

Optimization Framework Conceptual Overview

February 26, 2014

Contents

- 1 Conceptual Overview** **3**
 - 1.1 User Layer 3
 - 1.2 Server Layer 4
 - 1.3 Workstation Layer 5

- 2 Implementation** **6**
 - 2.1 Communication 6
 - 2.2 Software 6
 - 2.3 Security 7

- 3 Applications and Tools** **8**
 - 3.1 optimaster 8
 - 3.2 optiworker 9
 - 3.3 optirunner 9
 - 3.3.1 XML description 10
 - 3.4 Optimization Algorithms 14
 - 3.4.1 PSO 15
 - 3.4.2 ADPSO 16

3.4.3	SPSA	18
3.4.4	GA	19
3.4.5	Systematic Search	19
3.5	optirooter	20
3.6	optiextractor	20
3.7	opticommand	21
3.8	Dispatchers	21
3.8.1	Webots	21
3.8.2	External	22
3.8.3	Blender	23
4	Practical Information and Examples	25
4.1	Practical Information	25
4.2	Distributing Jobs	25
4.2.1	Becoming a Member	26
4.2.2	Tokens	26
4.2.3	Running on the biorob servers	26
4.2.4	Use of custom libraries	26
4.2.5	Specifying the master	27
4.2.6	Local storage	27
4.2.7	Common pitfalls	27
4.3	Examples	28
4.4	Monitoring Progress	28
	References	29

Chapter 1

Conceptual Overview

Figure 1.1 shows a schematic overview of the optimization framework. There are three distinct layers (user, server and workstation) in the system. The *user* layer runs the actual optimization process which will produce tasks that have to be evaluated. These tasks are passed to the *server* layer which queues them in a task queue. The *server* acts as a central hub to distribute any tasks it receives to the available workstations. A *workstation* in turn receives a single task from the *server* and executes it. When the task has been evaluated, the result is sent back to the *user* through the *server* layer. The different layers can run on a single PC, but can also be placed on different networked PCs. See section 2 for more information on the implementation of the different layers and communication protocols.

The overall concept of the framework is such that no specific restrictions are placed on the type of optimization algorithm or the manner in which a task has to be evaluated. Therefore, the framework can easily be used for many different and concurrent tasks.

1.1 User Layer

The user layer represents the front-end layer which is run by a user of the system. This layer is responsible for running the optimization algorithm. The algorithm produces tasks to be evaluated. As can be seen in figure 1.1, each optimization, run at the user layer, is encapsulated in a *Job* process. The job drives the optimization, sends tasks to the *server* layer, and feeds retrieved results back into the optimizer.

The optimizer consists of a population (of tasks) that need to be executed, an optional function which combines a multi-objective fitness evaluation into a single fitness value, and a data storage to store the

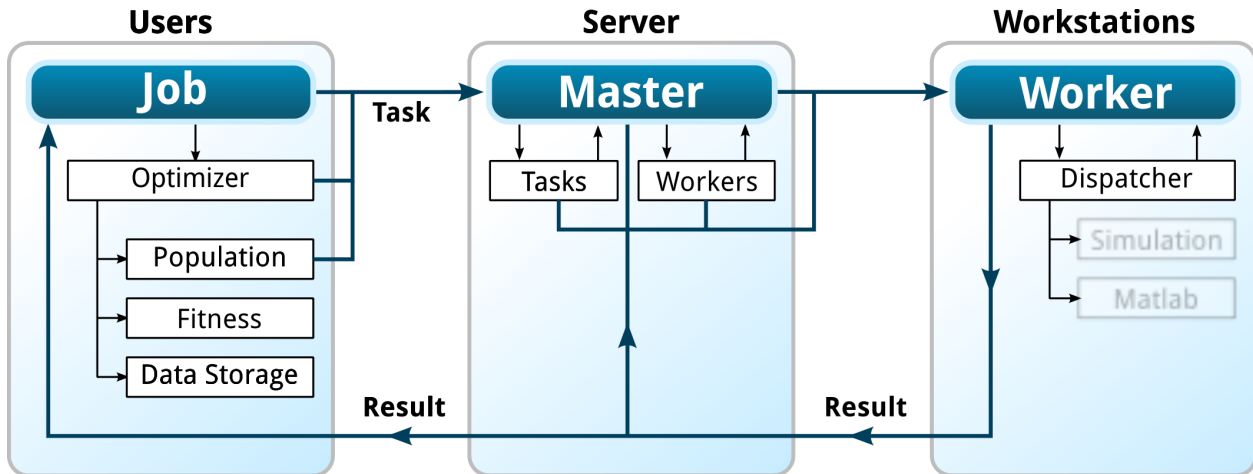


Figure 1.1: Schematic overview of the optimization framework architecture

results of the optimization. The optimizer population is a set of tasks that can be executed independently. For example, in Genetic Algorithms this would be a single generation of individuals. These represent the set of tasks that can be distributed at the *server* layer. The fitness function represents a mathematical expression that can be used to transform multiple objectives into a single fitness value, used by the optimizer.

A description of the task that is sent to the *server* layer is given in table 1.1. Once a task is executed, a result message as described in table 1.2 is sent back to the *user* layer.

Table 1.1: Task Message

Name	Description
Id	A unique task identifier
Dispatcher	The dispatcher with which to evaluate the ask
Parameters	A $\{name \rightarrow value\}$ dictionary of parameters to be evaluated
Settings	A $\{name \rightarrow value\}$ dictionary of settings to be passed to the dispatcher

1.2 Server Layer

The server layer consists of a single process which acts as a distribution center for tasks to be evaluated. This layer is used to allow multiple optimization processes to be run, while sharing the workstation resources that are currently available. The tasks are scheduled fairly with respect to the estimated execution time and

Table 1.2: Result Message

Name	Description
Id	The unique task identifier
Status	Whether the execution was successful or not
Fitness	A $\{name \rightarrow value\}$ dictionary of fitness values
Data	A $\{name \rightarrow value\}$ dictionary of additional, custom data to be stored with this task/solution

a task priority. The server automatically discovers new workstations as they come online through a simple discovery protocol. When new tasks are received from the *user* layer, it schedules these tasks (according to their priority) onto a task queue. Whenever a workstation becomes available, the task is sent to this workstation to be executed. The workstation then sends the result back to the server, which in turn relays it back to the *user* layer. Details of the implementation and the used communication protocols will be discussed in section 2.

1.3 Workstation Layer

The *workstation* layer is responsible for executing a single task, and sending the result back to the server layer. Each task is received from the *server* layer with the task information as specified in table 1.1. The worker process then resolves a dispatcher process from the task description that is to be used to evaluate the task. When the correct dispatcher is located, this dispatcher will be executed with the task that has to be evaluated. From the dispatcher, the worker will receive a result in terms of fitness, which it then relays back to the *server* layer.

Chapter 2

Implementation

This section will give a short overview of the implementation of various parts of the framework. For more detailed information, see the API documentation.

2.1 Communication

All communication in the framework is implemented using google protobuf. This is a small, strictly typed, object serialization/deserialization library developed and released under the BSD open source license by Google. This library was chosen because it is fast, object oriented, strictly typed and well maintained. Messages in the framework are mainly sent over TCP/IP, but UNIX domain sockets are also supported.

2.2 Software

There are two major parts of software developed for the framework. The first part is the lower level backbone of the framework consisting of the *server* and *workstation* layers. These layers are implemented in C++. Common functionality reused in the different components of these layers is provided in a separate C++ library. The developed applications are:

- master (*server* layer): the master distribution process which distributes task to workstations
- worker (*workstation* layer): the process that accepts tasks and executes the appropriate dispatcher

- dispatcher (*workstation* layer): the dispatchers that evaluate a specific task

The *user* layer on the other hand has been implemented in C# (using “*mono*” the open source version of the .NET framework). It consists of a .NET library containing classes with which optimizers can be easily implemented, and an application which can run jobs specified using XML. The C# language was chosen for this layer because it is easy to learn, modern, powerful, cross-platform and reduces the time needed to implement new optimization algorithms.

2.3 Security

To support a multi-user environment, the framework can be configured such that dispatcher processes will be executed under user privileges. When this functionality is enabled, the *worker* will execute a separate process that authenticates the task, and if authenticated, executes the dispatcher with the privileges of the owner of the task. The authentication is based on a challenge/response mechanisms such that the authentication key of the user is always send encoded over the network. After successful authentication, the credentials are confirmed with the permissions to execute the dispatcher process in question, before dropping to the user credentials. As such, even if the key would be retrieved, only executables that belong to the user to whom the key was given can be executed.

Chapter 3

Applications and Tools

The following section will describe the applications and tools distributed with the optimization framework.

3.1 optimaster

The `optimaster` application represents the server layer of the framework. It acts as a distribution center for optimization process. You can specify some options for the `optimaster` using a configuration file in `/etc/optimaster.conf`. An example configuration file is present in `/usr/share/optimaster/optimaster.conf.ex`. The available configuration settings are listed in table 3.1.

Table 3.1: optimaster Configuration

Name	Description
<i>discovery namespace</i>	the discovery namespace identifies to which discovery namespace the <code>optimaster</code> belongs. You can use this namespace to separate different <code>optimaster</code> and <code>optiworker</code> processes. By default, the <code>optimaster</code> will set the namespace to the user name of the current user.
<i>discovery address</i>	the address (IP:port) on which to broadcast and listen for discovery messages (usually a multicast address).
<i>listen address</i>	the address (IP:port) on which to listen for incoming optimizer connections.

3.2 optiworker

The `optiworker` application represents the workstation layer of the framework. It receives single tasks from the `optimaster` and executes them according to the task description. The `optiworker` can run in two modes, *normal* mode and *token* mode. In *normal* mode, the `optiworker` will launch dispatcher processes directly and under the same privileges as the `optiworker` process. In *token* mode, a token system will be used to securely identify users to which a certain task belongs. In this mode, the `optirooter` application will be used to launch the dispatcher process. See section 3.5 for more information.

You can specify some options for the `optiworker` using a configuration file in `/etc/optiworker.conf`. An example configuration file is present in `/usr/share/optiworker/optiworker.conf.ex`. The available configuration settings are listed in table 3.2.

Table 3.2: optiworker Configuration

Name	Description
<i>discovery namespace</i>	the discovery namespace identifies to which discovery namespace the <code>optiworker</code> belongs.
<i>discovery address</i>	the address (IP:port) on which to broadcast and listen for discovery messages (usually a multicast address).
<i>use tokens</i>	whether or not to use the token security system.
<i>dispatcher priority</i>	the linux priority level at which to run the dispatcher processes (see the ‘nice’ command for more information).

3.3 optirunner

The `optirunner` application is the most important application for the end user. This application will run the actual optimization process and will be started by the user. It is possible to use the optimization API to write your own application that communicates directly with the `optimaster`, but this should be rarely needed.

The default usage of `optirunner` is by specifying a description of your job in XML and calling `optirunner` to execute this job. A description of this XML job representation is given in the next section. Thereafter, a short explanation of how you can use your own custom optimization algorithms with `optirunner` is given.

3.3.1 XML description

The XML description of a job consists of a few distinct sections. The basic element is `job`:

```
<?xml version="1.0" encoding="UTF-8" ?>

<job name="{NAME}">
  <priority>{PRIORITY}</priority>
  <timeout>{TIMEOUT}</timeout>
  <token>{TOKEN}</token>
  <asynchronous-storage/>
  <!-- Rest -->
</job>
```

A job should always have a name. The name is used to generate the file name in which results are written. The `priority`, `timeout`, `token` and `asynchronous-storage` elements are all *optional* and specify:

- **priority**: the priority level at which you want to run the job. This value defaults to 1 if unspecified. Higher priority jobs get proportionally more time on the cluster (on average).
- **timeout**: timeout indicating the maximum allowed time for a task to run. Note that this is measuring *real time* and is only used as a safety mechanism for badly behaving dispatchers. You almost never have to specify this and is used mostly as a hack to not get jobs stuck in never ending tasks.
- **token**: an authentication token for the job. See also section [4.2.2](#) for more information on authentication tokens.
- **asynchronous-storage**: when specified, the SQLite database is written to disk in asynchronous mode. This can result in significantly faster optimizations for tasks that are very shortlived. Note however that using asynchronous storage can lead to data loss in case of unexpected failures (power outage). The default is to use synchronous storage.

The remainder of the job description consists of the `optimizer`, `boundaries`, `parameters` and `dispatcher` elements defined inside the job element.

The `optimizer` element describes the type of optimization algorithm to use and any settings that correspond to that particular optimization algorithm. For example:

```

<optimizer name="{NAME}">
  <setting name="{SETTING 1}">{VAL}</setting>
  <setting name="{SETTING 2}">{VAL}</setting>
</optimizer>

```

The name attribute identifies the optimization algorithm (for example 'pso'). The settings set settings specific to the optimization algorithm. When you for example use *PSO*, these settings include the maximum allowed particle velocity and different coefficients used in the algorithm. There are some settings available for all optimization algorithms, for more information see section 3.4.

The **boundaries** element specifies a set of boundary definitions (minimum and maximum values) which are used in the **parameters** element. An example boundaries definition:

```

<optimizer>
  <!-- ... -->
  <boundaries>
    <boundary name="{BOUND 1}" min="{VAL}" max="{VAL}" min-initial="{VAL}" max-initial="{VAL}"/>
    <boundary name="{BOUND 2}" min="{VAL}" max="{VAL}" min-initial="{VAL}" max-initial="{VAL}"/>
  </boundaries>
  <!-- ... -->
</optimizer>

```

Each boundary is named, and has a minimum and a maximum value. The name is used in the parameter definition to reference a specific boundary. The *min-initial* and *max-initial* attributes are *optional* and specify respectively the minimum and maximum value to which the initial population should be initialized (for population based algorithms).

The **parameters** element specifies the set of open parameters to be optimized. An example definition:

```

<optimizer>
  <!-- ... -->
  <parameters>
    <parameter name="{PARAMETER 1}" boundary="{BOUNDARY NAME}"/>
    <parameter name="{PARAMETER 2}" boundary="{BOUNDARY NAME}"/>
  </parameters>
  <!-- ... -->
</optimizer>

```

Each parameter is named, and specifies by which boundary its value is bound.

In addition to specifying boundaries in a separate section so that they can be reused, you can also write

boundary values in a more concise and simpler format directly in the parameter specification:

```
<optimizer>
  <!-- ... -->
  <parameters>
    <parameter name="{PARAM 1}" min="{VAL}" max="{VAL}" min-initial="{VAL}" max-initial="{VAL}"/>
    <parameter name="{PARAM 2}" min="{VAL}" max="{VAL}" min-initial="{VAL}" max-initial="{VAL}"/>
  </parameters>
  <!-- ... -->
</optimizer>
```

The optimization framework currently does not support the concept of vectors or arrays for parameters. Some tasks however are more easily defined using these concepts (for example, think of weights for a neural network). To somewhat ease the specification of such repeated parameters, the following syntax (note the `repeat` attribute) can be used:

```
<optimizer>
  <!-- ... -->
  <parameters>
    <parameter name="p" repeat="1-10"/>
  </parameters>
  <!-- ... -->
</optimizer>
```

This will generate 10 parameters with the names p1 to p10. The specified range can only contain simple integers (min and max) and these numbers are simply appended to the specified name. All other attributes (min, max, etc.) can still be used.

The last element in the optimizer node is the `fitness` element. This can be used if you have multiple objectives in your fitness function. By default, if you do not specify this element, the first fitness value the dispatcher returns is used. Dispatchers can return multiple fitness values if there are multiple objectives to be used for optimization. In this case, it can be useful to be able to define a mathematical expression combining these different values in a single fitness value. As such, you can adjust the way you combine these different objectives without having to modify the dispatcher. An example fitness description:

```

<optimizer>
  <!-- ... -->
  <fitness>
    <expression>{EXPRESSION}</expression>

    <variable name="{VARIABLE 1}">{EXPRESSION}</variable>
    <variable name="{VARIABLE 2}">{EXPRESSION}</variable>
  </fitness>
  <!-- ... -->
</optimizer>

```

The `expression` element in the `fitness` element describes the main fitness expression to be evaluated. This expression can be a mathematical expression consistent with most programming languages syntax (operators and a small set of functions such as log, sin, etc.). The dispatcher will return a dictionary (name → fitness) of fitness values, and you can refer to such a value by using the name in the expression.

Additionally, you can add any number of variables in the fitness description containing a mathematical expression. You can refer to these variables from any expression as well. This can be convenient to specify some weights or constants which you can later modify easily.

The `dispatcher` element describes the dispatcher and dispatcher settings to be used to evaluate a solution generated by the optimization algorithm. The specific settings depend on the type of dispatcher used. An example definition:

```

<dispatcher name="{NAME}">
  <setting name="{SETTING 1}">{VALUE}</setting>
  <setting name="{SETTING 2}">{VALUE}</setting>
</dispatcher>

```

The dispatcher name can be either a simple name, in which case it will be looked up in the system directory for optimization dispatchers (this is where system dispatchers such as the webots dispatcher are installed). On the other hand, you can also specify an absolute path to a dispatcher executable here.

The settings are specific per dispatcher, and are documented separately. Note that any number of additional settings can be specified here, custom to your specific job. For instance, using the webots dispatcher, you can retrieve any additional settings in your webots controller, and configure the simulation environment accordingly.

A full XML job description example:

```
<?xml version="1.0" encoding="utf-8"?>

<job name="example">
  <optimizer name="pso">
    <setting name="population-size">20</setting>
    <setting name="max-iterations">60</setting>
    <setting name="max-velocity">0.6</setting>
    <boundaries>
      <!-- The maximum speed of the e-puck is 1000 -->
      <boundary name="speed" min="100" max="1000"/>
    </boundaries>
    <parameters>
      <parameter name="left" boundary="speed"/>
      <parameter name="right" boundary="speed"/>
    </parameters>
    <fitness>
      <expression>radius - from_origin</expression>
    </fitness>
  </optimizer>
  <dispatcher name="webots">
    <setting name="world">YOUR_PATH/webots/worlds/example.wbt</setting>
    <setting name="mode">batch</setting>
    <setting name="max-time">10</setting>
  </dispatcher>
</job>
```

3.4 Optimization Algorithms

The optimization algorithms currently implemented are: Particle Swarm Optimization (PSO), Adaptive Diversity Particle Swarm Optimization (ADPSO), Genetic Algorithms (GA), Simultaneous Perturbation Stochastic Approximation (SPSA) and Systematic Search (Systematic).

These are available from the `optimizers2-sharp` library. When using `optirunner`, you can list the available optimization algorithms and their settings using the `--list` command line option. There are a few settings common to all optimization algorithms (although some might not apply to all algorithms). Table 3.3 lists these settings.

Each of the available algorithms and their settings will be briefly described.

Table 3.3: Optimization Settings

Name	Description
<i>population-size</i>	The population size (applies to population based methods and has a special meaning for the <i>SPSA</i> and <i>Systematic</i> algorithms).
<i>max-iterations</i>	The maximum number of iterations to run the optimization (does not apply to <i>Systematic</i>).
<i>convergence-threshold</i>	If specified, adds an additional stopping criterion which is based on measurement of convergence. If the convergence is below the threshold, the optimization will be stopped. Convergence is measured by: <div style="text-align: center;"> $\max_{i \in W} f_i(\mathbf{x}) - \min_{i \in W} f_i(\mathbf{x})$ </div> Where W is the set of the last <i>convergence-window</i> iterations and $f_i(\mathbf{x})$ is the best fitness in iteration i .
<i>convergence-window</i>	The window (iterations) over which to measure convergence.
<i>min-iterations</i>	The minimum number of iterations to run before the convergence stopping criterion is used.

3.4.1 PSO

The particle swarm optimization is a very elegant, simple and fairly recent optimization algorithm (Kennedy & Eberhart, 1995; Clerc & Kennedy, 2002). It is loosely based on the notion of swarm/flocking behavior. The basic algorithm behind the optimization can be described by:

$$\begin{aligned}
 v_{ij}(t+1) &= \chi(v_{ij}(t) + R_1\varphi_1(P_{ij} - x_{ij}(t)) + R_2\varphi_2(P_b - x_{ij}(t))) \\
 \chi &= \frac{2k}{|2 - \varphi - \sqrt{\varphi^2 - 4\varphi}|} \\
 \varphi &= \varphi_1 + \varphi_2 > 4
 \end{aligned}$$

It performs particularly well on real-valued problems with single single objectives. Table 3.4 lists the settings available for this optimization algorithm.

Table 3.4: PSO Settings

Name	Description
<i>max-velocity</i>	The maximum particle velocity as a fraction of the parameter space
<i>cognitive-factor</i>	The PSO cognitive factor as described in the literature (φ_1)
<i>social-factor</i>	The PSO social factor as described in the literature (φ_2)
<i>constriction</i>	The velocity update constriction as described in the literature (χ).
<i>boundary-condition</i>	The action to take when particles reach the parameter boundaries (None , Stick or Bounce). The default is Bounce .
<i>boundary-damping</i>	A velocity damping factor when the boundary condition is Bounce .

3.4.2 ADPSO

Adaptive Diversity Particle Swarm Optimization is a variant of the standard PSO algorithm that introduces collision detection and particle dispersion using adaptive, per particle collision radii and adaptive collision reaction forces (Monson & Seppi, 2006). It is a fairly elegant solution for resolving some of the deficits of the standard PSO such as premature convergence. It performs better on multi modal functions than the standard PSO.

The collision radius c_r is defined by:

$$c_r = \gamma^{b_i} + \gamma^{b_j} r$$

With γ an adaptation constant, b_i and b_j the respective number of times particles i and j have collided until now and r a basic collision radius. As can easily be seen, the collision radius per particle decreases the more it collides.

When two particles collide, their velocity is reflected and their new position is calculated by:

$$x'_{t+1} = x_t - \gamma^{-b}(x_{t+1} - x_t)$$

Thus reflecting the current position around the old position, weighted by γ^{-b} . Thus, as the adaptive radius decreases with increased number of collisions, the bounce distance increases. This allows both good convergence and escaping of local minima.

Table 3.5 lists the settings available for this optimizer. ADPSO is an extension of the standard PSO and the PSO settings defined in table 3.4 also apply to this optimization algorithm. For more information on the available settings, see [Monson and Seppi \(2006\)](#).

Table 3.5: ADPSO Settings

Name	Description
<i>adaptation-constant</i>	The adaptation constant (γ) with regard to the number of times a particle has collided (0 - 1)
<i>collision-radius</i>	The basic collision radius (r) as a fraction of the parameter space

3.4.3 SPSA

Simultaneous Perturbation Stochastic Approximation is an efficient, popular, stochastic gradient descend method (Spall, 1992). It can be defined by:

$$\begin{aligned}\hat{\theta}_{k+1} &= \hat{\theta}_k - a_k \hat{g}_k(\hat{\theta}_k) \\ \hat{g}_{ki}(\hat{\theta}_k) &= \frac{y(\hat{\theta}_k + c_k \Delta_k) - y(\hat{\theta}_k - c_k \Delta_k)}{2c_k}\end{aligned}$$

With $\hat{\theta}_k$ the solution estimation at iteration k , a_k a learning rate, $\hat{g}_k(\hat{\theta}_k)$ the estimation of the gradient at $\hat{\theta}_k$, y the objective function, c_k a perturbation rate and Δ_k the randomized perturbation vector.

The current implementation performs this two-sided perturbation to estimate the gradient with the perturbation vector Δ_k drawn from a Bernoulli \pm distribution to satisfy convergence conditions.

Table 3.6 lists the settings available for this optimizer.

Table 3.6: SPSA Settings

Name	Description
<i>learning-rate</i>	The learning rate a_k used to update the new solution according to the current gradient estimation. You can use a mathematical expression for this setting and use the variable k to indicate the current iteration number
<i>perturbation-rate</i>	The rate c_k with which to perturbate the solution to estimate the gradient. You can use a mathematical expression for this setting and use the variable k to indicate the current iteration number
<i>epsilon</i>	The normalized maximum parameter step size. You can use a mathematical expression for this setting and use the variable k to indicate the current iteration number
<i>boundary-condition</i>	Indicates how parameter boundaries should be handled. There are three possible settings: <i>None</i> (does not constrain the parameter space), <i>StickResult</i> (only constrains the actual solution to be within the parameter boundaries) and <i>StickAll</i> (constrains both the actual position and the perturbed solutions to be within the parameter boundaries).

3.4.4 GA

This optimizer implements the well known standard implementation of Genetic Algorithms. Table 3.7 lists the settings available for this optimizer.

Table 3.7: GA Settings

Name	Description
<i>selection</i>	The type of selection mechanism to use to make new populations. Available are “Tournament” and “RouletteWheel”. Tournament selection is the default
<i>tournament-size</i>	The number of individuals to use for tournament selection. Applies only when selection is “Tournament”
<i>tournament-probability</i>	Probability with which an individual is selected from a tournament: $p(1 - p)^i$ with i the individual order. You can use a mathematical expression for this setting and use the variable k to indicate the current iteration number. Applies only when selection is “Tournament”
<i>mutation-probability</i>	The probability with which to mutate an individual. You can use a mathematical expression for this setting and use the variable k to indicate the current iteration number
<i>mutation-rate</i>	The maximum amount of mutation as a fraction of the parameter space. You can use a mathematical expression for this setting and use the variable k to indicate the current iteration number
<i>crossover-probability</i>	The probability with which to use cross-over to create a new individual from two selected parent individuals. You can use a mathematical expression for this setting and use the variable k to indicate the current iteration number

The GA implementation supports both real valued as discrete valued parameters.

3.4.5 Systematic Search

The systematic search optimizer is not an optimization algorithm, but instead performs a systematic parameter search on a range of parameters. This is an example of how the distribution of tasks by the framework can be used for purposes other than strictly optimizations.

This optimizer extends the job XML description so that you can specify the range for each parameter to explore systematically instead of just the upper and lower boundary as done normally. An example

description:

```
<optimizer>
  <!-- ... -->
  <parameters>
    <parameter name="{NAME 1}" min="{MIN}" max="{MAX}" step="{STEP}"/>
    <parameter name="{NAME 2}" min="{MIN}" max="{MAX}" steps="{STEPS}"/>
  </parameters>
</optimizer>
```

You can specify ranges in two different ways, either by specifying the step size explicitly (first parameter in the example) or by specifying the number of steps that should be explored (second parameter in the example). In the second case the step size will be automatically calculated.

There are no additional settings for the systematic search. The default *population-size* setting determines the batch size in which solutions are sent to the `optimaster`. You should set this to a sensible value (around 200/300 or something).

3.5 optirooter

The `optirooter` application can be used by the `optiworker` to execute dispatcher process under the privileges of the user owning the actual task being executed. This application will use the token server to identify which user the current task belongs to. When used, the `optirooter` will receive a challenge key from the token server, which is relays back to the `optiworker`, `optimaster` and finally `optirunner` application. The `optirunner` will then encrypt this challenge with a unique token generated by the token server before starting the job. This is send back to the `optirooter` which verifies to which user this encrypted token belongs. On success, it will drop privileges to this user, and execute the dispatcher process. Before dropping privileges, the dispatcher is verified to be either a system installed dispatcher, or a dispatcher owned by the user (and located in the users' home directory) to which it will drop privileges. Note that the `optirooter` application has to be setuid and owned by root.

3.6 optiextractor

The `optiextractor` application can be used to inspect a results database. You can use it to view the settings, parameters and boundaries of a specific job. It also features the possibility to automatically replay a specific solution found during optimization. This is particularly useful in conjunction with for instance webots, to

quickly view a particular solution.

You can export the database to a matlab file which can be directly loaded into matlab. Either use the *Export* command from the menu, or use the `--export` command line argument.

3.7 opticommand

The opticommand application can be used to inspect the optimaster, displaying which jobs it is running, and how far along the jobs are. To use it, simply launch `opticommand` from the command line. For a list of commands, type `help`. If you want to connect to a remote master, use `opticommand -m <hostname>[:port]`.

3.8 Dispatchers

There are a few dispatchers available by default. This section will shortly describe these dispatchers.

3.8.1 Webots

The webots dispatcher can be used to evaluate solutions in webots. The dispatcher will launch a webots process for each task it receives and runs the world file specified in the dispatcher settings. In your controller, you can make use of the `optimization::Webots` class (see liboptimization API documentation for more information) to extract the task description and setup your controller accordingly. The specific dispatcher settings for the webots dispatcher are listed in table 3.8.

Sometimes it might be necessary to change the world file according to your task description. To this end, a special setting is available (`worldBuilderPath`) which specifies an executable that will dynamically build the webots world file to use instead of specifying it in the `world` setting. The world builder executable will be executed before the webots process is executed and will receive the task description on `standard in` (just as for the dispatcher). It should then generate the world file and write the path to this world file on `standard out`. It can be useful to use the `optimization::Dispatcher` class (see liboptimization API documentation) to extract the task description. Note that if you use several worker processes, you should take care to generate unique world file names. The world files will be automatically removed after the task has been run.

Table 3.8: Webots Dispatcher Settings

Name	Description
<i>mode</i>	The mode in which to run webots (defaults to <i>run</i>): <ul style="list-style-type: none"> • <i>run</i>: start webots graphically and start running simulation • <i>stop</i>: start webots graphically but do not start running simulation • <i>batch</i>: start webots in batch mode (< 6.1.6) • <i>minimize</i>: start webots in minimize mode (>= 6.1.6)
<i>world</i>	Absolute path to the webots world to run (required unless <i>worldBuilderPath</i> is used). In secure mode, the world must be owned by the user and must be in the users' home directory
<i>webotsPath</i>	Path to the webots executable to use (optional)
<i>webotsVersion</i>	The specific webots version to use. By default, the latest available webots version installed on the cluster is used. This version is occasionally updated (announced on the biorob mailing list). Currently available versions are: 6.1.5, 6.3.1, 6.3.3, 6.3.4 and 6.4.1 (this list can be out of date) (optional)
<i>environment</i>	Comma separated list of key=value environment variables to set in the environment in which to execute webots (optional)
<i>worldBuilderPath</i>	Path to the world builder to use to build to world before executing webots (optional)

3.8.2 External

The external dispatcher can be used to launch external processes that will handle the dispatch request. This can be seen as a very general dispatcher that abstract some parts of the optimization framework so that it becomes easier to write custom dispatchers. One example of using the external dispatcher is to evaluate a job in MATLAB. See table 3.9 for the dispatcher settings.

Table 3.9: External Dispatcher Settings

Name	Description
<i>path</i>	Path to the executable to use for evaluating the task
<i>arguments</i>	Arguments to provide to the executable
<i>working-directory</i>	The working directory in which to execute the external process
<i>mode</i>	The mode in which to describe and receive the task description. Can be either <code>protobuf</code> or <code>text</code> . The <code>protobuf</code> mode will send protobuf encoded messages, just as through the rest of the framework. The <code>text</code> mode will send tasks encoded in a simple text format. The default is <code>protobuf</code> (optional)
<i>environment</i>	Comma separated list of key=value environment variables to set in the environment in which to execute webots (optional)
<i>persistent</i>	Optional argument indicating the TCP port number to connect to to dispatch the task. You can use this if starting the environment in which you evaluate a task is expensive. If this is specified, the executable will be started the first time, and given the environment variable <code>OPTIMIZATION_EXTERNAL_PERSISTENT</code> , which contains the port number on which the external process should listen for tasks.
<i>startup-delay</i>	Specifies the number of seconds after which a connection should be established when the persistent executable is first launched (only relevant for persistent mode, optional)

3.8.3 Blender

The blender dispatcher can be used to distribute blender animation tasks. Each blender process will render a single frame and write it somewhere to disk. You can then later assemble all the single frames in a movie. Settings for this dispatcher are displayed in table 3.10. The dispatcher requires one parameter named “frame” which indicates the frame number to be rendered. Use the “systematic search“ optimizer to generate the frame numbers (using a step of 1).

Table 3.10: Blender Dispatcher Settings

Name	Description
<i>blender-file</i>	The blender file to render a frame of
<i>scene</i>	The scene name to render a frame from (optional)
<i>output-path</i>	The output directory where frames will be rendered to. By default, the output directory is set to the directory of the blender file. You can start the output-path with // to indicate “relative to the blender file directory“ (optional)
<i>format</i>	The frame output format. Defaults to PNG (optional)
<i>arguments</i>	Additional arguments that will be appended to the generated arguments for blender. (optional)
<i>environment</i>	Comma separated list of key=value environment variables to set in the environment in which to execute webots (optional)

Chapter 4

Practical Information and Examples

4.1 Practical Information

If you start running some optimizations, always first test it locally by launching an `optimaster` and one or more `optiworker` processes. Then use `optirunner` to run your job (it should automatically connect to the local `optimaster`).

If things are not working as they should, try to see if the `optiworker` is correctly connecting to the `optimaster`. To do this, run both the `optimaster` and the `optiworker` with the environment variable `DEBUG_WORKER=1` (for example: `DEBUG_WORKER=1 optimaster`). You should see in the output generated by `optimaster` if the `optiworker` is connecting correctly.

If things are still not working, try looking at the debug output of the `optiworker`. Any problems caused by the dispatcher should be displayed here.

4.2 Distributing Jobs

There are several small steps involved in running your job distributed on the system.

4.2.1 Becoming a Member

You will need to be a member of the group `cnusers`. If you are not, please contact the biorob system administrator to request group membership.

4.2.2 Tokens

You will need to generate a token so that the system can authenticate your job. Tokens can be generated on <http://eniac.epfl.ch/token.php>. Once you have a token, you should specify it in the `<token>...</token>` element of the job specification.

4.2.3 Running on the biorob servers

You will need to make sure that your job can run on the distributed systems. If you want to run a job on the biorob servers, you will need to make sure that your code is compiled for **64 bits**, Ubuntu Precise LTS (12.04). If needed, you can compile your code on `biوروبcn-gw` which should have all the required development files installed. If you need additional development files, please ask the biorob systems administrator to install these as per required.

4.2.4 Use of custom libraries

If you have custom libraries, make sure that they are either system wide installed on the distributed systems, or to specify `LD_LIBRARY_PATH` in the `environment` settings of the dispatcher. Note: The dispatchers will not be executing under your normal login environment! This means that if you do anything special in your `~/.bashrc` (or similar) you will not have the same environment for the job.

It is possible that `LD_LIBRARY_PATH` does not get set properly when specified in the `environment` settings of the dispatcher due to security reasons. If this is the case, a simple workaround is to create a wrapper script which sets `LD_LIBRARY_PATH` and then executes the actual dispatcher. For example, to do this for `webots`, create a script at `~/bin/webots-runner`:

```
#!/bin/bash

# Add $HOME/.local/lib to LD_LIBRARY_PATH
export LD_LIBRARY_PATH="$HOME/.local/lib:$LD_LIBRARY_PATH"
```

```
# Execute real webots
exec /usr/local/bin/webots "$@"
```

And specify `$HOME/bin/webots-runner` in the `webotsPath` dispatcher setting of the webots dispatcher (in the job file).

4.2.5 Specifying the master

Lastly, you need to specify the correct master to run the job on. When running the job on the biorob servers, specify `biorobcn-gw` as the master (you can specify which master to use for `optirunner` using the `-m` option).

4.2.6 Local storage

When running jobs on the biorob servers, you can make use of local storage on each of the server nodes. This local storage is located at `/data/cnusers` and is writable for every user in the `cnusers` group. Local storage is useful to write result databases when running `optirunner` from `biorobcn-gw`. You can use the `-d` flag of `optirunner` to indicate where to write result databases. **Note** that the `/data/cnusers` directory is **not** shared between the nodes. If you want to make files available to all nodes, it's therefore necessary to copy files to each individual node!

4.2.7 Common pitfalls

A common pitfall is wrong versions or architectures of binaries and shared libraries, or the unavailability of a shared library. Be sure to check the architecture for which your binaries are compiled (32 bits or 64 bits). You can do this by issuing the command `file` on your binaries (or shared libraries). Furthermore, always check if all the linked libraries can be found on the target systems. You can do this using the command `ldd`. This command will list all the shared libraries a binary links against, and should tell you whether some of them could not be found. You can login to `biorobcn-gw` to test whether or not everything can be found correctly. Be aware though, your login environment might not be the same as the environment in which the job is executed (be sure to check for custom libraries that they can also be found when running the job!).

Note for **Webots** jobs: be sure to check **all** your controllers and your possible plugin. `ldd` will always report `libController.so` and `libCppController.so` to not be found, but this is normal since they will be linked correctly by webots.

4.3 Examples

You can find a simple optimization example on the biorob website: <http://biorob2.epfl.ch/users/jvanden/docs/example/>

4.4 Monitoring Progress

You can track job progress of jobs running on the biorob servers at: <http://biorobcn-gw.epfl.ch/>. This website is only available from inside EPFL. To access the website outside of EPFL you can use the EPFL VPN. Alternatively, you can use `opticommand` to view the progress of a job.

References

- Clerc, M., & Kennedy, J. (2002). The particle swarm - explosion, stability, and convergence in a multidimensional complex space. *Evolutionary Computation, IEEE Transactions on*, 6(1), 58–73. Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=985692
- Kennedy, J., & Eberhart, R. (1995). Particle swarm optimization. In (Vol. 4, pp. 1942–1948 vol.4). Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=488968
- Monson, C. K., & Seppi, K. D. (2006). Adaptive diversity in pso. In (pp. 59–66). Seattle, Washington, USA: ACM. Retrieved from <http://portal.acm.org/citation.cfm?id=1144006>
- Spall, J. (1992). Multivariate stochastic approximation using a simultaneous perturbation gradient approximation. *IEEE Transactions on Automatic Control*, 37(3), 332–341. doi: 10.1109/9.119632