Numerical Analysis using Matlab

Ahmed Aboemeira

Omar Barakat

Omar Yousri

Karim Waguih ElAzzouni

Michael Wahid Azmy

Mohamed El Feky

Contents

LINEAR EQUATIONS

GAUSS ELIMINATION

CODE SNIPPETS

```
function [x, t] = GaussPivot(A, b)
tic;
n = length(A);
nb = n+1;
Aug=[A b];

% forward elimination
for k = 1 : n-1
    % partial pivoting
    [~,i] = max(abs(Aug(k:n, k)));
    ipr = i + k - 1;

    if ipr ~= k
        Aug([k ipr],:) = Aug([ipr k],:);
    end

    for i = k+1:n
        factor = Aug(i, k)/ Aug(k, k);
        Aug(i, k:nb) = Aug(i, k:nb) - factor*Aug(k, k:nb);
    end
end

if Aug(n,n) == 0
    disp('No unique solution');
else
    % back substitution
    x = zeros(n, 1);
    x(n) = Aug(n,nb) / Aug(n,n);
    for i = n-1:-1:1
     x(i) = (Aug(i,nb) - Aug(i,i+1:n)*x(i+1:n)) / Aug(i,i);
    end
end

t = toc;
end
```

GAUSS-JORDAN

CODE SNIPPETS

```
function [x, t] = GaussJordan(A, b)
tic;
n = length(A);
nb = n+1;
Aug=[A b];

k = 1;
for k = 1 : n
    i = k;
    while (i <= n && Aug(i, i) == 0)
        i = i + 1;
    end

    if (i > n)
        break;
    end

    if (i ~= k)
        Aug([i k], :) = Aug([k i], :);
    end

    toDiv = Aug(k, k);
    Aug(k, :) = Aug(k, :) / toDiv;

    for i = 1 : k-1
        Aug(i, :) = Aug(i, :) - Aug(i, k)*Aug(k, :);
    end
    for i = k+1 : n
        Aug(i, :) = Aug(i, :) - Aug(i, k)*Aug(k, :);
    end
end

disp(Aug);
```

```
if Aug(k, k) == 0
    disp('No unique solution');
else
    x = zeros(n, 1);
    for i = 1 : n
     x(i) = Aug(i, nb);
    end
end
t = toc;
end
```

LU-DECOMPOSITION

CODE SNIPPETS

```
function [x, t] = LUDecomp(A, B)
tic;

n = length(A);
U = A;
L = eye(n);

% Buliding U and L
for k = 1 : n-1
    for i = k+1 : n
        factor = U(i, k) / U(k, k);
        L(i, k) = factor;
        U(i, k:n) = U(i, k:n) - factor*U(k, k:n);
    end
end
%Now you have L and U ready

% Acuiring D vector
D = zeros(n,1);
D(1) = B(1);
for i = 2 : n
    D(i)= B(i) - L(i,i-1:-1:1)*D(i-1:-1:1);
end

% back substitution to get X out of D
x=zeros(n,1);
x(n)= D(n)/ U(n,n);
for i = n-1 : -1 : 1
    x(i)=(D(i) - U(i,i+1:n)*x(i+1:n)) / U(i,i);
end
t = toc;
end
```

ROOT FINDING

DIFFERENTIATION

We implemented central difference differentiation method.

CODE SNIPPETS

```
function derivative = Mydifferentiate(expression , subst)
try
    syms x;
    eval('yy = [];');
    yy = subs(expression, x, subst+0.01);
    z1 = eval(yy);
    yy = subs(expression, x,subst-0.01);
    z2 = eval(yy);
    yy = subs(expression,x,subst+0.02);
    z3 = eval(yy);
    yy = subs(expression,x,subst-0.02);
    z4 = eval(yy);
    derivative = (-1*z3 + 8*z1 -8*z2 +z4)/(12*0.01);
catch
    derivative = 0;
end
end
```

BISECTION

The method is applicable for solving the equation f(x) = 0 for the real variable x, where f is a continuous function defined on an interval [a, b] and f(a) and f(b) have opposite signs. In this case, a and b are said to bracket a root since, by the intermediate value theorem, the continuous function f must have at least one root in the interval (a, b).

The algorithm works as follows:

1. Calculate c, the midpoint of the interval, c = 0.5 * (a + b).

2. Calculate the function value at the midpoint, f(c).

3. If convergence is satisfactory (that is, a - c is sufficiently small, or f(c) is sufficiently small), return c and stop iterating.

4. Examine the sign of f(c) and replace either (a, f(a)) or (b, f(b)) with (c,f(c)) so that there is a zero crossing within the new interval.

CODE SNIPPETS

```
function [errorFlag,root, eps] = bisection(expression, upper, lower, epsilon,imax)
%
% root = zeros(imax,1);
% eps = zeros(imax,1);
try
    errorFlag = '';
    root = [0];
    eps = [0];
    for  i=2:(imax+1)
        eval('z=0;');
        z = (upper +lower)/2;        %lowerisectioiterNo method

        eval('f1 = 0;');
        eval('f2 = 0;');
        eval('y=[];');
        syms x;
        try
        y = subs(expression, x, z);
        f1 = eval(y);
        y = subs(expression, x, upper);
        f2 = eval(y);
        catch
            errorFlag = 'Invalid Function Expression';
            return;
        end
        if(f1 * f2 < 0)
            lower = z;
        else
            upper = z;
        end

        if(f1 == 0)
            root(i) = z;
            eps(i) = 0;
            break;
        end
        root(i) = z;
        eval('myeps=0.0;');
        myeps = abs((z-root(i-1))/z);
        eps(i) = abs(myeps);
        if(z==0)
            eps(i) = -1;
        end
        if(abs(myeps)<epsilon)
            break;
        end
    end
catch
    errorFlag = 'Un Idetified Error Occurred';
end

end
% display(root)
% display(iterNo)
% display(precision)
```

The method is guaranteed to converge to a root of the function f if f is a continuous function on the interval [a, b] and f(a) and f(b) have opposite signs. The absolute error is halved at each step so the method converges linearly, which is comparatively slow.

Specifically, if c1 = (a+b)/2 is the midpoint of the initial interval, and cn is the midpoint of the interval in the nth step, then the difference between cn and a solution c is bounded by[7]

This formula can be used to determine in advance the number of iterations that the bisection method would need to converge to a root

$$|c_n - c| \leq \frac{|b-a|}{2^n}.$$

Pros :

•Easy

•Always finds a root

•Number of iterations required to attain an absolute error can be computed a priori.

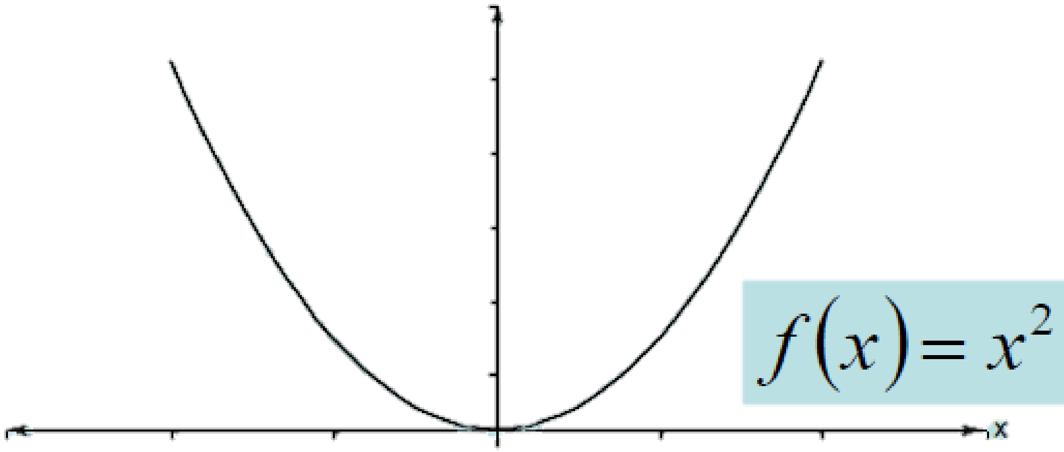
Cons:

•Relatively Slow

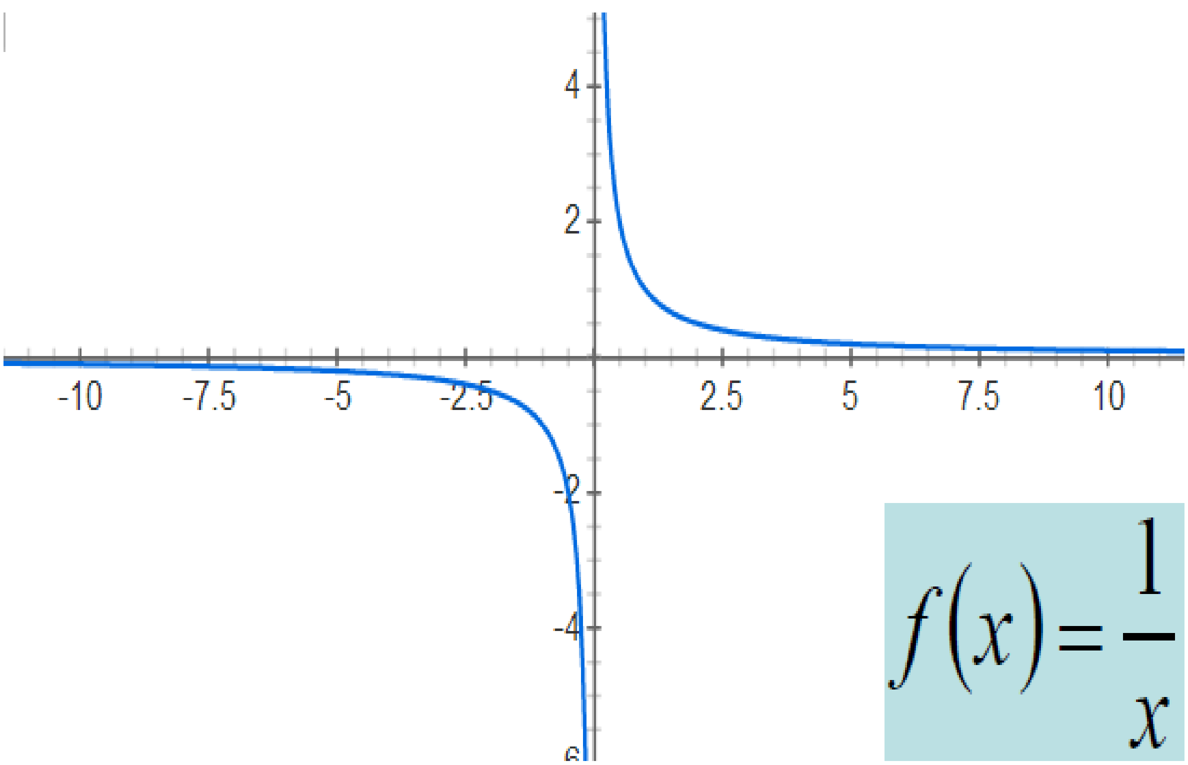
•Need to find initial guesses for xl and xu

•No account is taken of the fact that if f(xl) is closer to zero, it is likely that root is closer to xl .

•If a function f(x) is such that it just touches the x-axis it will be unable to find the lower and upper guesses

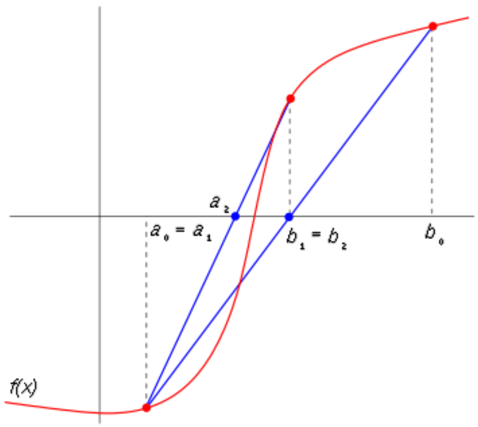


- Function changes sign but root does not exist it converge to a false root



FALSE-POSITION

Like the bisection method, the false position method starts with two pointsa_0 and b_0 such that $f(a_0)$ and $f(b_0)$ are of opposite signs. The method proceeds by producing a sequence of shrinking intervals $[a_k, b_k]$ that all contain a root of f.



At iteration number k, the number:

$$c_k = b_k - \frac{f(b_k)(b_k - a_k)}{f(b_k) - f(a_k)}$$

is computed. As explained below, c_k is the root of the secant line through $(a_k, f(a_k))$ and $(b_k, f(b_k))$. If $f(a_k)$ and $f(c_k)$ have the same sign, then we set $a_{k+1} = c_k$ and $b_{k+1} = b_k$, otherwise we set $a_{k+1} = a_k$ and $b_{k+1} = c_k$. This process is repeated until the root is approximated sufficiently well.

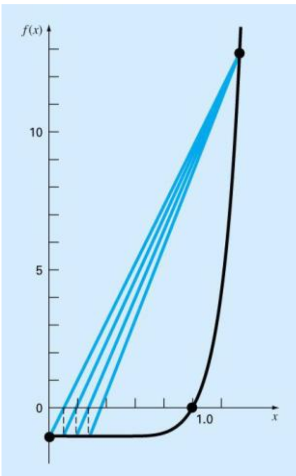
The above formula is also used in the secant method, but the secant method always retains the last two computed points, while the false position method retains two points which certainly bracket a root. On the other hand, the only difference between the false position method and the bisection method is that the latter uses $c_k = (a_k + b_k) / 2$.

CODE SNIPPETS

```
function [errorFlag,root, eps] = falsePosition(expression,oldPoint, newPoint, epsilon,imax)

% root = zeros(imax,1);
% eps = zeros(imax,1);
try
    errorFlag = '';
    root = [0];
    eps = [0];
    for i=2:(imax+1)
        syms x;
        eval('y = [];');
        eval('f1 = 0;');
        eval('f2 = 0;');
        try
        y = subs(expression,x,oldPoint);
        f1 = eval(y);
        y = subs(expression,x,newPoint);
        f2 = eval(y);
        eval('z = 0;');
        if(f1 ==f2)
            errorFlag = 'Division by zero';
            return;
        end

        z = oldPoint - (((oldPoint-newPoint).*(f1) )/ (f1-f2));
        eval('fz = 0;');
        y = subs(expression, x, z);
        fz = eval(y);
        catch
            errorFlag = 'Invalid Function Expression';
            return;
        end

        root(i) = z;
        eval('cureps = 0;');
        cureps = abs(z-root(i-1))/z;
        eps(i) = cureps;
        if(abs(cureps)<epsilon)
            break;
        end
    end
catch
    errorFlag = 'Un Identified Error Occurred';
end
end
```

ANALYSIS AND PITFALLS

One of the pitfalls is the “one sided nature”. This occurs when only one of the two points moves along while the other remains in position (new point comes at the same side every time). if one of the end points becomes fixed, it can be shown that it is still an O(h) operation, that is, it is the same rate as the bi-section method, usually faster, but possibly slower.



One way to mitigate the “one-sided” nature of the false position (i.e. the pitfall case) is to have the algorithm detect when one of the bounds is stuck.

If this occurs, then the original bisection formula xr = (xl + xu)/2 can be used.

FIXED POINT

To find the root for f (x) = 0, we reformulate f (x) = 0 so that there is an x on one side of the equation.

$$f(x) = 0 \quad \Leftrightarrow \quad g(x) = x$$

• If we can solve g (x) = x, we solve f (x) = 0. X is called fixed point of g(x)

• We solve g (x) = x by computing

$$x_{i+1} = g(x_i) \quad \text{with}\, x_0 \text{ given}$$

until xi+1 converges to x.

Error analysis:

$$\delta_{i+1} = g'(c)\delta_i$$

• Therefore, if |g'(c)| < 1, the error decreases with each iteration. If |g'(c)| > 1, the error increase.
• If the derivative is positive, the iterative solution will be monotonic.
• If the derivative is negative, the errors will oscillate.

CODE SNIPPETS

```
function [errorFlag,root,eps] = fixedPoint(expression, x0, es, imax)
% root = zeros(imax,1);
% eps = zeros(imax, 1);
root = [0];
eps = [0];
%expression = strcat(expression,'-x');
errorFlag = '';
try
    for i = 1 : imax
        syms x;
        eval('y = [];');
        eval('z = 0;');
        try
        y = subs(expression, x, x0)
        z = eval(y)
        catch
            errorFlag = 'Invalid Function Expression';
            return;
        end
        y = subs(expression,x,z);
        eval('fz = 0;');
        fz =eval(y);
        if(fz == 0)
            root(i) = z;
            eps(i) = 0;
            break;
        end
```

```
            root(i) = z;

            ea = abs(z-x0)/z;
            eps(i) = ea;
            if(abs(ea) < abs(es))
                break;
            end
            x0 = z;
        end
catch
    errorFlag = 'Un Identified Error Occurred';
    return;
end
end
```

PITFALLS

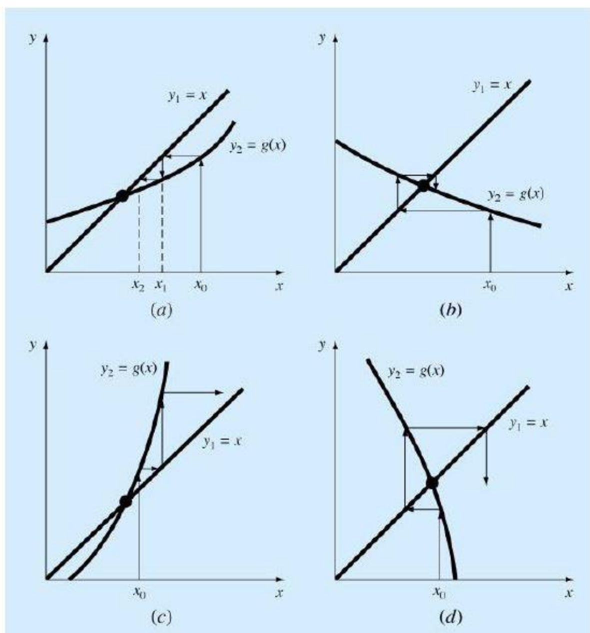
(a) $|g'(x)| < 1$, $g'(x)$ is +ve
$\Rightarrow$converge, monotonic

(b) $|g'(x)| < 1$, $g'(x)$ is -ve
$\Rightarrow$converge, oscillate

(c) $|g'(x)| > 1$, $g'(x)$ is +ve
$\Rightarrow$diverge, monotonic

(d) $|g'(x)| > 1$, $g'(x)$ is -ve
$\Rightarrow$diverge, oscillate

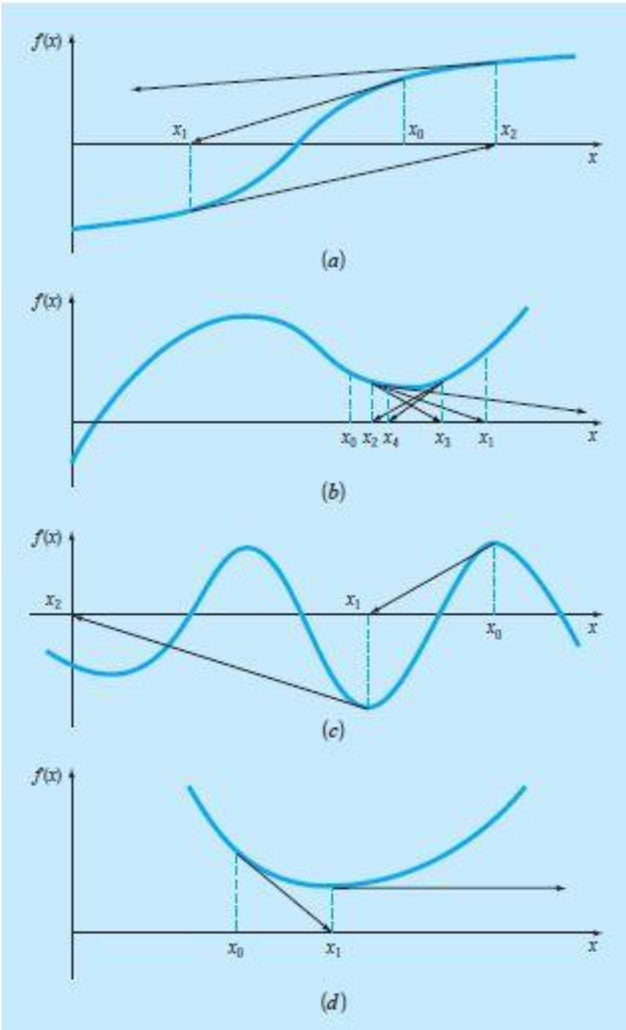
Demo



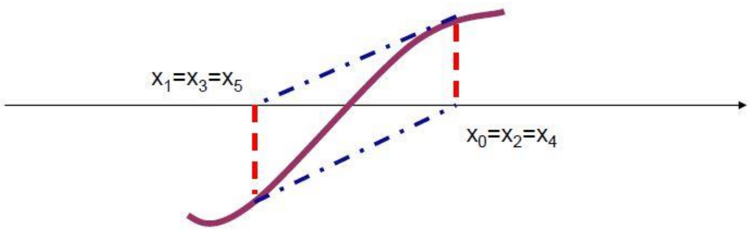
NEWTON-RAPHSON

CODE SNIPPETS

```
function [errorFlag,root, eps ] = newtonRaphson(expression, xl,es,imax)
% root = zeros(imax, 1);
% eps = zeros(imax, 1);

root = [0];
eps = [0];
errorFlag = '';
try
    for i =1:imax
        syms x;
        eval('y = [];');
        y = subs(expression, x, xl);
        if(y == '')
            errorFlag = 'Invalid Funciton Expression';
            break;
        end
        eval('dif = 0;');
        dif = Mydifferentiate(expression, xl);
        if(dif == 0)
            errorFlag = 'First Derivative = 0 Error';
            return;
        end
        xr = xl - eval(y)/dif;
        y = subs(expression, x, xr);
        eval('fz = 0;');
        fz = eval(y);
        if(fz == 0)
            root(i) = xr;
            eps(i) = 0;
            break;
        end

        ea = abs(xr-xl)/xr;
        xl = xr;
        root(i) = xr;
        eps(i) = ea;
        if(abs(ea)<abs(es))
            break;
        end
    end
catch
    errorFlag = 'Un Identified Error Occurred';
end
end
```

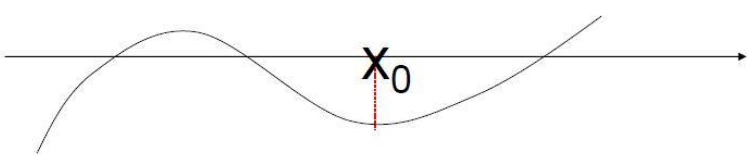

f(x)
x
(a)
(b)
(c)
(d)

Problems with Newton's Method
- Cycle -

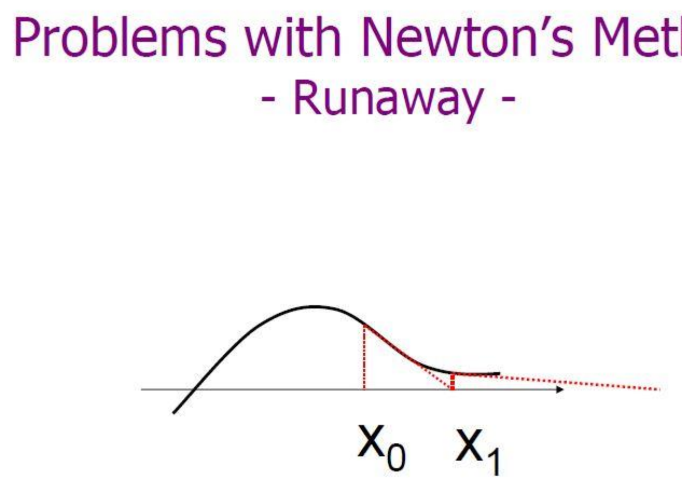


The algorithm cycles between two values x_0 and x_1

Problems with Newton's Method
- Flat Spot -



The value of ***f'(x)*** is zero, the algorithm fails.



SECANT

The secant method is defined by the recurrence relation

$$x_n = x_{n-1} - f(x_{n-1})\frac{x_{n-1} - x_{n-2}}{f(x_{n-1}) - f(x_{n-2})} = \frac{x_{n-2}f(x_{n-1}) - x_{n-1}f(x_{n-2})}{f(x_{n-1}) - f(x_{n-2})}$$

As can be seen from the recurrence relation, the secant method requires two initial values, x_0 and x_1, which should ideally be chosen to lie close to the root.

The recurrence formula of the secant method can be derived from the formula for Newton's method

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

by using the finite difference approximation

$$f'(x_{n-1}) \approx \frac{f(x_{n-1}) - f(x_{n-2})}{x_{n-1} - x_{n-2}}.$$

If we compare Newton's method with the secant method, we see that Newton's method converges faster (order 2 against α ≈ 1.6). However, Newton's method requires the evaluation of both f and its derivative f' at every step, while the secant method only requires the evaluation of f.

CODE SNIPPETS

```
function[errorFlag,root,eps] = secant(expression, xl,xu, es, imax)
% root = zeros(imax,1);
% eps = zeros(imax, 1);

root = [0];
eps = [0];
errorFlag = '';
try
    for i =1:imax
        syms x;
        eval('y = [];');
        eval('z = 0;');
        eval('z2 = 0;');
        try
        y = subs(expression, x, xl);
        z = eval(y);
        y = subs(expression, x, xu);
        z2 = eval(y);
        catch
            errorFlag = 'Invalid Function Expression';
            returen;
        end
        if(z2 == z)
            errorFlag = 'Division by zero';
            return;
        end

        ea = abs(xr-xl)/xr;
        xl = xu;
        xu = xr;
        root(i) = xr;
        eps(i) = ea;
        if(abs(ea)<abs(es))
            break;
        end
    end
catch
    errorFlag = 'Un Identified Error Occurred';
end
end
```

ANALYSIS AND PITFALLS

* The convergence of the secant method is superlinear, but not quite quadratic. (approximately 1.618).

* If the initial values are not close enough to the root, then there is no guarantee that the secant method converges. There is no general definition of "close enough", but the criterion has to do with how "wiggly" the function is on the interval $[\ x_0,\ x_1\ ]$. For example, if f is differentiable on that interval and there is a point where $f' = 0$ on the interval, then the algorithm may not converge.

* The secant method does not require that the root remain bracketed like the false position method does, and hence it does not always converge.

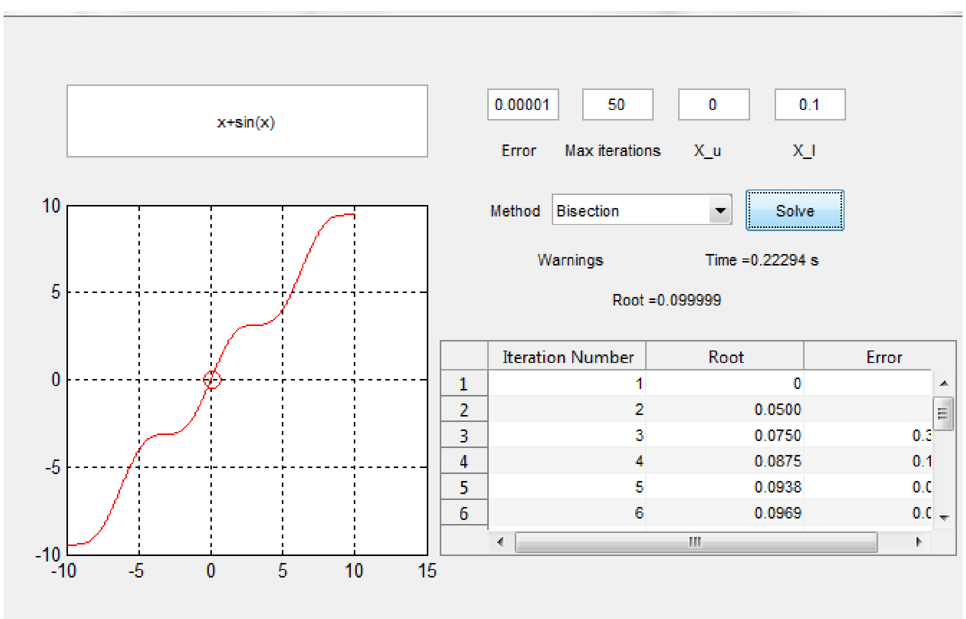
GUI

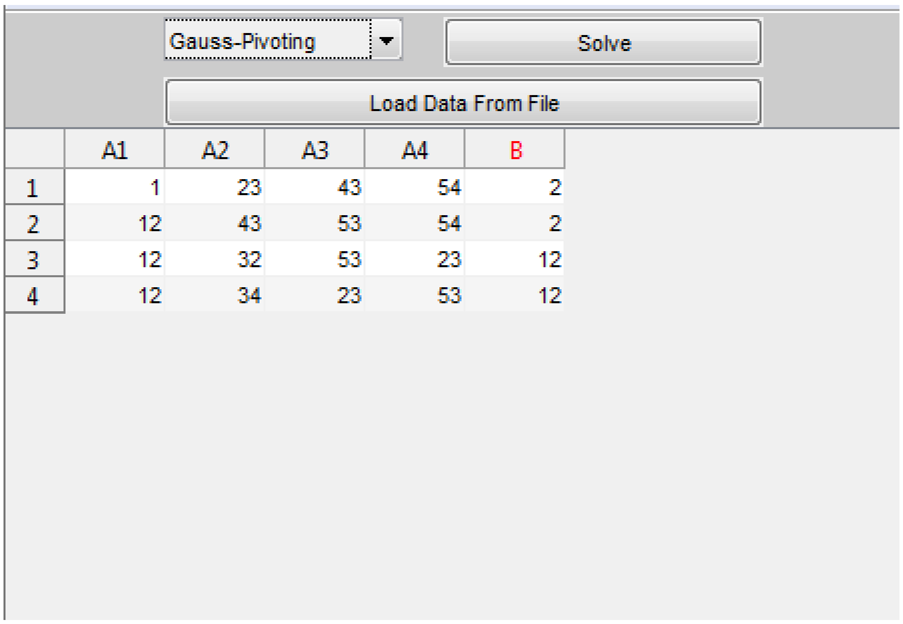
- Explanation here

SAMPLE SNAPSHOTS

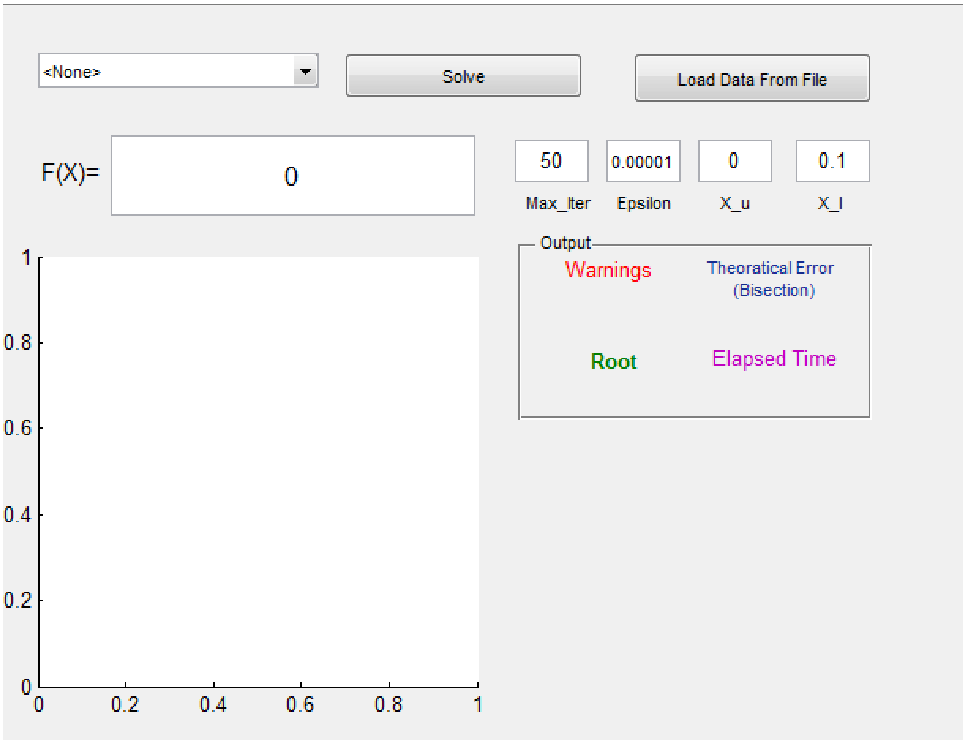
INTRODUCTORY

ROOT FINDING





	A1	A2	A3	A4	B
1	1	23	43	54	2
2	12	43	53	54	2
3	12	32	53	23	12
4	12	34	23	53	12



Elapsed Time =0.0004529 Seconds

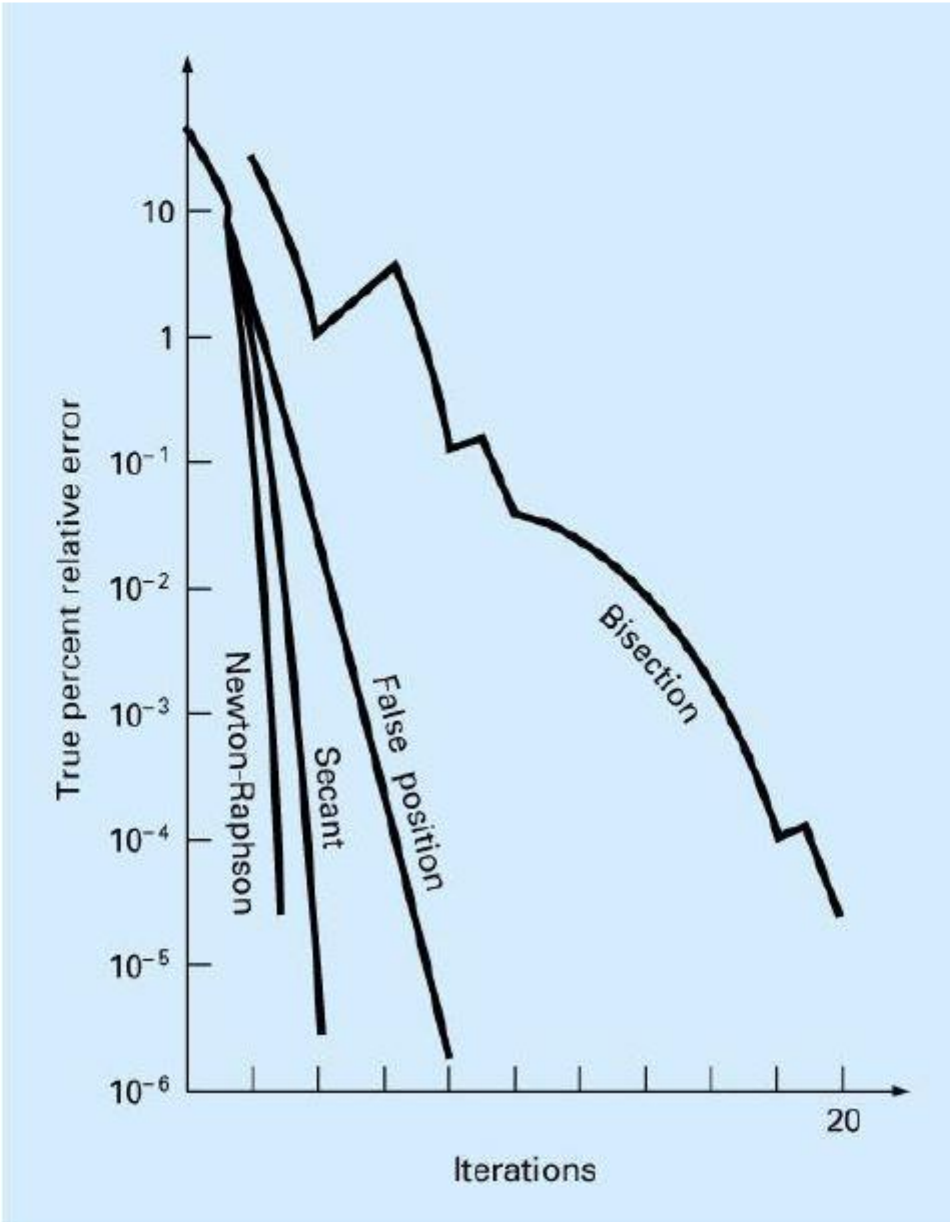
	X
1	5.3730
2	-3.2642
3	0.6181
4	0.8357

SUMMARY

COMPARISON

Summary

Method	Pros	Cons
Bisection	- Easy, Reliable, Convergent - One function evaluation per iteration - No knowledge of derivative is needed	- Slow - Needs an interval [a,b] containing the root, i.e., $f(a)f(b)<0$
Newton	- Fast (if near the root) - Two function evaluations per iteration	- May diverge - Needs derivative and an initial guess x_0 such that $f'(x_0)$ is nonzero
Secant	- Fast (slower than Newton) - One function evaluation per iteration - No knowledge of derivative is needed	- May diverge - Needs two initial points guess x_0, x_1 such that $f(x_0)- f(x_1)$ is nonzero



True percent relative error
10
1
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
Newton-Raphson
Secant
False position
Bisection
20
Iterations

DATA STRUCTURES

GUI

UI TABLE

uitable creates an empty uitable object in the current figure window, using default property values. If no figure exists, a new figure window opens.

This data structure creates a 2-D graphic table GUI component, and provide properties to handle it.

It was pretty useful to us in showing the steps for the root finding method with its three columns {Iteration Number, Root, and Epsilon} and we used it also in Linear Algebra methods for the handling the coefficient matrix and showing the results as well

CELL STRING ARRAY

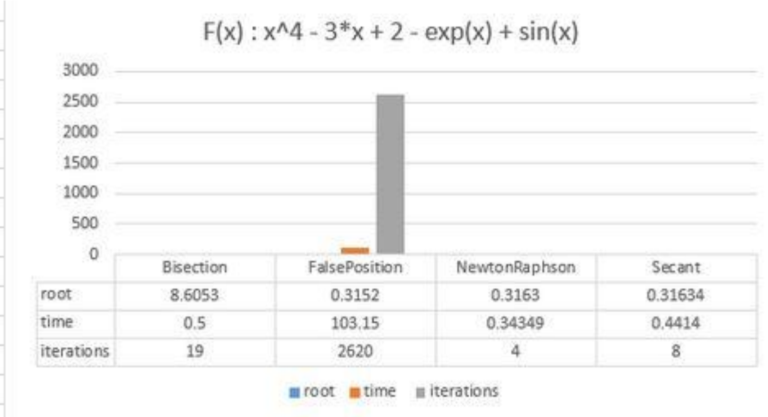
Create cell array of strings from character array.

It was very useful while creating the names of the variables (Row names) in the Part B of the Assignment

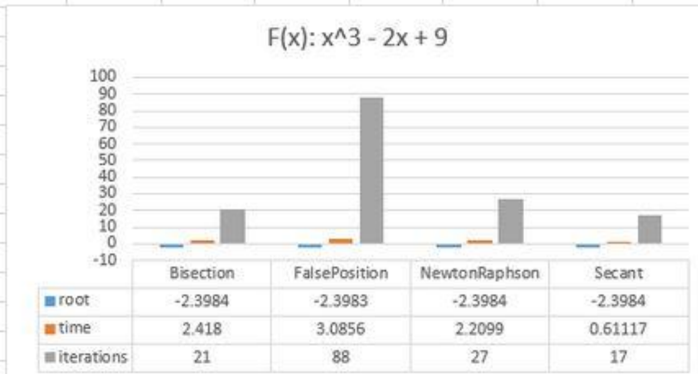
BEHAVIORAL ANALYSIS

CASE STUDY

Function	root	time	iterations
Bisection	8.6053	0.5	19
FalsePosi	0.3152	103.15	2620
NewtonRa	0.3163	0.34349	4
Secant	0.31634	0.4414	8



Function	root	time	iterations
Bisection	-2.3984	2.418	21
FalsePosi	-2.3983	3.0856	88
NewtonRa	-2.3984	2.2099	27
Secant	-2.3984	0.61117	17



Magic Function	Time	No. Iteration
sqrt(2*x+3)	0.27128	9
2/(x-3)	0.2272	10