CS100M Fall 2006   Project 2
Due: September 21, 2006 (Thursday) at 6pm

Submit your files on-line in CMS before the project deadline. Both correctness and good programming style contribute to your project score.

You must work either on your own or with one partner. You may discuss background issues and general solution strategies with others, but the project you submit must be the work of just you (and your partner). If you work with a partner, you and your partner must register as a group in CMS and submit your work as a group.

In this assignment, you will work with loops, write and call user-defined functions, and experiment with Matlab graphics. Along the way, you will explore some mathematical ideas and learn about the Bisection Method for root approximation. The last question talks about decomposition and modular design. We’ve done the decomposition for you in this project. Learn from this example and apply decomposition (modular design) in the future.

Do not use arrays (vectors) in this project.

1. Mode

The mode of a sequence of numbers is the number that occurs most frequently. For example,

\[
\begin{align*}
87, 92, 92, 98, 98, 98, 100 & \quad \text{mode is 98} \\
3, 4, 4, 6, 8, 9, 9 & \quad \text{mode is 4 or 9}
\end{align*}
\]

Write a script to determine the mode of a user-entered sequence of numbers. If there are multiple modes you may report any one of them. You may assume that the sequence is non-negative and is entered one number at a time (prompted by the program) in non-decreasing order. The user enters a negative number to terminate the sequence, but the negative number is a stopping signal only and does not belong to the sequence.

(The question has been discussed in lecture—refer to your lecture notes. The notes also show examples of the “interactive framework” for soliciting user input repeatedly.)

Submit your script file findMode.m.

2. Root approximation using the Bisection Method

Approximation is an important solution strategy for many problems. When an “exact” solution for a problem is too difficult or too computationally expensive to obtain, one may turn to approximation techniques. Simple numerical methods can be used to approximate roots or extrema (minimum or maximum). Some well-known root finding methods include Newton-Raphson, fixed-point iteration, and bisection. In this project, we will use the bisection method.

The root of a function \( f(x) \) is a value of \( x \) that makes \( f(x) = 0 \). The bisection method works with intervals to find the root: start with an interval that contains one root and systematically half the root-containing interval until it is so narrow that one can reasonably take the midpoint of the final interval as an approximation of the root. The key idea is that if an interval in \( x \) contains a single root, then a sign change in the function value must occur within that interval. In the diagram below, the root \( x = c \) lies between points \( a \) and \( b \) where \( f(a) < 0 \) and \( f(b) > 0 \). The bisection method is simple to use, but its limitations are plain to see.
Algorithm for finding a root using the bisection method:
- Set the tolerance (or precision). For example, a tolerance of 0.001 means that your answer is precise to the thousandth place.
- Guess where the root is: pick starting points $a$ and $b$ such that $[a, b]$ contains one root
- Until a stopping criterion is met:
  - Compute the midpoint $m$ between $a$ and $b$
  - If $f(a)$ and $f(m)$ have different signs, then $m$ is to the right of the root so $m$ becomes the new $b$ (right end of the interval)
  - If $f(m)$ and $f(b)$ have different signs, then $m$ is to the left of the root so $m$ becomes the new $a$ (left end of the interval)

When should the method stop? It should check three criteria and any one of them may stop the method:
- The interval is small enough
- The function value at mid-interval is close enough to zero
- Too many iterations have occurred (say, 1000)

Specifications

Write a script `bisection.m` to find the root of a function using the bisection method as described above. 
User input: tolerance, starting points $a$ and $b$ such that $[a, b]$ contains the root
Script output: the approximate root as well as the function value at the approximate root

The function $f(x)$ should be evaluated in a separate function file called `fun.m`. Function `fun` has one input parameter $x$ and one output argument $fx$, which is the value of the function evaluated at $x$. You may work with any function of interest to you, but keep in mind the limitations of the bisection method! For your initial program development, we suggest the simple function $f(x) = x^2 - 2$. It has exactly two roots, $\pm \sqrt{2} \approx \pm1.41$.

Consider the following situations:
- Some kind of warning message should be displayed if the method stops due to excessive iterations.
- If the user chooses an initial interval that does not contain a sign change, re-prompt the user for the starting points $a$ and $b$

Submit the files `bisection.m` and `fun.m`. 
3. What time is it?

Have you ever wondered how the analog clock is drawn on a computer screen or on some other kind of display for a digital device? In this question we’ll make only a still display, not a running clock! Write a set of Matlab functions to display a time (e.g., 2:30:40) on a clock with hour and minute hands. First, let’s see an example of the finished product:

```matlab
>> timeClock(14,30,40)
```

![Example clock display](image)

The clock is drawn in a figure window. Our clock’s hands have a “continuous pivot,” i.e., they are not restricted to the 60 individual minute marks. The hour value may be any integer in [0..24] and the minute and second values are in [0..59]. There are quite a few things that the solution will need: converting numeric time values to angles, converting angles to x-y coordinates for plotting, drawing the clock with tick marks, and of course drawing the hands to indicate the time correctly. Instead of worrying about all these things at once, practice problem decomposition, or modular design. We help you with this problem by decomposing it into sub-problems. Each sub-problem is a function that can be written and tested (mostly) independently. The three function headers and comment blocks are given below.

You will write function `timeClock`:

```matlab
function timeClock(h,m,s)
% Display the time h:m:s on a clock with hour and minute hands.
% Hour h may be in 12- or 24-hour clock format, i.e., range is [0..24].
% Minute m and second s are in the range [0..59].
```

Function `timeClock` is the main or “driver” function—it will make use of two other functions:

```matlab
function drawClockFace(hourSymbol, minsSymbol)
% Draw a clock face where the hours and minutes are marked.
% hourSymbol, minsSymbol are CHARACTERS representing the marker type
% used in a plot statement. Possibilities include:
%   '.'   'o'   'x'   '+'   '*'
% Type help plot in Matlab to see the list of color/marker/line options.
```
function [x, y] = polar2xy(r,theta)
% (x,y) are the Cartesian coordinates of polar coordinates (r,theta).
% r is the radial coordinate and theta is the angle, also called phase.
% theta is in degrees.

Function drawClockFace has been started for you—we have provided the graphics window setup and the code to draw the square clock face. Notice that the square is centered at (0,0). You need to draw the tick marks. Use different symbols for the hour and minute marks. Recall that the plot command plot(x, y, hourSymbol) will draw a marker at location (x,y). If parameter (variable) hourSymbol holds the character ‘o’ then the marker is a circle. Since the parameters are characters, when you call the function you will need to use single quotation marks to enclose the characters. For example, to get the tick marks shown in the example output, the function call to drawClockFace is:

\[ \text{drawClockFace('d', '.')} \]

\(d\) for diamond.

It is convenient to think in polar coordinates since hours, minutes, and seconds can be converted readily to degrees. Recall that a polar coordinate has two components: radial and angle (also called phase). The radial component is like the radius of a circle; the angle is measured counter-clockwise from 0°, which is the positive x-axis in the Cartesian system. For Matlab’s plot function we use the Cartesian coordinate system, so we need a conversion function polar2xy.

After the decomposition step (already done for you), you may want to start with the function that has the least or no dependence on the other functions! That would be polar2xy. You should write polar2xy for practice, but if you need help you can find that function associated with the lecture notes… Test it and make sure you know how to call it. Then work on the drawClockFace function. Make sure drawClockFace works properly, and finally work on the timeClock function. Be careful here: an angle is measured counter-clockwise from the positive x-axis, but the clock’s “angle measurement” goes clockwise (of course!) from 12 o’clock.

Submit three files: timeClock.m, drawClockFace.m, and polar2xy.m.