Chapter 5

Points In The Plane

All of the programs that we have considered so far involve relatively few variables. The variables that we have used are *scalar* variables where only *one* value is stored in the variable at any time. We have seen problems that involve a lot of data, but there was never any need to store it "all at once." This will now change. Tools will be developed that enable us to store a large amount of data that can be accessed during program execution. We introduce this new framework by considering various problems that involve sets of points in the plane. If these points are given by $(x_1, y_1), \ldots, (x_n, y_n)$, then we may ask:

- What is their centroid?
- What two points are furthest apart?
- What point is closest to the origin (0,0)?
- What is the smallest rectangle that contains all the points?

The usual `input`/`fprintf` methods for input and output are not convenient for problems like this. The amount of data is too large and too geometric. In this chapter we will be making extensive use of MATLAB's graphics functions such as `plot` for drawing an x-y plot, and `ginput` for reading in the x and y coordinates of a mouse click in a figure window on the screen. We postpone the detailed discussion about `plot` until the end of the chapter so that we can focus on another important concept in the early examples. For now, do not be concerned about the commands used to "set up the figure window" in the examples. Brief explanations are given in the program comments to indicate their effect. Be patient! Function `plot` will be explained in §5.2.

5.1 Centroids

Suppose we are given n points in the plane $(x_1, y_1), \ldots, (x_n, y_n)$. Collectively, they define a finite *point set*. Their *centroid* $(\bar{x}, \bar{y})$ is defined by

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \qquad \bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i.$$

See FIGURE 5.1. Notice that $\bar{x}$ and $\bar{y}$ are the averages of the x and y coordinates. The program

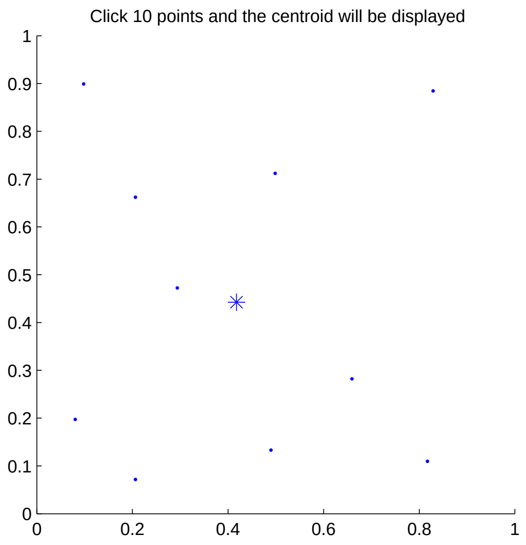


FIGURE 5.1 *Ten Points and Their Centroid*

`Example5_1` calculates the centroid of ten user-specified points to produce FIGURE 5.1. The xy values that define ten points are obtained by clicking the mouse. The statement `[xk,yk]= ginput(1)` stores the x value of a mouse click in variable `xk` and the the y value in `yk`. The summations that are required for the centroid computation are assembled as the data is acquired. The ten points are displayed as dots (.) using MATLAB's `plot` function while the centroid is displayed as an asterisk (`*`), again using the `plot` function. Notice the series of commands near the top of the script that set up the figure window. We wrote a comment for each command as a brief explanation to you—you don't need to write such detailed comments in general.

Now let us augment `Example5_1` so that it draws a line from each of the ten points to the centroid as depicted in FIGURE 5.2. If we try to do this, then we immediately run into a problem: *the x and y values have not been saved.* If the number of points is small, say 3, then we can solve this problem using existing techniques with a fragment like this:

```
    sx= 0; sy= 0;
    [x1,y1]= ginput(1); plot(x1,y1,'.')
    sx= sx+x1; sy= sy+y1;
```

```
% Example 5_1:  Display centroid of 10 user-selected points

n= 10; % Number of points user will click in

% Set up the window
close all             % Close all previous figure windows
figure                % Start a new figure window
hold on               % Keep the same set of axes (multiple plots on same axes)
axis equal            % unit lengths on x- and y-axis are equal
axis([0 1 0 1])       % x-axis limits are [0,1], y-axis limits are [0,1]
title(['Click ' num2str(n) ' points and the centroid will be displayed'])

% Plot the points
sx= 0;   % sum of x values entered so far
sy= 0;   % sum of y values entered so far
for k= 1:n
   [xk,yk]= ginput(1);    % xk = x-position of kth mouse click by user
                          % yk = y-position of kth mouse click by user
   plot(xk, yk, '.')    % Plot a dot at position(xk,yk)
   sx= sx + xk;
   sy= sy + yk;
end

% Compute and display the centroid
xbar= sx/n;
ybar= sy/n;
plot(xbar, ybar, '*', 'markersize', 10)   % Plot a '*' 10 units in size at (xbar,ybar)
```

For sample output, see Figure 5.1.

```
[x2,y2]= ginput(1); plot(x2,y2,'.')
sx= sx+x2; sy= sy+y2;
[x3,y3]= ginput(1); plot(x3,y3,'.')
sx= sx+x3; sy= sy+y3;
% Calculate and plot centroid
xbar= sx/3; ybar= sy/3;
plot(xbar,ybar,'*')
% Connect points to centroid
plot([x1 xbar],[y1 ybar])   % Plot a line from (x1,y1) to (xbar,ybar)
plot([x2 xbar],[y2 ybar])
plot([x3 xbar],[y3 ybar])
```

However, the feasibility of this approach diminishes rapidly as the nubmer of points, n, gets large because approximately $2n$ variables have to be declared and approximately $6n$ statements are required to carry out the computation.

To solve this problem conveniently, we need the concept of the *array*. `Example5_2` introduces this all-important construction. The program has two *array variables* `x` and `y` which may be visualized as follows:

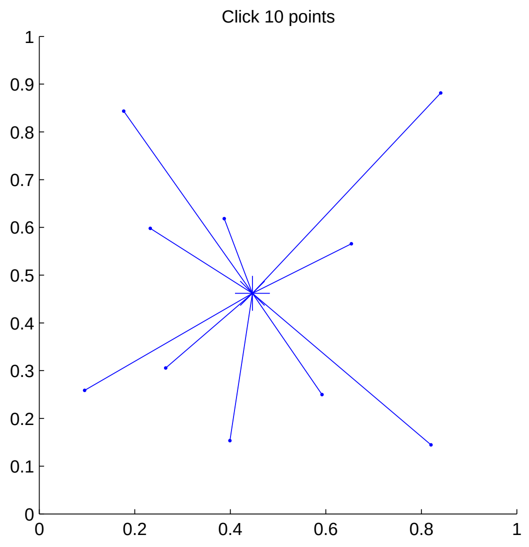
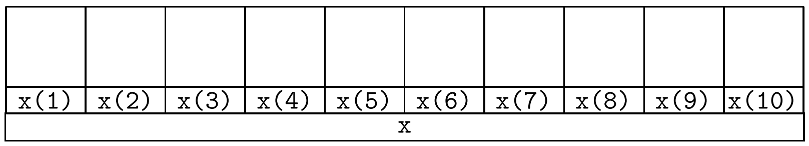
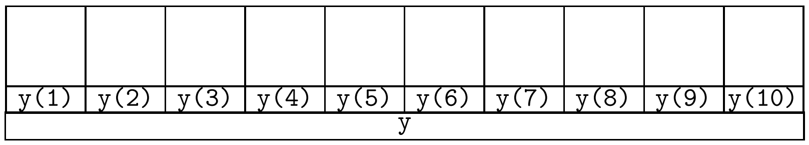


Figure 5.2 *Connecting the Centroid*





These arrays are each able to store up to 10 real values, one value per array *component*. We call these *one-dimensional* arrays *vectors*. In Matlab, vectors can be a *row* or a *column*. Above, `x` and `y` are pictured as row vectors. The program uses the `x` and `y` vectors to store the coordinates of the points whose centroid is required. To understand `Example5_2` fully, we need to discuss how arrays are created and how their components can participate in the calculations.

The built-in function

```
    zeros(1,n)
```

creates and returns a vector 1 row by n columns in size (a row vector) where each component

```
% Example 5_2:  Connect 10 user-selected points to the centroid

n= 10; % Number of points user will click in

% Set up the window
close all
figure
hold on
axis equal
axis([0 1 0 1])
title(['Click ' num2str(n) ' points and the centroid will be displayed'])

% Plot the points
x= zeros(1,n);   % x(k) is x value of the kth point, initialized to 0
y= zeros(1,n);   % y(k) is y value of the kth point, initialized to 0
sx= 0;   % sum of x values entered so far
sy= 0;   % sum of y values entered so far
for k= 1:n
   [x(k),y(k)]= ginput(1);
   plot(xk, yk, '.')
   sx= sx + xk;
   sy= sy + yk;
end

% Compute and display the centroid
xbar= sx/n;
ybar= sy/n;
plot(xbar, ybar, '*', 'markersize', 20)

% Connect the points to the centroid
for k= 1:n
   plot([x(k) xbar], [y(k) ybar])
end
```

For sample output, see Figure 5.2.

is initialized to the value zero. The two arguments of function `zero` specify the number of rows and the number of columns in that order. Therefore, the function call to create a column vector of zeros is `zeros(n,1)`.

The variables `x` and `y` are each assigned the returned vector from the `zeros` function. Therefore, `x` and `y` are *vector* variables. The individual component in such a vector will store values of one type, double precision numbers in our example. Each component has an *index*, or subscript, identifying its position in the vector. The index is an integer and goes from 1 to the number of components in the vector, or 10 in this case.

To illustrate how values can be assigned to an array, consider the fragment

```
x= zeros(1,10); y= zeros(1,10);
x(1)= 3;
y(1)= 4;
```

```
x(2)= 5;
y(2)= 2;
x(3)= 1;
y(3)= 7;
```

This results in the following situation:

3	5	1	0	0	0	0	0	0	0
x(1)	x(2)	x(3)	x(4)	x(5)	x(6)	x(7)	x(8)	x(9)	x(10)
x									

4	2	7	0	0	0	0	0	0	0
y(1)	y(2)	y(3)	y(4)	y(5)	y(6)	y(7)	y(8)	y(9)	y(10)
y									

The key is to recognize that the index is part of the component's name. For example, `x(2)` is the name of a variable that happens to be the second component of the vector `x`. The assignment `x(2)= 6` is merely the assignment of a value to a variable, as we have seen many times before.

A component of a vector can "show up" any place a simple real variable can "show up." Thus, the fragment

```
sx= 0; sy= 0;
[x(1),y(1)]= ginput(1);
sx= sx + x(1); sy= sy + y(1);
[x(2),y(2)]= ginput(1);
sx= sx + x(2); sy= sy + y(2);
[x(3),y(3)]= ginput(1);
sx= sx + x(3); sy= sy + y(3);
```

obtains the data for three points and stores the acquired x and y values in `x(1)`, `x(2)` and `x(3)` and `y(1)`, `y(2)` and `y(3)` respectively. But what makes arrays so powerful is that *array subscripts can be computed.* The preceding fragment is equivalent to

```
for k= 1:3
   [x(k),y(k)]= ginput(1);
   sx= sx + x(k);
   sy= sy + y(k);
end
```

When the loop index `k` has the value of 1, the effective loop body is

```
[x(1),y(1)]= ginput(1);
```

```
        sx= sx + x(1);
        sy= sy + y(1);
```

This solicits the coordinates of the first point, puts the values in the first component of `x` and `y`, and updates the running sums `sx` and `sy`. During the second and third passes through the loop, `k` has the values 2 and 3 and we have, effectively,

```
        [x(2),y(2)]= ginput(1);
        sx= sx + x(2);
        sy= sy + y(2);
```

and

```
        [x(3),y(3)]= ginput(1);
        sx= sx + x(3);
        sy= sy + y(3);
```

In general, array references have the form

⟨*Vector name*⟩(⟨*integer-valued expression*⟩)

The value enclosed within the parenthesis is the *index* or *subscript* and it must be an integer or an expression that evaluates to an integer. Moreover, the value must be in the subscript range of the vector. Given a vector, how do you know how many components it has? The function call `length(x)` returns the length of, or the number of components in, vector `x`. In our example where `x` has length 10, references like `x(0)` or `x(11)` are illegal and would result in program termination. Reasoning about subscript range violations gets complicated when the subscript values are the result of genuine expressions. For example, if `x` has length 10 and `i` is an integer, then `x(2*i-1)` is legal if the value of `i` is either 1, 2, 3, 4 or 5.

When referring to parts of arrays, it is useful to use Matlab's "colon expression." We first saw the colon expression in Chapter 2 when discussing `for`-loops. The expression `2:5` reads as "2 to 5" and refers to the numbers 2, 3, 4, and 5. When dealing with vectors, the Matlab expression `x(2:5)` refers to the components `x(2)`, `x(3)`, `x(4)` and `x(5)`. We refer to `x(2:5)` as a *subvector* of `x`. The subvector `x(i:j)` where $i > j$ denotes an *empty array*.[1]

In `Example5_2` we saw that function `zeros` returns a vector.[2] Similarly, a function can have vector parameters. For example, the centroid computation can be encapsulated as follows:

```
  function [xbar, ybar] = centroid(x, y)
  % Post:  (xbar,ybar) is centroid of points (x(k),y(k)), k=1:length(x)
  % Pre:  0 < length(x)=length(y)

  sx= 0;   % sum of x values so far
  sy= 0;   % sum of y values so far
```

[1]The empty array is analogous to the *null set* or *empty set* (∅) you have encountered in mathematics.

[2]Another frequently used function for creating vectors is `ones`. The function call `ones(1,n)` returns a row vector of length n while `ones(n,1)` returns a column vector of length n.

```
for k= 1:length(x)
   % incorporate the k-th point
   sx= sx + x(k);
   sy= sy + y(k);
end
xbar= sx/length(x);
ybar= sy/length(y);
```

`Example5_3` illustrates the use of this function and is equivalent to `Example5_2`.

```
% Example 5_3:  Connect 10 user-selected points to the centroid

n= 10; % Number of points user will click in

% Set up the window
close all
figure
hold on
axis equal
axis([0 1 0 1])
title(['Click ' num2str(n) ' points and the centroid will be displayed'])

% Plot the points
x= zeros(1,n);   % x(k) is x value of the kth point, initialized to 0
y= zeros(1,n);   % y(k) is y value of the kth point, initialized to 0
for k= 1:n
   [x(k),y(k)]= ginput(1);
   plot(xk, yk, '.')
end

% Compute and display the centroid
[xbar,ybar]= centroid(x,y);
plot(xbar, ybar, '*', 'markersize', 20)

% Connect the points to the centroid
for k= 1:n
   plot([x(k) xbar], [y(k) ybar])
end
```

For sample output, see Figure 5.2.

Problem 5.1. A line drawn from a vertex of a triangle to the midpoint of the opposite side is called a *median*. It can be proven that all three medians intersect at

$$(\bar{x},\bar{y}) = ((x_1+x_2+x_3)/3, (y_1+y_2+y_3)/3).$$

where the triangle's vertices are (x_1,y_1), (x_2,y_2), and (x_3,y_3). The point $(\bar{x},\bar{y})$ is the triangle's centroid and corresponds to its "center of mass." Write a program using arrays that solicits three points and then draws (a) the triangle that they define, (b) displays the triangle's centroid, and (c) draws the three medians.

PROBLEM 5.2. Rewrite function `centroid` to take advantage of useful built-in functions such as `sum` and `mean`. (You only need one of these functions.) `sum(v)` returns the sum of the values in all the components of vector `v` while `mean(v)` returns the mean, or average, of all the values in `v`.

In the last few examples we used built-in functions to create vectors. We also can "manually" enter values in a vector. `Example5_4` obtains data for two user-selected points and then inserts the midpoint, calculated using function `centroid`, into the `x` and `y` vectors. The script is essentially

```
% Example 5_4:  Insert the midpoint between 2 user-selected points

n= 2; % Number of points user will click in

% Set up the window
close all
figure
hold on
axis equal
axis([0 1 0 1])
title(['Click ' num2str(n) ' points and the centroid will be displayed'])

% Plot the points
x= zeros(1,n);   % x(k) is x value of the kth point, initialized to 0
y= zeros(1,n);   % y(k) is y value of the kth point, initialized to 0
for k= 1:n
   [x(k),y(k)]= ginput(1);
   plot(xk, yk, '.')
end

% Compute and display the centroid
[xbar,ybar]= centroid(x,y);
plot(xbar, ybar, '*', 'markersize', 20)

% Insert the midpoint in vectors x, y
x= [x(1) xbar x(2)];
y= [y(1) ybar y(2)];
```

For sample output, see FIGURE 5.3.

identical to `Example5_3` except for the last two statements

```
x= [x(1) xbar x(2)];
y= [y(1) ybar y(2)];
```

Here, vectors `x` and `y` are assigned new values enclosed in *square brackets*. The expression `[x(1) xbar x(2)]` creates a vector with three values—the values in `x(1)`, `xbar`, and `x(2)`, in that order. Using a space or a comma (,) to separate the values results in a *row* vector. If we use a semicolon (;) as the separator, then we will get a *column* vector. FIGURE 5.3 shows an

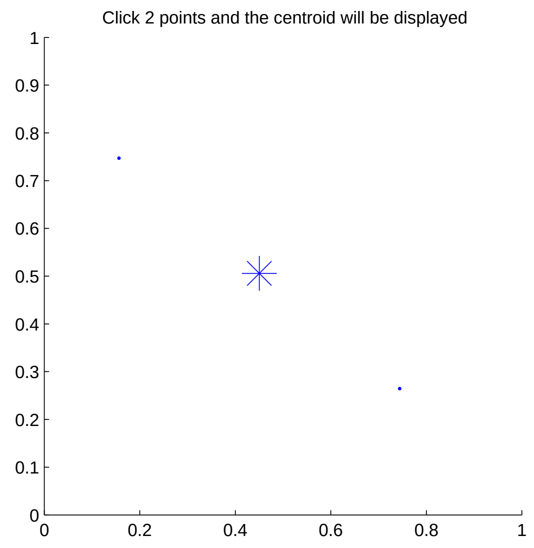


FIGURE 5.3 *Inserting the midpoint between two user-selected points*

example graphical output.

To summarize, the following syntax gives a *row* vector:

[⟨*value1*⟩ ⟨*value2*⟩ ⟨*value3*⟩]
[⟨*value1*⟩,⟨*value2*⟩,⟨*value3*⟩]

To get a *column* vector, the syntax is

[⟨*value1*⟩; ⟨*value2*⟩; ⟨*value3*⟩]

Any number of values can be specified in a vector.

Function `getPoints` gives another example of using the bracket notation to create vectors. This time, we "grow" the vector one cell at a time! Here is how it works. Points are clicked in until the user chooses to stop by clicking in the designated "stop" area at the top of the window. See FIGURE 5.4. The function has to store all the x- and y-coordinates without knowing at the beginning how many points the user will click. The problem is solved by adding on the latest data point `xp` to the end of the *current* `x` vector at every pass of the loop:

```
x= [x xp];
```

Notice that the separator is the space, meaning that `xp` is *concatenated* to the right of the current vector `x`. This means that vector `x` grows by one cell after each pass of the loop. To make the growing work, `x` must exist with some value when the statement `x= [x xp]` executes for the first

```
function [x, y] = getPoints()
% Post:  x, y are vectors storing the x and y values of mouse clicks.
% User clicks in a specified area to stop the collection of data.

% Set up the window
close all
figure
hold on
axis equal
axis([-1 1 -1 1.2])

% Set top 'bar' of window to be the area for user to indicate stop
line([-1 1], [1 1], 'color', 'r')   % draw line at y=1
text(-0.75, 1.1, 'Click here to stop or click below red line')

% Get point data
n= 0;   % Number of points so far
x= [];   % x-coordinates of points (initialized to empty array)
y= [];   % y-coordinates of points (initialized to empty array)
[xp, yp]= ginput(1);
while ( yp < 1)
   plot(xp, yp, '.')
   n= n + 1;
   x= [x xp];   % Concatenate xp to current vector x
   y= [y yp];
   [xp, yp]= ginput(1);   % Get next point
end
```

An example graphical output of a function call to `getPoints` is shown in Figure 5.4.

time. Yet at the start of the function, there are no points so x should be empty. We therefore use the *empty array* notation `[]` to be the initial "value" for `x` and `y`.

Problem 5.3. Modify function `getPoints` to implement the following function:

```
function [x, y]= randPoints(n)
% Post:  x,y are vectors of coordinates for n randomly generated points.
%   -1<x(k),y(k)<1 for all k= 1..n.  Draw all the points.  x, y are
%   empty arrays if n=0.
% Pre:  n>=0
```

Problem 5.4. Complete the following function:

```
function [x, y]= smoothPoints(x,y);
% Post:  Smooth the line represented by vectors x,y by averaging values of neighboring points.
%   Replace (x(i),y(i)) with midpoint of the line segment that connects this point with
```

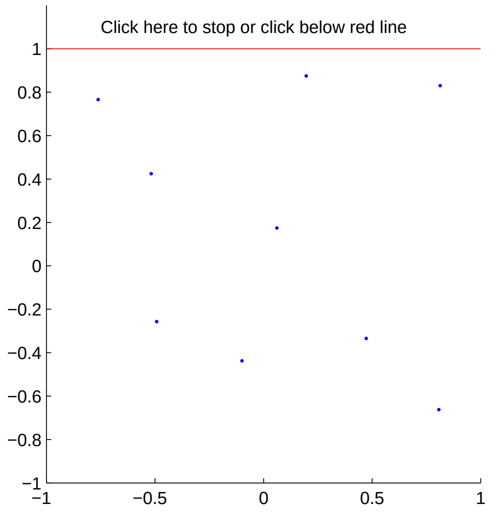


FIGURE 5.4 *The* `getPoints` *function*

```
%   its successor (x(i+1),y(i+1)).  Consier (x(1),y(1)) to be the successor of (x(n),y(n))
%   where n is length of x, y.
% Pre:  length(x)=length(y)>=1
```

5.2 Max's and Mins

Consider the problem of finding the smallest rectangle that encloses the points $(x_1, y_1), \ldots, (x_n, y_n)$. We assume that the sides of the sought-after rectangle are parallel to the coordinate axes as depicted in FIGURE 5.5. From the picture we see that the left and right edges of the rectangle are situated at

$$\begin{aligned} x_L &= \min\{x_1, \ldots, x_n\} \\ x_R &= \max\{x_1, \ldots, x_n\} \end{aligned}$$

while the bottom and top edges are specified by

$$\begin{aligned} y_B &= \min\{y_1, \ldots, y_n\} \\ y_T &= \max\{y_1, \ldots, y_n\} \end{aligned}$$

The problem that confronts us is clearly how to find the minimum and maximum value in a given array.

Suppose vector x has length 4. To obtain the maximum value in this array, we could proceed as follows:

```
s= x(1);
if  x(2) > s
   s= x(2);
end
if  x(3) > s
   s= x(3);
end
if  x(4) > s
   s= x(4);
end
```

The idea behind this fragment is (a) to start the search by assigning the value of `x(1)` to `s` and then (b) to scan `x(2:4)` for a larger value. This is done by "visiting" `x(2)`, `x(3)`, and `x(4)` and comparing the value found with `s`. The mission of `s` is to house the largest value "seen so far." A loop is required for a general n:

```
s= x(1);
for i= 2:n
   if  x(i)>s
      s= x(i);
   end
end
```

Note that after the i-th pass through the loop, the value of `s` is the largest value in `x(1:i)`. Thus, if

x(1:6) =

3	2	5	2	7	5

then the value of `s` changes as follows during the search:

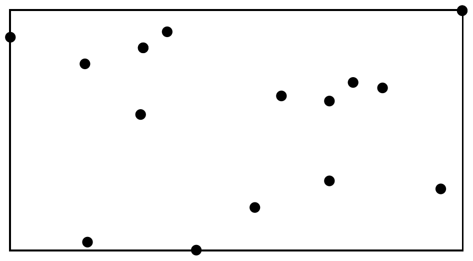


Figure 5.5 *Minimum Enclosing Rectangle*

i	Value of s after i-th pass.
Start	3
2	3
3	5
4	5
5	7
6	7

Packaging these find-the-max ideas we get

```
function s = maxInList(x)
% Post:  s is largest value in vector x
% Pre:  length(x)>=1
s= x(1);
for k= 2:length(x)
   if  x(k) > s
      s= x(k);   %{s = largest value in x(1:k)}
   end
end
```

Searching for the minimum value in an array is entirely analogous. We merely replace the conditional

```
if  x(k) > s
   s= x(k);
end
```

with

```
if  x(k) < s
   s= x(k);
end
```

so that s is revised downwards anytime a new smallest value is encountered.

```
function s = minInList(x)
% Post:  s is smallest value in vector x
% Pre:  length(x)>=1
s= x(1);
for k= 2:length(x)
   if  x(k) < s
      s= x(k);   %{s = smallest value in x(1..k)}
   end
end
```

With maxInList and minInList available we can readily solve the smallest enclosing rectangle problem. Example5_5 shows that two calls to each of these functions are required. We use function

```
% Example 5_5:  Smallest enclosing rectangle

[x, y]= getPoints();
xL= minInList(x);   % Left end of rectangle
xR= maxInList(x);   % Right end of rectangle
yB= minInList(y);   % Bottom of rectangle
yT= maxInList(y);   % Top of rectangle

% Draw the rectangle
cx= [xL xR xR xL];   % x-position of corners, clockwise from left
cy= [yT yT yB yB];   % y-position of corners, clockwise from left
% Function plot joins pairs of points in the order given in the vectors.
% Given 4 points, only 3 line segments will be plotted.  To "close" the
% rectangle, augment the vectors at the end with the 1st point:
cx= [cx cx(1)];
cy= [cy cy(1)];
plot(cx,cy)
title('Smallest enclosing rectangle')
```

For sample output, see Figure 5.6.

`getPoints` to obtain a set of user-selected points. Figure 5.6 shows an example output.

Throughout this chapter we have used the `plot` function to draw our points in the plane. No doubt you have some ideas now about how the graphics functions work. Let us now examine `plot` in detail.

In `Example5_5`, we have the statement

```
plot(cx,cy)
```

which draws line segments connecting the points (`cx(1)`,`cy(1)`) with (`cx(2)`,`cy(2)`), (`cx(2)`,`cy(2)`) with (`cx(3)`,`cy(3)`), and so on, until the last line segment connecting the points (`cx(4)`,`cy(4)`) with (`cx(5)`,`cy(5)`). In general,

```
plot(a,b)
```

draws a graph in Cartesian coordinates (x-y axes) using the data points (`a(k)`,`b(k)`) where $k = 1, 2, \ldots$, `length(a)`=`length(b)`. Joining the data points with line segments is the default graph format of `plot`. You can specify the line and/or marker format by adding a third argument in the function call to `plot`. For example, changing the plot statement in `Example5_5` to

```
plot(cx, cy, '*')
```

will plot the data points with asterisks instead of joining them up with lines. You can add even more arguments to specify the size of the asterisk:

```
plot(cx, cy,'*','markersize',20)   % 20-point size for the data marker
```

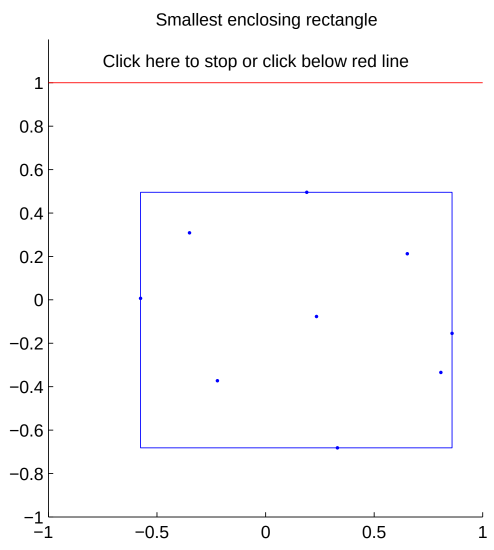


FIGURE 5.6 *Minimum Enclosing Rectangle from* `Example5_5`

An example output using the above statement to mark the four corners of the rectangle is shown in FIGURE 5.7. Want to have the data points connected with lines *and* marked by asterisks? Just add a dash (-) to the marker symbol specification:

```
plot(cx, cy,'-*','markersize',20)   % Note the specification '-*'
```

Here are some examples of the line/marker format that you can use with the `plot` function:

Line/marker specification	Effect
`-`	Join data with line, no markers (this is the default)
`:`	Join data with dotted line, no markers
`*`	Mark data with asterisks, no line
`.`	Mark data with dots, no line
`o`	Mark data with circles, no line
`x`	Mark data with crosses, no line
`-x`	Join data with line, mark data with crosses
`:x`	Join data with dotted line, mark data with crosses
`--x`	Join data with dashed line, mark data with crosses
`-xr`	Join data with line, mark data with crosses, in red
`-xb`	Join data with line, mark data with crosses, in blue
`-xk`	Join data with line, mark data with crosses, in black

These are just a few examples. Use MATLAB's *help* facility to find more details and options. You can also edit your plot *after* a `plot` command has been executed by using the menu bar on the top of the displayed figure window.

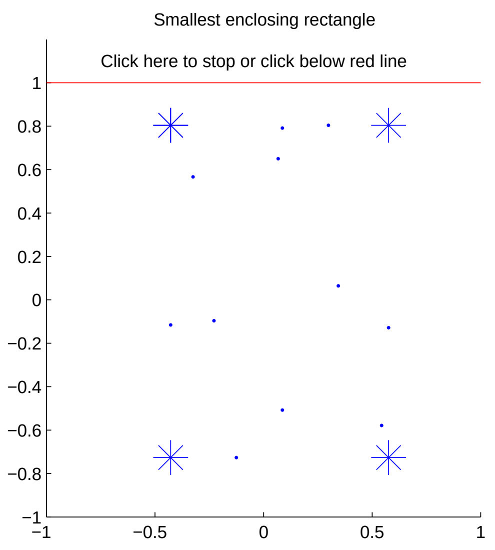


Figure 5.7 *Corners of Minimum Enclosing Rectangle marked with asterisks*

In all the examples in this chapter, we draw multiple graphs on the same set of axes by using the command

```
hold on
```

Once the `hold on` command has been issued, any subsequent call to `plot` will *add* graphs or points to the the current set of axes. To indicate the end of a current plot and that a later `plot` command should replace the current plot with a new one, issue the `hold off` command.

Let us now return to the topic of "max/min" thinking by considering an important "nearest point" problem. Assume the availability of a distance function

```
function d = distance2pts(x1,y1, x2,y2)
% Post:  d is the distance from (x1,y1) to (x2,y2)
```

and that our goal is to find that point in the set $\{(x_1, y_1), \ldots, (x_n, y_n)\}$ that is closest to a given point (x_0, y_0). See Figure 5.8. There are actually *two* things for us to find: the index of the closest point and its distance from (x_0, y_0). Function `nearest` below solves this problem:

```
function [index, dmin] = nearest(x0,y0, x,y)
% Post:  (x(index),y(index)) is the point closest to (x0,y0).
%        dmin is the distance of the closest point (x0,y0).
% Pre:  x and y are coordinate vectors of equal length

index= 1;   % Index of closest point so far
```

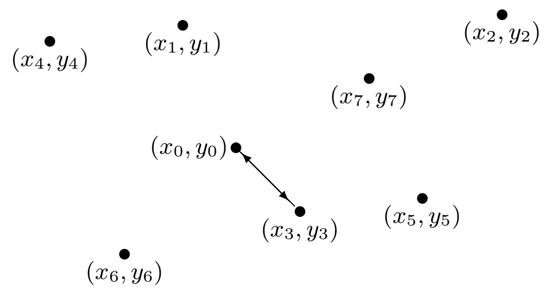


FIGURE 5.8 *Nearness to a Point Set*

```
dmin= distance2pts(x0,y0, x(index),y(index));
for k= 2:length(x)
   dk= distance2pts(x0,y0, x(k),y(k));
   if  (dk <dmin)
      dmin= dk;
      index= k;
   end
end
```

The "search philosophy" is identical to that used in the function `minInList`. An initial value for the minimum is established and then a loop steps through all the other possibilities. Whenever a new minimum distance is found, two variables are revised. The new minimum value is assigned to `dmin` and the index of the new closest point is assigned to `index`.

`Example5_6` illustrates the use of the function `nearest` and uses a *single* call to the function `plot` to draw two graphs on the axes. The first graph is the largest point-free circle centered at the origin, i.e., the circle centered at (0,0) with `dmin` as the radius. The second "graph" is the set of points stored in `x`, `y`. An example of the output is shown in FIGURE 5.9. In order to draw the circle, we first have to compute the coordinates of the points on the circle centered at the origin using the equations $x = r\cos(\theta)$ and $y = r\sin(\theta)$ where r is the radius. In `Example 5_6`, we use 100 points to draw the circle.

Notice how we use a single call of the `plot` function to draw two graphs on the same set of axes:

```
plot(xcircle,ycircle,'-', x,y,'*')
```

The first graph is the circle where the xy data are stored in vectors `xcircle` and `ycircle` and it is drawn with the data points connected by a line (format specification is `'-'`). The second graph is the set of user-specified points stored in vectors `x` and `y`, marked by asterisks (format specification `'*'`). Basically, in the argument list of the function `plot` every *pair* of vectors *of the same length* specifies the xy data for one graph, followed by the (optional) format specification to be applied to that graph.

```
% Example5_6:  Largest point-free circle

[x, y] = getPoints();
[index, radius] = nearest(0,0, x, y);

% Compute the coordinates of the circle centered at (0,0):
% the set of points (xcircle(p),ycircle(p)) for p=1:npoints
npoints= 100;   % draw the circle with n points
xcircle= zeros(1,npoints);
ycircle= zeros(1,npoints);
step= 2*pi/npoints;
for p= 1:npoints
   theta= step*p;
   xcircle(p)= cos(theta) * radius;
   ycircle(p)= sin(theta) * radius;
end
xcircle=[xcircle xcircle(1)]; ycircle=[ycircle ycircle(1)];

% Draw two graphs:  (1) the circle, (2) the user-specified points
plot(xcircle,ycircle,'-', x,y,'*')
title('Largest Point-Free Circle Centered at the Origin')
xlabel('X')
ylabel('Y')
```

For sample output, see Figure 5.9.

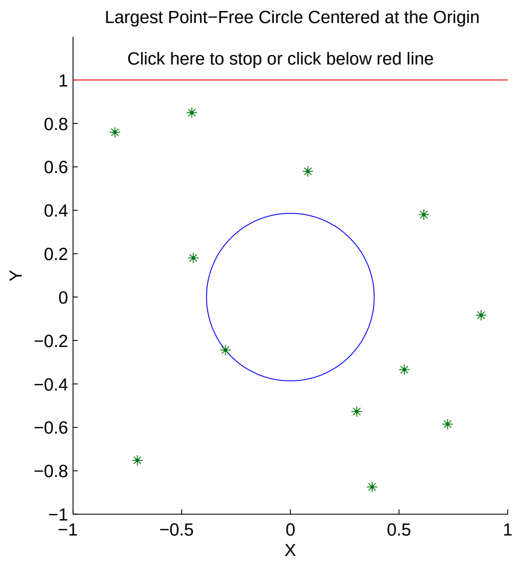


Figure 5.9 *Sample output from* `Example5_6`

We also add a title and the labels for the axes in `Example5_6` using the commands `title`, `xlabel`, and `ylabel`:

```
title('Largest Point-Free Circle Centered at the Origin')
xlabel('X')
ylabel('Y')
```

We simply enclose the text for the title or label inside quotation marks. If the title or label needs to incorporate variable values in addition to simple text, we can use the built-in function `sprintf` in much the same way as we have used `fprintf`. For example,

```
words= sprintf('Largest Point-Free Circle has Radius %f', radius);
title(words)
```

will print as the title the text

```
Largest Point-Free Circle has Radius 0.4
```

if the value stored in variable `radius` is 0.4. All the substitution sequences that we use with `fprintf` can be used with `sprintf`. The only difference is that `sprintf` *returns* the text that can then be stored in a variable for later use, whereas `fprintf` only *prints* text (to the screen) without the option to save it for later use.

Another way to format a plot is to use the menu bar items that are displayed in the figure window once the `plot` command has been executed. The format options in the menu bar allows you to change line/marker style and color, add a title, add axis labels, and add text anywhere on the graph, among other things.

Our discussion above covers the fundamentals of plotting using MATLAB. Later chapters will further demonstrate MATLAB's graphics facility. Experimentation is an important aspect of learning, so "play" with the examples that we have given to learn more! You can also use MATLAB's *Help* documentation to find out more about graphics.

PROBLEM 5.5. We say that the set

$$\{\ (x_1+h_x, y_1+h_y), \ldots, (x_n+h_x, y_n+h_y)\ \}$$

is a *translation* of the set

$$\{\ (x_1,y_1), \ldots, (x_n,y_n)\ \}\ .$$

Write a script that translates into the first quadrant, an arbitrary set of points obtained by `getPoints`. Choose the translation factors h_x and h_y so that the maximum number of points in the translated set are on the x and y axes. Display the translated set in a color different from that of the original point set.

PROBLEM 5.6. Complete the following procedure

```
function [p,q,dp,dq] = twoNearest(x0,y0, x,y)
% Post:   Points (x(p),y(p)) and (x(q),y(q)) are the closest and second closest points
%         to (x0,y0).  dp and dq are their respective distances.
% Pre:  length(x) = length(y) >=2
```

Using `twoNearest`, write a program that obtains an arbitrary set of points using `getPoints` and draws a triangle defined by the origin and the two points nearest to it.

Problem 15.7. Complete the following:

```
function [x, y] = smoothPoints(x, y);
% Post:    Replace (x(i),y(i)) with the midpoint between it and its successor.
%          (The successor of the ith point is point (i+1) unless i=n.  The successor
%          of the n-th point is the 1st point.)
% Pre:  length(x) = length(y) >=1

function [x, y] = translatePoints(x, y)
% Post:    For i=1:length(x), x(i) and y(i) are replaced by x(i)-xbar and y(i)-ybar}
%          where (xbar,ybar) is the centroid.
% Pre:  length(x) = length(y) >=1

function [x, y] = scalePoints(x, y, r)
% Post:    The point set is scaled so that r is the distance of the
%          furthest point to the origin.
% Pre:  length(x) = length(y) >=1

function ratio = variation(x, y)
% Post:  The ratio of the longest to shortest edge of the polygon obtained by
%          connecting the points (x(1),y(1)),...,(x(n),y(n)) in order.
% Pre:  length(x) = length(y) >=1
```

Using these functions, build an environment that permits the easy exploration of the changes that a random point set undergoes with repeated smoothing. After each smoothing, the point set should be translated and scaled for otherwise it "collapses" and disappears from view. You should find that the polygon defined by the points takes on an increasingly regular appearance. The function `variation` measures this and its returned value should be displayed.