Applied Programming

Solving Nonlinear Equations

in one variable

Part I:

Bracketing Algorithms

Example: Motor Speed

- We have an electric motor that operates in the 0 to 50V range
- We found, by **data fitting**, that the RPM of the motor depends on the applied voltage

$$\boxed{f(v) = 52.2\, v + 0.75\, v^2 - 0.02\, v^3}$$

Want to know: What **voltage** must be applied to the motor to run it at **1909** RPMs ?

Analytic Solution

- Re-arrange the speed equation in the ***"canonical form"***

$$f(v)=g(v)-h(v) = 0:$$

$$h(v) = -0.02v^3 + 0.75v^2 + 52.2v = 0$$

$$g(v) = 1909$$

$$f(v) = 0.02\, v^3 - 0.75\, v^2 - 52.2\, v + 1909 = 0$$

- We could analytically solve this equation for x to but…
  - We are only interested in one root (between 0 and 50) not all the 3 roots.
  - We would have to start from scratch to find x for a different motor speed ($\neq 1909$)

Motor Speed: Analytic Solution

- First, re-arrange the speed equation to write it in the ***"canonical form" $f(v)=g(v)-h(v) = 0$:***

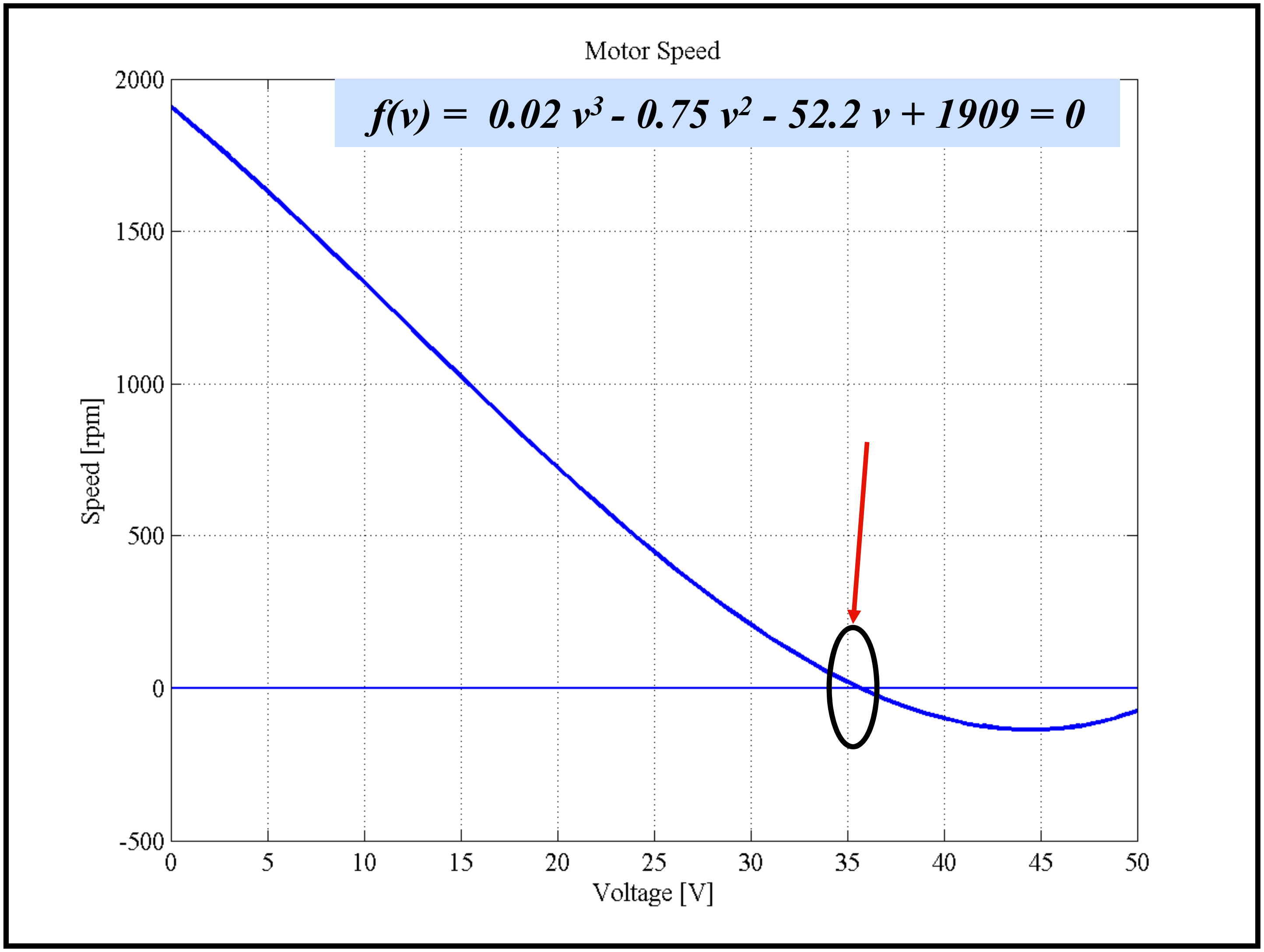
  $$h(v) = -0.02v^3 + 0.75v^2 + 52.2v = 0$$

  $$g(v) = 1909$$

  $$f(v) = 0.02\,v^3 - 0.75\,v^2 - 52.2\,v + 1909 = 0$$

- Solve the equation analytically, but…
  - We are only interested in one root (between 0 and 50) not in all the 3 roots.

Motor Speed: Graphical Solution



A root (zero—crossing) exists in the **"bracket" [0,50]**

Problems with no Analytic Solution

- In general it is ***not possible*** to ***solve nonlinear equations explicitly***, we must solve them ***numerically***

- ***Example:***
  Design a sky-diving "suit" with a ***drag coefficient "c"*** such that after ***t=10*** **sec** a **90.7kg** (~200 lb) sky-diver is traveling at ***v=8 m/s*** (~20mph)

$$v(t) = \frac{gm}{c}\left(1 - e^{-(c/m)\,t}\right)$$

g = 9.8 m/s ^{2}

c= 111,130 (a parachute). A typical car has a drag coefficient of 0.34

Root Finding Introduction

- The problem of solving (scalar) nonlinear equations of the form

$$g(x) = h(x)$$

can be ***transformed*** (setting $f(x) = g(x) - h(x)$) ***to*** the "canonical" ***root finding problem***:

$$\boxed{\text{Find } x \text{ such that } f(x) = 0}$$

The Root Finding Problem

- The root finding problem is:

Given a function f(x), find the values of x for which f(x)=0

- Such values of *x* are called **the zeros of *f(x)*** or **the roots of f(x)= 0**

Note: Some "roots" are simple (have multiplicity one) and others are repeated (have multiplicity greater than one) –

- *Simple example: roots of polynomials*

$$f(x) = x^2 - 2x + 1 = 0 \Rightarrow x = 1 \text{ of multiplicity } m = 2$$

(for more details see background slides on multiplicity)

Root finding Approaches

Two main approaches:

1. ***Bracketing methods***
   - Bisection

2. ***Open methods***
   - Newton
   - Secant (quasi-Newton)

Bracketing Methods

- Preconditions:
  - An ***interval [a,b]*** (the bracket) where ***f(x)=0*** (*e.g. f(x)*) has a root) is ***known a priori***
  - The function ***f(x)*** is ***continuous*** in the ***neighborhood of the root***.
- Principle: ***Reduce the size of the bracket [a,b]*** that contains the root until it is "small enough"
- ***Root inclusion criteria***:

root of f(x) is in [a,b] only if sign(f(a)) ≠ sign(f(b))

The Bisection Method

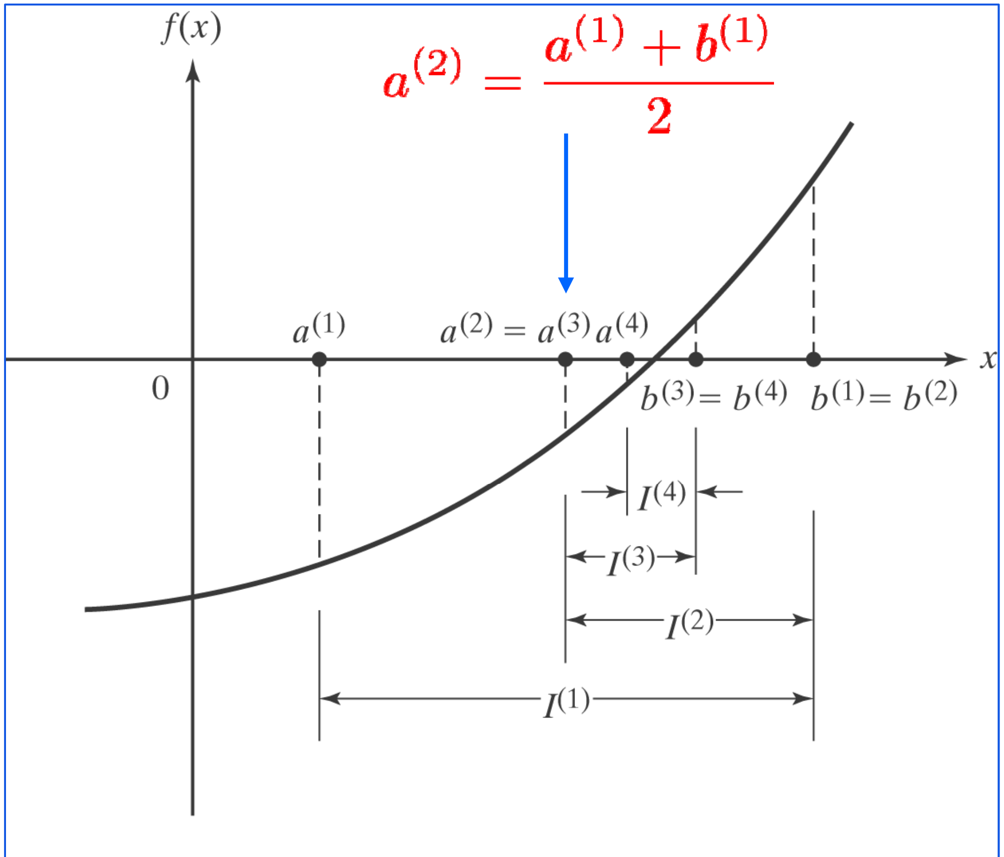


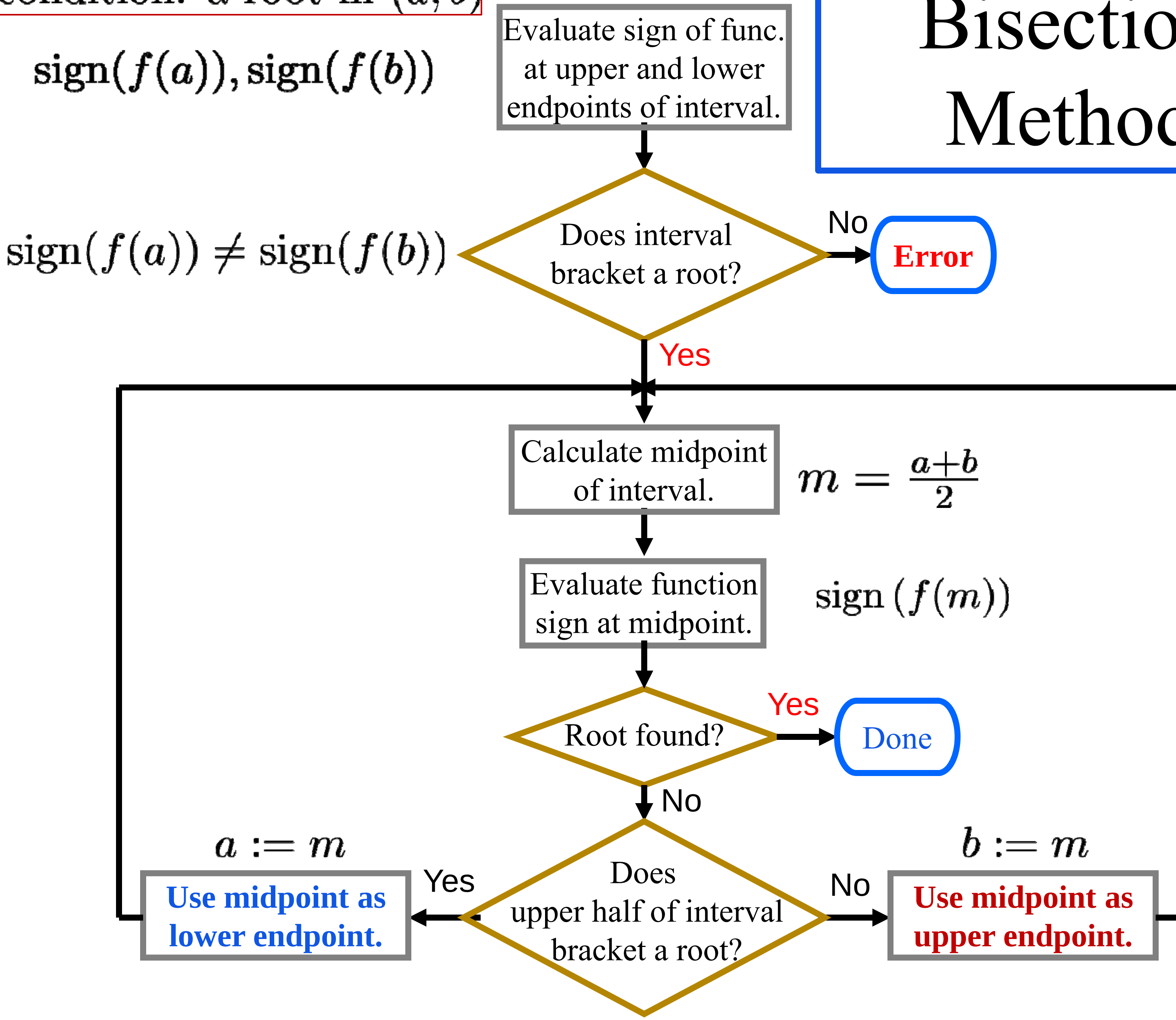
Figure 2.16
Bisection method.

- ***Halve the size*** of the bracketing interval enclosing the root (e.g., a binary search)
- Choose the new smaller bracket that includes (brackets) the root.
- Repeat ***until bracket size*** is ***small enough***
- Root inclusion criteria: value of function has ***opposite signs at bracket endpoints***

Used with permission from support materials for *Applied Numerical Methods for Engineers and Scientists*, S. S. Rao, Prentice Hall, 2002.

Bisection Method

Precondition: a root in (a, b)



Bisection and Repeated Roots

- Bisection Fails when bracket cannot be determined by change of sign of function

  This occurs when repeated roots have even multiplicity

Note: when root have ***odd multiplicity*** this issue does not arise

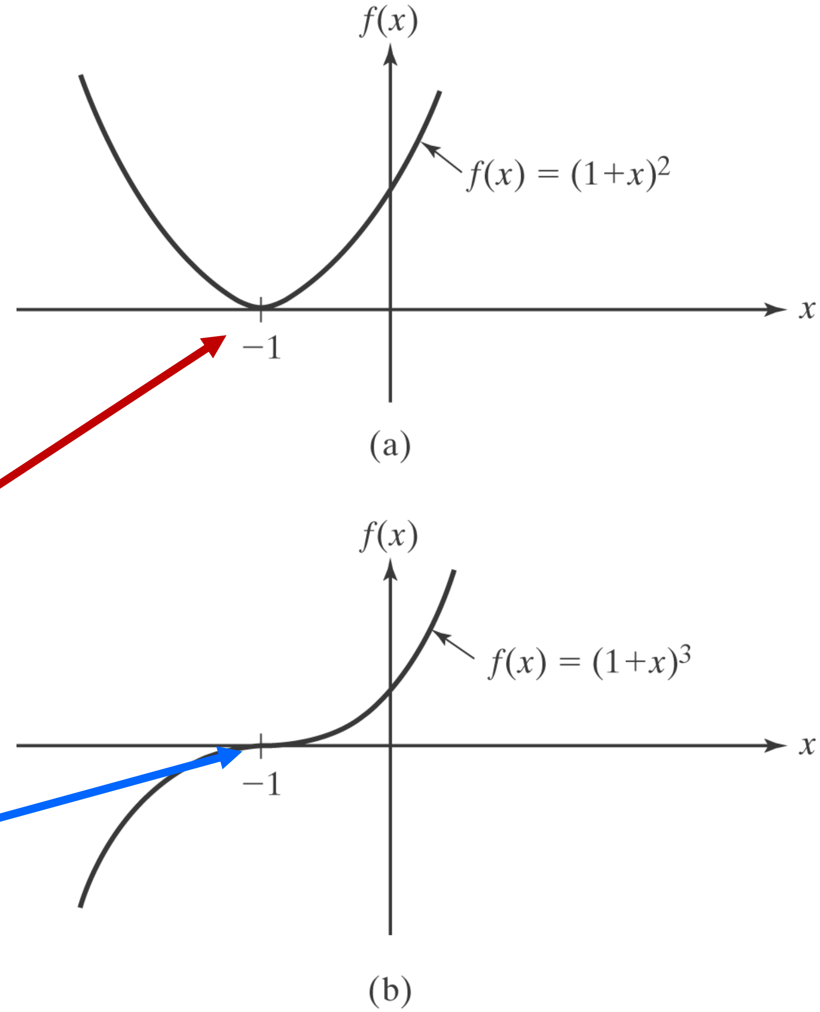


Figure 2.2
Multiple roots of $f(x) = 0$.

Used with permission from support materials for *Applied Numerical Methods for Engineers and Scientists*, S. S. Rao, Prentice Hall, 2002.

Poor Stopping Conditions

- Let ε denote the ***desired tolerance***
- The main stopping criteria *for any root finding algorithm* are:

1. Absolute approximation error

$$|x_n - x^*| < \varepsilon_a$$

2. Relative approximation error

$$|x_n - x^*| < \varepsilon_r |x_n|$$

x^* is not normally known !

3. Value of the function

$$|f(x)| < \varepsilon_f$$

The value of the function may be small even when we are not close to the root

Bisection Stopping Condition

- At the n^{th} iteration the following bound holds:

$$|x_n - x^*| < \left|\frac{b^n - a^n}{2}\right|$$

$[b^{(n)}, a^{(n)}]$ is the bracket at the n^{th} iteration

therefore we can satisfy the requirements only if

$$|x_n - x^*| < \left|\frac{b^n - a^n}{2}\right| < \epsilon_a$$

Stop when the bracket is small $|b^n - a^n| < 2\epsilon_a$

Note: Floating point absolute value in "C" is **fabs** NOT abs

Bisection Algorithm in a Nutshell

- Precondition:
  - ✓ Requires an interval that brackets the solution
- Limitations:
  - The order of ***convergence is linear*** and slow (asymptotic error constant is only ½)

- Advantages:
  - + Very simple
  - + Guaranteed convergence (to simple roots)
  - + Easy to control accuracy by choosing tolerance
  - + Number of iterations (N) for desired accuracy can be pre-computed (for prescribed absolute error of ε_a)

$$N = \left\lceil \frac{\log_2(b-a)}{2\varepsilon_a} \right\rceil$$

Implementation: Errors & Inefficiencies

```
## module bisection
''' root = bisection(f,x1,x2,switch=0,tol=1.0e-9).
    Finds a root of f(x) = 0 by bisection.
    The root must be bracketed in (x1,x2).
    Setting switch = 1 returns root = None if
    f(x) increases upon bisection.
'''
from math import log,ceil
import error

def bisection(f,x1,x2,switch=1,atol=1.0e-9):
    f1 = f(x1)
    if f1 == 0.0: return x1
    f2 = f(x2)
    if f2 == 0.0: return x2
    if f1*f2 > 0.0: error.err('Root is not bracketed')
    n = ceil(log(abs(x2 - x1)/atol)/log(2.0))
    for i in range(n):
        x3 = 0.5*(x1 + x2); f3 = f(x3)
        if (switch == 1) and (abs(f3) > abs(f1)) \
                         and (abs(f3) > abs(f2)):
            return None
        if f3 == 0.0: return x3
        if f2*f3 < 0.0: x1 = x3; f1 = f3
        else:           x2 = x3; f2 = f3
    return (x1 + x2)/2.0
```

Example of a poor and erroneous implementation of bisection (in Python)

Implementation Notes

- For efficiency the algorithm should be implemented to:
  1. ***Minimize the number of function evaluations***, *f(x)*
     - The number of iterations is determined by the size of the initial bracket
     - Efficiency is mainly affected by the ***number of function evaluations***

  2. ***Avoid unnecessary loss of precision***
     - ***Do not use $f(a^{(n)}) - f(b^{(n)})$ as bracketing criteria*** (as the algorithms gets closer to a root *f(a)* and *f(b)* both may become very small and subtraction can underflow)
     - Use the sign of the function

Bisection: Matlab/Octave Implementation

```
function f = fmotor(v)
% f(v) =  0.02 v3 - 0.75 v2 - 52.2 v + 1909 = 0
f = ((0.02*v - 0.75) * v- 52.2)*v+1909;
```

```
function r = bisection(fun,bracket,tol)
% Bisection method for rootfinding (Juan C. Cockbur
% Usage: r = bisection(fun,bracket,tol)

% I stripped most comments and argument checking (for class use)
  if nargin==0,help bisection.m, return, end;
% Initialize algorithm parameters
  MaxIt = 101;                          % 100 iteration maximum
  xtol = max(2*tol,6*eps);              % set tolerance, check against machine epsilon
  a = bracket(1,1);  b = bracket(1,2);  % initialize bracket endpoints
  fa = feval(fun,a); fb = feval(fun,b); % find f(a) and f(b)
% Start Bisection
  k = 0; % iteration counter
  while k < MaxIt,
    k = k + 1;              % increment iteration counter
    dx = b - a;             % compute bracket interval size
    xm = a + 0.5*dx;        % minimize round-off in computing the midpoint
    fm = feval(fun,xm);     % evaluate function at midpoint (**)
    % Check stopping criterion
    if (abs(dx) < xtol)    % true when root is “found”
      r = xm;  return;      % return root (exit here !)
    end
    % Update bracketing interval
      if sign(fm)==sign(fa) %  Avoid using fa*fb<0
        a = xm;  fa = fm;   %  Root on [xm,b]
      else
        b = xm;  fb = fm;   %  Root on [a,xm]
      end
  end % while
  warning(sprintf('Root not within tolerance %f after %d iterations\n',xtol,k-1));
end % function
```

Example: Motor Speed

Tolerance= 0.05 RPM, Range: 0-50 Volts

Result: V=35.69

```
» r=mybisection(@fmotor,[0 50],0.05)

k            [ a , b ]                     x            f(x)
-------------------------------------------------------------
   1  [           0,          50]         25          447.8
   2  [          25,          50]       37.5          -48.5
   3  [          25,        37.5]      31.25          155.7
   4  [       31.25,        37.5]      34.38          40.77
   5  [      34.375,        37.5]      35.94         -7.297
   6  [      34.375,     35.9375]      35.16          15.91
   7  [    35.15625,     35.9375]      35.55          4.095
   8  [    35.54688,     35.9375]      35.74         -1.654
   9  [    35.54688,    35.74219]      35.64          1.207
  10  [    35.64453,    35.74219]      35.69        -0.2271

r = 35.69
```

Note: The "exact" roots of the cubic polynomial are:

35.685609864217469, 52.633113343145020, -50.818723207362368

Summary: Bisection

- Requires that the root be bracketed.
- Has ***guaranteed linear convergence*** (to roots of ***odd multiplicity***) regardless of where we start.
- Works well for ***"arbitrary" functions*** (no "regularity" requirements, such as differentiability, except close to the root).
- Only requires ***evaluation of the sign of function***.

Conclusion: Bisection is ***slow but robust.***

Exercise 1

Explain the bisection algorithm in a nutshell.

- Choose any 2 points such that the sign f(x1) and f(x2) changes
- Halve the distance between the 2 points and evaluate f() again
  - Choosing the new pair of x's so the f(x) sign still changes
- Repeat until Δx is small