# Module Interface Specification for OptLib

Fasil Cheema

March 18, 2024

# 1 Revision History

| Date | Version | Notes |
|---|---|---|
| March 17, 2024 | 1.0 | Initial Upload |

# 2 Symbols, Abbreviations and Acronyms

See SRS Documentation which can be found at https://github.com/FasilCheema/OptimizationLibrary/blob/main/docs/SRS/SRS.pdf

| symbol | description |
|---|---|
| $\mathbf{A}$ | $\mathbf{A}$ is the $n \times n$ coefficient matrix of the quadratic. |
| $\vec{\boldsymbol{b}}$ | $\vec{\boldsymbol{b}}$ is the $n \times 1$ column vector that is the coefficient of the linear term. |
| $c$ | $c$ is the constant (real number) that is the constant term in the quadratic. |
| $f$ | $f$ is the function of we are trying to minimize, defined by the above three parameters. |
| $d$ | $d$ is a natural number and is the dimension of our problem. |
| $t$ | $t$ is the natural number that is the current step. |
| $\vec{x}_t$ | $\vec{x}_t$ is the $n \times 1$ column vector that is our current solution. |
| $\vec{x}_0$ | $\vec{x}_t$ is the $n \times 1$ column vector that is our initial 'guess' of the solution. |
| $\mathbf{H}_0$ | $\mathbf{H}_0$ is the $n \times n$ initial approximation of our Hessian matrix. |
| $\mathbf{H}_t$ | $\mathbf{H}_t$ is the $n \times n$ approximation of our Hessian matrix at the $t_{th}$ step. |
| $\mathbf{B}_0$ | $\mathbf{B}_0$ is the $n \times n$ initial approximation of the inverse of our Hessian matrix. |
| $\mathbf{B}_t$ | $\mathbf{B}_t$ is the $n \times n$ approximation of the inverse of our Hessian matrix at the $t_{th}$ step. |
| $\vec{s}_t$ | $\vec{s}_t$ is the $n \times 1$ column vector that is our current search direction. |
| $\alpha_t$ | $\alpha_t$ is the real number that is our current step size. |
| $\vec{p}_t$ | $\vec{p}_t$ is the $n \times 1$ column vector that is the product of the step size and search direction. |

# Contents

# 3 Introduction

The following document details the Module Interface Specifications for OptLib, a library of optimization solvers.

Complementary documents include the System Requirement Specifications and Module Guide. The full documentation and implementation can be found at https://github.com/FasilCheema/OptimizationLibrary.

# 4 Notation

The structure of the MIS for modules comes from Hoffman and Strooper (1995), with the addition that template modules have been adapted from Ghezzi et al. (2003). The mathematical notation comes from Chapter 3 of Hoffman and Strooper (1995). For instance, the symbol := is used for a multiple assignment statement and conditional rules follow the form $(c_1 \Rightarrow r_1|c_2 \Rightarrow r_2|...|c_n \Rightarrow r_n)$.

The following table summarizes the primitive data types used by OptLib.

| Data Type | Notation | Description |
|---|---|---|
| character | char | a single symbol or digit |
| integer | $\mathbb{Z}$ | a number without a fractional component in (-$\infty$, $\infty$) |
| natural number | $\mathbb{N}$ | a number without a fractional component in [1, $\infty$) |
| real | $\mathbb{R}$ | any number in (-$\infty$, $\infty$) |
| boolean | $\mathbb{B}$ | a binary variable in {TRUE, FALSE} |

The specification of OptLib uses some derived data types: sequences, strings, and tuples. Sequences are lists filled with elements of the same data type. Strings are sequences of characters. Tuples contain a list of values, potentially of different types. In addition, OptLib uses functions, which are defined by the data types of their inputs and outputs. Local functions are described by giving their type signature followed by their specification.

# 5 Module Decomposition

The following table is taken directly from the Module Guide document for this project.

| **Level 1** | **Level 2** |
|---|---|
| Hardware-Hiding | |
| Behaviour-Hiding | Main Module |
| | Parameter Configuration |
| | Vector Math Module |
| | Gradient Calculation |
| | Step-Size Calculator |
| | Search Direction Calculator |
| Software Decision | Input Verification |
| | Output Verification |
| | Optimization Solver |

Table 1: Module Hierarchy

# 6 MIS of Main Module

## 6.1 Module

main

## 6.2 Uses

- Vector Math Module
- Input Verification Module
- Output Verification Module
- Parameter Configuration Module
- Gradient Calculation
- Step-Size Calculator
- Search Direction Calculator
- Optimization Solver
- Hardware hiding module

## 6.3 Syntax

### 6.3.1 Exported Constants

None

### 6.3.2 Exported Access Programs

| Name | In | Out | Exceptions |
|---|---|---|---|
| main | - | - | - |

## 6.4 Semantics

### 6.4.1 State Variables

None

### 6.4.2 Environment Variables

None

### 6.4.3 Assumptions

### 6.4.4 Access Routine Semantics

main():

- transition: the control module acts as the entry point to the library. It oversees the order of execution of the different modules and allows for seamless communication between different modules:
  - Select the function to use to minimize via the Optimization Solver Module
  - The Input Parameter Module then verifies if the parameters are in the appropriate format.
  - If the input parameters are valid then the Parameter Configuration module is populated with the appropriate values.
  - If no exceptions occur, the Step Size Calculator Module computes the appropriate step size, this is called from the Optimization Solver. This will heavily use the vector math, gradient calculation, and the parameter configuration modules.
  - Again, barring no exceptions, the Search Direction Calculator module computes a search direction, this is called from the Optimization Solver module. This will heavily use the vector math, gradient calculation, and the parameter configuration modules.
  - The Optimization Solver Module continues with the specific computation given the step size, search direction, vector math, gradient calculation, and the parameter configuration modules are used.
  - The Output Verification Module is used after each iteration to see if the optimizer has reached a sufficient minimum.
- output: None (returns to the program that called the function)
- exception: Various exceptions can occur in sub modules

### 6.4.5 Local Functions

None

# 7 MIS of Input Verification

## 7.1 Module

inputverify

## 7.2 Uses

- Parameter Configuration Module
- Hardware hiding module

## 7.3 Syntax

### 7.3.1 Exported Constants

None

### 7.3.2 Exported Access Programs

| Name | In | Out | Exceptions |
|---|---|---|---|
| inputVerify | params | - | - |

## 7.4 Semantics

### 7.4.1 State Variables

$d$(dim): This is the dimensionality of our problem. This is recorded based off the input and is used to ensure the consistency of the parameters' dimensions. This is a natural number $\in \mathbb{N}$ that should be less than the max dimension specified in the configuration module.

valid: This is the state of the input parameters. If the input are invalid then this variable takes the value FALSE. It is set to TRUE by default (it is a Boolean variable).

err_msg: This is the error message associated with the termination of the program. This is a string.

### 7.4.2 Environment Variables

### 7.4.3 Assumptions

### 7.4.4 Access Routine Semantics

inputVerify(A_mat, b_vec, c, x_0, H_0, step_size, max_s, min_err):

- transition: verifies each of the input parameters to an optimizer:

  **A**,(A_mat): This is the matrix A. This should be a square matrix of reals $\mathbb{R}^{d \times d}$. First checks if a square numpy array was input and records the dimension. Ensures the dimension is below the value of MAX_DIM from the constants of the parameter configuration module and is also a square matrix; otherwise changes the value of valid to

FALSE. The variable err_msg is updated accordingly, and the local function displayErrMsg(err_msg) is invoked.

$\vec{b}$,(b_vec): The vector b which defines the linear term in the quadratic function we are trying to minimize. This should be a numpy array of reals $\in \mathbb{R}^d$. If the dimension is not equal to the dimension state variable or it is not in the appropriate data type (or shape) the value of valid becomes FALSE. The variable err_msg is updated accordingly, and the local function displayErrMsg(err_msg) is invoked.

$c$,(c): The scalar c should be a real number $\in \mathbb{R}$. This is checked; if it is not the value of valid is FALSE. The variable err_msg is updated accordingly, and the local function displayErrMsg(err_msg) is invoked.

$\vec{x}_0$,(x_0): The vector x_0 which is the initial 'guess' of our solution. This should be a numpy array of reals $\in \mathbb{R}^d$. If the dimension is not equal to the dimension state variable or it is not in the appropriate data type (or shape) the value of valid becomes FALSE. The variable err_msg is updated accordingly, and the local function displayErrMsg(err_msg) is invoked.

$\mathbf{H}_0$,(H_0): This is the matrix H_0, this is our initial 'guess' of our Hessian (for BFGS only). This should be a square numpy matrix of reals $\in \mathbb{R}^{d\times d}$. If the dimension is not equal to the dimension of A_mat or it is not in the appropriate data type (or shape) the value of valid becomes FALSE. The variable err_msg is updated accordingly, and the local function displayErrMsg(err_msg) is invoked.

$\mathbf{B}_0$,(H_0): This is the matrix B_0, this is our initial 'guess' of the inverse of our Hessian (for DFP only). This should be a square numpy matrix of reals $\in \mathbb{R}^{d\times d}$. If the dimension is not equal to the dimension of A_mat or it is not in the appropriate data type (or shape) the value of valid becomes FALSE. The variable err_msg is updated accordingly, and the local function displayErrMsg(err_msg) is invoked.

step_size: The step size the minimizer the user wishes to use. This should be a positive real (or -1, if not specified; by default). If it is not the value of valid becomes FALSE. The variable err_msg is updated accordingly, and the local function displayErrMsg(err_msg) is invoked.

$\max_s$,(max_s): This is the maximum number of steps. If not specified a value will be taken from the constants module. This should be a natural number $\in \mathbb{N}$. This should also not exceed the upper bound of number of steps (to not run into excessive runtimes; this is specified in the parameters configuration module). If any of these conditions fail

then the value of valid becomes FALSE. The variable err_msg is updated accordingly, and the local function displayErrMsg(err_msg) is invoked.

$\epsilon$(min_err): This is the minimum acceptable error to terminate the program early. This should be a scalar real number $\in \mathbb{R}$ below 1 and above 0. This is checked, if these conditions fail then the value of valid becomes FALSE. The variable err_msg is updated accordingly, and the local function displayErrMsg(err_msg) is invoked.

- output: None
- exception: N/A

#### 7.4.5 Local Functions

displayErrMsg(err_msg): This function takes the state variable err_msg and displays the error message if invoked.

# 8 MIS of Parameter Configuration

## 8.1 Module

paramconfig

## 8.2 Uses

- Optimization Solver Module
- Hardware hiding module

## 8.3 Syntax

#### 8.3.1 Exported Constants

$\text{upper}_d$,(UPPER_STEP_SIZE)= 50000: This is the default upper bound on the number of steps to avoid excessive runtimes.

$\text{max}_d$,(MAX_DIM)= 6: This is the default maximum dimensionality.

$\text{max}_s$,(MAX_STEP)= 10000: This is the default maximum number of steps.

$\text{max}_s$,(MAX_STEP)= 10000: This is the default maximum number of steps.

$\epsilon$(MIN_ERR)= 0.005: This is the default minimum acceptable error to terminate the program early. This number is compared to the gradient of a possible solution. The idea is to try to get the gradient to 0.

### 8.3.2 Exported Access Programs

| Name | In | Out | Exceptions |
|---|---|---|---|
| - | - | - | - |

## 8.4 Semantics

### 8.4.1 State Variables

**A**,(A_mat): This is the matrix A, given a quadratic function we are trying to minimize this will stay the same no matter the change in choice of optimizer. This value is the coefficient matrix for the quadratic term of our equation. This is a square matrix of reals $\in \mathbb{R}^{d \times d}$.

$\vec{b}$,(b_vec): The vector b which defines the linear term in the quadratic function we are trying to minimize. This is a vector of reals $\in \mathbb{R}^d$.

$c$,(c): The scalar c which defines the constant term in the quadratic function we are trying to minimize. This is a real number $\in \mathbb{R}$.

$\vec{x}_0$,(x_0): The vector x_0 which is the initial 'guess' of our solution. This is a vector of reals $\in \mathbb{R}^d$.

$\mathbf{H}_0$,(H_0): This is the matrix H_0, this is our initial 'guess' of our Hessian (for BFGS). This is a square matrix of reals $\in \mathbb{R}^{d \times d}$.

$\mathbf{B}_0$,(B_0): This is the matrix B_0, this is our initial 'guess' of the inverse of our Hessian (for DFP). This is a square matrix of reals $\in \mathbb{R}^{d \times d}$.

$\text{step}_s$,(step_size): This is the step size specified by the user. If not specified a value will be taken from the constants of this module. If the default value of -1, this indicates to the optimizer module to use an exact line search (The program will have to compute a step size at each iteration). This is a positive real if specified $\in \mathbb{R}^+$.

$\text{max}_s$,(max_s): This is the maximum number of steps. If not specified a value will be taken from the constants of this module. This is a natural number $\in \mathbb{N}$.

$\epsilon$(min_err): This is the minimum acceptable error to terminate the program early. If not specified a value will be taken from the constants of this module. This is a scalar real number $\in \mathbb{R}$.

#### 8.4.2 Environment Variables

None

#### 8.4.3 Assumptions

That our function is a quadratic and in the standardized form specified (see SRS).

#### 8.4.4 Access Routine Semantics

None

#### 8.4.5 Local Functions

None

# 9 MIS of Vector Math Module

## 9.1 Module

vecmath

## 9.2 Uses

- Hardware Hiding Module

## 9.3 Syntax

### 9.3.1 Exported Constants

None

### 9.3.2 Exported Access Programs

| Name | In | Out | Exceptions |
|---|---|---|---|
| vecAdd | vectors/matrices of dimensions $\mathbb{R}^{n\times m}$ | vector/matrix of type $\mathbb{R}^{n\times m}$ | Dimension Error & Type Error |
| vecProd | vectors/matrices of dimensions $\mathbb{R}^{n\times l}, \mathbb{R}^{l\times m}$ | vector/matrix of type $\mathbb{R}^{n\times m}$ | Dimension Error & Type Error |
| inverse | matrices of dimensions $\mathbb{R}^{n\times n}$ | matrix of type $\mathbb{R}^{n\times n}$ | Dimension Error & Type Error |
| vecDot | vectors of dimensions $\mathbb{R}^{n}$ | scalar of type $\mathbb{R}$ | Dimension Error & Type Error |
| vecT | vectors/matrices of dimensions $\mathbb{R}^{n\times m}$ | vectors/matrices of type $\mathbb{R}^{m\times n}$ | Dimension Error & Type Error |
| vecNorm | Computes the Euclidean norm of a vector $\mathbb{R}^{n}$ | scalar of type $\mathbb{R}$ | Dimension Error & Type Error |

## 9.4 Semantics

### 9.4.1 State Variables

None

### 9.4.2 Environment Variables

None

### 9.4.3 Assumptions

All input vectors/matrices are of the correct type and are verified initially in the Input Verification module.

### 9.4.4 Access Routine Semantics

vecAdd(mat1, mat2):

- transition: Takes two matrices of the same dimension (checks this) and adds them together.
- output: outputs a matrix with the same dimension as the input matrices.
  out := matrix **x**
- exception: exc:=
  - Dimension Error: If matrices are not the same dimension
  - Type Error: If one or both the matrices are not numpy arrays of reals.

vecProd(mat1, mat2):

- transition: Takes two matrices of appropriate dimensions ($\mathbb{R}^{n\times l}$ and $\mathbb{R}^{l\times m}$ this is checked) and multiplies them together.
- output: outputs a matrix of appropriate dimension $\mathbb{R}^{n\times m}$.
  out := matrix **x**
- exception: exc:=
  - Dimension Error: If matrices are not in the appropriate dimension
  - Type Error: If one or both the matrices are not numpy arrays of reals.

inverse(mat1):

- transition: Takes a square matrix($\mathbb{R}^{n\times n}$ this is checked) and computes the inverse.
- output: outputs a matrix with the same dimension as the input matrices.
  out := matrix **x**
- exception: exc:=
  - Dimension Error: If matrices are not the same dimension
  - Type Error: If one or both the matrices are not numpy arrays of reals.

vecDot(vec1, vec2):

- transition: Takes two vectors of the same size ($\mathbb{R}^n$ this is checked) and computes the dot product. A scalar value is returned ($\mathbb{R}$).
- output: outputs a real scalar
  out := scalar $x$

- exception: exc:=

  - Dimension Error: If vectors do not have the same dimension

  - Type Error: If one or both the vectors are not numpy arrays of reals.

vecT(mat1):

- transition: Takes a matrix of size $\mathbb{R}^{n \times m}$ and returns a matrix that is the transpose of size $\mathbb{R}^{m \times n}$

- output: outputs a matrix with reversed dimension of the input matrices.
  out := matrix $\mathbf{x}$

- exception: exc:=

  - Type Error: If the vector is not a numpy array of reals.

vecNorm(vec1):

- transition: Takes a vector ($\mathbb{R}^n$ this is checked) and computes the euclidean norm. A scalar value is returned ($\mathbb{R}$).

- output: outputs a real scalar
  out := scalar $x$

- exception: exc:=

  - Dimension Error: If a matrix and not a vector (can generalize to matrices however for our purposes we only need vector norms)

  - Type Error: If the vectors are not numpy arrays of reals.

#### 9.4.5 Local Functions

None

# 10 MIS of Gradient Calculator

## 10.1 Module

gradcalc

## 10.2 Uses

- Vector Math Module

- Parameter Configuration Module

- Hardware hiding module

## 10.3 Syntax

### 10.3.1 Exported Constants

None

### 10.3.2 Exported Access Programs

| Name | In | Out | Exceptions |
|---|---|---|---|
| gradCalc | $\vec{x}_t \in \mathbb{R}^d$ | $\nabla f(\vec{x}_t) \in \mathbb{R}^d$ | - |

## 10.4 Semantics

### 10.4.1 State Variables

None

### 10.4.2 Environment Variables

None

### 10.4.3 Assumptions

None

### 10.4.4 Access Routine Semantics

gradCalc(x_t):

- transition: Using the Parameters Configuration module we obtain the constants A_mat and b_vec. Since we know for our quadratic function the analytic form of the gradient is as follows: $2 \times$ A_mat + b_vec. We simply use the Vector Math module to compute the prior equation, and output a result.
- output: curr_grad which is a numpy vector $\in \mathbb{R}^d$
- exception: -

### 10.4.5 Local Functions

None

# 11 MIS of Optimization Solver

## 11.1 Module

optimizer

## 11.2 Uses

- Vector Math
- Parameter Configuration
- Step size Calculator
- Gradient Calculator
- Search Direction Calculator
- Hardware hiding module

## 11.3 Syntax

### 11.3.1 Exported Constants

None

### 11.3.2 Exported Access Programs

| Name | In | Out | Exceptions |
|---|---|---|---|
| BFGS | $\vec{x}_0 \in \mathbb{R}^d$, $\mathbf{H}_0 \in \mathbb{R}^{d \times d}$... | $\vec{x}_t \in \mathbb{R}^d$ | - |
| DFP | $\vec{x}_0 \in \mathbb{R}^d$, $\mathbf{B}_0 \in \mathbb{R}^{d \times d}$... | $\vec{x}_t \in \mathbb{R}^d$ | - |
| FRCG | $\vec{x}_0 \in \mathbb{R}^d$... | $\vec{x}_t \in \mathbb{R}^d$ | - |

## 11.4 Semantics

### 11.4.1 State Variables

$\alpha_t$, (alpha_t) : The current step size, although this can be customized as input to be one value from the input parameter $s$, the algorithms are designed to calculate a step size at each iteration. This will be computed via the step-size calculator module for the case where an exact line search is needed.
$\vec{s}_t$, (s_t) : The current search direction, this will be computed via the search direction calculator module for each specific algorithm.
$x_t$, (x_t) : The current iteration's solution, this value is initialized with $x_0$. Notably this value resides in $\in \mathbb{R}^d$
$H_t$, (H_t) : Approximate Hessian (only relevant for BFGS) which resides in $\in \mathbb{R}^{d \times d}$ this is changed for each iteration to each algorithm's specification. This is a matrix that takes values in $\in \mathbb{R}^{d \times d}$
$B_t$, (H_t) : Approximate inverse Hessian (only relevant for DFP) which resides in $\in \mathbb{R}^{d \times d}$ this is changed for each iteration to each algorithm's specification. This is a matrix that takes values in $\in \mathbb{R}^{d \times d}$

$\text{curr}_s$(t): The current step we are on. This is a common state variable for iterative algorithms' we will monitor how long we are running our algorithm for and this helps us decide if we need to terminate the execution of the program. This is a natural number $\in \mathbb{N}$.

### 11.4.2 Environment Variables

None

### 11.4.3 Assumptions

Assume that the user inputs appropriate choices for the function. That is that the matrix $\mathbf{A}$ is positive-semidefinite (PSD). If this assumption is violated then we cannot guarantee an appropriate solution (see SRS).

### 11.4.4 Access Routine Semantics

BFGS($\mathbf{A}$, $\vec{b}, c$, $\vec{x}_0, \mathbf{H}_0, s$ = STEP_SIZE, $\max_s$ = MAX_STEP, $\epsilon$ = MIN_ERR):
or alternatively:
BFGS(A_mat, b_vec, c, x_0, H_0, step_size = STEP_SIZE, max_s = MAX_STEP, min_err= MIN_ERR)

- transition: takes the function in its parametrized form of $\mathbf{A}$, $\vec{b}, c$, initial solution and Hessian. It also takes a step size $s$ (set to -1 to indicate calculating step size otherwise the custom stepsize will be used), and max number of steps (also having a default limit to ensure feasibility). From this the algorithm utilizes the step-size calculator function for BFGS to computes the step size (if needed). It will also utilize the search direction function to compute the necessary search direction. From this, the algorithm updates the approximate solution and iteratively continues with this process until termination. The output verification module also checks if the current solution is below the error threshold, if so; terminate the program and return the value. Otherwise update the approximate Hessian via the local function computeHessianBFGS() and continue with the procedure.

- output: $\vec{x}_t$: the output is the final solution at the end of the algorithm's runtime. The solution is a vector of reals: $\mathbb{R}^d$.

- exception: Runtime Error (current step number is greater than max)

DFP($\mathbf{A}$, $\vec{b}, c$, $\vec{x}_0, \mathbf{B}_0, s$ = STEP_SIZE, $\max_s$ = MAX_STEP, $\epsilon$ = MIN_ERR):
or alternatively:
DFP(A_mat, b_vec, c, x_0, B_0, step_size = STEP_SIZE, max_s = MAX_STEP, min_err= MIN_ERR)

- transition: takes the function in its parameterized form of $\mathbf{A}$, $\vec{b}, c$, initial solution and inverse Hessian. It also takes a step size $s$ (set to -1 to indicate calculating step size

otherwise the custom step size will be used), and max number of steps (also having a default limit to ensure feasibility). From this the algorithm utilizes the step-size calculator function for DFP to computes the step size (if needed). It will also utilize the search direction function to compute the necessary search direction. From this, the algorithm updates the approximate solution and iteratively continues with this process until termination. The output verification module also checks if the current solution is below the error threshold, if so; terminate the program and return the value. Otherwise update the inverse Hessian via the local function computeInvHessianDFP() and continue with the procedure.

- output: $\vec{x}_t$: the output is the final solution at the end of the algorithm's runtime. The solution is a vector of reals: $\mathbb{R}^d$.
- exception: Runtime Error (current step number is greater than max)

FRCG(**A**, $\vec{b}$, $c$, $\vec{x}_0$, $s$ = STEP_SIZE, $\max_s$ = MAX_STEP, $\epsilon$ = MIN_ERR):
or alternatively:
FRCG(A_mat, b_vec, c, x_0, step_size = STEP_SIZE, max_s = MAX_STEP, min_err= MIN_ERR)

  - transition: takes the function in its parametrized form of **A**, $\vec{b}$, $c$, and initial solution. It also takes a step size $s$ (set to -1 to indicate calculating step size otherwise the custom stepsize will be used), and max number of steps (also having a default limit to ensure feasibility). From this the algorithm utilizes the step-size calculator function for FRCG to computes the step size (if needed). It will also utilize the search direction function to compute the necessary search direction. From this, the algorithm updates the approximate solution and iteratively continues with this process until termination. The output verification module also checks if the current solution is below the error threshold, if so; terminate the program and return the value.
  - output: $\vec{x}_t$: the output is the final solution at the end of the algorithm's runtime. The solution is a vector of reals: $\mathbb{R}^d$.
  - exception: Runtime Error (current step number is greater than max)

### 11.4.5 Local Functions

computeHessianBFGS(H_t, x_new, x_t, s_t, alpha_t):

-Given the inputs, the following equation is utilized to update the approximate of the Hessian:

$$\mathbf{H}_{t+1} = \mathbf{H}_t + \frac{\vec{y}_t \vec{y}_t^T}{\vec{y}_t^T \vec{p}_t} - \frac{\mathbf{H}_t \vec{p}_t \vec{p}_t^T \mathbf{H}_t^T}{\vec{p}_t^T \mathbf{H}_t \vec{p}_t}$$

where

$$\vec{y}_t = \nabla f(\vec{x}_{t+1}) - \nabla f(\vec{x}_t)$$

and where

$$\vec{p}_t = \alpha_t \vec{s}_t$$

So we first compute the intermediary variable y_t which is the difference between the gradient of x_new and gradient of x_t. We then use this variable to update the value of the current approximation of the Hessian (H_new) according to the above equations. The Vector Math and Gradient Calculator modules are used heavily. Returns the new approximation of the Hessian; H_new, which replaces the value of the state variable H_t.

computeInvHessianDFP(B_t, x_new, x_t, s_t, alpha_t):

-Given the inputs, the following equation is utilized to update the approximate of the inverse Hessian:

$$\mathbf{B}_{t+1} = \mathbf{B}_t + \frac{\vec{p}_t \vec{p}_t^T}{\vec{p}_t^T \vec{y}_t} - \frac{\mathbf{B}_t \vec{y}_t \vec{y}_t^T \mathbf{B}_t^T}{\vec{y}_t^T \mathbf{B}_t \vec{y}_t}$$

where

$$\vec{y}_t = \nabla f(\vec{x}_{t+1}) - \nabla f(\vec{x}_t)$$

and where

$$\vec{p}_t = \alpha_t \vec{s}_t$$

So we first compute the intermediary variable y_t which is the difference between the gradient of x_new and gradient of x_t. We also compute the intermediate variable of p_t which is just the current step size times the current search direction: p_t = alpha_t $\times$ s_t. We then use this variable to update the value of the current approximation of the inverse Hessian (B_new) according to the above equations. The Vector Math and Gradient Calculator modules are used heavily. Returns the new approximation of the inverse Hessian; B_new, which replaces the value of the state variable B_t.

# 12 MIS of Step-Size Calculator

## 12.1 Module

stepsizecalc

## 12.2 Uses

- Vector Math Module
- Parameter Configuration Module
- Hardware Hiding Module

## 12.3 Syntax

### 12.3.1 Exported Constants

None

### 12.3.2 Exported Access Programs

| Name | In | Out | Exceptions |
|---|---|---|---|
| exactStep | $\vec{x}_t, \vec{s_t} \in \mathbb{R}^d$ | $\alpha_t \in \mathbb{R}$ | - |

## 12.4 Semantics

### 12.4.1 State Variables

None

### 12.4.2 Environment Variables

None

### 12.4.3 Assumptions

The function class we are minimizing is quadratic. Also the matrix **A** is PSD. The first assumption is quite critical because we compute the gradient analytically assuming the function is quadratic, then we only need to input the current vector and the parameters that define the quadratic ($\mathbf{A}$, $\vec{b}$) to obtain a value of the gradient at a point.

### 12.4.4 Access Routine Semantics

exactStep(x_t, s_t):

- transition: compute the gradient using the Gradient Calculator module. A_mat is obtained from the Parameters Configuration module. Now the exact step can be computed easily via matrix operations; for minimizing a quadratic function via line search returns a step size in the analytic form:

$$\alpha_t = -\frac{\nabla f(\vec{x}_t)^T \vec{s}_t}{\vec{s}_t^T \mathbf{A} \vec{s}_t}$$

- output: $\alpha_t$(alpha_t) a positive real number
- exception: none

### 12.4.5 Local Functions

None

# 13 MIS of Search Direction Calculator

## 13.1 Module

searchdir

## 13.2 Uses

- Vector Math Module
- Parameter Configuration Module
- Gradient Calculator Module
- Hardware Hiding Module

## 13.3 Syntax

### 13.3.1 Exported Constants

None

### 13.3.2 Exported Access Programs

| Name | In | Out | Exceptions |
|---|---|---|---|
| dirBFGS | $\mathbf{H}_t \in \mathbb{R}^{n \times n}$, $\vec{x}_t \in \mathbb{R}^n$ | $\vec{s}_t \in \mathbb{R}^d$ | - |
| dirDFP | $\mathbf{H}_t \in \mathbb{R}^{n \times n}$, $\vec{x}_t \in \mathbb{R}^n$ | $\vec{s}_t \in \mathbb{R}^d$ | - |
| dirFRCG | $\vec{x}_t, \vec{x}_{t-1}, \vec{s}_{t-1} \in \mathbb{R}^d$ $t \in \mathbb{N}$ | $\vec{s}_t \in \mathbb{R}^d$ | - |

## 13.4 Semantics

### 13.4.1 State Variables

None

### 13.4.2 Environment Variables

None

### 13.4.3 Assumptions

See SRS and Module 12's assumptions.

### 13.4.4 Access Routine Semantics

dirBFGS(H_t, x_t):

- transition: The search direction is computed by solving the following equation: $\mathbf{H}_t \vec{s}_t = -\nabla f(\vec{x}_t)$. Which is solved by $\vec{s}_t = -\mathbf{H}_t^{-1} \nabla f(\vec{x}_t)$. So we need to invert the current Hessian approximation and matrix multiply the negative of the current gradient.

- output: s_t; the current iteration's search direction. This is a numpy array of reals; a vector $\in \mathbb{R}^d$.

- exception: -

dirDFP(B_t, x_t):

- transition: The search direction is computed by solving the following equation: $\vec{s}_t = -\mathbf{B}_t \nabla f(\vec{x}_t)$. So we need the current inverse Hessian approximation and matrix multiply the negative of the current gradient.

- output: s_t; the current iteration's search direction. This is a numpy array of reals; a vector $\in \mathbb{R}^d$.

- exception: -

dirFRCG(x_t, x_prev, s_prev,t):

- transition: First check if we are on the first step via variable t. If we are we obtain the step size for the first step size via the Step Size Calculator module, and compute the current solution x_t (the passed variable would be irrelevant) by taking the previous iteration (x_prev) and subtracting the gradient of the previous iteration times the first step size. For all other computations we compute the direction as follows: compute the negative of the gradient of the current solution (x_t) and add beta_t times the previous search direction: $\vec{s}_t = -\nabla f(x_t) + \beta_t \vec{s}_{t-1}$. Where beta_t is computed as the dot product of the gradient of the current iteration with itself divided by the dot product of the gradient of the previous iteration with itself:

$$\beta_t = \frac{(\nabla f(x_t))^T (\nabla f(x_t))}{(\nabla f(x_{t-1}))^T (\nabla f(x_{t-1}))}$$

- output: s_t; the current iteration's search direction. This is a numpy array of reals; a vector $\in \mathbb{R}^d$.

- exception: -

### 13.4.5 Local Functions

betaFR(x_t, x_prev):

- transition: As stated prior beta_t is computed as the dot product of the gradient of the current iteration with itself divided by the dot product of the gradient of the previous iteration with itself:

$$\beta_t = \frac{(\nabla f(x_t))^T(\nabla f(x_t))}{(\nabla f(x_{t-1}))^T(\nabla f(x_{t-1}))}$$

- output: beta_t; the current iteration's beta. This is a scalar; $\in \mathbb{R}$.
- exception: -

# 14 MIS of Output Verification

## 14.1 Module

## 14.2 Uses

- Vector Math Module
- Parameter Configuration Module
- Gradient Calculator
- Hardware hiding module

## 14.3 Syntax

### 14.3.1 Exported Constants

None

### 14.3.2 Exported Access Programs

| Name | In | Out | Exceptions |
|---|---|---|---|
| outputVerify | $x_t \in \mathbb{R}^d$ | valid_sol $\in \mathbb{B}$ | - |

## 14.4 Semantics

### 14.4.1 State Variables

None

### 14.4.2 Environment Variables

None

### 14.4.3 Assumptions

None

### 14.4.4 Access Routine Semantics

outputVerify(x_t):

- transition: Takes the current value of the solution, uses the Gradient Calculator module to obtain a value of curr_grad. From this value they compute the norm via the Vector Math module. This function then compares this value to the error threshold from the Parameters Configuration module. If it is less than the threshold valid_sol is set to TRUE. This causes the program to return to the Optimization Solver module where the program will terminate and return the solution as an acceptable solution.

- output:
  valid_sol: If the current solution is below the error threshold then this variable takes the value TRUE. It is set to FALSE by default (it is a Boolean variable).

- exception: None

### 14.4.5 Local Functions

None

# References

Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, Upper Saddle River, NJ, USA, 2nd edition, 2003.

Daniel M. Hoffman and Paul A. Strooper. *Software Design, Automated Testing, and Maintenance: A Practical Approach*. International Thomson Computer Press, New York, NY, USA, 1995. URL http://citeseer.ist.psu.edu/428727.html.

# 15 Appendix

N/A

# 16 Reflection

The information in this section will be used to evaluate the team members on the graduate attribute of Problem Analysis and Design. Please answer the following questions:

1. What are the limitations of your solution? Put another way, given unlimited resources, what could you do to make the project better? (LO_ProbSolutions)

   The dimensionality of the problem is quite limited due to computational demands. This could be expanded with greater resources. Another factor tied to this would be an expanded upper limit on the number of maximum steps. We could theoretically let our algorithms run for a lot more iterations. Similarly, for min_error which if it was lower we could get better approximations of the final solutions.

2. Give a brief overview of other design solutions you considered. What are the benefits and tradeoffs of those other designs compared with the chosen design? From all the potential options, why did you select the documented design? (LO_Explores)

   Another design solution considered was having a singular entry point into the library via a function called 'optimize' which would specify the optimizer of choice via a string variable. This string variable would be called *func_name* and take values of 'BFGS', 'DFP', and 'FRCG.' Another module would be needed called Input (perhaps a different name would be better?) which would then allow a Control Module to design and control the flow for the particular optimizer. We chose not to go in this direction as this was not consistent with the way most libraries of functions are designed. This also led to further problems such as having a singular input to encompass all the optimizers which have varying inputs. For instance, B_0 is not an input in BFGS or FRCG and similarly H_0 is not an input in DFP or FRCG etc. This would in turn require further safeguards to worry about cases where incorrect inputs are made; ie 'BFGS' is *func_name* but H_0 is not provided but B_0 is. This would require further design decisions that would increase the complexity of the project needlessly.