,22),(2,8),(3,45),(4,22),
                       (5,9),(6,40),(7,2),(8,56),(9,34),
                       (10,51),(11,43),(12,12),(13,65),(14,22),
                       (15,15),(16,48),(17,16),(18,45),(19,19),
                       (20,33)])

    pieline4  = GraphLine([(0,21),(1,77),(2,129),(3,169),(4,260),
                           (5,377),(6,695),(7,434)], width=3)
    pielines4 = GraphSymbols([(0,21),(1,77),(2,129),(3,169),(4,260),
                           (5,377),(6,695),(7,434)],
                           marker='square', fillcolor='yellow')

    graphObject1 = GraphObjects([pie1])
    graphObject2 = GraphObjects([pie2])
    graphObject3 = GraphObjects([pie3])
    graphObject4 = GraphObjects([pie1, pieline4, pielines4])

    f1 = Frame(root)
    f2 = Frame(root)

    graph1= GraphBase(f1, 300, 300, relief=SUNKEN, border=2)
    graph1.pack(side=LEFT, fill=BOTH, expand=YES)
    graph1.draw(graphObject1)

# --- Code Removed ----------------------------------------------------
```

Code comments

❶ The pie chart implementation assigns a spectrum of colors to the slices of the pie, one color value per slice. This gives a reasonable appearance for a small number of slices.

```
colors  = Pmw.Color.spectrum(len(self.scaled))
```

❷ This code adjusts the position of the pie chart for cases where we are displaying the pie chart along with other graphs:

```
adj = 0
if multi: adj = 15
```

```
xy  = 25+adj, 25+adj, x1-25-adj, y1-25-adj
xys = 25+adj, 25+adj+10, x1-25-adj, y1-25-adj+10
```

The shadow disc (`xys`) is used if the pie chart is being displayed as a tilted disc.

❸ The shadow is drawn as a pie slice with an almost complete circular slice:

```
if not x1 == y1:
     canvas.create_arc(xys, start=0.0, extent=359.99,
          fill='gray60', outline=outline, style='pieslice')
```

❹ As in the case of adding bar graphs, adding pie charts requires a specialized `draw` routine.

❺ The scaling factors are determined by the presence of multiple graphs in the same widget.

❻ `self.multiple` is passed down to the graph object's `draw` method.

As you have seen in these examples, adding a new graph type is quite easy and it produces some reasonably attractive graphs. I hope that you can make use of them and perhaps create new visual formats for the Python community.

11.3 3-D graphs

If you have a large amount of data and that data follows a pattern that encourages examining the graphs on the same axes (same scale), there are a number of ways to display the graphs. One way is to produce a series of separate graphs and then present them side by side. This is good if you want to examine the individual graphs in detail, but it does not readily demonstrate the relationship between the graphs. To show the relationship you can produce a single diagram with all of the plots superimposed using different symbols, line styles, or combinations of both. However, there is often a tendency for the lines to become entangled or for symbols to be drawn on top of each other. This can produce very confusing results.

I always like to solve these problems by producing three-dimensional graphs. They allow the viewer to get a sense of the topology of the data as a whole, often highlighting features in the data that may be difficult to discern in other formats. The next example illustrates such a graph (see figure 11.7). I have taken a few shortcuts to reduce the overall amount of code. For example, I have made no provision for modifying the orientation of the axes or the viewing position. I'll leave that as an exercise for the enthusiastic reader!

3dgraph.py

```
from   Tkinter import *
import Pmw, AppShell, math

class Graph3D(AppShell.AppShell):
  usecommandarea = 1
  appname       = '3-Dimensional Graph'
  frameWidth     = 800
  frameHeight    = 650

  def createButtons(self):
       self.buttonAdd('Print',
                   helpMessage='Print current graph (PostScript)',
                   statusMessage='Print graph as PostScript file',
                   command=self.iprint)
```

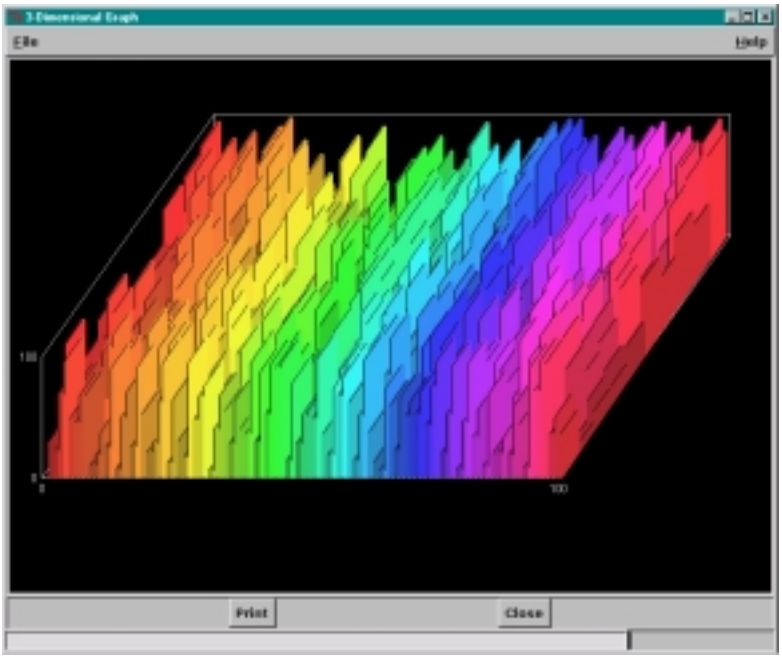


Figure 11.7 three-dimensional graphical display

```
        self.buttonAdd('Close',
                       helpMessage='Close Screen',
                       statusMessage='Exit',
                       command=self.close)

    def createBase(self):
        self.width  = self.root.winfo_width()-10
        self.height = self.root.winfo_height()-95
        self.canvas = self.createcomponent('canvas', (), None,
             Canvas, (self.interior(),), width=self.width,
             height=self.height, background="black")
        self.canvas.pack(side=TOP, expand=YES, fill=BOTH)

        self.awidth  = int(self.width  * 0.68)
        self.aheight = int(self.height * 0.3)
        self.hoffset = self.awidth  / 3
        self.voffset = self.aheight +3
        self.vheight = self.voffset / 2
        self.hrowoff = (self.hoffset / self.rows)
        self.vrowoff = self.voffset / self.rows
        self.xincr   = float(self.awidth) / float(self.steps)
        self.xorigin = self.width/3.7
        self.yorigin = self.height/3
        self.yfactor = float(self.vheight) / float(self.maxY-self.minY)

        self.canvas.create_polygon(self.xorigin, self.yorigin,
             self.xorigin+self.awidth, self.yorigin,
```

❶

```
                self.xorigin+self.awidth-self.hoffset, self.yorigin+self.voffset,
                self.xorigin-self.hoffset, self.yorigin+self.voffset,
                self.xorigin, self.yorigin, fill='', outline=self.lineColor)

            self.canvas.create_rectangle(self.xorigin, self.yorigin-self.vheight,
                self.xorigin+self.awidth, self.yorigin,
                fill='', outline=self.lineColor)

            self.canvas.create_polygon(self.xorigin, self.yorigin,
             self.xorigin-self.hoffset, self.yorigin+self.voffset,
             self.xorigin-self.hoffset, self.yorigin+self.voffset-self.vheight,
             self.xorigin, self.yorigin-self.vheight,
             fill='', outline=self.lineColor)

            self.canvas.create_text(self.xorigin-self.hoffset-5,
                self.yorigin+self.voffset, text='%d' % self.minY,
                fill=self.lineColor, anchor=E)
            self.canvas.create_text(self.xorigin-self.hoffset-5,
                self.yorigin+self.voffset-self.vheight, text='%d' % \
                self.maxY, fill=self.lineColor, anchor=E)

            self.canvas.create_text(self.xorigin-self.hoffset,
                self.yorigin+self.voffset+5, text='%d' % self.minX,
                fill=self.lineColor, anchor=N)
            self.canvas.create_text(self.xorigin+self.awidth-self.hoffset,
                self.yorigin+self.voffset+5, text='%d' % self.maxX,
                fill=self.lineColor, anchor=N)

        def initData(self):
            self.minY =   0
            self.maxY = 100
            self.minX =   0
            self.maxX = 100
            self.steps = 100
            self.rows =  10
            self.spectrum = Pmw.Color.spectrum(self.steps, saturation=0.8,
                                       intensity=0.8, extraOrange=1)
            self.lineColor = 'gray80'
            self.lowThresh =  30
            self.highThresh =  70

        def transform(self, base, factor):
            rgb = self.winfo_rgb(base)
            retval = "#"
            for v in [rgb[0], rgb[1], rgb[2]]:
                v = (v*factor)/256
                if v > 255: v = 255
                if v < 0:   v = 0
                retval = "%s%02x" % (retval, v)
            return retval

        def plotData(self, row, rowdata):
            rootx  = self.xorigin - (row*self.hrowoff)
            rooty  = self.yorigin + (row*self.vrowoff)
            cidx   = 0
            lasthv = self.maxY*self.yfactor
```

```
            xadj   = float(self.xincr)/4.0
            lowv   = self.lowThresh*self.yfactor
            for datum in rowdata:
                 lside = datum*self.yfactor
                 color = self.spectrum[cidx]
                 if datum <= self.lowThresh:
                      color = self.transform(color, 0.8)
                 elif datum >= self.highThresh:
                      color = self.transform(color, 1.2)

                 self.canvas.create_polygon(rootx, rooty, rootx, rooty-lside,
                           rootx-self.hrowoff, rooty-lside+self.vrowoff,
                           rootx-self.hrowoff, rooty+self.vrowoff,
                           rootx, rooty, fill=color, outline=color,
                           width=self.xincr)
                 base = min(min(lside, lasthv), lowv)
                 self.canvas.create_line(rootx-xadj, rooty-lside,
                           rootx-xadj-self.hrowoff, rooty-lside+self.vrowoff,
                           rootx-xadj-self.hrowoff, rooty+self.vrowoff-base,
                           fill='black', width=1)
                 lasthv = lowv = lside

                 cidx = cidx + 1
                 rootx = rootx + self.xincr

       def makeData(self, number, min, max):
            import random
            data = []
            for i in range(number):
                 data.append(random.choice(range(min, max)))
                 return data

       def demo(self):
            for i in range(self.rows):
                 data = self.makeData(100, 4, 99)
                 self.plotData(i, data)
                 self.root.update()

       def close(self):
            self.quit()

       def createInterface(self):
            AppShell.AppShell.createInterface(self)
            self.createButtons()
            self.initData()
            self.createBase()

  if __name__ == '__main__':
       graph = Graph3D()
       graph.root.after(100, graph.demo)
       graph.run()
```

Code comments

❶ Despite the complex diagram, the code is quite simple. Much of the code is responsible for drawing the frame and text labels.

❷ You may have seen the `transform` method used in "Adding a hex nut to our class library" on page 131. Its purpose is to calculate a lighter or darker color intensity when given a color.

```
def transform(self, base, factor):
```

❸ The transformed color is used to highlight values which exceed a high threshold and to deaccentuate those below a lower threshold.

❹ For this example, we generate ten rows of random data.

Because the data was generated randomly, the effect is quite busy. If data is supplied from topological sources, the plot may be used to provide a surface view. Figure 11.8 illustrates the kind of three-dimensional plot that can be produced with such data.

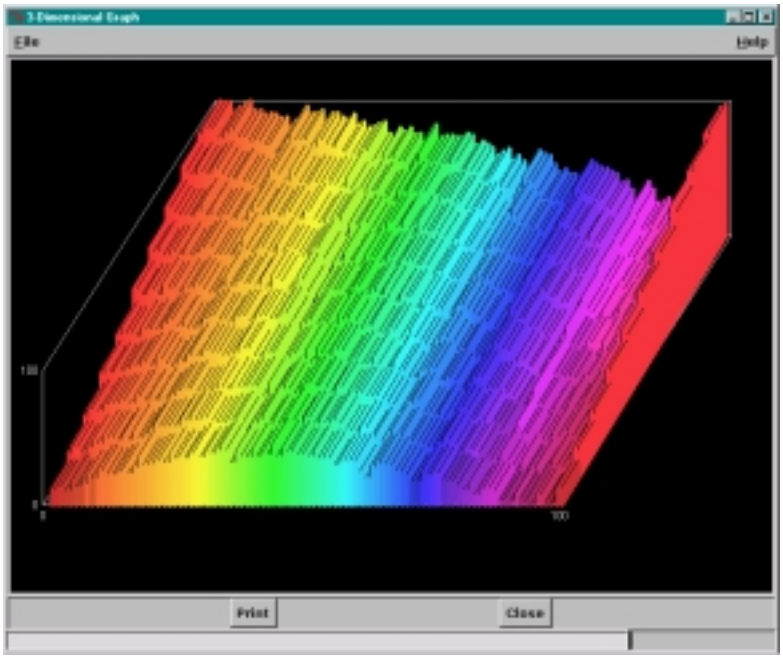


Figure 11.8 Using the 3-D to present topological data

11.4 Strip charts

In this final section we are going to look briefly at using strip charts to display data coming from a source of continuously changing data. Such displays will typically build a plot incrementally as

the data is made available or polled at some time interval, and then they reset the chart when the maximum space has been filled.

Strip charts are an ideal medium for displaying performance data; data from sensors, such as temperature, speed, or humidity; data from more abstract measurements (such as the average number of items purchased per hour by each customer in a grocery store); and other types of data. They also can be used as a means of setting thresholds and triggering alarms when those thresholds have been reached.

The final example implements a weather monitoring system utilizing METAR* data. This encoded data may be obtained via FTP from the National Weather Service in the United States and from similar authorities around the globe. We are not going to enter a long tutorial about how to decode METARs, since that would require a chapter of its own. For this example, I am not even going to present the source code (there is *really* too much to use the space on the printed page). The source code is available online and it may be examined to determine how a simple FTP poll may be made to gather data continuously.

Take a look at figure 11.9 which shows the results of collecting the data from Tampa Bay, Florida (station *KTPA*), for about nine hours, starting at about 8:00 a.m. EST. The graphs depict temperature, humidity, altimeter (atmospheric pressure), visibility, wind speed, wind direction, clouds over 10,000 feet and clouds under 25,000 feet.

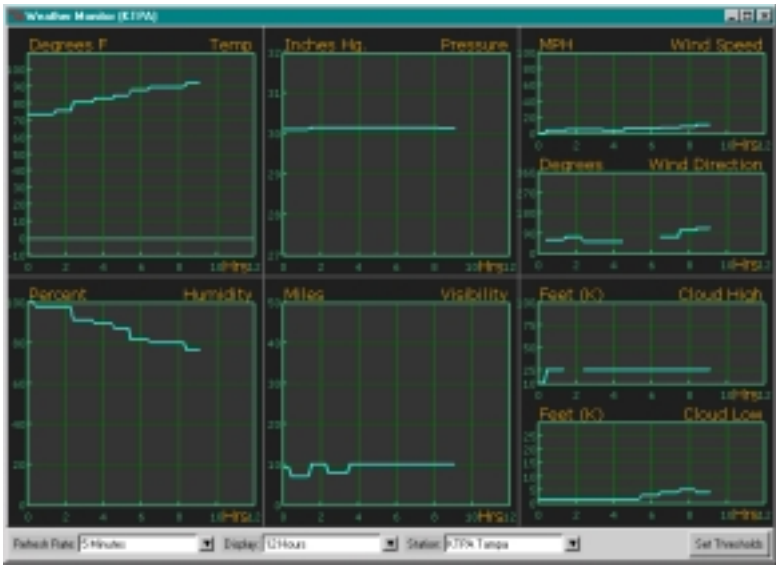


Figure 11.9 Strip chart display with polled meteorological data

* If you are a weather buff or a private pilot, you will be familiar with the automated, encoded weather observations that are posted at many reporting stations, including major airports, around the world. Updated on an hourly basis (more frequently if there are rapid changes in conditions), they contain details of wind direction and speed, temperature, dewpoints, atmospheric pressure, cloud cover and other data important to aviation in particular.

The presentation for the strip chart is intended to be similar to an oscilloscope or some other piece of equipment. Normally reverse-video is not the best medium for presenting data; this may be one of the exceptions.

The example code implements a threshold setting which allows the user to set values which trigger an alarm or warning when values are above or below the selected threshold. Take a look at figure 11.10 which shows how thresholds can be set on the data. This data comes from my home airport (Providence, in Warwick, Rhode Island) and it shows data just before a thunderstorm started.

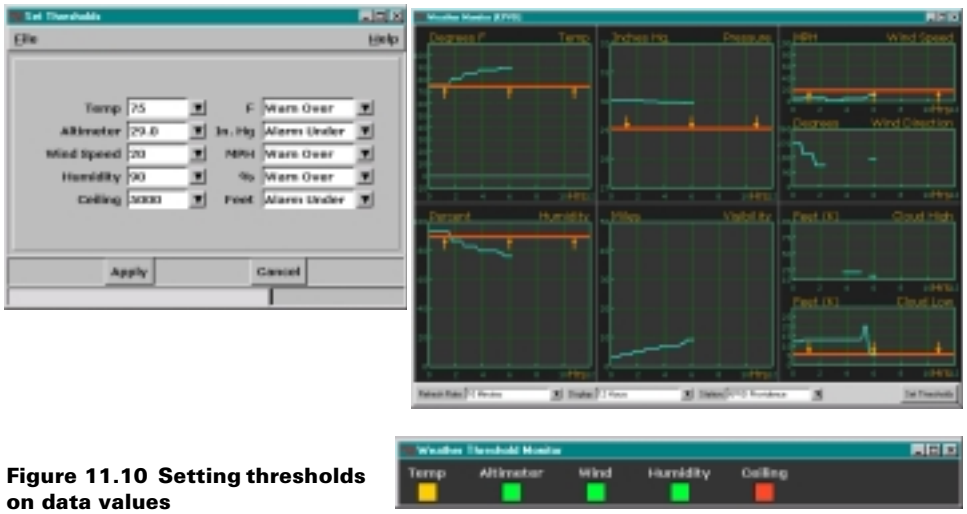


Figure 11.10 Setting thresholds on data values

If you look at figure 11.11 you can observe how the cloud base suddenly dropped below 5000 feet and triggered the threshold alarm.

If you do use this example please do not set the update frequency to a high rate. The data on the National Oceanic and Atmospheric Administration (NOAA) website is important for many pilots—leave the bandwidth for them!

11.5 Summary

Drawing graphs may not be necessary for many applications, but the ability to generate attractive illustrations from various data sources may be useful in some cases. While there are several general-purpose plotting systems available to generate graphs from arbitrary data, there is something satisfying about creating the code yourself.

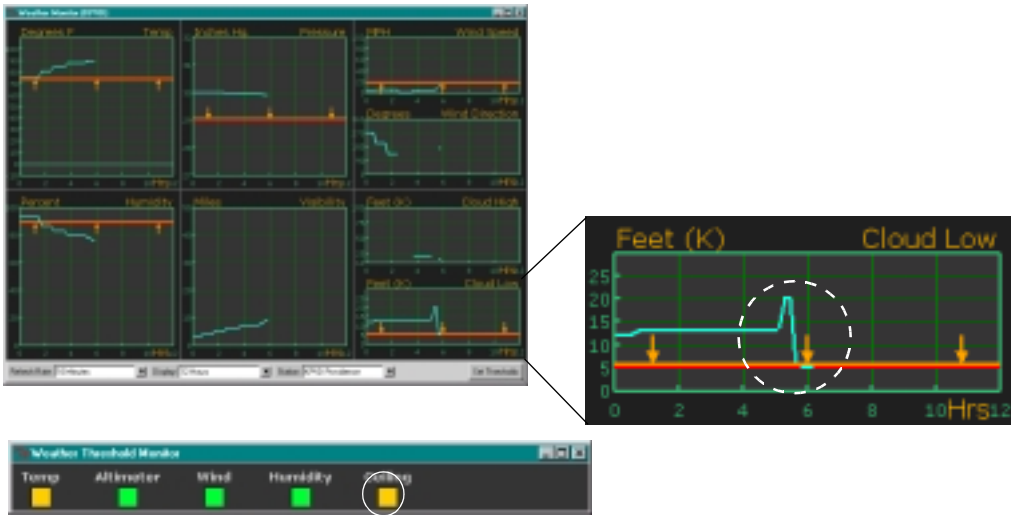


Figure 11.11 Alarm and warning thresholds