Root_Finding

November 7, 2022

1 The naive approach

In this notebook we want to find the zeros of a function, meaning we want to find x such that f(x)=0, whenever we want to take a look at roots or zeros it's a good idea to visualize the function in the neiborhood we are looking for zeros in. Let us consider one particular example. Let

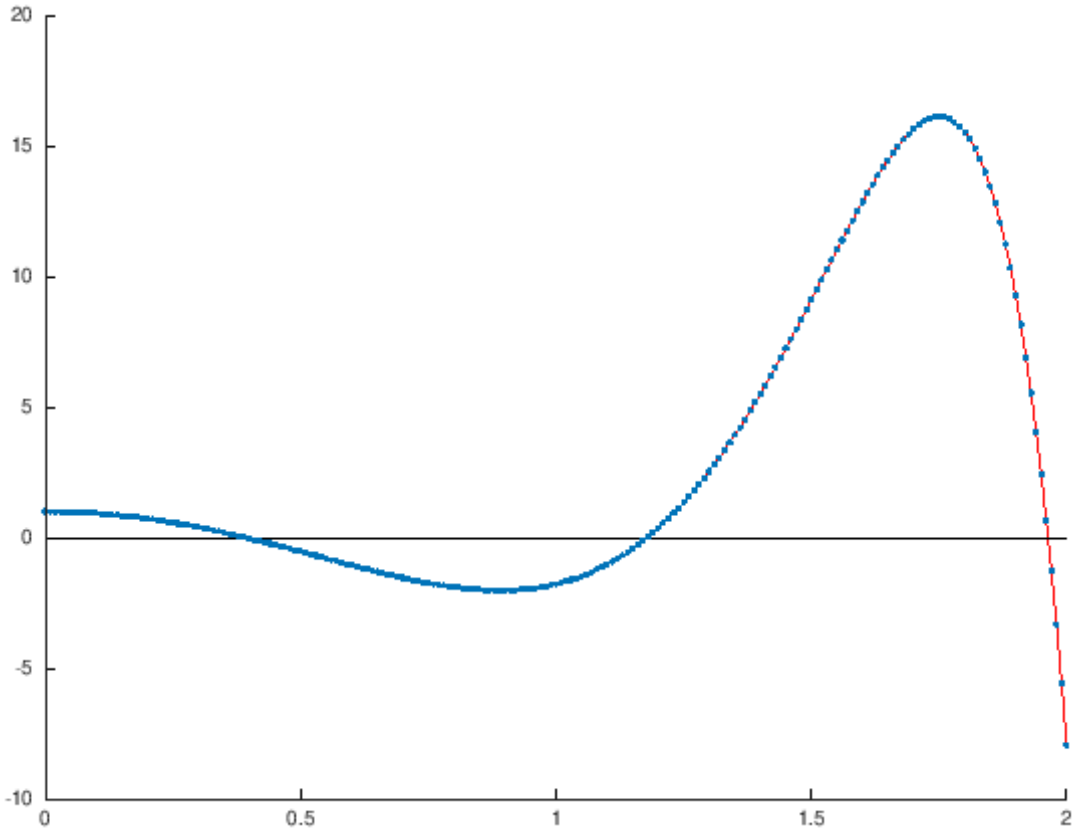
$$f(x) = e^{x^2} cos(4x) \tag{1}$$

Let us go and try to find the zeros of said function, as a first step let us define the function, our region of interest (where we'll look for zeros) and plot it

[1]:
```
function x =f(x)
% This function calculates e^{x^{2}} cos(4x), the input x can be a scalar or an␣
 ↪array
    x=exp(x.^2).*cos(4*x);
endfunction
```

[2]:
```
x=-0:0.01:2;# We define the interval where will be looking for zeros
```

[3]:
```
y=f(x);
hold on,plot(x,y,'r')
line("xdata",[0,2], "ydata",[0,0], "linewidth", 3)
plot(x,y,'.')
```



From the plot we can tell that there seems to be 3 zeros in this region, one around 0.4, one around 1.2 and one around 2 A brute force algorithm is to run through all points in our region of interest and check if one point is below the x axis and if the next point is above the x axis, or the other way around. if that happens then y is zero between those 2 points

At each stage we want to check if $y_i < 0$ and $y_{i+1} > 0$ (or the other way around). for both checks is $y_i y_{i+1} < 0$. If so, then y is 0 between x_i and x_{i+1}. Assuming a linear variation of the funcion f between x_i and x_{i+1}, we have the approximation

$f(x) \approx \frac{y_{i+1}-y_i}{x_{i+1}-x_i}(x-xi) + y_i$

We may solve for x when $f(x) = 0$ such that we obtain:

$x = x_i - \frac{x_{i+1}-x_i}{y_{i+1}-y_i} yi$

Then for the algorithm we simply check whether the condition is fulfilled, if it is then we compute x using our grid and stop searching for roots. A naive implementation of the method can be found below

[4]:
```
x = linspace(0, 0.5, 100001);
y = f(x);

root = NaN;  % Initialization value
```

```
for i = 1:(length(x)-1)
    if y(i)*y(i+1) < 0
         root = x(i) - (x(i+1) - x(i))/(y(i+1) - y(i))*y(i);
         break;  % Jump out of loop
    end
end
if isnan(root)
    fprintf('Could not find any root in the interval [%g, %g]\n', x(1), x(end));
else
    fprintf('Found a root at x=%g\n', root);
end
```

```
Found a root at x=0.392699
```

Task: Generalize this function so that it finds all roots on the given interval, you may use the template for the function below or define it from scratch, test it on f on [0,3]

[5]:
```
function all_roots = brute_force(f,a, b, n)
    x = linspace(a, b, n);
    y = f(x);
    roots = [];
    for i = 1:(n-1)
        if  < 0 % complete the comparison
            root = ;% complete this function
            roots = [roots; root];
        end
    end
    if isempty(roots)
    fprintf('Could not find any root in the interval [%g, %g]\n', x(1), x(end));
    else
    fprintf('Found a root at x=%g\n', roots);
    end
    all_roots = roots;
end
```

[6]:
```
brute_force(@f,0,3, 10000)
```

```
Found a root at x=0.392699
Found a root at x=1.1781
Found a root at x=1.9635
Found a root at x=2.74889
ans =

   0.39270
   1.17810
   1.96350
   2.74889
```

2 Newton-Raphson method

The geometrical intuition about the Newton Raphson method is to start at a given point, an initial guess and follow the tangent line to the curve at that point, the next guess is the point at which the tangent line is zero. We keep iterating this until we find $f(x) \approx 0$ each iteration getting us closer to such x, let us now briefly visualize this with the help of octave on

$$f(x) = x^2 - 4 \tag{2}$$

[7]:
```
function result =f2(x)
    result=x.^2 .- 4;
end
```

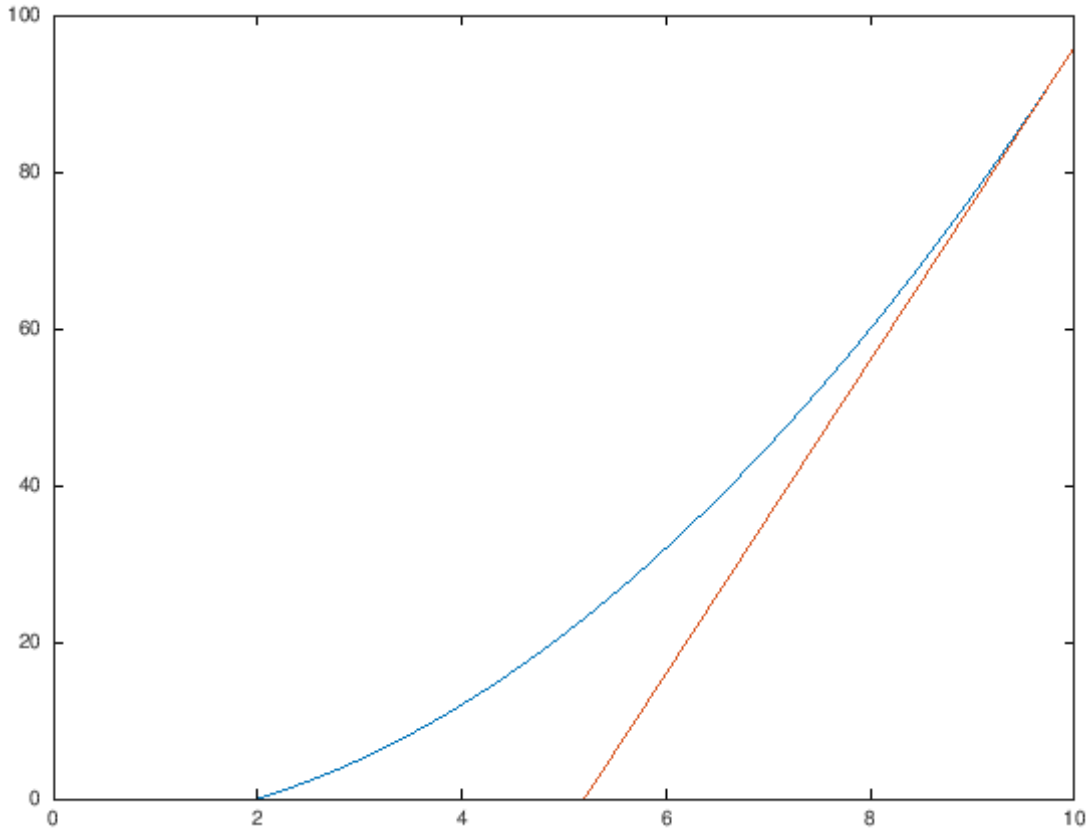
The slope of the tangent line is equal to the derivative of the curve, so for this method we need the derivative of f(x) (we may also estimate it numerically using the definition of derivative)

[18]:
```
function result = dfdx(x)
    result=2*x;
end
```

[9]:
```
function tangent=t(x,y)
tangent=dfdx(y).*(x-y) .+f2(y);
end
```

Let us say that the initial guess for our root is $x_0 = 10$, then let us plot the curve and its tangent line

[10]:
```
x=linspace(-10,10,100);
plot(x,f2(x))
hold on,plot(x,t(x,10))
xlim([0,10])
ylim([0,100])
```

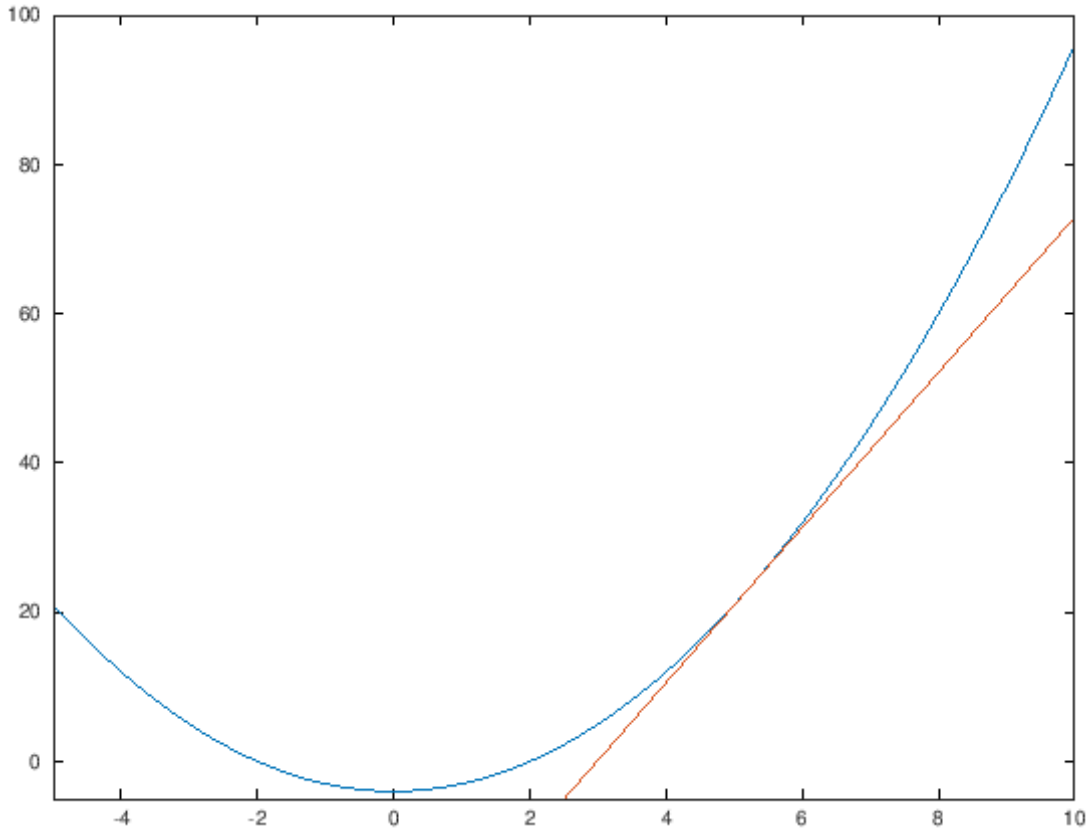


[11]:
```
z=brute_force(@(x) t(x,10),0,10,100) # point at which the tangent line is zero
```

```
Found a root at x=5.2
z =  5.2000
```

we continue with $x_1 = 5.2$

[12]:
```
x=linspace(-10,10,100);
#yTangentLine = slope * (x - xTangent) + yTangent;
plot(x,f2(x))
hold on,plot(x,t(x,5.2))
xlim([-5,10])
ylim([-5,100])
```



And continue this process as we get closer and closer to the solution

Now that we have some idea about the geometrical intuituon let us actually apply the method, now we previously defined the tangent line at the point which was given by:

$$T(x) = f'(x)(x - x_0) + f(x_0) \tag{3}$$

Now, we want to see where the tangent is 0 such that

$$x = x_0 - \frac{f(x_0)}{f'(x_0)} \tag{4}$$

where x_0 is our guess, we stop the method whenever $|f(x)| < \epsilon$, where ϵ is some small number we choose arbitrarily as threshold (remember that due to floating point error we may never get 0 so checking $|f(x)| = 0$ would be a bad idea)

[13]:
```
function [result,c] = naive_Newton(f,df,starting_value,eps)
    x = starting_value;
    c=0;
    while abs(f(x)) > eps
        x = x - f(x)/df(x);
```

```
        c+=1; # count the number of iterations
    end
    result = x;
end
```

[14]:
```
[result,c]=naive_Newton(@f2,@dfdx,0.5,eps=0.000001)
```

```
result =  2.0000
c =  6
```

[15]:
```
f2(result)
```

```
ans =    1.0125e-13
```

Task: Now, while newton's method is good it doesn't guarantee convergence and some choices of initial guess might go terribly wrong, for example try to find the roots of $x^2 + 1$ using 1.2 as an initial guess.

Does it ever happen? why?

What do you need to modify so that our implementation of the method finds those roots?

Modify the method so that it would work with 1.2 as an initial guess

Hint: Think about the kind of numbers you are using as well as the numerical standard used to represent them

Task Use Newtons method on tanh(x), use $x_0 = 1.08$, now use $x_0 = 1.09$

did the results change? If so explain what happened?

3 Bisection Method

From the previous task we can guess that in certain situations Newton's method may fail, so it does not guarantee that a solution will be found (When indeeed there is one). The brute force method will indeed find it, so it's more reliable. However it samples all of the points in our grid so it is not too efficient. The bisection method is simply a modification of the brute force method which makes it more efficient by reducing the number of sampled points, however it is still less efficient than Newton's method.

The main difference between this method and the brute force is to divide the interval into two equal parts, one on the left and one on the right of the midpoint (x_M). by evaluating the sign of $f(x_M)$, we can inmediatly know whether a solution exists on the right or the left of XM

We then proceed repeat this with half the interval only unless $|f(x)| \approx 0$ in which case a solution is found, so the only difference with respect to the brute force is that we check whether the solution may be on a subinterval and then reduce our search space.

Task: Complete the function below, to compute the bisection method or define your own

[16]:
```
function [midpoint, counter] = bisection(f, x_L, x_R, eps)
    if f(x_L)*f(x_R) > 0
```

```
        fprintf('Error! Function does not have opposite\n');
        fprintf('signs at interval endpoints! \n')
        fprintf('no roots seem to be there in this interval')
        exit(1)
    end
    x_M = (x_L + x_R)/2.0;% midpoint
    f_M = f(x_M);
    c = 1;
    while abs(f_M) > eps
        left_f = f(x_L);
        right_f = f(x_R);
        if > 0   % same sign no root on the left, Complete this
            x_L = x_M;
        else
            x_R = x_M;  % same sign no root on the right
        end
        x_M = ; % new midpoint Complete this line
        f_M = f(x_M);
        c = c + 1;
    end
    midpoint = x_M;
    counter= c;
end
function bisection_method(f,eps,a,b)
    [solution, no_iterations] = bisection(f, a, b, eps);
    if solution <= b   % Solution found
        fprintf('Number of function calls: %d\n', 1+2*no_iterations);
        fprintf('A solution is: %f\n', solution);
    else
        fprintf('Abort execution.\n');
    end
end
```

[17]:
```
bisection_method(@f2,1e-6,0,5)
```

```
Number of function calls: 47
A solution is: 2.000000
```

Task: Compare the number of iterations for the different methods, is there a big time difference between them ? could the bisection method as explained here work for expressions like $f(x) = x^2+1$ if so what do you need to modify?

Hint: In case you need some help to know wheter bisection can be used look at this video