Chapter 5

Points In The Plane

§5.1 Centroids
One-dimensional arrays—vectors, initializing vectors, functions with vector parameters, functions that return vectors.

§5.2 Max’s and Min’s
Algorithm for finding the max in a list, function plot

All of the programs that we have considered so far involve relatively few variables. The variables that we have used so far 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/printf 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 $n$ given 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

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

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](image)

Example 5.1 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 $[xk, yk] = \text{ginput}(1)$ stores the $x$ value of a mouse click in variable $xk$ and 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 Example 5.1 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 $n$ is small, say 3, then we can solve this problem using existing techniques with a fragment like this:

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

% 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 \( n \) 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 \textit{array}. \textbf{Example 5.2} introduces this all-important construction. The program has two \textit{array variables} \( x \) and \( y \) which may be visualized as follows:
These arrays are each able to store up to 10 real values, one value per array component. We call each of 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 Example 5.2 fully, we need to discuss how arrays are created and how their components can participate in the calculations.

The built-in function

\[
\text{zeros}(1,n)
\]

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
    [xk, yk] = 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 zeros specify the number of rows
and the number of columns in that order. Therefore, the function call to use for 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;
This results in the following situation:

\[
\begin{array}{cccccccccc}
3 & 5 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
x(1) & x(2) & x(3) & x(4) & x(5) & x(6) & x(7) & x(8) & x(9) & x(10) \\
\end{array}
\]

\[
\begin{array}{cccccccccc}
4 & 2 & 7 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
y(1) & y(2) & y(3) & y(4) & y(5) & y(6) & y(7) & y(8) & y(9) & y(10) \\
\end{array}
\]

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

\[
sx = 0; sy = 0; \\
[x(1),y(1)] = \text{ginput}(1); \\
sx = sx + x(1); sy = sy + y(1); \\
[x(2),y(2)] = \text{ginput}(1); \\
sx = sx + x(2); sy = sy + y(2); \\
[x(3),y(3)] = \text{ginput}(1); \\
sx = sx + x(3); sy = sy + y(3);
\]

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

\[
\text{for } k = 1:3 \\
\quad [x(k),y(k)] = \text{ginput}(1); \\
\quad sx = sx + x(k); \\
\quad sy = sy + y(k); \\
\text{end}
\]

When the loop index \(k\) has the value of 1, the effective loop body is

\[
[x(1),y(1)] = \text{ginput}(1);
\]
5.1. Centroids

\[ \text{sx} = \text{sx} + \text{x}(1); \]
\[ \text{sy} = \text{sy} + \text{y}(1); \]

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

\[ \text{[x}(2),\text{y}(2)] = \text{ginput}(1); \]
\[ \text{sx} = \text{sx} + \text{x}(2); \]
\[ \text{sy} = \text{sy} + \text{y}(2); \]

and

\[ \text{[x}(3),\text{y}(3)] = \text{ginput}(1); \]
\[ \text{sx} = \text{sx} + \text{x}(3); \]
\[ \text{sy} = \text{sy} + \text{y}(3); \]

In general, array references have the form

\[(\text{Vector name})(\text{integer-valued expression})\]

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 \text{length}(\text{x}) returns the length of, or the number of components in, vector \text{x}. In our example where \text{x} has length 10, references like \text{x}(0) or \text{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 \text{x} has length 10 and \( i \) is an integer, then \text{x}(2*\text{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 the “dot-dot” notation. The notation \text{x}(2..5) refers to the components \text{x}(2), \text{x}(3), \text{x}(4) and \text{x}(5). We refer to \text{x}(2..5) as a subarray. The subarray \text{x}(i..j) where \( i > j \) denotes an empty subarray (analogous to the empty set \( \emptyset \)).

In Example5.2 we saw that function \text{zeros} returns a vector.\(^1\) 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
for k= 1:length(x)
    \% incorporate the k-th point
    sx= sx + x(k);
```

\(^1\)Another frequently used function for creating vectors is \text{ones}. The function call \text{ones}(1,n) returns a row vector of length \( n \) while \text{ones}(n,1) returns a column vector of length \( n \).
sy = sy + y(k);
end
xbar = sx/length(x);
ybar = sy/length(y);

Example 5.3 illustrates the use of this function and is equivalent to Example 5.2.

% 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(x(k), y(k), '.');
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 \((x_1, y_1), (x_2, y_2), \text{ and } (x_3, y_3)\). 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. Example 5.4 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 identical to Example 5.3 except for the last two statements

```matlab
% 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, 'r*', '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.
```

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 example graphical output.
To summarize, the following syntax gives a row vector:

\[
\begin{bmatrix}
\langle \text{value1} \rangle \\
\langle \text{value2} \rangle \\
\langle \text{value3} \rangle
\end{bmatrix}
\]

To get a column vector, the syntax is

\[
\begin{bmatrix}
\langle \text{value1} \rangle \\
\langle \text{value2} \rangle \\
\langle \text{value3} \rangle
\end{bmatrix}
\]

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 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 creates a row vector. To make this “growing” work, x must exist with some value when the statement \( x = [x \ xp] \) executes for the first time. Yet at the start of the function, there are no points so x should be empty. We therefore use the empty array \( [\ ] \) to be the initial “value” for x and y.\(^2\)

---

\(^2\)The empty array is analogous to the null set or empty set you have encountered in mathematics.
Problem 5.3. Modify function \texttt{getPoints} to implement the following function:

\begin{verbatim}
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 \( k = 1 \ldots n \). Draw all the points. x, y are
% empty arrays if \( n = 0 \).
% Pre: \( n \geq 0 \)
\end{verbatim}

Problem 5.4. Complete the following function:

\begin{verbatim}
function [x, y] = smoothPoints(x, y);
% Post: Smooth the line represented by vectors x, y by averaging values of neighboring points.
% Replace \((x(1), y(1))\) with midpoint of the line segment that connects this point with
% its successor \((x(n+1), y(n+1))\). 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
\end{verbatim}
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
5.2. Max’s and Mins

situated at

\[ \begin{align*}
    x_L &= \min\{x_1, \ldots, x_n\} \\
    x_R &= \max\{x_1, \ldots, x_n\}
\end{align*} \]

while the bottom and top edges are specified by

\[ \begin{align*}
    y_B &= \min\{y_1, \ldots, y_n\} \\
    y_T &= \max\{y_1, \ldots, y_n\}
\end{align*} \]

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:

\[
\begin{align*}
    s &= x(1); \\
    &\text{if } x(2) > s \\
    &\quad s = x(2); \\
    &\text{end} \\
    &\text{if } x(3) > s \\
    &\quad s = x(3); \\
    &\text{end} \\
    &\text{if } x(4) > s \\
    &\quad s = x(4); \\
    &\text{end}
\end{align*} \]

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\(^3\). 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 general \( n \):

\[
\begin{align*}
    s &= x(1); \\
    &\text{for } i = 2:n \\
    &\quad \text{if } x(i) > s \\
    &\quad\quad s = x(i); \\
    &\quad\text{end} \\
    &\text{end}
\end{align*} \]

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

\[
x(1..6) = \begin{bmatrix} 3 & 2 & 5 & 2 & 7 & 5 \end{bmatrix}
\]

then the value of \( s \) changes as follows during the search:

\(^3\)The dot-dot notation reads as “on up to.” Thus, \( x(2..4) \) means \( x(2) \) on up to \( x(4) \).
Packaging these find-the-max ideas we get

```matlab
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
end
```

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

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

with

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

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

```matlab
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
end
```

With `maxInList` and `minInList` available we can readily solve the smallest enclosing rectangle problem. Example 5.5 shows that two calls to each of these functions are required. We use function

<table>
<thead>
<tr>
<th>$i$</th>
<th>Value of $s$ after $i$-th pass.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>
5.2. Max's and Mins

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 show 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 Example 5.5, 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 \( k = 1, 2, \ldots, \text{length}(a) = \text{length}(b) \). 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 Example 5.5 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
```
An example output is shown. See 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:

\[
\text{plot(cx, cy,'-*', 'markersize', 20)} \quad \% \text{ Note the specification '-*'}
\]

Here are some examples of the line/marker format that you can use with the \texttt{plot} function:

<table>
<thead>
<tr>
<th>Line/marker specification</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>Join data with line, no markers (this is the default)</td>
</tr>
<tr>
<td>:</td>
<td>Join data with dotted line, no markers</td>
</tr>
<tr>
<td>.</td>
<td>Mark data with asterisks, no line</td>
</tr>
<tr>
<td>o</td>
<td>Mark data with circles, no line</td>
</tr>
<tr>
<td>x</td>
<td>Mark data with crosses, no line</td>
</tr>
<tr>
<td>-x</td>
<td>Join data with line, mark data with crosses</td>
</tr>
<tr>
<td>:x</td>
<td>Join data with dotted line, mark data with crosses</td>
</tr>
<tr>
<td>-x</td>
<td>Join data with dashed line, mark data with crosses</td>
</tr>
<tr>
<td>-xr</td>
<td>Join data with line, mark data with crosses, in red</td>
</tr>
<tr>
<td>-xb</td>
<td>Join data with line, mark data with crosses, in blue</td>
</tr>
<tr>
<td>-xk</td>
<td>Join data with line, mark data with crosses, in black</td>
</tr>
</tbody>
</table>

These are just a few examples. You can also edit your plot using the menus on the figure window.

You can also draw multiple graphs using the same call to \texttt{plot}. For example

\[
\text{plot(a,b,c,d,'*')} \]

\textbf{Figure 5.6 Minimum Enclosing Rectangle from Example 5.5}
will draw two graphs in the same plot (same set of axes): a vs b will be joined by lines, c vs d will be marked by asterisks. Basically, in the argument list every pair of vectors of the same length specifies the data for one graph, followed by any line/marker specification to be applied to that graph.

Another way to make multiple graphs on the same set of axes involve the use of the command

\texttt{hold on}

You have seen the use of \texttt{hold on} in all the examples in this chapter. Once the \texttt{hold on} command has been issued, any subsequent call to \texttt{plot} will add graphs or points to the the current set of axes. To replace the current plot with a new one, issue the \texttt{hold off} command.

What about adding a plot title and labels for the axes. You can use the menus in the figure window or you can use the following commands:

\begin{verbatim}
  title('my title')
  xlabel('my x-axis label')
  ylabel('my y-axis label')
\end{verbatim}

Our discussion above covers the fundamentals of plotting using \textsc{Matlab}. Later chapters will further demonstrate \textsc{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 \textsc{Matlab}'s \textit{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.