AMATH 301
Homework 2
Due: Friday, January 18, 2019

Method for calculating square roots

Learning goal: See an example of a problem that can be turned into a root-finding problem.

Most calculators have buttons for basic operations like division or taking the square root, but unlike long division, most people don't know how to calculate a square-root by hand. Suppose we have a positive number s, and want to calculate its square root, $\sqrt{s} = x$. We can equivalently try to find $s = x^2$, which the clever student will recognize can be rearranged to look as follows,

$$x^2 - s = 0$$

which you should recognize as a root-finding problem: find x such that $f(x) = 0$, where

$$f(x) = x^2 - s.$$

This is the Babylonian method for calculating square roots.

Newton's method

Learning goal: Be able to explain the benefits and drawbacks of Newton's root-finding method.

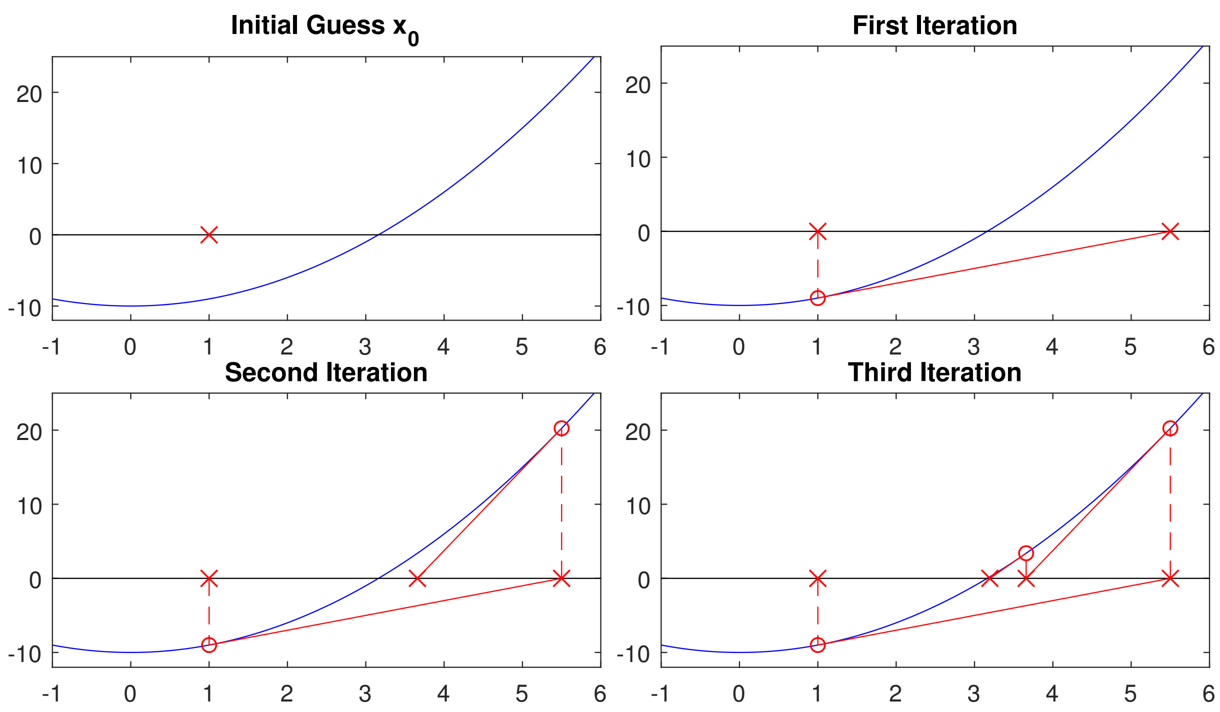
In the video lectures we saw the bisection method: an iterative root-finding method. The Newton–Rhapson method (Newton's method) is another iterative root-finding method. The method is geometrically motivated and uses the derivative to find roots. It has the advantage that it is very fast and works on problems with double (repeated) roots, where the sign of the function does not change across the root. The method also has the downside of requiring you to manually calculate the derivative. Further, the method can sometimes fail.

An illustration of Newton's method is shown in Figure 1 for the function $f(x) = x^2 - 3$. The steps are:

1. Start with a guess x_0 (red cross).
2. Go straight up (or down) to the curve (dashed red line to red circle).
3. Follow the tangent line from the point $(x_0, f(x_0))$ to the x-axis (red line).
4. Take the intersection of the tangent line and the x-axis as the new guess, x_1 (next red cross).
5. Repeat from step 1, with x_1 as the new x_0.

The formula for Newton's method is

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}. \tag{1}$$



Problem 1 (Scorelator).

Learning goal: Use the MATLAB built-in `fzero` *root-finding command.*

Before we implement our own root-finding method, let's try one of MATLAB's built-in root-finding method, `fzero`, to find the zeros of the function,

$$f(x) = xe^x - 5.$$

This is a transcendental equation that cannot be solved for in terms of elementary functions. Use a *function handle* to define this function as follows,

```
f = @(x) x.*exp(x) - 5;
```

Now MATLAB will recognize `f` as a function just like it recognizes `cos` or `exp`; for example, you can type `f(5)` to evaluate $f(5)$.

Use the `fzero` command to find the roots of $f(x)$. To do this you will need to use `f` and to supply an initial guess. Use an initial guess of $x_0 = 1$ and save the root to file as **A1.dat**.

Problem 2 (Scorelator).

Use a function handle to define the following function in MATLAB,

$$f(x) = x^2 - s,$$

for $s = 10$.

To verify that finding the root of this function yields the square-root of s, use the `fzero` command to find its zero. To do this you will need to use the function you defined above, and an initial guess x_0. Use $x_0 = 3$. Save the result that you got to file as **A2.dat**.

Problem 3 (Scorelator).

Learning goal: Implement an iterative algorithm in MATLAB with a loop.

Use MATLAB to implement Newton's method for the function $f(x) = x^2 - s$ for $s = 10$. Use the initial guess $x_0 = 1$ and a `for` loop to compute 20 iterates,

$$\begin{aligned} x_1 &= x_0 - \frac{f(x_0)}{f'(x_0)} \\ x_2 &= x_1 - \frac{f(x_1)}{f'(x_1)} \\ &\vdots \\ x_{20} &= x_{19} - \frac{f(x_{19})}{f'(x_{19})}. \end{aligned}$$

If you didn't know how to use `for` loops, you could perform two iterations manually as follows,

```
x = 1; % initial guess

% First iteration
x_new = x - f(x) / f_deriv(x); % compute new iterate
x = x_new; % update with new iterate

% Second iteration
x_new = x - f(x) / f_deriv(x); % compute new iterate
x = x_new; % update with new iterate
```

Here `f` is the MATLAB function you defined in Problem 2, and `f_deriv` is another function you should create to compute the derivative of $f(x)$.

Save the final iterate x_{20} to file as `A3.dat`.

Problem 4 (Scorelator).

Learning goal: Save the iterates of an iterative method into a 'save vector.'

In computing we often save not just the last value (final iterate) but all of the iterates $x_0, x_1, x_2, \ldots, x_{20}$ from the initial guess up to and including the final iterate.

Copy your code from the previous problem and add code to your loop to save the current iterate x_n to a *save vector* at each step (the length of the vector should grow at each step). An example of code that adds to a vector at each iteration is

```
clc; clear
x_save = [1];
for jj=1:10
    x_save(1+jj,1) = 1;
end
```

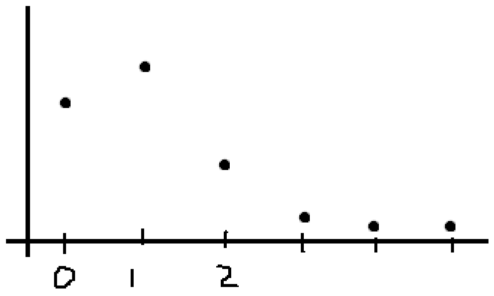
The first element of your save vector should be the initial guess x_0. Output the resulting vector of iterates in a **column vector** to the file `A4.dat`.

Problem 5 (Writeup).

Learning goal: Visually demonstrate the rate of convergence of an iterative method.

To get an idea of how quickly Newton's method converges, plot the absolute error $|x_n - x|$ of the iterates x_n from Problem 3, where $x = \sqrt{s}$ is the true solution. The error should go to zero, since the method does converge. Plot these errors on the vertical axis against iteration number, $(0, 1, 2, \ldots, 20)$ on the horizontal axis.

A crude representation of what your error trend will look like is shown below,



Add labels indicating what quantities are being plotted and a title summarizing what the plot shows.

Save your result to file with the `print` command as `newton_error.png`.

Problem 6 (Writeup).

Learning goal: Choose logarithmic scaling when data varies across many orders of magnitude.

The plot in Problem 4 should show that the error decreases, but it is hard to tell how quickly the error decreases. Change the plot to a *semilog* plot where the y-axis is shown on a logarithmic scale using the `semilogy` command (look this command up; it replaces `plot`).

Update your y-axis label to indicate that the axis measurements are on a log-scale.

Save your result to file with the `print` command as `newton_error_log.png`.

Problem 7 (Scorelator).

Learning goal: Implement a stopping condition in an iterative algorithm.

From the previous problem, it is apparent that the iterates converge well before the twentieth iteration. The extra iterations are wasted effort and it would be better if the loop stopped one the iterates were close enough to the right answer.

Copy your code from Problem 4. Add a stopping condition so that the loop stops when the absolute value of the function $f(x_n)$ is smaller than 10^{-8}. You can use the `abs` command to compute the absolute value, and an `if` statement to test the condition. This test should be the last thing to appear in your loop.

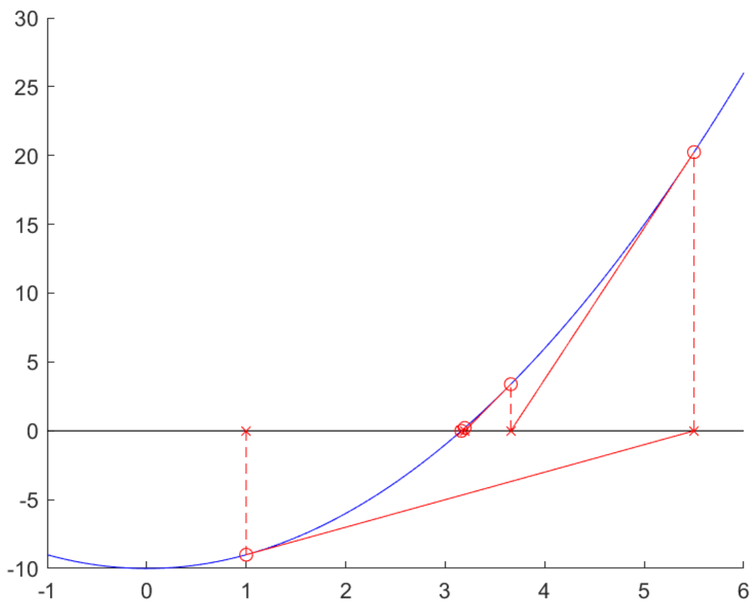
Your save vector should have significantly fewer than 20 iterates due to the stopping condition you added. Output the resulting save vector to the file `A5.dat`.

Problem 8 (Writeup).

Learning goal: Combine basic plot commands to produce a complicated figure.

Learning goal: Visually illustrate a mathematical method.

Write code to produce a graphical depiction of Newton's method like those shown in Figure 1. A reproduction like the one you should obtain is shown below



This plot may look complicated, but it can be broken down many simple `plot` commands.

Begin your plotting code by clearing the figure and plotting the function and a horizontal line for the x-axis:

```
clf
xs = linspace(-1.5,1.5);
plot(xs,0*xs,'k-'); % x-axis
hold on
plot(xs,f(xs),'b-');
```

Then add your code from Problem 4. Modify your code by adding a plot command to draw a red cross at the initial guess $(x_0, 0)$. Then add plot commands within the loop to do each of the following at each iteration:

1. A dashed red line connecting the points $(x_n, 0)$ and $(x_n, f(x_n))$.
2. A red circle at the point $(x_n, f(x_n))$.
3. A red line connecting the points $(x_n, f(x_n))$ and $(x_{n+1}, 0)$.
4. A red cross at the point $(x_{n+1}, 0)$.

Save your figure to file as `newton_babylon_iterations.png`.