Multi-start PSO with CCD Local Optimizer (MPSO-CCD)

A Global Optimization Technique

User Manual

by

Dr. Hum Nath Bhandari, Dr. Philip W. Smith

Dr. Xinyou Ma, Dr.William L. Hase

Email: humnath.bh@gmail.com, philip.smith@ttu.edu

Xinyou.Ma@ttu.edu, bill.hase@ttu.edu

Table of Contents

Introduction

This manual provides the stepwise guidelines for the serial as well as parallel implementation of different versions of the improved PSO algorithm [1], [2]. This manual is based on the PSO implementation on various projects that I had done during my PhD dissertation research.

Chapter 1

This chapter documents the C code of the PSO algorithm and its implementation, both in serial and parallel. Compilation and execution of the code will be demonstrated using some example optimization problems.

1.1 Files for Serial Implementation

The following are the files required to implement the serial versions of the PSO algorithm. All files must be in the same directory.

1.1.1 Header Files

1. Basic C Libraries.

   `stdio.h`, `stdlib.h`, `string.h`, `math.h`, `time.h`

2. Program Related Header files.

   `pso.h`, `randomlib.h`, `matmul2.h`

1.1.2 C Files

1. `pso.c`: PSO related functions file.

   This file contains the function headers of optimization algorithms such as PSO algorithm, CCD algorithm, Brent algorithm, MPSO, MPSO-CCD, and some other supportive functions. The corresponding function headers are included in `pso.h` file.

2. `matmul2.c`: Math related functions file.

   This file consists some math related functions which are not included in the basic math library, called `math.h`, but they are essential to facilitate the PSO algorithm implementation such as matrix multiplications, transposing matrix, etc. The corresponding function headers are included in the file `matmul2.h`.

3. `randomlib.c`: Random numbers generators.

   This file contains the functions related to random number generators. These random number generators are based on the algorithm in a FORTRAN version (Marsaglia et al., 1990). Since the PSO algorithm is stochastic algorithm, these functions are essential to initialize and re-initialize the PSO particle positions, velocities, etc.

4. Main files

   `main.c`, `main_noghost.c`, `main_withghost.c`

   These files perform the following tasks:

   - Create necessary local and global variables.
   - Setup random seeds.
   - Initialize random number generators.
   - Receive PSO parameters from command line.
   - Initialize PSO domain.
   - Read data if necessary.
   - Execute optimization functions.
   - Write outputs.

1.1.3 Additional Files

1. Additional input files for potential energy fitting problem.

   `bounds_noghost.txt`

   `bounds_withghost.txt`

   `energy.txt`

   `geom.txt`

2. Output file for benchmark functions.

   `outputpara.txt`

3. Output files for potential energy fitting problem.

   `output_noghost.txt`

   `output_withghost.txt`

4. Miscellaneous files.

   `job.sh`

   `job_noghost.sh`

   `job_withghost.sh`

1.2 Compiling and Executing the Code

Assuming all above mentioned files are in the same directory, the following commands are used to compile and run the program.

1.2.1 Compiling the Code

The following command can be used for compiling the code.

1. Benchmark functions.

   ```
   $g++ main.c pso.c matmul2.c randomlib.c -o pso
   ```

2. Potential energy fitting problem.

   (i) Without ghost atom:

   ```
   $g++ main_noghost.c pso.c matmul2.c randomlib.c
    -o noghost
   ```

   (ii) With ghost atom:

   ```
   $g++ main_withghost.c pso.c matmul2.c randomlib.c
    -o withghost
   ```

These commands produce `pso`, `noghost`, and `withghost` as a binary objects.

1.2.2 Executing the Serial Code in Command Line

1. Benchmark functions.

   ```
   $./pso Rosenbrock 50 10 500 50 -10 10 0
   ```

   where

   arg 1: function to be minimized

   arg 2: number of particles used in PSO

   arg 3: dimension of the function

   arg 4: maximum number of iterations allowed

   arg 5: maximum number of runs allowed

   arg 6: lower bound of the domain

   arg 7: upper bound of the domain

   arg 9: choice of the algorithm, where 0: PSO, 1: MPSO, 2: MPSO-CCD

2. Potential energy fitting problem.

   (i) Without ghost atom:

   ```
   $./noghost fit_noghost 50 16 500 10 0
   ```

   (ii) With ghost atom:

   ```
   $./withghost fit_withghost 50 25 500 10 0
   ```

   where

   arg 1: model problem to be minimized

   arg 2: number of particles used in PSO

   arg 3: dimension of the function

   arg 4: maximum number of iterations allowed

   arg 5: maximum number of runs allowed

   arg 6: choice of the algorithm, where 0: PSO, 1: MPSO, 2: MPSO-CCD

1.2.3 Executing the Serial Code Using Job Script

The following are the necessary commands for running the serial code using job scripts.

1. Benchmark functions.

```
#!/bin/bash
#$ -V
#$ -cwd
#$ -j y
#$ -S /bin/bash
#$ -N pso
#$ -q serial
#$ -P hostname
./pso Rosenbrock 50 10 500 10 -10 10  0
```

2. Potential energy fitting without ghost atom

```
#!/bin/bash
#$ -V
#$ -cwd
#$ -j y
#$ -S /bin/bash
#$ -N noghost
#$ -q serial
#$ -P hostname
./noghost fit_noghost 50 16 500 10 0
```

3. Potential energy fitting with ghost atom

```
#!/bin/bash
#$ -V
#$ -cwd
#$ -j y
#$ -S /bin/bash
#$ -N withghost
#$ -q serial
```

```
#$ -P hostname
./withghost fit_withghost 50 25 500 10 0
```

The serial code can be executed using the following job submission commands.

```
qsub job.sh
qsub job_noghost.sh
qsub job_withghost.sh
```

1.3 Files for Parallel Implementation

The following are the files required to implement the parallel versions of the PSO algorithm. All files must be in the same directory.

1.3.1 Header Files

1. Basic C Libraries.

   `stdio.h`, `stdlib.h`, `string.h`, `math.h`, `time.h`

2. MPI Library.

   `mpi.h`

3. Program Related Header files.

   `mpipso.h`, `randomlib.h`, `matmul2.h`

1.3.2 C Files

1. `mpipso.c`: PSO related functions file.

   This file contains the function headers of optimization algorithms such as CCD, Brent algorithm, parallel PSO, parallel MPSO, parallel MPSO-CCD algorithm, and some other supportive functions. The corresponding function headers are included in `mpipso.h` file.

2. `matmul2.c`: Math related functions file.

   This file is the same as that was in the serial case.

3. `randomlib.c`: Random numbers generators.

   This file is the same as that was in the serial case.

4. Main files

   `mpimain.c`, `mpimain_noghost.c`, `mpimain_withghost.c`

1.3.3 Additional Files

1. Additional input files for potential energy fitting problem.

   These files are the same as that were in the serial case.

2. Output file for benchmark functions.

   This file is the same as that was in the serial case.

3. Output files for potential energy fitting problem.

   These files are the same as that were in the serial case.

4. Miscellaneous files.

   `mpijob.sh`

   `mpijob_noghost.sh`

   `mpijob_withghost.sh`

1.4 Compiling and Running the Parallel Code

Assuming all above mentioned files are in the same directory, the following commands are used to compile and run the program.

1.4.1 Compiling the Parallel Code

1. Benchmark functions.

   Step 1: `$module load intel`

   Step 2: `$load impi`

   Step 3:

```
$mpicc -o mpipso mpimain.c matmul2.c
randomlib.c mpipso.c -lm
```

2. Potential energy fitting problem.

   Step 1: `$module load intel`

   Step 2: `$load impi`

   Step 3:

   (i) Without ghost atom:

```
$mpicc -o mpinoghost mpimain_noghost.c matmul2.c
randomlib.c mpipso.c -lm
```

   (ii) With ghost atom:

```
$mpicc -o mpiwithghost mpimain_withghost.c matmul2.c
randomlib.c mpipso.c -lm
```

1.4.2 Executing the Parallel Code in Command Line

1. Benchmark functions.

```
$mpirun -np num_processors
./pso Rosenbrock 50 10 500 50 -10 10 0
```

2. Potential energy fitting problem.

   (i) Without ghost atom:

```
$mpirun -np num_processors
./mpinoghost fit_noghost 50 16 500 10 0
```

   (ii) With ghost atom:

```
$mpirun -np num_processors
 ./mpiwithghost fit_withghost 50 25 500 10 0
```

Here, `np` represents the number of processors used in the parallel environment. This number depends on the hardware used in the host server. Moreover, all arguments are the same as that were in serial versions.

1.4.3 Executing the Parallel Code Using Job Script

The following are the necessary commands for running the parallel code using job script. The user needs to make a separate file `mpijob.sh` by including these commands.

1. Benchmark functions.

```
#!/bin/bash
#$ -V
#$ -cwd
#$ -j y
#$ -S /bin/bash
#$ -N mpipso
#$ -o $JOB_NAME.o$JOB_ID
#$ -e $JOB_NAME.e$JOB_ID
#$ -q queue type
#$ -pe fill num_processors
#$ -P hostname
module load intel impi
mpirun--machinefile machinefile.$JOB_ID -np $NSLOTS
./mpipso Rosenbrock 50 10 500 10 -10 10 0
```

2. Potential energy fitting without ghost atom

```
#!/bin/bash
#$ -V
#$ -cwd
#$ -j y
#$ -S /bin/bash
#$ -N mpinoghost
#$ -o $JOB_NAME.o$JOB_ID
#$ -e $JOB_NAME.e$JOB_ID
#$ -q queue type
#$ -pe fill num_processors
#$ -P hostname
module load intel impi
```

```
mpirun --machinefile machinefile.$JOB_ID -np $NSLOTS
./mpinoghost fit_noghost 50 16 500 10 0
```

3. Potential energy fitting with ghost atom

```
#!/bin/bash
#$ -V
#$ -cwd
#$ -j y
#$ -S /bin/bash
#$ -N mpiwithghost
#$ -o $JOB_NAME.o$JOB_ID
#$ -e $JOB_NAME.e$JOB_ID
#$ -q queue type
#$ -pe fill num_processors
#$ -P hostname
module load intel impi
mpirun --machinefile machinefile.$JOB_ID -np $NSLOTS
./mpiwithghost fit_withghost 50 25 500 10 0
```

The parallel code can be executed using the following job submission commands.

```
qsub mpijob.sh
qsub mpijob_noghost.sh
qsub mpijob_withghost.sh
```

1.5 Optimization Algorithms

We have implemented three versions of the PSO algorithm: standard PSO, multi-start PSO (MPSO), and multi-start PSO with local optimizer (MPSO-CCD). The code for serial implementation of these algorithms can be found in the file `pso.c` and the parallel implementation code can be found in the file `mpipso.c`. The function headers of these algorithms for serial and parallel implementations are included in `pso.h` and `mpipso.h` files respectively. The following are the major functions that are included in these files, together with some additional supportive functions.

1. Serial PSO versions:

   `pso_optimization()`

   `multi_pso_optimization()`

   `multi_pso_local_optimizer()`

2. Cyclic Coordinate Descent (CCD) local search method

   `local_optimizer()`

3. One dimensional local search method

   `brent_algorithm()`

4. Parallel PSO versions:

   `mpipso_optimization()`

   `mpimulti_pso_optimization()`

   `mpimulti_pso_local_optimizer()`

1.5.1 Serial Standard PSO: `pso_optimization()`

Purpose: to minimize a multi-dimensional optimization problem using the standard PSO algorithm.

Usage:

```
void pso_optimization(double (*fun)(double* x, int nd),
int np, int nd, int ni, double* lb, double* ub,
double* value, double* gbest, int ni_stop, double tol)
```

Arguments:

1. **double** `fun(`**double*** `x,` **int** `nd)` - function to be minimized with the following arguments:
   - **int** `nd` - length of `x`.
   - **double*** `x` - vector of length `nd` at which the function is evaluated (Input).
2. **int** `np` - number of particles (Input).
3. **int** `ni` - maximum number of iterations allowed (first terminating criteria) (Input).
4. **double*** `lb` - vector of length `nd` which consists lower bounds of `x` (Input).
5. **double*** `ub` - vector of length `nd` which consists upper bounds of `x` (Input).
6. **double*** `value` - function value at the `gbest` (Input/Output).
7. **double*** `gbest` - vector of length `nd` containing the best estimate of the minimum found (Input/Output).
8. **int** `ni_stop` - number of iterations before terminating the algorithm with second stopping criterion (Input). Algorithm stops if there is no significant improvement defined by `tol` achieved even after `ni_stop` successive iterations.

9. **double** `tol` - second terminating criterion (Input). The algorithm stops when the error is less than `tol`. The relative error is calculated as follows:

$$|\frac{fun(gbest\_old) - fun(gbest\_current)}{|fun(gbest\_current)| + tol}| < tol$$

where `gbest_current` is the function value at the current iteration and `gbest_old` is the function value calculated before `ni_stop` iterations.

1.5.2 Serial MPSO: `multi_pso_optimization()`

The standard PSO algorithm `pso_optimization()` can be executed multiple times to improve the quality of the solution. In this method, the algorithm is allowed to perform multiple runs denoted by `maxRun` by keeping the previous best estimate `gbest` as an initial guess for the next run. This can be performed by using an additional argument `maxRun` in the `pso_optimization()`. fptr fun, int np, int nd, int ni,double* lb, double* ub, double* value, double* gbest, int ni_stop, double tol **Usage**:

```
void multi_pso_optimization(double (*fun)(double* x, int nd),
 int np, int nd, int ni, int maxRun, int* type, double* lb,
double* ub, double* value, double* gbest, int ni_stop,
 double tol)
```

1.5.3 Serial MPSO-CCD: `multi_pso_local_optimizer()`

This function calls `pso_optimization()` and `local_optimizer()` repeatedly. First, the best estimate `gbest` is obtained by using the `pso_optimization()` and then gbest vector is provided to the `local_optimizer()` as an initial guess.
The `local_optimizer()` further improves the estimate by using a 1-dimensional line search, called `brent_algorithm()`. The process is repeated `maxRun` times. This can be performed by using some additional arguments in the `multi_pso_optimization()`. This process can be summarized as follows.

Step 1: Apply `pso_optimization()`

Step 2: Apply `local_optimizer()`

Step 3: Repeat Step 1 and Step 2 `maxRun` times or until second termination criterion is reached.

Usage:

```
void multi_pso_local_optimizer(double (*fun)(double* x, int nd),
int np, int nd, int ni, int maxRun,  int* type, double* lb,
double* ub, double* value, double* gbest, int ni_stop,
double tol)
```

1.5.4 CCD Algorithm: `local_optimizer()`

This is a local search algorithm, which is used to refine the solution obtained from the PSO algorithm.

Purpose: to minimize multi-dimensional function using Cyclic Coordinate Descent (CCD) algorithm.

Usage:

```
void local_optimizer(double (*fun)(double* y, int nd),
int nd, double* lb,  double* ub, double* value, double* x,
int ni_stop, double tol)
```

Arguments:

1. **double** `fun(double* y, int nd)` - function to be minimized with the following arguments:
   - **int** `nd` - length of `y` (Input).
   - **double*** `y` - vector of length `nd` at which the function is evaluated (Input).
2. **double*** `lb` - vector of length `nd` which consists lower bounds of `x` (Input).

3. **double*** ub - vector of length nd which consists upper bounds of x (Input).

4. **double*** value - function value (Input/Output).

5. **double*** x - initial guess (Input/Output).

6. **int** ni_stop - number of iterations before terminating the algorithm with second stopping criterion. Algorithm stops if there is no significant improvement defined by tol achieved even after ni_stop successive iterations.

7. **double** tol - second terminating criterion. The algorithm stops when the error is less than tol. The relative error is calculated as follows:

$$|\frac{fun(gbest\_old) - fun(gbest\_current)}{|fun(gbest\_current)| + tol}| < tol$$

where gbest_current is the function value at the current iteration and gbest_old is the function value calculated before ni_stop iterations.

1.5.5 1D Line Search: `brent_algorithm()`

This function is supplied in `local_optimizer()` to perform one dimensional line search using the Brent algorithm.

Purpose: to minimize one dimensional function using a Brent algorithm.

Usage: **double** brent_algorithm(**double** l, **double** u, **double**(*g)(**double** t), **double** tol)

Arguments:

1. **double** l - lower bound of the interval $[l, u]$ (Input).

2. **double** u - upper bound of the interval $[l, u]$ (Input).

3. **double**(*g)(**double** t) - one dimensional function to be minimized with the following arguments.

- **double** t - a point at which the function g is evaluated (Input).
- Output - function value at t.

4. **double** tol - Acceptable tolerance for the minimum location.
5. **Output**:
   - brent_algorithm() returns an estimate for the minimum location with accuracy 3*SQRT_EPSIlON*abs(val) + tol, where val is the return value from the Brent algorithm.
   - The function always obtains a local minimum which coincides with the global one only if a function under investigation being unimodular.
   - If a function being examined possesses no local minimum within the given range, it returns 'l' if $g(l) < f(u)$, otherwise it returns the right range boundary value u.

1.5.6 Parallel PSO Versions

The PSO algorithm can be executed in parallel by using some additional MPI commands in existing serial code.

1. Parallel PSO: mpipso_optimization()

   Usage:

```
void mpipso_optimization(double (*fun)(double* x, int nd),
int np, int nd, int ni, int* type,  double* lb, double* ub,
 double* value, double* gbest, int ni_stop, double tol,
int myrank)
```

2. Parallel MPSO: mpimulti_pso_optimization()

   Usage:

```
void mpimulti_pso_optimization(double (*fun)(double* x, int nd),
int np, int nd, int ni, int maxRun, int* type, double* lb,
```

```
double* ub, double* value, double* gbest, int ni_stop,
double tol, int myrank)
```

3. Parallel MPSO-CCD: `mpimulti_pso_local_optimizer()`

   Usage:

   ```
   void mpimulti_pso_local_optimizer (double (*fun)(double* x, int nd), int np, int nd, int ni, int maxRun, int* type, double* lb, double* ub, double* value, double* gbest, int ni_stop, double tol, int myrank)
   ```

These three parallel versions of the PSO algorithms are very similar to their corresponding serial versions except one additional argument, `myrank`. This additional argument is used to keep track of processor rank. All other supportive functions that were used in serial version will remain same and will be applicable in the parallel implementation.

1.5.7 Additional Supportive Functions

Some additional supportive functions are necessary to implement serial and parallel versions of the PSO algorithm. These functions are supplied to perform some specific tasks such as initializing domain, checking boundary of domain, adjusting domain, etc.

1. `void update_best(double * g, double * x, int nd)`
2. `void initialize_pso_domain( char * bound_file, int* type, double* lb, double* ub, int nd)`
3. `int lower_bound_check(double g, double l, double u, double epsilon)`
4. `int upper_bound_check(double g, double l, double u, double epsilon)`
5. `void adjust_pso_domain(double* lb, double* ub, int nd)`
6. `double fun_1D(double t)`

1.6 Math Related Functions

Some math related functions are also necessary to facilitate the PSO algorithm implementation such as matrix multiplications, transposing matrix, etc. The c code of these functions are included in the file `matmul2.c` and corresponding function headers are included in the file `matmul2.h`.

1. `struct matrix { double ** mat; int row; int col };`
2. `struct matrix * get_matrix( int row , int col)`
3. `struct matrix * matrix_mult( struct matrix* m1, struct matrix* m2)`
4. `struct matrix * transpose( struct matrix * m )`
5. `double ** get_mat( int row , int col)`
6. `void free_matrix( struct matrix * m)`
7. `struct matrix * read_matrix( char* filename, int nrow, int ncol)`
8. `void  write_matrix( char* filename, struct matrix * m)`

1.7 Random Number Generators

The files `randomlib.c` and `randomlib.h` contain some functions related to random number generators. This random number generator is based on the algorithm in a FORTRAN version (Marsaglia et al., 1990). Since the PSO algorithm is stochastic algorithm, these functions are essential to initialize and re-initialize the PSO particle positions and velocities.

1. `void RandomInitialise(int seed1, int seed2)`
2. `double RandomUniform(void)`
3. `double RandomGaussian(double a, double b)`

4. `int RandomInt(int n1, int n2)`

5. `double RandomDouble(double a, double b)`

1.8 Example

We provide an example `main.c` file which is used to minimize the Rosenbrock function using serial code. The Rosenbrock function can be defined as follows.

$$f(\mathbf{x}) = \sum_{j=1}^{n-1}[100(x_{j+1} - x_j^2) + (x_j - 1)^2]$$

It has a global minimum value $f(\mathbf{x}^*) = 0$ at $\mathbf{x}^* = (1, 1, 1, ..., 1)$.

main.c file

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include "randomlib.h"
#include "matmul2.h"
#include "pso.h"

\*=== Objective functions Headers ===*/

double Rosenbrock(double*x, int nd);

int main(int argc,char*argv[]){
int i,j,k;
int seed1,seed2;
srand((unsigned)time(NULL));
seed1= rand()%30000;
seed2= rand()%30000;

/* Reading command line inputs */
```

```
char func[80];
strcpy(func, argv[1]);
fptr f  = functionLookup(func);

int np = atoi(argv[2]);
int nd = atoi(argv[3]);
int ni = atoi(argv[4]);
int maxRun =atoi(argv[5]);
double l  = atof(argv[6]);
double u =  atof(argv[7]);
int option =atoi(argv[8]);

/* Alocationg memory spaces */
int* type = (int*)malloc(sizeof(int)*nd);
double*lb = (double*)malloc(sizeof(double)*nd);
double*ub = (double*)malloc(sizeof(double)*nd);
double * gbest = (double*)malloc(sizeof(double)*nd);
for(i = 0;i < nd ;i++){
type[i] = 0;
lb[i] = l; ub[i] = u;
gbest[i] = (ub[i]-lb[i])/3;
}

RandomInitialise(seed1, seed2);
double tol = 0.000001;
double value = 9999999;
int ni_stop = 100;
int ntries = 10;
/*Start Optimization */
double start= time(NULL);
if(option==0){
pso_optimization(f,np,nd,ni,lb,ub,&value,
gbest,ni_stop,tol);}
else if(option==1){
```

```
multi_pso_optimization(f,np,nd,ni,maxRun,type,lb,ub,
&value,gbest,ni_stop,tol);}
else if(option==2){
multi_pso_local_optimizer(f,np,nd,ni,maxRun,type,lb,ub,
&value,gbest,ni_stop,tol);}
else{printf("\n No option is found !!!\n");}
double finish= time(NULL);
double time=difftime(finish, start);

FILE* fp;
fp = fopen("outputpara.txt", "w");
fprintf(fp, "\n PSO parameters used:\n");
fprintf(fp, "seed1:%d\n", seed1);
fprintf(fp, "seed2:%d\n", seed2);
fprintf(fp, "nd:%d\n", nd);
fprintf(fp, "np:%d\n", np);
fprintf(fp, "ni:%d\n", ni);
fprintf(fp, "maxRun:%d\n", maxRun);
fprintf(fp, "Elapsed time:%lf\n", time);
fprintf(fp, "best value: %lf\n", value);
fprintf(fp, "\n Optimal parameters(gbest)\n");
for (j = 0; j < nd; j++) { fprintf(fp, "%lf\n", gbest[j]);}
fclose(fp);

free(lb);free(ub);free(gbest);free(type);

return 0;
}
\*=== Objective functioin definition ====*/
double Rosenbrock(double * x, int nd) {
double sum = 0;
int i;
for(i=0;i<(nd-1);i++){
sum += 100*pow(x[i+1]- pow(x[i],2),2)+pow(x[i]-1,2);
```

```
}
return(sum);
}
```

1.9 Source Code

The complete source code of both serial and parallel implementation can be found on GitHub: `https://github.com/humnath5/PSO-Package`.

Bibliography

[1] Bhandari, Hum Nath (2018), Particle swarm optimization (PSO) algorithm: analysis, improvements, and applications, PhD dissertation.

[2] Bhandari, H. N., Ma, X., Paul, A. K., Smith, P., & Hase, W. L. (2018). PSO Method for Fitting Analytic Potential Energy Functions. Application to I(H2O). Journal of chemical theory and computation, 14(3), 1321-1332.

[3] Kennedy, J. and Eberhart R. (1995). Particle swarm optimization. Proc. of IEEE Intl. Conference on Neural Networks, Perth, Australia, November 27-December 1, 4:19421948.

[4] Marsaglia, G., Zaman, A., & Tsang, W. W. (1990). Toward a universal random number generator. Stat. Prob. Lett., 9(1), 35-39.

[5] Poli, R. (2008). Analysis of the publications on the applications of particle swarm optimisation. Journal of Artificial Evolution and Applications.

[6] Bazarra, M., & Shetty, C. (1979). Nonlinear programming, theory and algorithms. John Wiley and Sons.

[7] Brent, R. (1973). Algorithms for minimization without drivatives. Prentice-Hall.

[8] Dekker, T. (1969). Finding a zero by means of successive linear interpolation. Constructive aspects of the fundamental theorem of algebra, 3751.

[9] Van den Bergh, F. (2002). An analysis of particle swarm optimization. November, Ph. D. Dissertation, Faculty of Natural and Agricultural Sci., Univ. Petoria, Pretoria, South Africa.