LAB 03

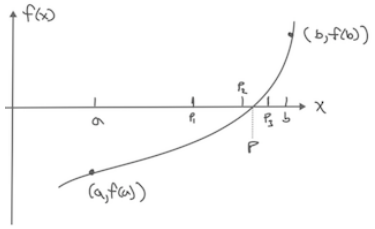
Finding Zeroes and Minima of Functions

Initial Setup:
1. Create a project, called Lab03 for example, in Visual Studio (or Xcode in OSX) and follow the instructions below.

Exercise 1: The Bisection method: Finding zeroes of a function.

Setup:
Create a file called `bisection.cpp`

Description:



The problem to solve here is to find a value $x = p$ that satisfy the equation $f(x) = 0$, for a function $f(x)$ of a single real value x, i.e., finding the zero of a function. The Bisection method searches for a solution p in the interval $[a, b]$, where a solution is known to exist. Is an iterative method that starts by searching in the middle point $p = (b + a)/2$ of the interval $[a, b]$, and test to see if that middle point solves the problem $|f(p)| < \epsilon$, where ϵ is a small tolerance. If that is true, or if the interval is very small $((b - a)/2 < \epsilon)$ it ends with solution p. Otherwise, it move either a or b to the middle point such that the solution is still contained in $[a, b]$. A sequence of values of x $\{p_1, p_2, p_3, \ldots\}$ is generated that quickly approach the solution. If it fails to find a solution in less than N iterations, it writes a fail to converge message. The algorithm requires that $f(a)$ and $f(b)$ have oposite signs. In an iteration, if the sign of $f(p)$ is the same as $f(a)$, then b is set to p, otherwise a is set to p.

Task:
Write a function to implement the bisection method. The function should take as parameters a pointer to a function of a single variable, the limits a and b, the maximum number of iterations N, and the tolerance ϵ. Write another function to output the progress of the algorithm as the interval shrinks. The function to be analyzed should be in another function by itself. The main function should define the input parameters, run the bisection algorithm, and write the results to the console as shown in the example below.

Input: (Hardcoded)
The initial boundary points $[a, b]$.
The tolerance ϵ.
The maximum number of iterations N.

Output:

The value that makes the function zero.

Optional: The interval and best solution as it converges

Run and Output example:

```
Enter endpoints a and b surrounding solution: 1 2
  i      a         b         p        fp
  1   1.00000   2.00000   1.50000  -1.37500
  2   1.50000   2.00000   1.75000   2.25000
…
17   1.59999   1.60001   1.60000  -0.00002
A zero was found at p = 1.60000
```

Pseudo code: (use given variable names and only discussed keywords when specified)

```
//  bisection.cpp
//  SCI120 Lab
//
//  This program iteratively searches for the zeroes of a 1-D function and
//  display its progress to the standard output.
//
// NOTE: user defined identifiers are shown surrounded by < >.
// DO NOT TYPE THE < > as part of the names.
//
// PROGRAM STARTS BELOW THIS LINE -----------------------------------------

// CODE: Include only all the necessary headers and namespaces


// --- PRINT_PARAMETERS FUNCTION ----------------------------------------
//  CODE: Define a function called <printpars>. It does not returns a value
//  and takes five parameters: an integer <i>, and four doubles named <a>,
//  <b>, <p>, and <fp>, respectively. The function contains the following:

//      1. CODE: if <i> is one,
//          A. CODE: Print the string literal
//          "  i      a         b         p        fp" on a line by itself.
//          B. CODE: Format the standard output to show floats to five
//          decimal places
//      2. CODE: write <i> to the std out using a width of 3 places
//      3. CODE: write <a> to the std out using a width of 10 places
//      4. CODE: write <b> to the std out using a width of 10 places
//      5. CODE: write <p> to the std out using a width of 10 places
//      6. CODE: write <fp> to the std out using a width of 10 places
//      7. CODE: write a new line to the std out


// --- BISECTION ALGORITHM ----------------------------------------------
//  CODE: Define a function called <bisection> that returns a double.
//  It takes five parameters: a function named <f> that takes a double
//  and returns a double (i.e, a functional), three doubles named <a>, <b>,
```

```
//  and <eps>, and an integer called <N>.
//  The function contains the following:

//      1. CODE: Call the function <f> twice with argument <a> and with <b>.
//      If the product of the return values is greater than zero, then
//          A. CODE: Write to the std out the message below:
//              Bisection Error: invalid endpoints.
//              function must have opposite signs at endpoints.
//          B. CODE: Return with value 1

//      2. CODE: Write a loop with integer index <i> with values
//      i = 1, 2, 3, ..., N. In the loop:
//          A. CODE: Define a double variable called <p>, initialized to
//          (a+b)/2.
//          B. CODE: Define a double type variable called <fp>, initialized
//          to the value returned by <f> with argument <p>.
//          C. CODE: Call the function <printpars> with arguments <i>, <a>,
//          <b>, <p> and <fp>
//          D. CODE: If either (b-a)/2 or the absolute value of <fp> are
//          less than <eps> then return <p>
//          E. CODE: Call <f> with argument <a> and if the return value
//          times <fp> is positive, then set a = p, otherwise set b = p.
//      3. CODE: (aft. loop) Write the message:
//              Method failed to converge after N iterations
//      replacing N by its actual value.
//      4. CODE: Return with value 1


// --- MATH FUNCTIONS WITH ZEROS -----------------------------------------
//  CODE: Define a function called <fun> that returns a double and takes one
//  parameter of type double called <x>. In the body of the function:

//      1. CODE: Evaluate and return 5x^2 - 1.75x - 10 using proper C++ math.


// --- MAIN FUNCTION -----------------------------------------------------
//  CODE: Define the <main> function with the following statements:

//      . CODE: Write to the console the prompt:
//              Enter endpoints a and b surrounding solution:
//      . CODE: Define two double precision variables called <a> and <b>
//      . CODE: Read the values for <a> and <b> from the standard input.
//      . CODE: If a is greater than b,
//              * CODE: write to the console the message:
//                      Error: invalid endpoints
//              * CODE: return with value 1

//      . CODE: (aft. Loop) Define a double type variable called <eps>
//      initialized to 10^-5 (use C++ scientific notation!)
//      . CODE: Define an integer variable called <N> initialized to 100

//      . CODE: Call the <bisection> function with arguments, <fun>, <a>,
//      <b>, <eps> and <N>, and use the returned value to initialize a
//      double variable <p>

//      . CODE: Write to the console the message:
//              A zero was found at p = #
//      replacing # by the actual p value
```

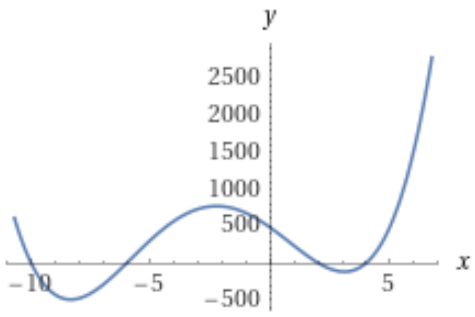
Programming Notes:

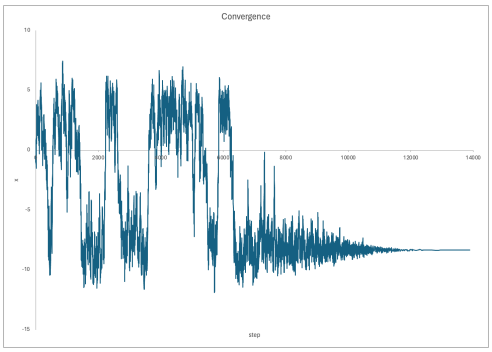
- You may need to adjust the steps N or the tolerance ϵ for different functions.
- There are better algorithms for calculating zeros, such as the Newton-Rapson method, but are more conceptually complex.

Exercise 2: The Monte Carlo method for finding the global minimum of a function.

Setup:
Create a file called `montecarlominima.cpp`

Description:
Monte Carlo methods are a broad class of computational algorithms that rely on repeated random sampling to obtain numerical results. The underlying concept is to use randomness to solve problems that might be deterministic in principle. Monte Carlo methods are mainly used in three distinct problem classes: optimization, numerical integration, and generating draws from a probability distribution. They can provide approximate solutions to problems that are otherwise intractable or too complex to analyze mathematically.
We will use the Monte Carlo method for finding the value x for which a function f(x) is the global minimum. The algorithm begins with a starting x value (entered by the user). This value is modified by a small amount Δx and the change $\Delta f = f(x + \Delta x) - f(x)$ is evaluated. If this change is negative (i.e., the function has a lower value) the change $x + \Delta x$ is accepted. Otherwise, the change is accepted, but only with a probability given by $\exp(-\Delta f / T)$, where T is a temperature. The latter criteria allows for "up-hill" changes in $f(x)$, which allows to escape local minima. The temperature parameter controls how accepting is the simulation to an increase in f. The figure below shows a function, given by $480 - 232x - 28x^2 + 10x^3 + x^4$, with two minima. The global minimum is at $x \simeq -8.348$. The simulation converges to this minimum by gradually lowering the temperature in a process called *simulated annealing*. This is shown in the second figure.





Task*:*
Implement the Monte Carlo method above (called the Monte Carlo method of Metropolis) as a separate function. The input to this function includes a pointer to a function of one variable, the initial value of x, the temperature, and the simulation time. It should loop over the simulation steps, in which it will compute a change Δx at random between -1.0 and 1.0 and the change Δf. Test for the acceptance criteria and if the change is accepted, record the new minimum value if it is lower than the previous. Also record the value of x at this point. After each step s, compute the temperatures from the formula $T = T_0[1 - (s/N)^2]$. Optionally, write the simulation intermediate steps to a csv file. Return the results to the main function and write to the standard output.

Input:
Starting value x.
Temperature T
maximum Steps = 50000.

Output:
The *x* value for the global minima and the value of the function at that point.
Optional: A csv file with the simulation trajectory (step vs x)

Run and Output example:

```
Starting x value: 0
Minima at x = -8.3448
Minima value f(x) = -495.636
```

`convergence.csv` first few lines:

```
1,-0.999984
2,-0.488774
3,-0.571473
4,-1.13356
5,-0.775826
6,-0.417233
7,0.452153
8,0.219157
```

Pseudo code: (use given variable names and only discussed keywords when specified)

```
//  montecarlominim.cpp
//  SCI120 Lab
//
//  This program searches for the global minimum of a function using the
//  Monte Carlo method of Metropolis with a slowly decreasing temperature
//  (annealing) and writes the search trajectory to a CSV file.
```

```
//  The output data can be read into a program such as Excel and plotted.
//
// NOTE: user defined identifiers are shown surrounded by < >.
// DO NOT TYPE THE < > as part of the names.
//
// PROGRAM STARTS BELOW THIS LINE ----------------------------------------

// CODE: Include only all the necessary headers and namespaces

// --- MC_MINIMA FUNCTION ----------------------------------------------------
//  CODE: Define the <mc_minima> function. It returns a double and takes four
//  parameters: a function named <f> that takes a double and returns a double
//  (i.e, a functional), two doubles named <x> and <T0>, and a long long
//  integer called <simSteps>. The function contains the following:

//      1. CODE: Define a random_device object called <rg>
//      2. CODE: Define a uniform_real_distribution object called <randDx> of
//          type double, with a rang of [-1,1)
//      3. CODE: Define a uniform_real_distribution object called <randOne>
//          of type double, with a rang of [0,1)
//      4. CODE: Define a stream object for writing to a file called <outf>,
//      and use it to open a file called convergence.csv.
//      5. CODE: Define a double-type variable called <fx>, initialized with
//      the returned value from calling the function <f> with argument <x>
//      6. CODE: Define a double-type variable called <xmin>, initialized
//      to <x>
//      7. CODE: Define a double-type variable called <fmin>, initialized
//      to <fx>
//      8. CODE: Define a double-type variable called <T>, initialized
//      to <T0>

//      9. CODE: Write a loop with integer index <step> with values
//      step = 1, 2, 3, ..., simSteps. In the loop:

//          A. CODE: Define a double-type variable called <dx>,
//          initialized the returned value of randDx(rg)
//          B. CODE: Define a double-type variable called <fxdx>,
//          initialized with a call to <f> with argument x+dx
//          C. CODE: Define a double-type variable called <df>,
//          initialized to <fxdx> minus <fx>

//          D. CODE: If <df> is less than or equal to 0, or if
//          a call to randOne(rg) returns a value less than exp(-df/T),
//          then do the following:
//              1) CODE: Increment <x> by <dx>
//              2) CODE: Set <fx> to <fxdx>
//              3) CODE: If <fx> is less than <fmin>
//                  a) CODE: Set <xmin> to <x>
//                  b) CODE: Set <fmin> to <fx>
//              4) CODE: (aft. if) Write to <outf>, <step>, a literal comma,
//              <x>, and an end line.

//          F. CODE: (aft. if). Set <T> to the function T0 [1 - (step/simStep)^2],
//          making sure that is properly type casted and using C++ math.

//      10. CODE: (aft. loop) return the value of <xmin>
```

```
// --- MATH FUNCTIONS WITH MINIMA ----------------------------------------
//  CODE: Define a function called <func> that returns a double and takes
//  one parameter of type double called <x>. In the body of the function:

//      1. CODE: Evaluate and return 480 – 232x – 28x^2 + 10x^3 + x^4
//      using proper C++ math.


// --- MAIN FUNCTION -----------------------------------------------------
//  CODE: Define the <main> function with the following statements:

//      . CODE: Write to the console the prompt:
//              Starting x value:
//      . CODE: Define a double precision variable called <xi>
//      . CODE: Read <xi> from the standard input

//      . CODE: Define a double precision variable called <T>, initialized
//      to 500.

//      . CODE: Define a long long integer variable called <maxStep>,
//      initialized to 50000

//      . CODE: Call the <mc_minima> with arguments <func>, <xi>, <T>, and
//      <maxStep>, and use its returned value to initialize a double variable
//      named <xmin>

//      . CODE: Write to the standard output
//              Minima at x = #
//      replacing # with the value of <xmin>
//      . CODE: Write to the standard output
//              Minima value f(x) = #
//      replacing # with the value of calling <func> with argument <xmin>
```

Programming Notes:

- Optional: write the polynomial as $480 + (-232 + (-28 + (10 + x)x)x)x$ for efficiency.
- For debugging, hard code all input and don't use time in the random number generator initialization function so that the errors are repeated until you find them.
- This algorithm is flexible, in particular the temperature annealing function and the size of the change in x (i.e., dx).
- In statistical physics, the Metropolis approach is used to compute equilibrium probability distributions using $\exp(-\Delta E / k_B T)$, where ΔE is a change in energy and k_B is the Boltzmann constant. Often, the function generating the energy comes from a simulation, such as the energy of a system of interacting molecules.
- For straight forward down hill minimization, which doesn't require escaping local minima, methods such as Conjugate Gradient methods are more efficient. This and other methods are used for training machine learning algorithms.