PSOFT Users' Manual

Mahesh B. Patil

Department of Electrical Engineering
Indian Institute of Technology Bombay
Mumbai-400076
e-mail: mbpatil@ee.iitb.ac.in

Contents

Chapter 1

Introduction

There are many situations in science and engineering where certain parameters need to be optimised for a specific objective. For example, we may have a model (a set of equations) whose parameter values are to be assigned so that the model produces an accurate description of the corresponding physical situation, such as the electrical behaviour of a transistor or the speed versus torque curve of a machine. Or we may have a design problem in which a circuit or a system needs to be designed to meet certain specifications. The design process would call for assigning appropriate values to the system parameters such that the system performance is as close as possible to the specifications.

Optimisation problems can be (a) single-objective where only one function (of one or more variables) is to be optimised, i.e., minimised or maximised, or (b) multi-objective where multiple functions are to be optimised.

Several methods have been developed for optimisation. Methods in which only the objective function values are used to locate the optimum (minimum or maximum) point are called "direct search" methods. Methods which require to use the objective function values as well as the derivative (gradient) values are called gradient-based methods. The success of these methods, i.e., whether and how fast it converges to the optimum point, depends on various factors, including the starting point used. A comprehensive description of the various methods, their advantages and disadvantages, is given in [1].

If the objective function space has several optimum positions, the direct search or gradient-based methods may find a local optimum but fail to find the global optimum. In such cases, stochastic methods, which incorporate randomness in the search process, are more effective. The randomness – which is inherent in stochastic methods – ensures that the search process does not get stuck in a narrow region (such as a local minimum) of the search space. In other words, stochastic methods enable a more thorough *exploration* of the search space. Furthermore, these methods do not require gradient information from the user. In the rest of this manual, we will focus on one of the stochastic methods, viz., particle swarm optimisation.

1.1 Particle Swarm Optimisation

As the name suggests, the Particle Swarm Optimisation (PSO) algorithm is based on a swarm (group) of particles. The motion of the particles within the search space is governed by certain rules such that each particle improves its *fitness* over a sufficiently long interval. What exactly

is a "particle" and how is the "fitness" of a particle evaluated? That depends on the problem being solved, as we will soon illustrate with an example.

The movement of the particles is carried out in discrete steps or "iterations." To understand the PSO algorithm, let us first define the attributes of the particles as follows[1].

$\mathbf{x}_k^i$: position of the i^{th} particle in the k^{th} iteration
$\mathbf{v}_k^i$: velocity of the i^{th} particle in the k^{th} iteration
$\mathbf{p}_k^i$: the best *personal* position of the i^{th} particle so far, i.e., the best position which the i^{th} particle has visited up to the k^{th} iteration
$\mathbf{g}_k$: the best *global* position in the whole swarm up to the k^{th} iteration, i.e., the best of all $\mathbf{p}_k^i$

In the "standard" PSO algorithm, which is the subject of this manual, the velocity of each particle is upgraded as

$$\mathbf{v}_{k+1}^i = \omega\,\mathbf{v}_k^i + \phi_1\,R_{1k}^i(\mathbf{p}_k^i - \mathbf{x}_k^i) + \phi_2 R_{2k}^i(\mathbf{g}_k - \mathbf{x}_k^i), \tag{1.1}$$

where ω, ϕ_1, ϕ_2 are constant positive real numbers, and R_{1k}^i, R_{2k}^i are random numbers distributed uniformly over $[0, 1]$. Let us look at the significance of each of the terms on the right-hand side.

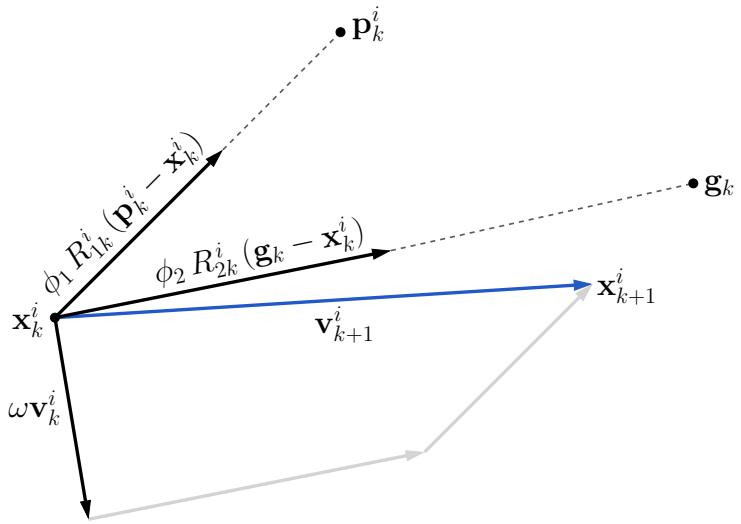

Figure 1.1: Schematic representation of velocity update (Eq. 1.1), assuming two-dimensional search space.

(a) $\omega\,\mathbf{v}_k^i$ represents the influence of the current velocity (in the k^{th} iteration) on the next velocity (in the $(k+1)^{\text{th}}$ iteration). In other words, it represents the tendency of the particle to continue its motion in the present direction, and the parameter ω is therefore termed as the "inertia weight." In the standard PSO algorithm, ω remains constant (with respect the the iteration number).

[1]We will use bold letters to denote vectors.

(b) $\phi_1 R^i_{1k}(\mathbf{p}^i_k - \mathbf{x}^i_k)$ makes the particle move toward the best personal position it has attained so far, i.e., up to the k^{th} iteration.

(c) $\phi_2 R^i_{2k}(\mathbf{g}_k - \mathbf{x}^i_k)$ makes the particle move toward the best global position so far.

The net velocity of the particle is the vector sum of these three terms (see Fig. 1.1) and is then used to move the particle using

$$\mathbf{x}^i_{k+1} = \mathbf{x}^i_k + \mathbf{v}^i_{k+1}, \tag{1.2}$$

which is similar to $\mathbf{x}^{\text{new}} = \mathbf{x}^{\text{old}} + \mathbf{v} \times \Delta t$, with $\Delta t = 1$.

The complete PSO algorithm can now be described as follows.

1. Define the search-space boundaries (a rectangle in case of a two-dimensional search).
2. Generate a population (swarm) of particles by assigning an initial position $\mathbf{x}^i_0$ (lying within the search space) and an initial velocity $\mathbf{v}^i_0$ to each particle.
3. Set PSO iteration number $k = 0$.
4. Set a criterion for termination of the PSO iterations, e.g., we may want to terminate the iterations when the particles attain a certain average fitness value or when the number of iterations exceeds a pre-specified number.
5. Compute the fitness of each particle.
6. Check if the termination criterion is satisfied; exit the PSO loop if it is.
7. Find the personal best position $\mathbf{p}^i_k$ for each particle and the global best position $\mathbf{g}_k$ for the entire population.
8. Compute the updated velocity $\mathbf{v}^i_{k+1}$ for each particle using Eq. 1.1.
9. Move the particles according to Eq. 1.2, assign the new position $\mathbf{x}^i_{k+1}$ for each particle. If the new position for any particle lies outside the search space, place it inside the search space.
10. PSO iteration $k \leftarrow k + 1$; go to step 5.

The PSO algorithm is conceptually very simple – it is in fact inspired by social behaviour of birds in search of food. During the search process, the particles explore the search space, and their trajectories get biased by their personal past best positions as well as the best position so far among all particles of the population. As time progresses, each particle improves its fitness. When most of the particles have attained sufficiently high fitness, the search process can be stopped, and we say that the PSO algorithm has *converged* to a solution.

The choice of the algorithm parameters (ϕ_1, ϕ_2, ω in Eq. 1.1) is important in deciding the performance of the PSO algorithm [2]. A particularly disastrous situation is "swarm explosion" in which the velocity becomes indefinitely large for all particles for certain values of ϕ_1, ϕ_2, ω. The set $\phi_1 = \phi_2 = 1.4962$, $\omega = 0.7298$ has been shown to be effective for a variety of problems [3]. This set of parameters has also been found to work well for a variety of applications considered at IIT Bombay [4]-[10]. In PSOFT, the above values are therefore used by default, and the user is given a choice to change them if required.

1.2 PSO Examples

We will now illustrate the ideas discussed in Sec. 1.1 with the help of specific examples. In this section, we will only show how PSO works through a few plots. The examples considered here are also available in the PSOFT distribution so that the user can run them and generate additional information and plots, as explained later.

1.2.1 Intersection of two lines

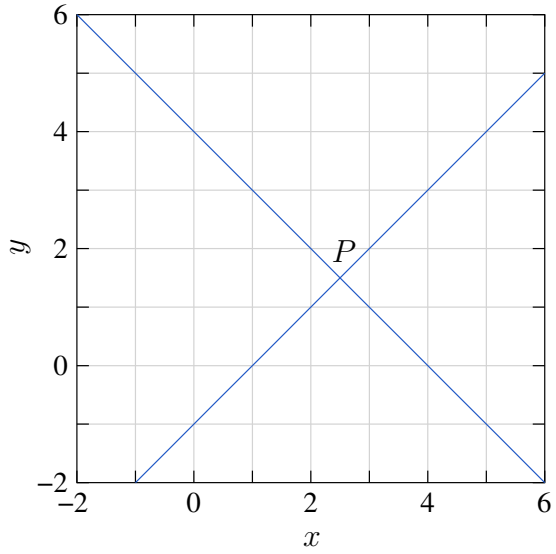

Figure 1.2: Graphs of $y = -x + 4$ and $y = x - 1$.

Consider two straight lines $y = -x + 4$ and $y = x - 1$, as shown in Fig. 1.2. We are interested in finding their intersection (point P in the figure). This problem is of course far too trivial, and the use of any optimisation program in this case would be like employing a construction-grade crane to lift a basket of flowers. However, it is a convenient problem for illustrating the functioning of the PSO algorithm, and we will view it in that spirit.

In this example, a "particle" is a point (x, y) in the search space. Let us suppose that we have an approximate idea of where the solution lies, and we define the search space accordingly as the region bounded by $x = x_1 = -1$, $x = x_2 = 5$, $y = y_1 = -1$, and $y = y_2 = 5$. We generate an initial population of 100 particles by assigning random values to the two attributes (parameters) x and y of each particle such that $x_1 < x_i < x_2$ and $y_1 < y_i < y_2$ (see Fig. 1.3).

What about the particle velocities? We would generally not have a clue about what is a reasonable number for the particle velocity, and it makes sense to initialise all velocities to zero. As the PSO iterations proceed, the particles will acquire suitable velocities anyway.

We know what a "particle" means in the context of the present example – it is specified by two parameters, x and y. How do we assign a "fitness" value to a given particle? That depends on the solution we are looking for, viz., the intersection of the two lines shown in Fig. 1.2. A particle should be considered to have a higher fitness if it is closer to the point of intersection. We cannot find the distance between a given particle and the point of intersection (which is not known). However, we can get an idea of this distance *indirectly*, as shown in Fig. 1.4. The

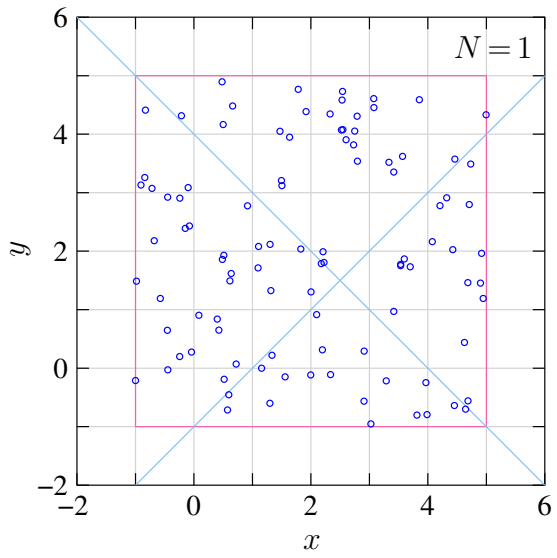

Figure 1.3: Search space (pink rectangle) and initial population of particles for the example of Sec. 1.2.1.

distances Δy_1 and Δy_2 in the figure should both be zero when the particle coincides with P, the point of intersection. The measure

$$\epsilon_i = \sqrt{\Delta y_1^2 + \Delta y_2^2} = \sqrt{[y_i - (m_1 x_i + c_1)]^2 + [y_i - (m_2 x_i + c_2)]^2}, \tag{1.3}$$

can thus be used to assess the fitness of the i^{th} particle with parameters (x_i, y_i). A particle with a smaller value of ϵ would be considered to have a higher fitness. It should be noted that, in the PSO algorithm, we need the fitness values of particles only for the purpose of ordering the particles; no computations using the above ϵ_i values are required.

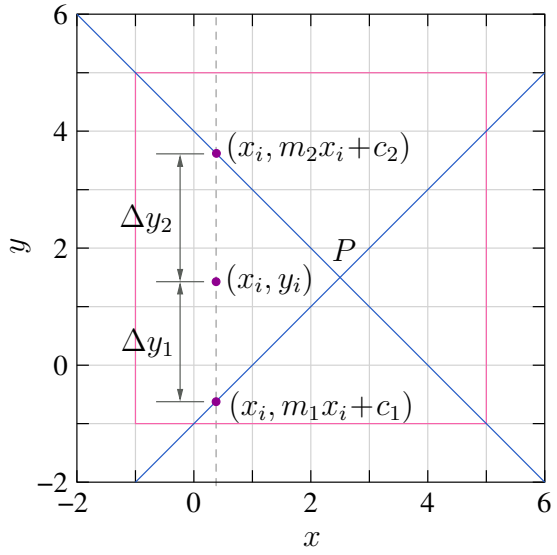

Figure 1.4: Fitness computation for the i^{th} particle with parameters x_i and y_i for the example of Sec. 1.2.1.

With the initial population, search-space definition, and a method for fitness evaluation, the PSO algorithm can now be implemented[2]. Fig. 1.4 shows how the particle positions change with time. As the PSO iterations proceed, the particles are seen to be moving closer to the solution, the point of intersection of the two lines. Fig. 1.6 (a) shows how the rms value of ϵ over the entire population varies as a function of the PSO iteration number on a linear scale, and Fig. 1.6 (b) shows the same on a logarithmic scale.

In this example, it was easy to view the evolution of the population directly (see Fig. 1.5) since we had only two parameters, viz., x and y, associated with each particle. In general, we could have more than two parameters per particle, and to assess the progress of the PSO process, we have to rely on an average measure of the closeness of the particles to the expected solution, such as $\epsilon_{\rm rms}$ in this example.

Figs. 1.7 (a) and 1.7 (b) show how the average parameter values (x and y) evolve. As the particles move toward the solution, the average values of x and y approach 2.5 and 1.5, respectively, i.e., the solution we are seeking.

[2]In this example, we will perform a fixed number of PSO iterations. In practice, it may be desirable to stop the PSO iterations when a suitable termination criterion is satisfied.

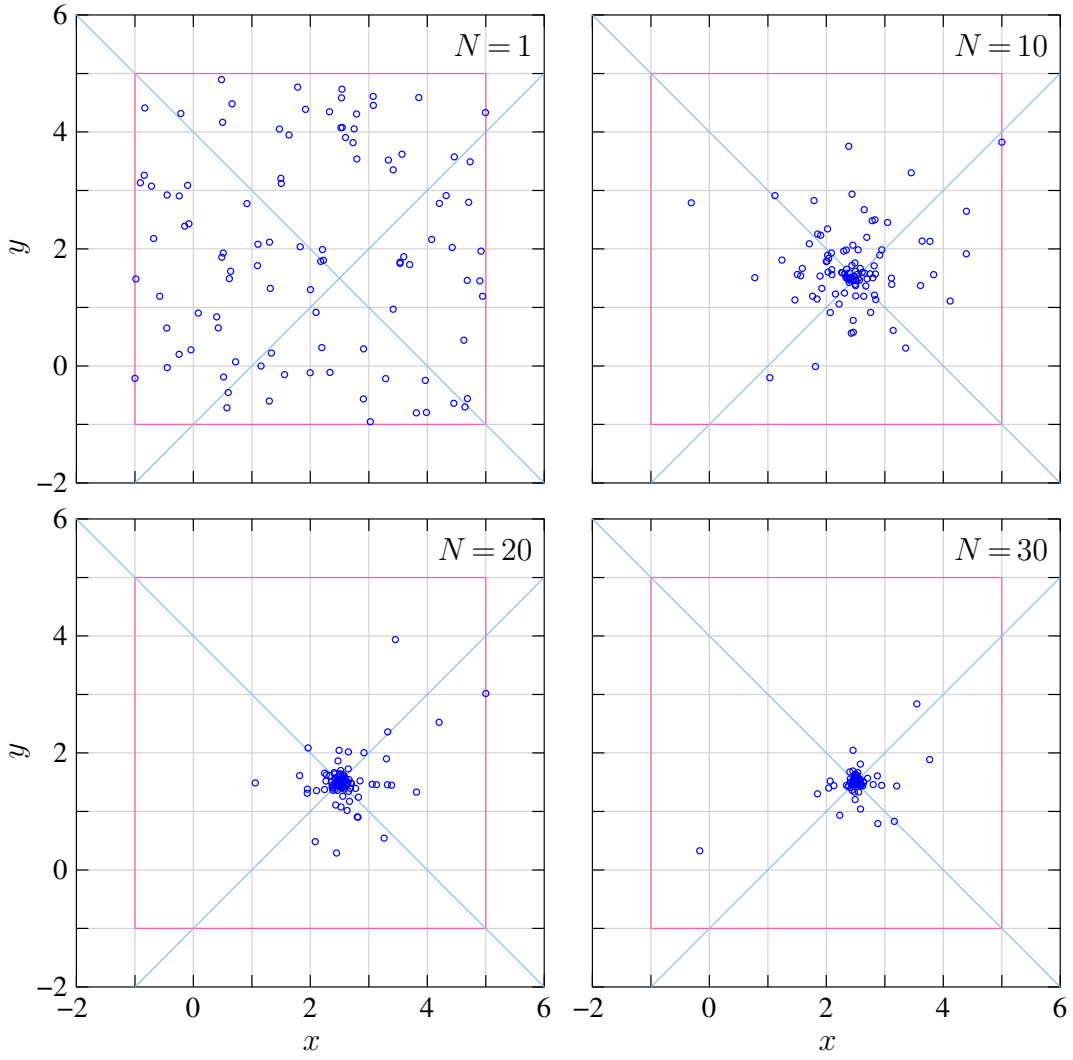

Figure 1.5: Distribution of particles for the example of Sec. 1.2.1 during the PSO process. N denotes the PSO iteration number.

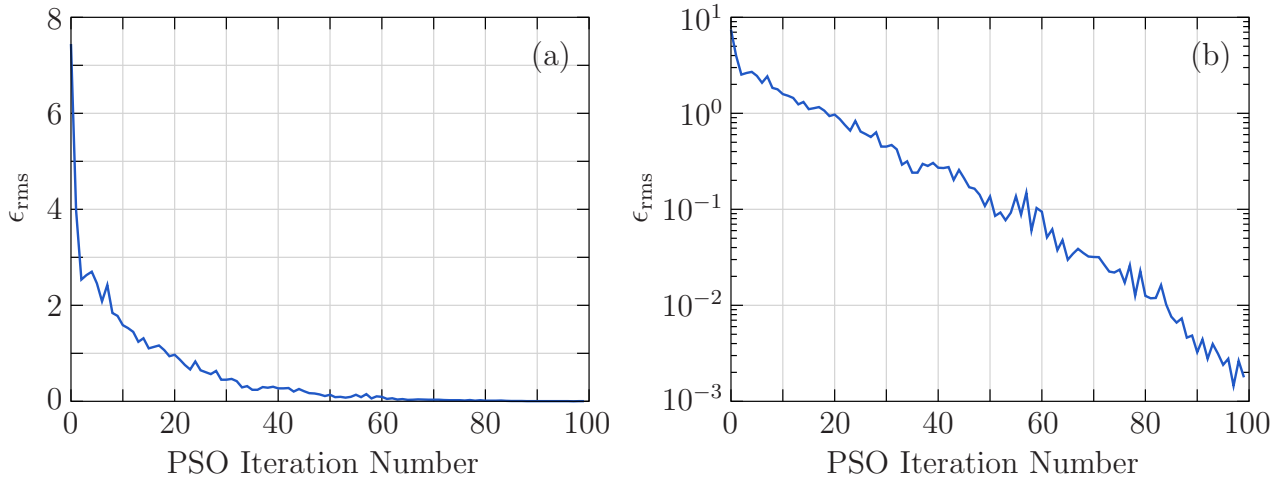

Figure 1.6: ϵ_{rms} for the example of Sec. 1.2.1 as a function of PSO iteration number: (a) linear scale, (b) logarithmic scale.

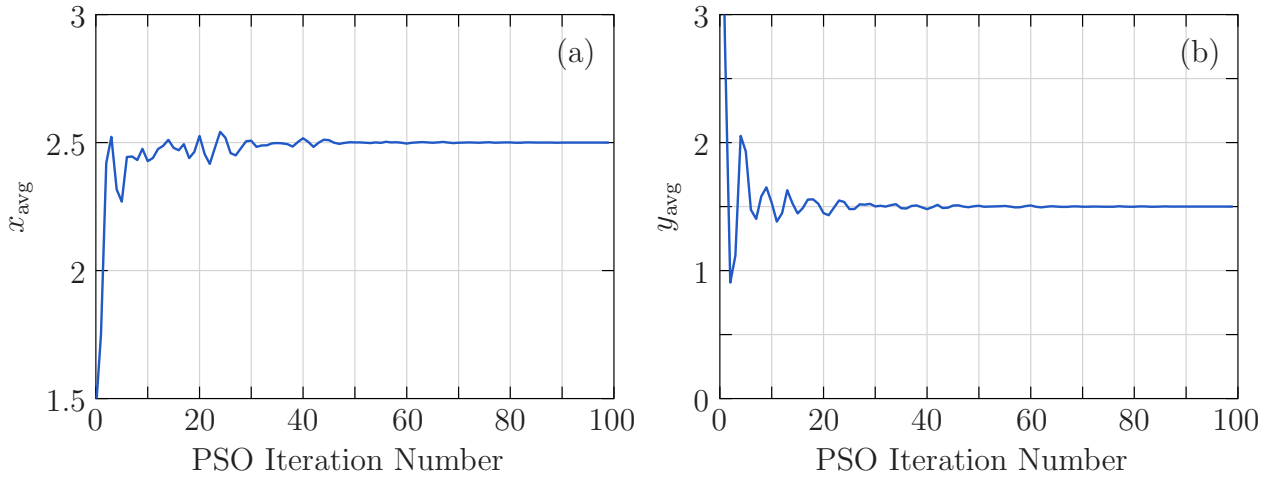

Figure 1.7: Evolution of average parameter values for the example of Sec. 1.2.1: (a) x_{avg}, (b) y_{avg}.

1.2.2 Design of RLC circuit for a specified frequency response

We now consider a design problem. The frequency response, i.e., $|\mathbf{I}|$ versus frequency for a series RLC circuit (see Fig. 1.8) is given, and we want to use PSO to obtain the circuit parameters, viz., R, L, C, which will satisfy the frequency response specification. This exercise can be carried out analytically by solving equations (in terms of R, L, C) for the resonance frequency, $|\mathbf{I}|_{\text{max}}$, and the bandwidth. However, our purpose here is to demonstrate the use of PSO in the design process, and we will therefore not take the analytical route. The frequency response shown in Fig. 1.8 (b) has been obtained by computing $|\mathbf{I}|$ at different frequencies, with $R = 10\,\Omega$, $L = 1\,\text{mH}$, and $C = 1\,\mu\text{F}$. We will compare the solution given by the PSO algorithm with these values.

For this example, our "particle" is the series RLC circuit, and the position vector $\mathbf{x}^i$ for the i^{th} particle has three components: the R, L, C values. The search space is therefore three-dimensional. We define the boundaries of the search space as $0.1\,\Omega < R < 10^3\,\Omega$, $0.01\,\text{mH} < L < 10\,\text{mH}$, $0.001\,\mu\text{F} < C < 10\,\mu\text{F}$, and generate an initial population of 100

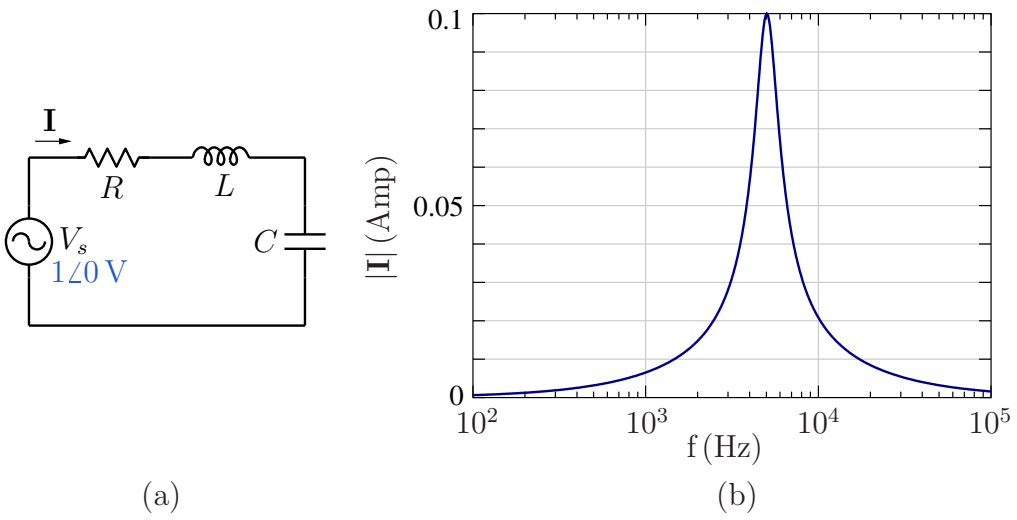

Figure 1.8: (a) Series RLC circuit, (b) Frequency response specification.

particles within this search space. Since the parameter ranges used are relatively wide, we generate uniform random numbers in the interval $(\log p_{\min}, \log p_{\max})$ – where p stands for R, L, or C – rather than in $(p_{\min}, p_{\max})$. The distribution of the initial L values is shown in Fig. 1.9 as an example.

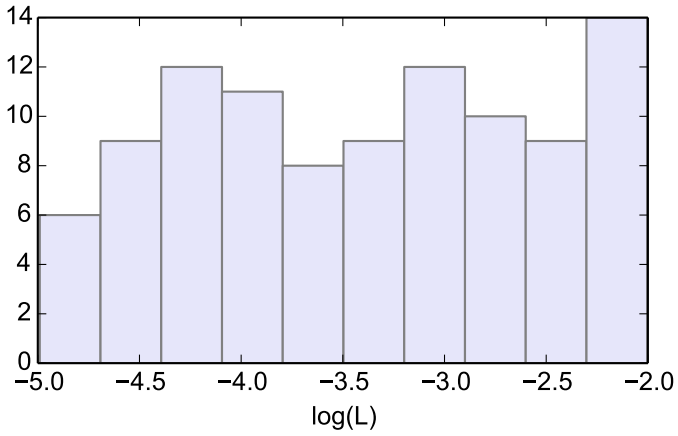

Figure 1.9: Histogram showing the initial distribution of L values in the RLC example of Sec. 1.2.2.

To evaluate the fitness of a particle, we generate the frequency response ($|\mathbf{I}|$ at the frequencies of interest) using the R, L, C values of the particle and compute

$$\epsilon_{\text{rms}} = \sqrt{\frac{1}{N_f} \sum_{j=1}^{N_f} \left(|\mathbf{I}|_j - |\mathbf{I}^{\text{ref}}|_j \right)^2}, \tag{1.4}$$

where $|\mathbf{I}|_j$ is the current magnitude for the given particle at frequency f_j, $|\mathbf{I}^{\text{ref}}|_j$ is the current magnitude in the frequency response specification (also at frequency f_j), and N_f is the total number of frequencies used in the frequency response specification. A particle with a smaller ϵ_{rms} is considered to have a higher fitness value.

It should be obvious that fitness evaluation in this example is computationally more expensive than in the previous example (compare Eqs. 1.3 and 1.4), and the PSO program would therefore take longer for the same number of particles and PSO iterations. In this example, the quantity of interest $|\mathbf{I}|_j$ can be computed analytically. However, in some situations, use of a circuit simulator may be required for fitness computation, and that would make the process substantially slower. Fitness evaluation is clearly the most expensive operation in each PSO iteration, the other operations, viz., computation of particle velocity (Eq. 1.1) and position update (Eq. 1.2), being relatively trivial.

Fig. 1.10 shows the evolution of the average values of the particle parameters. The parameter values are seen to converge to the expected solution, viz., $R = 10\,\Omega$, $L = 1\,\text{mH}$, and $C = 1\,\mu\text{F}$. Fig. 1.11 shows how the average value of ϵ_{rms} varies with the number of PSO iterations.

Histograms of the particle parameters are useful for visualising the progress of the PSO iterations. As an example, Fig. 1.12 shows a histogram for the L parameter after 50 PSO iterations. We observe that, at this stage, most of the particles have already attained the expected L value, viz., $L = 10^{-3}\,\text{H}$ (i.e., $\log L = -3$).

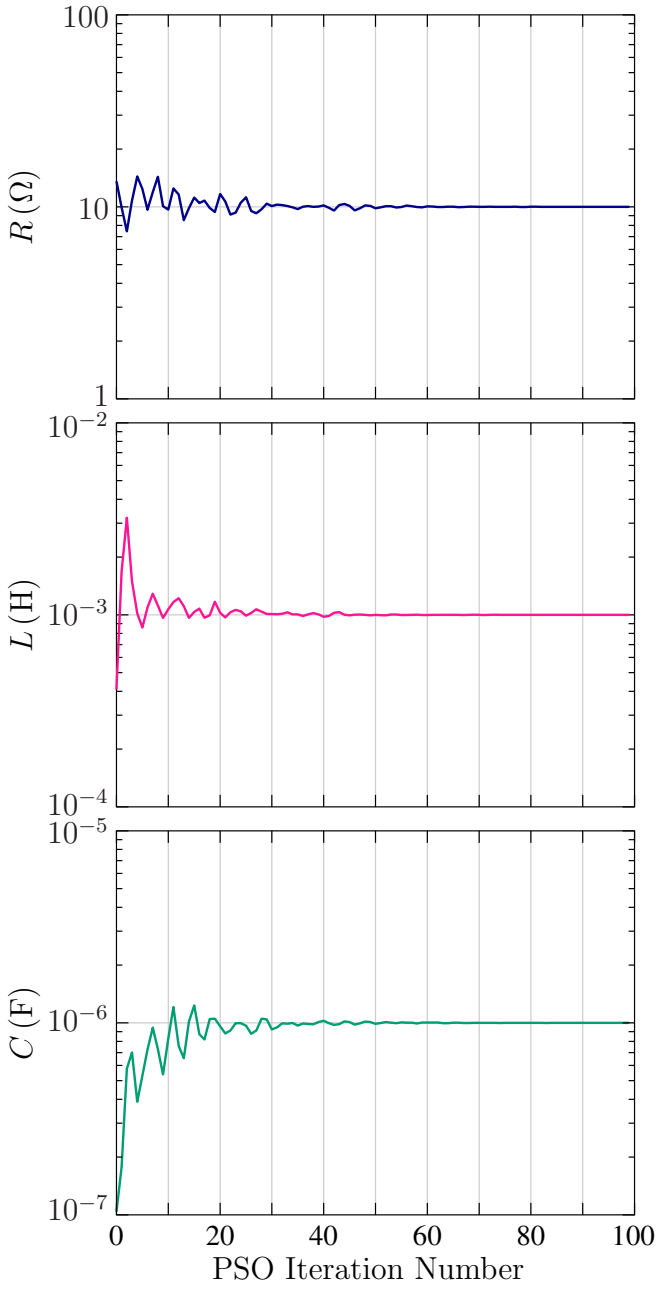

Figure 1.10: Average values of particle parameters versus PSO iteration number for the *RLC* example of Sec. 1.2.2.

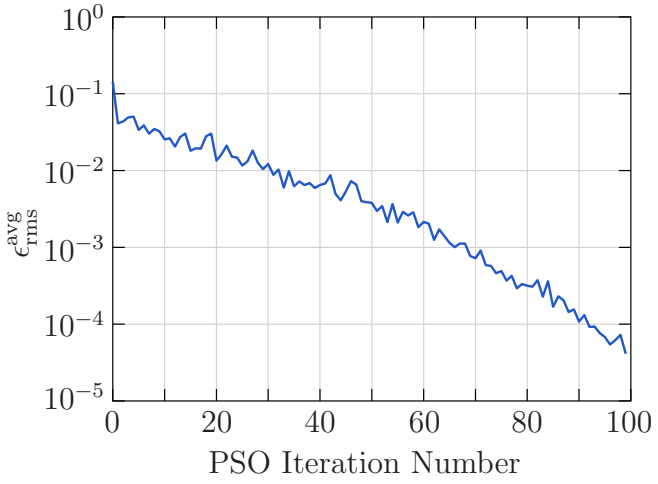

Figure 1.11: Average value of $\epsilon_{\rm rms}$ versus PSO iteration number for the *RLC* example of Sec. 1.2.2.

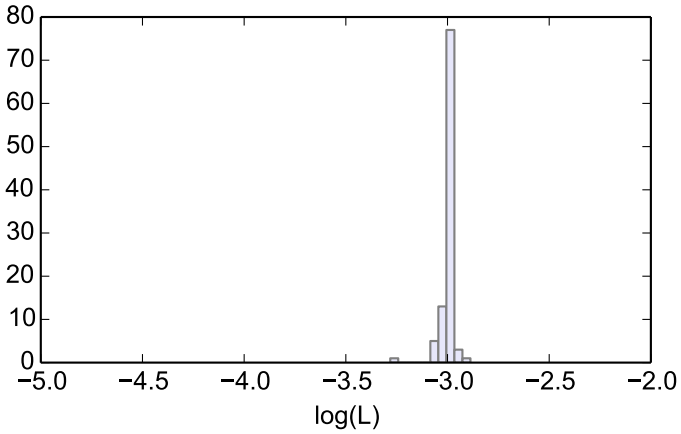

Figure 1.12: Histogram showing the distribution of L values after 50 PSO iterations in the *RLC* example of Sec. 1.2.2.

Chapter 2

PSOFT: Files and Syntax

The PSO algorithm has been used for various applications at IIT Bombay [4]-[10]. It was found to be easy to implement and yet very effective for a variety of applications. Based on this experience, the package PSOFT (Particle Swarm Optimisation with Flexible Templates) was developed to facilitate application of the PSO algorithm to new problems. In PSOFT, the PSO algorithm is decoupled from the application under consideration, thus making it easier for the user to focus on the application.

In this chapter, we describe the organisation of the PSOFT program and the syntax for the files to be supplied by the user. We will then describe the PSOFT implementation of the examples discussed in Chapter 1.

2.1 PSOFT Organisation

The organisation of the PSOFT program is shown in Fig. 2.1. The user supplies the files `particle_class.h` and `particle_class.cpp` which define the particle behaviour, and input files `pso_parms.in` and `pso_control.in`. The file `pso_parms.in` specifies the PSO algorithm parameters (ϕ_1, ϕ_2, ω in Eq. 1.1) which generally would not need to be altered from their default values. The other input file, `pso_control.in`, conveys various input values such as the number of particles, number of PSO iterations to be performed, parameter values related to fitness evaluation, names of particle parameters with their minimum and maximum limits, names and contents of the output files to be generated, and plots to be displayed at the end of the PSO loop. We will describe each of these files in more detail a little later.

The "preprocessor" (`psoprep`) creates a file `pso1.h` which defines some integer variables, based on the parameter list supplied by the user in `pso_control.in`. This will become more clear when we look at implementation of specific examples.

The user's C++ file `particle_class.cpp` is then compiled and linked with the PSO main program `pso_fixed.o`, to produce the final executable file, `pso`. The user needs to change only the problem-specific files (`particle_class.h`, `particle_class.cpp`, `pso_control.in`, and `pso_parms.in`) to solve a new problem. It is a good idea to save these four files with different names so that they remain available for later use. For example, in the PSOFT archive, there are files named `particle_class_xy.h`, `particle_class_xy.cpp`, and `pso_control_xy.in` for the example of Sec. 1.2.1. To run this example, the user can use the following commands.

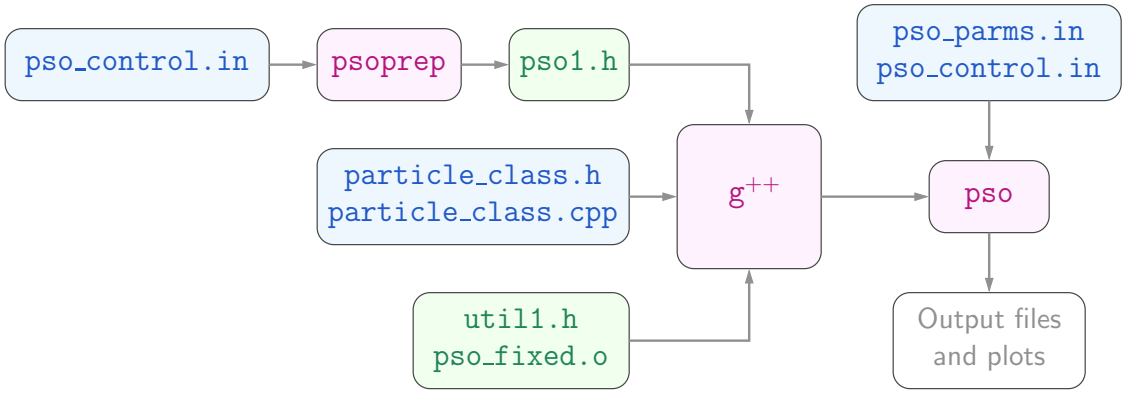

Figure 2.1: Organisation of the PSOFT package. Files to be supplied by the user are shown in blue, executable files in magenta, and other files in green.

```
cp particle_class_xy.h particle_class.h
cp particle_class_xy.cpp particle_class.cpp
cp pso_control_xy.in pso_control.in
./psoprep
g++ -O3 -c particle_class.cpp
g++ -O3 pso_fixed.o particle_class.o -o pso
./pso
```

The new files are actually used by PSOFT, and the old files remain available for future use.

2.2 Input File Syntax

As seen in Fig. 2.1, the user needs to supply two input files: `pso_parms.in` and `pso_control.in`. Before looking at the statements involved in these files, let us make a few general remarks about the syntax.

(a) Blank lines are ignored. Also, lines starting with `#` are treated as comments and ignored.

(b) The character `+` at the beginning of a line indicates continuation of the previous line.

(c) In a given line, strings are separated by one or more spaces, and in some cases, by the `=` character. In the latter case, spaces are allowed before or after the `=` character; they are ignored.

(d) A character string is denoted by `string`, an integer by `int`, and a real number by `real`.

(e) Keywords are shown in red in the description of the files below.

The input file `pso_parms.in` specifies the PSO algorithm parameters. It has the following lines.

`pso_parms.in`

```
p1 = 1.4962
p2 = 1.4962
w  = 0.7298
iseed = int
```

The parameters `p1`, `p2`, `w` represent ϕ_1, ϕ_2, ω, respectively, in Eq. 1.1. As mentioned in in Chapter 1, the parameter values mentioned above have been found to perform well for a variety of problems [3], and the user generally would not need to change these values.

The parameter `iseed` (a positive integer) is the "seed" for the random number generator used in the program. It controls the sequence of random numbers used in a specific run. If `iseed` is changed, it amounts to repeating the PSO "experiment" with a different set of random numbers. Typically, the user would not need to change this parameter, either.

The other input file, `pso_control.in`, is problem-specific, i.e., its content depends on the problem being solved. The general format of `pso_control.in` is given below.

`pso_control.in`

```
title: string(s)

n_particles=int
parameters: string1 string2 ...
fitness_int_parameters: string=int string=int ...
fitness_real_parameters: string=real string=real ...

output_format:
+  int_width=int
+  double_precision=int
+  double_width=int

plotting_program: gnuplot/python3

begin_opt
   ....
   ....
end_opt

begin_opt
   ....
   ....
end_opt
end_file
```

The first part of `pso_control.in` contains common information about the user's problem. That is followed by the optimisation block(s), starting with `begin_opt` and ending with `end_opt`. Let us look at the statements involved in the first part.

* `title: string(s)`

  Problem title.

* `n_particles=int`

  Number of particles.

* `parameters: string1 string2 ...`

  List of parameters (to be optimised).

* `fitness_int_parameters: string=int string=int ...`

  List of integer parameters related to fitness evaluation (parameter names and values).

* `fitness_real_parameters: string=real string=real ...`

  List of real parameters related to fitness evaluation (parameter names and values).

* `output_format:`
  + `int_width=int`
  + `double_precision=int`
  + `double_width=int`

  Formatting information used in writing integers (e.g., PSO iteration number) and real numbers (e.g., parameter values) to output files.

* `plotting_program: gnuplot/python3`

  Whether `gnuplot` or `python3` should be used for plotting output data.

The general form of an optimisation block is given below.

```
begin_opt
  begin_parms
    ....
    ....
  end_parms
  begin_algo_parms
    ....
    ....
  end_algo_parms
  begin_output
    ....
    ....
  end_output
  begin_plots
    ....
    ....
  end_plots
end_opt
```

Let is look at the various sub-blocks and statements within the optimisation block.

2.2.1 Parameter definition blocks

The optimisation block starts with the particle parameter block, starting with `begin_parms` and ending with `end_parms`. The purpose of this block is to specify various attributes for each particle parameter, and it has the following structure.

```
begin_parms
    parm_1 type=string string1=string2 ...
    parm_2 type=string string1=string2 ...
    ....
    ....
end_parms
```

`parm_1`, `parm_2`, etc. are the particle parameter names, and they must appear in the same order as that in the `parameters:` statement seen earlier (see p. 15). Immediately after the parameter name, the `type=string` assignment must appear. It can take one of the following forms.

* `type=free`

  The parameter is allowed to vary between $x^{(i)}_{\min}$ and $x^{(i)}_{\max}$ which are specified by the user. The superscript i denotes the parameter number.

* `type=hold`

  The parameter is allowed to vary between $k^{(i)}_{\min}x^{(i)}_0$ and $k^{(i)}_{\max}x^{(i)}_0$ where $k^{(i)}_{\min}$, $k^{(i)}_{\max}$, and $x^{(i)}_0$ are specified by the user.

* `type=previous`

  The parameter is allowed to vary between $k^{(i)}_{\min}x^{(i)}_g$ and $k^{(i)}_{\max}x^{(i)}_g$ where $k^{(i)}_{\min}$ and $k^{(i)}_{\max}$ are specified by the user, and $x^{(i)}_g$ is the value of the i^{th} parameter for the globally best particle in the previous optimisation block.

* `type=file`

  The parameter is allowed to vary between $k^{(i)}_{\min}x^{(i)}_{\text{file}}$ and $k^{(i)}_{\max}x^{(i)}_{\text{file}}$ where $k^{(i)}_{\min}$ and $k^{(i)}_{\max}$ are specified by the user, and $x^{(i)}_{\text{file}}$ is taken from a file specified by the user.

After the `type=string` assignment, we have assignments of the form `string1=string2` where `string1` and `string2` take different values, as listed below. Note that only some of these assignments are required, depending on the type of the parameter (`free`, `hold`, `previous`, or `file`).

* `log=yes/no`

  Suppose the parameter of interest is $x^{(i)}$, with limits $x^{(i)}_{\min}$ and $x^{(i)}_{\max}$.

  If `log=yes` is specified, then the initial values (at the beginning of the PSO loop) are assigned such that the distribution of $\log x^{(i)}$ is uniform between $\log x^{(i)}_{\min}$ and $\log x^{(i)}_{\max}$ (where $x^{(i)}_{\min}$ and $x^{(i)}_{\max}$ are expected to be positive). This option is useful when a wide range of values is to be explored for the i^{th} parameter.

  If `log=no` is specified, the distribution of $x^{(i)}$ is made uniform between $x^{(i)}_{\min}$ and $x^{(i)}_{\max}$ in the initial assignment.

  Note that, apart from affecting the initial distribution, $x^{(i)}_{\min}$ and $x^{(i)}_{\max}$ also serve to define the search space during the PSO process.

* `min=real`

  (required if `type` is `free`)

  Specifies the minimum allowed value $x^{(i)}_{\min}$ for the i^{th} parameter.

* `max=real`

  (required if `type` is `free`)

  Specifies the maximum allowed value $x^{(i)}_{\max}$ for the i^{th} parameter.

* `value_hold=real`

(required if `type` is `hold`)

If `type` is `hold`, this assignment gives $x_0^{(i)}$ (see the description of the `type=hold` option).

* `file=string`

(required if `type` is `file`)

If `type` is `file`, this assignment specifies the name of the file to be used for assigning $x_{\text{file}}^{(i)}$ (see the description of the `type=file` option). We will soon see how the file in question is to be generated.

* `k_min=real`

(required if `type` is `hold/previous/file`)

Specifies $k_{\min}^{(i)}$ for the i^{th} parameter (see the description of the `type=hold/previous/file` options).

* `k_max=real`

(required if `type` is `hold/previous/file`)

Specifies $k_{\max}^{(i)}$ for the i^{th} parameter (see the description of the `type=hold/previous/file` options).

* `user_init=yes/no`

If `user_init=yes` is specified, the initialisation of particle parameter values (at the beginning of the PSO loop) is performed by the user's routine, `user_init_parms` in file `particle_class.cpp`.

* `user_limit=yes/no`

If `user_limit=yes` is specified, the limiting of parameter values during the PSO process is performed by the user's routine, `user_limit_parms` in file `particle_class.cpp`.

* `round_to_int=yes/no`

If `round_to_int=yes` is specified, the parameter value is converted to integer before writing to output files meant for best particle and rank information.

That brings us to the end of the particle parameter block. After this block, we have the algorithm parameter block, starting with `begin_algo_parms` and ending with `end_algo_parms`, with the following statements.

* `itmax_pso=int`

(required) It specifies the number of PSO iterations to be performed.

The keyword `iter_last` can be used in the subsequent statements to refer to the value assigned to `itmax_pso`.

* `flag_fitness_computation=int`

(optional) `flag_fitness_computation` can be specified if the user wants to try out different fitness computation options. This flag is passed to `particle_class.cpp`.

2.2.2 Output blocks

With the information supplied in the particle parameter block and the algorithm parameter block, the program has all the required inputs to perform the PSO iterations[1]. The rest of the optimisation block contains the following (optional) blocks.

(a) Output block: This block starts with `begin_output` and ends with `end_output`. It conveys to the program the names of the output files to be generated and what information is to be written to each of the output files.

(b) Plotting block: This block starts with `begin_plots` and ends with `end_plots`. It tells the program how many plots are to be shown (when the PSO loop is completed), and what information is to be presented in each plot.

To display the plots requested by the user, the program prepares `python` or `gnuplot` script files and then calls `python3` or `gnuplot` for the actual plotting.

We should point out here that the use of a plotting block is optional. For example, if the user does not have `python3` or `gnuplot` installed, she can still make use of the output files generated by the output block, and view the data using other plotting programs such as `xmgrace`.

We now look at the syntax of the output block. The overall structure of the output block is shown below.

```
begin_output
  begin_file
    ....
    ....
  end_file
  begin_file
    ....
    ....
  end_file
  ....
  ....
end_output
```

The output block consists of one or more file blocks, each starting with `begin_file` and ending with `end_file`. A file block is meant to convey (a) the name of the output file to be created and (b) what information should be written to the output file. The syntax for the various types of file blocks is given below.

1.
```
begin_file
   type=best_particle
   filename=string
   variables: string1 string2 ...
   control: string1=string2 ...
end_file
```

[1]The other crucial component is of course the particle behaviour, supplied by the user in `particle_class.cpp`. We will look at that separately.

2. ```
   begin_file
      type=avg_value
      filename=string
      variables: string1 string2 ...
      control: string1=string2 ...
   end_file
   ```

3. ```
   begin_file
      type=track
      filename=string
      variables: string1 string2 ...
      control: string1=string2 ...
      particles: int1 int2 ...
   end_file
   ```

4. ```
   begin_file
      type=snapshot
      filename=string
      variables: string1 string2 ...
      control: string1=string2 ...
   end_file
   ```

5. ```
   begin_file
      type=fitness_stats
      filename=string
      control: string1=string2 ...
   end_file
   ```

6. ```
   begin_file
      type=ranks
      filename=string
      variables: string1 string2 ...
   end_file
   ```

7. ```
   begin_file
      type=gbest_final
      filename=string
   end_file
   ```

8. ```
   begin_file
      type=gvalues
      filename=string
   end_file
   ```

9. ```
   begin_file
      type=user_file
      filename=string
      control: string1=string2 ...
   end_file
   ```

Let us first look at the `type=string` statement and then the syntax of the other statements of a file block.

* `type=best_particle`

  Write parameter values of the globally best particle.

* `type=avg_value`

  Write the average parameter values (averaged over all particles) at regular intervals.

* `type=track`

  Write parameter values of specified particles at regular intervals.

* `type=snapshot`

  Write parameter values of all particles after the specified PSO iteration.

* `type=fitness_stats`

  Write information related to the average fitness (over all particles).

* `type=ranks`

  Write the rank (in terms of fitness) and parameter values of all particles at the end of the PSO loop.

* `type=gbest_final`

  Write all parameter values of the globally best particle at the end of the PSO loop. The output file created using this option is used to provide the initial parameter value when the option `type=file` is employed in the particle parameter block (see Sec. 2.2.1).

* `type=gvalues`

  Write values of special variables called "`gvalue`" (defined by the user in `particle_class.cpp`) to the output file.

* `type=user_file`

  Write user-specified information to the output file after the specified PSO iteration.

We now describe the syntax of the other statements in the file block.

* `filename=string file_format=cols/rows`

  (required)

  Specifies the name of the output file to be created.

  In addition, the `filename` statement is also used to specify (optionally) the format of the file – column-wise or row-wise – when the file type is `best_particle`, `avg_value`, `snapshot`, `ranks`, or `gvalues`. The default option is `cols`.

  For other file types, the file format is fixed and cannot be specified by the user.

* `variables: string1 string2 ...`

(required for `type=best_particle`, `avg_value`, `track`, `snapshot`, `ranks`)

Specifies which parameter values are to be written to the output file. `variables: ALL` can be used to specify all parameters.

The `variables:` statement, when required, must appear immediately after the `filename` statement.

* `particles: int1 int2 ...`

(required for `type=track`)

Specifies the particles to be tracked for the `type=track` option.

* `control: string1=string2 ...`

The `control` statement allows one or more assignments of the form `string1=string2`, where `string1` and `string2` take different values, as listed below.

- `interval=int`

  Specifies the writing interval in terms of PSO iterations. For example, if `interval` is 10, the information requested by the user to the output file once in every 10 iterations.

  (Default: 1. Relevant for `type=best_particle`, `avg_value`, `fitness_stats`, `track`)

- `write_start=int`

  Specifies the iteration number for which writing to the output file is to start.

  (Default: 1. Relevant for `type=best_particle`, `avg_value`, `fitness_stats`, `track`)

- `write_end=int`

  Specifies the iteration number for which writing to the output file is to end.

  (Default: The total number of PSO iterations specified by the statement `itmax_pso=int`. Relevant for `type=best_particle`, `avg_value`, `fitness_stats`, `track`)

- `iter_snapshot=int`

  Specifies the iteration number for which snapshot information is to written to the output file.

  (Must be specified if `type=snapshot`; otherwise, not relevant.)

- `iter_write_user=int`

  Specifies the iteration number for which the user's routine `write_user_file` (in `particle_class.cpp`) is called for writing to the output file.

  (Must be specified if `type=user_file`; otherwise, not relevant.)

- `index_user_file=int`

  Specifies an (optional) integer which can be used within the user's routine `write_user_file` (in `particle_class.cpp`).

2.2.3 Plotting blocks

When the PSO loop is completed, the output files specified in the output blocks get created, each containing information as specified in the corresponding output block. These files can be used by the user *independently* to plot, for example, the average value of a parameter or an average fitness measure as a function of the PSO iteration number. For this purpose, the user can use a plotting package of her choice.

Apart from displaying the plots of interest in the above manner, the user can also use the built-in plotting capability of PSOFT, if `python3` or `gnuplot` is installed on her computer. That can be achieved by incorporating appropriate plotting blocks in the user's input file (see p. 16). In short, the file blocks (discussed in Sec. 2.2.2) produce output files, and the plotting blocks use these output files to display the plot requested by the user. PSOFT does not do the actual plotting; all it does is to create appropriate `python` or `gnuplot` script files and then call `python3` or `gnuplot` to show the plots.

Each plotting block starts with `begin_plot` and ends with `end_plot`. All plotting blocks together need to be enclosed in an outer block starting with `begin_plots` and ending with `end_plots`, as shown below.

```
begin_plots
  begin_plots
    ....
    ....
  end_plots
  begin_plot
    ....
    ....
  end_plot
  ....
  ....
end_plots
```

The type of each plotting block is given by the `type=string` statement. In the following, we describe the syntax for different plotting block types.

1.
```
begin_plot
   type=xy_plot
   filename=string
   variables: x=string y_1=string y_2=string ...
   plot_control: string1=string2 ...
end_plot
```

2.
```
begin_plot
   type=fitness_stats
   filename=string
   variables: x=string y_1=string y_2=string ...
   plot_control: string1=string2 ...
end_plot
```

3.
```
begin_plot
   type=histogram
   filename=string
   variables: string1 string2 ...
```

```
   plot_control: string1=string2 ...
end_plot
```

4. ```
   begin_plot
      type=xy_plot_user
      filename=string
      variables: x=int y_1=int y_2=int ...
      plot_control: string1=string2 ...
   end_plot
   ```

5. ```
   begin_plot
     type=xy_plot_track
     filename=string
     variables: x=string y_1=string y_2=string ...
     plot_control: string1=string2 ...
     particles: int1 int2
   end_plot
   ```

Let us first look at the `type=string` statement and then the syntax of the other statements of a plotting block.

* `type=xy_plot`

  Display graph of the specified variables versus PSO iteration number.

* `type=fitness_stats`

  Display graph of the specified fitness-related variables versus PSO iteration number.

* `type=histogram`

  Display histogram of specified quantity at a given PSO iteration number.

* `type=xy_plot_user`

  Display graph of variables stored in a user-created file.

* `type=xy_plot_track`

  Display graph of specified parameters for specified particles versus PSO iteration number.

We now describe the syntax of the other statements in the plotting block.

* `filename=string`

  (required)

  Specifies the name of the file which is supposed to provide the data required for this plot. The file name must appear in one of the file blocks, i.e., each plotting block must have a corresponding file block, and the correspondence between the two is established by a common file name. Furthermore, the type of the plotting block and the type of the file block must be consistent, i.e., they must satisfy the rules given in the following table.

Plotting block	File block
`xy_plot`	`avg_value` or `best_particle`
`histogram`	`snapshot`
`fitness_stats`	`fitness_stats`
`xy_plot_user`	`user_file`
`xy_plot_track`	`track`

* `variables: ...`

  The syntax of this statement depends on the type of the plotting block, as explained below.

  - `variables: x=iter y_1=string y_2=string ...`

    (required for `type=xy_plot` and `type=xy_plot_track`)

    `string` (in the assignment of `y_1`, `y_2`, etc.) is the name of one of the parameters listed in the corresponding file block.

  - `variables: string`

    (required for `type=histogram`)

    `string` is the name of one of the parameters listed in the corresponding file block.

  - `variables: x=iter y_1=fitness_int y_2=fitness_int ...`

    (required for `type=fitness_stats`)

    `int` indicates which fitness parameter (starting with 1) is to be plotted.

  - `variables: x=int y_1=int y_2=int ...`

    (required for `type=xy_plot_user`)

    `int` indicates the column number in the output file created by the user.

* `particles: int1 int2 ...`

  (required for `type=xy_plot_track`)

  `int1`, `int2`, etc. specify the particles for which the plot is to be displayed.

* `plot_control: string1=string2 ...`

  The `plot_control` statement allows one or more assignments of the form `string1=string2`, where `string1` and `string2` take different values, as listed below.

  - `xlog=yes/no`

    (Default: `no`)

    Specifies if the x-axis should be logarithmic.

  - `ylog=yes/no`

    (Default: `no`)

    Specifies if the y-axis should be logarithmic.

- `wait=yes/no`

  (Default: `no`)

  Specifies whether the program should wait (and ask the user whether to continue) after this particular plot has been dispayed.

- `title=string`

  Specifies the title of the plot (to be displayed at the top of the plot window). If the title is not specified, the program generates a default title.

- `show_plotfile_file=yes/no`

  (Default: `yes`)

  Specifies if the name of the `python` or `gnuplot` file created for this particular plot should be displayed as part of the plot title. It is convenient if the user wishes to edit the `python` or `gnuplot` file to change fonts, add some embellishments to the plot, etc.

- `xmin=real`

  Specifies the lower limit for the x-axis. If `xmin` is not specified, the program would allow `python` or `gnuplot` to decide on a suitable lower limit, based on the data being plotted.

- `xmax=real`

  Specifies the upper limit for the x-axis. If `xmax` is not specified, the program would allow `python` or `gnuplot` to decide on a suitable upper limit, based on the data being plotted.

- `n_bins=int`

  (Default: 10)

  Number of bins to be used for plot type `histogram`.

That completes our discussion of the input file syntax. Several options and rules have been described, and at this stage, it could be somewhat confusing. In the next chapter, we will look at the files for solving the optimisation problems discussed in Chapter 1. The usage of the various statements would then become clear.

Chapter 3

Files for Specific Examples

We have looked at some examples in Chapter 1 to understand how the PSO algorithm works. We have also seen the syntax of the various files involved in PSOFT in Chapter 2. In this chapter, we will go through the files used in solving a few specific optimisation problems.

As explained in Chapter 2, there are four files to be supplied by the user:

* `pso_parms.in`
* `pso_control.in`
* `particle_class.h`
* `particle_class.cpp`

The PSO algorithm paramters are given by `pso_parms.in` as already discussed in Chapter 2, and the user would generally not have to make any changes in this file. We will describe in the following the three other files in the context of different examples.

3.1 Intersection of lines

The problem statement and related plots for this example have been described in Sec. 1.2.1 and not repeated here. The files related to this example (and others) can be found in the `/examples` directory, with `_xy` in the file names. For convenience, we will reproduce the files here and include comments at appropriate places to explain the purpose of specific statements. The reader is encouraged to relate the statements to the various details described in Sec. 1.2.1.

3.1.1 `pso_control_xy.in`

```
title: intersection of two lines

# Obtain the intersection of the lines,
# y1 = m1*x1 + c1, y2 = m2*x2 + c2, using PSO.

# number of particles

n_particles = 100

# names of particle parameters
```

```
parameters: x y

# integer parameters related to fitness evaluation

fitness_int_parameters:

# real parameters related to fitness evaluation

fitness_real_parameters:
+  m1=-1.0e0
+  c1= 4.0e0
+  m2= 1.0e0
+  c2=-1.0e0

# format specification for integers and real numbers
# (in output files)

output_format: int_width=5 double_precision=6 double_width=13
plotting_program: gnuplot

begin_opt
  begin_parms
    x type=free min=-1.0 max=4.0
    y type=free min=-1.0 max=10.0
  end_parms
  begin_algo_parms
    itmax_pso = 100
  end_algo_parms
  begin_output
    begin_file
      type=best_particle
      filename=best.dat file_format=cols
      variables: ALL
      control: interval=10
    end_file
    begin_file
      type=avg_value
      filename=avg.dat file_format=rows
      variables: x y
      control: interval=1
    end_file
    begin_file
      type=track
      filename=track.dat
      variables: x y
      control: interval=1
      control: write_start=1 write_end=10
      particles: 10 12 15
    end_file
    begin_file
      type=gbest_final
      filename=gbest_final.dat
    end_file
    begin_file
```

```
    type=snapshot
    filename=snap1.dat file_format=cols
    variables: x y
    control: iter_snapshot=40
  end_file
  begin_file
    type=fitness_stats
    filename=err.dat file_format=cols
    control: interval=1
  end_file
  begin_file
    type=ranks
    filename=rank1.dat file_format=rows
    variables: x y
  end_file
end_output
begin_plots
  begin_plot
    type=xy_plot
    filename=best.dat
    variables: x=iter y_1=y y_2=x
    plot_control: xlog=no ylog=no wait=no
    plot_control: title=best_particle_parms(1)
  end_plot
  begin_plot
    type=xy_plot
    filename=avg.dat
    variables: x=iter y_1=x y_2=y
    plot_control: xlog=no ylog=no wait=no
    plot_control: title=avg(1)
  end_plot
  begin_plot
    type=xy_plot_track
    filename=track.dat
    variables: x=iter y_1=x y_2=y
    plot_control: xlog=no ylog=no wait=no
    plot_control: title=track(1)
    particles: 10 12
  end_plot
  begin_plot
    type=histogram
    filename=snap1.dat
    variables: y
    plot_control: xlog=no wait=no
    plot_control: xmin=-2 xmax=5
    plot_control: n_bins=35
    plot_control: title=y(iter.eq.40)
  end_plot
  begin_plot
    type=fitness_stats
    filename=err.dat
    variables: x=iter y_1=fitness_1
    plot_control: xlog=no ylog=no wait=no show_plotfile_name=no
    plot_control: title=error(1)
  end_plot
```

```
  end_plots
end_opt

end_file

begin_opt
  begin_parms
    x type=free  log=no min=0 max=4.0
    y type=file  log=no k_min=0.9 k_max=1.1 filename=gbest_final.dat
  end_parms
  begin_algo_parms
    itmax_pso = 100
  end_algo_parms
  begin_output
    begin_file
      type=best_particle
      filename=best1.dat
      variables: ALL
      control: interval=10
    end_file
    begin_file
      type=avg_value
      filename=avg1.dat
      variables: x y
      control: interval=1
    end_file
    begin_file
      type=fitness_stats
      filename=err1.dat
      control: interval=1
    end_file
  end_output
  begin_plots
#   begin_plot
#     type=xy_plot
#     filename=best1.dat
#     variables: x=iter y_1=x y_2=y
#     plot_control: xlog=no ylog=no wait=no
#     plot_control: title=best_particle_parms(2)
#   end_plot
    begin_plot
      type=xy_plot
      filename=avg1.dat
      variables: x=iter y_1=x y_2=y
      plot_control: xlog=no ylog=no wait=no
      plot_control: title=avg(2)
    end_plot
    begin_plot
      type=fitness_stats
      filename=err1.dat
      variables: x=iter y_1=fitness_1
      plot_control: xlog=no ylog=no wait=no
      plot_control: title=error(2)
    end_plot
  end_plots
```

```
end_opt
end_file
```

3.1.2 particle_class_xy.h

```
#ifndef PARTICLE_CLASS_H
#define PARTICLE_CLASS_H

#include <iostream>
#include <cstring>
#include <cstdlib>
#include <fstream>
#include <sstream>
#include <complex>
#include <iomanip>
#include <math.h>

// File pso1.h gets created by the preprocessor (psoprep), using
// the information from pso_control.in.
// It has definitions of nr_x, nr_y, nfr_m1, nfr_c1, nfr_m2, nfr_c2.

#include "pso1.h"

using namespace std;

class particle_class {

public:

// class variables:

   double x,y;
   double m1,c1,m2,c2;

public:

// function prototypes:

particle_class(
   const int n_particles,
   double **p_rparm,
   int    *f_iparm,
   double *f_rparm,
   int& n_gvalue,
   int& n_fitness_print,
   int& flag_fitness);

void get_fitness(
   const int iter_pso,
   const int flag_fitness_computation,
   const int n_particles,
   bool *flag_fit,
   double **p_rparm,
   int    *f_iparm,
   double *f_rparm,
```

```
   double *fvalue,
   const int n_gvalue,
   double **gvalue,
   double* fitness_print,
   int& flag_fitness,
   int& flag_converged);

void write_user_file(
   const int n_particles,
   bool *flag_fit,
   const int flag_fitness_computation,
   double **p_rparm,
   int    *f_iparm,
   double *f_rparm,
   double *fvalue,
   const int n_gvalue,
   double **gvalue,
   double* fitness_print,
   int& flag_fitness,
   const int index_user_file,
   std::ofstream& outf,
   const int n_ranks,
   int *n_rank_particles,
   int **rank_index);

void user_init_parms(
   const int n_particles,
   bool *flag_fit,
   const int n_rparms,
   int *flag_user_init,
   double **p_rparm,
   double *rparm_min,
   double *rparm_max);

void user_limit_parms(
   const int n_particles,
   bool *flag_fit,
   const int n_rparms,
   int *flag_user_limit,
   double **p_rparm,
   double **p_vel,
   double *rparm_min,
   double *rparm_max);

};
#endif
```

3.1.3 particle_class_xy.cpp

```
#include "particle_class.h"

particle_class::particle_class(
   const int n_particles,
   double **p_rparm,
   int    *f_iparm,
```

```
   double *f_rparm,
   int& n_gvalue,
   int& n_fitness_print,
   int& flag_fitness) {

// Use this part for "one-time" allocations, assignments, and computations.

   flag_fitness = flag_minimise;

// n_fitness_print is the number of fitness measures we are interested in.

   n_fitness_print = 2;

// number of user-defined "special" variables to be written to output file(s)

   n_gvalue = 0;

// f_rparm: parameters used in fitness evaluation. These are NOT
// particle parameters. The indices nfr_m1, etc. are supplied by
// pso1.h.

   m1 = f_rparm[nfr_m1];
   c1 = f_rparm[nfr_c1];
   m2 = f_rparm[nfr_m2];
   c2 = f_rparm[nfr_c2];

   return;
}
// ------------------------------------------------------------------------------
void particle_class::get_fitness(
   const int iter_pso,
   const int flag_fitness_computation,
   const int n_particles,
   bool *flag_fit,
   double **p_rparm,
   int    *f_iparm,
   double *f_rparm,
   double *fvalue,
   const int n_gvalue,
   double **gvalue,
   double* fitness_print,
   int& flag_fitness,
   int& flag_converged) {

   double k1,k2;
   double err2,err_rms;
   double err_stop=1.0e-10;

// find fitness value (fvalue) for each particle:

   for (int i_part = 0; i_part < n_particles; i_part++) {
//   get x and y from the array p_rparm passed by the calling routine:

     x = p_rparm[i_part][nr_x];
     y = p_rparm[i_part][nr_y];
```

```
      k1 = y-(m1*x+c1);
      k2 = y-(m2*x+c2);

      fvalue[i_part] = k1*k1 + k2*k2;
    }

// check for convergence

    err2 = 0.0;

    for (int i_part = 0; i_part < n_particles; i_part++) {
      err2 = err2 + fvalue[i_part];
    }
    err_rms = sqrt(err2/((double)n_particles));

// flag_converged should be set to 1 if convergence condition is met;
// else to 0.

    if (err_rms < err_stop) {
      flag_converged = 1;
    } else {
      flag_converged = 0;
    }

// assign values to fitness measures:

    fitness_print[0] = err_rms;
    fitness_print[1] = err2;

    return;
}
// ------------------------------------------------------------------------------
void particle_class::write_user_file(
    const int n_particles,
    bool *flag_fit,
    const int flag_fitness_computation,
    double **p_rparm,
    int    *f_iparm,
    double *f_rparm,
    double *fvalue,
    const int n_gvalue,
    double **gvalue,
    double* fitness_print,
    int& flag_fitness,
    const int index_user_file,
    std::ofstream& outf,
    const int n_ranks,
    int *n_rank_particles,
    int **rank_index) {

// not used in this example
    return;
}
// ------------------------------------------------------------------------------
```

```
void particle_class::user_init_parms(
   const int n_particles,
   bool *flag_fit,
   const int n_rparms,
   int *flag_user_init,
   double **p_rparm,
   double *rparm_min,
   double *rparm_max) {

// not used in this example
   return;
}
// ----------------------------------------------------------------------------
void particle_class::user_limit_parms(
   const int n_particles,
   bool *flag_fit,
   const int n_rparms,
   int *flag_user_init,
   double **p_rparm,
   double **p_vel,
   double *rparm_min,
   double *rparm_max) {

// not used in this example
   return;
}
```

3.2 Design of *RLC* circuit

In this problem, the frequency response of a series *RLC* circuit is specified, and we want to find R, L, C to match the same (see Sec. 1.2.2). The files related to this example can be found in the `/examples` directory, with `_rlc1` in the file names. For convenience, we reproduce the files here.

3.2.1 `pso_control_rlc1.in`

```
title: series RLC response

# Given the frequency response (magnitude of current phasor
# versus frequency), obtain R, L, C which match that response.

# number of particles

n_particles = 100

# names of particle parameters

parameters: r l c

# integer parameters related to fitness evaluation

fitness_int_parameters:
```

```
# real parameters related to fitness evaluation

fitness_real_parameters:

# format specification for integers and real numbers
# (in output files)

output_format: int_width=5 double_precision=6 double_width=13
plotting_program: gnuplot

begin_opt
# Note: log=yes is selected because of the wide the parameter
# ranges specified.

  begin_parms
    r  type=free  log=yes min=0.1     max=1.0e3
    l  type=free  log=yes min=0.01e-3 max=10.0e-3
    c  type=free  log=yes min=1.0e-9  max=10.0e-6
  end_parms
  begin_algo_parms
    itmax_pso = 100
  end_algo_parms
  begin_output
    begin_file
#     write best particle parameters once in every 10 iterations
      type=best_particle
      filename=best.dat
      variables: r l c
      control: interval=10
    end_file
    begin_file
#     write average parameter values in each iteration
      type=avg_value file_format=cols
      filename=avg.dat
      variables: r l c
      control: interval=1
    end_file
    begin_file
#     write fitness values in each iteration
      type=fitness_stats
      filename=err.dat
      control: interval=1
    end_file
    begin_file
#     write parameter values for all particles at iteration no. 1
      type=snapshot
      filename=snap1.dat
      variables: r l c
      control: iter_snapshot=1
    end_file
    begin_file
#     write parameter values for all particles at iteration no. 50
      type=snapshot
      filename=snap2.dat
      variables: r l c
```

```
      control: iter_snapshot=50
    end_file
    begin_file
#     index_user_file is used in particle_class.cpp as the particle index
#     This file block will write information (as defined in particle_class.cpp)
#     for particle 10 at iteration no. 2
      type=user_file
      filename=user1a.dat
      control: index_user_file=10
      control: iter_write_user=2
    end_file
    begin_file
#     This file block will write information (as defined in particle_class.cpp)
#     for particle 10 at iteration no. 50
      type=user_file
      filename=user1b.dat
      control: index_user_file=10
      control: iter_write_user=50
    end_file
    begin_file
#     This file block will write information (as defined in particle_class.cpp)
#     for particle 11 at iteration no. 2
      type=user_file
      filename=user2a.dat
      control: index_user_file=11
      control: iter_write_user=2
    end_file
    begin_file
#     This file block will write information (as defined in particle_class.cpp)
#     for particle 11 at iteration no. 50
      type=user_file
      filename=user2b.dat
      control: index_user_file=11
      control: iter_write_user=50
    end_file
    begin_file
#     write rank of each particle and its parameter values
#     at the end of the PSO loop
      type=ranks
      filename=rank1.dat
      variables: r l c
    end_file
  end_output

  begin_plots
    begin_plot
      type=xy_plot
      filename=avg.dat
      variables: x=iter y_1=r
      plot_control: xlog=no ylog=yes wait=no
      plot_control: title=avg(r)
    end_plot
    begin_plot
      type=xy_plot
      filename=avg.dat
```

```
  variables: x=iter y_1=l
  plot_control: xlog=no ylog=yes wait=no
  plot_control: title=avg(l)
end_plot
begin_plot
  type=xy_plot
  filename=avg.dat
  variables: x=iter y_1=c
  plot_control: xlog=no ylog=yes wait=no
  plot_control: title=avg(c)
end_plot
begin_plot
  type=xy_plot_user
  filename=user1a.dat
  variables: x=1 y_1=2 y_2=3
  plot_control: xlog=yes ylog=no wait=no
  plot_control: title=user1a.dat
end_plot
begin_plot
  type=xy_plot_user
  filename=user1b.dat
  variables: x=1 y_1=2 y_2=3
  plot_control: xlog=yes ylog=no wait=no
  plot_control: title=user1b.dat
end_plot
begin_plot
  type=xy_plot_user
  filename=user2a.dat
  variables: x=1 y_1=2 y_2=3
  plot_control: xlog=yes ylog=no wait=no
  plot_control: title=user2a.dat
end_plot
begin_plot
  type=xy_plot_user
  filename=user2b.dat
  variables: x=1 y_1=2 y_2=3
  plot_control: xlog=yes ylog=no wait=no
  plot_control: title=user2b.dat
end_plot
begin_plot
  type=histogram
  filename=snap1.dat
  variables: l
  plot_control: xlog=yes wait=no
  plot_control: xmin=0.01e-3 xmax=10.0e-3
  plot_control: title=l(iter.eq.1)
end_plot
begin_plot
  type=histogram
  filename=snap2.dat
  variables: l
  plot_control: xlog=yes wait=no
  plot_control: xmin=0.01e-3 xmax=10.0e-3
  plot_control: title=l(iter.eq.50)
end_plot
```

```
    begin_plot
      type=fitness_stats
      filename=err.dat
      variables: x=iter y_1=fitness_1
      plot_control: xlog=no ylog=yes wait=no
      plot_control: title=error
    end_plot
  end_plots
end_opt
```

3.2.2 particle_class_rlc1.h

```
#ifndef PARTICLE_CLASS_H
#define PARTICLE_CLASS_H

#include <iostream>
#include <cstring>
#include <cstdlib>
#include <fstream>
#include <sstream>
#include <complex>
#include <iomanip>
#include <math.h>

// File pso1.h gets created by the preprocessor (psoprep), using
// the information from pso_control.in.
// It has definitions of nr_r, nr_l, nr_c.

#include "pso1.h"

// util1.h is required since we will use some "utility" routines
// included there.

#include "util1.h"

using namespace std;

class particle_class {

public:

// class variables:

   double r,l,c;
   double err_rms;
   double err_stop;
   double twopi;

   int n_cols;
   int *col;

   double **x_ref;
   double *x;
   int n_data;
```

```
   int col_freq,col_mag,col_phase;

public:

// function prototypes:

particle_class(
   const int n_particles,
   double **p_rparm,
   int    *f_iparm,
   double *f_rparm,
   int& n_gvalue,
   int& n_fitness_print,
   int& flag_fitness);

void get_fitness(
   const int iter_pso,
   const int flag_fitness_computation,
   const int n_particles,
   bool *flag_fit,
   double **p_rparm,
   int    *f_iparm,
   double *f_rparm,
   double *fvalue,
   const int n_gvalue,
   double **gvalue,
   double* fitness_print,
   int& flag_fitness,
   int& flag_converged);

void write_user_file(
   const int n_particles,
   bool *flag_fit,
   const int flag_fitness_computation,
   double **p_rparm,
   int    *f_iparm,
   double *f_rparm,
   double *fvalue,
   const int n_gvalue,
   double **gvalue,
   double* fitness_print,
   int& flag_fitness,
   const int index_user_file,
   std::ofstream& outf,
   const int n_ranks,
   int *n_rank_particles,
   int **rank_index);

void user_init_parms(
   const int n_particles,
   bool *flag_fit,
   const int n_rparms,
   int *flag_user_init,
   double **p_rparm,
   double *rparm_min,
```

```
      double *rparm_max);

   void user_limit_parms(
      const int n_particles,
      bool *flag_fit,
      const int n_rparms,
      int *flag_user_limit,
      double **p_rparm,
      double **p_vel,
      double *rparm_min,
      double *rparm_max);

   };
   #endif
```

3.2.3 particle_class_rlc1.cpp

```
   #include "particle_class.h"

   particle_class::particle_class(
      const int n_particles,
      double **p_rparm,
      int    *f_iparm,
      double *f_rparm,
      int& n_gvalue,
      int& n_fitness_print,
      int& flag_fitness) {

   // Use this part for "one-time" allocations, assignments, and computations.

      const int M1=3;
      double a0 = -1.0;
      double pi;
      int n_ttl;

   // number of user-defined "special" variables to be written to output file(s)

      n_gvalue = 0;

      pi = acos(a0);
      twopi = pi + pi;

      flag_fitness = flag_minimise;

   // n_fitness_print is the number of fitness measures we are interested in.

      n_fitness_print = 1;

   // The frequency response data, which serves as an input to the program,
   // is assumed to have been stored in rlc_out_ref.dat. Read that file,
   // and get the number of entries (n_data).

      file_count_lines((char*)"rlc_out_ref.dat",n_data);

   // M1 (3): no. of variables to be read from the frequency response file
```

```
// (rlc_out_ref.dat). The varialbes are assumed to be stored in three columns:
// col.1: frequency
// col. 2: phase of current phasor
// col. 3: magnitude of current phasor

// allocate memory space:

   x_ref = new double*[M1];

   for (int i=0; i < M1; i++) {
     x_ref[i] = new double[n_data];
   }

   x = new double[n_data];

   n_cols = 3;
   col = new int[n_cols];

// define indices for frequency, phase, magnitude.

   col_freq  = 0;
   col_phase = 1;
   col_mag   = 2;

   col[col_freq ] = 1;
   col[col_phase] = 2;
   col[col_mag  ] = 3;

// read from the input file frequency, phase, magnitude of current phasor:

   file_get_double_arrays((char*)"rlc_out_ref.dat",n_data,3,col,x_ref,n_ttl);

   return;
}
// ------------------------------------------------------------------------
void particle_class::get_fitness(
   const int iter_pso,
   const int flag_fitness_computation,
   const int n_particles,
   bool *flag_fit,
   double **p_rparm,
   int    *f_iparm,
   double *f_rparm,
   double *fvalue,
   const int n_gvalue,
   double **gvalue,
   double* fitness_print,
   int& flag_fitness,
   int& flag_converged) {

   double r,l,c,omg;
   double err_rms;
   double err2;
   double err_rms_avg;
   double err_stop=1.0e-6;
```

```
   int i_sys;
   std::complex<double> zc,zl,zttl;
   double zc1,zl1;

// find fitness value (fvalue) for each particle:

   for (int i_part = 0; i_part < n_particles; i_part++) {
//   get r,l,c from the array p_rparm passed by the calling routine:

     r = p_rparm[i_part][nr_r];
     c = p_rparm[i_part][nr_c];
     l = p_rparm[i_part][nr_l];

     for (int i=0; i < n_data; i++) {
//     for each frequency value, calculate the impedances of capacitor
//     and inductor.

       omg = twopi*x_ref[col_freq][i];

       zc1 = -1.0/(omg*c);
       zl1 = omg*l;

       zc = std::complex<double>(0.0e0,zc1);
       zl = std::complex<double>(0.0e0,zl1);

//     total impedance:
       zttl = std::complex<double>(r,0.0e0) + zc + zl;

//     magnitude of the current phasor (assuming the input voltage source
//     to have magnitude = 1.0)

       x[i] = 1.0e0/(abs(zttl));
     }

//   Compute the rms error, i.e., rms value of the difference between the
//   reference |I| and the particle |I|, over the frequency range of interest:

     rms_error_1(n_data,0,&(x[0]),&(x_ref[col_mag][0]),err_rms);

//   assign the rms error as fitness value of the particle:
     fvalue[i_part] = err_rms;
   }

// Find the average rms error over all particles:

   err2 = 0.0;

   for (int i_part = 0; i_part < n_particles; i_part++) {
     err2 = err2 + fvalue[i_part]*fvalue[i_part];
   }
   err_rms_avg = sqrt(err2/((double)n_particles));

// check for convergence

   if (err_rms_avg < err_stop) {
```

```
      flag_converged = 1;
    } else {
      flag_converged = 0;
    }

// assign value to fitness measure:

    fitness_print[0] = err_rms_avg;

    return;
}
// ------------------------------------------------------------------------
void particle_class::write_user_file(
    const int n_particles,
    bool *flag_fit,
    const int flag_fitness_computation,
    double **p_rparm,
    int    *f_iparm,
    double *f_rparm,
    double *fvalue,
    const int n_gvalue,
    double **gvalue,
    double* fitness_print,
    int& flag_fitness,
    const int index_user_file,
    std::ofstream& outf,
    const int n_ranks,
    int *n_rank_particles,
    int **rank_index) {

    int i_part;
    double r,l,c,omg;
    std::complex<double> zc,zl,zttl;
    double zc1,zl1,mag_i;

// write information about a particle with index given
// by the variable index_user_file (this gets assigned
// in pso_control.in and passed by the main program).
// We will call the particle index i_part.
//
// We will write the frequency response (|I| versus frequency)
// for the particle, which can be used to observe how its
// fitness improves with PSO iterations, i.e., how the
// frequency response comes closer to the specified frequency
// response.

    i_part = index_user_file;

    r = p_rparm[i_part][nr_r];
    c = p_rparm[i_part][nr_c];
    l = p_rparm[i_part][nr_l];

    outf << std::scientific;
    outf << setprecision(6);
    outf << setw(13);
```

```
   for (int i=0; i < n_data; i++) {
     omg = twopi*x_ref[col_freq][i];

     zc1 = -1.0/(omg*c);
     zl1 = omg*l;

     zc = std::complex<double>(0.0e0,zc1);
     zl = std::complex<double>(0.0e0,zl1);

     zttl = std::complex<double>(r,0.0e0) + zc + zl;

     mag_i = 1.0e0/(abs(zttl));

//   write |I| (ref and for i_part) vs f to output file:

     outf << x_ref[col_freq][i] << " ";
     outf << x_ref[col_mag ][i] << " ";
     outf << mag_i << endl;
   }

   return;
}
// ------------------------------------------------------------------------------
void particle_class::user_init_parms(
   const int n_particles,
   bool *flag_fit,
   const int n_rparms,
   int *flag_user_init,
   double **p_rparm,
   double *rparm_min,
   double *rparm_max) {

// this routine is not used in this example
   return;
}
// ------------------------------------------------------------------------------
void particle_class::user_limit_parms(
   const int n_particles,
   bool *flag_fit,
   const int n_rparms,
   int *flag_user_init,
   double **p_rparm,
   double **p_vel,
   double *rparm_min,
   double *rparm_max) {

// this routine is not used in this example
   return;
}
```

3.3 CMOS buffer design

Driving a large capacitive load with a single CMOS inverter results in unacceptably large delays [11]. To circumvent this problem, a cascade of inverters is employed in practice, with the transistor sizes increasing progressively from the first stage to the last stage (see Fig. 3.1). For a given number of stages N and a given value of the load capacitance C_L, W/L ratios[1]

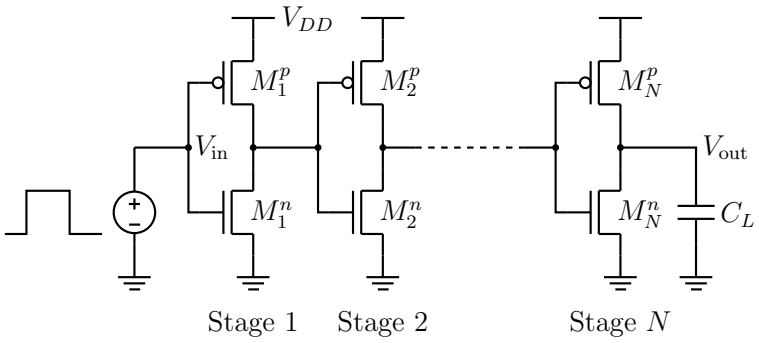

Figure 3.1: Cascade of inverters used in driving a large load capacitance [11].

of the transistors can be found analytically for minimum delay by making some simplifying approximations [11]. This procedure gives a reasonable first-order design, but to get a more accurate picture, circuit simulation must be used.

Our goal is to use PSO in designing the inverter cascade – using circuit simulation – for the least possible delay between the input and the output voltages, with the following constraints:

(a) Since the n- and p-channel transistors have different transconductances, we will take the width of the p-transistor to be twice that of the n-transistor (of the same stage). The channel length is assumed to be 1 (i.e., the minimum feature size) for all transistors.

(b) W_1^n, the width of M_1^n, is equal to 3 (i.e., three times the minimum feature size).

(c) The width for any transistor cannot exceed $W_{\max}$. Since we are assuming $W_k^p/W_k^n = 2$, it means that the upper limit for the n-channel transistor widths is $W_{\max}/2$.

We will consider an inverter cascade with $N = 5$ which can be simulated using the following NGSPICE netlist:

`mos_buffer5x.cir`

```
* Five-stage CMOS inverting buffer

* specifies the minimum feature size
.option scale=50n

* supply voltage
vdd  vdd  0  DC  1
```

[1]Typically, W and L are expressed as multiples of the minimum feature size for the MOS technology being used in implementing the circuit.

```
* input voltage is a pulse going from 0 V to 1 V at
* 500 psec, and back to 0 V at 10 nsec

Vin  in   0  DC  0  pulse 0 1 500p 10p 10p 10n 30n

* MOS transistor syntax: D G S B
* For the first stage, the widths are fixed. For the
* others, the widths will be supplied by the PSO program
* by substituting WN2,WP2,... with appropriate numbers.

MN1  out1   in     0    0    NMOS  L=1  W=3
MP1  out1   in     vdd  vdd  PMOS  L=1  W=6

MN2  out2   out1   0    0    NMOS  L=1  W=WN2
MP2  out2   out1   vdd  vdd  PMOS  L=1  W=WP2

MN3  out3   out2   0    0    NMOS  L=1  W=WN3
MP3  out3   out2   vdd  vdd  PMOS  L=1  W=WP3

MN4  out4   out3   0    0    NMOS  L=1  W=WN4
MP4  out4   out3   vdd  vdd  PMOS  L=1  W=WP4

MN5  out5   out4   0    0    NMOS  L=1  W=WN5
MP5  out5   out4   vdd  vdd  PMOS  L=1  W=WP5

CL  out5  0  20p

* 50nm BSIM4 models
....
....

* this part has not been listed; please see the
* file mos_buffer5x.cir in the PSOFT distribution
* for the transistor models.

* The following lines are required to call NGSPICE from the PSO
* program and create ASCII output files with input and output
* voltage at different time points.

.options noacct
.control
set filetype=ascii
run
write spice_buffer.txt v(in) v(out5)
.endc

.TRAN 10p 20n 0 10p

.end
```

The parameters we want to optimise are the widths of the transistors M_2^n, M_3^n, M_4^n, M_5^n in Fig. 3.1 which will be denoted by W_2^n, W_3^n, W_4^n, W_5^n, respectively. The widths of the p-transistors are related to the widths of the n-transistors by $W_k^p/W_k^n = 2$ and therefore not treated as independent parameters. A particle in this case is an inverter cascade circuit with

parameters W_2^n, W_3^n, W_4^n, W_5^n. To evaluate the particle fitness, we replace the strings `WN2`, `WP2`, $\cdots$, `WP5` in the above netlist with the width values for the concerned particle, simulate the circuit, find the rise and fall times (t_r and t_f, respectively) of the output voltage (V_{out} in Fig. 3.1), and then compute the fitness as $f = t_r + t_f$, a smaller value of f implying a higher fitness.

Figs. 3.2 (a) and 3.2 (b) show the results for $W_{\text{max}} = 400$ and $C_L = 20\,\text{pF}$. For this case, the minimum rise and fall times obtained by PSO were $t_r = 1.8\,\text{nsec}$ and $t_f = 1.1\,\text{nsec}$.

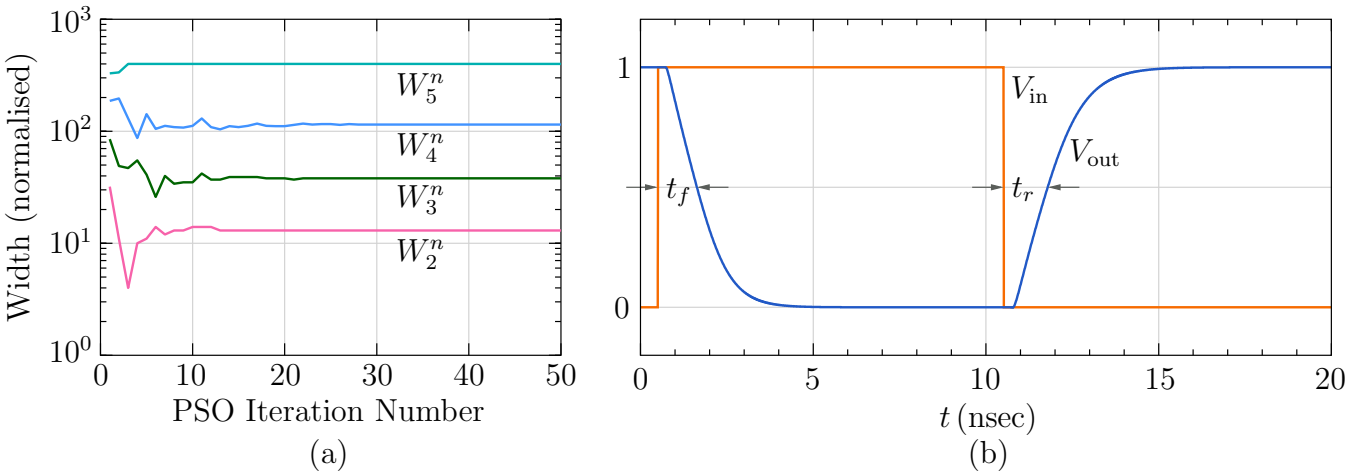

Figure 3.2: (a) Parameter values for the best particle versus PSO iteration number in the inverter cascade example of Fig. 3.1 with $W_{\text{max}}^n = 400$ and $C_L = 20\,\text{pF}$. (b) Input and output voltage waveforms for the best particle at the end of the PSO loop.

We can repeat this procedure for other values of W_{max}^n and obtain t_r and t_f in each case. Table 3.1 shows the parameter values for several values of W_{max}^n, and Fig. 3.3 shows the variation of t_r and t_f with respect to W_{max}^n, as obtained by PSO. Let us now look at the implementation

W_{max}^n	W_2^n	W_3^n	W_4^n	W_5^n
150	11	28	66	150
200	12	31	79	200
250	12	33	89	250
300	12	34	97	300
350	13	37	108	350
400	13	38	115	400

Table 3.1: Widths of the n-transistors in the inverter cascade as obtained by PSO.

aspects.

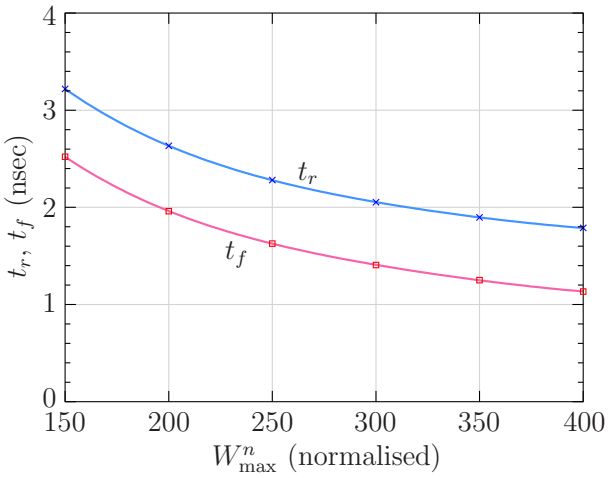

Figure 3.3: Rise and fall times versus $W^n_{\max}$ for the inverter cascade example of Fig. 3.1, as obtained by PSO ($C_L = 20\,\text{pF}$).

3.3.1 pso_control_mos_buffer.in

```
title: MOS buffer (inverter cascade)

# five-stage CMOS inverter cascade for driving a large
# capacitive load

# number of particles

n_particles = 50

# names of particle parameters

parameters: wn2 wn3 wn4 wn5

# integer parameters related to fitness evaluation

fitness_int_parameters:

# real parameters related to fitness evaluation

fitness_real_parameters:

# format specification for integers and real numbers
# (in output files)

output_format: int_width=5 double_precision=6 double_width=13
plotting_program: python3

begin_opt
  begin_parms
    wn2  type=free  min=4  max=400  round_to_int=yes
    wn3  type=free  min=4  max=400  round_to_int=yes
    wn4  type=free  min=4  max=400  round_to_int=yes
```

```
    wn5  type=free  min=4  max=400  round_to_int=yes
  end_parms
  begin_algo_parms
    itmax_pso = 50
  end_algo_parms
  begin_output
    begin_file
#     write best particle parameters once in 10 iterations
      type=best_particle
      filename=best.dat file_format=cols
      variables: ALL
      control: interval=1
    end_file
    begin_file
#     write average parameter values in each iteration
      type=avg_value
      filename=avg.dat file_format=rows
      variables: ALL
      control: interval=1
    end_file
    begin_file
#     write fitness related values in each iteration
      type=fitness_stats
      filename=err.dat
      control: interval=1
    end_file
    begin_file
#     write rank of each particle and its parameter values
#     at the end of the PSO loop
      type=ranks
      filename=rank.dat file_format=rows
      variables: ALL
    end_file
  end_output

  begin_plots
    begin_plot
      type=xy_plot
      filename=avg.dat
      variables: x=iter
+       y_1=wn2
+       y_2=wn3
+       y_3=wn4
+       y_4=wn5
      plot_control: xlog=no ylog=yes wait=no
      plot_control: title=avg
    end_plot
    begin_plot
      type=xy_plot
      filename=best.dat
      variables: x=iter
+       y_1=wn2
+       y_2=wn3
+       y_3=wn4
+       y_4=wn5
```

```
      plot_control: xlog=no ylog=yes wait=no
      plot_control: title=best
    end_plot
    begin_plot
      type=fitness_stats
      filename=err.dat
      variables: x=iter y_1=fitness_1
      plot_control: xlog=no ylog=no wait=no
      plot_control: title=error
    end_plot
  end_plots
end_opt
```

3.3.2 particle_class_mos_buffer.h

```
#ifndef PARTICLE_CLASS_H
#define PARTICLE_CLASS_H

// File pso1.h gets created by the preprocessor (psoprep), using
// the information from pso_control.in.
// It has definitions of nr_wn2, nr_wn3, nr_wn4,nr_wn5

#include <iostream>
#include <cstring>
#include <cstdlib>
#include <fstream>
#include <sstream>
#include <complex>
#include <iomanip>
#include <math.h>

#include "pso1.h"

// util1.h is required since we will use some "utility" routines
// included there.

#include "util1.h"

using namespace std;

class particle_class {

public:

// class variables:

   int M_DATA;
   double wn2,wn3,wn4,wn5;

   char **string_old;
   int *a;

   int n_cols;
   int *col;
```

```
   double **x;

   int col_t,col_vin,col_vout;
   int col_wn2,col_wp2,col_wn3,col_wp3,col_wn4,col_wp4,col_wn5,col_wp5;
   double t_cross_rise,t_cross_fall;
   double k_rise,t_start_rise,t_end_rise,x_initial_rise,x_final_rise;
   double k_fall,t_start_fall,t_end_fall,x_initial_fall,x_final_fall;

public:

// function prototypes:

particle_class(
   const int n_particles,
   double **p_rparm,
   int    *f_iparm,
   double *f_rparm,
   int& n_gvalue,
   int& n_fitness_print,
   int& flag_fitness);

void get_fitness(
   const int iter_pso,
   const int flag_fitness_computation,
   const int n_particles,
   bool *flag_fit,
   double **p_rparm,
   int    *f_iparm,
   double *f_rparm,
   double *fvalue,
   const int n_gvalue,
   double **gvalue,
   double *fitness_print,
   int& flag_fitness,
   int& flag_converged);

void write_user_file(
   const int n_particles,
   bool *flag_fit,
   const int flag_fitness_computation,
   double **p_rparm,
   int    *f_iparm,
   double *f_rparm,
   double *fvalue,
   const int n_gvalue,
   double **gvalue,
   double *fitness_print,
   int& flag_fitness,
   const int index_user_file,
   std::ofstream& outf,
   const int n_ranks,
   int *n_rank_particles,
   int **rank_index);

void user_init_parms(
```

```
      const int n_particles,
      bool *flag_fit,
      const int n_rparms,
      int *flag_user_init,
      double **p_rparm,
      double *rparm_min,
      double *rparm_max);

   void user_limit_parms(
      const int n_particles,
      bool *flag_fit,
      const int n_rparms,
      int *flag_user_limit,
      double **p_rparm,
      double **p_vel,
      double *rparm_min,
      double *rparm_max);

   };
   #endif
```

3.3.3 particle_class_mos_buffer.cpp

```
   #include "particle_class.h"

   particle_class::particle_class(
      const int n_particles,
      double **p_rparm,
      int    *f_iparm,
      double *f_rparm,
      int& n_gvalue,
      int& n_fitness_print,
      int& flag_fitness) {

   // Use this part for "one-time" allocations, assignments, and computations.

      const int M=8;
      const int M1=3;
      const int MCHR=20;

      M_DATA=10000;

   // flag_fitness should be set to 0 if high fvalue implies high fitness;
   // else to 1.

      flag_fitness = flag_minimise;

   // number of user-defined "special" variables to be written to output file(s)

      n_gvalue = 2;

   // n_fitness_print is the number of fitness measures we are interested in.

      n_fitness_print = 1;
```

```
   string_old = new char*[M];
   a = new int[M];

   for (int i=0; i < M; i++) {
     string_old[i] = new char[MCHR];
   }

   strcpy(string_old[0],"WN2");
   strcpy(string_old[1],"WP2");
   strcpy(string_old[2],"WN3");
   strcpy(string_old[3],"WP3");
   strcpy(string_old[4],"WN4");
   strcpy(string_old[5],"WP4");
   strcpy(string_old[6],"WN5");
   strcpy(string_old[7],"WP5");

// this order is the same as the strings above.

   col_wn2 = 0;
   col_wp2 = 1;
   col_wn3 = 2;
   col_wp3 = 3;
   col_wn4 = 4;
   col_wp4 = 5;
   col_wn5 = 6;
   col_wp5 = 7;

// we will save t, vin, vout in x (although vin is not
// really required)

   x = new double*[M1];

   for (int i=0; i < M1; i++) {
     x[i] = new double[M_DATA];
   }

   n_cols = 3;
   col = new int[n_cols];

   col_t    = 0;
   col_vin  = 1;
   col_vout = 2;

// this assignment depends on the order of the variables
// in the spice output file.

   col[col_t   ] = 1;
   col[col_vin ] = 2;
   col[col_vout] = 3;

// k_fall = 0.5 -> 50 % point is used to find the fall time
// t_start_fall: starting point of the output voltage fall transient
// t_end_fall: ending point of the output voltage fall transient
// x_initial_fall: value of output voltage at the beginning of the fall transient
// x_final_fall: value of output voltage at the end of the fall transient
```

```
   k_fall = 0.5;
   t_start_fall = 0.5e-9;
   t_end_fall   = 10.0e-9;
   x_initial_fall = 1.0;
   x_final_fall   = 0.0;

// (see comments for the fall transient)

   k_rise = 0.5;
   t_start_rise = 10.0e-9;
   t_end_rise   = 20.0e-9;
   x_initial_rise = 0.0;
   x_final_rise   = 1.0;

   return;
}
// -------------------------------------------------------------------------
void particle_class::get_fitness(
   const int iter_pso,
   const int flag_fitness_computation,
   const int n_particles,
   bool *flag_fit,
   double **p_rparm,
   int    *f_iparm,
   double *f_rparm,
   double *fvalue,
   const int n_gvalue,
   double **gvalue,
   double *fitness_print,
   int& flag_fitness,
   int& flag_converged) {

   int i_sys;
   int n_ttl;
   int flag_rise,flag_fall;
   int n1;
   double delt_rise,delt_fall;
   double delta1;

// initialise fitness_print[0] and n1. We will use n1 to count the number
// of particles for which a rise and fall time could be found.

   fitness_print[0] = 0.0;
   n1 = 0;

// find fitness value (fvalue) for each particle:

   for (int i_part = 0; i_part < n_particles; i_part++) {
//   round to integer the parameter values (wn2,wn3,...) as required
//   by the NGSPICE netlist

     a[col_wn2] = round_1(p_rparm[i_part][nr_wn2]);
     a[col_wn3] = round_1(p_rparm[i_part][nr_wn3]);
     a[col_wn4] = round_1(p_rparm[i_part][nr_wn4]);
```

```
      a[col_wn5] = round_1(p_rparm[i_part][nr_wn5]);

//    compute the p-transistor widths as twice the width of the
//    n-transistor of the same stage:

      a[col_wp2] = 2*a[col_wn2];
      a[col_wp3] = 2*a[col_wn3];
      a[col_wp4] = 2*a[col_wn4];
      a[col_wp5] = 2*a[col_wn5];

//    replace strings WN2,WP2,WN3,WP3,.. in the circuit file mos_buffer5x.cir
//    with the above integers and write to a new circuit file,
//    mos_buffer5y.cir.

      file_replace_strings_with_ints(
        (char*)"mos_buffer5x.cir",(char*)"mos_buffer5y.cir",8,string_old,a);

//    Run NGSPICE on the new circuit file mos_buffer5y.cir:

      i_sys = system("ngspice -b -o dum.txt mos_buffer5y.cir");

//    get the time, input voltage, and output voltage data from
//    the output file (spice_buffer.txt) created by NGSPICE:

      file_spice_get_trns_1(
        (char*)"spice_buffer.txt",n_cols,M_DATA,3,col,x,n_ttl);

//    get fall time from the above data stored in array x[i][j]:

      get_transition_time_1(
        t_start_fall,t_end_fall,x_initial_fall,x_final_fall,k_fall,n_ttl,
        &(x[0][0]),&(x[2][0]),flag_fall,t_cross_fall);

      if (flag_fall == 1) {
//      if fall time was successfully found, obtain the rise time:
        delt_fall = t_cross_fall-t_start_fall;
        get_transition_time_1(
          t_start_rise,t_end_rise,x_initial_rise,x_final_rise,k_rise,n_ttl,
          &(x[0][0]),&(x[2][0]),flag_rise,t_cross_rise);

        if (flag_rise == 1) {
          delt_rise = t_cross_rise-t_start_rise;

//        define particle fitness value:

          fvalue[i_part] = delt_rise + delt_fall;

//        assign variables related to fitness stats:
          n1++;
          fitness_print[0] = fitness_print[0] + fvalue[i_part];
        } else {
//        set a large fitness value
          delt_rise = 1.0;
          fvalue[i_part] = 1.0;
        }
```

```
      } else {
//      set a large fitness value
        delt_fall = 1.0;
        fvalue[i_part] = 1.0;
      }
//    use gvalue to store the fall and rise times.

      gvalue[i_part][0] = delt_fall;
      gvalue[i_part][1] = delt_rise;

//    write fall/rise time to console (helps in monitoring the progress
//    of the PSO program)

      if (flag_fall == 1) cout << "delt_fall = " << delt_fall << endl;
      if (flag_rise == 1) cout << "delt_rise = " << delt_rise << endl;
      cout << "i_part=" << i_part << ", fvalue = " << fvalue[i_part] << endl;
    }
    if (n1 == 0) {
//    assign a large number; this is not likley to happen.
      fitness_print[0] = 1.0;
    } else {
//    this would be the average value of (tr+tf)
      fitness_print[0] = fitness_print[0]/((double)n1);
    }

// convergence: make flag_converged = 0 -> no convergence criterion to be used

    flag_converged = 0;

    return;
}
// ------------------------------------------------------------------------
void particle_class::write_user_file(
    const int n_particles,
    bool *flag_fit,
    const int flag_fitness_computation,
    double **p_rparm,
    int    *f_iparm,
    double *f_rparm,
    double *fvalue,
    const int n_gvalue,
    double **gvalue,
    double* fitness_print,
    int& flag_fitness,
    const int index_user_file,
    std::ofstream& outf,
    const int n_ranks,
    int *n_rank_particles,
    int **rank_index) {

// not used in this example
    return;
}
// ------------------------------------------------------------------------
void particle_class::user_init_parms(
```

```
   const int n_particles,
   bool *flag_fit,
   const int n_rparms,
   int *flag_user_init,
   double **p_rparm,
   double *rparm_min,
   double *rparm_max) {

// not used in this example
   return;
}
// ----------------------------------------------------------------------------
void particle_class::user_limit_parms(
   const int n_particles,
   bool *flag_fit,
   const int n_rparms,
   int *flag_user_init,
   double **p_rparm,
   double **p_vel,
   double *rparm_min,
   double *rparm_max) {

// not used in this example
   return;
}
```

3.4 CMOS ring oscillator design

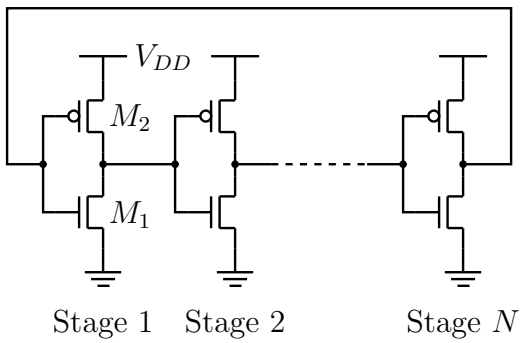

Figure 3.4: An N-stage CMOS ring oscillator [11].

A CMOS ring oscillator consists of N identical inverters connected in a ring, as shown in Fig. 3.4. Each inverter drives a rather complex load due to the gate capacitance – which varies with time – of the next stage. Design of a ring oscillator involves a choice of N and the widths W_1 and W_2 of transistors M_1 and M_2, respectively, for a specified oscillation frequency. The device widths affect both the current carrying capability and the device capacitances and thereby the oscillation frequency. An approximate analysis with some simplifying assumptions is given in [11]. However, as in the case of the inverter cascade example of Sec. 3.3, circuit simulation is required for higher accuracy.

Our goal is to obtain W_1 and W_2 for a given number of stages (N) and a given oscillation frequency (f_0), using circuit simulation. Each particle in this case is a ring oscillator with N

stages, and the particle parameters are W_1 and W_2. In addition, we look for the lowest power dissipation, i.e., the smallest value of the average current supplied by the power supply (V_{DD} in Fig. 3.4).

The following NGSPICE netlist is used to simulate a ring oscillator with $N = 21$ stages.

`cmos_ring_osc1a.cir`

```
* 21-stage CMOS ring oscillator

* specifies the minimum feature size
.option scale=50n
.ic v(out21)=0

* The following lines are required to call NGSPICE from the PSO
* program and create ASCII output files with input and output
* voltage at different time points.

.options noacct
.control
set filetype=ascii
run
write inv_out.txt v(out21) i(VDD)
.endc

.tran .005n 10n UIC

* supply voltage:

VDD   VDD  0  DC  1

* individual inverters, each being called as a sub-circuit

X1   VDD  out21  out1   INV
X2   VDD  out1   out2   INV
X3   VDD  out2   out3   INV
X4   VDD  out3   out4   INV
X5   VDD  out4   out5   INV
X6   VDD  out5   out6   INV
X7   VDD  out6   out7   INV
X8   VDD  out7   out8   INV
X9   VDD  out8   out9   INV
X10  VDD  out9   out10  INV
X11  VDD  out10  out11  INV
X12  VDD  out11  out12  INV
X13  VDD  out12  out13  INV
X14  VDD  out13  out14  INV
X15  VDD  out14  out15  INV
X16  VDD  out15  out16  INV
X17  VDD  out16  out17  INV
X18  VDD  out17  out18  INV
X19  VDD  out18  out19  INV
X20  VDD  out19  out20  INV
X21  VDD  out20  out21  INV

.subckt  INV  VDD  in  out
```

```
M2  out  in  0    0    NMOS  L=1  W=W1
M3  out  in  VDD  VDD  PMOS  L=1  W=W2
.ends

* 50nm BSIM4 models
....
....

* this part has not been listed; please see the
* file cmos_ring_osc1a.cir in the PSOFT distribution
* for the transistor models.

.end
```

To evaluate the particle fitness, we replace the strings `W1`, `W2` in the above netlist with the corresponding particle parameter values, simulate the circuit for a fixed time interval, and obtain the oscillation frequency f_0 from the output waveform of one of the inverters[2]. In addition, we compute the average current supplied by the power supply (denoted by $\overline{I}_{DD}$) over one cycle. Using f_0 and $\overline{I}_{DD}$, we compute the particle fitness as

$$f = k_1 \left| f_0 - f_0^{\text{ref}} \right| + k_2 \, \overline{I}_{DD}, \tag{3.1}$$

where k_1 and k_2 are constants (selected such that the two terms are comparable in magnitude), and f_0^{ref} is the specified oscillation frequency. A smaller value of f indicates a higher fitness.

For each specified f_0^{ref}, both W_1 and W_2 were allowed to vary from 1 to 20. Fig. 3.5 shows how W_1 and W_2 for the best particle vary with the PSO iteration number when $f_0^{\text{ref}} = 500\,\text{MHz}$ was specified. The oscillation frequency for the best particle was $f_0 = 493\,\text{MHz}$ in this case. The output voltage waveform for the best particle is shown in Fig. 3.6.

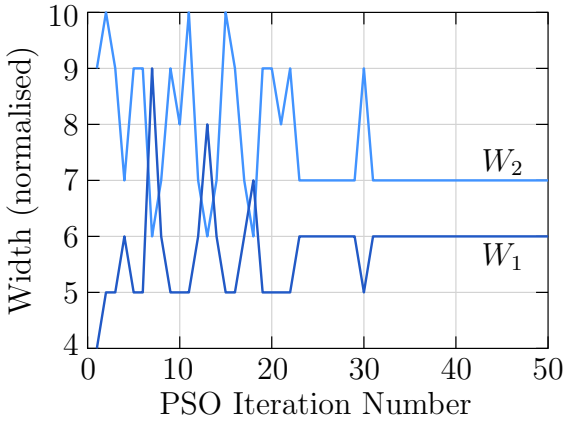

Figure 3.5: (a) Parameter values for the best particle versus PSO iteration number in the ring oscillator example of Fig. 3.4 with $f_0^{\text{ref}} = 500\,\text{MHz}$.

Table 3.2 shows the parameters and performance of the first few ranks for three values of f_0^{ref}. This information could be of interest to the user for making a suitable choice among

[2]In a ring oscillator, all inverter output waveforms are identical except for a shift in time; therefore, we can look at any one of them to obtain the oscillation frequency.

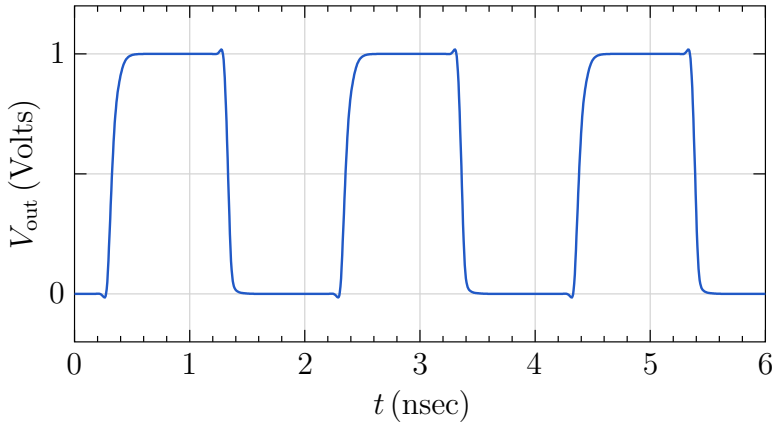

Figure 3.6: Output voltage waveform obtained for the best particle at the end of the PSO loop (50 iterations) with $f_0^{\text{ref}} = 500\,\text{MHz}$.

solutions which are close to each other, based on criteria other than those used in the fitness computation.

The implementation details for the ring oscillator example are given next.

f_0^{ref} (MHz)	Rank	W_1	W_2	f_0 (MHz)	$\overline{I}_{DD}$ (μA)
400	1	4	5	396.5	54.4
	2	3	6	378.0	52.6
500	1	6	7	493.1	80.6
	2	5	9	494.8	85.4
	3	5	8	484.5	80.0
	4	7	7	509.7	86.2
600	1	10	10	601.3	125.2
	2	9	10	590.6	119.6
	3	12	9	594.6	126.1
	4	13	9	597.3	130.0

Table 3.2: Tansistor widths W_1 and W_2, oscillation frequency f_0, and average supply current $\overline{I}_{DD}$ for different values of f_0^{ref} for the ring oscillator example, as obtained by PSO.

3.4.1 pso_control_ring1.in

```
title: MOS ring oscillator

# The circuit consists of 21 identical CMOS inverters connected
# in a ring oscillator configuration. w1 and w2 are the normailised
# transistor widths for the n and p MOS transistors, respectively,
# in each inverter. Our goal is to obtain w1 and w2 using PSO for
# a specified frequency of oscillation, f_osc_ref.

# number of particles

n_particles = 20

# names of particle parameters

parameters: w1 w2

# integer parameters related to fitness evaluation

fitness_int_parameters:

# real parameters related to fitness evaluation

fitness_real_parameters:
+  f_osc_ref = 600.0e6

# format specification for integers and real numbers
# (in output files)

output_format: int_width=5 double_precision=6 double_width=13
plotting_program: python3

begin_opt
  begin_parms
    w1  type=free  log=no  min=1  max=20 round_to_int=yes
    w2  type=free  log=no  min=1  max=20 round_to_int=yes
  end_parms
  begin_algo_parms
    itmax_pso = 50
  end_algo_parms
  begin_output
    begin_file
#     write best particle parameters in each PSO iteration
      type=best_particle
      filename=best.dat file_format=cols
      variables: ALL
      control: interval=1
    end_file
    begin_file
#     write additional (user-defined) values for the best particle
#     in each PSO iteration
      type=gvalues
      filename=gval.dat file_format=cols
      control: interval=1
```

```
    end_file
    begin_file
#     write average parameter values in each iteration
      type=avg_value
      filename=avg.dat file_format=cols
      variables: ALL
      control: interval=1
    end_file
    begin_file
#     write fitness values in each iteration
      type=fitness_stats
      filename=err.dat
      control: interval=1
    end_file
    begin_file
#     write rank of each particle and its parameter values
#     at the end of the PSO loop
      type=ranks
      filename=rank.dat
      variables: ALL
    end_file
  end_output

  begin_plots
    begin_plot
      type=xy_plot
      filename=avg.dat
      variables: x=iter
+       y_1=w1
+       y_2=w2
      plot_control: xlog=no ylog=no wait=no
      plot_control: title=avg
    end_plot
    begin_plot
      type=xy_plot
      filename=best.dat
      variables: x=iter
+       y_1=w1
+       y_2=w2
      plot_control: xlog=no ylog=no wait=no
      plot_control: title=best
    end_plot
    begin_plot
      type=fitness_stats
      filename=err.dat
      variables: x=iter y_1=fitness_1
      plot_control: xlog=no ylog=no wait=no
      plot_control: title=error
    end_plot
  end_plots
end_opt
```

3.4.2 particle_class_ring1.h

```
#ifndef PARTICLE_CLASS_H
```

```
#define PARTICLE_CLASS_H

#include <iostream>
#include <cstring>
#include <cstdlib>
#include <fstream>
#include <sstream>
#include <complex>
#include <iomanip>
#include <math.h>

// File pso1.h gets created by the preprocessor (psoprep), using
// the information from pso_control.in.
// It has definitions of nr_w1, nr_w2.

#include "pso1.h"

// util1.h is required since we will use some "utility" routines
// included there.

#include "util1.h"

using namespace std;

class particle_class {

public:

// class variables:

   int M_DATA,M_VCOIN;
   double w1,w2;

   char **string_old;
   char **string_new;

   int n_cols;
   int *col;

   double **x;

   int col_t,col_vout,col_ivdd;
   int index_W1,index_W2;

   double f_osc_ref;

   double f_large;
public:

// function prototypes:

particle_class(
   const int n_particles,
   double **p_rparm,
   int    *f_iparm,
```

```
   double *f_rparm,
   int& n_gvalue,
   int& n_fitness_print,
   int& flag_fitness);

void get_fitness(
   const int iter_pso,
   const int flag_fitness_computation,
   const int n_particles,
   bool *flag_fit,
   double **p_rparm,
   int    *f_iparm,
   double *f_rparm,
   double *fvalue,
   const int n_gvalue,
   double **gvalue,
   double *fitness_print,
   int& flag_fitness,
   int& flag_converged);

void write_user_file(
   const int n_particles,
   bool *flag_fit,
   const int flag_fitness_computation,
   double **p_rparm,
   int    *f_iparm,
   double *f_rparm,
   double *fvalue,
   const int n_gvalue,
   double **gvalue,
   double *fitness_print,
   int& flag_fitness,
   const int index_user_file,
   std::ofstream& outf,
   const int n_ranks,
   int *n_rank_particles,
   int **rank_index);

void user_init_parms(
   const int n_particles,
   bool *flag_fit,
   const int n_rparms,
   int *flag_user_init,
   double **p_rparm,
   double *rparm_min,
   double *rparm_max);

void user_limit_parms(
   const int n_particles,
   bool *flag_fit,
   const int n_rparms,
   int *flag_user_limit,
   double **p_rparm,
   double **p_vel,
   double *rparm_min,
```

```
   double *rparm_max);

};
#endif
```

3.4.3 particle_class_ring1.cpp

```
#include "particle_class.h"

particle_class::particle_class(
   const int n_particles,
   double **p_rparm,
   int    *f_iparm,
   double *f_rparm,
   int& n_gvalue,
   int& n_fitness_print,
   int& flag_fitness) {

// Use this part for "one-time" allocations, assignments, and computations.

   const int M=2;
   const int M1=3;
   const int MCHR=20;

// maximum no. of time points we expect in the NGSPICE output file:
   M_DATA = 20000;

   flag_fitness = flag_minimise;

// number of user-defined "special" variables to be written to output file(s).
// We will store the oscillation frequency and i(VDD) as gvalues.

   n_gvalue = 2;

// n_fitness_print is the number of fitness measures we are interested in.

   n_fitness_print = 1;

// string_old and string_new will be used in passing the particle
// parameter values w1 and w2 to NGSPICE.

   string_old = new char*[M];
   string_new = new char*[M];

   for (int i=0; i < M; i++) {
     string_old[i] = new char[MCHR];
     string_new[i] = new char[MCHR];
   }

   strcpy(string_old[0],"W1");
   strcpy(string_old[1],"W2");

// this order is the same as the strings above.

   index_W1      = 0;
```

```
   index_W2      = 1;

// f_osc_ref is the reference oscillation frequency which we want
// to achieve.
// f_rparm: parameters used in fitness evaluation. These are NOT
// particle parameters. The index nfr_f_osc_ref is supplied by
// pso1.h.

   f_osc_ref = f_rparm[nfr_f_osc_ref];

// We will use x[i][j] to store output data created by NGSPICE
// (time, output voltage, supply current)

   x = new double*[M1];

   for (int i=0; i < M1; i++) {
     x[i] = new double[M_DATA];
   }

   n_cols = 3;
   col = new int[n_cols];

// We will use x[i][0], x[i][1], x[i][2] to store the time value,
// output voltage, and supply current, respectively, for the ith
// data point in the NGSPICE output file.

   col_t    = 0;
   col_vout = 1;
   col_ivdd = 2;

// this assignment depends on the order of the variables
// in the spice output file.

   col[col_t   ] = 1;
   col[col_vout] = 2;
   col[col_ivdd] = 3;

   f_large = 1000.0;

   return;
}
// ------------------------------------------------------------------------
void particle_class::get_fitness(
   const int iter_pso,
   const int flag_fitness_computation,
   const int n_particles,
   bool *flag_fit,
   double **p_rparm,
   int    *f_iparm,
   double *f_rparm,
   double *fvalue,
   const int n_gvalue,
   double **gvalue,
   double *fitness_print,
   int& flag_fitness,
```

```
   int& flag_converged) {

   int i_sys;
   int n_ttl;
   int flag_osc;
   double t_period,f_osc,sum,ivdd_avg,ivdd_avg_1;
   int n_osc;
   double k1,k2,k1a,k2a;
   int i_dummy;

// n_osc is the number of particles (circuits) which show oscillations
// in the given simulation time interval and is used in computing the
// overall fitness of the group of particles (see later).

   n_osc = 0;
   fitness_print[0] = 0.0;

// k1a will multiply the error in frequency (of order 1e6),
// k2a will multiply the error in supply current (of order 1e-6).
// These scaling factors are chosen such that the oscillation
// frequency and supply current will get comparable weights in
// fitness evaluation.

   k1a = 1.0e-6;
   k2a = 1.0e6;

   for (int i_part = 0; i_part < n_particles; i_part++) {
//   convert the particle parameters w1 and w2 to strings:

     int_to_char(round_1(p_rparm[i_part][nr_w1]),string_new[index_W1]);
     int_to_char(round_1(p_rparm[i_part][nr_w2]),string_new[index_W2]);

//   replace strings W1 and W2 in the circuit file cmos_ring_osc1a.cir
//   with the above strings and write to a new circuit file,
//   cmos_ring_osc1b.cir.

     file_replace_strings(
       (char*)"cmos_ring_osc1a.cir",(char*)"cmos_ring_osc1b.cir",2,
       string_old,string_new);

//   Run NGSPICE on the new circuit file cmos_inv_1b.cir:

     i_sys = system("ngspice -b -o dum.txt cmos_ring_osc1b.cir");

//   get the time, output voltage, and supply current data from
//   the output file (inv_out.txt) created by NGSPICE:

     file_spice_get_trns_1(
       (char*)"inv_out.txt",n_cols,M_DATA,3,col,x,n_ttl);

//   Get the oscillation period (if no oscillations are found
//   in the simulated time interval, flag_osc will be set to 0).

     get_osc_period_1(
       x[0][1],
```

```
        x[0][n_ttl-2],
        0.5,
        n_ttl,
        &(x[0][0]),
        &(x[1][0]),
        flag_osc,
        t_period);

      if (flag_osc == 1) {
//      compute the frequency of oscillation from the period:

        f_osc = 1.0/t_period;

//      write to console; it helps to see the progress of the program.
        cout << "i_part=" << i_part
          << ", frequency = " << f_osc << endl;
        cout
          << "w1=" << round_1(p_rparm[i_part][nr_w1]) << ", "
          << "w2=" << round_1(p_rparm[i_part][nr_w2]) << endl;

//      compute the average supply current (for the last one period):

        get_average_1(
          &(x[0][0]),
          &(x[2][0]),
          (x[0][n_ttl-2]-t_period),
          x[0][n_ttl-2],
          n_ttl,
          ivdd_avg);
//      NGSPICE convention gives a negative supply current; take
//      magnitude:

        ivdd_avg_1 = fabs(ivdd_avg);

        n_osc++;
        k1 = fabs(f_osc-f_osc_ref);
        k2 = ivdd_avg_1;

//      compute particle fitness value as a weighted average of
//      the fabs(f_osc-f_osc_ref) and ivdd_avg_1.

        fvalue[i_part] = k1a*k1 + k2a*k2;

        cout << "k1=" << k1
         << ", k2=" << k2
         << ", k1a=" << k1a
         << ", k2a=" << k2a
         << ", k1*k1a=" << k1*k1a
         << ", k2*k2a=" << k2*k2a << endl;

//      define special variables (to be used in writing to output
//      files):

        gvalue[i_part][0] = f_osc;
        gvalue[i_part][1] = ivdd_avg_1;
```

```
        fitness_print[0] = fitness_print[0] + k1;
      } else {
        flag_fit[i_part] = false;
        fvalue[i_part] = f_large;
        gvalue[i_part][0] = 1.0;
        gvalue[i_part][1] = 1.0;
      }
    }
    if (n_osc == 0) {
      fitness_print[0] = 10.0;
    } else {
      fitness_print[0] = fitness_print[0]/((double)n_osc);
    }

// We will not check for convergence; that means the program will continue
// to execute until the last PSO iteration.

    flag_converged = 0;

    return;
}
// ------------------------------------------------------------------------------
void particle_class::write_user_file(
    const int n_particles,
    bool *flag_fit,
    const int flag_fitness_computation,
    double **p_rparm,
    int    *f_iparm,
    double *f_rparm,
    double *fvalue,
    const int n_gvalue,
    double **gvalue,
    double* fitness_print,
    int& flag_fitness,
    const int index_user_file,
    std::ofstream& outf,
    const int n_ranks,
    int *n_rank_particles,
    int **rank_index) {

// not used in this example.
    return;
}
// ------------------------------------------------------------------------------
void particle_class::user_init_parms(
    const int n_particles,
    bool *flag_fit,
    const int n_rparms,
    int *flag_user_init,
    double **p_rparm,
    double *rparm_min,
    double *rparm_max) {

// not used in this example.
```

```
   return;
}
// ------------------------------------------------------------------------
void particle_class::user_limit_parms(
   const int n_particles,
   bool *flag_fit,
   const int n_rparms,
   int *flag_user_init,
   double **p_rparm,
   double **p_vel,
   double *rparm_min,
   double *rparm_max) {

// not used in this example.
   return;
}
```

Bibliography

[1] K. Deb, *Optimization for Engineering Design*. New Delhi: Prentice-Hall India, 2012.

[2] M. R. Bonyadi and Z. Michalewicz, "Particle swarm optimization for single objective continuous space problems: A review," *Evolutionary Computation*, vol. 25, pp. 1–54, 2016.

[3] R. Eberhart and Y. Shi, "Comparing inertia weights and constriction factors in particle swarm optimization," in *Proc. Congress on Evolutionary Computing*, San Diego, USA, 2000, pp. 84–89.

[4] A. M. Chopde, S. Khandelwal, R. A. Thakker, M. B. Patil, and K. G. Anil, "Parameter extraction for MOS model 11 using particle swarm optimization," in *Int. Workshop on Physics of Semiconductor Devices*, Mumbai, India, 2007, pp. 253–257.

[5] R. A. Thakker, M. S. Baghini, and M. B. Patil, "Automatic design of low-power low-voltage analog circuits using particle swarm optimization with re-initialization," *J. Low-Power Electronics*, vol. 5, pp. 291–302, 2009.

[6] ——, "Low-power low-voltage analog circuit design using HPSO," in *IEEE Int. Conf. VLSI Design*, India, 2009.

[7] R. A. Thakker, C. Sathe, A. B. Sachid, M. S. Baghini, V. R. Rao, and M. B. Patil, "Automated design and optimization of circuits in emerging technologies," in *IEEE ASP-DAC*, Japan, 2009, pp. 504–509.

[8] ——, "A novel table-based approach for design of FINFET circuits," *IEEE Trans. CAD*, vol. 28, pp. 1061–1070, 2009.

[9] A. Chopde, D. Magare, M. B. Patil, R. Gupta, and O. S. Sastry, "Parameter extraction for dynamic PV thermal model using particle swarm optimisation," *Appl. Therm. Eng.*, vol. 100, pp. 508–517, 2016.

[10] ——, "Accurate prediction of electric power for photovoltaic modules using particle swarm optimzation," in *Proc. IEEE PVSC*, Portland, OR, USA, 2016, pp. 2638–2642.

[11] R. J. Baker, H. W. Li, and D. E. Boyce, *CMOS Circuit Design, Layout, and Simulation*. New York: IEEE Press, 1998.