INTEL DEVMESH PROJECT [1]

Direction Field Visualization with Python

Author: Oluwatosin Odubanjo

August 1, 2022.

[1] Personal Development Project

Appreciation

I thank Yahweh, my family and the Intel® community.

Summary

The overall objective of this project is to demonstrate the visualization of a direction field with Python using the differential equation of a falling object as a case study.

Specifically, the project shows the following -

1. The use of open source software (Python) in developing the programs for the direction field plot.
2. The use of the matplotlib.pyplot.quiver() and straight line equation methods to develop the programs used in obtaining the direction field.
3. The effectiveness of the numerical package (numpy) offered by python, in the programs developed.
4. The effectiveness of heterogeneous computing by exploring the added functionalities offered by Intel® Distribution for Python (IDP*).

Performance results obtained show that the best performing program is based on the matplotlib.pyplot.quiver() method, and has the additional functionality of data parallel control (dpctl) package offered by IDP*, enabling it to run on the Integrated Graphics, Intel® UHD Graphics P630 [0x3e96] of the Intel® E-2176G processor.

Keywords:

- Python
- oneAPI
- Direction Field
- Differential Equation
- Data Parallel Python
- Heterogeneous Computing
- High Performance Computing
- Intel® Distribution for Python*

Author's Note

Intel® oneAPI for Heterogeneous Computing

Heterogeneous simply means more than one kind, on the other hand, computing is the science and art of using computers to solve problems.

- **The scientific nature of computing**

This involves (but is not limited to):

1. The use of existing theories and principles to solve field specific problems.
2. Evaluation and Comparison of computer programs to determine the most suitable.
3. Fault finding, or most appropriately, bug finding.
4. Correct use of already made tools to develop a computer program.
5. Correct use of already made tools to increase the efficiency of a computer program.

- **The artistic nature of Computing**

This involves developing creative, intentional and innovative ways to solve problems and will greatly depend on :

1. Algorithmic and pattern recognition and usage in computer programs.
2. Adaptation of a technology solution to solve a particular problem.

What then is heterogeneous computing? The term heterogeneous computing is used to describe the use of a computing solution (in this case, a technology, framework, language, algorithm) in the design and development of computer programs / computer applications for computing systems possessing more than one kind of computational architecture (processors - CPUs, GPUs, FPGAs, other accelerators).

The Effectiveness of a Computing Solution

Computing solution here means designed technologies (such as the compiler, oneAPI, OpenMP, CUDA, OpenCL, applications for program performance checks e.t.c), designed frameworks (such as DPC++, OpenMP, CUDA, OpenCL, et.c), developed resources (such as processors - CPUs, GPUs, FPGAs), developed languages and standards for the structure of computer programs (such as C, C++, python, e,t,c), developed algorithms for specific problems (such as an algorithm for matrix multiplication, e.t.c).

Effectiveness is a measure of a competitive advantage. It places an item in an accepted, a favourable and superior position because it was tested and tried with some accepted standards and it triumphed against other alternatives. In computing, the major metrics for effectiveness are cost (time and memory) and resources (processors). To determine a most suitable computing solution, computer program developers, computer engineers and scientists go through routes of benchmarking, mean execution time tests, tests for speedup e.t.c.

Computing systems (- laptops, HPC systems, embedded systems), today, no longer come with one kind of processor, thus the need for heterogeneous computing. This shift in computer system design creates the need to make certain decisions, if high performing computer programs are to be developed. Decisions made could arise from questions such as :

1. What is the best part of my computer program for a specific device ?
2. What is the best algorithm or pattern for my computer program without sacrificing accuracy?
3. How can I write my computer program in such a way that it is maintainable and reusable ?
4. What are the available performance-driven technologies to aid the development of my computer program ?
5. What are the available performance-driven technologies to allow portability of my computer program across a number of devices ?

The bottom line here is, as a developer you want to be aware of available technologies, resources, techniques, patterns for developing an efficient computer program for a problem. The good news is that, developers do not have to look too far. Intel® oneAPI [2] is a group of

[2] one Application Programming Interface

computing solutions that is focused on the development of high performance solutions (computer programs, applications, e.t.c) for field specific problems (finance, physical, biological and chemical sciences, Artificial Intelligence, Data Science, e.t.c).

Intel oneAPI is based on open standards and specifications, it is cross-industry and fosters developments of high performing applications across diverse architectures (processors - CPUs, GPUs, FPGAs, other accelerators). If you would like to learn more about the characteristics of the computing solutions offered by oneAPI, head out to section 1.2 (Key things to note about Intel® oneAPI) of this article - Intel® oneAPI: The Key Essentials.

Earlier, I mentioned that effectiveness of an item is obtained when it has been tested and tried with some acceptable standards and thus proves itself worthy. Now, some example projects showing the capabilities of oneAPI include:

1. High Performance Computing: Evaluation of Intel's DPC++ Compatibility Tool in heterogeneous computing (Rodinia Application Benchmark suite)
2. Artificial Intelligence: Added dpcpp support for Huawei AI chipset
3. Mathematical;, numerical, scientific computing High Performance Implementation of Boris Particle Pusher on DPC++

The above projects are based on one out of several high performing compute solutions offered by oneAPI- Data Parallel C++, DPC++.

This project - **Direction Field Visualization with Python**, is an example of what oneAPI can do; it explores another high performance compute solution offered by oneAPI - the Intel® Distribution for Python*, IDP*.

The beauty of science is in its application to solve problems. One interesting thing I love about the scientific nature of computing is that you do not just theorize it, you get to do it and see the results for yourself. With these words, let's go on to see what IDP can do.*

Oluwatosin Odubanjo
Thank you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A DIRECTION FIELD also referred to as a slope field is a graphical representation of solutions of the differential equations of the form, equation 1.1. It is a vital tool for visualizing and investigating the behaviour of solutions of such differential equations without having to first solve them. Therefore, no matter how simple or complex the differential equation looks, there is always a direction field for it.

$$\frac{dy}{dx} = f(x, y) \tag{1.1}$$

where, f is a given function of x and y.

1.1 Solutions of a Differential Equation

1. **Equilibrium solution** : This solution is also known as the equilibrium point, critical value or the balance solution. It is the solution which defines the behaviour of other solutions as the independent variable in a differential equation tends towards infinity. Other solutions either converge to the equilibrium solution or diverge from the equilibrium solution. An equilibrium solution can be described as unstable, asymptotically stable, and semi-stable depending on the behaviour of other solutions.
2. **General solution** : solution with an arbitrary constant.
3. **Particular solution** : solution with no arbitrary constant. It is associated with an initial condition.

1.2 How to construct a Direction Field

To construct a direction field, we need to :

1. **Draw a 2-dimensional grid** - A rectangular grid of a few one hundred points. .

2. **Draw a short line segment at each point on the grid** - The slope of this line is the value of the given differential equation at that point.

1.3 Tools for Constructing a Direction Field

Of course we can always use a graph sheet and a pencil to draw a direction field, but it is more efficient to automate the process with :

1. A Computer.

2. An application software that has functionalities :
: - to repeatedly evaluate a given function over some numerical interval and
: - to plot the resulting values.

Chapter 2

About this project

The scope of this project is mathematics, numerical / scientific computing and high performance computing and the case study differential equation is the *differential equation of a falling object.* This project is an extension of my **How to Plot a Direction Field with Python** article posted on medium. You can find it here.

2.1 The difference

The medium article is a quick guide on how to plot a direction field with python and the python programs were tested on my local computer. However, this project contains an updated python program for the case study of a falling object; the updated python program includes the concept of *the safe zero-bound limit* to properly determine the equilibrium solution computationally. The project also includes another method for plotting a direction field asides the matplotlib.pyplot.quiver() method.

Also, in the article, I do not talk about code performance. But, this project is all about performance. Therefore, in developing the python program, I make use of the Intel® Distribution for Python® (containing Intel's accelerated python packages) on the Intel® DevCloud - a remote development sandbox that allows me develop the python programs and test their performances on latest Intel XPUs (processors). For this project, the development and testing of the python programs is on and with the Intel® E-2176G embedded with Intel® UHD Graphics P630 [0x3e96].

Furthermore, in this project, I do not show other examples shown in the medium article. Here, I focus only the differential equation of a falling object.

2.2 Objectives

The overall objective of this project is to demonstrate the visualization of a direction field with Python. Specifically, the project aims to address the following objectives:

O1. To use the differential equation of a falling object as a case study.

O2. To use an open source application software in developing programs for the direction field plot.

O3. To explore the numerical package of the python software (numpy) in developing programs for the direction field plot.

O4. To extend the functionality of the programs developed in O3. with packages provided in the Intel's Distribution for Python* (IDP*) in order to demonstrate heterogeneous computing.

O5. To explore the possibilities of increasing performance of the python programs developed in 03. with the data parallel package offered by Intel.

2.3 Approach

In this project, the implementation of a direction field plot with python is based on two methods:

- The matplotlib.pyplot.quiver() (MPQ) method
- The straight line equation (SLE) method.

I consider these two methods for the purpose of comparison of performance based on the time of execution as well as for the purpose of exploring the following packages in the Intel® Distribution of Python* (IDP*) :

1. Numpy - an optimized Python numerical package.
2. The *numba.prange* expression provided by Numba* - an open-source, NumPy-aware optimizing compiler for Python developed by Anaconda, Inc in collaboration with an open-source community.
3. SYCL-based XPU programming provided by Data Parallel Python (DPPY). The following packages in DPPY are explored :
    - Data Parallel Control (dpctl) - A package for controlling execution on SYCL devices and for SYCL USM data management.
    - Numba_dppy - A standalone extension to Numba adding SYCL kernel programming to Numba*.

2.4 Python Program Design

The git repository for this project can be found here. Also, a chart showing the program design flow can be seen in section B.

2.5 Performance metrics

Based on the execution time of a computer program, the following conclusions can be made depending on if performance increase or decrease is observed:

Performance Increase:

1. Time Reduction :
$$\frac{old\ value - new\ value}{old\ value} \times 100\% \tag{2.1}$$
2. Time Deduction :
$$\frac{Time\ Reduction\ value\%}{100\%} \times old\ value \tag{2.2}$$
3. Time Gain :
$$\frac{100\%}{(100\% - Time\ Reduction\ value\%)} \tag{2.3}$$
or simply
$$\frac{old\ value}{new\ value} \tag{2.4}$$

Performance Decrease

1. Time Increase :
$$\frac{new\ value - old\ value}{new\ value} \times 100\% \tag{2.5}$$
2. Time Loss :
$$\frac{100\%}{(100\% - Time\ Increase\ value\%)} \tag{2.6}$$
or simply
$$\frac{new\ value}{old\ value} \tag{2.7}$$

In this project, performance conclusions are made with respect to mathematical formulas given in the instance of a performance increase, equations 2.1, 2.2 and 2.3. For a visual representation of results a Percentage Difference Graph (PDG), is used. Values used in the PDG are those obtained from the time reduction metric. The program rated at 100% is the

least performing program and it represents the reference program to view the degree (extent) of percentage difference seen in the other better performing programs. The percentage difference of other programs is obtained by evaluating -

$$100\% - Time\ Reduction\ value \tag{2.8}$$

. Therefore, in other to obtain a PDG, the factors needed are:

- The reference program which is the least performing program rated at 100%.
- Time reduction of other programs with respect to the least performing program.

2.6 Technologies Used

Development Environment: Intel devcloud - for developing, testing and running the project.
Intel Hardware: Intel® Xeon® E-2176G.
oneAPI Toolkit(s): oneAPI Base Toolkit.
Programming Support: Intel® Distribution for Python*.

Table 2.1: Characteristics of Development Environment and Processor

Development Environment
Environment: Intel DevCloud **File Storage**: 220GB **RAM**: 192GB **Terminal Interface**: Linux **Accessed from**: HP Notebook-15-r029wm, Intel® Pentium® CPU N3540 @ 2.16GHz, 4 Core(s), 4 Logical.
Characteristics of Processor
Processor [1] : Intel® Xeon® E-2176G **Frequency**: 3.70 GHz (4.70 GHz turbo) **Cache**: 12 MB Intel® Smart Cache **Cores**: 6 (12 Logical) **Threads**: 12 **Max Memory Size**: 128GB **Memory Type**: DDR4-2666 **Max Memory Bandwidth**: 41.6 GB/s **Processor Graphics**: Intel® UHD Graphics P630 [0x3e96] **Frequency**: 0.35 – 1.20 GHz **Execution Units**: 24

[1] https://www.intel.com/content/www/us/en/products/sku/134860/intel-xeon-e2176g-processor-12m-cache-up-to-4-70-ghz/specifications.html

Chapter 3

Intel® Distribution for Python*

Intel® Distribution for Python*, IDP*, is a complete Python distribution that includes the necessary Python packages to develop high-performing code targeting Intel® XPUs (CPUs, GPUs, e.t.c) using Python. The distribution includes the following:

1. Optimized Python numerical and scientific packages NumPy, SciPy, Scikit-learn, MKL-FFT that use Intel® oneAPI Math Kernel Library (oneMKL) and Intel® oneAPI Data Analytics Library (oneDAL) to offer near-native performance.
2. Customized version of the Numba* JIT compiler - for generating fast code for Intel® XPUs.
3. Data Parallel Python (DPPY) - a set of packages enabling SYCL-based XPU programming.
    - Data Parallel Control (dpctl): A package for controlling execution on SYCL devices and for SYCL USM data management.
    - Data Parallel Numeric Python (dpnp): An implementation of the NumPy API using SYCL and oneMKL.
    - Numba-dppy: A standalone extension to Numba adding SYCL kernel programming to Numba*.
4. XGBoost*, scikit-learn, and advanced ML usages, including multiple devices, with daal4py - for Faster machine learning
5. Scikit-ipp for image warping, image filtering, and morphological operations. Support for transform function multithreading and partial multithreading for filters using OpenMP is also included.

Chapter 4

Differential Equation of a Falling Object

The differential equation that describes the motion of a falling object in the atmosphere near sea level is a first order ordinary linear differential equation and it is given as:

$$m\frac{dv}{dt} = mg - C_d v \tag{4.1}$$

where:

- m = mass of the object in kilogram, kg.
- v = velocity of the object in $meter\ per\ second$, ms^{-1}
- t = time in $seconds$, s
- dv/dt = acceleration of the object in $meter\ per\ second\ square$, ms^{-2}
- g = acceleration due to gravity, 9.8 $meter\ per\ second\ square$, ms^{-2}
- C_d = Drag Coefficient (constant value which varies from one object to another).

  Drag Coefficient is a unitless value given as:

$$\frac{F_d}{0.5\rho v^2 A} \tag{4.2}$$

  where:

  - F_d = Drag force in Newtons, $kilogram\ meter\ per\ secondsquare$, $kgms^{-2}$
  - ρ = Density in $kilogram\ per\ meter\ cube$, kgm^{-3}
  - v^2 = velocity in $meter\ square\ per\ second\ square$, m^2s^{-2}

– A = Area in $meter\ square$, m^2

To derive the unit of the drag coefficient, equation in 4.2 can be rewritten as :

$$\frac{kgms^{-2}}{kgm^{-3} \times ms^{-2} \times m^2} \tag{4.3}$$

simplifying equation 4.3:

$$kgms^{-2} \times kg^{-1}m^3 \times m^{-2}s^2 \times m^{-2} \tag{4.4}$$

simplifying equation 4.4 using the multiplication law of indices:

$$kg^{1-1} \times m^{1+3-2-2} \times s^{-2+2} \tag{4.5}$$

$$kg^0 \times m^0 \times s^0 = 1 \tag{4.6}$$

The derivation shows that drag coefficient has no unit.

The differential equation given in equation 4.1, is of the form :

$$\frac{dy}{dx} = ay - b \tag{4.7}$$

In order to solve it, we need to find a function v = v(t) that satisfies the equation. However, we shall not concern ourselves with the solution for now. We shall only consider generating the direction field (if you are however interested in seeing the solution, head out to section C.

In this project, for obtaining the direction field for the equation 4.1, we shall assume the following :

- mass, m, of the object is 10kg.
- Drag Coefficient, C_d is 2.

If we input these values into the equation 4.1, it will become:

$$\frac{dv}{dt} = 9.8 - \frac{1}{5}v \tag{4.8}$$

Comparing equation 4.7 to equation 4.8, the following relations can be obtained:

- $\frac{dy}{dx} = \frac{dv}{dt}$
- $a = -\frac{1}{5}$

- $y = v$
- $-b = 9.8$

It is also important to note the following about equation 4.8

- v = the dependent variable
- t = the independent variable

Similarly, in equation 4.7:

- y = the dependent variable
- x = the independent variable

4.1 Equilibrium Solution Identification

In section 1.1, we already learnt the definition of the equilibrium solution of a differential equation. But, how exactly do we solve for it ? It is obtained when we make the left hand side of the differential equation equal to zero and then solve for the dependent variable. At the equilibrium solution, the differential equation is constant with time.

If we make the left hand side of the equation 4.8 = 0, we will have:

$$0 = 9.8 - \frac{1}{5}v \tag{4.9}$$

Now, solving for v: multiply all terms in equation 4.9 by 5:

$$0 \times 5 = 9.8 \times 5 - \frac{1}{5}v \times 5 \tag{4.10}$$

$$0 = 49 - v \tag{4.11}$$

$$v = 49 \tag{4.12}$$

Therefore, v(t) = 49, is the equilibrium solution. When, the direction field of equation 4.8 is plotted with the inclusion of the equilibrium solution, we will notice an horizontal line, this line indicates the equilibrium solution. And it is horizontal to show that the differential equation at that value does not change with time, that is, it is constant with time.

Chapter 5

matplotlib.pyplot.quiver() (MPQ) method

Earlier in section 1.2, I discussed how to construct a direction field. In this chapter, we will see how the steps for plotting a direction field relates to the functions provided by the python programs that use the MPQ method and we will also see the functionalities of the individual python programs that make use of the MPQ method.

You can find the python programs here Basically, all python programs based on the MPQ method make use of the *numpy.arange()* [1], *numpy.meshgrid()* [2] and *matplotlib.pyplot.axhline* [3] *and .quiver()* [4] functions of the numpy library.

Following the steps from section 1.2 -

1. **A rectangular grid of a few one hundred points** - this is obtained with the *numpy.arange()*, *numpy.meshgrid()* and *matplotlib.pyplot.plot()* functions. The *numpy.meshgrid()* function in python is used to create a rectangular grid out of two defined one-dimensional (1-D) arrays. The 1-D arrays (which are a numerical interval; they can be defined using the *numpy.arange() function*) represent the points on the grid and the matrix (ij) or cartesian (xy) indexing of the *numpy.meshgrid function()*. You can find the python codes that illustrate how to generate a rectangular grid using the *numpy.meshgrid()* function for both matrix and cartesian indexing use cases in the github repository for this project (click here).

2. **Short line segment at each point on the grid** - In step 1 above, we have been able to generate a rectangular grid. Now, this next step involves drawing a line segment, whose slope is the value of the given differential equation at each point on the grid. Generally, a quiver plot is a type of 2-D plot that is made up of vector lines, these vector lines are in shape

[1] https://numpy.org/doc/stable/reference/generated/numpy.arange.html
[2] https://numpy.org/doc/stable/reference/generated/numpy.meshgrid.html
[3] https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.axhline.html
[4] https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.quiver.html

of arrows to indicate the direction of the vectors (the lines can however be drawn without arrows). These plots are mainly used to visualize slopes / gradients. To obtain a quiver plot in python, we use the *matplotlib.pyplot.quiver()* function which is used to visualize a vector field over a grid of points which are numerical intervals.

5.1 Python Programs

In section 5, we have been able to establish a theoretical-computational relationship between steps to construct a direction field found in section 1.2 and the python software. In this section, we will take a look at the individual python programs developed using the *matplotlib.pyplot.quiver()* method and explain their functionalities.

5.1.1 fall_numpy.py

In this program, the task of plotting of a direction field is divided amongst three functions -

1. equilibrium_solution(...) - This function obtains the plot for the equilibrium solution of the differential equation using the matplotlib.pyplot.axhline() function. In order to obtain the equilibrium solution of the differential equation computationally, I derive a term : *the safe zero-bound limit.* The term Safe Zero-bound Limit is an adaptation of the Economics term zero-bound limit.

   The Safe Zero-bound Limit Meaning : **If ‘a‘ conatins k number of elements and there is an element of the power E-n; where n is a integer. This integer shall determine the ‘the zero-bound element‘; The zero-bound element shall impose on other elements in ‘a‘, a standard for filtering ‘a‘. This standard shall be called the ‘safe zero-bound limit‘. The safe zero-bound limit shall be a user-defined integer, greater than or equal to 5, to maintain a degree of accuracy (such as : 0E-5 == ‘5‘ decimal places, 0E-6 == ‘6‘ decimal places, e.t.c). Then all elements in ‘a‘ shall be rounded this limit and no further; and ‘a‘ shall be filtered to obtain the zero-bound element. In the case of obtaining the equilibrium solution in this project: The safe zero-bound limit determines the cut-down / round condition for the elements of the array and therefore allows the filtering of an array in order to obtain the equilibrium solution.**

   In this project, the safe zero-bound limit determines the cut-down / round condition for the elements of the array and therefore allows the filtering of an array in order to obtain the equilibrium solution (see this python script illustrating this concept here).

2. other_solutions(...) - This function obtains the plot for other solutions of the differential equation using this matplotlib.pyplot.quiver(...) function.

3. main() - This is the caller function for the equilibrium_solution(...) and other_solutions(...) functions.

5.1.2 fall_dpctl_cpu.py

This program is same as the above program in section, 5.1.1 except that the data parallel control, dpctl, package is added to enable the execution of the main function on the CPU device of the processor.

5.1.3 fall_dpctl_gpu.py

This program is same as that in section 5.1.1 except that the data parallel control, dpctl, package, is added to enable the execution of the main function on the GPU device of the processor.

5.2 Performance Analysis

I have created a different chapter to talk about the performance of the individual python programs making use of the MPQ method. Please see chapter 7.

Chapter 6

Straight Line Equation (SLE) method

In this chapter, we will see how the steps for plotting a direction field in section 1.2 relate to the functions provided by the python programs that use the SLE (the equation of straight line with a given slope **m** and passing through a given point **P(**x_1**,** y_1**)**) method and we will also see the functionalities of the individual python programs that make use of the SLE method.

The individual python programs making use of the SLE method can be found here.

Following the steps from section 1.2 -

1. **A rectangular grid of a few one hundred points** - this is obtained with the use of two for loops and the *matplotlib.pyplot.plot()*[1] function. The for loops set the iterator variables (such as i, j) to each value defined inside two one-dimensional (1-D) arrays and repeats the code in the body of the innermost for loop for each value of the iterator variable. You can find the python script illustrating this here.

2. **Short line segment at each point on the grid** - This step is obtained by using the definition of the equation of straight line with a given slope **m** and passing through a given point **P(**x_1**,** y_1**)** given as:

$$y - y_1 = m(x - x_1) \tag{6.1}$$

To use this definition, we would have to:

1. Define a numerical interval (1-D array) for x, the independent variable axis, and y, the dependent variable axis of the rectangular grid in step 1. above.
2. Define a for loop for numerical interval on the x-axis.
3. Define another for loop for numerical interval on the y-axis.
4. Obtain the slope of the line by evaluating the given differential equation for the numerical interval on the y-axis.

[1] https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.plot.html

5. Define points x for each points y, which will be obtained by setting y as the subject of formula in the formula 6.1 above.

6. plot points x and y obtained from step 5.

6.1 Python Programs

In section 6, we have been able to establish a theoretical-computational relationship between steps to construct a direction field found in section 1.2 and the python software. In subsequent sections, we will take a look at the individual python programs developed using the Straight Line Equation (SLE) method and explain their functionalities.

In the folder for the SLE method (here), the python programs are divided into two folders - numpy_linspace and my_linspace. We will first look at the programs in the numpy_linspace folder and then the programs in the my_linspace folder.

6.1.1 Folder numpy_linspace

This folder contains python programs that make use of the numpy.linspace() function to generate points x which will be used to get corresponding points y using the mathematical formula, 6.1. The following programs are contained in this folder:

6.1.1.1 fall_numpy.py

In this program, the task of plotting of a direction field is divided amongst 4 functions -

1. differential_equation(...): This function returns the result of the evaluated differential equation over a numerical interval.

2. line_equation(...): This function returns points y for corresponding points x and slope m, which is the result of the differential_equation(...) function above.

3. equilibrium_solution(): This function uses the safe zero-bound limit explained in section 5.1.1 to obtain the equilibrium solution as well as uses the matplotlib.pyplot.axhline() function to plot the equilibrium solution for the given differential equation.

4. solutions(..): This function obtains the plot for other solutions for the given differential equation over a numerical interval using two for loops, points x generated with the numpy.linspace()[2] function, the line_equation(...) and the equilibrium_solution(...) functions.

5. main(): This is the caller function for the solutions(...) function.

[2] https://numpy.org/doc/stable/reference/generated/numpy.linspace.html

6.1.1.2 fall_dpctl_cpu.py

This program is same as that in section 6.1.1.1 except that the data parallel control, dpctl, package, is added to enable the execution of the main function on the CPU device of the processor.

6.1.1.3 fall_dpctl_gpu.py

This program is same as that in section 6.1.1.1 except that the data parallel control, dpctl, package, is added to enable the execution of the main function on the GPU device of the processor.

6.1.2 Folder my_linspace

This folder contains python programs that make use of a user defined function as an alternative to the numpy.linspace() function to generate points x which will be used to get corresponding points y using the mathematical formula, 6.1. The user defined function is based on the **arithmetic sequence and series** mathematics formula for finding the rth term of a sequence :

$$U_r = a + (r-1)d \tag{6.2}$$

where:
Ur = sequence of numbers
a = first term in a sequence
r = rth term in a sequence
d = common difference in a sequence

So that the arithmetic sequence is then given by:

$$a, a+d, a+2d, a+3d, ..., a+(r-1)d, ... \tag{6.3}$$

The following programs are contained in the my_linspace folder:

6.1.2.1 fall_myfunc.py

In this program, the task of plotting of a direction field is divided amongst 5 functions -

1. differential_equation(...): This function returns the result of the evaluated differential equation over a numerical interval.
2. line_equation(...): This function returns points y for corresponding points x and slope m, which is the result of the differential_equation(...) function above.

3. my_linspace(...): This function returns an arithmetic sequence between two numbers. It imitates the numpy.linspace() function.

4. equilibrium_solution(): This function uses the safe zero-bound limit explained in section 5.1.1 to obtain the equilibrium solution as well as uses the matplotlib.pyplot.axhline() function to plot the equilibrium solution for the given differential equation.

5. solutions(..): This function obtains the plot for other solutions for the given differential equation over a numerical interval using two for loops, points x generated with the my_linspace() function, the line_equation(...) and the equilibrium_solution(...) functions.

6. main(): This is the caller function for the solutions(...) function.

6.1.2.2 fall_myfunc_numba.py

This program is same as the program above (section 6.1.2.1), except that every numpy.arange() implementation is replaced with the prange() function from the numba package.

6.1.2.3 fall_myfunc_dpctl_cpu.py

This program is same as the program above (section 6.1.2.2) with the added functionality of the data parallel control, dpctl, package, to enable the execution of the main function on the CPU device of the processor.

6.1.2.4 fall_myfunc_dpctl_gpu.py

This program is same as the program above (section 6.1.2.2) with the added functionality of the data parallel control, dpctl, package, to enable the execution of the main function on the GPU device of the processor.

6.2 Performance Analysis

A different chapter to talk about the performance of the individual python programs making use of the SLE method has been created. Please see chapter 8.

Chapter 7

Performance Analysis - MPQ method

In this chapter, the performance of the MPQ python programs explained in chapter 5, section 5.1 is discussed. The performance of the individual programs will be evaluated with the mathematical formulas for performance introduced in chapter 2.5 and will be based on on their execution times.

7.1 Performance Results

The table D.1 in chapter D, shows the timing performance of the individual python programs implemented with the MPQ method. The timing results tells us that:

1. The best performing program is the fall_dpctl_gpu.py.

   comments:

   i. Time reduction in fall_dpctl_gpu.py with respect to fall_numpy.py = 19.34547%.

   Therefore, the deducted time from fall_dpctl_gpu.py is 19.34547% of fall_numpy.py (= 0.00733).

   ii. fall_dpctl_gpu.py is about 1.23986X faster than fall_numpy.py.

   iii. Time reduction in fall_dpctl_gpu.py with respect to fall_dpctl_cpu.py = 0.29364%.

   Therefore, the deducted time from fall_dpctl_gpu.py is 0.29364% of fall_dpctl_cpu.py (= 0.00009).

   iv. fall_dpctl_gpu.py is about 1.00295X faster than fall_dpctl_cpu.py.

2. The second best performing program is the fall_dpctl_cpu.py.

   comments:

i. Time reduction in fall_dpctl_cpu.py with respect to fall_numpy.py = 19.10794%.

Therefore, the deducted time from fall_dpctl_cpu.py is 19.10794% of fall_numpy.py (= 0.00724).

ii. fall_dpctl_cpu.py is about 1.23622X faster than fall_numpy.py.

3. The least performing program is the fall_numpy.py.

Furthermore, the timing behaviour of the programs (based on the rating - best, second best and least performing) can also be observed in the percentage difference graph shown in figure 7.1. From the plot, we can see the performance of other programs with respect to the reference program. The reference program being the least performing program - fall_numpy.py. The program with the lowest percentage difference indicates the best performing program; in this case it is the fall_dpctl_gpu.py program as indicated in the analysis above. The direction field plot for all programs is shown in figure 7.2.

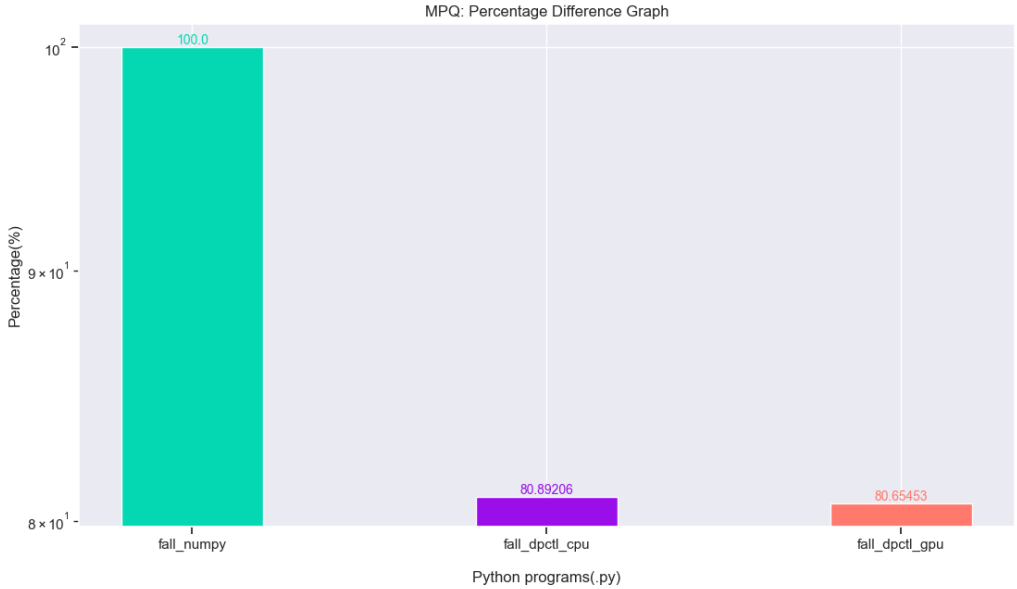
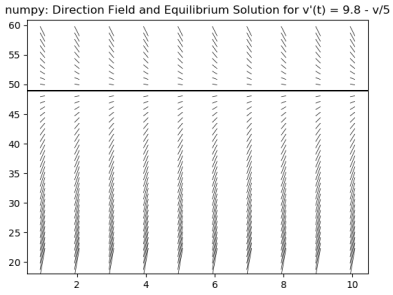
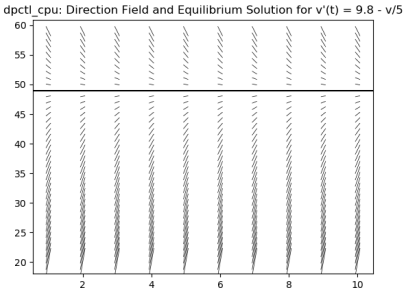


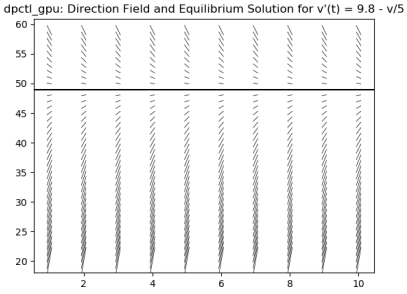
Figure 7.1: MPQ: Percentage Difference Graph



(a) fall_numpy.py



(b) fall_dpctl_cpu.py



(c) fall_dpctl_gpu.py

Figure 7.2: Direction Field of a falling object

Chapter 8

Performance Analysis - SLE method

In this chapter, the performance of the SLE python programs explained in chapter 6, section 6.1 is discussed. The performance of the individual programs will be evaluated with the mathematical formulas for performance introduced in chapter 2.5 and will be based on on their execution times.

8.1 Performance Results

In the projects git repository (here), the programs based on the SLE have been separated into two folders :

- numpy_linspace
- my_linspace

The major difference been that the programs in the second folder are implemented with a user defined function (based on the mathematics of an arithmetic sequence, 6.1.2) that is similar to the functionality of the numpy.linspace() function.

8.1.1 Folder numpy_linspace

Here, we will see the performance results for a numpy.arange() implementation and a prange() implementation. The table D.5 represent the timing of the programs implemented with the additional benefits of the *prange* function from the *numba package*, it is therefore an optimized version of the programs in section 6.1.1 and table D.3.

8.1.1.1 numpy.arange() implementations

The timing results in table D.3, show that:

1. The best performing program is the fall_dpctl_gpu.py.

   comments:

   i. Time reduction in fall_dpctl_gpu.py with respect to fall_numpy.py = 1.83310%.

   Therefore, the deducted time from fall_dpctl_gpu.py is 1.83310% of fall_numpy.py (= 0.00679).

   ii. fall_dpctl_gpu.py is about 1.01867X faster than fall_numpy.py.

   iii. Time reduction in fall_dpctl_gpu.py with respect to fall_dpctl_cpu.py = 0.14829%.

   Therefore, the deducted time from fall_dpctl_gpu.py is 0.14829% of fall_dpctl_cpu.py (= 0.00054).

   iv. fall_dpctl_gpu.py is about 1.00149X faster than fall_dpctl_cpu.py.

2. The second best performing program is the fall_dpctl_cpu.py.

   comments:

   i. Time reduction in fall_dpctl_cpu.py with respect to fall_numpy.py = 1.68732%.

   Therefore, the deducted time from fall_dpctl_cpu.py is 1.68732% of fall_numpy.py (= 0.00625).

   ii. fall_dpctl_cpu.py is about 1.01716X faster than fall_numpy.py.

3. The least performing program is the fall_numpy.py.

Furthermore, the timing behaviour of the programs (based on the rating - best, second best and least performing) can also be observed in the percentage difference graph shown in figure 7.1. From the plot, we can see the performance of other programs with respect to the reference program. The reference program being the least performing program - fall_numpy.py. The program with the lowest percentage difference indicates the best performing program; in this case it is the fall_dpctl_gpu.py program as indicated in the analysis above. The direction field plot for the programs is shown in figure 8.2.

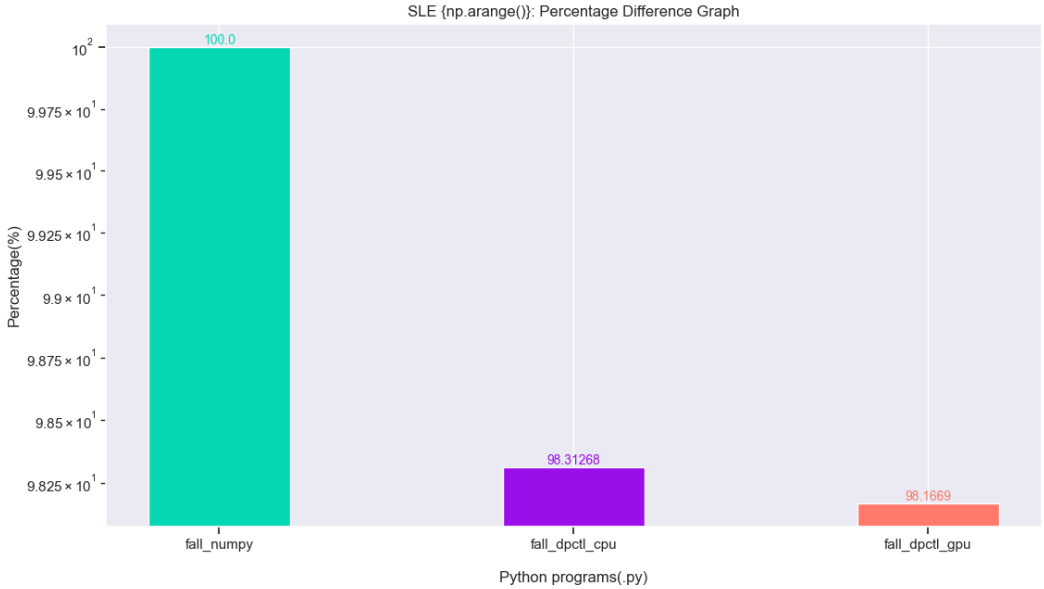
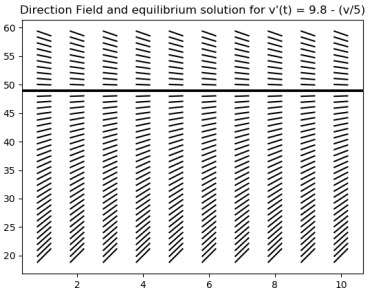
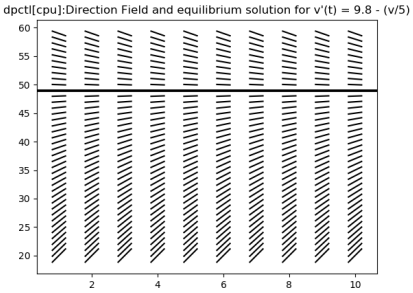


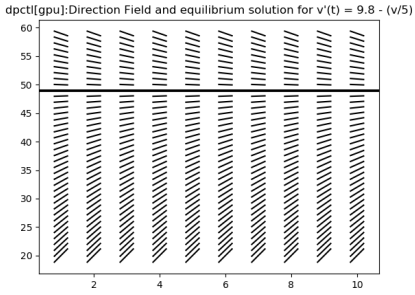
Figure 8.1: SLE np.arange(): Percentage Difference Graph



(a) fall_numpy.py



(b) fall_dpctl_cpu.py



(c) fall_dpctl_gpu.py

Figure 8.2: Direction Field of a falling object

8.1.1.2 prange() implementations

Next, we shall examine the performance of the programs above optimized with the *prange* function of the *numba* package. The results obtained in table, D.5 show that:

1. The best performing program is the fall_dpctl_cpu.py.

   comments:

   i. Time reduction in fall_dpctl_cpu.py with respect to fall_numpy.py = 1.44074%.

   Therefore, the deducted time from fall_dpctl_gpu.py is 1.44074% of fall_numpy.py (= 0.00499).

   ii. fall_dpctl_gpu.py is about 1.01462X faster than fall_numpy.py.

   iii. Time reduction in fall_dpctl_cpu.py with respect to fall_dpctl_gpu.py = 0.25422%.

   Therefore, the deducted time from fall_dpctl_gpu.py is 0.25422% of fall_dpctl_cpu.py (= 0.00087).

   iv. fall_dpctl_cpu.py is about 1.00255X faster than fall_dpctl_gpu.py.

2. The second best performing program is the fall_dpctl_gpu.py.

   comments:

   i. Time reduction in fall_dpctl_gpu.py with respect to fall_numpy.py = 1.18955%.

   Therefore, the deducted time from fall_dpctl_gpu.py is 1.18955% of fall_numpy.py (= 0.00412).

   ii. fall_dpctl_cpu.py is about 1.01204X faster than fall_numpy.py.

3. The least performing program is the fall_numpy.py.

The figure 8.3 below is the percentage difference graph for each python programs discussed above. From the plot, we can see the performance of other programs with respect to the reference program. The reference program being the least performing program - fall_numpy.py. The program with the lowest percentage difference indicates the best performing program; in this case it is the fall_dpctl_cpu.py program as indicated in the analysis above. The direction field plot for the programs is shown in figure 8.4.

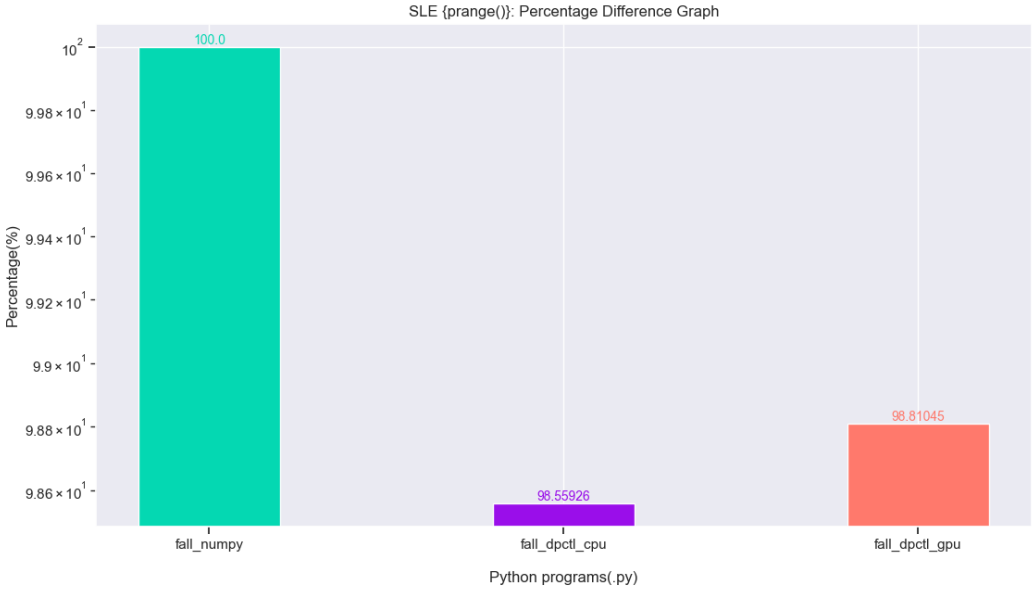
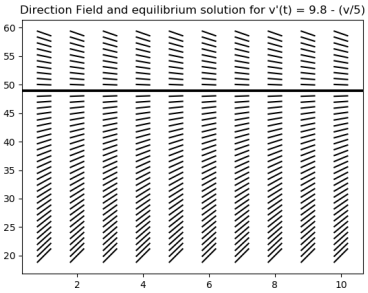
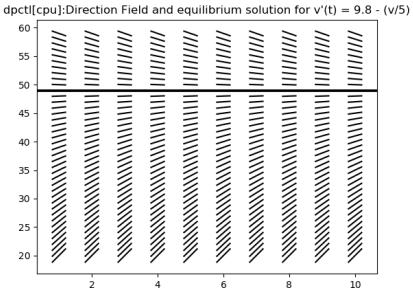


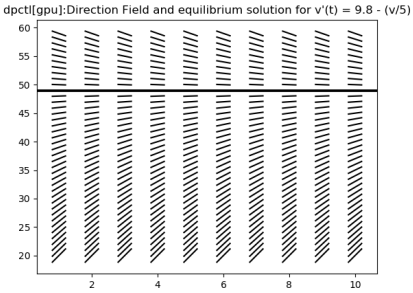
Figure 8.3: SLE prange(): Percentage Difference Graph



(a) fall_numpy.py



(b) fall_dpctl_cpu.py



(c) fall_dpctl_gpu.py

Figure 8.4: Direction Field of a falling object

8.1.2 np.arange() implementations versus prange() implementations

In this section, we shall compare the performance results of the prange() implementations and the numpy.arange() implementations.

The plot shown in figure 8.5 is a comparison of the execution times of the programs with the *prange* implementation with respect to those of the *np.arange* implementation.

The following relations are used in the plot:

prange programs……..np.arange programs

fall_numpy.py……..->……..fall_numpy.py = A

fall_dpctl_cpu.py…..->……fall_dpctl_cpu.py = B

fall_dpctl_gpu.py…..->……fall_dpctl_gpu.py = C

The performance comparison shows that all implementations of the prange version in section 8.1.1.2, perform better than the numpy.arange version in section 8.1.1.1.

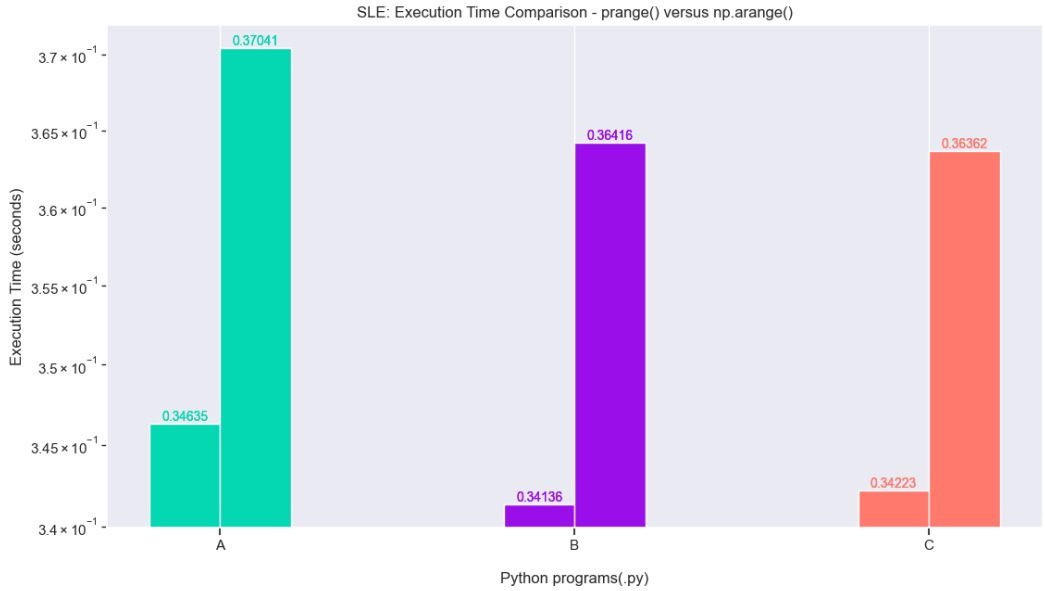


Figure 8.5: SLE : Execution Time Comparison prange() versus np.arange()

8.1.3 Folder my_linspace

In this section, we will see the performance results of the python programs in section 6.1.2 implemented with the user defined linspace function (similar to the numpy.linspace function). The table D.8 represent the timing of the programs. The results obtained show that:

1. The best performing program is the fall_myfunc_dpctl_cpu.py.

   comments:

   i. fall_myfunc_dpctl_cpu.py is 1.10224X, 1.04598X, 1.00794X faster than fall_myfunc.py, fall_myfunc_numba.py, and fall_myfunc_dpctl_gpu.py programs respectively.

   ii. The evaluated values for time reduction of, and the time deducted from fall_myfunc_dpctl_cpu.py with respect to fall_myfunc.py, fall_myfunc_numba.py, and fall_myfunc_dpctl_gpu.py can be found in table D.9.

2. The second best performing program is the fall_myfunc_dpctl_gpu.py.

   comments:

   i. fall_myfunc_dpctl_cpu.py is 1.09355X, 1.03774X faster than fall_myfunc.py and fall_myfunc_numba.py programs respectively.

   ii. The evaluated values for time reduction of, and the time deducted from fall_myfunc_dpctl_gpu.py with respect to fall_myfunc.py and fall_myfunc_numba.py can be found in table D.9.

3. The third best performing program is the fall_myfunc_numba.py.

   comments:

   i. fall_myfunc_numba.py is 1.05379X faster than fall_myfunc.py program.

   ii. The evaluated values for time reduction of, and the time deducted from fall_myfunc_numba.py with respect to can be found in table D.9.

4. The least performing program is fall_myfunc.py

Furthermore, the plot in figure 8.6 below, shows the percentage difference of other programs with respect to the least performing program. The direction field plot for the programs is shown in figure 8.7.

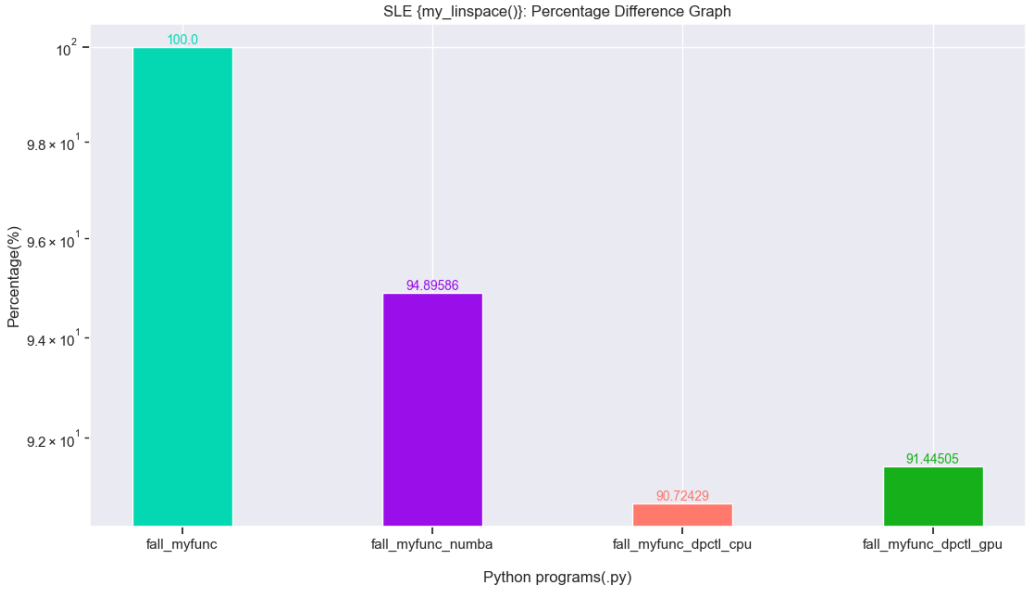
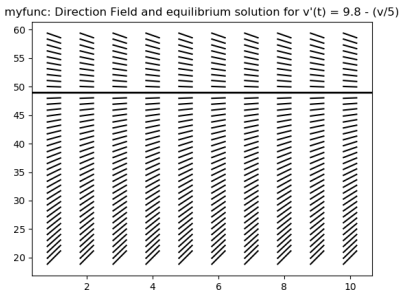
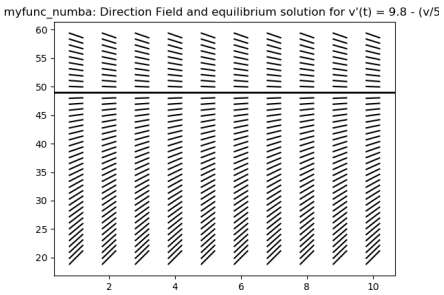


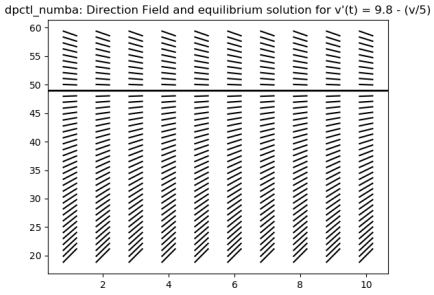
Figure 8.6: SLE : my_linspace(): Percentage Difference Graph



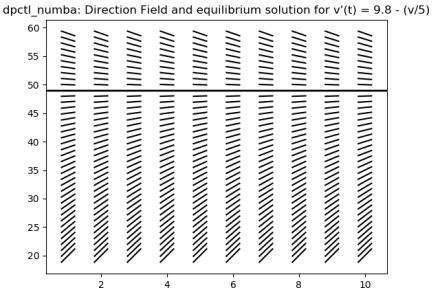
(a) fall_myfunc.py



(b) fall_myfunc_numba.py



(c) fall_myfunc_dpctl_cpu.py



(d) fall_myfunc_dpctl_gpu.py

Figure 8.7: Direction Field of a falling object

8.1.4 my_linspace() implementations versus prange() implementations

In this section, we shall compare the performance results of the my_linspace implementations with the prange() implementations in section 8.1.1.2.

The plot shown in figures 8.8 and 8.9 is a comparison of the execution times of the programs with the *my_linspace()* implementation with respect to those of the *prange* implementation. The following relations are used in the plot in figure 8.8:

my_linspace programs.....np.arange programs

fall_myfunc.py.....->.....fall_numpy.py = A

fall_myfunc_dpctl_cpu.py.....->.....fall_dpctl_cpu.py = B

fall_myfunc_dpctl_gpu.py.....->.....fall_dpctl_gpu.py = C

The following relations are used in the plot in figure 8.9:

my_linspace programs.....np.arange programs

fall_myfunc_numba.py.....->.....fall_numpy.py = A

fall_myfunc_dpctl_cpu.py.....->.....fall_dpctl_cpu.py = B

fall_myfunc_dpctl_gpu.py.....->.....fall_dpctl_gpu.py = C

The performance comparison shows that all implementations of my_linspace() perform better than the prange() implementations in section 8.1.1.2. 8.1.1.1.

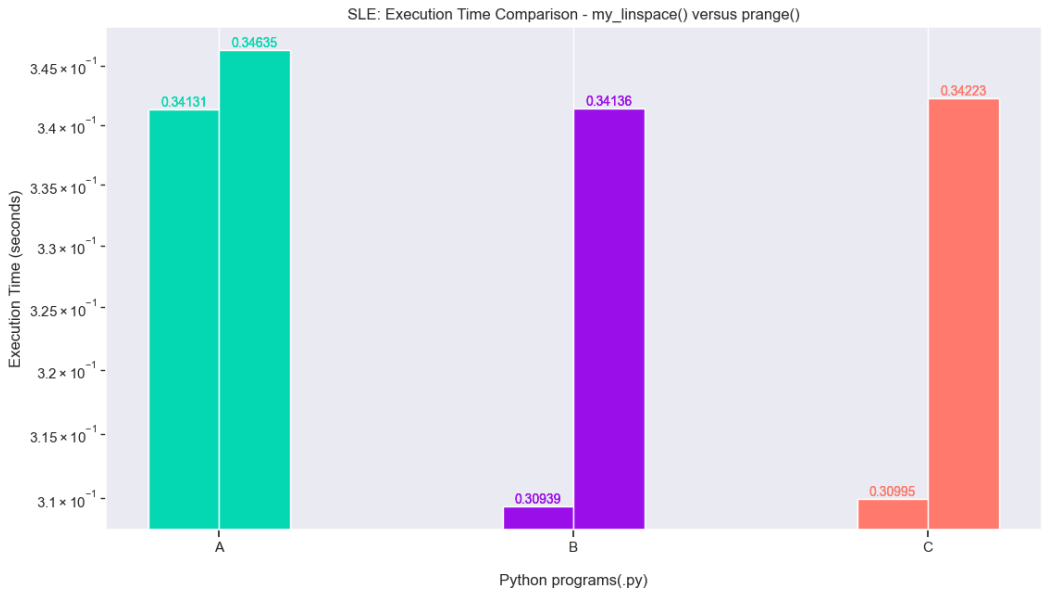


Figure 8.8: SLE : Execution Time Comparison my_linspace() versus prange()

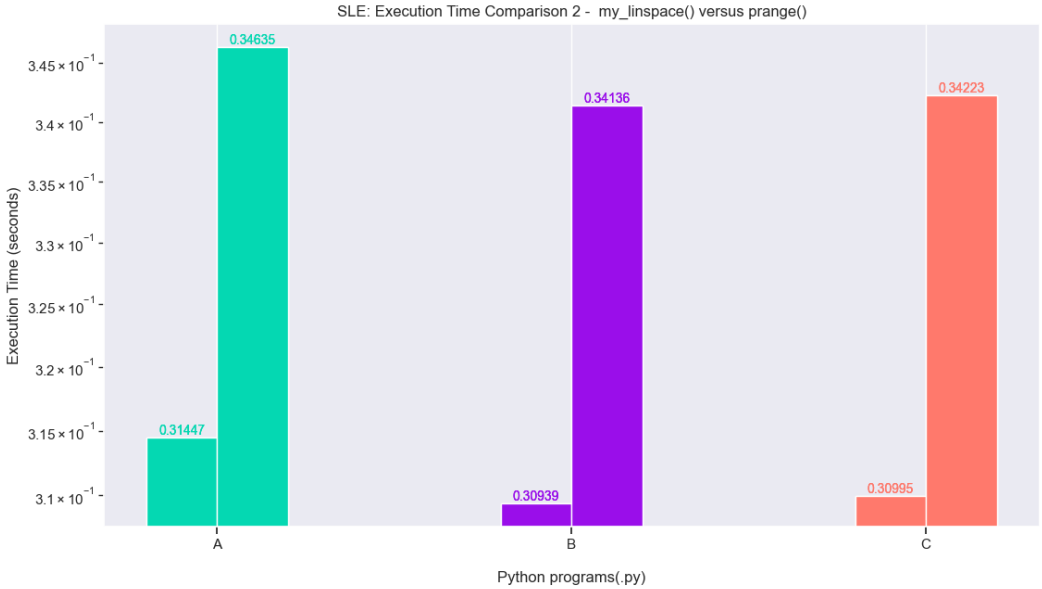


Figure 8.9: SLE : Execution Time Comparison 2 my_linspace() versus prange()

Chapter 9

Comparative Analysis: MPQ versus SLE

Which approach is better ?

The top three programs for the MPQ method are :

1. Folder quiver: fall_dpctl_gpu.py at 0.03056 seconds
2. Folder quiver: fall_dpctl_cpu.py at 0.03065 seconds
3. Folder quiver: fall_numpy.py at 0.03789 seconds

And the top three programs for the SLE method are:

1. Folder my_linspace: fall_myfunc_dpctl_cpu.py at 0.30939 seconds
2. Folder my_linspace: fall_myfunc_dpctl_gpu.py at 0.30995 seconds
3. Folder my_linspace: fall_myfunc_numba.py at 0.31447 seconds

The time results indicate that the MPQ method is the best performing method. But why ? This brings us to examining the complexity of the methods.

9.1 Complexity of the MPQ method

The MPQ method has a linear complexity, O(n); this is imposed by the step for finding the equilibrium solution of the differential equation, which depends on the size of the evaluated differential equation over a 2D-array - a coordinate vector returned by the numpy.meshgrid function. Although there are no use of for loops and other look-out-fors for determining the complexity of an algorithm. The equilibrium_solution() function in the MPQ method however

uses a numpy.where() method which searches through an array, therefore, the timing of the function will grow as the input size grows. Timing tests for the numpy.where() method and the linear search method for finding the equilibrium solution can be found in the doc folder of the git repository of this project (here).

9.2 Complexity of the SLE method

The SLE method on the other hand has a subquadratic complexity, O(n * m); this is imposed by the equilibrium_solution() and solutions() functions. The method depends on the sizes of numerical interval for the x-axis and y-axis; since they are not of the same size, therefore the complexity is less than a quadratic time complexity O(n * n).

At this point, the reason for the difference in performance is clear, the complexity of the MPQ method is lesser than that of the SLE method. This makes the MPQ approach the better approach. There is one good thing about the SLE approach though, it can be implemented with other programming languages other than the python programming language.

Chapter 10

Interpreting the Direction Field

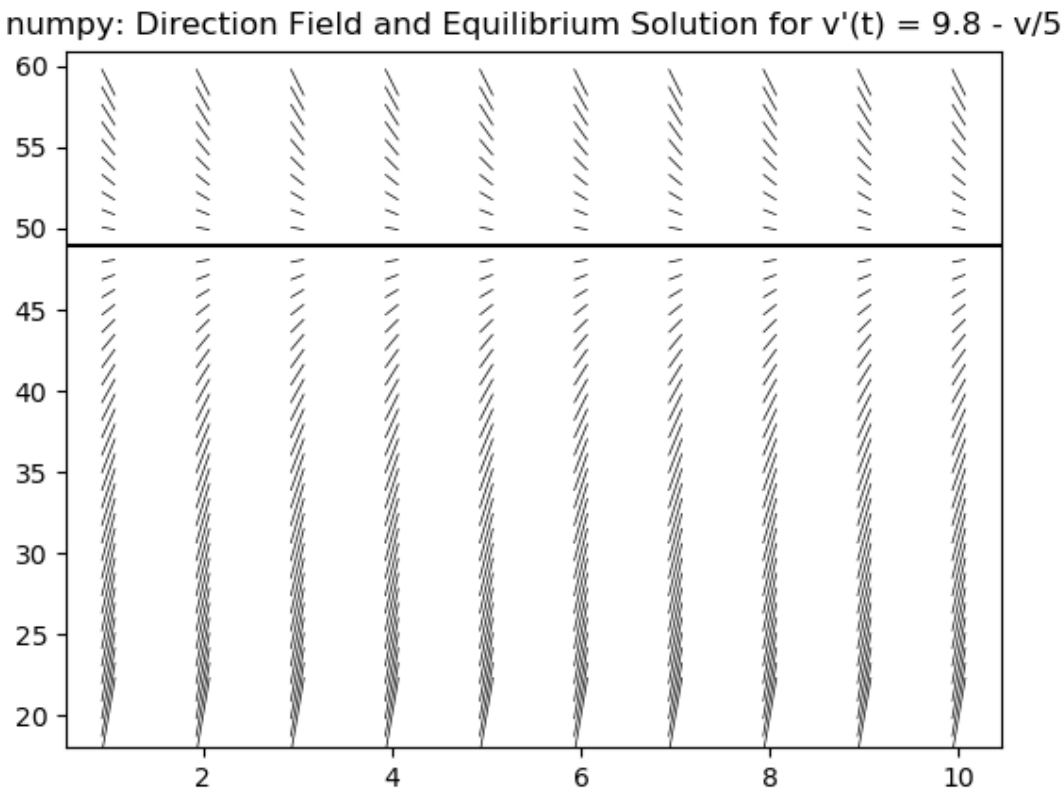


Figure 10.1: Direction Field for dv/dt = 9.8 - v/5

The following statements are true about the figure 10.1:

1. The initial value of v at t = 0 is calculated as 49. This value is known as the equilibrium

solution or the critical value.

2. All solutions converge to or are attracted to the equilibrium solution as t increases. Thus, v = 49, is a asymptotically stable equilibrium solution.

3. If the value of v is greater than the equilibrium solution, the slope, (dv/dt), is negative, so solutions, decrease.

4. If the value of v is lesser than the equilibrium solution, the slope is positive, so solutions, increase.

Chapter 11

Adaptation of the Python Programs to Other Differential Equations

In this chapter, we examine what to do when the python programs presented in this project is to be adapted to plot the direction field of other differential equations of the form 1.1.

11.1 MPQ programs

To adapt the programs based on the matplotlib.pyplot.quiver() method :

- Change the differential_equation function() to evaluate the given differential equation.
- You might have to tune the matplotlib.pyplot.quiver function().
- In the main() function, tune the numpy.arange functions for defining the numerical interval for both the independent and dependent variables accordingly.

11.2 SLE programs

To adapt the programs based on the straight line equation method :

- Change the differential_equation function() to evaluate the given differential equation.
- In the solutions() function, you may have to tune the numerical interval of the variable **X** for obtaining points for the independent variable accordingly.
- In the main() function, tune the numerical interval for both the independent and dependent variables accordingly.

Finally, if you would like to adapt the programs using the MPQ method to display quiver arrows. see sections6, 6.1, 6.2 and 7 of the medium article - How to Plot a Direction Field with Python.

Chapter 12

Conclusions

The following conclusions have been made based on the objectives listed in section 2.2,

12.1 Objective 1

O1. To use the differential equation of a falling object as a case study.

comment:

In this work, the differential equation of a falling object has being used as a case study to demonstrate the visualization of a direction field. Not only has the direction field of the differential equation of a falling object been generated, also included is a step-by-step solution (section C) to show the terms explained in section 1.1 - Solutions of a Differential Equation.

Rating: Satisfactory.

12.2 Objective 2

O2. To use an open source application software in developing programs for the direction field plot.

comment:

The Python Software has been used for developing the programs for this project. It was easy to use and it is readily available on the Intel DevCloud without need for installation. However, to view the images generated by the programs, the *.png file had to be transferred to my local computer.

Rating: Satisfactory.

12.3 Objective 3

O3. To explore the numerical package of the python software (numpy) in developing programs for the direction field plot.

comment:

The use of numpy.meshgrid(), matplotlib.pyplot.quiver() libraries from the numerical package (numpy) proved to be a more efficient method for obtaining the direction field plots. They greatly reduced the complexity of the program and this is the sole reason why the MPQ (matplotlib.pyplot.quiver()) method is the most outstanding approach when compared to the SLE (Straight Line Equation) method. But, like I have pointed earlier (section 9.2), the advantage of the SLE method is that it can be implemented with other programming languages asides Python.

From the statement, above, when developing computer programs, we can learn to always leverage on built-in functions and libraries provided by programming language vendors, for the purpose of balancing performance and productivity.

Rating: Satisfactory.

12.4 Objective 4

O4. To extend the functionality of the programs developed in O3. with packages provided in the Intel's Distribution for Python (IDP*) in order to demonstrate heterogeneous computing.

comment:

Heterogeneous computing is all about discovering the best device for a particular workload. In other words, if an algorithm is best suited for the CPU, run it on the CPU; if it best suited for the GPU, run it on the GPU, e.t.c. This would however not be possible if there is no interface by which a code can run on a selected device.

The Data Parallel Control (dpctl) package provided in IDP* is a great initiative. Python Developers can get to choose which section of their codes run on a particular device (if running the program on an Intel device).

Performance results for this project show that it is well suited for both CPU and GPU devices, but most suited for a GPU device. The overall best program is the GPU version of the program based on the MPQ method.

Rating: Satisfactory.

12.5 Objective 5

O5. To explore the possibilities of increasing performance of the python programs developed in 03. with the data parallel package offered by Intel.

comment:

At the beginning of this project, I have written a note that says on the last line *"...let's go on to see what IDP* can do"*. The results clearly show that the additional functionalities provided in IDP* used in this project improved the performance of the programs not implemented with the additional functionalities.

The best performing program is based on the MPQ method, and has the additional functionality of data parallel control (dpctl) package, enabling it to run on the Integrated Graphics, Intel® UHD Graphics P630 [0x3e96] of the Intel® E-2176G processor.

Rating: Satisfactory.

Appendix

Appendix A

Glossary of acronyms

A.1 WHAT

API *Application Programming Interface.*

CPUs *Central Processing Units.*

CUDA *Compute Unified Device Architecture*

DPC++ *Data Parallel C++*

DPPY *Data Parallel Python*

dpctl *data parallel control*

dpnp *data parallel numerical python*

FPGAs *Field Programmable Gate Arrays*

GPUs *Graphics Processing Units*

IDP* *Intel® Distribution for Python**

NumPy *Numerical Python*

MKL-FFT *Math Kernel Library - Fat Fourier Transforms*

MPQ *matplotlib.pyplot.quiver*

OpenCL *Open Computing Language*

OpenMP *Open Multi-Processing*

Scikit-ipp *Scikit - integrated performance primitives*

SciPy *Scientific Python*

SLE *Straight Line Equation*

XPUs *Other Processing Units*

Appendix B

Python Program Flow

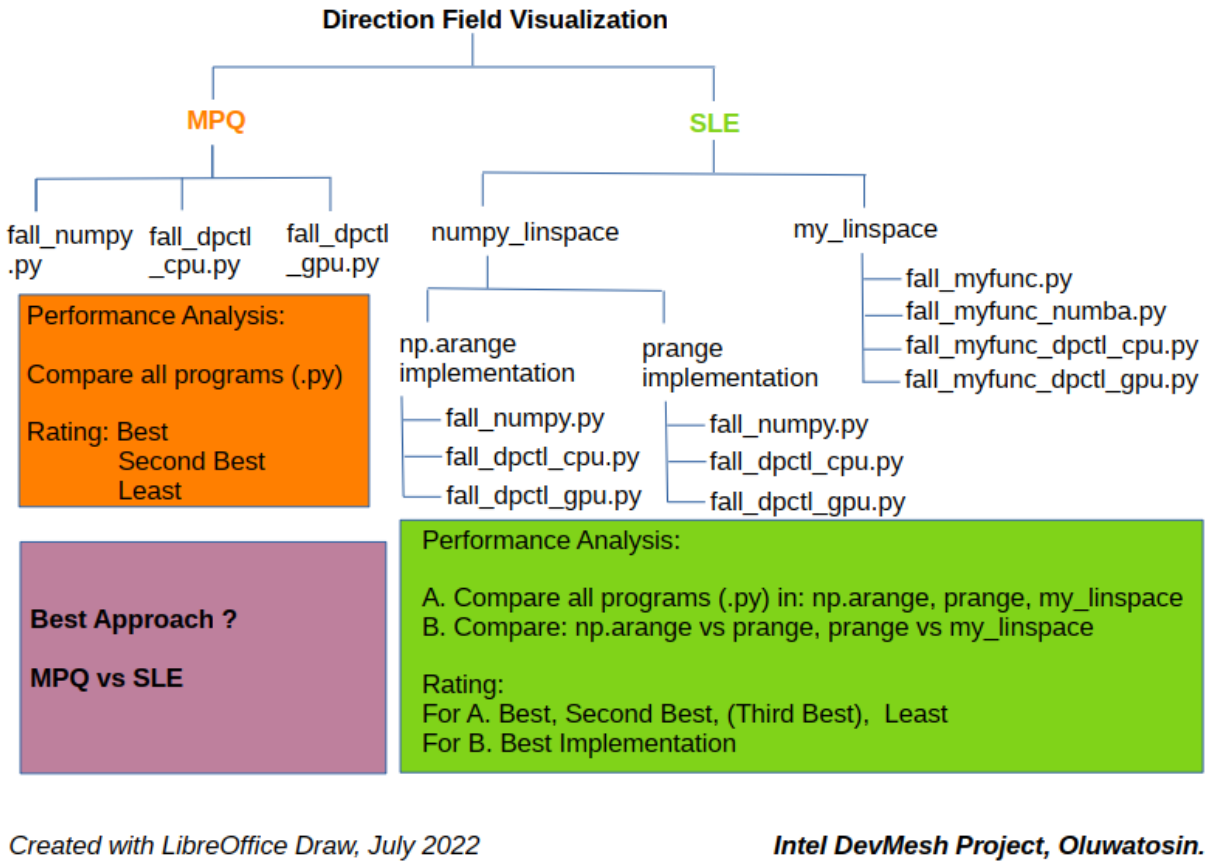


Figure B.1: Program Flow

Appendix C

Solving the Differential Equation

In this section steps to solving the differential equation of the falling object in equation 4.8 is shown.

Question:

Solve the differential Equation -

$$\frac{dv}{dt} = 9.8 - \frac{1}{5}v \tag{C.1}$$

To solve the equation C.1, we need to find functions $v = v(t)$ that when substituted into the equation, satisfies it.

- **Step 1 :** Establish a relationship

  Equation C.1 is a differential equation involving constants and is of the form

  $$\frac{dy}{dx} = ay - b \tag{C.2}$$

  Equation C.1 can therefore be rewritten as

  $$\frac{dv}{dt} = -\frac{1}{5}v + 9.8 \tag{C.3}$$

- **Step 2 :** Transformation

  keys: L.H.S -> Left Hand Side; R.H.S -> Right Hand Side R.H.S of equation C.1, $-\frac{1}{5}v+9.8$ can be rewritten as:

  step 1: split the RHS expression

  $$-\tfrac{1}{5}v = 9.8$$

  step 2: solve for v

  $$v = -49$$

step 3: combine the terms and multiply by the coefficient of v in equation C.3

$$(v - 49) \times -\frac{1}{5} \tag{C.4}$$

Using the new expression in C.4, equation C.3, can now be rewritten as

$$\frac{dv}{dt} = (v - 49) \times -\frac{1}{5} \tag{C.5}$$

Now, transform the equation in C.5, so that only the constant term is left on the R.H.S: (multiply both sides of C.5 by $\frac{1}{v-49}$)

$$\frac{dv/dt}{v - 49} = -\frac{1}{5} \tag{C.6}$$

Finally, transform the equation in C.6, so that the independent variable is on the R.H.S, and the dependent variable is on the L.H.S: (multiply both sides of C.6 by dt)

$$\frac{dv}{v - 49} = -\frac{1}{5}\, dt \tag{C.7}$$

- **Step 3 :** Integration

  Integrate the equation in C.7

$$\int \frac{dv}{v - 49} = \int -\frac{1}{5}\, dt \tag{C.8}$$

$$\int \frac{dv}{v - 49} = -\frac{1}{5} \int dt \tag{C.9}$$

$$\ln(v - 49) = -\frac{1}{5}t + C \tag{C.10}$$

  C is an arbitrary constant of integration

- **Step 4 :** Solve for the general solution:

  Make v in equation C.10 the subject of formula.

  " The exponential function is the inverse of the logarithmic function.

  Thus if $y = e^x$, $x = \ln y$ ".

  Contrasting the statement above with the equation in C.10

$$x = \ln y \tag{C.11}$$

$$-\frac{1}{5}t + C = \ln(v - 49) \tag{C.12}$$

And then using the relation indicated in the "if....." statement above, we have

$$(v - 49) = e^{-\frac{1}{5}t + C} \tag{C.13}$$

Using the multiplication law of indices, the equation in C.12 can be rewritten as

$$v - 49 = e^{-\frac{1}{5}t} \times e^{C} \tag{C.14}$$

Now, let c = e^C, equation in C.12, will become

$$v - 49 = ce^{-\frac{1}{5}t} \tag{C.15}$$

And, solving for v, we have

$$v = ce^{-\frac{1}{5}t} + 49 \tag{C.16}$$

Just a little rearrangement

$$v = 49 + ce^{-\frac{1}{5}t} \tag{C.17}$$

c is also an arbitrary constant. If c = 0; the value of v(t) = 49, which the equilibrium solution.

The equation C.17, is known as the general solution of equation C.1. which results in an infinite family of solutions.

- **Step 5 :** Solving for a particular solution

  Solving the equation C.1 for a particular solution, requires an initial condition. If an initial condition is given, the equation C.1 is called an initial value problem.

  For the initial condition, we will use:

  Initial velocity = 0; this also means the object is at rest at this velocity and so t = 0.

  Therefore, the initial condition is $t = 0\ ; v = 0$. So,

$$v(0) = 0 \tag{C.18}$$

With this information, we first find the arbitrary constant, c by substituting the initial conditions into equation C.17, we will have

$$0 = 49 + ce^{-\frac{1}{5}\times 0} \tag{C.19}$$

$$0 = 49 + c \tag{C.20}$$

$$c = -49 \tag{C.21}$$

Therefore, the solution of the initial value problem (which is also a particular solution due to the initial conditions given) by substituting the value of c obtained in equation C.21 into equation C.17 is

$$v = 49 + (-49e^{-\frac{1}{5}t}) \tag{C.22}$$

$$v = 49 - 49e^{-\frac{1}{5}t} \tag{C.23}$$

Factoring the equation C.23, we have

$$v = 49(1 - e^{-\frac{1}{5}t}) \tag{C.24}$$

Appendix D

Tables of Results

The execution time of the individual python programs was taken on the node: s001-n158.

D.1 MPQ Implementation

Table D.1: MPQ implementations

Programs	T_1**(secs)**	T_2**(secs)**	T_3**(secs)**	T_{avg}**(secs)**
fall_numpy.py	0.03892	0.03766	0.03709	0.03789
fall_dpctl_cpu.py	0.03057	0.03092	0.03046	0.03065
fall_dpctl_gpu.py	0.03043	0.03070	0.03054	0.03056

Table D.2: Time Reduction, Time Deduction and Time Gain of MPQ implementations

Program comparison	**Time Reduction (%)**	**Time Deduction (secs)**	**Time Gain**
fall_dpctl_gpu.py vs fall_numpy.py	19.34547	0.00733	1.23986
fall_dpctl_gpu.py vs fall_dpctl_cpu.py	0.29364	0.00009	1.00295
fall_dpctl_cpu.py vs fall_numpy.py	19.10794	0.00724	1.23622

D.2 SLE implementation

Folder numpy_linspace

D.2.1 SLE np.arange() Implementation

Table D.3: SLE (np.arange) implementations

Programs	T_1**(secs)**	T_2**(secs)**	T_3**(secs)**	T_{avg}**(secs)**
fall_numpy.py	0.36817	0.36941	0.37365	0.37041
fall_dpctl_cpu.py	0.36377	0.36345	0.36526	0.36416
fall_dpctl_gpu.py	0.36344	0.36194	0.36549	0.36362

Table D.4: Time Reduction, Time Deduction and Time Gain of SLE (np.arange) implementations

Program comparison	**Time Reduction (%)**	**Time Deduction (secs)**	**Time Gain**
fall_dpctl_gpu.py vs fall_numpy.py	1.83310	0.00679	1.01867
fall_dpctl_gpu.py vs fall_dpctl_cpu.py	0.14829	0.00054	1.00149
fall_dpctl_cpu.py vs fall_numpy.py	1.68732	0.00625	1.01716

D.2.2 SLE prange() Implementation

Table D.5: SLE (prange) implementations

Programs	T_1**(secs)**	T_2**(secs)**	T_3**(secs)**	T_{avg}**(secs)**
fall_numpy.py	0.34512	0.34650	0.34744	0.34635
fall_dpctl_cpu.py	0.34093	0.33942	0.34373	0.34136
fall_dpctl_gpu.py	0.34086	0.34192	0.34391	0.34223

Table D.6: Time Reduction, Time Deduction and Time Gain of SLE (prange) implementations

Program comparison	**Time Reduction (%)**	**Time Deduction (secs)**	**Time Gain**
fall_dpctl_cpu.py vs fall_numpy.py	1.44074	0.00499	1.01462
fall_dpctl_cpu.py vs fall_dpctl_gpu.py	0.25422	0.00087	1.00255
fall_dpctl_gpu.py vs fall_numpy.py	1.18955	0.00412	1.01204

D.2.3 SLE prange() vs SLE np.arange() Implementation

Table D.7: Comparison: SLE (prange) vs SLE (np.arange) implementations

Program comparison	**Time Reduction (%)**	**Time Deduction (secs)**	**Time Gain**
fall_numpy.py	6.49550	0.02406	1.06947
fall_dpctl_cpu.py	6.26098	0.02280	1.06679
fall_dpctl_gpu.py	5.88251	0.02139	1.06250

Folder my_linspace

D.2.4 SLE my_linspace() Implementation

Table D.8: sle (my_linspace) implementations

Programs	T_1**(secs)**	T_2**(secs)**	T_3**(secs)**	T_{avg}**(secs)**
fall_myfunc.py	0.33992	0.34272	0.34130	0.34131
fall_myfunc_numba.py	0.32257	0.31143	0.30941	0.31447
fall_myfunc_dpctl_cpu.py	0.30839	0.31253	0.30724	0.30939
fall_myfunc_dpctl_gpu.py	0.31084	0.30908	0.30994	0.30995

Table D.9: Time Reduction, Time Deduction and Time Gain of SLE (my_linspace) implementations

Program comparison	**Time Reduction (%)**	**Time Deduction (secs)**	**Time Gain**
fall_myfunc_dpctl_cpu.py vs fall_myfunc.py	9.27571	0.03153	1.10224
fall_myfunc_dpctl_cpu.py vs fall_myfunc_numba.py	4.39677	0.014183	1.04598
fall_myfunc_dpctl_cpu.py vs fall_myfunc_dpctl_gpu.py	0.78819	0.00245	1.00794
fall_myfunc_dpctl_gpu.py vs fall_myfunc.py	8.55495	0.02908	1.09355
fall_myfunc_dpctl_gpu.py vs fall_myfunc_numba.py	3.63642	0.01173	1.03774
fall_myfunc_numba.py vs fall_myfunc.py	5.10414	0.01735	1.05379

Table D.10: Comparison: SLE (my_linspace) vs SLE (numpy_linspace [prange]) implementations

Program comparison	**Time Reduction (%)**	**Time Deduction (secs)**	**Time Gain**
fall_myfunc.py vs fall_numpy.py	9.20456	0.03188	1.10138
fall_myfunc_dpctl_cpu.py vs fall_dpctl_cpu	9.36547	0.03197	1.10333
fall_myfunc_dpctl_gpu.py vs fall_dpctl_gpu	9.43225	0.03228	1.10415

Appendix E

Staying Up-To-Date

Explore these Intel's repositories to stay abreast of Intel's latest methods applicable in Numerical and Scientific Computing.

1. Python : idp*.html, github- IntelPython
2. one Math Kernel Library : onemkl.html, github - oneMKL, oneAPI-samples, oneAPI-samples-libraries
3. one Data Parallel Library : dpc-library.html, github - oneDPL, oneAPI-samples, oneAPI-samples-libraries
4. one Threading Building Block : onetbb.html, github - oneAPI-samples, oneAPI-samples-libraries
5. Data Parallel C++ : dpc-compiler.html, github - dpc++ compiler, oneAPI-samples
6. Intel DPC++ Compatibility tool : dpc-compatibility-tool.html
7. Intel MPI Library : mpi-library.html
8. Intel openMP* Runtime Library : openmp.html, github - oneAPI-samples

To find out about other oneAPI components, visit oneapi-components.html.

References

[1] W. E. Boyce and R. C. DiPrima, *Elementary Differential Equations and Boundary Value Problems, eigth edition.* USA: John Wiley & Sons, Inc., 2005.

[2] S. Holzner, *Differential Equations For Dummies.* Indiana: Wiley Publishing, Inc., 2008.

[3] B. D. Bunday and H. Mulholland, *Pure Mathematics for Advanced Level, second edition.* Heinemann Eductional Books (Nigeria) Plc, 2004.

[4] P. Howard, "Ordinary differential equations in matlab," 2013.

[5] O. Odubanjo, "How to plot a direction field with python," Medium, December, 27 2021. [Online]. Available: https://medium.com/@olutosinbanjo/how-to-plot-a-direction-field-with-python-1fd022e2d8f8

[6] P. Dawkins, "Paul's online notes." [Online]. Available: https://tutorial.math.lamar.edu/

[7] Mic, "Generate slope fields in r and python," R-bloggers, 2015. [Online]. Available: https://www.r-bloggers.com/2014/09/generate-slope-fields-in-r-and-python/amp/