SM-2302: Software for Mathematicians

Matlab4: Root-finding in MATLAB

Dr. Huda Ramli

Mathematical Sciences, Faculty of Science, UBD

huda.ramli@ubd.edu.bn

Semester I, 2023/24

Root-finding in MATLAB

In this chapter, we will focus on the problem of solving an equation:

$$\boxed{f(x) = 0}$$

Let's turn to finding roots, i.e. the points x_* such that $f(x_*) = 0$.
There are three methods of root-finding in Calculus:

1. The bisection method
2. Newton's method
3. The secant method

Newton's method

The basis for Newton's method is approximation of a function by its linearization at a point:

$$f(x) \approx f(x_0) + f'(x_0)(x - x_0) \tag{1}$$

Since we wish to find x so that $f(x) = 0$, then

$$x \approx x_0 - \frac{f(x_0)}{f'(x_0)}$$

Defining a sequence $\{x_0, x_1, x_2, \ldots\}$ leads to the **Newton iteration**

$$\boxed{x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}} \tag{2}$$

Example 1

Write a `function` program that does n steps (iterations) of Newton's method.

```
function x = mynewton(f,f1,x0,n)
% Solves f(x) = 0 by doing n steps of Newton's method starting at x0
% Inputs:
%     f - the function
%     f1 - its derivative
%     x0 - starting guess, a number
%     n - the number of steps to do
% Output: x - the approximate solution

x = x0; % set x equal to the initial guess x0
   for i = 1:n
         x = x -f(x)/f1(x)     % Newton's formula
   end
```

Don't forget to save the program as `mynewton.m` to match the `function` name.
In the command window,

- set to print more digits: >> format long
- set to not print blank lines: >> format compact

Now, let's define a function: $f(x) = x^3 - 5$ and then call `mynewton` on this function:

```
f = @(x) x^3 - 5
f1 = @(x) 3*x^2  % its derivative
x = mynewton(f,f1,0.2,10)
```

By simple algebra, the true root of this function is $\sqrt[3]{5}$. How close is the program's answer to the true value?

Convergence

Newton's method converges rapidly when $f'(x_*)$ is nonzero and finite, and x_0 is close enough to x_* that the linear approximation (1) is valid.

(i) For $f(x) = x^{1/3}$ we have $x_* = 0$ but $f'(x_*) = \infty$. What happens when we enter the following code?

```
f = @(x) x^(1/3)
f1 = @(x) (1/3)*x^(-2/3)
x = mynewton(f,f1,0.1,10)
```

(ii) For $f(x) = x^2$ we have $x_* = 0$ but $f'(x_*) = 0$. What happens when we enter the following code?

```
f=@(x)x^2
f1 = @(x) 2*x
x = mynewton(f,f1,1,10)
```

Error control

- There are a few different ways to measure the error of root-finding approximation, the most direct method is the error at step n: $e_n = x_n - x_*$, where x_n is the n^{th} approximation and x_* is the true value.

- However we usually do not know the value of x_*, so it is more practical to measure how close the equation is to be satisfied, i.e. how close $y_n = f(x_n)$ is to 0.

- We often use the absolute value of the **residual** quantity: $r_n = f(x_n) - 0$ as a measure of how close the solution is:

$$|r_n| = |f(x_n)|$$

Example 2

Incorporate a certain tolerance for $|r_n| = |f(x_n)|$ into our Newton method program in Example 1 using an `if ... end` statement.

(a) Call the Newton's method program for the function $f(x) = x^3 - 5$, with $n = 3, x_0 = 2$ and check for the following tolerance values:

(i) $\text{tol} = 0.01$

(ii) $\text{tol} = 10^{-10}$

(b) Modify the `mynewton` program to iterate until the residual $|r_n| = |f(x)| = |y|$ is small enough. *(Hint: use the `while ... end` loop)*

```matlab
function x = mynewton(f,f1,x0,n,tol)
% New Inputs:
%     tol - desired tolerance, prints warning if |f(x)|>tol
x = x0;
    for  i = 1:n
            x = x -f(x)/f1(x)     % Newton's formula
    end
    r = abs(f(x))
    if  r > tol
        warning('The desired accuracy was not attained')
    end
end
```

(a) (i)
```
f = @(x) x^3-5
f1 = @(x) 3*x^2
x = mynewton(f,f1,2,3,0.01)
```
(ii)
```
f = @(x) x^3-5
f1 = @(x) 3*x^2
x = mynewton(f,f1,2,3,1e-10)
```

(b) add the following syntax in the program:

```
x = x0; y = f(x);
i=0;
while abs(y) > tol & i < 1000
    x = x - y/f1(x);
    y = f(x);
    i = i+1;
end
```

Bisection method

Suppose that $c = f(a) < 0$ and $d = f(b) > 0$. If f is continuous, then $x_* \in [a, b]$. The bisection method steps are follows:

- Compute the midpoint: $\boxed{m = (a + b)/2}$
- Determine if $x_* \in [a, m]$ or $x_* \in [m, b]$, by checking the sign of $f(m)$:
  - if c and $f(m)$ have the same sign then $x_* \in [m, b]$
  - if c and $f(m)$ have the different signs then $x_* \in [a, m]$
- Continue to subdivide the new interval and stop if the length of the interval becomes sufficiently small.

Knowing the current interval of the root x_* allows us to determine the maximum error, i.e. half of the length of the current interval $[a, b]$:

$$\text{Absolute error} = |x - x_*| < \frac{b - a}{2}$$

where x is the center point between current a and b.

Example 3

Write a function program `mybisect.m` that does n iterations of the bisection method and returns the final value and the maxium possible error.

```
function [x e] = mybisect(f,a,b,n)
% Does n iterations of the bisection method for a function f
% Inputs:  f -- a function
%          a,b -- left and right edges of the interval
%          n -- the number of bisections to do
% Outputs: x -- the estimated solution of f(x) = 0
%          e -- an upper bound on the error
% evaluate at the ends and make sure there is a sign change
c = f(a); d = f(b);
if  c*d > 0.0
    error('Function has same sign at both endpoints.')
end
disp('         x                        y')
for i = 1:n
    % find the middle point and f(m)
    m = (a + b)/2;
    y = f(m);
    disp([          m                        y])
    if y == 0.0  % solve the equation exactly
```

```
            a = m;
            b = m;
        break % jumps out of the for loop
        end
        % decide which half to keep, so that the signs at the ends differ
        if c*y < 0
            b=m;
        else
            a=m;
        end
    end
    % set the best estimate for x and the error bound
    x = (a + b)/2; e = (b-a)/2;
```

Locating the roots

Recall that both the bisection and Newton's method require starting points:

- Bisection method requires two points a and b that have a root between them.
- Newton's method requires one point x_0 which is reasonable close to a root.

One way to determine these starting points, is to plot the function and see approximately where the graph crosses zero.

Other situations where we need to solve the equation more than once, we can simply check for sign changes of the function at fixed number of points inside the interval:

```
function [a,b] = myrootfind(f,a0,b0)
% Looks for subintervals where the function changes sign
% Inputs: f -- a function
%         a0 -- the left edge of the domain
%         b0 -- the right edge of the domain
% Outputs: a -- an array, giving the left edges of the
%               subintervals
%          b -- an array, giving the right edges of the
%               subintervals

n = 1001; % number of test points to use
a = [];   % start empty array
b = [];

% split the interval into n-1 intervals and evaluate at the break points
x = linspace(a0,b0,n);
y = f(x);

% loop through the intervals
```

```
for i = 1:(n-1)
    if y(i)*y(i+1) <= 0 % The sign changed, record it
        a = [a x(i)];
        b = [b x(i+1)];
    end
end
if size(a,1) == 0
    warning('no roots were found')
end
```

To see this program in action, enter the following:

```
f = @(x) sin(x)-2.*x.^4 + 0.5;
x = -1:.01:1; y = f(x);

plot(x,y,'b', x,zeros(length(x)),'k')  % see that there are two roots
[a,b] = myrootfind(f,-1,1)  % observe that it finds intervals of the two ...
    roots
```

The `roots` and `fzero` functions

MATLAB provides two built-in root finding methods:

`fzero`	find a root of $f(x) = 0$, where $f(x)$ is a general function of one variable
`roots`	find a root of $p(x) = 0$, where $p(x)$ is a polynomial

Example 4

(a) Find the root of $f(x) = x - e^{-x}$ using `fzero(f,x0)` with a starting interval of $[0, 1]$. Then, try with a single initial guess.

(b) Find the roots of $x^4 - 5x^2 + 4 = 0$ using `roots(p)`.

The Secant Method

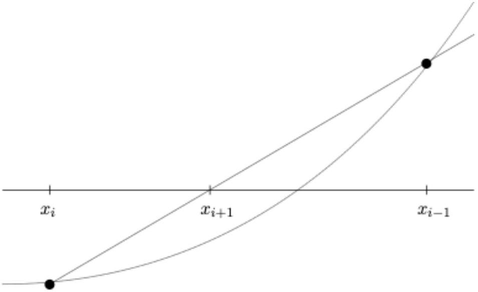
The secant method requires two initial approximations x_0 and x_1, both reasonably close to x_*. Then, we determine $y_0 = f(x_0)$ and $y_1 = f(x_1)$, to find the equation of the connecting (secant) line:

$$y - y_1 = \frac{y_1 - y_0}{x_1 - x_0}(x - x_1).$$

Setting $y = f(x) = 0$ and solve for x, we get the repeated iteration:

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

with $y_i = f(x_i)$.



```
function x = mysecant(f,x0,x1,n)
% Solves f(x) = 0 by doing n steps of the secant method
% starting with x0 and x1.
% Inputs: f -- the function
%         x0 -- starting guess, a number
%         x1 -- second starting guess
%         n -- the number of steps to do
% Output: x -- the approximate solution

y0 = f(x0); y1 = f(x1);
for i = 1:n
    x = x1 - (x1-x0)*y1/(y1-y0); % secant formula
    y = f(x); % y value at the new approximate solution

    % Update numbers to get ready for the next step
    x0 = x1; y0 = y1;
    x1 = x; y1 = y;
end
```

The *Regula Falsi* (False Position) method

- The *Regula Falsi* method is a combination of the secant method and bisection method.
- Starting with two approximations a and b for which $f(a)$ and $f(b)$ have different signs and following the secant line to get a new approximation:

$$x = b - \frac{b-a}{f(b)-f(a)} f(b)$$

- Check the sign of $f(x)$; if it is the same sign then $a = x$ and otherwise (different signs) then $b = x$.
- In general, either $a \to x_*$ or $b \to x_*$ but not both, so $b - a \not\to 0$.

Method comparison

Method	Starting point(s)	Convergence
Newton's	x_0	converges rapidly if x_0 is close enough to x_* and $f'(x)$ is required
Bisection	$[a, b]$ with different signs of $f(a)$ and $f(b)$	always works because the root must be located within starting interval.
Secant	x_0 and x_1	good starting interval required, little slower than Newton's method but can be used when formula $f(x)$ is not known.