Purple 0

An Open Source Numerical Optimization C++ Library.

www.cimne.com/purple

User's Guide.

Roberto Lopez
International Center for Numerical Methods in Engineering (CIMNE)
Technical University of Catalonia (UPC)
Barcelona, Spain
E-mail: rlopez@cimne.upc.edu

June 1, 2006

Preface

`Purple` is a comprehensive class library for numerical optimization in the C++ programming language. The library includes unconstrained and constrained example objective functions, as well as local and global optimization algorithms. It also comes with extensive documentation. `Purple` is placed under the GNU Lesser General Public License.

Contents

Chapter 1

Preliminaries

1.1 The Purple namespace

Each set of definitions in the Purple library is 'wrapped' in the namespace Purple. In this way, if some other definition has an identical name, but is in a different namespace, then there is no conflict.

The using directive makes a namespace available throughout the file where it is written [2]. For the Purple namespace the following sentence can be written:

```
using namespace Purple;
```

1.2 The Vector and Matrix classes in Purple

The Purple library includes its own Vector and Matrix container classes. The Vector and Matrix classes are templates, which means that they can be applied to different types [2]. That is, we can create a Vector of int numbers, a Matrix of MyClass objects, etc.

For example, in order to construct an empty Vector of int numbers we use

```
Vector<int> vector1;
```

The following sentence constructs a Vector of 3 int numbers and sets the 0, 1 and 2 elements of the Vector to 1, 2 and 3, respectively. Note that indexing always starts at zero.

```
Vector<int> vector2(3);
vector2[0] = 1;
vector2[1] = 2;
vector2[2] = 3;
```

If we want to construct Vector of 5 int numbers and intialize all the elements 0, we can use

```
Vector<int> vector3(5, 0);
```

The following sentence copies vector3 into vector1.

```
vector1 = vector3;
```

The method getSize(void) returns the size of a Vector.

```
int sizeOfVector1 = vector1.getSize();
```

In order to construct an empty `Matrix` of `MyClass` objects with 2 rows and 3 columns we use

```
Matrix<MyClass> matrix1(2,3);
```

The methods `getNumberOfRows(void)` and `getNumberOfColumns(void)` return the numbers of rows and columns in a `Matrix`, respectively.

```
int numberOfRowsInMatrix1 = matrix1.getNumberOfRows();
int numberOfColumnsInMatrix1 = matrix1.getNumberOfColumns();
```

In order to put all these ideas together, we list below the source code of a sample application which makes use of the `Vector` and `Matrix` container classes.

```
// Vector and Matrix Application

#include <iostream>

// Utilities includes

#include "../Library/Utilities/Vector.h"
#include "../Library/Utilities/Matrix.h"

using namespace Purple;

int main(void)
{
   std::cout << std::endl
             << "Purple Neural Network. Vector and Matrix Application."
             << std::endl;

   // Construct an empty vector of integers

   Vector<int> vector1;

   // Construct a vector of integers with 3 elements

   Vector<int> vector2(3);

    // Initialize all the elements of vector2 to 1

   vector2[0] = 1;
   vector2[1] = 1;
   vector2[2] = 1;

   // Construct a vector of integers with 5 elements and initialize them to 0

   Vector<int> vector3(5, 1);

   // Construct a vector which is a copy of vector2

   Vector<int> vector4 = vector2;

   // Get size of vector4
```

```
int size4 = vector4.getSize();

// Print the elements of vector4 to the screen

std::cout << std::endl
          << "Vector 4:" << std::endl;


for(int i = 0; i < size4; i++)
{
   std::cout << vector4[i] << " ";
}

std::cout << std::endl;


// Construct an empty matrix of double precision numbers

Matrix<double> matrix1;

// Construct a matrix of double precision numbers with 2 rows and 3 columns

Matrix<double> matrix2(2,3);

// Initialize all the elements of matrix2 to 1

matrix2[0][0] = 1.0;
matrix2[0][1] = 1.0;
matrix2[0][2] = 1.0;
matrix2[1][0] = 1.0;
matrix2[1][1] = 1.0;
matrix2[1][2] = 1.0;

// Construct a matrix of double precision numbers with 4 rows and 2 columns,
// and initialize all the elements to 1

Matrix<double> matrix3(4,2,1.0);

// Construct a matrix which is a copy of matrix2

Matrix<double> matrix4 = matrix2;

// Get number of rows and columns of matrix4

int numberOfRows4 = matrix4.getNumberOfRows();
int numberOfColumns4 = matrix4.getNumberOfColumns();

// Print the elements of matrix4 to the screen

std::cout << std::endl
          << "Matrix 4:" << std::endl;

for(int i = 0; i < numberOfRows4; i++)
```

```
   {
      for(int j = 0; j < numberOfColumns4; j++)
      {
         std::cout << matrix4[i][j] << " ";
      }

      std::cout << std::endl;
   }

   std::cout << std::endl;

   return 0;
}
```

Chapter 2

The Objective Function

The function optimization problem is formulated in terms of finding an extremal argument of some objective function. In this way, the objective function defines the optimization problem itself.

2.1 The unconstrained function optimization problem

The simplest function optimization problems are those in which no constraints are posed on the solution. The general unconstrained function optimization problem can be formulated as follows:

Problem 1 (Unconstrained function optimization problem) *Let $X \subseteq \mathbf{R}^n$ be a real vector space. Find a vector $\mathbf{x}^* \in X$ for which the function*

$$\begin{aligned} f : X &\rightarrow \mathbf{R} \\ \mathbf{x} &\mapsto f(\mathbf{x}) \end{aligned}$$

takes on a minimum or a maximum value.

The function $f(\mathbf{x})$ is called the objective function. The domain of the objective function for a function optimization problem is a subset X of $\mathbf{R}^n$, and the image of that function is the set $\mathbf{R}$. The integer n is known as the *number of variables* in the objective function.

The vector at which the objective function takes on a minimum or maximum value is called the minimal or the maximal argument of that function, respectively. The tasks of minimization and maximization are trivially related to each other, since maximization of $f(\mathbf{x})$ is equivalent to minimization of $-f(\mathbf{x})$, and vice versa. Therefore, without loss of generality, we will assume function minimization problems.

On the other hand, a *minimum* can be either a *global minimum*, the smallest value of the function over its entire range, or a *local minimum*, the smallest value of the function within some local neighborhood. Functions with a single minimum are said to be *unimodal*, while functions with many minima are said to be *multimodal*.

2.2 The constrained function optimization problem

A function optimization problem can be specified by a set of constraints, which are equalities or inequalities that the solution must satisfy. Such constraints are expressed as functions. Thus, the constrained function optimization problem can be formulated as follows:

Problem 2 (Constrained function optimization problem) *Let $X \subseteq \mathbf{R}^n$ be a real vector space. Find a vector $\mathbf{x}^* \in X$ such that the functions*

$$\begin{aligned} c_i : X &\rightarrow \mathbf{R} \\ \mathbf{x} &\mapsto c_i(\mathbf{x}) \end{aligned}$$

hold $c_i(\mathbf{x}^*) = 0$, for $i = 1, \ldots, l$, and for which the function

$$\begin{aligned} f : X &\rightarrow \mathbf{R} \\ \mathbf{x} &\mapsto f(\mathbf{x}) \end{aligned}$$

takes on a minimum value.

In other words, the constrained function optimization problem consists of finding an argument which makes all the constraints to be satisfied and the objective function to be an extremum. The integer l is known as the *number of constraints* in the function optimization problem.

A common approach when solving a constrained function optimization problem is to reduce it into an unconstrained problem. This can be done by adding a penalty term to the objective function for each of the constrains in the original problem. Adding a penalty term gives a large positive or negative value to the objective function when an infeasibility due to a constrain is encountered.

For the minimization case, the general constrained function optimization problem can be reformulated as follows:

Problem 3 (Reduced constrained function optimization problem) *Let $X \subseteq \mathbf{R}^n$ be a real vector space, and let $\rho_i > 0$, for $i = 1, \ldots, l$, be real numbers. Find a vector $\mathbf{x}^* \in X$ for which the function*

$$\begin{aligned} \bar{f} : X &\rightarrow \mathbf{R} \\ \mathbf{x} &\mapsto \bar{f}(\mathbf{x}), \end{aligned}$$

defined by

$$\bar{f}(\mathbf{x}) = f(\mathbf{x}) + \sum_{i=1}^{l} \rho_i \left(c_i(\mathbf{x})\right)^2,$$

takes on a minimum value.

The parameters ρ_i, $i = 1, \ldots, l$, are called the *penalty term ratios*, being l the number of constraints. Note that, while the squared norm of the penalty term is the metric most used, any other suitable metric can be used. For large values of the ratios ρ_i, $i = 1, \ldots, l$, it is clear that the solution $\mathbf{x}^*$ of Problem 3 will be in in a region where $c_i(\mathbf{x})$, $i = 1, \ldots, l$, are small. Thus, for increasing values of ρ_i, $i = 1, \ldots, l$, it is expected that the the solution $\mathbf{x}^*$ of Problem 3 will approach the constraints and, subject to being close, will minimize the objective function $f(\mathbf{x})$. Ideally then, as $\rho_i \rightarrow \infty$, $i = 1, \ldots, l$, the solution of Problem 3 will converge to the solution of Problem 2 [5].

2.3 The objective function gradient vector

Many optimization algorithms use the gradient vector of the objective function to search for the minimal argument. The gradient vector of the objective function is written:

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n} \right). \tag{2.1}$$

While for some objective functions the gradient vector can be evaluated analytically, there are many applications when that is not possible, and the objective function gradient vector needs to be computed numerically. This can be done by perturbing each variable in turn, and approximating the derivatives by using the *central differences* method

$$\frac{\partial f}{\partial x_i} = \frac{f(x_i + h) - f(x_i - h)}{2h} + \mathcal{O}(h^2), \tag{2.2}$$

for $i = 1, \ldots, n$ and for some small numerical value of h.

2.4 The objective function Hessian matrix

There are some optimization algorithms which also make use of the Hessian matrix of the objective function to search for the minimal argument. The Hessian matrix of the objective function is written:

$$\mathbf{H}f = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{pmatrix} \tag{2.3}$$

As with the gradient vector, there are many applications when analytical evaluation of the Hessian is not possible, and it must be computed numerically. This can be done by perturbing each argument element in turn, and approximating the derivatives by using the *central differences* method

$$\begin{aligned} \frac{\partial^2 f}{\partial x_i \partial x_j} &= \frac{f(x_i + \epsilon, x_j + \epsilon)}{4\epsilon^2} - \frac{f(x_i + \epsilon, x_j - \epsilon)}{4\epsilon^2} \\ &- \frac{f(x_i - \epsilon, x_j + \epsilon)}{4\epsilon^2} + \frac{f(x_i - \epsilon, x_j - \epsilon)}{4\epsilon^2} + \mathcal{O}(h^2). \end{aligned} \tag{2.4}$$

2.5 Some example function optimization problems

This section describes a number of test functions for optimization. These functions are taken from the literature on both local and global optimization.

2.5.1 The De Jong's function optimization problem

One of the simplest test functions for optimization is the De Jong's function, which is an unconstrained and unimodal function. The De Jong's function optimization problem in n variables can be stated as:

Problem 4 (De Jong's function optimization problem) *Let* $X = [-5.12, 5.12]^n$ *be a real vector space. Find a vector* $\mathbf{x}^* \in X$ *for which the function*

$$\begin{array}{rcl} f : X & \rightarrow & \mathbf{R} \\ \mathbf{x} & \mapsto & f(\mathbf{x}), \end{array}$$

defined by

$$f(\mathbf{x}) = \sum_{i=1}^{n} x_i^2, \tag{2.5}$$

takes on a minimum value.

The De Jong's function has a unique minimal argument $\mathbf{x}^* = (0, \ldots, 0)$, which gives a minimum value $f(\mathbf{x}^*) = 0$. Figure 2.1 is a plot of this function in 2 variables.

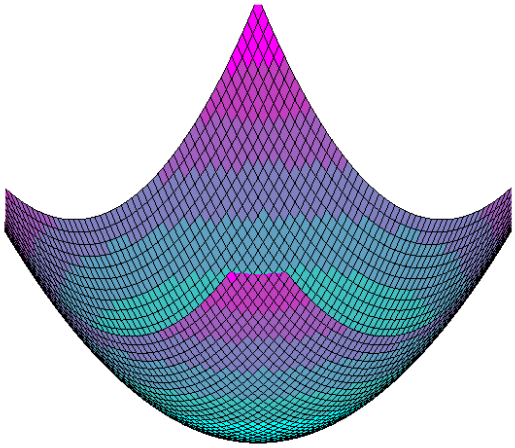

Figure 2.1: The De Jong's Function in 2 variables.

The gradient vector for the De Jong's function is given by

$$\nabla f = (2x_1, \ldots, 2x_n), \tag{2.6}$$

and the Hessian matrix by

$$Hf = \begin{pmatrix} 2 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 2 \end{pmatrix} \tag{2.7}$$

2.5.2 The Rosenbrock's function optimization problem

The Rosenbrock's function, also known as banana function, is an unconstrained and unimodal function. The optimum is inside a long, narrow, parabolic shaped flat valley. Convergence to this optimum is difficult and hence this problem has been repeatedly used in assess the performance of optimization algorithms. The Rosenbrock's function optimization problem in n variables can be stated as:

Problem 5 (Rosenbrock's function optimization problem) *Let* $X = [-2.048, 2.048]^n$ *be a real vector space. Find a vector* $\mathbf{x}^* \in X$ *for which the function*

$$\begin{aligned} f : X &\rightarrow \mathbf{R} \\ \mathbf{x} &\mapsto f(\mathbf{x}), \end{aligned}$$

defined by

$$f(\mathbf{x}) = \sum_{i=1}^{n-1} 100(x_{i+1} - x_i)^2 + (1 - x_i)^2 \tag{2.8}$$

takes on a minimum value.

The minimal argument of the Rosenbrock's function is found at $\mathbf{x}^* = (1, \ldots, 1)$. The minimum value of this function is $f(\mathbf{x}^*) = 0$. Figure 2.2 is a plot of the Rosenbrock's function in 2 variables.

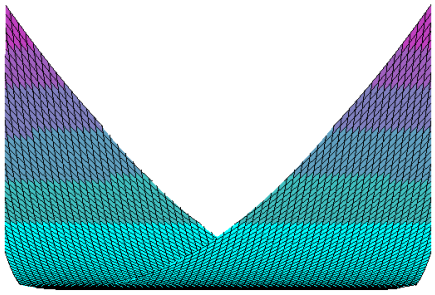

Figure 2.2: The Rosenbrock's function in 2 variables.

2.5.3 The Rastrigin's function optimization problem

The Rastrigin's function is based on the De Jong's function with the addition of cosine modulation to produce many local minima. As a result, this function is highly multimodal. However, the location of the minima are regularly distributed. The Rastrigin's function optimization problem in n variables can be stated as:

Problem 6 (Rastrigin's function optimization problem) *Let* $X = [-5.12, 5.12]^n$ *be a real vector space. Find a vector* $\mathbf{x}^* \in X$ *for which the function*

$$\begin{aligned} f : X &\rightarrow \mathbf{R} \\ \mathbf{x} &\mapsto f(\mathbf{x}), \end{aligned}$$

defined by

$$f(\mathbf{x}) = 10n + \sum_{i=1}^{n} (x_i^2 - 10\cos(2\pi x_i)) \tag{2.9}$$

takes on a minimum value.

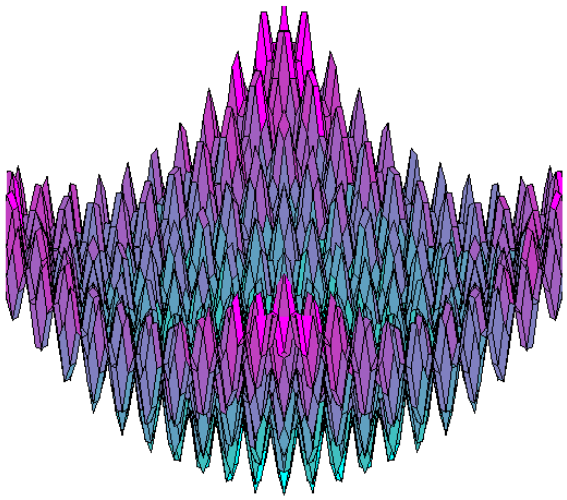

Figure 2.3: The Rastrigin's function in 2 variables.

The global minimum of the Rastrigin's Function is at $x_i^* = 0$. At this minimal argument the value of the function is $f(\mathbf{x}) = 0$. Figure 2.3 is a plot of the Rastrigin's function in 2 variables.

The gradient vector for the Rastrigin's function is given by

$$\nabla f = (2x_1 + 10\sin(2\pi x_1)2\pi, \ldots, 2x_n + 10\sin(2\pi x_n)2\pi), \tag{2.10}$$

and the Hessian matrix by

$$Hf = \begin{pmatrix} 2 + 10\cos(2\pi x_1)4\pi^2 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 2 + 10\cos(2\pi x_n)4\pi^2 \end{pmatrix} \tag{2.11}$$

2.5.4 The Plane-Cylinder function optimization problem

The problem in this example is to find the minimum point on the plane $x_1 + x_2 = 1$ which also lies in the cylinder $x_1^2 + x_2^2 = 1$. Figure 2.4 is a graphical representation of this function optimization problem.

This constrained function optimization problem can be stated as:

Problem 7 (Plane-cylinder function optimization problem) *Let $X = [-1, 1]^2$ be a real vector space. Find a vector $\mathbf{x}^* \in X$ such that the function*

$$\begin{aligned} c : X &\to \mathbf{R} \\ \mathbf{x} &\mapsto c(\mathbf{x}), \end{aligned}$$

defined by

$$c(\mathbf{x}) = x_1^2 + x_2^2 - 1, \tag{2.12}$$

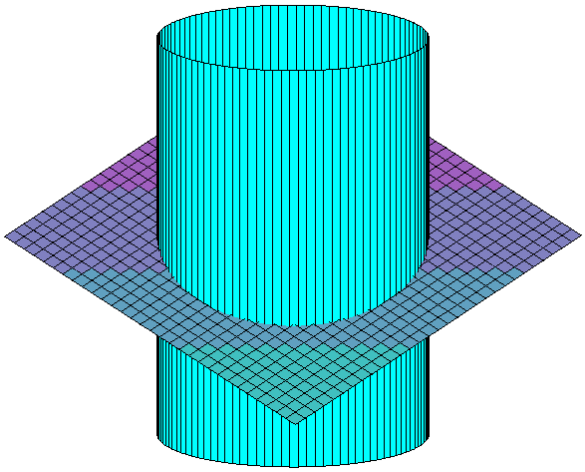

Figure 2.4: The plane-cylinder function optimization problem.

holds $c(\mathbf{x}^*) \leq 0$ *and for which the function*

$$\begin{aligned} f : X &\rightarrow \mathbf{R} \\ \mathbf{x} &\mapsto c(\mathbf{x}), \end{aligned}$$

defined by

$$f(\mathbf{x}) = x_1 + x_2 - 1, \tag{2.13}$$

takes on a minimum value.

This constrained problem can be reduced to an unconstrained problem by the use of a penalty function:

Problem 8 (Reduced plane-cylinder function optimization problem) *Let* $X \subseteq \mathbf{R}^2$ *be a real vector space, and let* $\rho\mathbf{R}^+$ *be a positive real number. Find a vector* $\mathbf{x}^* \in X$ *for which the function*

$$\begin{aligned} \bar{f} : X &\rightarrow \mathbf{R} \\ \mathbf{x} &\mapsto \bar{f}(\mathbf{x}), \end{aligned}$$

defined by

$$\bar{f}(\mathbf{x}) = x_1 + x_2 - 1 + \rho\left(x_1^2 + x_2^2 - 1\right)^2,$$

takes on a minimum value.

2.6 ObjectiveFunction classes in Purple

`Purple` includes the classes `DeJongFunction`, `RosenbrockFunction`, `RastriginFunction`, and `PlaneCylinder` to represent the concepts of the De Jong's, Rosenbrock's, Rastrigin's and Plane-Cylinder objective functions, respectively.

2.6.1 The DeJongFunction class

To construct the default De Jong's function object with 2 variables we can use the following sentence:

```
DeJongFunction deJongFunction;
```

The `getDomain(void)` method returns the domain of the objective function,

```
Matrix<double> domain = deJongFunction.getDomain();
```

The `getEvaluation(Vector<double>)` method returns the evaluation of the objective function for a given argument,

```
Vector<double> argument(2, 0.0);

argument[0] = 1.2;
argument[1] = -3.6;

double evaluation = deJongFunction.getEvaluation(argument);
```

On the other hand, the method `getGradient(Vector<double>)` returns the objective function gradient vector for a given argument,

```
Vector<double> gradient = deJongFunction.getGradient(argument);
```

Similarly, the `getHessian(Vector<double>)` method returns the objective function Hessian matrix for a given argument,

```
Matrix<double> hessian = deJongFunction.getHessian(argument);
```

2.6.2 The RosenbrockFunction class

To construct the default Rosenbrock's function object with 2 variables object we use

```
RosenbrockFunction rosenbrockFunction;
```

The `setNumberOfVariables(int)` method sets a new number of variables in an objective function. For example, to set 3 variables in the Rosenbrock's function we can write

```
int numberOfVariables = 3;

rosenbrockFunction.setNumberOfVariables(numberOfVariables);
```

The `setLowerBound(Vector<double>)` and `setUpperBound(Vector<double>)` methods set new lower and upper bounds in the the domain of the objective function,

```
Vector<double> lowerBound(numberOfVariables, -5.12);

Vector<double> upperBound(numberOfVariables, 5.12);

rosenbrockFunction.setLowerBound(lowerBound);

rosenbrockFunction.setUpperBound(upperBound);
```

The `getEvaluation(Vector<double>)` method returns the evaluation of the objective function for a given argument,

```
Vector<double> argument(numberOfVariables, 1.2);

double evaluation = rosenbrockFunction.getEvaluation(argument);
```

On the other hand, the method `getGradient(Vector<double>)` returns the objective function gradient vector for a given argument,

```
Vector<double> gradient = rosenbrockFunction.getGradient(argument);
```

Similarly, the `getHessian(Vector<double>)` method returns the objective function Hessian matrix for a given argument,

```
Matrix<double>  = rosenbrockFunction.getHessian(argument);
```

2.6.3 The RastriginFunction class

To construct the default Rastrigin's function object with 2 variables we can use the following sentence:

```
RastriginFunction rastriginFunction;
```

The `getEvaluation(Vector<double>)` method returns the evaluation of the objective function for a given argument,

```
int numberOfVariables = rastriginFunction.getNumberOfVariables();

Vector<double> argument(numberOfVariables, 1.2);

double evaluation = rastriginFunction.getEvaluation(argument);
```

On the other hand, the method `getGradient(Vector<double>)` returns the objective function gradient vector for a given argument,

```
Vector<double> gradient = rastriginFunction.getGradient(argument);
```

Similarly, the `getHessian(Vector<double>)` method returns the objective function Hessian matrix for a given argument,

```
Matrix<double>  = rastriginFunction.getHessian(argument);
```

2.6.4 The PlaneCylinder class

To construct the default plane-cylinder objective function object we can use the following sentence:

```
PlaneCylinder planeCylinder;
```

The `getError(Vector<double>)` method returns the squared error made in the constraint by a given argument,

```
Vector<double> argument(2, 1.5);

double error = planeCylinder.getError();
```

The `getEvaluation(Vector<double>)` method returns the evaluation of the objective function for a given argument,

```
Vector<double> argument(2, 1.2);

double evaluation = planeCylinder.getEvaluation(argument);
```

On the other hand, the method `getGradient(Vector<double>)` returns the objective function gradient vector for a given argument,

```
Vector<double> gradient = planeCylinder.getGradient(argument);
```

Similarly, the `getHessian(Vector<double>)` method returns the objective function Hessian matrix for a given argument,

```
Matrix<double>  = planeCylinder.getHessian(argument);
```

Chapter 3

The Optimization Algorithm

The procedure used to find the minimal argument of an objective function is called the optimization algorithm. There are many different algorithms for numerical optimization of functions. Some of the most widely used are the conjugate gradient or the evolutionary algorithm.

3.1 The function optimization process

The objective function is, in general, a non linear function. As a consequence, it is not possible to find closed optimization algorithms for the minimal argument. In this way, to find the minimal argument of the objective function we start with an initial estimation of the optimal argument $\mathbf{x}^{(0)}$ (often chosen at random) and we generate a sequence of arguments

$$\mathbf{x}^{(0)}, \mathbf{x}^{(1)}, \ldots,$$

so that the objective function f is reduced at each iteration of the optimization algorithm, that is

$$f(\mathbf{x}^{(0)}) \geq f(\mathbf{x}^{(1)}) \geq \ldots.$$

The optimization algorithm stops when a specified condition is satisfied. Some stopping criteria commonly used are [1]:

1. A maximum number of iterations is reached.
2. A maximum amount of computing time has been exceeded.
3. Evaluation has been minimized to a goal.
4. The gradient norm of the objective function falls below a goal value.

Figure 3.1 is a state diagram of this iterative procedure, showing states and transitions in the optimization process.

The optimization process determined by the way in which the adjustment of the optimal argument takes place. There are many different optimization algorithms, which have a variety of different computation and storage requirements. Moreover, there is not a optimization algorithm best suited to all problems [10].

Optimization algorithms might require information from the objective function only, the gradient vector of the objective function or the Hessian matrix of the objective function [7]. These methods, in turn, can perform either global or local optimization.

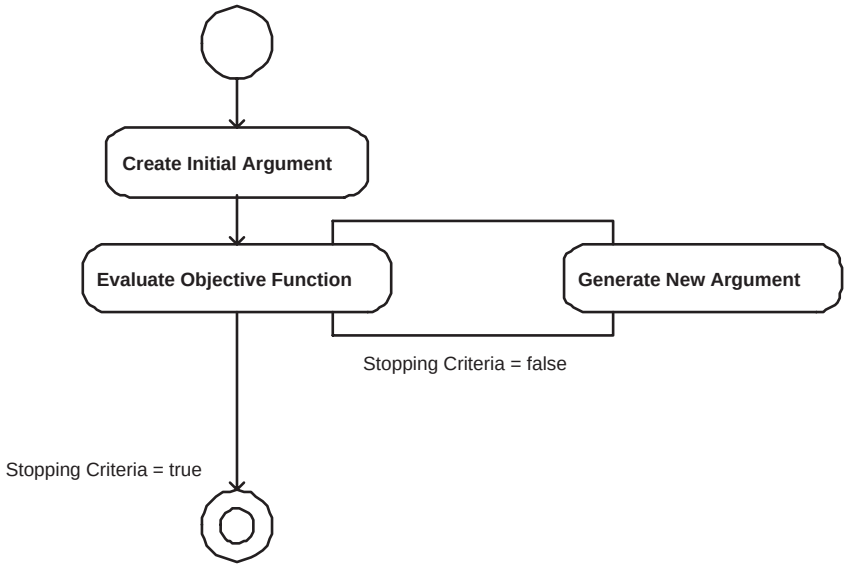

Figure 3.1: State diagram for the numerical function optimization process.

Zero-order optimization algorithms make use of the objective function only. The most significant zero-order optimization algorithms are stochastic, which involve randomness in the optimization process. Examples of these are *Random Search* or the *Evolutionary Algorithm*, which are global optimization methods [4] [3].

First-order optimization algorithms use the objective function and its gradient vector. Examples of these are the *Gradient Descent Method*, the *Conjugate Gradient Method*. Gradient descent and conjugate gradient are local optimization methods [5].

Second-order optimization algorithms make use of the objective function, its gradient vector and its hessian matrix. An example of a second-order method is the *Newton's method* which is a local optimization method [5].

3.2 Line search algorithms

Line search methods are function optimization algorithms in one variable. They are also used by different optimization algorithms in many variables. In this section, we describe two different line searches widely used, the golden section algorithm and the Brent's method.

The *golden section* method brackets a minimum until the distance between the two outer points in the bracket is less than a defined *tolerance* [7].

The *Brent's method* performs a parabolic interpolation until the distance between the two outer points defining the parabola is less than a *tolerance* [7].

3.3 The gradient descent method

One of the simplest optimization algorithms is gradient descent, sometimes also known as steepest descent. Gradient descent is a local method which requires information from the objective function, and the gradient vector, but not from the Hessian matrix.

The method starts at an initial argument $\mathbf{x}^{(0)}$ and, until a stopping criterium is satisfied, moves from $\mathbf{x}^{(\tau)}$ to $\mathbf{x}^{(\tau+1)}$ along the line extending from $\mathbf{x}^{(\tau)}$ in the search direction

$-\nabla(f(\mathbf{x}^{(\tau)})$, the local downhill gradient . In the gradient descent algorithm, the step size is adjusted at each iteration. A search is made along the gradient descent direction to determine the optimal step size, which minimizes the objective function along that line.

Therefore, starting from an initial argument $\mathbf{x}^{(0)}$, the gradient descent method takes the form of iterating

$$\mathbf{x}^{(\tau+1)} = \mathbf{x}^{(\tau)} - \lambda^{(\tau)} \nabla f(\mathbf{x}^{(\tau)}), \tag{3.1}$$

for $\tau = 0, 1, ...$, and where $\lambda^{(\tau)}$ is called the *optimal step size*. A search is made along the negative gradient direction to determine the optimal step size, which minimizes the objective function along that line.

The value of f will decrease at each successive step, eventually reaching a vector of free parameters $\mathbf{x}^*$ at which the necessary *local minimum condition*

$$\nabla f(\mathbf{x}^*) = 0 \tag{3.2}$$

is satisfied.

This method has the severe drawback of requiring many iterations for functions which have long, narrow valley structures. In such cases, a conjugate gradient method is preferable. See [5] for a detailed discussion of the gradient descent method.

3.4 The conjugate gradient method

The local downhill gradient is the direction in which the objective function decreases most rapidly. Nevertheless, this does not necessarily produce the fastest convergence. In the conjugate gradient algorithm search is performed along conjugate directions, which produces generally faster convergence than steepest descent directions [1]. Conjugate gradient methods have proved to be very effective in dealing with general objective functions [5].

In the conjugate gradient algorithm that we discuss now, the step size is adjusted at each iteration. A search is made along the conjugate gradient direction to determine the optimal step size, which minimizes the objective function along that line.

Let denote $\mathbf{g} \equiv \nabla f$ and $\mathbf{h}$ the *search direction vector*. Then, starting with an initial free parameter vector $\mathbf{x}^{(0)}$ and an initial search direction vector $\mathbf{h}^{(0)} = -\mathbf{g}^{(0)}$, the conjugate gradient method constructs a sequence of search direction vectors from the recurrence

$$\mathbf{x}^{(\tau+1)} = \mathbf{x}^{(\tau)} + \lambda^{(\tau)} \mathbf{h}^{(\tau)}, \tag{3.3}$$

for $\tau = 0, 1, ...$, and where $\lambda^{(\tau)}$ is called the *optimal step size*. A search is made along the conjugate gradient direction to determine the optimal step size, which minimizes the objective function along that line.

The value of f will decrease at each successive step, eventually reaching a vector of free parameters $\mathbf{x}^*$ at which the necessary *local minimum condition*

$$\nabla f(\mathbf{x}^*) = 0 \tag{3.4}$$

is satisfied.

The various versions of conjugate gradient are distinguished by the manner in which the parameter $\gamma^{(\tau)}$ is constructed. For the Fletcher-Reeves update the procedure is

$$\gamma_{F-R}^{(\tau+1)} = \frac{\mathbf{g}^{(\tau+1)} \mathbf{g}^{(\tau+1)}}{\mathbf{g}^{(\tau)} \mathbf{g}^{(\tau)}}, \tag{3.5}$$

where $\gamma_{F-R}^{(\tau+1)}$ is called the *Fletcher-Reeves parameter* [1].

For the Polak-Ribiere update the procedure is

$$\gamma_{P-R}^{(\tau+1)} = \frac{(\mathbf{g}^{(\tau+1)} - \mathbf{g}^{(\tau)})\mathbf{g}^{(\tau+1)}}{\mathbf{g}^{(\tau)}\mathbf{g}^{(\tau)}}, \tag{3.6}$$

where $\gamma_{P-R}^{(\tau+1)}$ is called the *Polak-Ribiere parameter* [1].

For all conjugate gradient algorithms, the search direction is periodically reset to the negative of the gradient. The standard reset point occurs every n iterations, where n is the number of variables in the objective function.

Figure 3.2 is a state diagram for the optimization process with the conjugate gradient. See [5] for a detailed discussion of the conjugate gradient method.

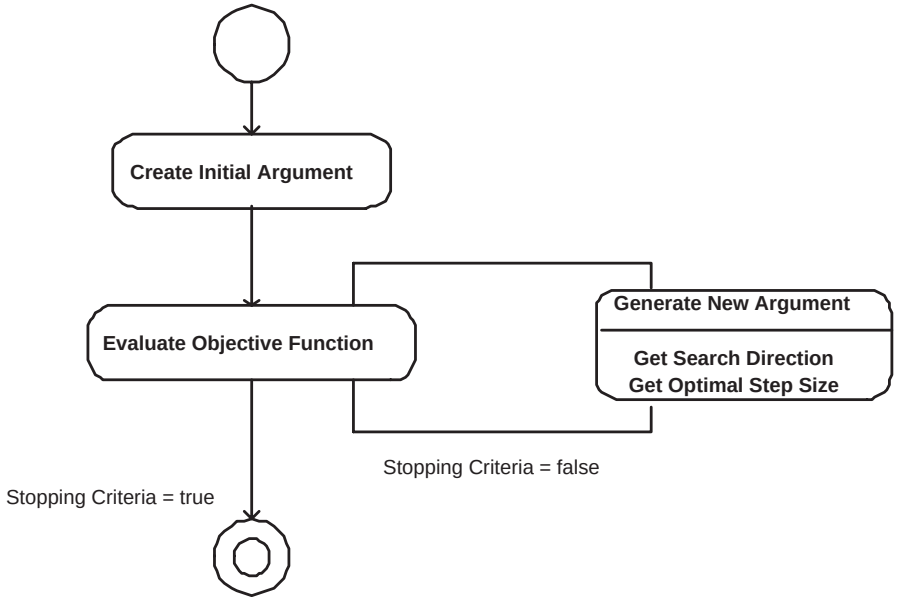

Figure 3.2: State Diagram for the optimization process with the onjugate gradient.

3.5 The Newton's method

The Newton's method is a class of optimization algorithm which makes use of the Hessian matrix of the objective function. Using a local quadratic approximation, we can obtain directly an expression for the location of the minimum of the objective function,

$$\mathbf{x}^* = \mathbf{x} - \mathbf{H}^{-1}\mathbf{g} \tag{3.7}$$

The vector $\mathbf{H}^{-1}\mathbf{g}$ is known as the Newton direction or the Newton step . Since the quadratic approximation used to obtain (3.7) is not exact, it would be necessary to apply it iteratively, with the Hessian being re-evaluated at each new search point.

There are several difficulties with such an approach, however. First, an exact evaluation of the Hessian is computationally demanding. This evaluation would be prohibitively expensive if done at each stage of an iterative algorithm. Second, the Hessian must be inverted, and

so is also computationally demanding. Third, the Netwon step in (3.7) may move towards a maximum or a saddle point rather than a minimum. This occurs if the Hessian is not positive definite, so that there exist directions of negative curvature. Thus, the objective function evaluation is not guaranteed to be reduced at each iteration. Finally, the step size predicted by (3.7) may be sufficiently large that it takes us outside the range of validity of the quadratic approximation. In this case the algorithm could become unstable.

3.6 The random search optimization algorithm

Random search is the simplest optimization algorithm possible. It is a stochastic direct search method which requires information from the objective function only, but not from the gradient vector or the hessian matrix.

The random search method simply consists of sampling a stream of arguments

$$\mathbf{x}^{(0)}, \mathbf{x}^{(1)}, \ldots \tag{3.8}$$

distibuted at random in the objective function domain, while evaluating

$$f^{(0)}(\mathbf{x}^{(0)}), f^{(1)}(\mathbf{x}^{(1)}), \ldots \tag{3.9}$$

and until a stopping criterium is satisfied.

That sequence is guaranteed to converge, with probability one, to the global optimum of the objective function. Unfortunately, convergence is extremely slow in most cases. Random search can be used to obtain a good initial guess for other more efficient methods.

3.7 The evolutionary algorithm

A global optimization algorithm is the evolutionary algorithm, also called genetic algorithm. The evolutionary algorithm is a stochastic direct search method based on the mechanics of natural genetics and biological evolution which requires information from the objective function only, but not from the gradient vector or the Hessian matrix.

The evolutionary algorithm can be used for problems that are difficult to solve with traditional optimization techniques, including problems that are not well defined or are difficult to model mathematically. It can also be used when computation of the objective function is discontinuous, highly nonlinear, stochastic, or has unreliable or undefined derivatives.

The evolutionary algorithm starts with an initial population of individuals, represented by vectors of free parameters often chosen at random

$$\mathbf{P}^{(0)} = \begin{pmatrix} x_{11}^{(0)} & \cdots & x_{1n}^{(0)} \\ \vdots & \ddots & \vdots \\ x_{s1}^{(0)} & \cdots & x_{sn}^{(0)} \end{pmatrix}$$

where $\mathbf{P}$ is called the *population matrix*. The number of individuals in the population s is called the *population size* .

The objective function is then evaluated for all the individuals

$$\mathbf{F}^{(0)} = \{f_1^{(0)}(\mathbf{x}_1^{(0)}), ..., f_s^{(0)}(\mathbf{x}_s^{(0)})\}, \tag{3.10}$$

where $\mathbf{F}$ is called the *evaluation vector*. The individual with best evaluation is then chosen. If no stopping criterium is met the generation of a new population $\mathbf{P}^{(1)}$ starts by performing fitness assignment to the old population $\mathbf{P}^{(0)}$.

In rank-based fitness assignment the population evaluation is sorted. The fitness assigned to each individual depends only on its position in the individuals rank and not on the actual evaluation value,

$$\Phi^{(0)} = \{\Phi_1^{(0)}, ..., \Phi_s^{(0)}\}, \tag{3.11}$$

where Φ is called the *fitness vector*. *Linear ranking* assigns a fitness to each individual which is linearly proportional to its rank. On the other hand, *non-linear ranking* assigns a fitness non-linearly proportional to the rank [6]. Both operators are controlled by a single parameter called *selective pressure*. Linear ranking allows values for the selective pressure in the interval $[1, 2]$. Non-linear ranking allows values for the selective pressure in the interval $[1, N-2]$, where N is the population size.

After fitness assingment has been performed, some individuals in the population are selected for recombination, according to their level of fitness

$$\mathbf{S}^{(0)} = \{S_1^{(0)}, ..., S_s^{(0)}\}, \tag{3.12}$$

where $\mathbf{S}$ is called the *selection vector*. The simplest selection operator is *roulette-wheel*. A better selection operator might be *stochastic universal sampling* [6].

Recombination produces a population matrix by combining the free parameters of the selected individuals,

$$\mathbf{P} = \begin{pmatrix} x_{11} & \dots & x_{1n} \\ \vdots & \ddots & \vdots \\ x_{s1} & \dots & x_{sn} \end{pmatrix}$$

In *line recombination* the arguments of the offspring are chosen in the hyperline joining the arguments of the parents. In *intermediate recombination* the arguments of the offspring are chosen somewhere in and around the hypercube defined by the arguments of the parents [6]. Both line and intermediate recombination are controlled by a single parameter called *recombination size*. The recombination size value must be equal or greater than 0.

Finally every offspring undergoes mutation to obtain the new generation,

$$\mathbf{P}^{(1)} = \begin{pmatrix} x_{11}^{(1)} & \dots & x_{1s}^{(1)} \\ \vdots & \ddots & \vdots \\ x_{s1}^{(1)} & \dots & x_{sn}^{(1)} \end{pmatrix}$$

The probability of mutating a free parameter is called the *mutation rate* [6]. The mutation rate allows values in the interval $[0, 1]$. On the other hand, mutation is achieved by adding or substracting a random quantity to the free parameter. *Uniform mutation* uses a random number with uniform distribution. *Normal mutation* uses a random number with normal distribution [6]. Both the uniform and normal mutation operators are controlled by a single parameter called *mutation range*, which allows values equal or greater than 0.

The whole process is repeated until a stopping criterium is satisfied. That sequence is guaranteed to converge, with probability one, to the global optimum of the objective function. Figure 3.3 is a state diagram for the optimization process with the evolutionary algorithm.

3.8 OptimizationAlgorithm classes in Purple

`Purple` includes the classes `GradientDescent`, `ConjugateGradient`, `NewtonMethod`, `RandomSearch` and `EvolutionaryAlgorithm` to represent the concepts of the different optimization algorithms described in this chapter. That classes contain:

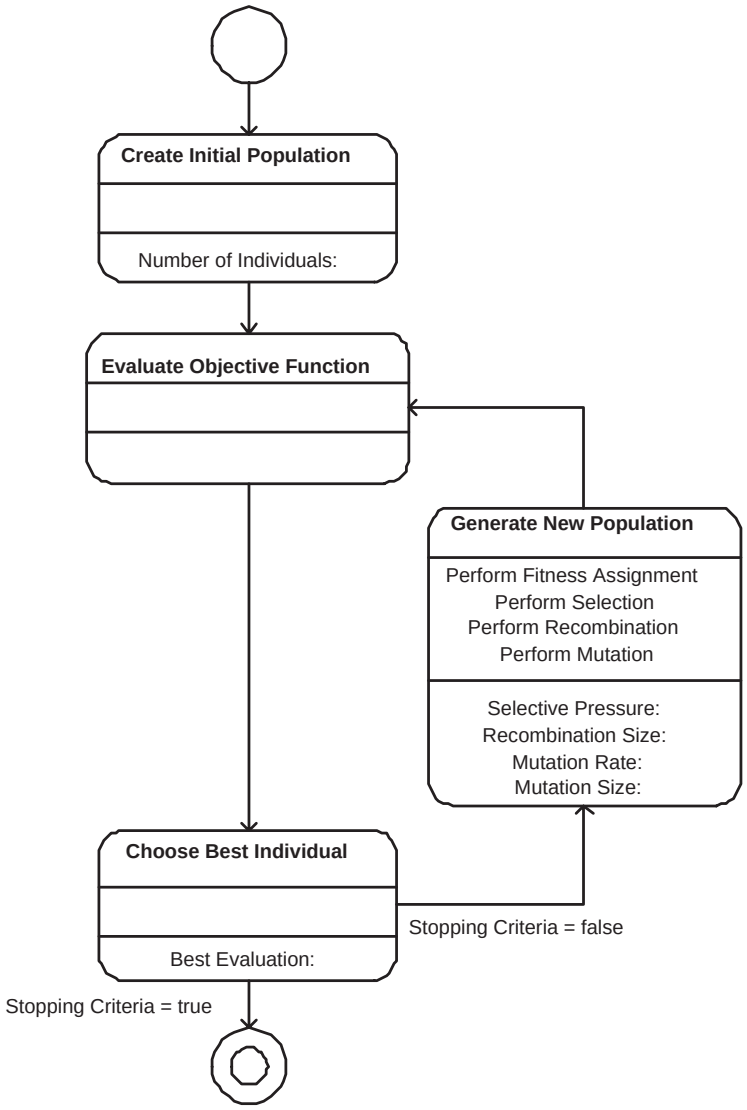

Figure 3.3: State Diagram for the training process with the evolutionary algorithm.

1. A relationship to an objective function object.
2. A set of training operators
3. A set of training parameters.
4. A set of stopping criteria.

3.8.1 The GradientDescent class

To construct a `GradientDescent` object associated, for example, to a De Jong's function object, we can use the following sentence

```
GradientDescent gradientDescent(&deJongFunction);
```

where `&deJongFunction` is a reference to a `DeJongFunction` object.

The method `setInitialArgument(Vector<double>)` sets an initial argument for the gradient descent method.

```
Vector<double> initialArgument(numberOfVariables, 1.0);

gradientDescent.setInitialArgument(initialArgument);
```

where `numberOfVariables` is the number of variables in the objective function.

The method `getMinimalArgument(void)` gets the minimal argument of an objective function according to the gradient descent method.

```
Vector<double> minimalArgument =
gradientDescent.getMinimalArgument();
```

We can save or load a gradient descent object to or from a data file, by using the methods `save(char*)` and `load(char*)`, respectively. Figure 3.4 shows the format of a gradient descent data file in `Purple`.

```
% Purple: An Open Source Numerical Optimization C++ Library.
% Gradient Descent Object.
% Any comment here.
InitialArgument:

OptimalStepSizeMethod:

FirstStepSize:

OptimalStepSizeTolerance:

EvaluationGoal:

GradientNormGoal:

MaximumTime:

MaximumNumberOfIterations:

WarningStepSize:

ShowPeriod:
```

Figure 3.4: `Purple` gradient descent file format.

3.8.2 The ConjugateGradient class

To construct a `ConjugateGradient` object associated to a Rosenbrock's function object we can write

```
ConjugateGradient conjugateGradient(&rosenbrockFunction);
```

where `&rosenbrockFunction` is a reference to a `RosenbrockFunction` object.

The method `setSearchDirectionMethod(SearchDirectionMethod)` sets the search direction method to be used by the conjugate gradient method. Similarly, the method `setOptimalStepSizeMethod(OptimalS` sets the optimal step size method to be used by the conjugate gradient method. For example, to use the Fletcher-Reeves search direction and the Brent's method for the optimal step size, we can write

```
conjugateGradient.setSearchDirectionMethod(ConjugateGradient::FletcherReeves);
conjugateGradient.setOptimalStepSizeMethod(ConjugateGradient::BrentMethod);
```

The method `getMinimalArgument(void)` searches for the minimal argument of an objective function according to the conjugate gradient method.

```
Vector<double> minimalArgument =
conjugateGradient.getMinimalArgument();
```

We can save or load a conjugate gradient object to or from a data file, by using the methods `save(char*)` and `load(char*)`, respectively. Figure 3.5 shows the format of a conjugate gradient data file in `Purple`.

```
% Purple: An Open Source Numerical Optimization C++ Library.
% Conjugate Gradient Object.
% Any comment here.
InitialArgument:

SearchDirectionMethod:

OptimalStepSizeMethod:

FirstStepSize:

OptimalStepSizeTolerance:

EvaluationGoal:

GradientNormGoal:

MaximumTime:

MaximumNumberOfIterations:

WarningStepSize:

ShowPeriod:
```

Figure 3.5: `Purple` conjugate gradient file format.

3.8.3 The NewtonMethod class

To construct a `NewtonMethod` object associated to a De Jong's function object we can write

```
NewtonMethod newtonMethod(&deJongFunction);
```

where `&deJongFunction` is a reference to a `deJongFunction` objective function object.

The method `getMinimalArgument(void)` returns the minimal argument of an objective function according to the Newton's method.

```
Vector<double> minimalArgument =
newtonMethod.getMinimalArgument();
```

We can save or load a Newton's method object to or from a data file, by using the methods `save(char*)` and `load(char*)`, respectively. Figure 3.6 shows the format of a Newton's method data file in `Purple`.

```
% Purple: An Open Source Numerical Optimization C++ Library.
% Newton Method Object.
% Any comment here.
InitialArgument:

EvaluationGoal:

GradientNormGoal:

MaximumTime:

MaximumNumberOfIterations:

ShowPeriod:
```

Figure 3.6: `Purple` Newton's method file format.

3.8.4 The RandomSearch class

To construct a `RandomSearch` object associated, for instance, to a Rosenbrock's function object, we can use the following sentence

```
RandomSearch randomSearch(&rosenbrockFunction);
```

where `&rosenbrockFunction` is a reference to a `RosenbrockFunction` object.

The method `getMinimalArgument(void)` searches for the minimal argument of an objective function according to the random search method.

```
Vector<double> minimalArgument =
randomSearch.getMinimalArgument();
```

We can save or load a random search object to or from a data file, by using the methods `save(char*)` and `load(char*)`, respectively. Figure 3.7 shows the format of a random search data file in `Purple`.

3.8.5 The EvolutionaryAlgorithm class

To construct a `EvolutionaryAlgorithm` object associated to a Rastrigin's function object we can write

```
EvolutionaryAlgorithm evolutionaryAlgorithm(&rastriginFunction);
```

```
% Purple: An Open Source Numerical Optimization C++ Library.
% Random Search Object.
% Any comment here.
EvaluationGoal:

MaximumTime:

MaximumNumberOfEvaluations:

ShowPeriod:
```

Figure 3.7: `Purple` random search file format.

where `&rastriginFunction` is a reference to a `RastriginFunction` object.

The method `getMinimalArgument(void)` returns the minimal argument of an objective function according to the evolutionary algorithm method.

```
Vector<double> minimalArgument =
evolutionaryAlgorithm.getMinimalArgument();
```

We can save or load an evolutionary algorithm object to or from a data file, by using the methods `save(char*)` and `load(char*)`, respectively. Figure 3.8 shows the format of an evolutionary algorithm data file in `Purple`.

```
% Purple: An Open Source Numerical Optimization C++ Library.
% Newton Method Object.
% Any comment here.
PopulationSize:

NumberOfVariables:

FitnessAssignmentMethod:

SelectionMethod:

RecombinationMethod:

MutationMethod:

SelectivePressure:

RecombinationSize:

MutationRate:

MutationRange:

EvaluationGoal:

MaximumTime:

MaximumNumberOfGenerations:

ShowPeriod:

Population:
```

Figure 3.8: `Purple` evolutionary algorithm file format.

Chapter 4

The Software Model of Purple

In this chapter we present the software model of Puple. The whole process is carried out in the Unified Modeling Language (UML), which provides a formal framework for the modeling of software systems.

4.1 The Unified Modeling Language

The Unified Modeling Language (UML) is a general purpose visual modeling language that is used to specify, visualize, construct, and document the artifacts of a software system [8].

UML class diagrams are the mainstay of object-oriented analysis and design. They show the classes of the system, their interrelationships and the attributes and operations of the classes.

In order to construct the software model of Purple, we follow a top-down development. This approach to the problem begins at the highest conceptual level and works down to the details. In this way, to create and evolve a conceptual class diagram for a numerical optimization tool, we iteratively model:

1. Classes.
2. Associations.
3. Derived classes.
4. Attributes and operations.

4.2 The conceptual model

In colloquial terms a concept is an idea or a thing. In object-oriented modeling concepts are represented by means of classes [9]. Therefore, a prime task is to identify the main concepts (or classes) of the problem domain. In UML class diagrams, classes are depicted as boxes [8].

Through this work, we have seen that the numerical optimization problem is characterized by an objective function and an optimization algorithm. The characterization in classes of these two concepts is as follows:

Objective function The class which represents the concept of objective function in a function optimization problem is called ObjectiveFunction.

Optimization algorithm The class representing the concept of optimization algorithm is called OptimizationAlgorithm.

Figure 4.1 depicts a starting UML class diagram for the conceptual model of Purple.

Figure 4.1: The conceptual diagram of `Purple`.

4.3 Aggregation of associations

Once identified the main concepts in the model it is necessary to aggregate the associations among them. An association is a relationship between two concepts which points some significative or interesting information [9]. In `UML` class diagrams, an association is shown as a line connecting two classes. It is also possible to assign a label to an association. The label is typically one or two words describing the association [8].

The appropriate associations in the system are next identified to be included to the `UML` class diagram of the system:

Objective function - Optimization algorithm An objective function *is optimized by* an optimization algorithm.

Figure 4.2 shows the above `UML` class diagram with these associations aggregated.

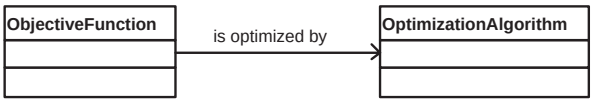

Figure 4.2: Aggregation of the associations to the conceptual diagram of `Purple`.

4.4 Aggregation of derived classes

In object-oriented programming, some classes are designed only as a parent from which subclasses may be derived, but which is not itself suitable for instantiation. This is said to be an *abstract class*, as opposed to a *concrete class*, which is suitable to be instantiated. The derived class contains all the features of the base class, but may have new features added or redefine existing features [9]. Associations between a base class and a derived class are of the kind 'is a' [8].

The next task is then to establish which classes are abstract and to derive the necessary concrete classes to be added to the system. Let us then examine the classes we have so far:

Objective function The class `ObjectiveFunction` is abstract, because it does not represent a concrete objective function for an optimization problem.

Here we have seen a number of test functions, De Jong's, Rosenbrock's, Rastrigin's and plane-cylinder. In order to solve that function optimization problems we must derive a concrete class for each of those objective functions. These are to be called `DeJongFunction`, `RosenbrockFunction`, `RastriginFunction` and `PlaneCylinderFunction`, respectively. It is necessary to derive a new objective function for any new optimization problem it is required to be solved.

Optimization algorithm The class `OptimizationAlgorithm` is abstract, because it does not represent a optimization algorithm for an objective function of an optimization problem.

An optimization algorithm which is applicable to any differentiable objective function, is gradient descent, but it has some limitations. A faster optimization algorithm, with the same properties as gradient descent, is conjugate gradient. Thus, classes representing the gradient descent and the conjugate gradient optimization algorithms are derived. These are called `GradientDescent` and `ConjugateGradient`, respectively. The main disadvantage of the gradient descent and the conjugate gradient optimization algorithms is their local nature. In order to include global optimization algorithms in the model we derive the classes `RandomSearch` and `EvolutionaryAlgorithm`. It is always possible to derive any new optimization algorithm to be added to the system.

Figure 4.3 shows the `UML` class diagram of `Purple` with all the derived classes included.

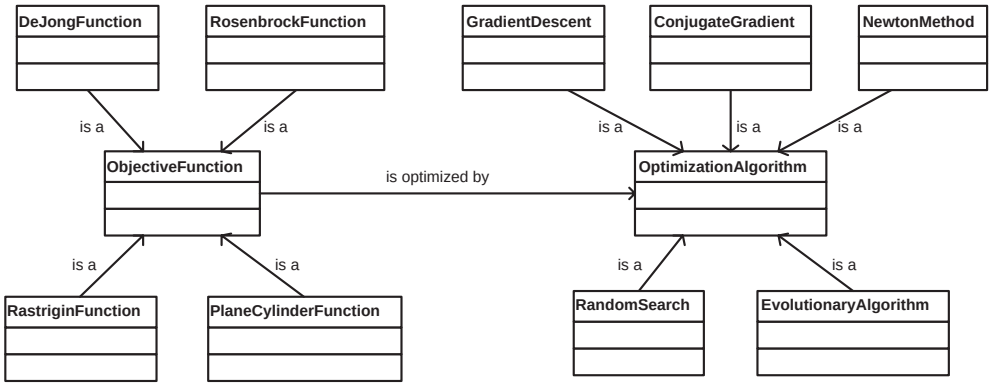

Figure 4.3: Aggregation of the derived classes to the conceptual diagram of `Purple`.

4.5 Aggregation of attributes and operations

An attribute is a named value or relationship that exists for all or some instances of a class. An operation is a procedure associated with a class [9]. In `UML` class diagrams, classes are depicted as boxes with three sections: the top one indicates the name of the class, the one in the middle lists the attributes of the class, and the bottom one lists the operations [8].

An objective function has the following attributes:

1. A number of variables.
2. A domain.

and performs the following operations:

1. Get the evaluation for a given argument.
2. Get the gradient vector for a given argument.
3. Get the Hessian matrix for a given argument.

An optimization algorithm has the following attributes:

1. A relationship to an objective function. In C++ this is implemented as a pointer to an objective function object.
2. Optimization operators.
3. Optimization parameters.
4. Stopping criteria.

and performs the following operations:

1. Get the minimal argument of an objective function.

Bibliography

[1] H. Demuth and M. Beale. *Neural Network Toolbox for Use with MATLAB. User's Gide.* The MathWorks, 2002.

[2] B. Eckel. *Thinking in C++. Second Edition.* Prentice Hall, 2000.

[3] D. B. Fogel. An introduction to simulated evolutionary optimization. *IEEE Transactions on Neural Networks*, 5(1):3–14, 1994.

[4] D.E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison Wesley, 1988.

[5] D.G. Luenberger. *Linear and Nonlinear Programming.* Addison Wesley, 1984.

[6] H. Pohlheim. Geatbx - genetic and evolutionary algorithm toolbox for use with matlab. http://www.geatbx.com, 2005.

[7] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. *Numerical Recipes in C++: The Art of Scientific Computing.* Cambridge University Press, 2002.

[8] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual.* Addison Wesley, 1999.

[9] B. Stroustrup. *The C++ Programing Language.* Addison Wesley, 2000.

[10] D. H. Wolpert and W. G. MacReady. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.

Index