UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA
Denkleiers • Leading Minds • Dikgopolo tša Dihlalefi

Department of Computer Science
Faculty of Engineering, Built Environment & IT
University of Pretoria

COS110 - Program Design: Introduction

Assignment 1 Specifications

Release Date: 29-07-2024 at 06:00

Due Date: 23-08-2024 at 23:59

Total Marks: 250

Contents

1 General Instructions

- ***Read the entire assignment thoroughly before you start coding.***
- This assignment should be completed individually, no group effort is allowed.
- **To prevent plagiarism, every submission will be inspected with the help of dedicated software.**
- Be ready to upload your assignment well before the deadline, as **no extension will be granted.**
- If your code does not compile, you will be awarded a mark of 0. The output of your program will be primarily considered for marks, although internal structure may also be tested (eg. the presence/absence of certain functions or classes).
- Failure of your program to successfully exit will result in a mark of 0.
- Note that plagiarism is considered a very serious offence. Plagiarism will not be tolerated, and disciplinary action will be taken against offending students. Please refer to the University of Pretoria's plagiarism page at `https://portal.cs.up.ac.za/files/departmental-guide/`.
- Unless otherwise stated, the usage of c++11 or additional libraries outside of those indicated in the assignment, will **not** be allowed. Some of the appropriate files that you have submit will be overwritten during marking to ensure compliance to these requirements. **Please ensure you use c++98**
- All functions should be implemented in the corresponding `cpp` file. No inline implementation in the header file apart from the provided functions.
- The usage of ChatGPT and other AI-Related software is strictly forbidden and will be considered as plagiarism.
- Note, UML notation for variable and function signatures are used in this assignment.
- Please note that there is a late deadline which is 1 hour after the initial deadline, but you will lose 20% of your mark.

2 Overview

For this assignment, you will be implementing a Particle Swarm Optimization (PSO) algorithm in C++. This algorithm will be used to demonstrate how to handle dynamic memory, and how to correctly use classes. The algorithm will also be used to illustrate the different ways function signatures can be designed in a class to allow for different functionality. In this assignment, you will gain experience with semi-complex class structures and pointers.

3 Background

3.1 PSO

Particle Swarm Optimization (PSO) is a population-based optimisation algorithm that is based on the cooperative behaviour of a flock of birds [3, 2]. PSO algorithms have been used to great success in finding the approximate global minimum of an arbitrary N-dimensional non-linear function [1]. The global minimum of a function is the lowest value that can be calculated by the function[1]. For example, the global minimum of $f(x) = x^2 + 5$ is 5 (plot $f(x) = x^2 + 5$ onto an online graphing tool to visualize that 5 is the minimum of the function). The process of finding the global minimum becomes complex when dealing with multi-variate functions.

The main idea behind the basic PSO algorithm (which is described by Algorithm 1) is that through a series of iterations the population (list) of particles will slowly search around different combinations of possible values (search space), guided by a special update rule, for the approximate global minimum of a function.

Algorithm 1 PSO Pseudo code

```
Require: a list of particles
Require: a number of iterations
  for each iteration do
    bestParticle ← inf
    for particle in the list of particles do
      value := evaluate(particle)
      if value < bestParticle then
        bestParticle := particle
      end if
    end for
    for particle in the list of particles do
      updateParticle(particle, bestParticle);
    end for
  end for
```

Each particle has a position ($\mathbf{x}$) and a velocity ($\mathbf{v}$). The position represents the input values to the function that is being minimalized and the velocity represents the amount of movement a particle will experience after each iteration. The velocity is used to move the particle around the search space towards the approximate global minimum.

In order to update the position of a particle after each iteration the following expressions (which are simplified from [3] for the sake of this assignment) will be used[2]:

[1] Note this is an oversimplification of the definition

[2] Bold letters represent vectors and non-bold letters represent scalars.

The velocity update rule

$$\mathbf{v}' = \omega\mathbf{v} + c_1(\mathbf{x}_{best} - \mathbf{x}) + c_2(\mathbf{x}_{global\_best} - \mathbf{x}) \tag{1}$$

Where:

- $\mathbf{v}'$ is a vector which represents the new velocity
- $\mathbf{v}$ is a vector which represents the particle's current velocity.
- $\mathbf{x}$ is a vector representing the current position of the particle.
- ω is the inertia weight, which regulates the importance of the previous velocity.
- c_1 is the cognitive acceleration coefficient which regulates the importance of the particle's own success.
- c_2 is the social acceleration coefficient which regulates the importance of the global best particle.
- $\mathbf{x}_{best}$ is a vector that represents the best **position** that the particle has obtained so far[3].
- $\mathbf{x}_{global\_best}$ is a vector that represents the best **position** from all of the particles for a specific iteration.

The position update rule

$$\mathbf{x}' = \mathbf{x} + \mathbf{v}' \tag{2}$$

Where:

- $\mathbf{x}'$ is a vector that represents the new position of the particle.
- $\mathbf{x}$ is a vector that represents the current position of the particle.
- $\mathbf{v}'$ is a vector that represents the new velocity of a particle.

An example of the calculations is provided in Section 10.

An illustrative example of a PSO algorithm is given in Figure 1. Here the particles are attempting to find the center of the block. As can be seen in Figure 1 over a series of iterations all the particles

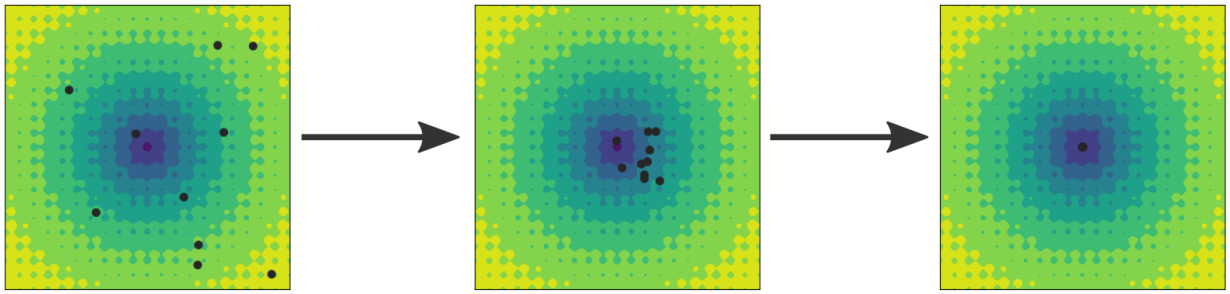

Figure 1: Illustration of PSO algorithm

were able to cluster onto the centre of the block.

[3]Best is defined as the lowest result from the evaluation function

4 Your Task:

You are required to implement the following class diagram illustrated in Figure 2. Pay close attention to the function signatures as the `h` files will be overwritten, thus failure to comply with the UML, will result in a mark of 0.

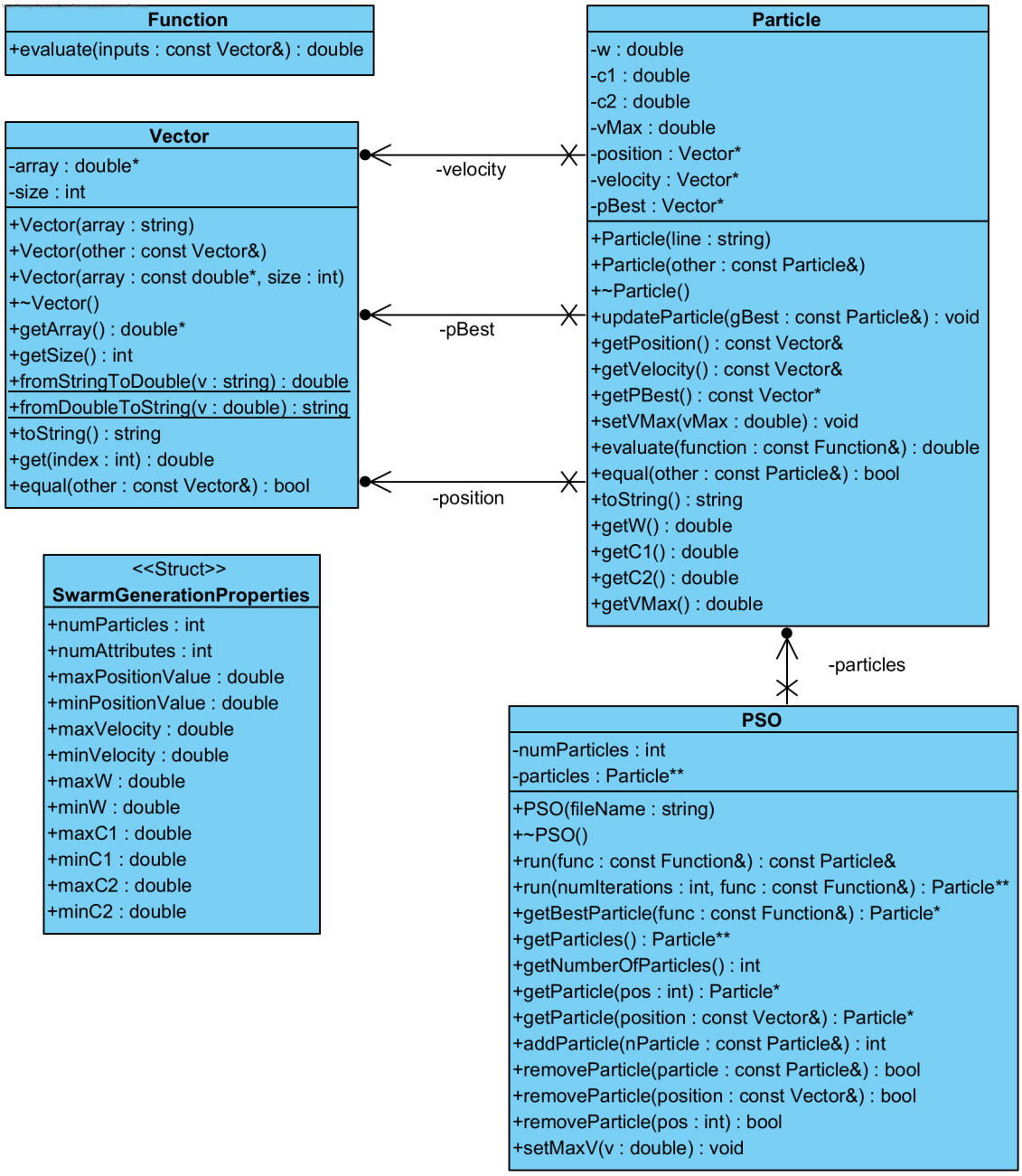

Figure 2: Class diagrams

Note, the importance of the arrows between the classes are not important for COS 110 and will be expanded on in COS 214.

4.1 Vector

The vector class will form a fundamental part of the assignment.

4.1.1 Members

- -array: double*
  - This member variable is a dynamic 1D array that contains the components that are contained in the vector.
- -size: int
  - This member variable contains the dimensionality or size of the vector (which will be represented as an array in this assignment).

4.1.2 Functions

- +Vector(array: string)
  - This is a string constructor for the Vector class.
  - The format of the string is given below:

```
[component1, component2, ..., componentN]
```

  - Your code will need to determine the number of components that are present in the string and initialize the size and array member variables accordingly.
  - After this, your code will need to extract each component and convert it from string to double and place it in the appropriate position in the array.
  - It can be assumed that the input string is valid.
  - Example: Given the following input string:

```
[0.1, 33.4, 4.132]
```

    The array needs to be populated as follows, with the size being 3:

Index	Value
0	0.1
1	33.4
2	4.132

Table 1: Example of vector array

- +Vector(other: const Vector&)
  - This is the copy constructor for the Vector class.
  - This copy constructor needs to create a deep copy of the passed-in parameter.

- +Vector(array: const double*, size: int)
  - This is the array constructor for the Vector class.
  - Your code needs to create a new array using the passed-in size parameter and populate it with the contents of the array parameter.
  - A deep copy of the array needs to be made.
  - If the size is negative or the passed-in array is NULL, initialize the array member variable to be NULL and the size member variable to be 0.
- +~Vector();
  - This is the destructor for the Vector class.
  - The destructor should deallocate all dynamic memory allocated to the current object.
- +getArray(): double*
  - This is a constant function.
  - This function needs to return the array member variable.
- +getSize(): int
  - This is a constant function.
  - This function needs to return the size member variable.
- +fromStringToDouble(v: string): double
  - This is a static function,
  - This function needs to convert the passed-in string to a double value.
  - *Hint: The use of the stringstream library may come in handy*
  - It can be assumed that only valid strings will be tested on FitchFork.
- +fromDoubleToString(v: double): string
  - This is a static function,
  - This function needs to convert the passed-in double to a string value.
  - If the double is an integer number, you will need to manually add ".0" to the end of said integer. In other words, if the resulting string is "1" you will need to change it to "1.0", but if the resulting string is "5.4" you need to leave it as "5.4".
- +toString(): string
  - This function needs to return a string representation of the vector.
  - The format of the result is the same as the format that is passed-into the string constructor of the vector class.
  - Ensure that this function is correct as it will be used extensively while testing your code on FitchFork.

- +get(index: int): double
  - This function needs to return the component at the specified index.
  - If the passed-in index is outside the bounds of the array, the function needs to return 0.
- +equal(other: const Vector&): bool
  - This function needs to determine if the current *Vector* object and the passed-in parameter are equal.
  - Two vectors are deemed equal if each component in the array is the same.
  - If the two vectors are equal return true, else return false.

4.2 Function

This class represents the function that will be minimized. This class will be overwritten on FitchFork and as such, you are allowed to change the function ($f(x, y) = 0.1x^2 + 3y^2$) that will be minimized for your own testing.

4.2.1 Function

- +evaluate(inputs: const Vector&): double
  - This function will perform some calculations based on the components of the passed-in vector parameter and return the result of these calculations.
  - You have been provided with an the function $f(x, y) = 0.1x^2 + 3y^2$, which you may change to test your code.

4.3 Particle

Objects of this class represent the particles that will be used in the PSO algorithm.

4.3.1 Members

- -w: double
  - This member variable represents the inertia weight of the particle.
- -c1: double
  - This member variable represents the cognitive acceleration coefficient.
- -c2: double
  - This member variable represents the social acceleration coefficient.
- -vMax: double
  - This member variable represents the absolute maximum velocity of the particle.

- -position: Vector*
  - This member variable represents the position of the particle.
- -velocity: Vector*
  - This member variable represents the velocity of the particle.
  - It can be assumed that the position and the velocity have the same amount of components.
- -pBest: Vector*
  - This member variable represents the best position that the current particle has ever obtained.

4.3.2 Functions

- +Particle(line: string)
  - This is the string constructor of the particle.
  - The format of the string is as follows:

    ```
    |[pC1,...,pCN]|[vC1,...,vCN]|w|c1|c2|
    ```

    where:
    - *pC* stands for positionComponent
    - *vC* stands for velocityComponent
    - *w* is the inertia weight
    - *c1* is the cognitive acceleration coefficient
    - *c2* is the social acceleration coefficient
  - Your code will need to initialize each of the respective member variables accordingly.
  - Set *vMax* to have an initial value of 1 and *pBest* a value of NULL.
  - An example of an input string will be:

    ```
    |[0.1,1.22]|[3,2.2]|0.1|0.5|0.01|
    ```

- +Paricle(other: const Particle&)
  - This is the copy constructor for the particle class.
  - This constructor needs to perform a deep copy of the passed-in parameter.
- +∼Particle()
  - This is the destructor of the particle class.
  - The destructor needs to deallocate all dynamic memory allocated to the current object.

- +updateParticle(gBest: const Particle&): void
  - This function needs to update the particle's position and velocity using the method described in Algorithm 2

Algorithm 2 Update rule algorithm for a particle

Require: global best particle
  for i between [0;size of the position vector) **do**
    if pBest not NULL **then**
      velocity's i^{th} component := using Equation 1, calculate the new i^{th} velocity component.
    else
      velocity's i^{th} component := using Equation 3, calculate the new i^{th} velocity component.
    end if
    if velocity's i^{th} component is greater than vMax **then**
      velocity's i^{th} := vMax
    else
      if velocity's i^{th} component is less than $-1\times$vMax **then**
        velocity's i^{th} := $-1\times$vMax
      end if
    end if
    position's i^{th} component := using Equation 2, calculate the new i^{th} position component.
  end for

  - *Hint: Remember to use the position of gBest.*
  - Please see Section 10 for an example of the calculations.
- +getPosition(): const Vector&
  - This is a constant function.
  - This function needs to return the *position* member variable.
- +getVelocity(): const Vector&
  - This is a constant function.
  - This function needs to return the *velocity* member variable.
- +getPBest(): const Vector*
  - This is a constant function.
  - This function needs to return the *pBest* member variable as is, including if it is NULL.
- +setVMax(vMax: double): void
  - This function needs to set the *vMax* member variable to the absolute value of the passed-in member variable.
- +evaluate(function: const Function&): double
  - This function needs to call the evaluate function of the passed-in parameter and pass to it the position member variable.

  - The function then needs to return the result obtained from the *evaluate* function.
  - This function also needs to determine if the current position of the particle is better than the position stored in the *pBest* member variable. Algorithm 3 described the algorithm for determine if *pBest* needs to be updated.

Algorithm 3 Algorithm for determining if the pBest needs to be updated

```
if pBest not NULL then
    currentValue := function's evaluate function called with the position member variable.
    pBestValue := function's evaluate function called with the pBest member variable.
    if currentValue < pBestValue then
        pBest := deep copy of the position member variable
    end if
else
    pBest := deep copy of the position member variable
end if
```

- +equal(other: const Particle&): bool
  - This is a constant function.
  - This function needs to determine if two particles are equal.
  - Two particles are seen as equal if the position vectors and velocity vectors of both particles are equal.
  - If the particles are equal return true, else return false.
- +toString(): string
  - This is a provided function and will be overwritten.
  - This function will be used to print out and test the particles.
- +getW(): double
  - This is a constant function.
  - This function needs to return the *w* member variable.
- +getC1(): double
  - This is a constant function.
  - This function needs to return the *c1* member variable.
- +getC2(): double
  - This is a constant function.
  - This function needs to return the *c2* member variable.
- +getVMax(): double
  - This is a constant function.
  - This function needs to return the *vMax* member variable.

4.4 PSO

This class is responsible for the management and overall overhead of the PSO algorithm.

4.4.1 Members

- -particles: Particle**
  - This is a dynamic 1D array of dynamic Particle objects.
- -numParticles: int
  - This member variable stores the total number of particles present in the array.

4.4.2 Functions

- +PSO(fileName: string)
  - This is the textfile constructor for the PSO class.
  - The textfile has the following format:

```
|[pC1,...,pCN]|[vC1,...,vCN]|w|c1|c2|
|[pC1,...,pCN]|[vC1,...,vCN]|w|c1|c2|
|[pC1,...,pCN]|[vC1,...,vCN]|w|c1|c2|
|[pC1,...,pCN]|[vC1,...,vCN]|w|c1|c2|
|[pC1,...,pCN]|[vC1,...,vCN]|w|c1|c2|
```

    where each row is a new particle.
  - The constructor will need to determine the number of particles in the textfile, extract each particle, and then populate the particles array correctly after creating the array.
  - It can be assumed that all the particles in the textfile is correct.
- +~PSO()
  - This is the destructor for the PSO class.
  - The destructor needs to deallocate all dynamic memory allocated to the current PSO object.
- +run(func: const Function&): const Particle&
  - This function will act as the main driver function for each iteration of the PSO algorithm.
  - The function will need to find the best particle in the list of particles and use that to update each of the particles in the list of particles.
  - The function should then return the best particle that was used to update the rest of the particles.
  - Algorithm 4 describes the internal workings of the function.

Algorithm 4 Main driver function for the PSO algorithm

Require: Function object to minimize.
  gBest := the best particle in the list of particles.
  for each particle in the list of particles **do**
    update the particle using gBest.
  end for
  return gBest

- +run(numIterations: int, func: const Function&): Particle**
  - This function will act as the outer loop of the PSO algorithm.
  - The function will return a 1D dynamic array of dynamic objects which represents the best particle for each iteration.
  - This function will loop *numIterations* amount of times and in each iteration call `run(func: const Function&)`.
  - A deep copy of the result from this mentioned function call needs to be stored in the mentioned 1D dynamic array.
- +getBestParticle(func: const Function&): Particle*
  - The function needs to determine and return the best particle in the population using the passed-in *func* object.
  - The best particle is the particle that produces the lowest value.
- +getParticles(): Particle**
  - This is a constant function.
  - This function needs to return the *particles* array.
- +getNumberOfParticles(): int
  - This is a constant function.
  - This function needs to return how many particles are in the current object.
- +getParticle(pos: int): Particle*
  - This is a constant function.
  - This function needs to return the particle located at the passed-in parameter in the particles array.
  - If the passed-in parameter is outside the bounds of the array, the function needs to return NULL.

- +getParticle(position: const Vector&): Particle*
  - This is a constant function.
  - This function needs to return the first particle that has the same position as the passed-in parameter.
  - If no particle is found return NULL.
- +addParticle(nParticle: const Particle&): int
  - This function needs to add the deep copy of the passed-in particle to the end of the array.
  - This means that the function will need to expand the array to accommodate the new particle.
  - Remember to correctly manage your memory and to change the number of particles accordingly.
  - Return the index in the array that the new particle was inserted into.
- +removeParticle(particle: const Particle&): bool
  - This function needs to remove and deallocate any particles that are equal to the passed-in parameter.
  - After removing the particles you will need to shift up all the particles and resize the array such that there are no gaps.
  - Remember to change the number of particles accordingly.
  - If a particle(s) was removed return true, else return false.
- +removeParticle(position: Vector&): bool
  - This function needs to remove and deallocate any particles whose positions are equal to the passed-in parameter.
  - After removing the particles you will need to shift up all the particles and resize the array such that there are no gaps.
  - Remember to change the number of particles accordingly.
  - If a particle(s) was removed return true else return false.
- +removeParticle(pos: int): bool
  - This function needs to remove and deallocate the particle at the passed-in index in the particles array.
  - If the passed-in position is outside of the bounds of the array, return false.
  - If the particle was removed, return true.
- +setMaxV(v: double): void
  - This is a constant function.
  - This function needs to loop through all of the particles in the particles array and set the maximum velocity to the passed-in parameter.

4.5 SwarmGenerationProperties

This struct and functions have been provided for you and will be overwritten on FitchFork. The intention of this struct is to be able to randomly generate a set of particles for which you can then run your PSO algorithm on. The struct's member variables can be set to configure the particles that will be generated. The `generateSwarm` function will generate a swarm based on the passed-in properties object and seed. The `toFile` function will save the a string to the specified textfile name.

5 Memory Management

As memory management is a core part of COS110 and c++, each task on FitchFork will allocate approximately 10% of the marks to memory management. The following command is used:

```
valgrind --leak-check=full ./main
```

Please ensure, at all times, that your code *correctly* de-allocate *all* the memory that was allocated.

6 Testing

As testing is a vital skill that all software developers need to know and be able to perform daily, 10% of the assignment marks will be allocated to your testing skills. To do this, you will need to submit a testing main (inside the main.cpp file) that will be used to test an Instructor Provided solution. You may add any helper functions to the main.cpp file to aid your testing. In order to determine the coverage of your testing the gcov [4] tool, specifically the following version *gcov (Debian 8.3.0-6) 8.3.0*, will be used. The following set of commands will be used to run gcov:

```
g++ --coverage *.cpp -o main
./main
gcov -f -m -r -j Particle PSO Vector
```

This will generate output which we will use to determine your testing coverage. The following coverage ratio will be used:

$$\frac{\text{number of lines executed}}{\text{number of source code lines}}$$

We will scale this ration based on class size.

The mark you will receive for the testing coverage task is determined using Table 2:

[4] For more information on gcov please see `https://gcc.gnu.org/onlinedocs/gcc/Gcov.html`

Coverage ratio range	% of testing mark
0%-5%	0%
5%-20%	20%
20%-40%	40%
40%-60%	60%
60%-80%	80%
80%-100%	100%

Table 2: Mark assignment for testing

Note the top boundary for the Coverage ratio range is not inclusive except for 100%. Also, note that only the function stipulated in this specification will be considered to determine your mark. Remember that your main will be testing the Instructor Provided code and as such it can only be assumed that the functions outlined in this specification are defined and implemented.

7 Implementation Details

- You must implement the functions in the header files exactly as stipulated in this specification. Failure to do so will result in compilation errors on FitchFork.
- You may only use **c++98**.
- You may only utilize the specified libraries. Failure to do so will result in compilation errors on FitchFork.
- Do not include using namespace std in any of the files.
- You may only use the following libraries:
  - String
  - CMath
  - IOStream
  - FStream
  - StringStream
- You are supplied with a **trivial** main demonstrating the basic functionality of the assignment.

8 Upload Checklist

The following c++ files should be in a zip archive named uXXXXXXXX.zip where XXXXXXXX is your student number:

- Vector.h
- Vector.cpp
- Function.h
- Function.cpp
- Particle.h
- Particle.cpp
- PSO.h
- PSO.cpp
- main.cpp
- testingFramework.h and testingFramework.cpp if you used these files

The files should be in the root directory of your zip file. In other words, when you open your zip file you should immediately see your files. They should not be inside another folder.

9 Submission

You need to submit your source files on the FitchFork website (https://ff.cs.up.ac.za/). All methods need to be implemented (or at least stubbed) before submission. Your code should be able to be compiled with the following command:

```
g++ *.cpp -o main
```

and run with the following command:

```
./main
```

Remember your `h` file will be overwritten, so ensure you do not alter the provided `h` files.
You have 5 submissions and your best mark will be your final mark. Upload your archive to the Assignment 1 slot on the FitchFork website. Submit your work before the deadline. **No late submissions will be accepted!**

References

[1] Ahmed G Gad. Particle swarm optimization algorithm and its applications: a systematic review. *Archives of computational methods in engineering*, 29(5):2531–2561, 2022.

[2] James Kennedy and Russell Eberhart. Particle swarm optimization. In *Proceedings of ICNN'95-international conference on neural networks*, volume 4, pages 1942–1948. ieee, 1995.

[3] Anna S Rakitianskaia and Andries P Engelbrecht. Training feedforward neural networks with dynamic particle swarm optimisation. *Swarm Intelligence*, 6:233–270, 2012.

10 Example

Given that a particle has the following information:

- A position vector of $\begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix}$
- A velocity vector of $\begin{bmatrix} 0.2 \\ 0.1 \end{bmatrix}$
- A ω value of 0.4
- A c_1 value of 0.3
- A c_2 value of 0.6
- A global best ($gBest$) with a position of $\begin{bmatrix} 0.8 \\ 0.4 \end{bmatrix}$

10.1 Example without pBest

If for example, the particle has no $pBest$ the velocity should be calculated as follows but set c_1 as 0:

$$\mathbf{v}' = \omega\mathbf{v} + c_1(\mathbf{x}_{best} - \mathbf{x}) + c_2(\mathbf{x}_{global\_best} - \mathbf{x})$$

$$\mathbf{v}' = \omega\mathbf{v} + 0(\mathbf{x}_{best} - \mathbf{x}) + c_2(\mathbf{x}_{global\_best} - \mathbf{x})$$

$$\mathbf{v}' = \omega\mathbf{v} + c_2(\mathbf{x}_{global\_best} - \mathbf{x}) \tag{3}$$

Now swapping in the values:

$$\mathbf{v}' = 0.4\left(\begin{bmatrix} 0.2 \\ 0.1 \end{bmatrix}\right) + 0.6\left(\begin{bmatrix} 0.8 \\ 0.4 \end{bmatrix} - \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix}\right)$$

We can thus calculate each component as follows:

$$v'[0] = 0.4(0.2) + 0.6(0.8 - 0.5)$$

$$v'[0] = 0.26$$

$$v'[1] = 0.4(0.1) + 0.6(0.4 - 0.5)$$

$$v'[1] = -0.02$$

Thus a final new velocity vector of:

$$\begin{bmatrix} 0.26 \\ -0.02 \end{bmatrix}$$

10.2 Example with pBest

If for example, the particle has a particle best (pBest) $\left(\begin{bmatrix}0.1\\0.1\end{bmatrix}\right)$ the velocity should be calculated as follows:

$$\mathbf{v}' = \omega\mathbf{v} + c_1(\mathbf{x}_{best} - \mathbf{x}) + c_2(\mathbf{x}_{global\_best} - \mathbf{x})$$

$$\mathbf{v}' = 0.4\left(\begin{bmatrix}0.2\\0.1\end{bmatrix}\right) + 0.3\left(\begin{bmatrix}0.1\\0.1\end{bmatrix} - \begin{bmatrix}0.5\\0.5\end{bmatrix}\right) + 0.6\left(\begin{bmatrix}0.8\\0.4\end{bmatrix} - \begin{bmatrix}0.5\\0.5\end{bmatrix}\right)$$

Now calculate the components:

$$v'[0] = 0.4(0.2) + 0.3(0.1 - 0.5) + 0.6(0.8 - 0.5)$$

$$v'[0] = 0.14$$

$$v'[1] = 0.4(0.1) + 0.3(0.1 - 0.5) + 0.6(0.4 - 0.5)$$

$$v'[1] = -0.1$$

Thus this gives us the final new velocity of:

$$\begin{bmatrix}0.14\\-0.14\end{bmatrix}$$

10.3 Example of position

In order to calculate the new position of the particle use:

$$\mathbf{x}' = \mathbf{v}' + \mathbf{x}$$

Assume that our new velocity vector is $\begin{bmatrix}0.14\\-0.1\end{bmatrix}$. We can calculate the new position vector as follows:

$$\mathbf{x}' = \begin{bmatrix}0.14\\-0.14\end{bmatrix} + \begin{bmatrix}0.5\\0.5\end{bmatrix}$$

Using the same approach as used above we obtain the following new position vector:

$$\begin{bmatrix}0.64\\0.36\end{bmatrix}$$

11 Accessibility

This section describes all figures that are used in this document in greater detail.

Figure 1 shows three square blocks. In the middle of each block, there is a purple colour. This purple colour fades from purple, to dark green, to light green and then yellow in a circular pattern. Scattered around the blocks are a series of black dots which represent the particles. Over the three blocks, the particles slowly cluster onto the centre of the block.

Figure 2 shows the UML of the following four classes:

- Function
- Vector
- Particle
- PSO

and a `SwarmGeneratorProperties` struct. The member functions and variables for each class is discussed in Section 4. The following relations exists in Figure 2:

- The Particle class has a relation with the Vector class due to the *position*, *velocity* and *pBest* member variables of Particle.
- The PSO class has a relation with the the Particle class due to the *particles* member variables of PSO.