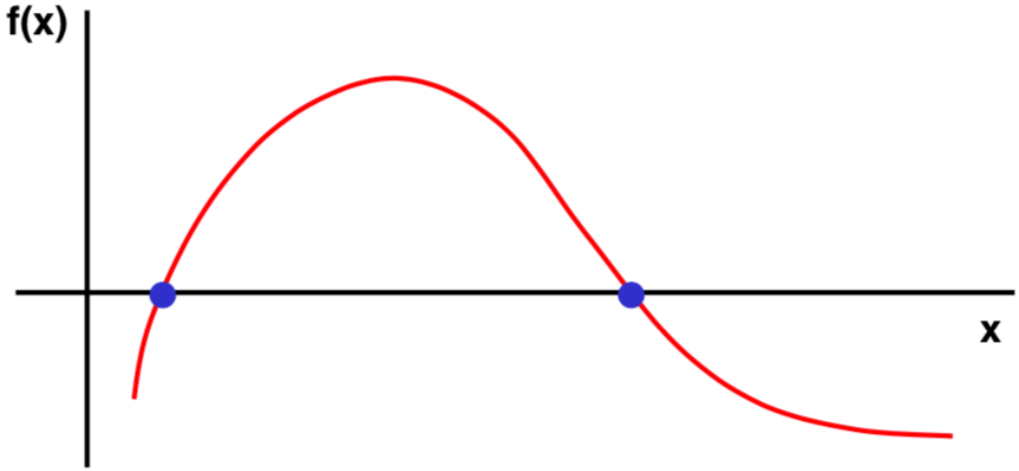
# *Ch. 05*
## *Root Finders & Nonlinear Equations*

Andrea Mignone
Physics Department, University of Torino
AA 2017-2018

# *Finding Roots of nonlinear Equations*



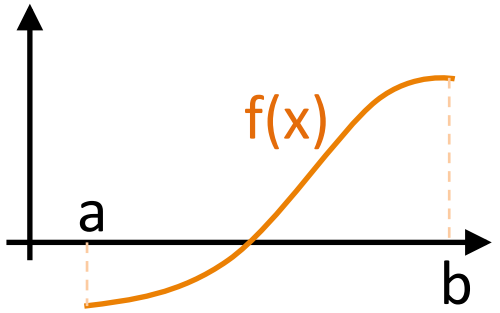

f(x) is given

$f(x_r) = 0 \quad \rightarrow \quad x_r = ?$

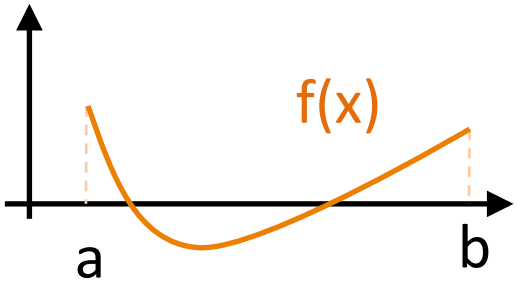
- Except for linear problems, root finding proceeds by iteration: starting from some approximate trial solution, a useful algorithm will improve the solution until some predetermined convergence criterion is satisfied.
- For smoothly varying functions, good algorithms will always converge, *provided* that the initial guess is good enough.
- Success crucially depends on having a good first guess for the solution, especially for multidimensional problems. This crucial beginning usually depends on analysis rather than numerics.

# Conditions for a Root to Exist

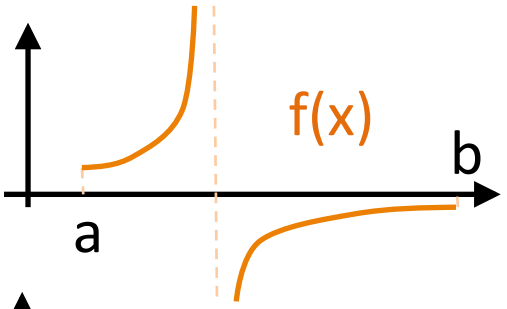
- In most cases, a change of sign of the function in a given interval [a,b] is a *necessary* (but not sufficient) condition for it to have a root.



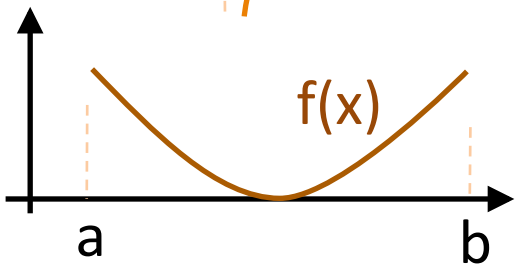

- However, for an even number of roots in [a,b], this is no longer true:




- A change of sign may also given by a function with vertical asymptote:

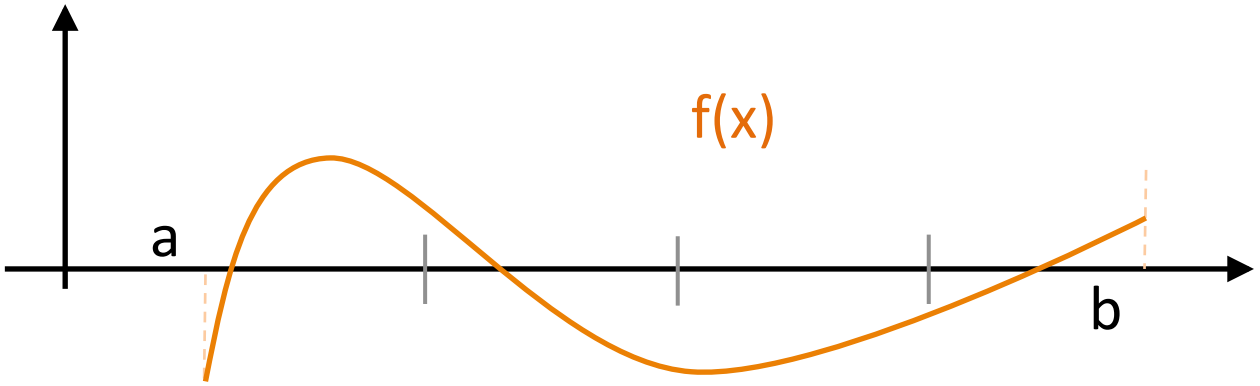



- Multiple roots, or very close roots, are a real problem, especially if the multiplicity is an even number: there may be no readily apparent sign change in the function, so the notion of bracketing a root (and maintaining) the bracket — becomes difficult.



# *Bracketing a Root*

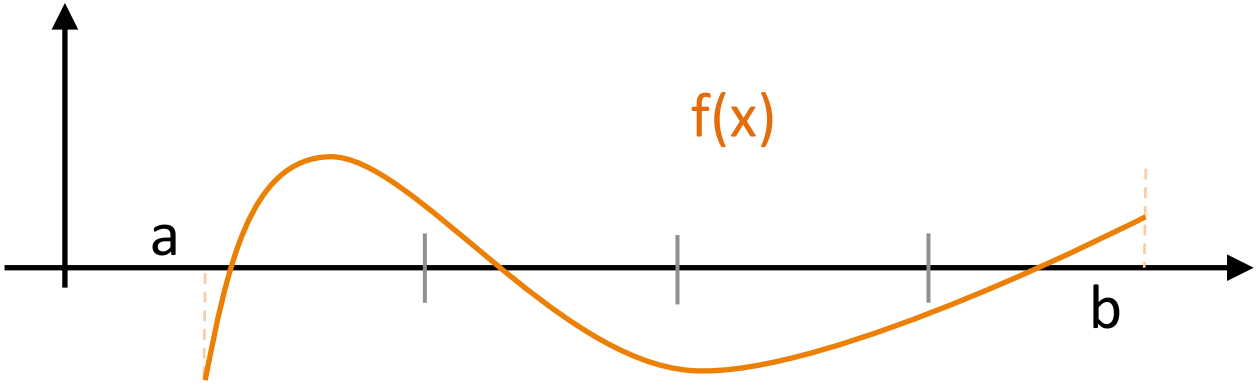
- In order to isolate a single root, we need to pinpoint the intervals where a sign change is detected:




- For continuous function, and excluding roots with even multiplicity, we will say that a root is *bracketed* in the interval $[x_n,x_{n+1}]$ - with $x_n=a+nh$, $x_{n+1}=a+(n+1)h$ - if $f(x_n)$ and $f(x_{n+1})$ have opposite signs: then at least one root must lie in that interval (the *intermediate value theorem*).

- Once we know that an interval contains a root, several classical procedures are available to refine it. These proceed with varying degrees of speed and sureness towards the answer.

# Finding All of the Roots



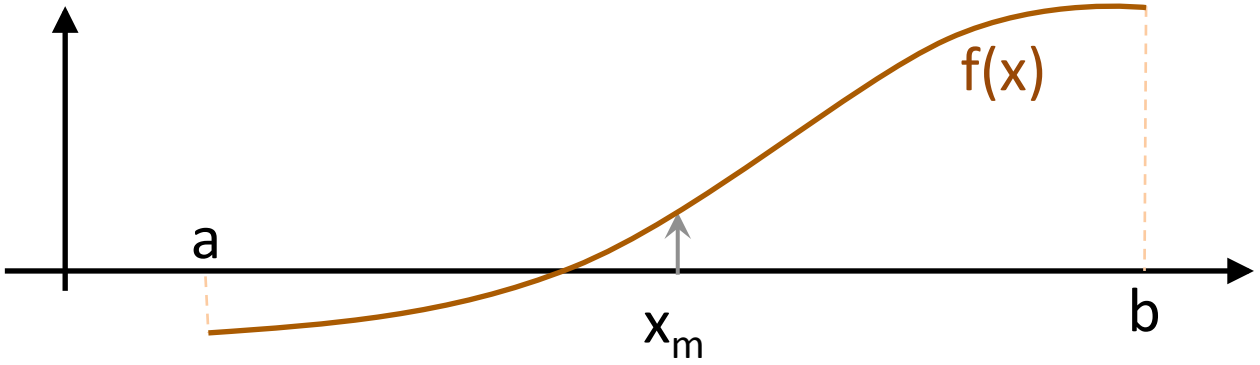

- In general, root finder methods will iterate over an initial interval and be able to converge to one root at a time.

- For this reason, we will first conceive algorithms that operates in a given interval by assuming *that the function changes sign only once in this interval* (i.e. only one root will be assumed).

- In a second step, we will bracket the N possible roots of a function f(x) and call our root finder algorithm once for each of them, by properly specifying the interval.

# Bisection Method

- The *bisection method* is one that cannot fail. It is thus not to be sneered at as a method for otherwise badly behaved problems. The idea is simple.




1. Start with an interval [a,b], bracketing the root.
2. Estimate the root as the midpoint of the interval: $x_m = (a+b)/2$, compute $f(x_m)$ and use the midpoint to replace whichever limit has the same sign.

$$\begin{cases} \text{if} \quad f(a)f(x_m) < 0 \qquad \text{root in} \quad [a, x_m] \;\rightarrow\; b = x_m \\ \text{if} \quad f(a)f(x_m) > 0 \qquad \text{root in} \quad [x_m, b] \;\rightarrow\; a = x_m \end{cases}$$

3. Repeat from step 1. until convergence is reached.

- After each iteration the bounds containing the root decrease by a factor of two.

# Convergence of Bisection Method

- If after n iterations the root is known to be within an interval of size $\delta_n$, then after the next iteration it will be bracketed within an interval of size $\delta_{n+1} = \delta_n/2$ neither more nor less.

- Since the new interval is proportional to the the first power, it is said to converge *linearly*.

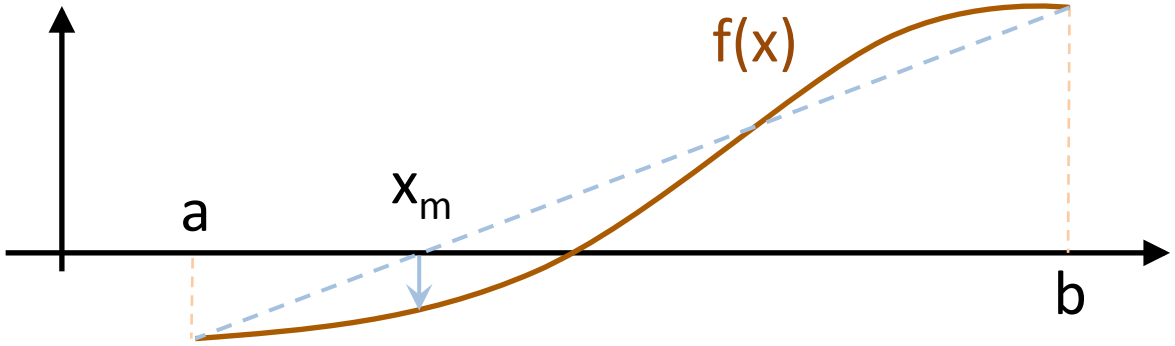
- Thus, we know in advance the number of iterations required to achieve a given tolerance in the solution $n = \log_2 (\delta_o/\delta)$, where $\delta_o$ is the size of the *initially bracketing interval*, $\delta$ is the desired ending tolerance.

- Bisection must succeed:
  - If the interval happens to contain two or more roots, bisection will find one of them.
  - If the interval contains no roots and merely straddles a singularity, it will converge on the singularity.

- Methods that converge as a higher power, $\delta_{n+1} = \text{constant} \times (\delta_n)^m$ with $m > 1$ are said to converge superlinearly.

# False Position (Regula Falsi) Method

- In the false position method, the function is assumed to be approximately linear in the local region of interest, and the next improvement in the root is taken as the point where the approximating line crosses the axis:




- After each iteration one of the previous boundary points is discarded in favor of the latest estimate of the root.

- It converges faster than bisection to the root because it makes usage of appropriate weighting of the initial end points x1 and x2 using the information about the function.

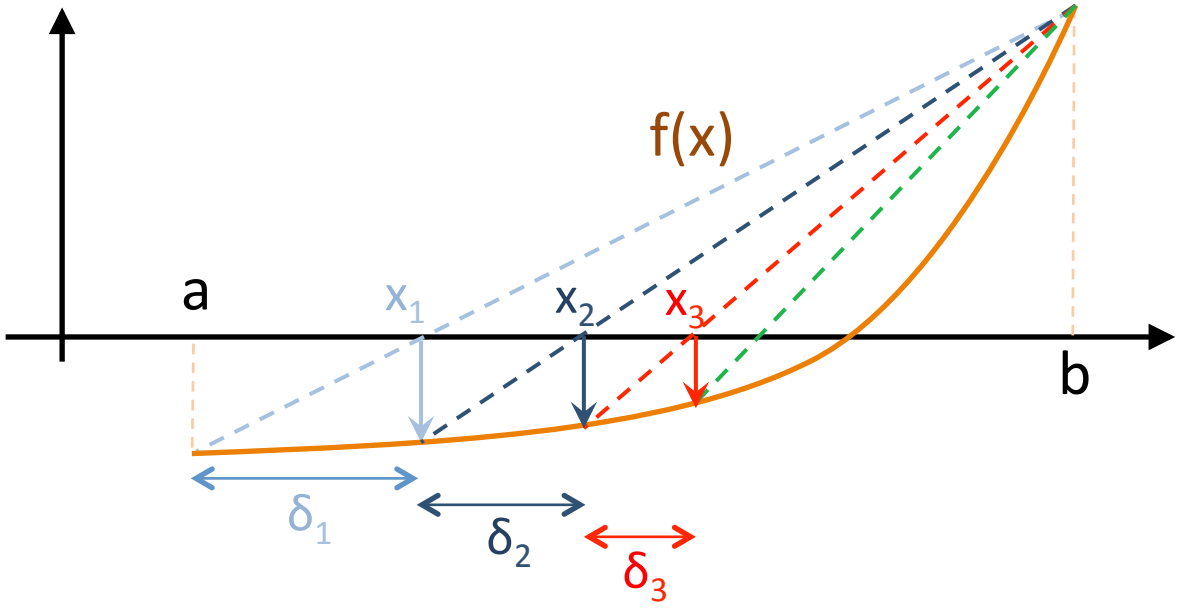
- Usually, the convergence rate of False Position is superlinear, but estimation of its exact order is not so easy.

# Some Caveats on the False Position Method

The algorithm is very similar to the Bisection method and may implemented in the same way, but it differs in two aspects:

1. The next estimate does not coincide with the interval midpoint.

2. The halting condition for the false-position method is different from the bisection method: if, for example, the function is concave up (see the figure below), only the left bound is ever updated. Thus, instead of checking the width of the interval, we check the change in the end points to determine when to stop.

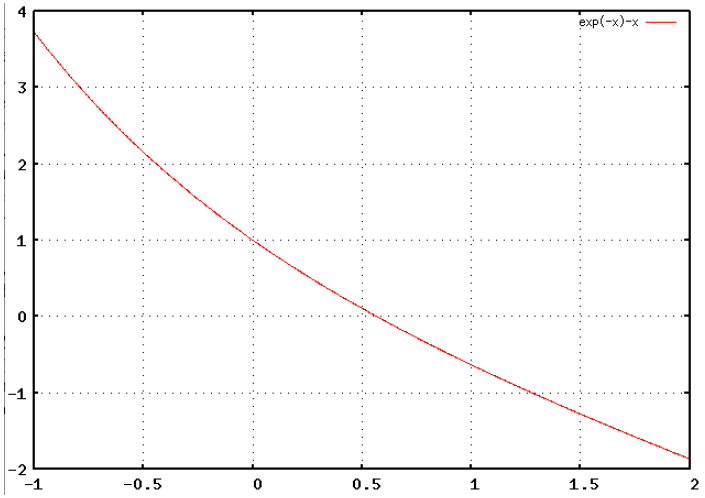


# Practice Session #1

`froot.cpp`: write a program that finds the zero of the trial function

$$f(x) = e^{-x} - x$$

using both bisection and false position methods.




From the plot in the figure it is clear that a single zero occurs in the interval [-1,1].

Make sure to write your function in a rather general (read "re-usable") way.

A possibility (but not the only one) is

```
int Bisection(double (*func)(double), double a, double b,
              double tol, double &zero)
```

or,

```
double Bisection(double (*func)(double), double a, double b,
                 double tol)
```

- How many iteration are necessary to reach a tolerance $10^{-7}$ ?
- What happens if the initial interval is 10 times larger ?

# Practice Session #1

- Using an interval [-1,1] and a tolerance of $10^{-7}$ (exit condition `|b-a| < tol`), the output produced by the *bisection method* should read

```
Bisection(): k = 1; [a,b] = [-1.000000e+00, 1.000000e+00]; xm = 0.000000e+00; dx = 2.000000e+00; fm = 1.000000e+00
Bisection(): k = 2; [a,b] = [0.000000e+00, 1.000000e+00]; xm = 5.000000e-01; dx = 1.000000e+00; fm = 1.065307e-01
Bisection(): k = 3; [a,b] = [5.000000e-01, 1.000000e+00]; xm = 7.500000e-01; dx = 5.000000e-01; fm = -2.776334e-01
Bisection(): k = 4; [a,b] = [5.000000e-01, 7.500000e-01]; xm = 6.250000e-01; dx = 2.500000e-01; fm = -8.973857e-02
Bisection(): k = 5; [a,b] = [5.000000e-01, 6.250000e-01]; xm = 5.625000e-01; dx = 1.250000e-01; fm = 7.282825e-03
Bisection(): k = 6; [a,b] = [5.625000e-01, 6.250000e-01]; xm = 5.937500e-01; dx = 6.250000e-02; fm = -4.149755e-02
Bisection(): k = 7; [a,b] = [5.625000e-01, 5.937500e-01]; xm = 5.781250e-01; dx = 3.125000e-02; fm = -1.717584e-02
Bisection(): k = 8; [a,b] = [5.625000e-01, 5.781250e-01]; xm = 5.703125e-01; dx = 1.562500e-02; fm = -4.963760e-03
Bisection(): k = 9; [a,b] = [5.625000e-01, 5.703125e-01]; xm = 5.664062e-01; dx = 7.812500e-03; fm = 1.155202e-03
Bisection(): k = 10; [a,b] = [5.664062e-01, 5.703125e-01]; xm = 5.683594e-01; dx = 3.906250e-03; fm = -1.905360e-03
Bisection(): k = 11; [a,b] = [5.664062e-01, 5.683594e-01]; xm = 5.673828e-01; dx = 1.953125e-03; fm = -3.753492e-04
Bisection(): k = 12; [a,b] = [5.664062e-01, 5.673828e-01]; xm = 5.668945e-01; dx = 9.765625e-04; fm = 3.898588e-04
Bisection(): k = 13; [a,b] = [5.668945e-01, 5.673828e-01]; xm = 5.671387e-01; dx = 4.882812e-04; fm = 7.237912e-06
Bisection(): k = 14; [a,b] = [5.671387e-01, 5.673828e-01]; xm = 5.672607e-01; dx = 2.441406e-04; fm = -1.840599e-04
Bisection(): k = 15; [a,b] = [5.671387e-01, 5.672607e-01]; xm = 5.671997e-01; dx = 1.220703e-04; fm = -8.841203e-05
Bisection(): k = 16; [a,b] = [5.671387e-01, 5.671997e-01]; xm = 5.671692e-01; dx = 6.103516e-05; fm = -4.058732e-05
Bisection(): k = 17; [a,b] = [5.671387e-01, 5.671692e-01]; xm = 5.671539e-01; dx = 3.051758e-05; fm = -1.667477e-05
Bisection(): k = 18; [a,b] = [5.671387e-01, 5.671539e-01]; xm = 5.671463e-01; dx = 1.525879e-05; fm = -4.718446e-06
Bisection(): k = 19; [a,b] = [5.671387e-01, 5.671463e-01]; xm = 5.671425e-01; dx = 7.629395e-06; fm = 1.259729e-06
Bisection(): k = 20; [a,b] = [5.671425e-01, 5.671463e-01]; xm = 5.671444e-01; dx = 3.814697e-06; fm = -1.729360e-06
Bisection(): k = 21; [a,b] = [5.671425e-01, 5.671444e-01]; xm = 5.671434e-01; dx = 1.907349e-06; fm = -2.348157e-07
Bisection(): k = 22; [a,b] = [5.671425e-01, 5.671434e-01]; xm = 5.671430e-01; dx = 9.536743e-07; fm = 5.124565e-07
Bisection(): k = 23; [a,b] = [5.671430e-01, 5.671434e-01]; xm = 5.671432e-01; dx = 4.768372e-07; fm = 1.388203e-07
Bisection(): k = 24; [a,b] = [5.671432e-01, 5.671434e-01]; xm = 5.671433e-01; dx = 2.384186e-07; fm = -4.799769e-08
Bisection(): k = 25; [a,b] = [5.671432e-01, 5.671433e-01]; xm = 5.671433e-01; dx = 1.192093e-07; fm = 4.541132e-08
Bisection(): k = 26; [a,b] = [5.671433e-01, 5.671433e-01]; xm = 5.671433e-01; dx = 5.960464e-08; fm = -1.293185e-09
```

- Here `[a,b]` represent the interval at the $k^{th}$ iteration, while $x_m$ is the interval midpoint and $f_m = f(x_m)$.

# Practice Session #1

- Using an interval [-1,1] and a tolerance of $10^{-7}$ (exit condition `|b-a| < tol`), the output produced by the *false potision method* should read

```
FalsePos(): k = 1; [a,b] = [-1.000000e+00, 1.000000e+00]; xm = 7.093967e-01; |del| = 2.000000e+00; fm = -2.174559e-01
FalsePos(): k = 2; [a,b] = [-1.000000e+00, 7.093967e-01]; xm = 6.149498e-01; |del| = 2.906033e-01; fm = -7.428178e-02
FalsePos(): k = 3; [a,b] = [-1.000000e+00, 6.149498e-01]; xm = 5.833191e-01; |del| = 9.444693e-02; fm = -2.527607e-02
FalsePos(): k = 4; [a,b] = [-1.000000e+00, 5.833191e-01]; xm = 5.726287e-01; |del| = 3.163067e-02; fm = -8.587983e-03
FalsePos(): k = 5; [a,b] = [-1.000000e+00, 5.726287e-01]; xm = 5.690049e-01; |del| = 1.069039e-02; fm = -2.916387e-03
FalsePos(): k = 6; [a,b] = [-1.000000e+00, 5.690049e-01]; xm = 5.677752e-01; |del| = 3.623874e-03; fm = -9.901954e-04
FalsePos(): k = 7; [a,b] = [-1.000000e+00, 5.677752e-01]; xm = 5.673578e-01; |del| = 1.229665e-03; fm = -3.361785e-04
FalsePos(): k = 8; [a,b] = [-1.000000e+00, 5.673578e-01]; xm = 5.672161e-01; |del| = 4.173945e-04; fm = -1.141327e-04
FalsePos(): k = 9; [a,b] = [-1.000000e+00, 5.672161e-01]; xm = 5.671680e-01; |del| = 1.416957e-04; fm = -3.874778e-05
FalsePos(): k = 10; [a,b] = [-1.000000e+00, 5.671680e-01]; xm = 5.671517e-01; |del| = 4.810423e-05; fm = -1.315475e-05
FalsePos(): k = 11; [a,b] = [-1.000000e+00, 5.671517e-01]; xm = 5.671461e-01; |del| = 1.633111e-05; fm = -4.465995e-06
FalsePos(): k = 12; [a,b] = [-1.000000e+00, 5.671461e-01]; xm = 5.671443e-01; |del| = 5.544341e-06; fm = -1.516190e-06
FalsePos(): k = 13; [a,b] = [-1.000000e+00, 5.671443e-01]; xm = 5.671436e-01; |del| = 1.882283e-06; fm = -5.147413e-07
FalsePos(): k = 14; [a,b] = [-1.000000e+00, 5.671436e-01]; xm = 5.671434e-01; |del| = 6.390284e-07; fm = -1.747529e-07
FalsePos(): k = 15; [a,b] = [-1.000000e+00, 5.671434e-01]; xm = 5.671433e-01; |del| = 2.169479e-07; fm = -5.932799e-08
```

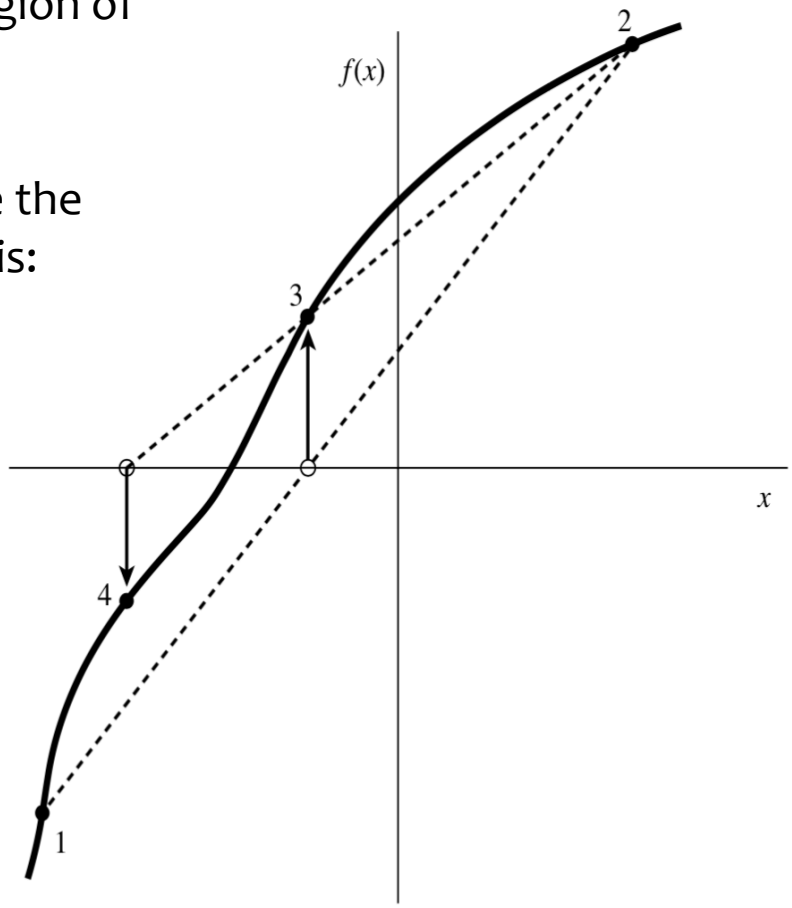
- Here `[a,b]` represent the interval at the $k^{th}$ iteration, while $x_m$ is the interval midpoint and $f_m = f(x_m)$. The distance between endpoints is denoted with `|del|`.

# The Secant Method

- The secant method is a root-finding algorithm which assumes a function to be approximately linear in the region of interest.

- Each improvement is taken as the point where the approximating line (the *secant*) crosses the axis:

$$x^{(k+1)} = x^{(k)} - f^{(k)} \frac{x^{(k)} - x^{(k-1)}}{f^{(k)} - f^{(k-1)}}$$




- Differently from the false position method, the secant method retains only the most recent estimate, so the root does not necessarily remain bracketed.

- The initial interval may not contain the root: in this case, *two* guesses two the root must be provided.

# The Secant Method: Pseudocode

- A simple pseudo code for the secant method is the following:

```
Fa  = F(a), Fb = F(b)          // Initialize
Δx = b-a, k = 1
while (|Δx| > ε){              // Start iteration cycle
   Δx = Fb*(b—a)/(Fb-Fa)       // Compute increment

   a  ← b                      // Shift values
   Fa ← Fb
   b  ← b — Δx
   fb = F(b)
   k++;
}
```

- Note that, as it is the case for previous algorithms, only one function evaluation per iteration is needed.
- Further checks may be necessary:
  - number of iterations should not exceed a certain limit;
  - Interval width should decrease (i.e. solution must not diverge)

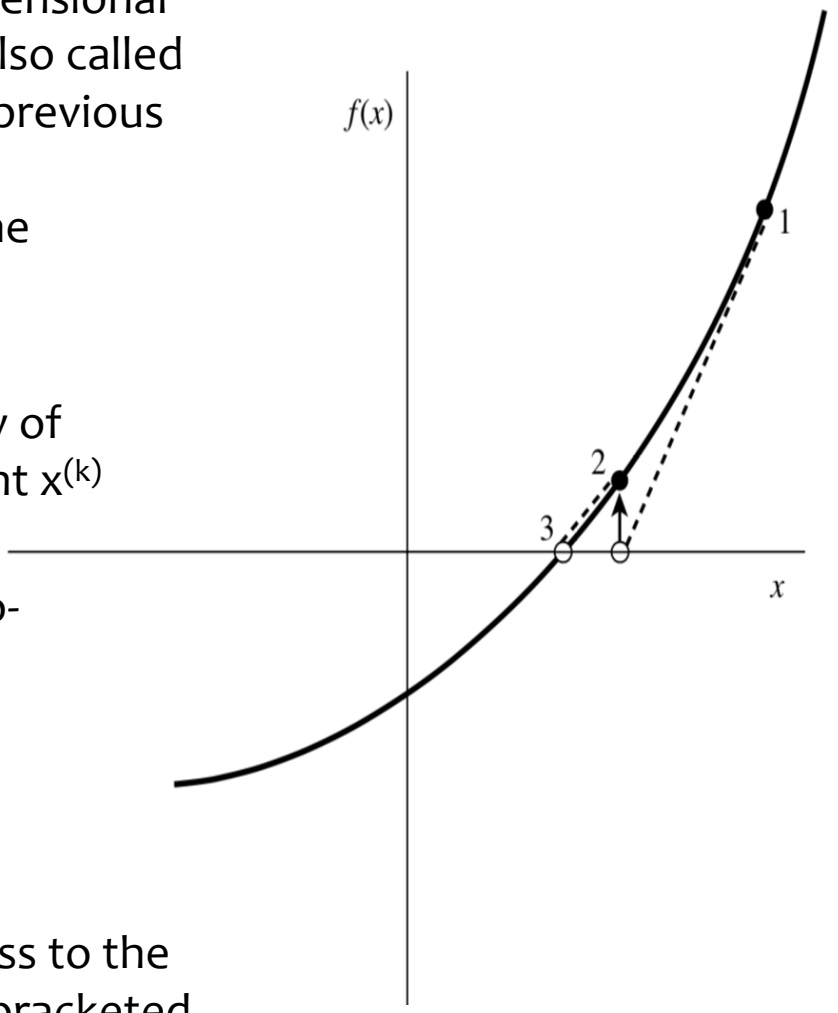
# The Newton Method

- Perhaps the most celebrated of all one-dimensional root-finding methods, *Newton' s method* (also called the *Newton-Raphson method)* differs from previous root finders by the fact that it requires the evaluation of both the function f (x), *and* the derivative df/dx, at arbitrary points x.

- The Newton method consists geometrically of extending the tangent line at a current point $x^{(k)}$ until it crosses zero, then setting the next guess $x^{(k+1)}$ to the abscissa of that zero-crossing:

$$x^{(k+1)} = x^{(k)} - \frac{f^{(k)}}{f'^{(k)}}$$

- The Newton method requires only one guess to the solution, but it does not maintain the root bracketed



# Convergence of the Newton's Method

- The Newton-Raphson method is more powerful than the previous algorithms since it converges quadratically to the solution.

- Within a small distance ε of x, the function and its derivative can be approximated as

$$f(x+\epsilon) = f(x) + \epsilon f'(x) + \epsilon^2 \frac{f''(x)}{2} + \cdots,$$
$$f'(x+\epsilon) = f'(x) + \epsilon f''(x) + \cdots$$

- By Newton's formula: $x^{(k+1)} = x^{(k)} - \dfrac{f^{(k)}}{f'^{(k)}} \rightarrow \epsilon^{(k+1)} = \epsilon^{(k)} - \dfrac{f^{(k)}}{f'^{(k)}}$

where $\varepsilon = x^{(k)} - x_0$ is the difference between a trial value from the true root.

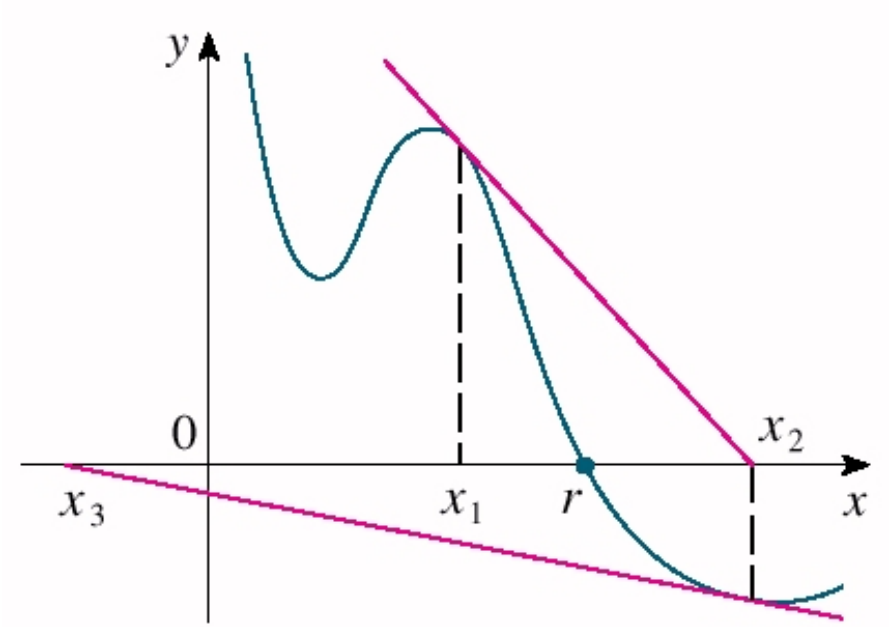
- Using the previous relations we can express $f^{(k)}$ and $f'^{(k)}$ in terms of ε and the derivatives at the root itself.
- The result is a recurrence relation for the deviations of the trial solutions:

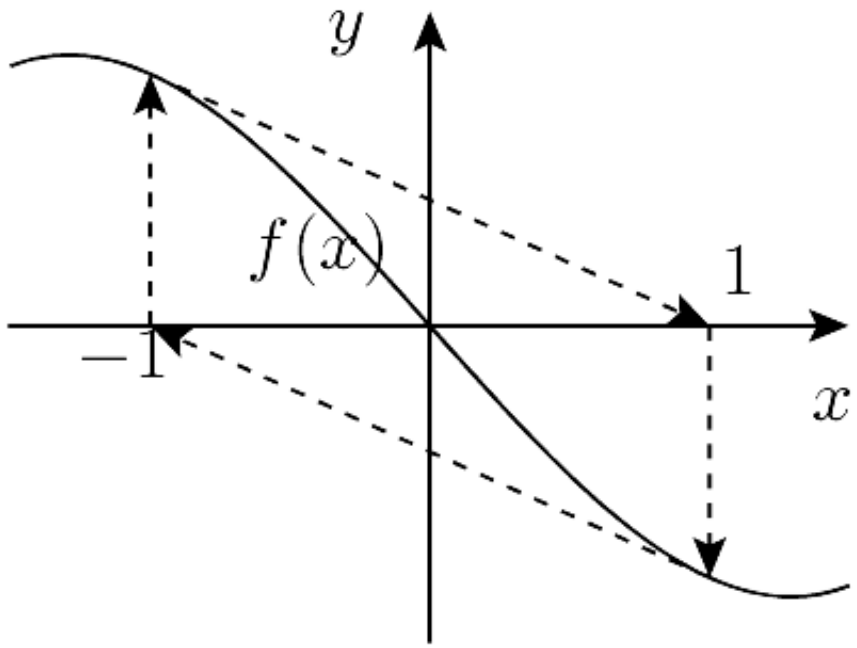
$$\epsilon^{(k+1)} = -\left(\epsilon^{(k)}\right)^2 \frac{f''(x_0)}{2f'(x_0)}$$

# Convergence of the Newton's Method

- Near a root, the number of significant digits approximately *doubles* with each step.
- This very strong convergence property makes Newton-Raphson the method of choice for any function whose derivative can be evaluated efficiently, and whose derivative is continuous and nonzero in the neighborhood of a root.

- However: there're a number of situation in which Newton-Raphson method may fail.



# Summary

- We can summarize, in the table the following properties of the 4 methods explained so far:

| Method | Convergence | MulitD / Complex Plane Extension | Bracket |
|---|---|---|---|
| Bisection | m=1 | No | Yes |
| False Position | 1< m<1.6 | No | Yes |
| Secant | m=1.618 | Yes | No |
| Newton | m=2 | Yes | No |

- On roots with higher multiplicity, Newton and Secant converge at slower rates.

# Practice Session #2

- Add Newton and Secant method to `froot.cpp` and compare the number of iterations with Bisection and False Position.

- Note that, for an efficient realization of Newton-Raphson, the user should provide not only the function f(x) but also its first derivative f'(x) at the desired point x.

- The output of the secant algorithm (using [a,b]=[-1,1] and a tolerance of $10^{-7}$) should read

```
Secant(): k = 1; xa = -1.000000e+00; xb = 1.000000e+00; dx = 2.906033e-01
Secant(): k = 2; xa = 1.000000e+00; xb = 7.093967e-01; dx = 1.523963e-01
Secant(): k = 3; xa = 7.093967e-01; xb = 5.570004e-01; dx = -1.039871e-02
Secant(): k = 4; xa = 5.570004e-01; xb = 5.673991e-01; dx = 2.553492e-04
Secant(): k = 5; xa = 5.673991e-01; xb = 5.671438e-01; dx = 4.702440e-07
Secant(): k = 6; xa = 5.671438e-01; xb = 5.671433e-01; dx = -2.176575e-11
```

where `xa` and `xb` are the most recent iteration and `dx` is the next increment.

# A Special Class of Functions: Polynomials

- Consider $P(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n$

- If you're thinking about doing $F(x) = a_n * pow(x,n) + a_{n-1} * pow(x,n-1) + \ldots a_1 * x + a_0$ by using looping like:

```
double P = 0
for (int i = 0; i <= n; i++) P += a[n]*pow(x,n);     // NOOOOO !!!!!!!
```

*don't even dare!* (It's obvious that there's a lot of repetitive computations being done by raising x to successive powers).

This method is quite inefficient: it requires *n additions* and *n(n+1)/2 multiplications*.

- A possibility would be an *iterative method*, by simply keeping the previous power of x between iterations:

```
double P = 0.0, xn = 1.0;
for (int i = 0; i <= n; i++){
  P += a[i]*xn;
  xn *= x;            // the current power of x
}
```

It's easy to see that there are *2n multiplications* and *n additions* for each computation. The algorithm is now linear instead of quadratic.

# *Horner's Method for Polynomial Evaluation*

- An even cheaper solution is given by Horner's Method. Take

$$P(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n$$

- Divide the polynomial into monomials starting from the largest power: the result obtained from one monomial is added to the result obtained from the next monomial and so forth in an addition fashion. Then you rewrite

$$P(x) = a_0 + x(a_1 + x(a_2 + x(a_3 + \ldots + x(a_{n-1} + x a_n))$$

Each monomial involves a maximum of one multiplication and one addition processes: *n multiplications* and *n additions* are involved !

- With a simple modification, we can also obtain the derivative at the same time:

```
p    = a[n];
dpdx = 0;
for (int j = n-1; j >= 0; j--){
  dpdx = dpdx*x + p;
  p    = p*x + a[j];
}
```

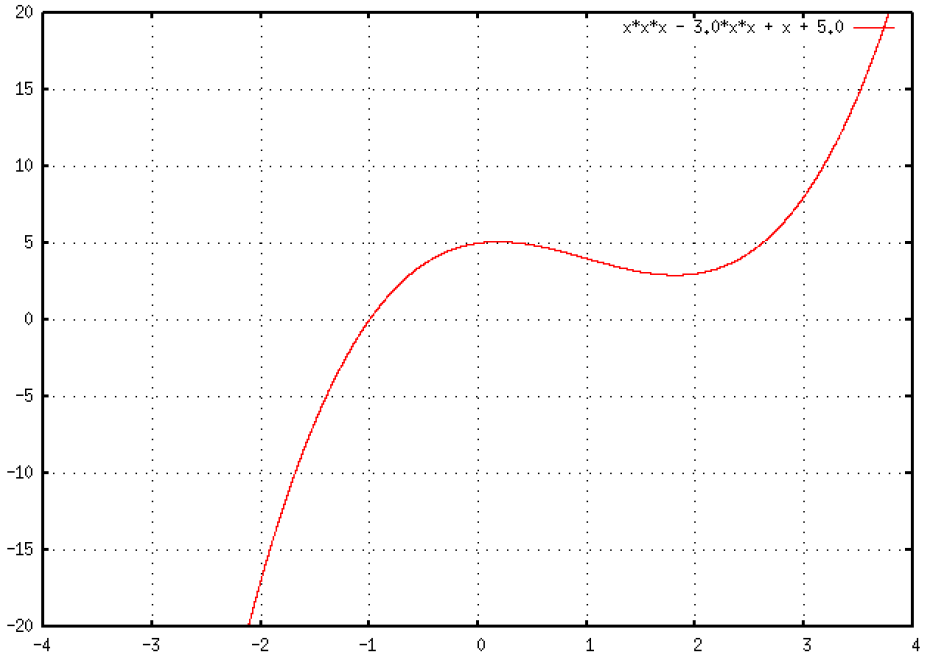
# Practice Session #3

- Consider the function

$$f(x) = x^3 - 3x^2 + x + 5$$

This function has a root in x = -1.
Repeat the search over the interval [-5,0].

The following table gives the number of iterations obtained with the different method using a tolerance $10^{-8}$.




| | Bisection | False Position | Secant | Newton |
|---|---|---|---|---|
| # Iterations | ≈ 30 | ≈ 80 | ≈ 12 | ≈ 6 |

- Can you explain why False Position performs so badly ?
- What happens when the initial interval is reduced to [-2,0] ?

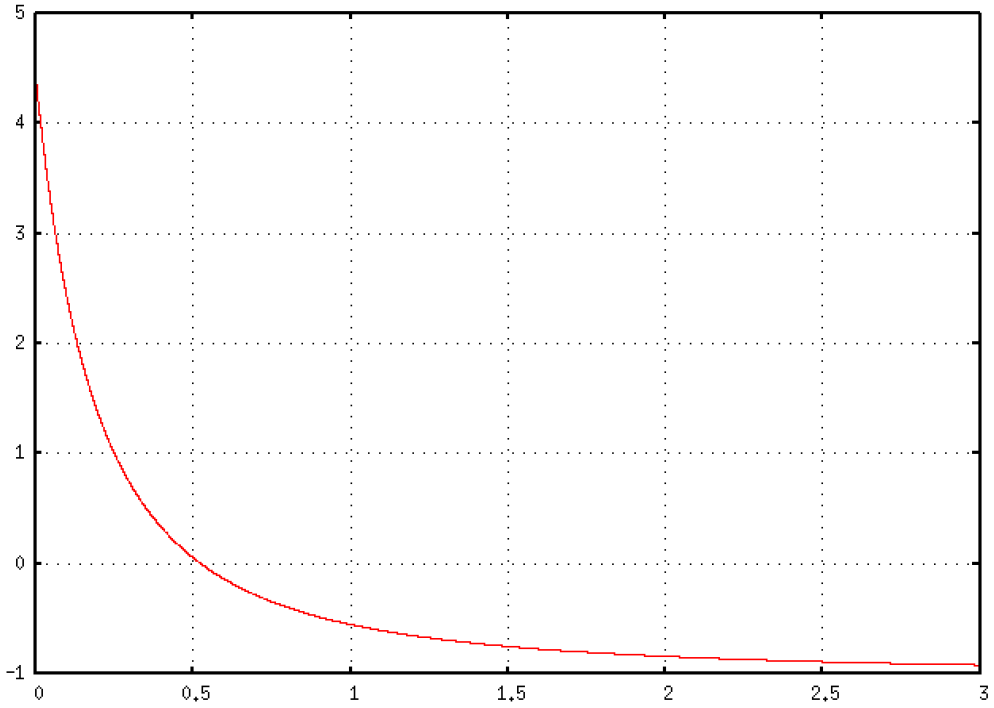
# Practice Session #4

- Consider the function

$$f(x) = \exp\left(\frac{1}{x+1/2}\right) - \frac{3+2x}{1+x}$$

Search the zero over the interval [0,2] with tolerance $10^{-7}$.

Any problem ? Explain.




| | Bisection | False Position | Secant | Newton |
|---|---|---|---|---|
| # Iterations | ≈ 26 | 55 | ?? | ≈ 8 |

# Reorganizing your Code

- As the code becomes larger, we should split the programs up into modules: these would be separate source files: main() would be in one file, the others will contain functions.

- In such a way, we can create our own library of functions by writing a suite of subroutines in one (or more) modules which can then be shared amongst many programs by simply including them at compilation time.

- There are many advantages of spreading a program across several files:
  1. Teams of programmers can work on programs. Each programmer works on a different file.
  2. An object oriented style can be used: each file defines a particular type of object as a data-type and operations on that object as functions. The implementation of the object is private from the rest of the program. This makes for well structured programs which are easy to maintain.
  3. Files can contain all functions from a related group (e.g., all matrix operations). These can then be accessed like a function library.
  4. Well implemented objects or function definitions can be re-used in other programs, reducing development time.
  5. In very large programs each major function can occupy a file to itself. Any lower level functions used to implement them can be kept in the same file. Then programmers who call the major function need not be distracted by all the lower level work.
  6. When changes are made to a file, only that file need be re-compiled to rebuild the program. The UNIX make facility is very useful for rebuilding multi-file programs in this way.

# Reorganizing your Code: an Example

**froot.cpp**

```
#include "my_header.h"

int main()
{
  ...
  root1 = Bisection(...);
  root2 = Secant(...);
  ...
}
```

**my_header.h**

```
// Include headers you may need
#include <iostream>
#include <iomanip>
...

using namespace std;

// Function prototype goes here
double Bisection (...);
double Secant (...);
...
```

**root_solvers.cpp**

```
#include "my_header.h"

double Bisection(...)
{
  ...
  ...
}

double Secant(...)
{
  ...
  ...
}
```

- In the example, the program is split into three parts:
  - **`froot.cpp`**: contains the `main()` function and all problem-dependent functions and definitions;
  - **`my_header.h`**: this is the "header file" containing only definitions of data types, function prototypes and C preprocessor commands. Header files helps to centralize the definitions in one file and share this file amongst the modules.
  - **root_solvers.cpp**: contains a general implementation of the root-finder methods.

# Reorganizing your Code

- How do we compile more than one file ?

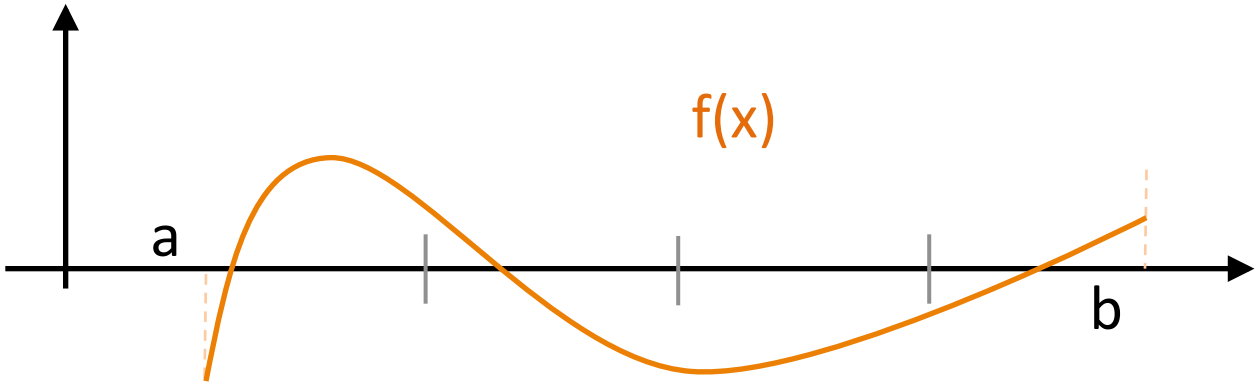
```
> g++ froot.cpp root_solvers.cpp –o program_name
```

- This compilation technique is feasible when
  - Just a few files are present ( < 3-4);
  - All files (including header) reside in the same directory.
  - You do not need to re-use the library files outside the current directory.

- For larger/more complex problem, we shall use the `make` utility that comes with UNIX system (later).

# Bracketing Several Roots

- We will now try to identify all of the roots potentially lying in a given interval [a,b].




- To this end, we divide the initial interval `[a,b]` into `N` equally spaced segments.
- We then cycle over `N` in order to identify those segments across which the function changes sign.
- Count and store them using arrays, e.g. `xL[i]`, `xR[i]`, where `i = 0...`$N_r$`-1`, while $N_r$ is the number of bracketing intervals (e.g. in the figure above `N = 4,` $N_r$ `= 3`).
- For each segment, use your favorite root finder scheme to refine search and pinpoint the zero of the function with accuracy.

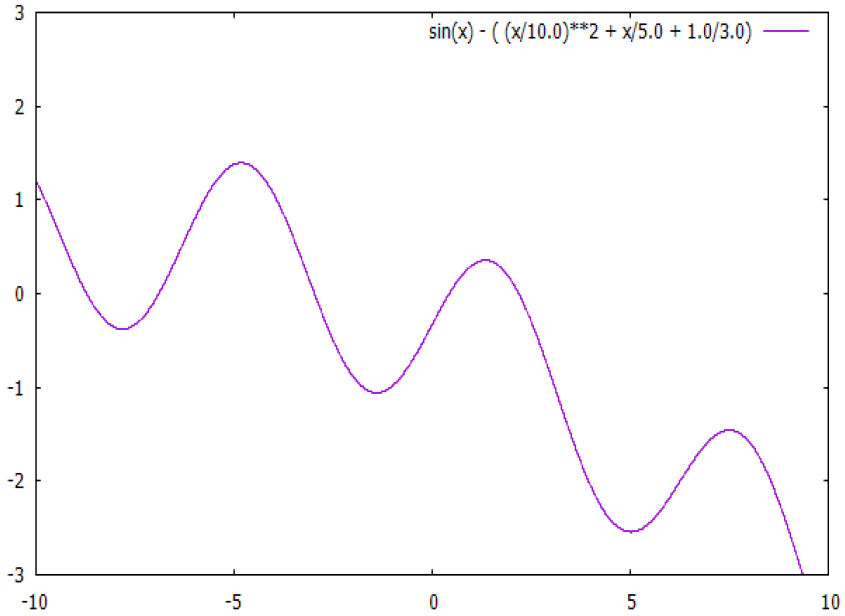
# Practice Session #5

- Modify your program to find all of the zeros of the function:

$$F(x) = \sin(x) - \left[\left(\frac{x}{10}\right)^2 + \frac{x}{5} + \frac{1}{3}\right]$$

by first using a bracketing function and then any of the root finder methods to refine the search.

This function has five roots in the interval [-10,10], corresponding to:




| Root #1 | Root #2 | Root #3 | Root #4 | Root #5 |
|---|---|---|---|---|
| -8.716925e+00 | -6.889594e+00 | -2.968485e+00 | 4.361680e-01 | 2.183971e+00 |

Are you able to find them all ? Which root-finder performs the best ?

# Practice Session #6

- Write a code that can found the zero of Legendre polynomial of arbitrary order **n**.

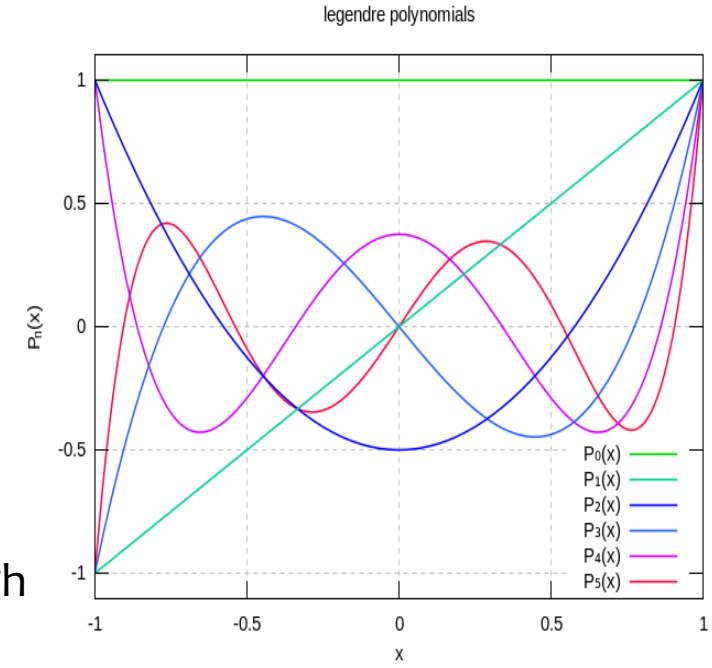
  In order to evaluate Pn(x) use Bonnet's recursion formula:

$$(n+1)P_{n+1}(x) = (2n+1)xP_n(x) - nP_{n-1}(x)$$

  with Po(x) = 1, P1 (x)= x.

  For Newton-Raphson method, the derivative may be computed in terms of Pn(x) and Pn-1(x) through

$$\frac{x^2-1}{n}\frac{dP_n(x)}{dx} = xP_n(x) - P_{n-1}(x)$$




- Now that you have found the roots with high precision, you may use them to produce your Gaussian weights:

$$w_i = \frac{2}{(1-x_i^2)[P_n'(x_i)]^2}$$

(hints: to increase the number of digits in your output, use (e.g.)

```
cout << setprecision(12);
```