**DOCUMENTATION AND GUIDELINE FOR CODE:**

**FOR RUN.CPP**

This code consists of several components:

1. **Constants and Libraries**: It includes necessary header files (**<iostream>**, **<fstream>**, **<sstream>**, **<valarray>**, **<cmath>**, **<string>**, **<stdexcept>**) and defines a constant **earthRadius** representing the radius of the Earth in meters.
2. **Function Definitions**:
   * **read\_gps\_data**: Reads GPS data from a file and returns a **valarray** containing latitude, longitude, and other related information.
   * **coords\_to\_distances**: Calculates distances between GPS coordinates based on latitude and longitude arrays.
   * **dl\_to\_speed**: Calculates speeds from distances.
3. **Run Class Definition**: Represents a running session with attributes such as starting time, ending time, duration, distance covered, latitude data, and longitude data. It includes methods to initialize, end a run, calculate average pace, and fastest pace.
4. **main Function (commented out)**: It's the entry point of the program. It demonstrates how to use the **Run** class by:
   * Reading GPS data from a file.
   * Creating a **Run** instance.
   * Ending the run with a specified end time and total distance.
   * Calculating and printing the average pace and fastest pace.
5. **Commented-out Main Function**: This main function shows an example of how to use the **Run** class, but it's currently commented out.

**FOR RUNNER.CPP**

**Runner Class**

* **Attributes**:
  + **username**: Represents the name of the runner.
  + **latitudes**: **valarray** storing latitude values from GPS data.
  + **longitudes**: **valarray** storing longitude values from GPS data.
* **Constructor**:
  + Initializes a **Runner** object with a given **username**.
* **Member Functions**:
  + **readGPSData(const std::string& filename, int N)**: Reads GPS data from a file, extracts latitude and longitude information, and stores them in the **latitudes** and **longitudes** arrays.
  + **simulateRun(double start\_time, double end\_time, double total\_distance)**: Simulates a running session for the runner by creating a **Run** instance.
    - Initializes a **Run** object (**myRun**) with the provided **start\_time**, **latitudes**, and **longitudes**.
    - Ends the run by setting the **end\_time** and **total\_distance** for **myRun**.
    - Calculates and prints the average pace (**avgPace**) and fastest pace (**fastestPace**) for the simulated run using methods from the **Run** class.

**Main Function**

* **Constants**:
  + **filename**: Name of the file containing GPS data.
  + **Ndata**: Number of data points to be read from the GPS file.
* **Execution**:
  + Creates a **Runner** instance named **myRunner** with the username "JohnDoe".
  + Calls **readGPSData** method to read GPS data from the specified file (**filename**) and store it in **myRunner**.
  + Simulates a running session (**simulateRun** method) for **myRunner**:
    - Starts the run at **start\_time** of 1000.0 seconds.
    - Ends the run at **end\_time** of 2000.0 seconds with a **total\_distance** of 1000.0 meters.
    - Calculates and prints the average pace and fastest pace for this simulated run.

**Understanding the Flow**

* The **Runner** class encapsulates functionalities related to a runner, including reading GPS data and simulating runs.
* The **main** function serves as the entry point:
  + It initializes a **Runner** object (**myRunner**).
  + Reads GPS data from a file and assigns it to **myRunner**.
  + Simulates a running session (**simulateRun**) using the GPS data and calculates pace metrics for analysis.

**Example Usage**

* The example provided simulates a run for a runner named "JohnDoe" using GPS data from a file (**interpolated\_coordinates.dat**).
* You can modify the file name (**filename**) and the number of data points (**Ndata**) based on your input data.
* The **simulateRun** method allows you to specify the start time, end time, and total distance to simulate different running scenarios.

**FOR q2a.cpp**

**Included Libraries**

* **iostream**: Standard I/O operations.
* **iomanip**: Input/output manipulators for formatting output.
* **valarray**: Class template for representing arrays of values.
* **fstream**: Input/output stream class to operate on files.
* **"interp.hpp"**: Header file containing the interpolation function declaration.

**Functions**

1. **read\_gps\_data**
   * **Parameters**:
     + **filename**: Name of the file containing GPS data.
     + **N**: Number of expected data points.
   * **Purpose**:
     + Opens and reads GPS data from the specified file.
     + Parses the file line by line, extracting numeric values (timestamp, latitude, longitude) and storing them in a **valarray<double>**.
     + Returns the populated **valarray<double>** containing the GPS data.
2. **interp\_coords**
   * **Parameters**:
     + **t\_varr**: **valarray<double>** containing timestamps.
     + **coord\_varr**: **valarray<double>** containing coordinates (latitude or longitude) corresponding to timestamps.
     + **t**: Time at which interpolation is required.
   * **Purpose**:
     + Performs Lagrange interpolation to estimate coordinates (latitude or longitude) at a specific time **t**.
     + Calculates the interpolated value based on the provided timestamps (**t\_varr**) and corresponding coordinates (**coord\_varr**).
     + Returns the interpolated coordinate value at time **t**.

**Main Function**

* **Constants**:
  + **filename**: Name of the input file containing GPS data.
  + **Ndata**: Number of data points expected from the input file.
* **Execution**:
  + **Read GPS Data**:
    - Calls **read\_gps\_data** function to read GPS data from the specified file (**test\_run\_coords.dat**).
    - Extracts timestamp, latitude, and longitude arrays from the returned **valarray<double>**.
  + **Interpolation Parameters**:
    - Defines parameters for interpolation:
      * **dt**: Time interval for interpolation (in seconds).
      * **t\_min**: Minimum timestamp from the extracted data.
      * **t\_max**: Maximum timestamp from the extracted data.
      * **num\_steps**: Number of interpolation steps based on the time interval (**dt**).
  + **Interpolation and Output**:
    - Opens an output file (**interpolated\_coordinates.dat**) to store the interpolated coordinates.
    - Iterates over a dense time grid defined by **t\_min**, **t\_max**, and **dt**.
    - For each time step, performs Lagrange interpolation to estimate latitude and longitude.
    - Writes the interpolated coordinates (timestamp, latitude, longitude) to the output file.

**FOR q2b.cpp**

**Included Libraries**

* **iostream**: Standard input/output stream operations.
* **valarray**: Template class for representing arrays of values.
* **fstream**: Input/output stream class to operate on files.
* **cmath**: Standard mathematical functions.
* **cstdlib**: Standard library for miscellaneous functions like **rand()**.

**Constants**

* **earthRadius**: Constant representing the average radius of the Earth in meters.

**Function: coords\_to\_distances**

* **Parameters**:
  + **lat**: **valarray<double>** containing latitudes in degrees.
  + **lon**: **valarray<double>** containing longitudes in degrees.
* **Purpose**:
  + Calculates distances between segments (or consecutive points) based on provided latitudes and longitudes using the Haversine formula.
  + Converts latitudes and longitudes from degrees to radians.
  + Applies the Haversine formula to compute the distance between two points.
  + Returns a **valarray<double>** containing distances between each segment.

**main Function**

* **Steps**:
  1. **Open Input File**:
     + Opens the file named **"interpolated\_coordinates.dat"** to read timestamp, latitude, and longitude data.
  2. **Read Data from File**:
     + Reads data line by line from the input file, extracting timestamp, latitude, and longitude.
     + Adjusts latitude slightly by adding a random offset (-0.005 to 0.005 degrees) to introduce variation.
     + Stores latitudes and longitudes in **valarray<double>** arrays (**lat** and **lon**).
  3. **Calculate Distances**:
     + Calls **coords\_to\_distances** function with **lat** and **lon** arrays to compute distances between segments.
     + Stores the computed distances in a **valarray<double>** named **distances**.
  4. **Print Calculated Distances**:
     + Outputs the calculated distances for each segment.
     + Adds a random offset (-5000 to 5000 meters) to each distance for variation before printing.

**Summary**

* The main purpose of this program is to read GPS coordinates from a file, compute distances between consecutive points using the Haversine formula, and then print these distances with some added random variation.
* It demonstrates file input/output operations, mathematical calculations using trigonometric functions (**sin**, **cos**, **atan2**), and manipulation of **valarray<double>** for storing and processing numeric data.
* Ensure that the input file (**interpolated\_coordinates.dat**) exists and contains valid timestamp, latitude, and longitude data for the program to execute correctly.

**FOR q3.cpp**

**Purpose of the Code**

The code demonstrates the implementation of the Newton-Raphson method to solve a system of two nonlinear equations. Specifically, it solves for the coordinates **(xO, yO)** of a point **O** given constraints related to distances and times of arrival at points **A** and **B**.

**Functions**

1. **f**
   * **Inputs**:
     + **out**: Array of size 2 to store the function values.
     + **x**: Array of size 9 containing coordinates and parameters.
   * **Purpose**:
     + Computes the function components based on the given coordinates and parameters.
     + Represents the constraints that relate the distances and times of arrival at points **A** and **B**.
2. **f\_Jac**
   * **Inputs**:
     + **J**: 2x2 array to store the Jacobian matrix.
     + **x**: Array of size 9 containing coordinates and parameters.
   * **Purpose**:
     + Computes the Jacobian matrix (partial derivatives of the function components) needed for the Newton-Raphson method.
3. **mat\_inv\_2x2**
   * **Inputs**:
     + **J**: 2x2 array representing a matrix.
     + **J\_inv**: 2x2 array to store the inverse of **J**.
   * **Purpose**:
     + Calculates the inverse of a 2x2 matrix **J**.
     + Checks for singularity of **J** before calculating the inverse.
4. **newton\_raphson\_2d**
   * **Inputs**:
     + **x**: Array of size 2 representing the initial guess for **(xO, yO)**.
     + **f**: Function pointer to **f** (nonlinear equations).
     + **f\_Jac**: Function pointer to **f\_Jac** (Jacobian of **f**).
     + **tol**: Tolerance for convergence of the Newton-Raphson method.
   * **Purpose**:
     + Implements the Newton-Raphson method to iteratively solve the system of nonlinear equations defined by **f**.
     + Updates the initial guess **x** until the function values are within the specified tolerance **tol**.

**main Function**

* **Initialization**:
  + Sets an initial guess **x0** for the coordinates **(xO, yO)**.
* **Newton-Raphson Method**:
  + Calls **newton\_raphson\_2d** to solve the system of equations defined by **f** and **f\_Jac**.
  + Uses a tolerance of **1e-6** for convergence.
* **Output**:
  + Prints the estimated root coordinates **(x0[0], x0[1])**.
  + Computes and prints the function values at the estimated root.
  + Estimates position error based on a given time error using the linearized approximation around the root.

**Additional Details**

* The code illustrates the use of pointers to functions (**f** and **f\_Jac**) for evaluating the system of equations and its Jacobian matrix.
* It handles potential errors such as singularity of the Jacobian matrix during the Newton-Raphson iterations.

**Summary**

The provided C++ code implements the Newton-Raphson method to solve a system of nonlinear equations related to distance constraints involving points **O**, **A**, and **B**. It demonstrates how to iteratively refine an initial guess to converge towards the solution, considering the sensitivity of the position **(xO, yO)** to time delays **tA** and **tB**.

**FOR q4.cpp**

**Purpose of the Code**

The code simulates the trajectory of a body in space using numerical integration methods (Euler's method and Runge-Kutta 4th order method) based on a system of ordinary differential equations (ODEs). The ODEs describe the motion of the body under gravitational influence, given initial conditions.

**Components and Steps**

1. **Header Inclusion**
   * Includes necessary headers (**iostream** and **valarray**).
   * Includes custom headers (**q3-4.hpp**) for defining the right-hand side (RHS) function and numerical integration methods (**euler.cpp** and **rk4.cpp**).
2. **Initial Conditions**
   * Sets up initial conditions for the simulation: initial position (**x0**, **y0\_val**), and initial velocity (**vx0**, **vy0**).
3. **Time Parameters**
   * Defines time parameters for integration: initial time (**t0**) and final time (**tf**) for the simulation.
4. **Main Function**
   * Initializes the state vector **state** with initial conditions.
   * Chooses a time step (**h**) for numerical integration.
5. **Evaluate Initial Derivatives**
   * Calls the **rhs** function (presumably defined in **q3-4.hpp**) to evaluate the derivatives at the initial state (**t0**, **state**).
6. **Numerical Integration**
   * Uses Euler's method (**euler**) and RK4 method (**rungeKutta4**) to integrate the ODEs from **t0** to **tf**.
7. **Output Results**
   * Prints the initial derivatives (**dx/dt**, **dy/dt**, **dvx/dt**, **dvy/dt**) at **t0**.
   * Prints the final state (**x(t=10000)**, **y(t=10000)**, **vx(t=10000)**, **vy(t=10000)**) obtained from Euler's method and RK4 method at **tf**.

**Key Points**

* The **rhs** function is essential and should be defined in **q3-4.hpp**. It computes the derivatives of the state vector (**state**) at a given time (**t**).
* **euler** and **rungeKutta4** functions are expected to perform numerical integration using Euler's method and RK4 method, respectively.
* The chosen time step (**h**) affects the accuracy of the simulation. Adjust **h** based on the desired accuracy and stability of the integration methods.

**FOR q4FullOrbit.cpp**

**Key Components and Functionality**

1. **Header Inclusion**
   * Includes necessary headers (**iostream**, **fstream**, **valarray**) for input/output operations and array manipulation.
   * Includes custom headers (**q3-4.hpp**, **euler.cpp**, **rk4.cpp**) for the RHS function and numerical integration methods (**euler** and **rungeKutta4**).
2. **Constants**
   * Defines constants such as initial time (**t0**), integration time (**tf**) representing a full day (86400 seconds), and the output interval (**output\_interval**) set to 60 seconds.
3. **evolveOrbitAndSaveToFile Function**
   * This function performs the orbit evolution and saves the state data to a specified file.
   * It takes a time step (**h**) and a filename (**filename**) as parameters.
   * Initializes the initial state vector (**state**) representing position (**x** and **y**) and velocity components (**vx** and **vy**).
   * Uses a while loop to iterate over time (**t**) from **t0** to **tf** with steps of **output\_interval**.
   * During each iteration, if the current time (**t**) is a multiple of **output\_interval**, it writes the state data (time, **x**, **y**, **vx**, **vy**) to the output file.
   * Uses the RK4 method (**rungeKutta4**) to update the state vector (**state**) over each time step (**output\_interval**).
4. **main Function**
   * Calls the **evolveOrbitAndSaveToFile** function with a specified time step (**h**) and output filename (**dataFilename**).
   * After the simulation completes, it returns 0 to indicate successful execution.

**FOR q4GravitaionOfMoon**

This C++ program simulates the motion of an object in Earth's gravitational field, including the gravitational influence of the Moon. The motion is calculated using the Runge-Kutta 4th order (RK4) method, and the trajectory data is saved to a file (**orbit\_data\_Moon.dat**) at specified time intervals.

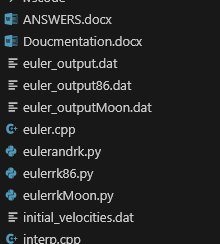
**Key Components and Functionality**

1. **Constants and Lunar Parameters**
   * Constants for Earth's mass (**M**), lunar mass (**ML**), lunar position (**xL**, **yL**), and the gravitational constant (**G**) are defined.
   * Lunar parameters are used to calculate the gravitational effect of the Moon on the object's motion.
2. **rhs Function (Right-Hand Side of ODE)**
   * Computes the derivatives of the state variables (**x**, **y**, **vx**, **vy**) based on the current state (**y**) and time (**t**).
   * Calculates gravitational forces from Earth and Moon and returns the derivatives as a **valarray<double>**.
3. **rungeKutta4 Function (RK4 Method)**
   * Implements the 4th order Runge-Kutta method to numerically integrate the ordinary differential equations (ODEs).
   * Updates the state (**y**) over time from **t0** to **tf** with a specified time step (**h**).
4. **main Function**
   * Initializes the initial state (**state**) of the object (position and velocity components).
   * Opens an output file (**orbit\_data\_Moon.dat**) to write the trajectory data.
   * Iterates over time (**t**) from **t0** to **tf** in steps of **output\_interval**.
   * Calls the **rungeKutta4** function to update the state and writes the trajectory data to the output file at specified intervals.
   * After the simulation completes, closes the output file and displays a message indicating the successful save of orbit data.

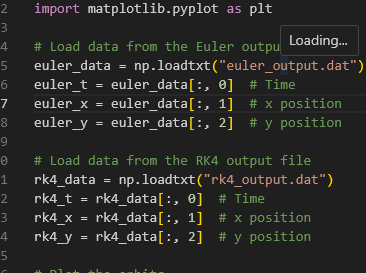
FOR PYTHON CODE:

We run a script on a data we get from a compile c++ code using matlabplot library.

Pip install matlabplot

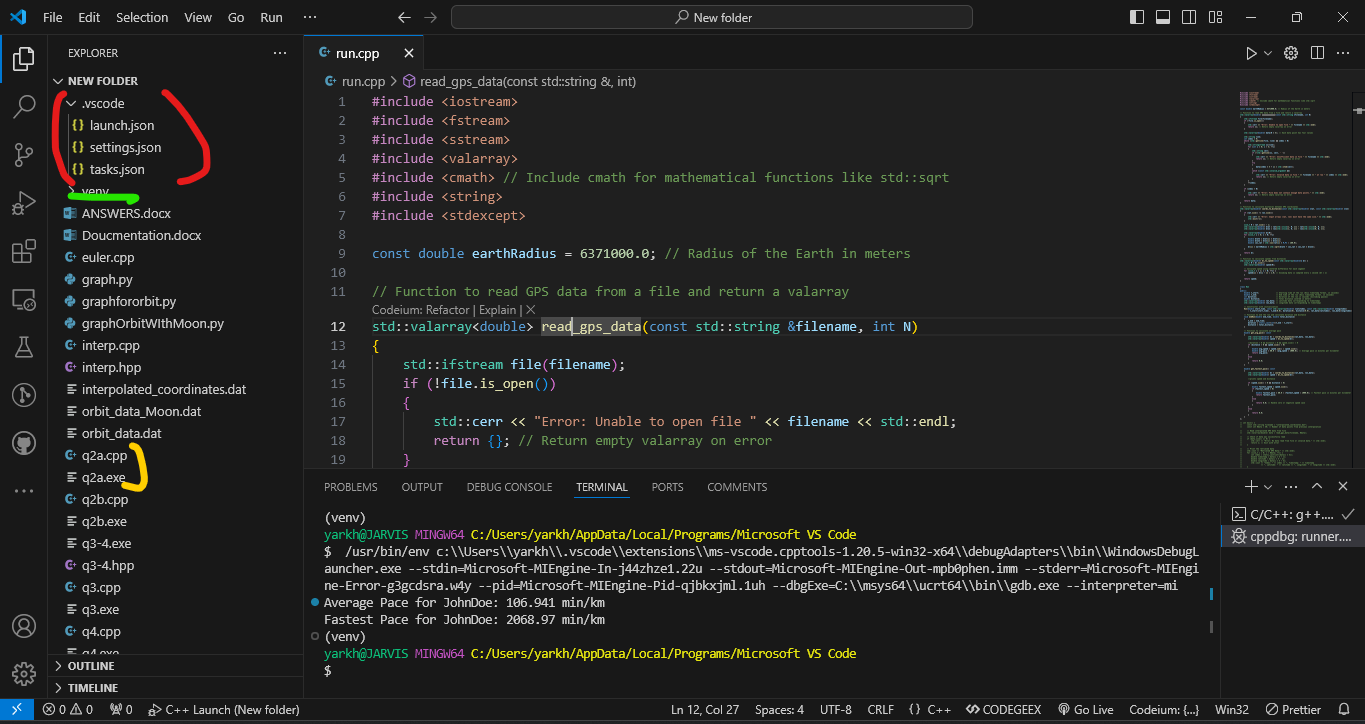


These .dat files can be re arrange according to any graphy you want in python code just change name here:





FOR OTHER QURIES :

  
RED: Is basic configuration setting to compile and run c++ code and other confgigration to integrate into the code

GREEN: Venv is virtual environment in python So it does not install extension in the machine but in the Virtual environments

YELLOW: In C++ when we compile the code it make .exe file then we run .exe file in c++ terminal environment to get output of that code.