

Genetic Daemon

Version 0.2

Revised: 2002 - 06 - 10

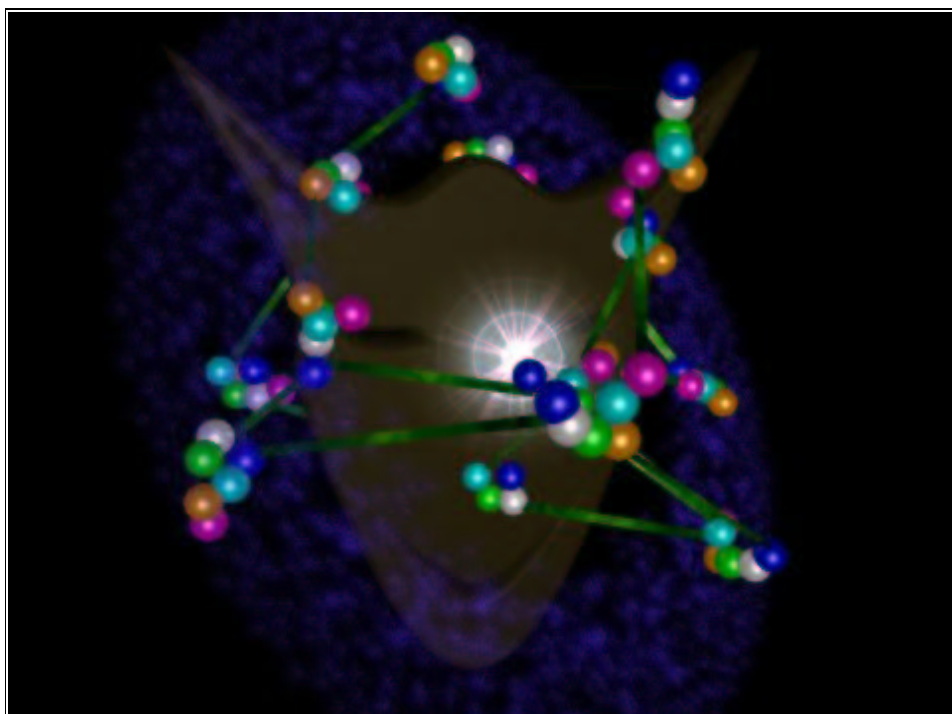


*By Giancarlo Nicolai
(giancarlo@niccolai.ws)*

Table of Content

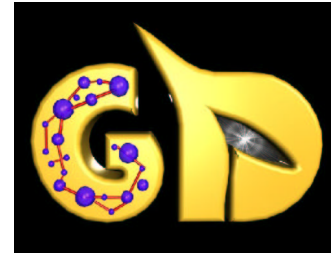
1. Introduction.....	4
1.1. Features.....	4
2. Installation.....	5
2.1. How to obtain Geneticdaemon.....	5
2.2. Requirements.....	5
2.3. Compilation and installation.....	5
2.3.1. Fancier configuration.....	5
3. Internals.....	7
3.1. Engines.....	7
3.2. Genetic code.....	8
3.3. Evolution.....	8
3.4. Learning Set.....	9
4. Administration.....	10
4.1. Basic Survival commands.....	10
4.2. Stopping and resuming the server.....	11
4.3. Plugins.....	12
4.3.1. Using plugins.....	12
4.3.2. Command line parameters.....	12
4.3.3. Configuration file.....	13
5. Usage.....	14
5.1. Before usage.....	14
5.1.1. Command-answer model.....	14
5.1.2. Common reply codes.....	14
5.1.3. Daemon States.....	15
5.2. A Brief tutorial.....	15
5.2.1. Connecting and logging.....	15
5.2.2. Creating an engine.....	16
5.2.3. Running an engine.....	17
5.2.4. Saving and restoring an engine.....	18
5.2.5. Multiple copies of engines.....	19
5.2.6. Signing off.....	19
5.3. 5.3 Advanced Engine Programming.....	19
5.3.1. Naming agents.....	20
5.3.2. Deleting undesired agents.....	20
5.3.3. Saving and restoring agents.....	21
5.3.4. Limiting calculation power.....	21
5.3.5. News on the engine: engine FEEDs.....	22
5.3.6. Limited runs.....	22
5.3.7. Altering Mutation factors.....	22
5.3.8. Turned reproduction.....	23
5.3.9. Adding Randomness.....	23
5.3.10. Turned Randomness.....	24
5.4. Using the Genetic Function Engine.....	24
5.4.1. Creating and loading gfunc.....	24

5.5. Static learning sets in deep.....	25
5.6. Engine error management.....	26
5.7. Parallel processing & Master Engine.....	26
5.8. Using a GEMaster engine.....	27
6. Technical guide.....	29
6.1. Introduction.....	29
6.1.1. Genetic Daemon API.....	29
6.2. Architecture.....	29
6.2.1. The DNA Class.....	30
6.2.2. The Genetic Class.....	30
6.2.3. The GeneticPopulation Class.....	30
6.2.4. The GeneticEnvironmet Class.....	31
6.2.5. The GeneticEngine Class.....	31
6.2.6. The GeneticEvaluator Class.....	32
6.2.7. Learning sets and units.....	32
6.3. 6.3 Programming plugins.....	33
6.3.1. The abstract engine model.....	33
6.3.2. Factory Functions.....	33
6.3.3. The engine type.....	33
6.3.4. Providing new commands.....	35
6.3.5. Command overloading.....	37
6.3.6. Plugin initialization.....	38
6.3.7. Programmer's notes.....	39
7. Questions and Answers.....	41
8. Copyright.....	42



A daemon running genetic engines

1. Introduction



Gd's logo

GeneticDaemon (GD) is a complex piece of software meant to run and to manage remotely a set of Genetic Engines; a Genetic Engine (GE) is a collection of Genetic Agents (GA) each of which is searching for a solution of a complex problem. Each engine iterates indefinitely until an agent or a set of agents are capable to find a reasonable solution, or to get near enough. Each iteration, or turn, the set of agents are "evolved" as a living population: they are reproduced and mutated in several fashion we will see later; only the fittest agents are carried on to the next turn. This way, the population adapts using a process similar to natural evolution.

Genetic Daemon has a client-server architecture; this means that the server you get in this package could be used with clients that can be found separately. However, GD can be used -almost- at top functionality also through a normal telnet session from every operative system. The basic server-to-client interface has been designed to be both human readable and program-friendly.

1.1. Features

This version of the server, even if in alpha version, is fully functional and has been tested; so with a little effort in configuration and maintaining, can be used in production environments, but it's using terms are regulated by the LICENSE and WARRANTY statements that ships with this documentation bundle. In short they state that 1) using the GD is free; 2) there is no guarantee that the GD will suit your needs (or work at all).

Features included in this version are:

- Support for user-written plugins
- Lightweight multithreading
- User security management
- Safe session handling
- Safe engine handling
- Built-in GeneticFunction type engine
- Plug in examples.
- Massive parallel processing control

2. Installation

2.1. How to obtain Geneticdaemon

GeneticDaemon can be found at <http://geneticd.sf.net> or from other sources, that can be found with a quick Internet search on the major search engines.

2.2. Requirements

To compile and install Genetic Daemon, you need Linux with kernel 2.2 or higher, networking available (at least a loopback device), and gcc 2.1 or above. Any way, this version of Genetic Daemon should compile in any well known unix environment, with an advanced C++ compiler.

2.3. Compilation and installation

For the impatient ones, become root (or an administrative user) and launch:

```
% ./configure
% make
% make install
% make install-user
```

This is often enough. The last step will create a user called "geneticd" with /var/geneticd as a home directory. The installation process will put a copy of the "geneticd" binary file in /usr/local/bin, and necessary libraries in /usr/local/lib/geneticd. Make sure to have /usr/local/lib in your LD_LIBRARY_PATH environment variable, or the dynamic linker will not be able to load the libraries. A file called /etc/gdusers will be created: it's the default user database. Finally, an rc script (geneticd) will be placed in the /etc/rc.d/init.d directory. The rc script (to launch at the daemon at the startup) is stand-alone: it can run on any unix with bash and sed installed. A bunch of header line have been added, so that it can be used with chkconfig, a script that automatically creates links in the rc.d directories to start rc scripts.

Now you have your copy of GeneticDaemon up and (ready to) running.

2.3.1. Fancier configuration

Should you wish to install geneticd in other places, (eg. in /usr), you can set some parameters in the command line, using the grammar:

```
%. /configure --param value
```

Here follows the list of useful variables.

- --prefix : where all executable and library files will be put. Defaults to /usr/local. Binaries will

be put in /prefix/bin, and libraries in /prefix/lib.

- --bindir : where all executable files will be put, under the --prefix hat. Defaults to /bin.
- --libdir : where geneticd shared objects (libraries) will be put, under the --prefix hat. Defaults to /lib.
- --sysconfdir : where configuration files will be put. Defaults to /etc (root system /etc).
- --localstatedir : where state will be put. Defaults to /var/run.

Should you run into problems please report them to the the author at [Giancarlo Nicolai](#)

3. Internals

YOU SHOULD REALLY READ THIS. Even if you are fond in AI and genetic algorithms, you should read carefully this section, because it contains notes on how the daemon is built, and thus, how you can use it.

3.1. Engines

GD can be considered as a collection of engines (GE), each of which is independent from the others. Engines have different types, that are like "subclasses" in a object oriented language. Different engines from the same engine type will have similar interface and behavior. Any way, each engine is unique, and is meant to run an unique set of agent. Let's go through the basic engine features.

Every engine is characterized by three distinct entities: the genetic environment (or agent space), the evaluator and the learning set. Every engine, no matter it's type, will run this way: each turn the agent space and the learning set are fed into the evaluator. The evaluator will test the results of each agent on the given learning set against a solution known to be right. That way, a good agent, or a good population, will be evolved in several turns. The evaluator gives a premium to each agent; the premium can be different depending by the engine type. It could be a number from 1 to 1000, from 0 to 1 or it could have no limit.

The best agents will now be literally fed. Each environment has two parameters: the competitiveness and the food. Each turn, the best agent (the one(s) that the evaluator has esteemed to be better) will get an amount of food equal to the competitiveness; the second best will get a lesser amount; the "mean" agent, or the agent that reached a premium in-between the maximum and the minimum premium given, will get exactly half the food of the best. The gained food is a linear function of the premium, but since food in the environment can be limited to be less than the "needs" of a full-grown population, some agent can be left without. Any way, the food conquered by each agent is summed up with the energy that the agent had before being fed. Depending on the engine type, there can be a maximum for the energy an agent can store. If an agent reaches it's maximum allowed energy, he will still "consume" all the food that the environment will hand him, but won't grow any further.

Now, each turn agents will grow-up. Growing up will consume exactly 1 unit of energy. At the end of each turn, if an agent has 0 or less energy, it will be removed from the environment. This is why the competitiveness should be a value greater than one and lesser than two; this way, the best(s) agents will grow, while the ones that gets less than 1 unit of food will starve till death... With a competitiveness of 2, provided that there is enough food in the environment, the agents above the mean premium will receive more than 1 unit of food, and will survive; the best agent will receive 2 food units. The exact amount of received food is a linear function based upon the minimum and maximum premium.

3.2. Genetic code

The core of the genetic system is the Genetic code (I will call it DNA for brevity). A DNA is a portion of raw data assigned to each agent. Every engine type will handle DNA in different ways, giving to each sequence different meanings. GD's work is to create new DNA mutating old ones, in a fashion that we will examine in a moment.

The evaluator is responsible for decoding the content of the DNA: an evaluator should extract the DNA from the agent it's examining, do something with that DNA basing upon the learning set rules, and give the agent a premium proportional to the fitness of the results obtained. Each auto-significant unit of the DNA is called 'gene'. A gene could be everything: integers, real numbers, strings, even program instruction. The DNA type (based upon the engine type) is responsible to create / verify if the gene sequence that evolves is consistent. If the genes are plain English words, the DNA creation routine should ensure that each new born agent has a consistent sequence, in this case a grammatically correct sentence. This burden can also be left to the evaluator, that should punish agents with inconsistent DNA with very low premiums. This is useful if 1) the probability to create an inconsistent sequence from a random mutation is less than 30%, 2) the consistence of the DNA can be (partially) known only when evaluating them against a learning set. I.e., if your DNA carries mathematical operations likes square roots, their consistence can be tested only against a learning set: square roots are meaningless when applied to negative numbers (when not in a complex space...).

3.3. Evolution

In GD, we intend the evolution as the passage from a turn to the next one, with a "generation" in between. The generation is the creation of a new set of agents starting from the old ones, with the application of random mutations to their DNA. Some engine type will work better when the a reproduction / mutation occurs every turn; in other cases more time is needed to evaluate the fitness of the agents. How often a reproduction occurs is determined by TREP (turns for reproduction) variable.

The mutation is regulated by 5 parameters: `addRate`, `delRate`, `sameRate`, `coupling` and `meanCrossOverRate` (called `mRate`). `AddRate` is the rate at which a random gene will be added to the DNA when forming a new agent. `DelRate` is the probability that a mutated DNA will have a random gene removed. `SameRate` will determ how often a random gene of the DNA will be changed to another (random) value. Next comes the coupling.

Nature has found a very efficient way to create whole new sets of genes still maintaining traits from the original ones: coupling. The `coupling` rate is the chance that when a new agent is created, it's DNA is partially derived from two pre-existing agents. When the reproduction algorithm decides that its time for a coupling, it picks up two random agents, extracts their DNA, chooses a random portion of the two DNA and mixes them together. The length of the resulting DNA will be a random choice between the lengths of the originals DNA; the firsts N genes will come from one agent, and the rest will be taken from the "tail" of the second agent's DNA.

The mean cross over rate has particular purposes, and should be used by engines that have their DNAs set to numbers or interchangeable symbols. Once decided that a coupled reproduction should happen, the engine will look to this value to decide if the bridal should be a mean cross over mutation. In this case, the length of the resulting DNA is the mean between the two original DNA lengths, and each gene is calculated as the mean of the corresponding genes in the originals

DNA. Genes from the longest agent's DNA that haven't any counterpart in the shorter DNA will be copied.

In example:

Bride's 1 DNA	1	2	6	3	4	1
Bride's 2 DNA	1	5	4	9		
Resulting DNA	1	3.5	5	6	4	

Mutation rates (add, del and same) are applied to the resulting DNA both if that DNA is copied from a gene (singleton or meiosis replication) and if the DNA is a coupled reproduction result.

All this rates are expressed as a probability, that is as a real number between 0 and 1 included.

3.4. Learning Set

Learning sets are a collection of input values and "results". Results in a learning set should be the "right" values that is known to be tied to the input values.

Since Learning sets are used by evaluators to esteem the fitness of the agents, they can contain any kind of data, even if different from the genes. Is up to the evaluator to use learning sets. A learning set could be static or dynamic. It could be a bunch of statistical data, or it could be an interface to i.e. sensors (input) and actuators (output) of a robot. Generally, a dynamic learning set will be able to provide only one test for each turn: an agent enters the evaluator, the evaluator extracts the DNA and "senses" the input from the learning set. Then applies the DNA to that inputs and sends the outputs to the set; the set, acting as an interface, will give some feedback, and the evaluator will get some conclusion. Au contrair, a static learning set will contain a table of a pre-recorded input data an their pre-recorded output, i.e. a daily year long statistical observation of humidity and pressure data and the related amount of fallen rain.

Obviously, agents that have to match against a static pre-recorded learning set should be evaluated in about one turn: an agent can give the desired result or not. When dealing with dynamic learning sets, more life span must be given to agents, and the reproduction should be regulated accordingly.

4. Administration

Geneticd is a complex genetic engine server, accepting connections from remote users that will have different rights and will be allowed to do different things. The administrator has the power to modify other users rights, and to manage running engine or the whole daemon.

4.1. Basic Survival commands

To run geneticdaemon launch

```
% /etc/init.d/geneticd start
```

a `ps -A` will show "geneticd" entries (3 processes should be on).

Now, you can exit from root shell, and login as a common user. To contact geneticdaemon, use the command:

```
% telnet localhost 2001
```

That connects with the local server at port 2001 (the way to change the TCP/IP port is explained later). The only user defined for GD at this moment is root, with password root. As soon as you get a reply from geneticdaemon, issue the commands:

```
user root
pass root
admin
help
```

The first two commands will cause you to log in as administrator. The "admin" command will change geneticd status to administrative, so you will be able to issue admin commands. An "help" is always good...

First thing to do is to change your password. With the command

```
upwd root <my-brand-new-password>
```

you will change your password.

Now, to add some more users, send the commands

```
usad <user-name> <id> <password> <role> <rights>
```

The Id is a number that *can* be used to determine user rights in the system. In the future versions, all the I/O operations that a user is able to perform in the system will be mapped to that user-id. The password is the second part of the user login. It has not to be the same password used in the system login (and the user-name and id does not need to exist in the system). Role is reserved for

future versions. For now, just use 'GUEST', 'USER' and 'ADMIN' (please, uppercase!). The user rights system allows to fine-tune the access grant to users. A symbolic string (similar to the rwx format) is used: a letter means that the attribute is present, a '-' (minus) indicates that the attribute is absent (turned off). Attributes are the followings:

- *a* - *admin*: the user can switch to administrator state, and use administrative commands. The user can also manage other user's engines (if the other rights are granted).
- *c* - *create*: the use can create or delete engines.
- *f* - *feed*: user can open and close feeds for engines he creates or owns.
- *m* - *mangle*: user can alter the population of it's own engines.
- *l* - *learning set*: user can load or alter learning sets for engines he owns.
- *l* - *engine*: user can change engine parameters (just food and competitiveness, for now).
- *p* - *population*: user can change population parameters (for now just max number of agents).
- *d* - *dump*: user can see (dump) engine contents.

A complete right string is 'acfmlepd'. Most users will have '-cfmlepd': they can do everything with engine they create, but nothing with other user's engine.

A simple

```
ulst
```

will list the user database. Now, you should see only the 'root' user, and it's rights. Passwords are never listed: they are only stored as DES ciphered entries, and it's difficult to have them back in clear text. As you can see, the root user has all attributes turned on. To alter rights or role of a user, issue the command:

```
uro1 <username> <role>
```

and

```
urig <username> <rights>
```

Finally, when you finish to edit your user database, you need to call the command

```
ussa <filename>
```

GeneticDaemon will read the database from the files /etc/gdusers, /etc/gdusers.default, ./gdusers, ./gdusers.default, in this order (when the first of these files is found, it's read). You can save the user database on /etc/gdusers to ensure that the change you made will be available next time you start the server; or you can save spare copies to have them back with a system 'cp'. Any way, changes acts immediately, as the commands 'usad', 'upwd', 'uro1' and 'urig' are issued.

4.2. Stopping and resuming the server

When you need to get out from admin state and restart normal server operations, issue the command

```
admin off
```

Remember to do this before quitting, or the genetic daemon will remain in admin status and normal users won't be allowed to log in (in future versions...).

To shut down the server, you can both issue the following command from the system command line:

```
/etc/rc.d/init.d/geneticd stop
```

or log in as root (or another administrative user), enter administrative status and issue the 'down' command:

```
user root
pass <pwd>
admin
down
```

4.3. Plugins

4.3.1. Using plugins

Geneticdaemon has a complex and full featured plugin system. It can be extended almost endlessly with plugins: new commands and engine types can be plugged in while GD is solving complex problems, without stopping and restarting.

At the startup, GD searches for .so files in the /usr/local/lib/geneticd/plugin directory (this could be changed in the command line or with configuration file options). Then, it tries to load them all. If you add some plugin after starting the daemon, you can make GD load them issuing:

```
admin
plug <filename>
admin off
```

If <filename> begins with a "/", it's intended as an absolute path, else, the name will be relative to the plugin directory you set at start time. *Filename* must be complete of the .so extension, as some version of Linux or other unix might have different extension for dynamic loadable objects.

You can have a list of installed plugin with the command

```
plst
```

and if you want to uninstall a plugin (i.e. want to replace a plugin with a newer version), issue the command

```
unplug <plugin-name>
```

from the admin state. Plugin-name is one of the name given by 'plst', not the filename. Remember to stop and free all engines using a particular plugin, before to unload it.

4.3.2. Command line parameters

Geneticdaemon can be controlled through command line and configuration file.

If you're used to start GD from it's rc script, you'll have to add command line parameters to the "start" section of the command line invocation from the rc script (usually in /etc/init.d directory). Command line parameters are also useful if you are testing GD as a developer, or if you want to launch temporary GDs.

This is the list of command line parameters:

- `-p <port>`: TCP/IP port on which GD will wait for connection.
- `--no-syslog`: do not use syslog to log messages. The default is to send log messages to the syslog daemon through kernel call.
- `--logfile <filename>`: Log messages will be written into this file.
- `-w <loglevel>`: verbosity of logging, in a range between 0 (none) and 9 (all). Default level is 6, meaning all important messages.
- `-c`: print logs also to standard output.
- `-t <seconds>`: sets timeout to *t* seconds; when a connection from a client will be idle for that timeout, it will be closed. A value of 0 disables timeout; the default is 300 (5 minutes).
- `--daemon`: Do not detach from console. Default is to fork to a child thread as soon as all start-up operations have been done, and then close the main process, resulting in releasing the console. This is mainly meant for debugging.
- `--pid`: Print child thread's pid on standard output. Used in scripts bash scripts (not working now).
- `--user <database>`: use alternative file as user's database (normally /etc/gdusers).

4.3.3. Configuration file

Configuration file (normally /etc/geneticd.conf) contains some variables that GD reads at start-up time. Entries in the configuration file assumes the form of

```
<variable> = <value>
```

Here is the list of them:

- `port`: the TCP/IP port on which the daemon will wait for connections.
- `loglevel`: as `--loglevel` command line option
- `logfile`: as `--logfile` command line option
- `plugdir`: as `--plugdir` option
- `savendir`: Where system dumps should reside.
- `name`: Name of this server, that will be displayed during connection.
- `maxts`: Maximum time share (% of CPU power) used by the engine.
- `users`: File name of an alternative user database.

Some of the parameters are configurable at runtime, after the daemon is started. In admin state, the command `parm` will give you a list of parameters and their values. Using `parm <variable> <value>` you will be able to change the value of a certain parameter.

5. Usage

5.1. Before usage

The GD's interface is designed to be both human readable and client programming ready; this way, even if a user is accustomed to use a specific client in his/her production environment (i.e. at university), the GD can be queried and/or programmed also in a foreign environment (i.e. when visiting other universities, or at home) via a standard telnet connection.

5.1.1. Command-answer model

The telnet interface of GD is designed in a fashion similar to a FTP command connection. User will be prompted for one-line commands, terminated by a CRLF sequence, that can have a variable number of parameters; GD will hand back answers that can be single line or multi line. A single line answer will begin with a '+' or a '-', a three cipher numeric code, a space ' ' (0x20, ASCII 32), and a human readable description of the answer; the line will be terminated by a CRLF sequence.

A multi-line response will begin with a "response line" that is a '+' or a '-', three cipher code, a '-' (minus sign) and a human readable description. Then, a set of CRLF terminated lines will be sent. Finally, GD signals that he has nothing more to send, it will issue a '.' character as the first character of a line, immediately followed by a CRLF sequence. So, the sequence CR, LF, '.' CR, LF signals the end of a multi-line response.

There is a third kind of answer, meant to be dealt only by client programs: binary reply. As reply for certain commands, some binary data are handled back to the connection; the fourth character of the reply line, in this case, is a '+', followed by a readable response. From the end of line, binary data is sent to the stream; this can make a poor telnet client to print unreadable character, or worse, loose connection.

Sometimes, GD handles back some values meant to be used by client application directly in the reply line, surrounding them with two apices (") or with round brackets.

As FTP, SMTP, POPx or IMAP protocols, GD does not echoes characters sent by client. Most advanced telnet clients will sense this automatically, but some old clients (WinNT or Win98 standard "telnet" command) should be properly configured. If possible, to reduce LAN or INTERNET over heading, you should also set telnet clients to send characters only when you have typed an entire line of text (again, advanced telnet client will sense this automatically).

5.1.2. Common reply codes

At this state of development, the designer(s) didn't felt the necessity for a rigid reply code structure. In general, replies beginning with '+2xx are 'OK' replies and 5xx are 'FAIL' or error replies. In detail:

- +200 - OK, everything gone fine.

- +201 - OK, but some input is still needed. Go on.
- -401 - WARNING, something went wrong, but the operation has been completed.
- 500 - General error. Something went wrong.
- 501 - Error in parameters. You mistyped something.
- 507 - Permission denied. User has not the right to do that.

5.1.3. Daemon States

At each time, after connection and before signing off, GD can be in one of the following three states:

- AUTH - remote user must authenticate himself to gain rights to use the daemon
- NORMAL - GD is ready to fulfill user requests.
- ADMIN - GD can handle error condition and can do some administrative tasks.

5.2. A Brief tutorial

This section introduces the reader in using basic functionality of the GD, in a step-by-step example. We will create and run a basic engine, just to get accustomed to the simplest part of the daemon.

First of all, let's get connected. Obtain the name and the port of the server on which the daemon is running. You should also obtain a userid/password from the system administrator. If you are the system administrator, and you are running the engine for the first time, only the "root" user is defined, with "root" as password. For this example we will assume that your login userid is "myname" and your login password is "mypass"; we will also assume that the GD is installed on a locally visible server with the name "myserver.edu", on the default port (2001). Finally, we assume that your telnet connection is properly configured (local echo / send a row per time).

Note: the telnet client of WinNT and Win98 won't intercept properly the "backspace" key. If you make some mistake, just press enter and let GD to complain about your mistyping.

Another note: all commands are case sensitive. Do not mix UPPER CASE and lower case letters. Just use lower case.

5.2.1. Connecting and logging

On Unix and Windows system, the standard telnet client can be summoned this way:

```
telnet myserver.edu 2001
```

You will get a "+200" greeting message. Now, GD is in AUTH state. Let's begin with the most useful command of all: type "help" and press enter. You will get a list of available commands: "user", "pass", and, of course, "help". Calling "help <command>" will give you a deeper description of a certain command.

Now, let's do something more useful. Send the command

```
user myname
```

and upon getting a "+201" response, send the command

```
pass mypass
```

Now GD is in NORMAL state. Try to issue an "help" again; this time you will get a wide list of commands

5.2.2. Creating an engine

The next step is about creating and setting up a functional engine. To see what engine types are available, we can use the command

```
elst
```

(engine list). The result is a list of names; if the default server has been set up correctly, you should see an entry called "intseq". The intseq engine is a very simple test-and-debug engine type, that we'll use in this tutorial. Note that intseq engine is stored in a plugin file, so if it does not appear, you have to ask your administrator to put it on line.

Intseq stands for "integer sequence": intseq engines will grow those agents whose gene, once summed up, get near to a target value. Default target value is 100.

Now, we can create an intseq engine with the command:

```
crea intseq
```

If everything goes fine, you'll get a +200 reply telling you the engine ID assigned to your engine. Write down this number, because we will use it later. We assumed that you got the ID '0', that should be true if no one else have used GD before. Also, gd returns a 12 characters string in round brackets, called UNID (universal id). Unid is used in complex multi engine remote operations; if you need to use it, just prepend a star '*' in front of it. Now, let's see how the engine is set up. Issuing the command

```
stat
```

we'll have a summary of the engines currently managed by Genetic Daemon, their creators (or owners) and their status. At this moment, the status of the engine you created is 'NONE'.

With the command

```
stat 0
```

(remember, 0 is the 'ID' of your engine), you can have an in-depth detail of the engine internal status. Assumed that your UNID was "XMsfedcddde", you can use it in all commands expecting an engine ID in that way:

```
stat *XMsfedcddde
```

Now we have to set up basic parameters for our engine. Before we can do anything else, we have to set the environment and population parameters. Let's issue the command

```
maxp 0 15
```

This will set the maximum agent number (population) to 15. Now, let's set the amount of food and competitiveness available for the population:

```
samb 0 10 1.4
```

set ambient (samb) has told the engine that the food available every turn is 10, and that the best engine will get 1.4 food units. 1.4 as the competitiveness is often a good match; also, to reduce the burden of useless agents in the population, the level of food should be from half to three quarter the maximum population.

Every turn, the best agent will be given 1.4 food units; the next one will get less; the "mean" agent, the one who gets nearest to the mean premium given to the agents, will get half the competitiveness value (0.7). The worst agent get 0; if the amount of food is less than the needs of the population, some agents will be left without. Each turn, each agents consumes 1 food unit to live. If the "energy" balance reaches 0, the agent dies.

Every agent borns with 1 energy unit, or with an amount of unit derived from it's ancestors.

Now, sending again the "stat 0" command, we'll see that the engine has entered "SETTING" state. In this state, if we change our mind, we can still change the maximum amount of agents or the ambient parameters; now, if we had to load a learning set, this would be the moment. But since the intseq is so simple, we don't need a learning set at all.

After entering the SETTING state, when you are ready to start the engine you can issue the command

```
prep 0
```

that will prepare the engine for being run. The status now turns to 'READY'. If we had an engine that needs a learning set, we should have provided it before issuing the prep command.

If you like, you can send the command

```
sett 0 130
```

sett (set target) will work only for intseq engines. It changes the default target value for the population. A short "help sett" will explain this.

5.2.3. Running an engine

With the command

```
start 0
```

your engine will begin to evolve the population, changing it's state from READY to 'RUNNING'. If you set the target as a number greater than 100, there is a big chance that it will reach a solution in a few steps. A result of 100 can often be achieved in just one turn (an agent can "born" perfect!). Try now the usual "stat 0". You will see that the engine is (likely) in status "DONE".

Now it's time to see how the engine worked. Send the command:

```
dump 0
```

this will print a list of all agents in the population at this moment ordered in terms of the last premium they received. The best agent is always the one called '0'. The worst agent is the last. You can send the command

```
dage 0 0
```

(dump agent 0 from engine 0) that will print an in-depth description of the "winning agent". "dage 0 1" will show the second-best agent and so on. As you can see, the sum of the integer numbers that represent the DNA of the best agent for this engine type is equal to the target you given.

Now, let's change the target and let's run an engine again. First of all, we reset the engine, putting it back to the "READY" state:

```
rset 0
```

Then, we change the target using

```
sett 0 140
```

Now we start the engine again with "start 0". In a few more turns, a solution will be reached, starting from the old population.

It's unlikely that you'll have to, since intseq engine is very fast, but if the engine runs very long, you can put it in "SUSPENDED" state by issuing the "stop 0" command. Stop is a "async" command; you can issue it at every time (while the engine is running), and your connection won't be suspended; however, there will be a short while before that the engine really stops.

5.2.4. Saving and restoring an engine

Now let's try to save the hard work we did. Issue the command:

```
save 0 filename.eng
```

GD separates different user's file: when saving the engine, the daemon will automatically prepend your userid to the filename. In this way, if you and your colleagues save a "test.eng" file, each user will still have it's own test, without overwriting other's one.

A binary client could call the command "save <eng-id>", that would store the engine data to the telnet stream with a *binary reply*; then client program could store it locally, and send back to the server (or to another server) when needed. But if you try to do that with a telnet connection, your telnet client could be messed up by some binary character sent by the server.

To get rid of the old copy of the engine, use the command:

```
free 0
```

Please, note that an engine must not be in 'RUNNING' state to be deleted. Now, we can load back the engine we saved with:

```
load filename.eng
```

If everything goes fine, GD will reply that the engine has been loaded, and will handle a newly

created engine ID back. Repeating this step again will create multiple copies of the same engine. A quick "dump <ID>" (where ID is the number that GD sends back when loading the engine) will show you that the population is exactly in the same state that we have left it.

Freeing all unused engines before leaving is a good practice.

5.2.5. Multiple copies of engines.

Sometimes, you'll want to have more than one copies of an engine. This could be i.e. because you want to test different strategies of evolutions from a certain point.

Each engine has a unique UNID, that can't be replicated, and is stored with each saved engine file. This means that if you try to load an exact copy of an engine into GD, you will get an error message. To make it possible, you have to change the UNID of the existing copy of the engine *before* loading the saved copy.

The command "reid" can be used to do this: it gives an engine a brand new UNID. To duplicate an engine, you have to follow this steps:

```
save <eng-id> <filename>
reid <eng-id>
load <filename>
```

NOTE: most likely, future version of GD will have a command to duplicate directly an engine.

5.2.6. Signing off

To quit a GD connection, simply send the command

```
quit
```

If your connection has been idle for a long time (i.e. you didn't typed anything for a while), GD will close the session. Closing your telnet connection, or suffering a network failure, will have the same effect. Any way, the user serving session and the engine space are completely asynchronous, and if you connect back to GD, you'll find that your engines are evolved (if they were running) or are in the same state as you left them (if they were suspended or done). You can let the engines running for your entire summer holidays, and come back to them when you get back home: they'll be fine on their own.

5.3. 5.3 Advanced Engine Programming

This section describes how you (or an "intelligent" client) can interact with the engine to get faster/better results. In fact, GD let's you move agents between compatibles engines, remove undesired agents, create new random agents or custom agents with pre-recorded capabilities. I.e. you can run several engines for a while, and save their winning agents; next you can create a new engine and load the winners, making them to compete in a sort of "genetic" championship. Or you can create a "laboratory" agent, esteemed to be good, and make it to compete with a evolution generated agents. Possibilities are almost endless.

This techniques goes under the name of *engine mangling*.

5.3.1. Naming agents

Often, can be useful to follow the destiny of a particular agent in a population; recognizing it with its genetic code would be difficult, so agents can be given a name for an easier tracking. The agents created when the engine is first initialized are all named "default0" to "defaultN-1", where N is the numerosity of the population. Each agent name is followed by a "-" sign and a number. That number represent the "generation" of the named agent; the whole string (name + generation) is called symbolic ID.

In example, if we want to name an agent as "eddie", it will become the root of the "eddie" family, and its full symbolic ID will be "eddie-0". If the system derives a child from that agent, it will be called "eddie-1". The child of "eddie-1" will be "eddie-2" and so on.

In commands, agents can be referred both with their numerical position in the population (0, 1, 2 and so on) and with it's symbolic ID.

The symbolic ID naming convention brings two drawbacks. If two agents are derived from the same ancestor, they will have the same symbolic ID; in other words we could have two or more "eddie-1" in our population. In this case, commands referencing agents with symbolic ID that have multiple instances in the population will act only on the first one (that should be the one with the greatest premium among them).

Another problem is that in cross over generation, the newborn agent will inherit the name and generation code only from the first one of the two "married" agents.

This problem will probably be issued by next versions of GD.

To change the name of an agent, the command "nage" is provided:

```
nage <eng-id> <agent-id> <name>
```

agent-id can be both an integer id or a symbolic id (name+generation). *name* must be without "-" sign, that could fool client applications; a "-0" will be always added as generation counter.

5.3.2. Deleting undesired agents

Sometimes, engine algorithms will produce a set of very similar genetic agents. GD won't care for this, and, often, neither the engine type: it would be a burden to great to test for similar genetic agents not to be in the population.

However, to allow a greater variety in the population and thus have better chances to evolve better agents, you are allowed to delete agents that have grown too similar to their ancestors, and which get a high premium, making some different but less optimal solution die for starvation. Obviously, you can also have a client program do that for you...

The command

```
dele <engine-id> <agent-id>
```

will delete the desired agent. Be careful, since there isn't any way to bring a deleted agent back. Note that the "dele" command can be issued when the engine is running (in 'RUNNING' status),

but in this case, the execution of the command will be delayed until the engine finishes next calculation turn.

Note also that agent ID are simply their order in the population; so, deleting an agent will shift up all the IDs of the following agents by one. As an example, if you want to delete the agents from 5 to 9 in the engine 3 you have to send the following command sequence:

```
stop 3 ( if the engine is running )
dele 3 5
dele 3 5
dele 3 5
dele 3 5
dele 3 5
dele 3 5
start 3 ( if you want to start it again )
```

5.3.3. Saving and restoring agents

Is often useful to have a single agent saved: both for restoring it at a later state, to make it compete again and (possibly) create a new set of different descendant, and to have a snapshot of the evolution at different times. Is also useful to save the agents from an engine and restore them in another engine, or in another server. This allow to "move" population champions around, and create new evolution branches.

The command to save an agent is:

```
sage <eng-id> <agent-id> [filename]
```

The third parameter (filename) is optional: if not given, the agent will be sent in a binary form to the telnet stream. If given, a file with that name will be created.

To restore an agent, you will need the command

```
lage <eng-id> [filename]
```

The agent will be created only if there is room in the population, and will be added to the bottom of the population with the energy that it had before saving. Be careful: it's likely that the agent will move upward in the very next turn: you'll find it elsewhere, and with a different ID. Refer to the agent symbolic id to find it. Of course, loading agent(s) in an engine is allowed only if the agents are coming from engines of the same type.

Sage and lage commands are sync commands: if the engine is running, you'll have to wait for it to reach a "keypoint" before the command is executed. Any way, this wait is assured to be less than a second long.

5.3.4. Limiting calculation power

Sometimes there is the need to reduce the CPU consumption of certain engines. You could have a server that is dedicated to many tasks, and want GD not to consume all the computation power; or you could want to balance evenly the weight of different engines.

To limit the consumption of CPU for a given engine, issue the command:

```
slim <eng-id> <cpu%>
```

The `cpu%` parameter is a real number between 1 and 100 that represent the maximum CPU power that we want the engine to use. When `cpu%` is set to 100 or 0, the limitation is turned off.

Note that the calculation of cpu usage is done in chunks a second long; this can result in imprecise allocations, in the range of about 2% random difference between the desired and the actual values.

5.3.5. News on the engine: engine FEEDs

Engine feeds are meant to help client programs to keep track of engine status, but can be also useful if used by hand. An engine feed consists of a "stream" of data about an engine, sent every time the engine reaches a different turn. The data is sent to a TCP/IP port, and consist of the output of `'stat <eng-id>'` command followed by the output of `'dump <eng-id>'`.

Using a feed, a client can show updated data about the engine to the user without having to query periodically the daemon; a human reader can know when something changed on his/her engine because a new feed is arrived.

To open a feed, just send the command:

```
feed <eng-id>
```

the GD will send back a response indicating the TCP/IP port number used by the feed. Opening a telnet connection to that port will begin the transaction.

Feeds are one-way only: the server will send data to the client. To stop a feed, you can both close the feed port connection on the client side or issue the

```
fest <eng-id>
```

(feed stop) command on the standard connection. Only one feed per engine can be active at a time. This is because, at this moment, GD is meant to be a multi-user application, and engines are meant as single user application spaces. This could be changed in future versions (since internal engine structures are multiuser-like).

5.3.6. Limited runs

Sometimes is useful to have the engine run for a limited number of turns (i.e. if you want to get snapshots of the populations at precise intervals).

The command

```
start <engine-id> <count>
```

will let the engine run for `<count>` more turns. There are two ways to know when the engine reaches the given turn (or enters in "DONE" status before): the engine can be periodically queried about it's status, or a feed can be opened (see 3.3).

5.3.7. Altering Mutation factors

Each engine have five basic mutation factor meant to vary new born agent genetic code. When reproduction is started, after that all agents with an energy value less or equal to 0 are removed

from the engine, this randomness factor will be applied to all the agents derived from the old ones. Random mutation factors are `addRate`, `delRate`, `sameRate`, `coupling` and `meanCrossOverRate` (called `mRate`).

`AddRate` is the rate at which a random gene will be added to the DNA when forming a new agent. `DelRate` is the probability that a mutated DNA will have a random gene removed. `SameRate` will explain how often a random gene of the DNA will be changed to another (random) value. The coupling factor is the chance that a new born agent's genetic code will be derived from two "parent" agents, both living after the grow-up phase of the turn. The mean cross over rate has particular purposes, and should be used by engines that have their DNAs set to numbers or interchangeable symbols. Mutation rates (add, del and same) are applied to the resulting DNA both if that DNA is copied from a gene (singleton or meiosis replication) and if the DNA is a coupled reproduction result.

All this rates are expressed as a probability, that is as a real number between 0 and 1 included.

Default mutation factor can be found using the "`stat <eng-id>`" command, and can be changed at every moment; since they are "sync" commands, if the engine is running they will be delayed until a "key point" is reached.

To alter singleton mutation factor, you can use the command

```
muta <add-rate> <del-rate> <same-rate>
```

while you'll use the command

```
xover <copulation> <mRate>
```

to alter coupling parameters.

5.3.8. Turned reproduction

Some engines will need to run several turns before to decide which agent is worth to survive and thus reproduce. This is the case of dynamic learning set, where the agents need to develop the capacity to adapt to the mutating environment conditions. With the command

```
trep <eng-id> <tcount>
```

you can set a turn interval before a new selection/reproduction is started. I.e., you can send '`trep 1 15`' to tell the engine N. 1 to check for survival and reproduction only once in 15 turns. The default of this value is determined by the engine types, for static engine types will probably be 1 (survival/reproduction is done at every turn).

The current value of this variable can be found with the command '`stat <eng-id>`'.

5.3.9. Adding Randomness

From time to time, engines will tend to discard agents that do not display an high premium. This could decimate your population, or sterilize it to the point that further evolution is difficult. In this occasions, you can create a "complement" for the population that "possibly" can deviate the progress of a sterile growth. The command

```
rand <eng-id>
```

will fill the population with some randomly generated agents having an high energy value. By default, only half the spare space in the population will be filled. This two factors ensure that the "distilled" agents that were present before the "rand" command will interact for some turns with the newly created agents, maybe mangling their genetic code and having a chance to derive a "side" population development. You can override this default by specifying the number of agents to be created, with the command:

```
rand <eng-id> <agents>
```

5.3.10. Turned Randomness

If you are running genetic algorithms that tend to sterilize population often, you can instruct the engine to execute a "rand" command at regular intervals. The command

```
tram <eng-id> <turn-count>
```

(turned random mutation) will instruct the engine to execute a "rand" command each *turn-count* turns. To turn this feature off, use 0 as turn-count value.

5.4. Using the Genetic Function Engine

The Genetic Function Engine is a very complex piece of software, meant to find unknown complex mathematical relationship in statistical data series. In other words, it's capable of defining an $f(X)=Y$, where X is an $m \times n$ matrix of statistical observations and Y is a n -dimension vector. Every row of the X matrix is a random variable instance, observed in a natural phenomenon, that is "somehow" related to an "output" variable y .

Also, the Genetic Function Engine (gfunc from now on) is an example of static learning set based engine. This section covers it's peculiarities, as well as the generic description of this kind of engines. We'll follow a tutorial approach describing how to create and use the gfunc.

5.4.1. Creating and loading gfunc

As for the non static learning set engines (or the engines that don't need a learning set definition at all), you can create and set up a gfunc with the following commands:

```
crea gfunc
maxp <maximum agent number>
samb <maximum food> <competitiveness>
```

You can also set default mutation parameters. Gfunc sets addRate, delRate and sameRate are set by default to 0.33 (one third) respectively, and copulation to 1 (every new agent's genetic code is derived from two ancestors). mRate is set to 0; don't change this value, since it's hard to define the "mean" value in between two mathematical operators. The default values had proven to be good in many tests, so you can let them alone.

Now it's time to load the learning set. A randomly generated 5x5 real number matrix is a good example. Issue the

```
ldat <eng-id> [filename]
```

command. The "filename" parameter is optional; without it, a software client can send ASCII coded real numbers afterwards, but this operation can also be performed by hand. If a filename is given, the learning set will be read from a file stored on the GD server side.

Now, if you try to issue the 'ldat <eng-id>' command, you will be prompted to enter the statistical observations one per row: the independent variables before (Xk vector) and the dependent variable afterwards (y). Entering a row of five blank (space) separated real numbers will tell the gfunc that we want to find a 4-dimension function:

$$y = f(x_0, x_1, x_2, x_3)$$

Enter the data as in the following example:

```
3  5.2  8  1  6
3  4.1  3.1 4  3
7  3.3  1  6  4
4  9    5  2  1
5  3    8  2  1
```

When you're done press enter on a blank row.

Now we are ready to issue the 'prep <eng-id>' command. This will create a default population: some common interpolating functions are created first (linear, geometric, quadratic, exponential...), and then a bunch of random agents are added. You can dump the population now, if you want. You will see a list of functions. Dumping a single agent (dage <eng-id> <agent-id>) will show also the values of constant, "nearly" optimized to achieve the most "interpolating" function as possible (see considerations below).

The population is ready to start. The engine will reach 'DONE' status if an agent gets a premium near to 1000. Premium is a reverse function of the mean distance of the function's given result vector from the know result vector (the last column we entered when loading the set). More exactly, the premium is 1000 minus the per thousand mean error of the interpolating genetic functions. A premium of 995, for example, means that the agent is interpolating the known results with a mean error of 0.5%.

5.5. Static learning sets in deep.

Static learning sets are loaded into the engine by both reading them from a user-readable file or a pre-compiled file. Suppose that you have loaded a set as explained above into an engine (whith the "ldat" command), and that the engine have been prepared. Now, the set can be saved as binary data with the command

```
sset <eng-id> [filename]
```

If filename is given, the learning set will be saved on a locally stored file; else, the set will be sent as a binary reply thorough the connection.

To load a set, you can use the "ldat" command or this one:

```
lset <eng-id> [filename]
```

Static learning sets can be loaded (by both `ldat` and `lset` commands) also after the "prep" command; when the engine is ready, you can load a different set. This way you can i.e. simulate a changing environment. On very long engine runs, (that can go through for months, or years), you can add new observations in the set, and then load the new set in the engine. This will allow further adaptation of your genetic population with regards to ever-changing phenomena that have to be analyzed.

5.6. Engine error management

Sometimes, complex engines can meet up with some erroneous condition. It could be a condition raised by the engine itself, if it meets up with some unexpected situation. Imagine, for example, an engine with an evaluator that queries an external device (or program) about the fitness of the agents. If the external device fails, or if the external program reports an error, then the engine could signal the unexpected condition entering in the 'ERROR' state.

Other times, the engine could receive a system error, i.e. accessing a wrong memory address, or dividing a number by zero. This condition, caused by run-time programming errors, are caught by a "guardian" thread that forces the engine to stop all its operations, and enter the ERROR state.

In either way, the engine can be queried by the user on what caused that error, and can be resumed and started again. The command

```
derr <eng-id>
```

(dump error) will print the description of what happened. Using the command

```
cler <eng-id>
```

(clear error) the engine will be put again in "ready" status, but nothing else will be done. The engine will be in the same condition as before the error was found, and if the condition that caused it is not cleared, it is likely that after a start command the engine will be back in error state. A more radical way to handle error is the command

```
rset <eng-id>
```

This command, also used to move the engine out of the DONE state, will clear the error and copy the previous turn population into the working population of the engine, effectively issuing a sort of "undo" command on the last turn processing.

5.7. Parallel processing & Master Engine

In this version (0.2) parallel processing is still in an embryonic phase, but is fully working and has some features worth to be looked at. At this moment, it is not possible to create an integrated population spreading upon different nodes in a network, but it is possible to have a master engine coordinating an arbitrary number of slave engines, presenting them to the user as if they were one. This elevates to various degrees the calculation power of the genetic engines, and allows to simplify the management of various populations all of which working on the same problem.

Future version will allow mainly two improvements: slave integration through population exchanges (geographically mapped genetic clusters) and Master engines controlling other master engines...

Deep differences between self-sufficient engines and engines controlling other engines is implemented through C++ sub-classing.

Engine class is determined at creation time, with the "crea" command; the class is listed as one of the parameters of the engine when using the "stat" command. The default form of the "crea" command, the one that we have seen so far, creates what is called a GEInABox: genetic engine in a box. This class of engine is fully self-sufficient; it is capable of evolving a population, that is working on a problem, completely by itself. The "crea" command as another form:

```
cler <engine-type> <engine-class>
```

We already discussed about the engine-type parameter. The new one is the engine-class. It can assume one of the following values:

- GEInABox
- GEMaster

The first class is the one that we have used up to date, and that is created by default by the crea command. The second class represents the master engine, capable to manage many engines on remote servers of a given engine type.

5.8. Using a GEMaster engine

We'll go through a simple tutorial to describe how to create and manage masters engine, then we'll go through some consideration about how the master engines can be used to solve really complex problems.

First thing, let's suppose to have tree GD running on three different servers; they can be both in our organization or they could be reached through a remote internet connection. The main one, the one we are contacting, will be myserver.edu. The other ones will be foo.edu and bar.edu. All the GD have the same configuration (that is to say, the same plugins installed), and we have three different accounts on those server. We are 'root' with password 'root' on myserver.edu, and 'guest' with password 'guest' on foo.edu and bar.edu.

After contacting myserver.edu, we create our master engine.

```
user root
pass root
crea intseq GEMaster
maxp 0 20
samb 0 10 1.4
prep 0
```

This is the same as we have seen, except for the "GEMaster" parameter of the crea command. If we had created an engine of a type requiring a learning set, we should have issued a ldat or lset command before prep.

Now, we begin to do a new thing: with the command "slav", we can enslave remote GD. The command is as follow:

```
slav <eng-id> <slave-url> <port> <user-id> <password>
```

slave-url can be both a domain name or an ip address. The other parameters should be clear enough. Let use this command to contact our slave GD.

```
slav 0 foo.edu 2001 guest guest
slav 0 bar.edu 2001 guest guest
```

In response to the slav commands, we'll get a "slave ID". This is a number that is used to address a slave in the group managed by the master engine. This slave ID (both the number returned or the slave's UNID) must be used when accessing agents. Let's try to see our work:

```
dump 0
dage 0 0
dage 0 1.0
dage 0 2.0
```

With the dump command, we'll have a list of all the agents in the cluster. As you can see, the dump is a little more complex now: in front of the name you can see a number (in our case 1 or 2) and a UIND between parenthesis. When accessing a single agent, you can use it's numeric ID in the cluster (as in the second instruction of the above example), or you can route the agent in the cluster by make it to be preceded by the slave id (in our case, 1 or 2). You can combine any of the address methods of GD: engine by slave-id, engine by unid, agent name or agent id:

```
dage 0 *UNID-SLAVE1.agentname0-0
dage 0 *UNID-SLAVE1.0
dage 0 1.agentname0-0
dage 0 1.0
```

Note also that SLAVEID starts from 1, not from 0 as usually.

Exept this special commands and agent addressing, master engines obey to all the commands that can be issued to their slaves, broadcasting them to the engines linked with them. Some little behavior difference can be observed: i.e., there will be a short delay when issuing start and stop commands. Other than that, master engines are assured to have a command interface compatible with the one used by slave engines.

An important feature of the master engine is the ability to reset the slaves that have fallen in error state. In case of an internal error of an engine (i.e. a division by 0 or a memory access fault) the master will reset it. After a few try, the slave engine is put off line and will require manual intervention.

6. Technical guide

6.1. Introduction

This guide is meant to introduce basic technical issues about geneticdaemon, to help developers in getting their hands on the code.

First of all, try to load GD plugins and see if they works. If so (plugins are loaded and listed with 'plst' command), you are ready to go on.

6.1.1. Genetic Daemon API

From the same location where you downloaded this manual, at <http://geneticd.sf.net> is available also a complete API documentation, with a detailed description of all the classes, method, global functions and data structure used by Genetic Daemon. Developers willing to extend or debug the program will find it useful. The rest of this document is meant to be a thing half way from a tutorial to a technical guide, helping to understand the structure of Genetic Daemon, and how to implement new engine types, mainly through plugins. For a more detailed description, please refer to the API Documentation.

6.2. Architecture

GeneticDaemon is designed to have the lesser mess possible in it. Before to embrace this project, I programmed a genetic function algorithm with JAVA. Boys, it was SLOW! Not for the CPU, neither for JAVA, but simply because I wrote the genetic functions as a tree of operators (it's natural for a oop programmer!), and each generation carried an heavy burden of tree calculation. Worse than that, every turn I had a lot of 'new' and 'delete' around: for creating new agents, for creating new agent's new genes etc. This worked under Java, but carried an extra burden of garbage collecting. Then I ported that algorithm under C++/Win 32. About once every 400 turns, I had an error caused from within the malloc function... "Well, let's use Linux" I thought. Things went better: I had a malloc generated crash once every 1500 turns. Definitely, freeing and allocating so many small memory areas so often was a crash generator for every operative system.

So I decided to build up things in a way that memory is allocated only once for each engine, when it's created. And this is why I decided to use the following architecture.

6.2.1. The DNA Class

In GD, DNA is the class holding agent's genetic code. DNA is a sequence of bytes, subdivided in fixed length units. Length of each unit is determined when the engine is created, in the constructor; so a DNA holding double precision values is internally represented as a sequence of bytes, that is holding units 8 characters long. As for gfunc engine type, DNA chunks can be also a small structure; or they could be a sequence of pointers to statically allocated strings. The important thing is that the DNA is a sequence of self-meaning fixed length data; exactly as the natural DNA.

DNA has only the ability to move this fixed length data around: i.e., inserting a gene in the sequence at a certain point means:

- shifting forward data to make room for the new gene
- copy the gene bytes in the new space.

This allows to allocate a fixed amount of memory for each DNA: a DNA will have a maximum number of genes, but this should be fine for most problems.

Saying it simply, the DNA is a sophisticated fixed length data vector, with a lot of method to manipulates single elements.

6.2.2. The Genetic Class

Oh, yes, I know. The correct definition for this is 'agent', but somehow I always felt more comfortable programming them as 'Genetic'. So, you'll have to stick with it. Also, I just love to declare in programs things like

```
Genetic *agent; //nicer than Agent *agent;
```

Genetic class implements an agent. As a living cell, it incorporates a DNA, that can be used by itself when reproducing, or by the program if it wants to extract the DNA for "genetic manipulation". The most important user for the DNA synthesized by the Genetics will always be the Evaluator, that will use the code to do something meaningful, and will give premiums to the agents that have created better DNAs.

While the DNA is just a vector, Genetics are living. They have an age (integer counter) and an energy (double floating) that must stay above 0 to survive. They can be fed, and the food they receive will go in to their energy reserve. Aging will cost exactly 1 energy unit.

An agent can be born from synthesis (it's DNA can be created from external functions) or can be reproduced and mutated from one agent. With coupled reproduction, it can be created with some parts of it's DNA from both parents.

Finally, an agent can be given a premium, that he will use when requesting food.

6.2.3. The GeneticPopulation Class

Having a statically created DNA wouldn't be enough if a gene born or death would signify a 'new' and 'delete' action. The population is a vector of pre-allocated genetic agents, each with it's own pre-allocated DNA. When we need space for a new agent, we use the method allocate(), and when we don't need an agent any more, a simple Genetic::kill() will turn it off.

In other words, reproduction, mutation and elimination of agents does not lead to any memory allocation!

This brings great advantages of 1) speed and 2) stability.

GeneticPopulation is basically a Genetic vector, but it's also responsible for starting a new generation(), or for aging() the whole population, and to start the survival() loop.

GeneticPopulation will also gather informations on the population: living agents count, maximum, minimum, mean premium and best agent.

IMPORTANT: those values are calculated in the aging() loop. So, when you alter the population outside the normal grow-up/survival loop, you have to call the aging(false) method, that will calculate those values without really aging the agents. This lets you make "block" operations: you can add more than one agent and then "close" the population with "aging(false)".

6.2.4. The GeneticEnvironment Class

Environment is where a population resides: it has the role to evaluate agents and then feed the best ones. The evaluating process is done by a specialized class, called GeneticEvaluator, that we'll see later.

Environment is also responsible for the normal turn processing. See the "turn()" method. The turn method creates a local copy of the population; actually, that copy is pre-allocated when the Environment is created, so there isn't the need to allocate and delete a whole population each turn. All turn operations (evaluating, feeding, growing up and survival) are done on this copy; turn() locks the environment only 1) when copying the actual population on the work copy and 2) when moving the copy to the work population to the "official one".

When querying the environment about its population (i.e. when executing a "dump" command), only the official version of the population is used; this ensures that commands will be immediately executed and will read a consistent data. When changing the environment in some meaningful way (i.e. removing an agent), the operation is queued.

All altering commands that a GeneticEnvironment can process must be passed to the "Alter" method. This method creates an EnvCommand instance; that instance represents a command that could change the structure of the environment and/or the population; this can be immediately processed, if the environment is not engaged in turn calculation, or can be delayed until the turn is over.

6.2.5. The GeneticEngine Class

GeneticEngine class is a base class for all engines, both self sufficient engines and masters requiring to be linked with slaves. GEInABox is a child class of the GeneticEngine, and is at the same time a fully functional self sufficient engine and the base class for all other kinds of slave engines. GEMaster is another child, and is at the same time a self sufficient master engine, capable to deal with slaves, and the base class for other masters.

The `GEInABox` class is holding three elements: the environment, which we have seen, the evaluator and the learning set. Each turn, the environment will call the `turn()` method of the `GeneticEnvironment` class, which will handle the population to the `GeneticEvaluator::evaluate()` method.

The engine is also responsible for setting initial population parameters, and calling user-defined population creation methods (or provide one if creation function is not available).

The `GEMaster` engine has a vector of informations needed to deal with slaves (called `SlaveDef Vector`) and his main duty is to pass command issued by the user to the slaves. It also do other tasks, as organizing the slave results, managing and verifying the running status of slaves and resuming slaves from error state transparently. Finally, it has the responsibility to decide when to suspend or remove a slave from it's list of links.

Engine belongs to a particular user that can do almost anything with it's creations. Other users will be able to access to engine not belonging to them in a degree defined by their attributes, unless they have a root or equivalent account. The "admin" bit is not enough: if you want to make a "superuser" other than root, you have to set on all it's access rights.

6.2.6. The GeneticEvaluator Class

The role of the evaluator is the most central of the entire system. It have to "give" a vote to every agent in the population. It's "voting" is not bound to any particular rule other than it must be a 'double' C numeric data type. The higher the "premium", the more food will receive the agent who got the price, and the higher the survival chance it has. When the evaluator thinks that it's satisfied with the best(s) agents in the current population, it set's it satisfaction flag, and the engine enters in the `DONE` state.

The evaluator can rely on a `LearningSet`, if it wish, and is the sole responsible for learning set updating (if necessary). There isn't a precise rule on how to use learning sets: the evaluator can use them as it likes, or do not use them at all.

6.2.7. Learning sets and units

Learning sets are mostly learning units vectors. A learning unit is a set of dependent data and independent data. The objective of the engine is to grow a population that somehow transforms the independent data of the learning units in their dependent data. The evaluator has to decide how much an agent gets near in doing some tasks. With a static learning set, there is no need to derive a `LearningSet` subclass: just derive `LearningUnits` subclasses to fit your needs, and use the built in vector of the `LearningSet` class (we'll discuss it later).

A developer needs to subclass `LearningSet` only if:

- the set is not a static vector of units, but it can change both in size or quality of it's units.
- the problem to solve does not require the set to be a vector at all, but something different (i.e. a target value, an external program, an hardware device etc.).

6.3. 6.3 Programming plugins

When I decided to develop a plugin system for the genetic daemon, I faced a choice on how to implement an "extension" system. I had two hypothesis: 1) use the standard C++ sub classing system; 2) use the standard `dlopen()` dynamic loading library. The first solution seemed the most natural to me, but carried the need of dynamic class management. That was easy in Java or in other 4th generation languages, but it would have added some degree of complexity to my C++ program. The other solution seemed to be more immediate, but embrace it completely would have signified to drop C++ class advantages. I decided for a middle way solution: I generalized the `GeneticEngine` class (and some of the other classes) to have the maximum flexibility without sub classing; I adopted a sub-typing mechanism, that is simpler to implement than a full range sub classing system, and lift some burden from the plugin writers.

6.3.1. The abstract engine model

To generalize the engine model, I needed to isolate the common parts of very different genetic algorithms: so plugin writers would have to write just the different algorithm implementations. Every genetic engine is made of three parts: the environment, the learning set and the evaluator.

The evaluator is different for every genetic algorithms; also learning sets are likely to be dissimilar in different algorithms. Environments have all the same structure: they grow up a population and pass it to the evaluator.

Each environment has a population, which is mainly a genetic agent vector; also, genetic agents have just an energy level, an age and a genetic code (that I call improperly DNA). The DNA can change greatly.

We can assume that an abstract engine type is defined by a specific DNA, `LearningSet` and `Evaluator`.

The implementation of this abstract model required me to encapsulate what we have seen above in a "structure" called "engine type". Engine types heavily depends on "factory functions"; we'll discuss about that in a while. The final result is an highly flexible API, that is still simple to implement for a programmer. Implementability had been my first concern in developing this API.

6.3.2. Factory Functions

Factory functions are the simplest mean to generate a wide range of objects in a class tree hierarchy without the need to have a dynamic sub classing mechanism. The main role of a factory function is making an instance and initialize it. Since factory functions can be `dlsym()` loaded, it's easy to use them as subclass object generators in plugins.

6.3.3. The engine type

To create an engine type, the initialization function of the plugin (or any other function in GD) have to call the following:

```
#include <genetic.h>

int registerEngineType(
    char *type,
```

```

DNA_FACTORY_FUNC dna_fact,
DNA_DEF_FACTORY_FUNC dna_defs_fact,
EVALUTOR_FACTORY_FUNC eval_fact,
LEARNING_SET_FACTORY_FUNC ls_fact,

//optional parameters

POP_AFTER_BORN_FUNC pop_afterborn,
POP_CREATE_DEFAULT_FUNC pop_default,
POP_EQUIV_FUNC pop_equiv
);

```

The first parameter (type) is the name of the engine type. It must be an unique symbolic name; it will be used when listing available engine types (with `elst` command) and as a parameter for the `"crea"` command.

`"dna_fact"` is the factory of our DNA. It's declared as:

```
typedef DNA * (DNA_FACTORY_FUNC DNA ) (DNA_DEF *defs);
```

The factory function will receive a structure called `DNA_DEF`, that will contain basic DNA initialization parameters. The `DNA_DEF` passed to `dna_fact` will be dynamically created with the `dna_defs_fact`; so you are in control of how this parameters will be formed.

`"dna_defs_fact"` is defined as follows:

```
typedef DNA_DEFS * ( DNA_DEFS_FACTORY_FUNC ) (LearningSet *set);
```

this factory function receives the learning set as a parameter because it could possibly influence basic DNA initialization. Some genetic engines could find useful to have the learning set handy when creating a new DNA. The `DNA_DEFS` are defined as follows:

```

typedef {
    int ngenes;    // number of genes allowed in the DNA
    int gdim;     // lenght of a single gene (in bytes)
    double food_limit; // default maximum food for an agent using
that dna
    double start_energy; // def. max. energy for an agent using that
dna
    void *params; // other DNA-subclass specific data, for your use
    int paramLen; // it might be useful for functions using
"params" value.
    LearningSet *lset;
} DNA_DEF;

```

NOTE: DNA factory functions are responsible for decoding and using `params`, `paramLen` and `lset` fields: they are not used directly by other parts of the engine type. As you see, `DNA_DEF` holds parameters for both the DNA and the agent creation: they are used when creating new agents (each of which holds a unique DNA). If the sequence you intend to create is just a seamless data sequence, you don't have to subclass the DNA class and to create a new dna factory. All you have to do is to create a `DNA_DEF` factory that returns a `DNA_DEF` structure with the correct value of `"gdim"`: the size of your structure. If you don't need to keep genetic code transformations necessarily self-consistent, the default DNA class will be enough to handle the mutations. You need to overload the DNA class only if your starting genetic code must be a consistent sequence of symbols and the newborn agents (with mutated code) must still be consistent with a rule set known by your evaluator.

The "eval_fact" parameters is the factory of your evaluator:

```
typedef GeneticEvaluator * ( EVALUTOR_FACTORY_FUNC ) ( LearningSet
*set );
```

Again, the evaluator could find useful to have the learning set handy at it's very creation.

The "lset_fact" is the function used to create the learning set. It's defined as:

```
typedef LearningSet *( LEARNING_SET_FACTORY_FUNC ) ( );
```

In most cases, LearningSets is just a vector of learning units: it's often the learning unit that does all the job. If the base LearningSet class it's what you're looking for, your learning set factory will have just to call the base constructor of the LearningSet class, using a LEARNING_UNIT_FACTORY_FUNC you will provide. In fact, the unit factory are defined as:

```
typedef LearningUnit * ( * LEARNING_UNIT_FACTORY_FUNC )( );
```

The other parameters of the registerEngineType function are optional (you can fill them with 0 or NULL). If not null, pop_default will be called by the engine when first creating the population: it will fill the environment with the "ancestor" agents; if it's not present, the engine will create a random population.

The parameter pop_afterborn, if present, will be called each time a new agent is created. It should do some agent post-creation cleanup, and if it wills, it can even kill it.

The eval_func parameter will be used to compare two agents every time is needed. The standard evaluation function will compare two DNA sequences: if they are identical, the agents are supposed to be equivalent. If you need a more sophisticated check, you'll have to provide a better equivalence function. At this moment, this comparison happens only when creating a new agent, right after "after_born": if present, and if it estimates that two agents are identical (returning "true"), one of the agents will be killed. This is to prevent the best agents to make endless copies of themselves, thus sterilizing the variety of the population. Those function are defined as follows:

```
typedef void ( * POP_CREATE_DEFAULT_FUNC )
( GeneticPopulation *pop, LearningSet *set );

typedef void ( * POP_AFTER_BORN_FUNC )
( Genetic *g, GeneticPopulation * p, LearningSet *set );

typedef bool ( * POP_EQUIV_FUNC ) ( Genetic *g, Genetic *g1 );
```

6.3.4. Providing new commands

Often, a plugin writer will feel the need to add some more commands for the engine he/she is writing. GeneticDaemon plugin systems allows to:

- create a brand new command
- overload existing commands
- create engine specific versions of existing or new commands
- create generic (not engine-bound) commands.

This is done by writing a command interpreter function, and calling the following function:

```
int CmdManager::add(
    char *name,
    char *eng_type,
    char *eng_class,
    char *synopsis,
    char *helpShort,
    char *helpLong,
    short int has_eng,
    short int minp,
    short int maxp,
    int state_mask,
    int rights_mask,
    CMD_INTERPRET_FUNC p );
```

An instance of a command manager object will be passed to the plugin initialization function as a parameter.

The `name` parameter is the very command itself. The `eng_type` parameter is the engine type on which the command is applicable. It can be "*" if it's applicable to all engines, or if it is not bound to a particular engine type. `eng_class` is the name of the engine class that will accept this command. It can be "*" if the command is applicable to all classes, "GEMaster" or "GEInABox" if the command must be issued only to one of this predefined classes, or any class you redefine.

`Synopsis`, `help_short` and `help_long` are meant to inform the human user: the first one is a usage description shown by both 'help' and 'help <cmd>' commands. The second is the descriptive hint given in the command list. The third is the in-depth explanation given by the 'help >cmd<' statement.

`has_eng` is an integer signaling that the command is bound to an engine; the value of the engine is the position of the engine ID in the parsed argument list, 0 being the command itself, 1 the first parameter and so on (generally it's 1: the first parameter).

`minp` and `maxp` indicates the minimum and maximum number of parameters allowed without signaling a parsing error, included the eventual 'has_eng' parameter but excluding the command itself. I.e., if `minp` = 1 and `maxp` = 3, this means that the command must be followed by at least one and at maximum three blank separated strings, regardless of the 'has_eng' parameter value.

The parameter `state_mask` indicates on which GD states the command will be available. GD states are declared as follows:

```
#include <session/sessionmanager.h>

#define GD_STATE_NONE 1 //user just connected and must
authenticate
#define GD_STATE_USER 2 //user keyword give, waiting for
password
#define GD_STATE_NORMAL 4 //user logged in
#define GD_STATE_ADMIN 8 //server in administrative state
#define GD_STATE_ALL 15 //a shortcut instead of summing up all
```

Values can obviously undergo bit field operation (and, or, not) to get a finer mask.

The parameter `rights_mask` indicates what clearance is to be obtained prior to use that command. Since most rights are intended as rights to do something on a certain engine, a "rich" `rights_mask` will be more suited on engine-bound commands. Here follows the list of rights

```
#include <authent/gdrighs.h>
```

```

#define GD_RIGHT_ADMIN      0x0080 //administrator or staff
#define GD_RIGHT_CREATE     0x0040 //can create engines
#define GD_RIGHT_FEED       0x0020 //can open and manage feeds
#define GD_RIGHT_MANGLE     0x0010 //can alter agents
#define GD_RIGHT_M_LSET     0x0008 //can alter learning sets
#define GD_RIGHT_M_ENG      0x0004 //can add or remove agents
#define GD_RIGHT_M_POP      0x0002 //can change pop/ambient
parameters
#define GD_RIGHT_DUMP       0x0001 //can see
#define GD_RIGHT_ANY        0x0     //anyone can use it

```

Combining more than one right will have the effect to require that the user must have *all* the combined rights to access a certain command on an engine that he/she does not own. All commands are applicable to the engines created by a certain user, if it's the same user that is issuing them.

The last parameter is the command interpreter function that you have to provide this way:

```

typedef int ( * CMD_INTERPRET_FUNC )
( Vector *command, SessionData *s, GeneticEngine *ge );

```

The first parameter of your interpreter is the sequence of command and parameter issued by the user. The first element of the Vector is the command itself (so you can use the same interpreter function for slightly different commands). SessionData is a collection of data and classes that manages the current client connection; the most important element is the io() method which returns a GStream object, that has a useful printf like method; see the client/gstream.h(cpp) session/sessiondata.h(cpp) to have the details, or peek through the daemon/corecommands.cpp file to have lot of examples.

The third element is, for your convenience, the engine referred by the user; the parser can understand it using the 'has_eng' element of the command definition. If 'has_eng' is 0, this parameter will be null. The parser takes care of preliminary checks: it ensures that the parameter count is right, that the engine id parameter is a valid ID, that the engine referred by the user has a compatible engine type and that the user is the owner of the engine, or has rights to operate on other's engines.

The function should return an integer value, which invokes a specific behavior of GD. Return values are to be intended as "requests" that the daemon will fulfill unless it can't; return values are listed below:

```

#include <corecmd/sessiondata.h>

#define GD_REQ_CONTINUE 1 // continue to serve the client
#define GD_REQ_CLOSE_CON 0 // last command from client (i.e.
quit)
#define GD_REQ_ADMIN_ON 2 // enter admin state
#define GD_REQ_ADMIN_OFF 3 // exit from admin state
#define GD_REQ_DOWN -1 // shut down GD

```

6.3.5. Command overloading

Command overloading can be done from plugins (or any other element of GD) by simply registering a command "over" an existing one. There are basically two kinds of overload: full range and focus overloading. A focus overloading is meant to create a more specific version of a preexisting command, aimed to manage particular aspects of a given engine type and/or class.

A full range overloading will replace a preexisting command, be it a 'core' type command (available for all engine types) or a previously focused overloaded command. Overloaded commands does not need to have the same synopsis of their ancestors: in fact they can have completely different values for 'has_eng', 'maxp' and 'minp' parameters.

Some examples: let's say that you create the gstring engine type to manage sequences of letters ACGT (oops... reminds me someting :-)) and you want to create a more sophisticated version of the 'dump' command that will act for that engine. You create a gs_dump function to interpret the command, and register it with CmdManager::add("dump","gstring",... gs_dump). Now, if a user issues a dump command on another engine type, the old 'core' dump will still take control, while if the user issues a dump and the engine id refers to a gstring type engine, your gs_dump will be called.

Command overloading is sensible to plugin loading order: a command loaded in at the GD startup will be overloaded following loaded plugins, and those commands can be still overloaded again. If a plugin which defines a command is unloaded (with the unplug command), the previously defined command will take control.

6.3.6. Plugin initialization

Plugin initialization is done in two steps: first, the plugin manager kindly asks the plugin to prove that it's a fully qualified plugin by describing itself. If the plugin is successful in describing, it's initialization function will be called, where the plugin is cleared to register new engine types and commands. Finally, when a plugin will be unloaded, a cleaning up function will be called.

Each plugin must define the following functions:

```
#include <plugin.h>

/* Must initialize the data that the engine must know:
   (in example, the unique id of the plugin) */
extern "C" int gd_describePlugin( PLUGIN_DATA *dt );

/* this will start plugin; it's a hook function */
extern "C" int gd_initPlugin( PLUGIN_DATA *dt );

/* this will stop the plugin ... */
extern "C" int gd_closePlugin( PLUGIN_DATA *dt );
```

The first one is responsible to define some start-up parameters, and to describe the plugin so that the core GD can recognize it. Actually, the recognition is mutual: gd_describePlugin carries a chance for the plugin to "decline" the invite to be loaded. Let's see how it happens.

The structure passed to the plugin can be very different between GD versions. The only part that will never change is:

```
typedef struct {
    short int version;
    short int subversion;
    void *handle;
    char name[ GD_PLUG_NAME_LEN ];
} PLUGIN_DATA;
```

The plugin must peek the version and subversion values: they are the current version number of GD; if the plugin is not compatible with GD (at the sole discretion of the plugin writer), this

function have to return `GD_PLUG_ERR_VER` (or -1). Else, the function has the role to write it's name into the 'name' field of the structure, and to return `GD_PLUG_OK` (or 1). This field is the name listed by the `plst` command, and used when referring to plugin-specific command (as 'unplug'). `GD_PLUG_NAME_LEN` is 32.

If the description step is correctly completed, the `gd_initPlugin` function is called next. This function can safely cast the `PLUG_DATA` structure it receives to a version specific data structure. The structure passed to `gd_initPlugin` by geneticDaemon 0.2 is the following:

```
typedef struct {
    short int version;
    short int subversion;
    void *handle;
    char name[ GD_PLUG_NAME_LEN ];
    CmdManager *cmd;
} PLUGIN_DATA_0_1;

// NOTE: PLUGIN_DATA_0_1 does not refers to current version of Genetic
// Daemon, but
// to current version of plugin management system; so the _0_1 suffix
// can be found
// in many version of Genetic Daemon, until a major change in the
// plugin system is
// achieved.
```

Casting the parameter to a variable of this type, the initialization function can access the `cmd` element, and then register new commands. The initialization function will be also responsible to eventually register new engine types.

NOTE: since the `PLUGIN_DATA_0_1` have not been changed between 0.1 and 0.2 version, this structure can be used by plugin for both versions.

When the plugin is unloaded, it's `gd_closePlugin` function is called. It's responsibility is to unregister engine types and remove commands defined by the plugin. Function used to do that things are:

```
#include <corecmd/cmdmanager.h>
int CmdManager::del( char *name, char *type );

#include <genetic/genetic.h>
extern "C" int removeEngineType(char *type);
```

They both return 0 on failure (mistyped the command name or engine type?) and 1 on success.

6.3.7. Programmer's notes

Why did I used this double initialization style? Engine types are created with a global function, which does not require any parameter passing; it's somehow a more handy method for the final plugin writer, but it leads to a slight lesser freedom in extending the model. Commands are declared using a "master" object; methods take care of adding and removing commands to the object; this is a far more flexible and extendible method, but it carries the burden to have the need of moving an important object around, and let it handle to many foreign and uncontrolled functions.

My target here is not to describe the full advantages and disadvantages of the two methods, but only to explain why I did this choice: since, through I am able to fully describe plus and minus of

the two methods, at the moment I am not capable to evaluate which is better for the project development, I left the problem open and intentionally used this two different methods to let time, usage and developers community to decide on this question. Embracing one way or the other will be simple: the engine registration functions are simply encapsulating a meta-engine object method call (as the `CmdManager::add` and `CmdManager::del`), and the plugin data structure is capable to hold another parameter with a painless effort. So, the only question is which is better? When we'll be able to decide, I will issue the problem and make a uniform interface for "adding" dynamic elements to Genetic Daemon.

7. Questions and Answers

Nothing asked up to date...

8. Copyright

Geneticdaemon Copyright 2002 Giancarlo Nicolai , giancarlo@niccolai.ws

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

A copy of the GNU License, version 2.0, is shipped along with Genetic Daemon source files, in the file called "COPYING" under the main directory.



Genetic Daemon waves you goodbye