Numerical Methods with MATLAB

University of Exeter

April 25, 2023

1 Introduction

This workshop is intended to give an overview to numerical methods to find roots to equations. The aims are the following:

- describe what numerical methods are,
- explain why mathematicians employ them, and
- give you an opportunity to try one out using MATLAB.

No coding experience is necessary for this workshop, and all solutions are available.

2 Numerical Methods

Engineering problems are usually solved using mathematical modelling, often involving assumptions and simplifications. However, even simplified models are often too complex to be solved using direct mathematical techniques (analytical methods). Some examples include:

- They may be non-linear
- The geometry may be complex
- The problem may be too "big", with too many inter-related equations

We need ways of getting answers, even if only approximate. Given the complexity of the problems, we also need efficiency and understanding and/or control of accuracy.

In order to solve this kind of engineering problems, often we need to calculate an approximate solution, in a variety of ways. We can make use of computers and their speed and power of computation. At the end of the process, we get an approximate solution for a variety of reasons:

- Because we approximated the problem (simplifying assumptions)
- Because we used an approximate method
- Because the representation of numbers and number handling in computers only gives approximate answers

In this session, we will focus on finding the roots of equations.

3 Motivation

Consider the following equation:

$$f(x) = x^2 + x - 6. \tag{1}$$

If we were interested when this equation was equal to zero, it would be fairly easy to determine that the roots are $x = 3$ and $x = -2$. This is an example where we can solve for x **analytically**. However, if we sought the roots of

$$f(x) = x^2 \sin(x) - e^x, \tag{2}$$

it would be much more difficult. Sketching this equation may help somewhat (see figure 1), but it does not tell us *exactly* where the root is. This is where **numerical** methods are much more helpful. Numerical methods help mathematicians, physicists and engineers solve problems that cannot be solved analytically. For example, the Navier-Stokes equations describe how fluids behave and are notoriously challenging to solve.

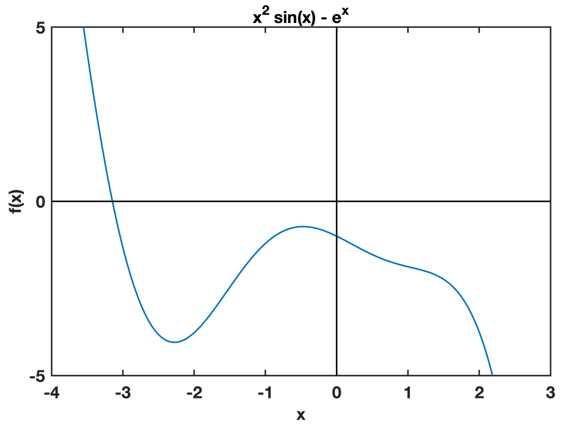


Figure 1: The function $(x^2 \sin(x) - e^x)$ within the bounds $-4 \leq x \leq 3$, showing a single root

4 Bisection method

One method of finding a root of an equation is *bisection.* When there is a root, there typically is a change of sign (positive to negative). Looking at our example equation in figure 1, we can see that the function has a root about $x = -3$.

A bisection algorithm is shown in Algorithm 1. It take two points, x_0 and x_2, and finds the midpoint, x_1. If there is a change of sign when we evaluate these values, there is a root within these boundaries.

Q: What issues are there with this algorithm?

Algorithm 1 Bisection method for finding roots

```
Require: Initial points $x_0$, $x_2$, function $f(x)$ to find roots for.
 1: $x_1 \leftarrow \frac{x_0+x_2}{2}$                ▷ Find the midpoint between $x_0$ and $x_2$.
 2: Evaluate $f(x_0)$, $f(x_1)$
 3: if $\frac{f(x_0)}{f(x_1)} < 0$ then:            ▷ If there is a change of sign
 4:     $x_2 \leftarrow x_1$
 5:     return to step 1
 6: end if
 7: Evaluate $f(x_1)$, $f(x_2)$
 8: if $\frac{f(x_1)}{f(x_2)} < 0$ then:            ▷ If there is a change of sign
 9:     $x_0 \leftarrow x_1$
10:     $x_1 \leftarrow x_2$
11:     return to step 1
12: end if
13: return No roots found between $x_0$ and $x_2$
```

5 Using MATLAB

Before we get onto the bisection algorithm, let's begin by showing the basics of MATLAB. This software allows the user to write code, but essentially MATLAB acts like a glorified calculator. If you open up MATLAB, you will see the screen shown in figure 2. If the **Editor** is not shown, click the yellow '+' circled in red.

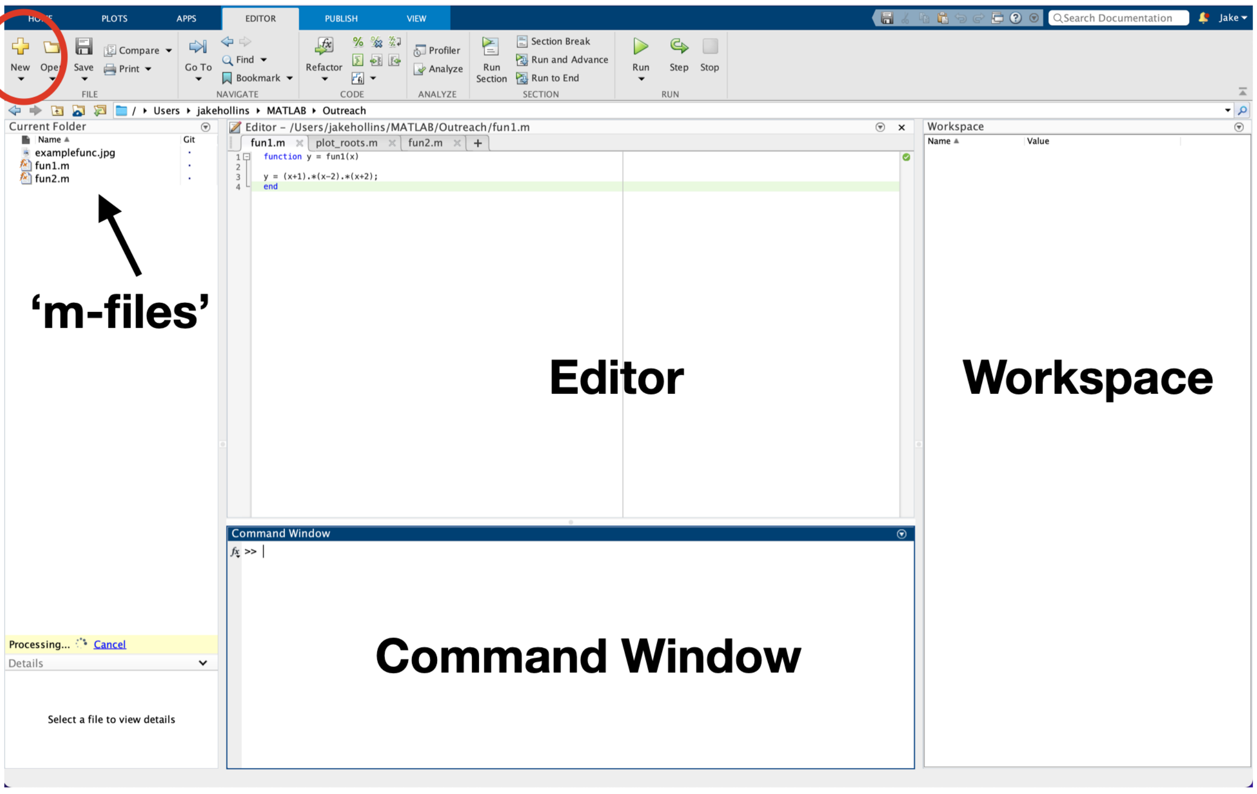
Once the Editor is available, we are going to make our first function. Copy the following code and save as 'fun1' (top-left corner).

```
function y = fun1(x)
    y = (x+1).*(x-2).*(x+2);
end
```

This is the polynomial function $(x + 2)(x + 1)(x - 2)$, which will have roots $[-2, -1, 2]$.

Lets now make a variable x. We can write this is the **Command Window**, and press enter:

```
x = -4:0.01:4
```



```
function y = fun1(x)

y = (x+1).*(x-2).*(x+2);
end
```

Figure 2: MATLAB Interface

What this command does is it will make a list of numbers increasing from -4 to 4 in increments of 0.01. You can view this variable in the **Workspace** in the top-right side of the screen. The Workspace is where any saved variables can be viewed. Alternatively, you can type "x" into the Command Window and press enter.
We are now ready to plot our function:

```
plot(x,fun1(x))
```

This should now look like figure 3. If you would like the axes to be shown, type the following into Command Window:

```
yline(0)
```

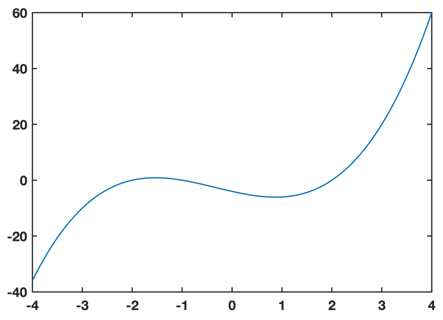


Figure 3: Plot of polynomial function 1

Try creating and plotting the following functions.
When multiplying (*) or dividing (/), make sure to use a dot (.) just before the operation.

- $x^2 - 3x - 2$
- $\sin(x) + x\cos(2x)$
- $x^2\sin(x) - e^x$

Feel free to also make your own!

6 Applying Bisection Method

Let's now try applying the bisection method to our functions. Don't worry if it doesn't look the same - the code will still work!

```
function [root,error,iterations] = find_root(initial_x,fun)
error = 1;
tolerance = 0.0000001;
x = initial_x;
delta_x = 1.0001;
iterations = 0;
while error > tolerance && iterations < 10000
    iterations = iterations+1;
    y0 = fun(x);
    y1 = fun(x+delta_x);
    y2 = fun(x+2*delta_x);

    if y0/y1 < 0
        delta_x = delta_x/2;
        error = abs(delta_x/(x+delta_x));
    elseif y2/y1 < 0
        delta_x = delta_x/2;
        error = abs(delta_x/(x+delta_x));
        x = x+delta_x;
    else
        x = x+2*delta_x;
    end
end

root = x;
end
```

The input of this function is an initial x to start the algorithm (initial-x) and the function you wish to find the roots for ('fun'). To insert your function, you will need to use the '@' handle:

```
[root, error, iterations] = find_root(-4,@fun1)
```

You may also notice that an 'error' is calculated in each step, or 'iteration'. Keep an eye on these when we compare to the Newton-Raphson method!

For each of the different functions you made, try different initial guesses and see if you can find all the roots.

7 Newton-Raphson Method

This is the most widely used of all root-locating formulas. One way of deriving the Newton-Rhapson formula is by graphical derivation:

The slope $f'(xi)$ from figure 4 can be written as:

$$f'(x_i) = \frac{f(x_i) - 0}{x_i - x_{i+1}} \tag{3}$$

Rearranging, we have the Newton-Raphson formula:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \tag{4}$$

This equation assumes we know the gradient of the function. If this is not known, then we can *approximate* by taking a small step, Δ, in either direction and calculating the gradient (for example $\Delta = 0.001$:

$$f'(x_i) = \frac{f(x + \Delta) - f(x - \Delta)}{2\Delta} \tag{5}$$

The below code is a working Newton-Raphson method.

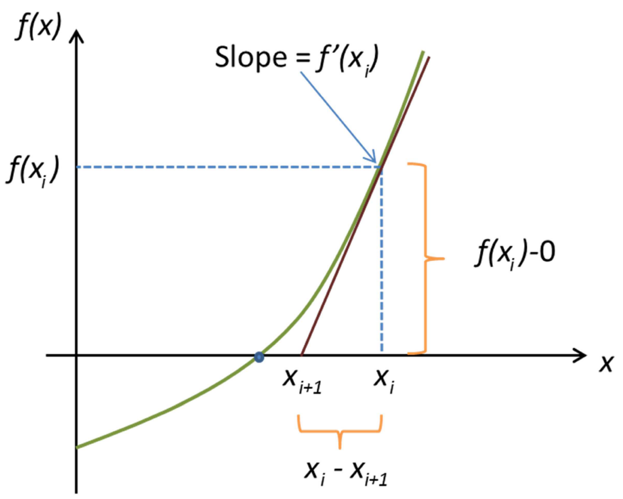


Figure 4: Illustration of Newton Raphson method.

```
function [root,error,iterations] = NewtonRaphson(initial_x,fun)

error = 1;
tolerance = 0.00001;
x = initial_x;
delta_x = 0.001;
iterations = 0;

while error > tolerance && iterations <100
    iterations = iterations + 1;
    est_grad = (fun(x)-fun(x+delta_x))/(delta_x);
    x_new = x + fun(x)/est_grad;
    error = abs(x-x_new);
    x = x_new;
end

root = x;
end
```

For each of the different functions you made, try different initial guesses and see if you can find all the roots.

Q: How do the bisection and Newton-Raphson methods compare?

8 Multiple Restarts

The method can fail to find a root at all, or can fail to find the root we want, or can get trapped in an infinite cycle. This usually happens either if we are near an inflection point, or near local max/min points. Figure 5 shows four examples where the Newton-Raphson method will fail. Sometimes we may need to try different values to find the roots.

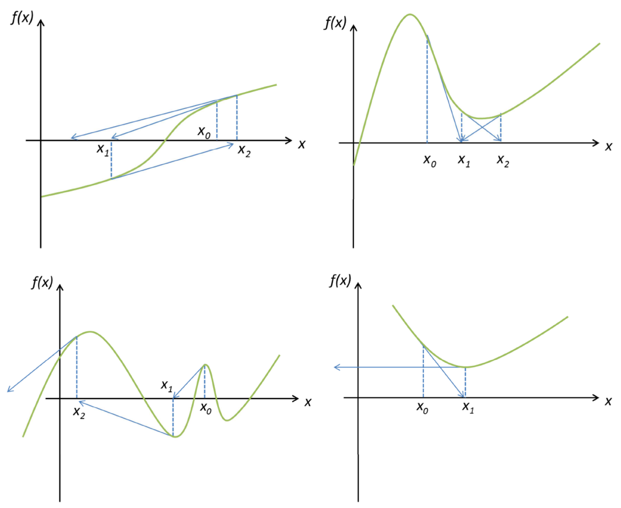


Figure 5: Some examples of where the Newton-Raphson method fails.

Trying different initial values of x is tedious, and we may wish to automate the process. One way we can do this is to have multiple values tried sequentially. We can use a random number generator to produce some initial values, run the algorithm and output only unique values. The code below is an example of this, and it plots the function and its roots. Its first input is the function (remember the '@'), the second is the method, which you can swap between the bisection and Newton-Raphson method, and final input is number of restarts. As the number of restarts increeases, computational time will increase. Figure 6 shows the function and roots for

$$f(x) = (x^2 - 8)(x - 9)(2x + 3)(3x - 5); \tag{6}$$

using 100 restarts.

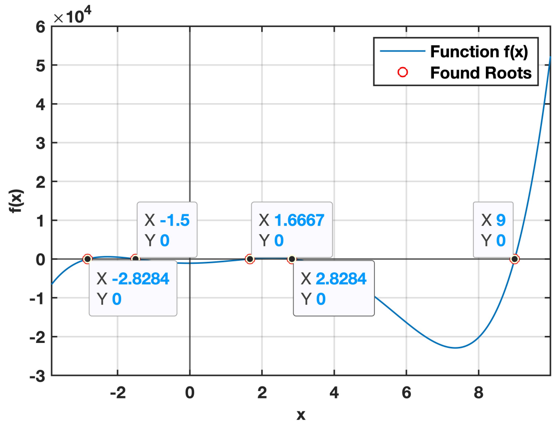


Figure 6: Example of function and plots

```
function AllRoots(fun,Method,restarts)
% fun should be the function to find roots for. Use '@'
% Method is the method to find roots. This is either @NewtonRaphson or @find_roots
% restarts is the number of restarts to use.


randomStarts = -5 + 10*rand([1,restarts]);
Roots = [];

for i = 1:restarts
    [root,~,iterations] = Method(randomStarts(i),fun);

    if iterations == 100
        root = [];
    end

    Roots = [Roots,root];


end
Roots = unique(round(Roots,4));

x = min(Roots)-1:0.01:max(Roots)+1;

figure()
fig = plot(x,fun(x),LineWidth=1);
hold on
fig = scatter(Roots,zeros([1,length(Roots)]),'r','o','LineWidth',1);
xlabel('x')
ylabel('f(x)')
xline(0)
yline(0)
xlim([min(x),max(x)])
for r = 1:length(Roots)
    datatip(fig,Roots(r)+0.1,-0.5);
end
legend('Function f(x)','Found Roots')
grid on
end
```

Try the following:

1. **Compare how many roots each root-finding algorithm can find for a given number or restarts.**
2. **Try more complex functions, such as products of sine or cosine.**
3. **If feeling confident, write a code that plots how the error reduces throughout the iterations. Hint: replace 'plot' with 'semilogy' which plot the log plot of the errors.**