LAB 04

Numeric Calculus and Ordinary Differential Equations

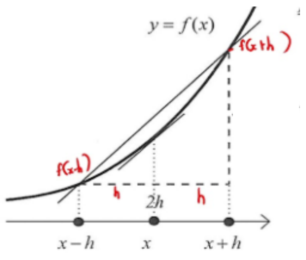
Initial Setup:

1. Create a project, called Lab04 for example, in Visual Studio (or Xcode in OSX) and follow the instructions below.

Exercise 1: Computing Slopes and Areas Under (and Over) Curves (i.e., Differentiation and Integration)

Setup:
Create a file called `calculus.cpp`

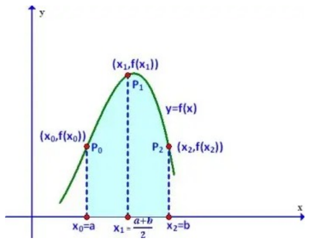
Description:
Calculus is the mathematical study of continuous change, for example, dealing with functions over the real or complex numbers. There are two major branches: differential calculus and integral calculus. The former concerns instantaneous rates of change and the *slopes* of curves, while the latter concerns accumulation of quantities and *areas* under (or above) curves. These calculations are essential across all areas of science and technology and even other fields such as economics. When the curves are straight lines, the calculation of slopes and areas are trivial. However, for arbitrary curves is more complicated.



In one dimension, a numerical approach is to partition the curve $f(x)$ into small segments, each of which could be approximated by a straight line or a polynomial of low degree. We will use this approach to compute slopes and areas "under" the curve. For slopes, we will use the so called three point formula, which is similar to computing the rise vs the run, except using three points (or more precisely, the two points around a point). The formula for the slope at x is

$$\text{slope}(x) = [-f(x-h) + f(x+h)]/2h,$$

where h is the step size between points.



For integration, we will use the Simpson 1/3 rule, which computes the area between points a and b:

$$\text{area}(a,b) = \frac{h}{3}\left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right]$$

where $h = (b-a)/2$ is the step size. The area for a larger portion of a curve is computed by adding all the 3-point segment contributions.

Task:
Write functions for computing the slope and area according to the formulas above. Write other functions to call the above functions for a range of points x_i, along the function f(x) passed as an arguments. As either the slope and the areas are computed, the results should be written to a csv file. In the case of the area, the cumulative area is written.

The main function should call these functions for given boundaries *a,b* and partition steps *h*. Test the function with the normalized Gaussian function $f(x) = \exp(-x^2/2)/\sqrt{2\pi}$.

Input: (Hardcoded)
function *f(x)*; boundaries *a*, *b*; partition size *n*.

Output:
Two csv files with the derivative (slope) and integral (area) as a function of *x*.

Run and Output example:

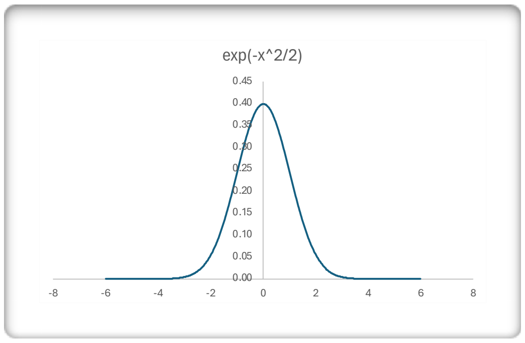
```
Integral from -6 to 6 is 1
```

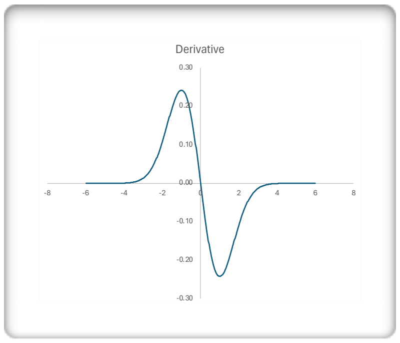
First few lines of `derivative.csv:`

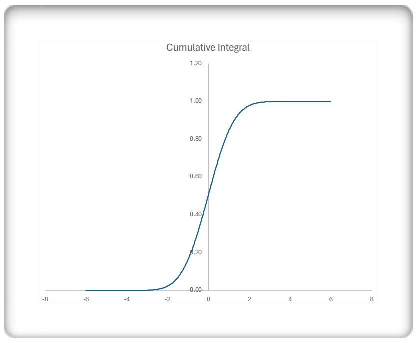
```
-5.976,4.20529e-08
-5.952,4.83281e-08
-5.928,5.55068e-08
-5.904,6.37141e-08
```

First and last few lines of `integrate.csv:`

```
-6,1.56827e-10
-5.976,3.37839e-10
-5.952,5.46645e-10
...
5.928,1
5.952,1
5.976,1
```







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

```
//  calculus.cpp
//  SCI120 Lab
//
```

```
//  This program will compute the derivative and cumulative integral of a
//  function and write the results to two files. This 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


// --- DIFFERENTIATION FUNCTION ---------------------------------------------
//  CODE: Define a function called <differentiation>. It returns a double
//  and takes three parameters:
//      . a function object named <f> that takes a double and returns
//      a double
//      . a double named <x>
//      . a double named <h>
//  The function contains the following single statement:
//      1. CODE: Return the results of evaluating
//
```

$$\frac{-f(x-h)+f(x+h)}{2h}$$

```
// --- DERIVATIVE FUNCTION --------------------------------------------------
//  CODE: Define a function called <derivate>. It does not returns a value
//  and takes five parameters:
//      . a function object named <f> that takes a double and returns
//      a double
//      . a double named <a>
//      . a double named <b>
//      . an integer named <n>
//      . a string named <fname>
//  The function contains the following statements:
//      1. CODE: Define an output stream object named <fout> and use it to
//      open the file indicated by <fname>
//      2. CODE: Define a double variable named <dx> and initialized it with
//                  (b-a)/n
//      3. CODE: Create a loop for a double variable named <x>, starting
//      from a+dx and ending before <b>, in steps of <dx>. That is
//          x = a+dx, a+2dx, ... , b-dx
//      The loop executes the following:
//          A. CODE: Write to <fout> the value of <x>, a literal comma, and
//          the returned value of <differentiation> called with arguments
//          <f>, <x>, and <dx>, all in one line


// --- INTEGRATION FUNCTION -------------------------------------------------
//  CODE: Define a function called <integration>. It returns a double
//  and takes three parameters:
//      . a function object named <f> that takes a double and returns
//      a double
//      . a double named <a>
//      . a double named <b>
//  The function contains the following :
//      1. CODE: define a double variable named <h> and initialize it with
//      the following formula:
```

```
//              h = (b-a)/2
//      2. CODE: Return the results of evaluating
//                  $\frac{h}{3}\left[f(a)+4f\left(a+h\right)+f(b)\right]$


// --- INTEGRATE FUNCTION ---------------------------------------------------
//  CODE: Define a function called <integrate>. It returns a double and
//  takes five parameters:
//      . a function object named <f> that takes a double and returns
//      a double
//      . a double named <a>
//      . a double named <b>
//      . an integer named <n>
//      . a string named <fname>
//  The function contains the following statements:
//      1. CODE: Define an output stream object named <fout> and use it to
//      open the file indicated by <fname>
//      2. CODE: Define a double variable named <dx> and initialize it with
//                  (b-a)/n
//      3. CODE: Define a double variable named <Itot> and initialized to 0
//      4. CODE: Create a loop for a double variable named <x>, starting
//      from <a> and ending before <b>, in steps of <dx>. That is
//          x = a, a+dx, a+2dx, ... , b-dx
//      The loop executes the following:
//          A. CODE: Add to <Itot> the returned value of <integration>
//          called with arguments <f>, <x>, and x + dx
//          B. CODE: Write to <fout> the value of <x>, a literal comma, and
//          <Itot>, all in one line
//      5. CODE: (aft. the loop) return the value of <Itot>


//  CODE: Define a global named-constant of type double called <SQRT2PI>,
//  initialized to $\sqrt{2\pi}$ (the square root of two times pi).

// --- GAUSS FUNCTION -------------------------------------------------------
//  CODE: Define a function called <gauss>. It returns a double and
//  takes one parameters: a double named <x>. The function contains the
//  following statement:
//      1. CODE: return the results of calculating
//              $\exp(-x^2/2)/\sqrt{2\pi}$
//          using <SQRT2PI> in place of $\sqrt{2\pi}$


// --- MAIN FUNCTION -------------------------------------------------------
//  CODE: Define the <main> function with the following statements:
//      . CODE: Define a double variable named <a> initialized to -6
//      . CODE: Define a double variable named <b> initialized to 6
//      . CODE: Define an integer variable named <n> initialized to 500
//      . CODE: Call <derivate> with arguments <gauss>, <a>, <b>, <n>,
//      and the string literal "derivative.csv"
//      . CODE: Call <integrate> with arguments <gauss>, <a>, <b>, <n>,
//      and the string literal "integrate.csv". Assign the return value
//      to a double called <intabn>
//      . CODE: Write to the standard output the line
//          Integral from <a> to <b> is <intabn>
//      with the variables in brackets replaced by their values
// --- END
```

Programming Notes:

- There are variations of these algorithms using more than three points.

Exercise 2 (optional): Runge Kutta - Method for solving ordinary differential equations (ODEs)

Setup:
Create a file called `rk4.cpp`

Description:
In calculus, the derivative of a function $y(t)$ is basically the slope of the function at t, written as $\Delta y/\Delta t$ for a small interval Δt (or in calculus notation dy/dt). In turn, a differential equation is an equation (or equations) that relates a function and its derivatives. Solving this equation mean finding this function. An ordinary differential equation (ODE) is a differential equation dependent on only a single independent variable t. For example, in physics, Newton's equations of motion for the position y and velocity v of a particle as a function of time are given by

$$\frac{dv}{dt} = F(y, v)$$

$$\frac{dy}{dt} = v$$

indicating the rate of change of v and y. Note that $F(y, v)$ may also depend explicitly on the independent variable t.

The numerical solution of an ODE involves replacing derivatives dy/dt by small interval slopes $\Delta y/\Delta t$ and turning them into maps that approaches the solution interactively. For example, a crude solution to the Newton's equations above would be to iterate $v_{i+1} = v_i + F(y_i, v_i)h$ and $y_{i+1} = y_i + v_i h$, for a small steps $\Delta t = h$ and initial values y_0 and v_0.

A better approximation is given by the Runge Kutta 4 (RK4) method. For one dependent variable y and the ODE given by $dy/dt = f(y, t)$ with $y(t_0) = y_0$, the RK4 method is

$$t_{n+1} = t_n + h$$
$$y_{n+1} = y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

where

$$k_1 = f(y_n, t_n)$$
$$k_2 = f(y_n + hk_1/2,\ t_n + h/2)$$
$$k_3 = f(y_n + hk_2/2,\ t_n + h/2)$$
$$k_4 = f(y_n + hk_3,\ t_n + h)$$

and where $h = \Delta t$. In the Newton's example, the calculation must be done for two dependent variables y_n and v_n.

Task:

Write program to solve an ODE using the RK4 method. Pass the derivatives function *f* as an argument to the RK4 function and iterate for many steps. The RK4 function should work for one or more variables. Write the results of all the variables as a function of the independent variables to a csv file. Test by solving it for the Lorenz equations shown below.

Input: (Hardcoded)

The ODE function in any number of variables
Initial conditions for each variable
The total integration time, `time`
Integration step `dt`
Frequency in which the trajectory is written to a file, `frequency`

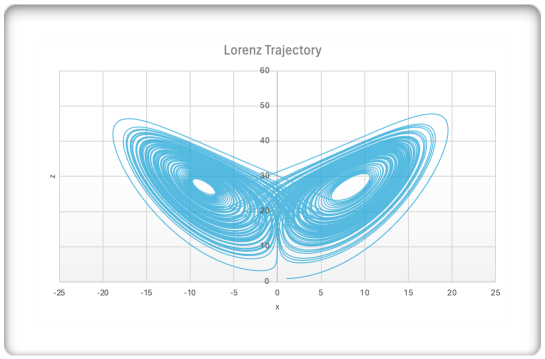
Output:

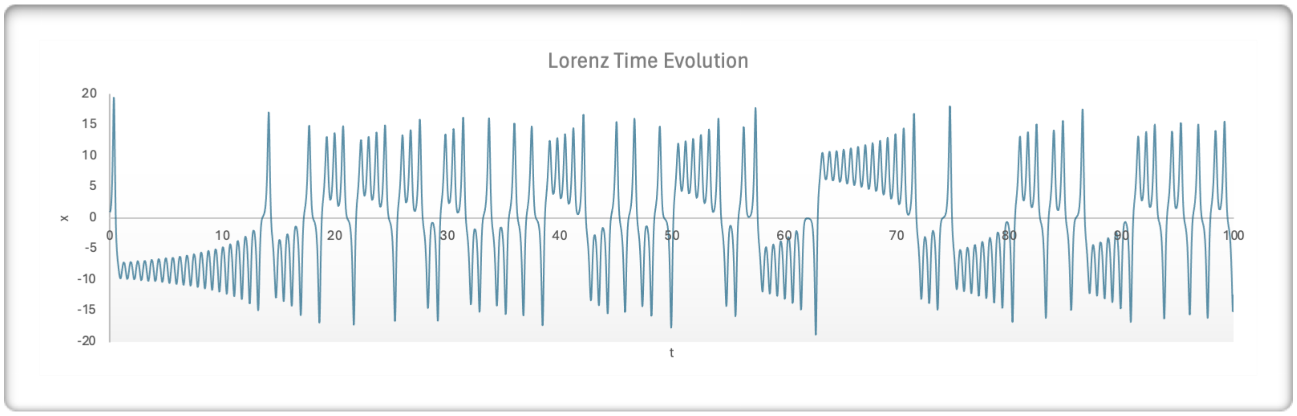
`trajectory.csv` file with the trajectory for all the dependent variables as function of the independent one

Run and Output example:

First few lines of `trajectory.csv:` (time, x, y, z) for a system with three variables

```
0.01,1.01257,1.25992,0.984891
0.02,1.04882,1.524,0.973114
0.03,1.10721,1.79831,0.965159
0.04,1.18687,2.08855,0.961737
0.05,1.28755,2.40016,0.963806
...
```





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

```
//  rk4.cpp
//  SCI120 Lab 04
//
//  This program solves an ODE in several variables using the Runge Kutta 4
//  method, given some initial conditions.
//
// NOTES:
// . user defined identifiers are shown surrounded by < >. DO NOT TYPE THE
//   < > as part of the names.
// . Access to some arrays elements are represented with an underscore, e.g.,
//   y_1 means y[1]
//
// PROGRAM STARTS BELOW THIS LINE -----------------------------------------

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


// --- GET_YT FUNCTION ----------------------------------------------------
//  CODE: Define a function called <get_yt>. It returns a vector of
//  doubles takes three parameters:
//      . a vector of doubles named <y>
//      . a vector of doubles named <dydx>
//      . a double named <hh>
//  The function contains the following statements:
//      1. CODE: Define variable with appropriate type called <N> and
//      initialize it with the number of elements of <y>
//      2. CODE: Create a loop with integer control variable <i>,
//      for i=0,...,N-1. The loops contains the following:
//          A. Add to <y> with index <i> the amount <hh> times <dydx> with
//          index <i>
//      3. CODE: return the vector <y>


// NOTE: rk4 is the implementation of the Runge Kutta 4 algorithm, with
// no explicit time dependence and dependent variables given by the array y
// --- RUNGE KUTTA FUNCTION -----------------------------------------------
//  CODE: Define a function called <rk4>. It returns a vector of
//  doubles takes four parameters:
//      . a function object named <derivs> that returns a vector of
//      doubles and takes two (nameless) parameters: a vector of doubles
//      and a double
//      . a vector of doubles named <y>
//      . a double named <x>
//      . a double named <h>
//  The function contains the following statements:
//      1. CODE: Define a double variable called <h6> initialized to h/6.
//      2. CODE: Define a double variable called <hh> initialized to h/2.
//      3. CODE: Define a double variable called <xh> initialized to x + hh

//      4. CODE: Define a vector of doubles called <yt> initialized
//      to <y>
//      5. CODE: Call <derivs> with arguments <yt> and <x> and save the
//      returned value in a vector of doubles called <D1>.

//      6. CODE: Call get_yt with arguments <y>, <D1>, and <hh>, and save
//      the returned value in <yt>
//      7. CODE: Call <derivs> with arguments <yt> and <xh> and save the
//      returned value in a vector of doubles called <D2>.
//      8. CODE: Call get_yt with arguments <y>, <D2>, and <hh>, and save
//      the returned value in <yt>
//      9. CODE: Call <derivs> with arguments <yt> and <xh> and save the
//      returned value in a vector of doubles called <D3>.

//      10. CODE: Call get_yt with arguments <y>, <D3>, and <h>, and save
//      the returned value in <yt>
//      11. CODE: Call <derivs> with arguments <yt> and x + h and save the
//      returned value in a vector of doubles called <D4>.

//      13. CODE: Create a loop for all elements of <y>, with index <i>,
//      that does the following:
//          A. CODE: Add to <y> with index <i> the amount
//              h6*[D1_i + 2*(D2_i + D3_i) + D4_i]
//          Note: D1_i is the element of <D1> with index <i>. Change to C++

//      14. CODE: (aft. loop) return the vector <y>


// --- LORENZD FUNCTION -----------------------------------------------------
//  CODE: define a double named-constant called <sigma> with value 10.
//  CODE: define a double named-constant called <beta> with value 8/3.
//  CODE: define a double named-constant called <rho> with value 28.
//  CODE: Define a function called <lorenzD>. It returns a vector of
//  doubles takes two parameters:
//      . a vector of doubles named <y>
//      . a double named <x>
//  The function contains the following statements:
//      1. CODE: Define three double variables <dy0>, <dy1>, and <dy2>
//      initialized as follows:
//          dy0 = sigma·(y_1 - y_0)
//          dy1 = y_0·(rho - y_2) - y_1
//          dy2 = y_0·y_1 - beta·y_2
//      Note: y_0, y_1, y_2 are the array elements. Change to C++ syntax.
//      2. CODE: Return a vector of doubles with elements <dy0>, <dy1>,
//      and <dy2> (trick: return an initialization list)


// --- MAIN FUNCTION -------------------------------------------------------
//  CODE: Define the <main> function with the following statements:
//      1. CODE: Define an output stream object named <fout> and use it to
//      open the file named tajectory.csv
//      2. CODE: Define an integer named <outStep> and initialize it to 10
//      3. CODE: Define a variable named <time> of type double initialized
//      to 100
//      4. CODE: Define a variable named <dt> of type double initialized
//      to 1.E-3
//      5. CODE: Define an integer named <totalSteps> and initialize it with
//      <time> divided by <dt>

//      6. CODE: Write to the standard output the line
//          steps = <totalSteps>

//      7. CODE: Define a variable named <t> of type double initialized to 0
//      8. CODE: Define a vector of doubles named <y>, initialized with
//      the three values 1.0, 1.0, and 1.0
//      9. CODE: Write to the file indicated by <fout> in a single line,
//      <t>, followed by the three values of <y>, all separated by literal
//      commas
//      10. CODE: Write a loop with integer index <step>,
//      for step = 1,2,...,totalSteps, that do the following statements:
//          A. CODE: Call the <rk4> function with arguments <lorentzD>, <y>,
//          <t>, and <dt>, and assign the return value to <y>
//          B. CODE: Increment <t> by <dt>
//          C. CODE: Whenever <outStep> perfectly divides <step>, write to
//          the file indicated by <fout> in a single line, <t>, followed by
//          the three values of <y>, all separated by literal commas.
//           (Hint: use the if statement and the modulus operator)
// --- END
```

Programming Notes:

- You can modify rk4 to account for explicit time dependence of the ODE
- The Lorenz trajectory is an example of a chaotic strange attractor. The term "butterfly effect" in popular media may stem from the real-world implications of the Lorenz attractor, namely that a small change in the initial conditions (other than 1.0, 1.0, 1.0) causes the trajectory to become unpredictable. Try it.