Gnans

A Program for Stochastic and Deterministic Dynamical Systems

Reference Manual

Edition 0.96 for Gnans 1.3

Bengt Mårtensson

Abstract

Gnans is a program (and language) for the numerical study of deterministic and stochastic dynamical systems. The dynamical systems may evolve in continuous or discrete time.

Gnans loads a *system*, a definition of a dynamical system in a special, equation oriented language. The description consists of declarations of states etc, and equations describing the dynamics of the system. As an advanced feature, arbitrary C++-code may also be contained in the system description. Gnans sorts the equations, translates them into C++, which is subsequently compiled and linked into the running program. It is then able to solve the system equations numerically with the speed of a compiled (as opposite to interpreted) program. Several numerical integrators, also for stochastic differential equations, are provided. Gnans has an intuitive user interface, making it possible control the program and to change all relevant parameters using an intuitive point-and-click interface. In this operation, it can be considered "an initial value problem (IVP) engine". Using a simple script language, this IVP-engine can be programmed. As a by-product, this offers the possibility of a command line interface. (Actually, as another by-product, Gnans contains a rather powerful pocket calculator!) Simple interactive two-dimensional plot routines are provided.

1 Introduction

When performing numerical simulations on dynamical systems, it is often advantageous to have access to powerful interactive high-level programs. In general, a researcher or an engineer is interested in his/her mathematical model and its properties, and does not want to worry about numerical algorithms or plot routines. For example, when studying systems described by differential equations, the question on selection of integration algorithm and step size does not belong to the problem domain, but are technical problems.

The present program, Gnans, is an attempt to write a program in this tradition. It defines a simple, equation-oriented language for the description of a continuous or discrete time dynamical system. The system can be of arbitrary complexity, with states, parameters (easily changeable between runs), auxiliary variables, etc. A system consists of a number of declaration and a number of equations. The equations need not be ordered. (There are presently no possibilities for structuring systems in subsystems, or object-oriented modeling.) A translator sorts the equation to an ordering that can be computed (if the equations do not contain an algebraic loop), and generates C++ code, which is subsequently compiled and linked. However, Gnans differs from older programs using the pre-processor approach by linking the code dynamically into the executing program.

Gnans was originally (version 1.0.0) written in ANSI C, but is now “rewritten” in C++. The main program consists of approximately 12000 lines of code, the translator approximately 3000 lines (in both cases not counting code from other sources, just code written for the purpose). It uses the X toolkit and the Athena Widgets. The program was originally written to be run with a graphical user interface, with possibilities to invoke scripts in a simple script language (in particular, no control structures). More complicated tasks, e.g. requiring control structures, could be implemented by having scripts in other language (e.g. Perl) writing on standard input. There is also a command line interface with powerful command line editing features.

The present document is written with the aim of being a reasonably complete documentation of Gnans 1.3, rather than being a tutorial. For a tutorial on simulation of dynamical systems, the reader is referred to other sources. Some chapters contain a section in which possible future enhancements are discussed, and the design is discussed.

The reader is assumed to be reasonably familiar with the elementary concepts of Unix and the X Window System, as well as fundamental concepts of digital simulation of dynamical systems. A tutorial and/or users guide, aimed at a less computer-literate audience, is still waiting to be written.

This manual does not cover the installation of the program. This is covered in another document, which is found together with the sources.

1.1 Design considerations and goals

Most importantly, Gnans should be a vehicle for ongoing research, both for investigating particular systems, and for the study of methods for e.g. numerical integration of stochastic differential equations. For this, particular emphasis has been put on the programmer interface to the integration routines, using object oriented programming. Gnans is intended to be “professor friendly”, i.e. allowing the expert in the problem domain to concentrate on the problem, not on "details" such as numerical algorithms and computer usage etc. (The reader may object at this point, and claim that the listed assumptions on computer knowledge assumed for this document, together with the terminology, shows that this goal has not been accomplished. That argument is wrong: This manual aims at a complete documentation of the program, and is to be thought of as a reference manual. A tutorial and/or users manual is still to be written.)

Gnans is aimed at an environment consisting of networked Unix workstations, running the X Window System. In this environment, there is no need for a simulation program to provide, e.g. command which can just as well be run as another process (editors, shell commands, printing, etc.). Its design is more geared towards the needs of mathematicians and teaching than towards engineers, which is why possibilities for structured systems and object oriented modeling are missing (at least presently). It can be pointed out that it is conceivable to write different pre-processors or graphical editors to generate Gnans code, to implement e.g. a graphical system editor or an object oriented modeling language.

A requirement has been that the program should run with compiled speed, i.e., at most marginally slower than a carefully written, optimized C or Fortran program written exclusively for a particular problem.

1.2 The name of the game

Like all programs with self respect these days, there are special esoteric rules on how "Gnans" should be pronounced, to separate the informed from the not-so informed. It should be pronounced like an English word, not "G.N.A.N.S.". The "G" in Gnans should be pronounced like in Dutch (the sound that sounds like you are tearing mattresses apart), as in "Groningen" or "van Gogh". Gnans should be capitalized "Gnans", not "GNANS" or "gnans", and typeset with normal Roman typeface, not a fixed-width typeface. Note the difference between the program named Gnans, and the command `gnans`!

1.3 Gnans is a free program

Gnans is a *free* program: the program, including its source code, is available to anyone. This is not to be "nice" to people, it is to maximize its usefulness. You are welcome to study the code, and improve it. Scientifically, a proprietary "high-level problem solving tool" for which sources are not available creates a credibility problem: How can a scientist check exactly what the program is doing, and how it is arriving at its results? Can his colleagues? A proprietary program does whatever it pleases (of course, often based on solid experience from its creators), but you simply has to trust the program (often there are ways of checking the result thought). If a scientist arrives at some result using free programs, (s)he can give it away to anyone to repeat the computations and check the results. With a program without sources, you also can not experiment with algorithms, etc, at least not in a manner the creator did not expect. You cannot learn from the program code, neither can it "learn from you". My decision to write Gnans was very much influenced by my bad experiences with proprietary, binary-only programs.

To make sure that Gnans, and possible future derived works, will stay free, I have released the code under the GNU General Public License.

1.4 Mailing Group

There is a mailing group for Gnans. Please send submissions for the mailing group to `gnans@mathematik.uni-Bremen.de`. Requests to be added/dropped to/from the list should be sent to `gnans-request@mathematik.uni-Bremen.de`.

1.5 Acknowledgments

First of all, I would like to thank all the authors of the free software which is incorporated in this program. Numerical routines for ordinary differential equations are from Netlib, and in particular I would like to mention the LSODE algorithm. Gnans would not exist without the software from the Free Software Foundation: gcc, flex, Bison. Code and libraries from FSF are: `readlib`, `getopt`, `libg++`, `texinfo`, and the signal code from Emacs. Imake and the X libraries from the MIT X11 release 5 are an important part of Gnans. The "3D-Athena widgets" look much prettier than the ordinary ones. At University of California at Berkeley the `random` routine was written, on which my random number generator is based. The Athena plot widget is used to implement the plots. (Name of individual authors are found in the source, or in README files.)

Except when clearly marked otherwise ("stolen" files and libraries, with no or very small changes), all parts of Gnans has been written by myself. The exception for this is the interface to the plot routines, which is based on work done by Patrick Rosenberg.

Work has been done by Frank Op't Hof on integrating routines for Riccati equations. This will be integrated in Gnans as soon as the vector and matrices has been added to the language in a final way.

The community of Gnans users is unfortunately not yet as large as I would desire. However, in particular I would like to thank the following people for their feedback and bug reports: Thomas Eberhard, German Giancoman, Eduardo Sontag.

2 A Simple Example

This chapter will present a simple example, without paying too much attention to details. The goal is to informally introduce the concepts, and to demonstrate how simple it is to interactively solve initial value problems with Gnans. These examples are geared towards the future user of Gnans who will define his/her own systems. It can be pointed out, that a possible use of Gnans is just to run models, translated into the Gnans language by someone else.

2.1 The Lorenz equation

We would next like to study the Lorenz equation:

$$\frac{dx}{dt} = \sigma(y - x)$$
$$\frac{dy}{dt} = rx - y - xz$$
$$\frac{dz}{dt} = xy - bz$$

where σ, r, and b are parameters, having the values 10, 28, and 2.666667 respectively. To this end, we write the following system description:

```
CONTINUOUS TIME SYSTEM lorenz;
STATE x = -1;
STATE y = 0;
STATE z = 15;
TIME t;

PARAMETER sigma = 10;
PARAMETER b = 2.666667;
PARAMETER r = 28;

AT TIME t:
d(x) = sigma*(y - x);
d(y) = r*x - y - x*z;
d(z) = x*y -  b*z;
```

A few short comments on the format: the first few lines are declarations to Gnans, it states that, e.g., x, y, and z are states in the dynamical system. (Some might prefer to say that the state of the system has components named x, y, and z, in that order). Part of the declaration is the initial condition. The declaration '`TIME t`' declares the name of the time variable. The line '`AT TIME t:`' indicates the start of the dynamical equations which are in effect at time t.

(The following description assumes that you have access to a fully ported Gnans using dynamical loading. This is presently only implemented on Sun3 and Sun4. On machines where Gnans is not ported using dynamic loading the procedure is much more primitive. See Section 14.7 [Gnans without dynamic loading], page 49.)

Create (or copy) this file in your current working directory using the filename '`lorenz.gnans`'. Probably you can start Gnans just by typing `gnans` at the shell prompt, otherwise see Section 4.1 [Invocation], page 8. Press the

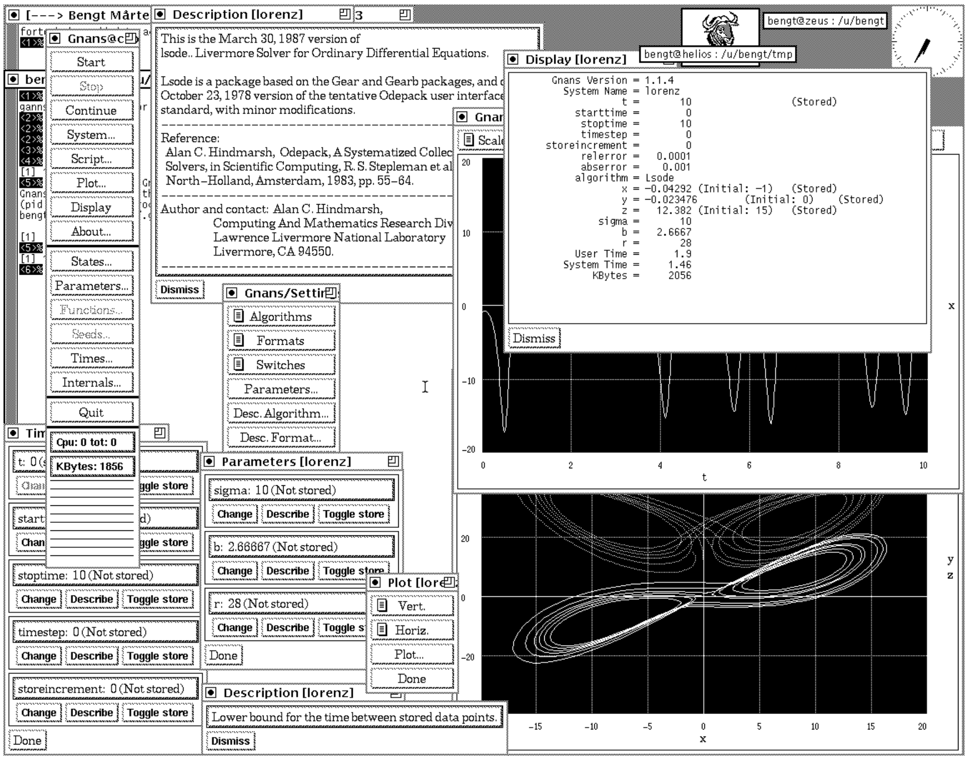

Figure 1. A possible screen when simulating a simple system.

button marked `System`, and fill in the file requester with the proper filename, in our case '`lorenz.gnans`'. Gnans will now invoke a few "friends" (compiler and loader) to create the loadable module '`lorenz.so`'. This will take a few seconds. The module is then loaded into the executing Gnans, which is now ready to run the system. You will also see that Gnans' appearance changes slightly, less buttons are grayed out, and the title of the main window has changed (may depend on your window manager). Press the button marked `Start`, which starts the run. After a short while, the run is ended, and you have received a data file in the current directory, containing the trajectories. Press `Display` to get a listing of all interesting variables, both in our system an internals to Gnans. Plot the trajectories using the plot routines: First press the `Plot` button on the main panel, then press `Plot` on the pop-up panel that just appeared. Change parameters by pressing `Parameters` and then `Change` for the parameter you want to change. Repeat the run using other parameters. Initial conditions can be changed analogously. Press `Describe` on some state. By pressing `Internals` you can change the integrating algorithm, the error bounds, and other things. Try it out by pressing everything that looks interesting...

Your screen might now look as in Figure 1.

At this stage, you might want to browse through the examples in Chapter 13 [Examples], page 37.

3 Concepts of Gnans

This chapter discusses some general aspects of Gnans.

3.1 A few terms

Developer The developer is a person debugging, improving, or extending Gnans.

System author
(S)he writes system descriptions in the Gnans system description language. Typically, does mathematical modeling. Often mathematician or engineer, but not necessarily a programmer. Can tell a semicolon from a comma, though.

User Runs the system descriptions written by the system author. Does not necessarily know how to write system descriptions, neither the difference between a semicolon and a comma.

Normally, the system author and the user is the same person, in particular when using systems which can be described by a few lines. It is quite conceivable, however, that the system author and the user are two different persons—think of the system author as an expert on mathematical modeling and the user as an industrial, economical, or political person.

An important concept in the design of Gnans is to *separate the description of the dynamical system from the experiments to be performed upon it.* Therefore there is the *system description language*, describing the dynamical system, and the *script language*, describing what is to be done with the dynamical system. See Chapter 7 [Script Language], page 18.

3.2 Concept of System Language

The systems language is not a normal programming language, neither a imperative nor a functional programming language. It is described in detail in Chapter 9 [System Language], page 23. For now, we mention that it consists of *declarations* and *equations*, where *equation* has the meaning it has in mathematics, and is not to be confused with assignment statements in programming languages.

The system description is translated to a C++ file, and subsequently compiled and loaded to an loadable file, which is loaded into the running program.

It is also allowed to include arbitrary C++ code in the system description. There may be several reasons for this:

- Some systems are not naturally described by differential or difference equations or a combination thereof.
- Accessing external disk files.
- Accessing external libraries.

This facility should only be used if absolutely necessary, since it by-passes all sanity-checking made by the translator. It gives the user quite an amount of rope to hang himself.

4 Running Gnans

This chapter describes how to invoke Gnans. The command line options are described.

Gnans is normally run using a graphical user interface (GUI). It can also be run with a command line interface.

(The following description assumes that you have access to a fully ported Gnans using dynamical loading. This is presently only implemented on Sun3 and Sun4. On machines where Gnans is not ported using dynamic loading the procedure is much more primitive, see Section 14.7 [Gnans without dynamic loading], page 49.)

4.1 Invocation

Gnans is invoked by the command

`gnans` *options system_file*

where both *options* and *system_file* are optional. Using the X Window system, running in the background (i.e. with a '`&`' character at the end of the command line) is normally preferred, unless you intend to use the command line interface later the same session, in which case running in the foreground makes more sense. *options* include all the X toolkit options ('`-display`', '`-geometry`' etc[1]), plus the following:

`-nw` ("No Window".) Use the command line interface (CLI) instead of the GUI.

`-scriptfile` *scriptfile*
: Before giving control to the user, *scriptfile* is read/executed (See Chapter 7 [Script Language], page 18 and Section 4.2 [Start-up Procedure], page 9).

`-development`
: Enables a few options and switches only of interest when debugging and developing Gnans. Exactly what this means is "undocumented", meaning, among other things, that it can change it anytime.

`-expertmode`
: Enables a few options and switches which should be hidden for the "non-expert user".

`-debug` *debugcode*
: Enable debugging options, depending on *debugcode*. Normally used only by development and debugging of Gnans itself. *debugcode* is an integer, see the code in the file '`debug.cc`'. The behavior of debugging options is to be considered as "undocumented", that is, you have to read the code, and it can change at any time.

`-version` Just prints the version number and exits.

Invoking Gnans under the name `ttygnans` is equivalent to using the '`-nw`' option.

(It can be mentioned that all these features, with the exception of `-nw`, can be achieved within a running Gnans using the GUI.) The option names

[1] The Unix command '`man X`' gives a full description of these options.

may be abbreviated as long as they remain unique. They may *not* be grouped in the sense of `getopt`.

The optional argument, *system_file*, denotes a system to be translated, compiled, and loaded before the control is given to the user.

Although the GUI is fairly consequent, Gnans is (at least presently) designed with the assumptions that `stdout` and `stderr` go to a terminal emulator (xterm or equivalent) where it is seen by the user.

4.2 Start-up Procedure

On start, Gnans first checks if the environment variable `GNANSLIB` exists. If not, a compiled-in default (typically '`/usr/local/lib/gnans`') will be used. Gnans first checks that this library is accessible. Gnans now checks for write permissions for the current working directory. Next, Gnans scans the arguments to find out whether the user explicitly requested the GUI or CLI. If none is explicit requested, it tries to use the GUI. If it fails (for example if you are not using an X server, or did not set the `DISPLAY` environment variable correctly, or your local X display refused the connection (forgot `xhost`?)), the command line interface is used. If using the GUI, X resources are now queried from the X server. Next, the command line options are scanned. Since Gnans creates save- and log-files in the current working directory, is not a good idea to run several copies in the save directory at the same time. Therefore, it creates a lock file in the current directory, prohibiting more than one copy of it to be active in the same directory at the same time. If there is a lock file already, this fails, and Gnans exits. It will next attempt to read/execute (see see Chapter 7 [Script Language], page 18) the file '`.gnans`' in your home directory, if it exists. If a *system_file* argument was given, Gnans tries to load this file just as if the `System` (see Chapter 10 [System Compilation], page 30) command was given. Finally, if a scriptfile *scriptfile* was given, Gnans tries to read/execute it, like the `Script` command, see Chapter 7 [Script Language], page 18 however, as opposed to the normal script file execution, it is not run in the "background" (technically: the Xt workprocs are not used).

Environment variables used by Gnans are listed in Section 12.1 [Environment Variables], page 35. See also Chapter 12 [Customization], page 35.

5 Commands and Variables

A command can often (but not always) be invoked both from the Graphical User Interface (GUI), from the Command Line Interface (CLI), or, which is essentially the same, from a script. The `start` command, e.g., can be invoked both by pressing the button with the name `Start`, and by writing `start` in a script, or by the terminal prompt.[1] This chapter will therefore discuss all interactive commands, without regards on how to invoke them. However, not very much of imagination is required to guess that the `Quit` button invokes the `quit` command, as does the `quit` command in the script language...

5.1 Commands

This section gives a complete list of all commands in Gnans. The reader who is reading this manual sequentially is warned that this chapter contains some forwards references.

`start` An integration[2] is started, from `starttime` to `stoptime`, using the initial conditions selected. The last point computed on the trajectory is guaranteed to be equal to `stoptime`. After completing the integration, the trajectories are written to disk using the selected file format, see Appendix D [Formats], page 62.

`stop` The currently running integration or script is interrupted, and the partially finished trajectory is written to disk.

`system` This command requires the name of a file containing the system to be loaded. This attempts to load the system with the given file name. See Chapter 10 [System Compilation], page 30.

`script` This command requires a filename as argument, the name of a script file to be read/executed (see Chapter 7 [Script Language], page 18), or an empty string (for standard input). If the script file is executable (for you) Gnans attempts to execute the file, which is suppose not to take any input or arguments and to produce Gnans script command on standard output. If it is not executable, it is assumed to contain Gnans script commands, see Chapter 7 [Script Language], page 18. If the filename is the empty string, commands are taken from standard input, see Chapter 6 [GUI], page 15. If reading from a "terminal" (in the sense of UNIX, in general from a keyboard), it returns only on end-of-file, otherwise, it also returns on the first error encountered.

`display` Takes an optional argument (only from the CLI). In the GUI, a pop-up appears, displaying the values of all interesting states, parameters, internal parameters, etc. Using the terminal interface, the variables are printed on standard output instead. A few thing are worth mentioning: You can have an unlimited number

[1] The next version of Gnans will probably be more strictly a front end–engine model, where (possibly alternative) front-end(s) communicate(s) with a computational engine, possibly on another machine.

[2] We use the term *integration* also for discrete time systems.

of display-pop-ups. The information is not updated, but to be considered as a snapshot. Secondly, calling `display` *during* an integration (or execution of a script) is permitted. The pop-up uses the Athena text widget, so you can move around with you favorite key bindings, search, copy to buffers, scroll (if applicable) etc.

`continue`
: `continue` is like `start`, except for two things: First, as initial values for the integration the final values of the previous integration will be used. Secondly, if the current system time is less than `stoptime` (assuming integrating forwards in time), the system time is not reset at the start of the integration, otherwise it is reset to `starttime`. This is the case in two situations: either the last simulation was interrupted with the `Stop` button (or with `^C` in the CLI), or `stoptime` has been changed since the previous integration. Strictly speaking, only the first behavior is the one normally associated with the English word "Continue", and if the `systemtime` >= `stoptime` it (assuming integrating forwards in time) should be an empty operation. Since "most" interesting dynamical systems in fact are time invariant (the dynamical equations do not depend on time explicitly), it often makes sense to `continue` in the weaker sense here. "Run the system for 100 seconds more!" Gnans will not allow you to `continue` without having `start`'ed first. (Not too surprising really: how can one "continue" something that has not "started"?) The state of possible random number generators are reset just as in `start`. See Chapter 11 [Random Number Generators], page 33.

`continue_init`
: (This command is presently only available from the CLI.) `continue_init` is the same as `continue`, with the exception that the state in the random number generators are not reset. See Chapter 11 [Random Number Generators], page 33.

`set algorithm`
: Takes one argument, the name of an integration algorithm to be selected. Sanity tests are also performed when starting an integration, prohibiting the user from selecting e.g. discrete time integrators for continuous time systems. The GUI includes possibilities to examine the documentation for the different algorithms. See Appendix C [Algorithms], page 59.

`set format`
: Takes an argument, the name of the format to store generated data in. The GUI includes possibilities to examine the documentation for the different formats. See Appendix D [Formats], page 62. Note: If possible, try to avoid saving large amounts of data in Ascii format. Yes, Ascii data is portable between anything, and can be read and manipulated by any text editor, but do you *really* need it? In general it takes *much* longer to convert the data between internal format and Ascii than it takes to generate it.... Ascii data takes up considerably more disk space, too.

`plot` (Presently only available in the GUI.) Pressing the `Plot` button pops up a *plotter controller*. The plotting facilities are described in detail in Chapter 8 [Plotting], page 21.

`about` This brings up information on the current Gnans version, and its copyright.

`quit` Terminates the program.

There are also commands for examining/changing/saving quantities in the currently active system: states, parameters, functions, and seeds, see Chapter 9 [System Language], page 23. Since the commands are fundamentally different in the GUI and the script language, the description will be deferred to Chapter 6 [GUI], page 15 and Chapter 7 [Script Language], page 18. Note that it does not make sense to change some of these: for *states*, only the initial condition can be changed, *functions* cannot be changed at all.

5.2 Variables

There are a number of internal parameters and switches internal to Gnans. We will describe these next. Note that the IEEE 754 special values NaN (denoting "Not a Number") and "Infinity" (can be abbreviated "Inf" and "inf") are allowed[3], and are sometimes natural to use. For example, setting '`storeincrement = Infinity`' is a natural way of saying "don't store any results except the final value". (However, the NaN and Infinity "values" may make e.g. numerical routines barf, and cause floating point exceptions.)

5.2.1 Times

`t` The system time, with name as defined in the current system. It may have another name than `t`, however, people "very often" use `t` for time.... Not changeable; stored by default.

`starttime`
`stoptime` `starttime` and `stoptime` are the starting time and ending time for an integration, respectively. Integrating backwards (`starttime > stoptime`) is allowed for ordinary differential equations. Data generated by Gnans is guaranteed not to "overshoot" (produce data points beyond `stoptime`, which would lead to ugly plots if using auto scaling). Defaults 0 and 100 respectively.

`timestep` (Only relevant if the current system is a continuous time system). For fixed step size algorithms, this is the actual step length. For algorithms with automatic step length selection, the solution is solicited at points this far apart. If `timestep` = 0, the algorithm is free to deliver the solution at times of its own choice, (if supported by the algorithm). `timestep` should always be non-negative, also if integrating backwards. Default is 0 for deterministic systems, 0.01 for stochastic systems.

[3] May not work on all machines, it depends on the implementation of the system subroutine `sscanf`.

`storeincrement`
: This is the least distance (in time) between consecutive saved points on the trajectory. With the possibly exception of the last and next last point stored, all stored points are at least this far apart. `storeincrement` is always non-negative, also if integrating backwards. If 0, all points are stored. Default is 0.

`wienerincrement`
: (Only relevant if the current system contains Wiener processes.) If `wienerincrement` $= h$, running an integration with `timestep` $= nh$ (where $n = 1, 2, 3, \ldots$) will all correspond to the same realization of the Wiener process by generating random numbers with a certain overhead. If 0, which is the normal setting, random numbers will be generated with no overhead. See Chapter 11 [Random Number Generators], page 33 for details. Default is 0.

As a general principle, by default Gnans selects to store everything that is not constant during a run, and not to store the rest. According to this rule, the system time `t` is the only *time* that is stored per default.

5.2.2 Internal Parameters

`abserror`

`relerror` These are the maximum relative and absolute (local) error in the integration. Ignored for discrete time systems and by algorithms without error control. Gnans does nothing but handles these parameters over to the numerical algorithms. Defaults are 0.001 and 0.0001 respectively.

`methodflag`
`algorithm_parameter_1`
`algorithm_parameter_2`
`algorithm_parameter_3`
: These are only available if *development mode* (see Section 5.2.3 [Switches], page 13) is on. However, `methodflag` is available also if *expert mode* is on. Integration algorithms are free to use these parameters for any purpose it desires.

5.2.3 Switches

Gnans has a number of *switches*: boolean variables changeable by the user, which changes the way Gnans behaves. Flipping a switch in some cases invokes an action. Note that some of the names below contain spaces, in which case "one space" is exactly that, not "any number of spaces and/or tabs".[4]

`logging` If this is true, all commands will be logged to a log file with the name '`gnans.log`' in the current directory. By default on.

`log script commands`
: If true, also commands executed from script files are logged. If using the command line interface, commands given from the terminal are logged even if this switch is false. By default off.

[4] I am interested to head if someone thinks this is brain damage.

`append trajectories`
: If true, the trajectories from subsequent integrations are just appended to the store file, instead of overwriting if. Nothing is written do disk until the switch is flipped to “off”. By exiting the program, the store file will also be written to disk if necessary.

`store first value`
: If true, the first value of the trajectory is always stored. If false, the first stored point is at least `storeincrement` time units larger than `starttime` (less than, if integrating backwards). Default on.

`store as default`
: If true, *STATE*s and *FUNCTION*s in the systems are all marked for storage when a new system is loaded. Default on.

`development`
: The user comes into development mode, giving him/her a few more switches and buttons. This is not of interest for normal users. The exact behavior of the development mode is “undocumented”. Note: if turning this on from the GUI, you should pop down the “Internals” panel and pop up it again, in order for the menus to be rebuilt. Default off.

`expert mode`
: The user comes into expert mode, giving him/her a few more switches and buttons. Default off.

5.3 Changing defaults

All defaults mentioned above, as well as default integration algorithm etc., can be changed without recompiling the program. See Chapter 12 [Customization], page 35.

6 The Graphical User Interface (GUI)

In this chapter we describe the GUI buttons, menus, etc. The "Graphical User Interface" is a (possible intuitive) way to *invoke* commands. The *commands* were described in Chapter 5 [Commands], page 10; this chapter describes their invocation through the GUI.

6.1 General

Considering Gnans as an IVP-engine, most "buttons" are fairly obvious. What follows is more of an attempt to "define" the actions, rather than to provide a completely idiot-proof step-by-step description. For example, we omit phrases like "Press OK (or press return) to finish you entry. Press Cancel if you change your mind and do not wish to perform the operation."

The functionality of buttons named `Ok`, `Done`, `Cancel`, and `Quit` is believed to be self-explaining. All entries from the keyboard may be terminated with `RETURN`, which is equivalent to pressing `Ok`. Also `WM_DELETE_WINDOW` (the "Delete" command from the X Window manager) is equivalent to `Cancel` or `Done`[1].

I have followed the convention found in most GUI specifications that a button causing a pop-up will have an ellipse '...' appended to its name. Buttons corresponding to pop-up menus has a menu-like bitmap to the left, and a different cursor.

6.2 Buttons

The upper panel is divided into four areas: The upper-most area contains some buttons with direct control of the IVP-engine. Next area contains buttons enabling the user to change parameters of the system, as well as internal parameters of Gnans. The third area contains only the quit button, while the fourth area is devoted to the status indicator. In more detail:

`Start` invokes the `start` command. Some buttons, like `Start` itself, will be grayed out during the integration. The program will still be responsive for new events such as mouse button presses on relevant buttons (such as `Display`), as well as X event such as expose. (Technically, the Xt workprocs are used.) When finished with the integration, the data is written to disk. During this time, the program is insensitive for input events, and the cursor cursor changes to the beloved (?) watch.

`System...` A file requester comes up, to which you respond with the name of a system description file (absolute name or name relative to current working directory). The file must end with '`.gnans`'. After clicking `Ok` (or pressing `RETURN`), Gnans invokes the `system` command, see Chapter 5 [Commands], page 10. If the translation/compilation/linking fails (see Chapter 10 [System Compilation], page 30), the error messages will be presented as a pop-up message. Otherwise, the new system will be dynamically loaded into the running Gnans. To prevent inconsistencies, most previously created pop-ups are destroyed.

[1] Not implemented yet.

`Script...` Brings up a file requester, where you enter a filename, or the empty string. In the first case, the file is read or executed (see Chapter 5 [Commands], page 10). In the latter case, the command line interface is invoked, and commands are taken from standard input (which normally is the user's keyboard, but may also be redirected from e.g. a file or a pipe). If the file name ends with a '`|`', Gnans tries to execute the file (using the `popen` system call) (after stripping of the leading `|`). The executed file should then generate Gnans commands on standard output. See Chapter 7 [Script Language], page 18 for information on how to write scripts or using the command line interface. If starting Gnans in the background, without having redirected standard input, don't forget the Unix command `fg` to get at the process again.

`Quit` After a confirmation question, calls `quit`, thereby terminating the program.

`States...`
`Parameters...`
`Functions...`
`Seeds...`
`Times...` Brings up a pop-up allowing the user to change these quantities, look at the documentation supplied, and select what should be stored or not. Buttons are grayed out when they do not make sense, i.e. if the currently loaded system do not contain any such entries.

`Internals...`
Brings up a new panel with more selections, considering the "internal parameters" (i.e. those who do not reside in the system) of Gnans. These are described in the next subsection.

6.2.1 The Internals Panel

`Algorithms`
Brings up a menu of integration algorithms possible for the current system. This may depending on the system, e.g. you are only presented with stochastic integration algorithms if your system contains Wiener processes. Pressing `Describe Algorithm` brings up a description of the selected algorithm. Calls the `set algorithm` command.

`Formats` Brings up a menu of different store formats. Press `Describe Format` for a description of the currently selected format. Calls the `set format` command.

`Switches` Brings up a menu of switches. These were described in Section 5.2.3 [Switches], page 13.

`Describe Algorithm`
`Describe Format`
Pops up a window with documentation on the presently selected integration algorithm or store file format.

6.3 Status Indicators

The lower part of the main panel is devoted to a status indicator: it shows the amount of currently allocated memory, and the consumed CPU time, both for the the ongoing/most recent integration (or script execution), and cumulative, total usage. Finally, there is an Athena strip-chart widget, indicating the progress of integration(s) versus wall time: at every update (the x-axis) it is indicated how far the simulation has proceeded (the y-axis). More exactly, the number 10 * (`systemtime` − `starttime`)/(`stoptime` − `starttime`) is showed. (The strip-chart is somewhat hard to explain and to grasp the first time, but is actually rather nice once you have understood it. For example, the slope tells you the speed of the machine you are on.)

6.4 Discussion

The GUI is implemented with the X toolkit (Xt) and the Athena (3D-) widget set. The more I learn about Xt, the more I like beer.... Its major problem, as I see it, is that its design is nicely objected oriented, but it is implemented in a primitive language which does not support object oriented programming, namely C. Passing of completely untyped pointers (in callbacks for example), which are subsequently cast to whatever appropriate **stinks**. (Or more precisely: the compiler and linker has absolutely no chance of checking the call. This is very error prone, and has probably costed me days of debugging effort.) A major redesign of Gnans should be based on the previously mentioned front end-engine model, and the preferred graphical front end could instead be implemented either with InterViews in C++ or with the more interpretative Tk toolkit. For this reason, no major effort should go into improvements of the present Xt-based GUI.

The general consensus about the Athena Widgets is that they are *ugly*. It is correct that the typical Athena program is indeed ugly. However, with a careful selection of resources (measures, colors, fonts, cursors, etc), and possibly by using the 3D Athenas instead of the original, an Athena program can look nice. A sensible *toggle* (a button that *looks* either *on* or *off*) is however remarkably absent.

7 The Command Line Interface and Scripts

The GUI and the scripting languages (which is the same as the command line interface language) are two different ways of invoking the commands described in Chapter 5 [Commands], page 10. It is therefore logical to treat the scripting language after the GUI chapter. However, the reader who has access to the GUI can skip this chapter on the first reading.

7.1 Introduction

The script language is a simple interpreted language. It has only one data type, the double precision floating point number. The syntax is very close to C, but without control structures, blocks, and functions. It consists of statements, ended by semicolons and/or newlines. A statement is either a command or an expression. If an expression is ended by a newline, its value is written to standard output, otherwise (when it is ended by a semicolon) nothing is printed. An important special case of an expression is the assignment, (`a = b`) where the value of the variable `b` is assigned to the variable `a`. Variables are the Gnans internal variables, as described in Section 5.2 [Variables], page 12 (e.g. `starttime`), the quantities in the current system, and the variables introduced by the script/terminal user. A new variable is introduced just by typing its name, it is automatically created, and, if not assigned to, initialized with the value NaN[1]. Note that some variables (like *functions* in the sense of Chapter 9 [System Language], page 23) may not been assigned to. States may not been assigned to, however, their initial conditions may. The name of the initial condition is the same as the name of the state, with an '`@`' appended. For example, the state `x` has its corresponding initial condition, available with the name `x@`.[2] Expressions may also contain calls to the usual mathematical functions like sin and log. The value of the last expression is always available in a variable named `ans` (as in Matlab).

The symbol `Pi` is recognized, and equivalent to the numerical value 3.14159265358979323846. It is not an l-value. Gnans also accepts octal and hexadecimal integers on input. An octal number is characterized by a leading '`0`', while a hexadecimal number starts with the string '`0x`'. In all these three cases, the conversion is done in the scanning phase, so the parser does not see the difference between '`Pi`' and '`3.14159265358979323846`', or between '`0x14f`' and '`335`'.

If the selected file is not executable, it is assumed to contain Gnans' command, which are read and executed until either the file ends or an error occurs. If the file selected by the `script` command is executable, it will be executed, and its `stdout` connected to `stdin` of the Gnans process, which will execute the commands as Gnans commands, again terminating on end-of-file or the first error. Control structures can often be implemented by

[1] This is according to the IEEE 754 standard, and denotes "Not a Number". Gnans tries to be very careful to ensure that quantities which have not received a value (or has received its "value" through an invalid operation such as the logarithm of a negative number) has the "value" NaN.

[2] The rationale for this funny convention is that there is a slight visual similarity between `x0` (used to denote the initial condition) and `x@`.

using other programs, written in any programming language[3], writing on `stdin` of the Gnans process.

When reading terminal input, user input is processed by the excellent `readline` library from the Free Software Foundation. This allows very sophisticated possibilities for editing the command line. A full description of the user interface to the command line editing functions is given in Appendix A [Command Line Editing], page 50. For the first word on the line, `readline` completes on Gnans script-commands, and, for subsequent words on the command line, it completes on (existing) file names. (A later version might contain more elaborate command line completion features.)

If the *first* character (*counting whitespace!*) on the line is an exclamation point '`!`', the rest of the line is taken as a shell command, and executed by the shell '`/bin/sh`'. An exclamation point in all other positions is taken as the unitary negation operator in the sense of C: after casting its argument to integer, it turns 0 to 1, and everything else to 0.

7.2 Script Language Commands

In general, there is one script language command for every command described in Chapter 5 [Commands], page 10, doing the same as the corresponding button. In detail:

`start`

`continue`

`continue_init`
: These invoke the commands with the same name. See Chapter 5 [Commands], page 10.

`quit`
: Terminates the program (i.e. does more than just quits the scripts!)[4]

`system "`*system file*`"`
: Calls `system` with *system_file* as its argument. Note that the double quotes are typed in.

`display` *display˙unit*
: This is similar to the `Display` button with the exception of the optional argument. The argument *display_unit* might be `states`, `variables`, `parameters`, `functions`, `times`, `internals`, `resources`, or `randoms`. In this case, only the requested quantities will be displayed.

`set algorithm = "`*algorithm-name*`"`
: Calls `set algorithm` with *algorithm-name* as argument. Note that the double quotes are typed in. Matching is done in a case-insensitive manner. Example: '`set algorithm = "LSODE"`'.

`set format = "`*format-name*`"`
: Calls `set format` with *format-name* as argument. Note that the double quotes are typed in. Matching is done in a case-insensitive manner. Example: '`set format = "Ascii"`'.

---

[3] Perl is a good choice, since it is very powerful and does not require compilation.

[4] There should be a confirmation for interactive use in a later version.

`switch` "*switch name*" `on`
`switch` "*switch name*" `off`

These commands sets the values of the switch with the name *switch name* to `true` and `false` respectively. Note the double quotes and that the names of some switches contain spaces. The switches were described in the Section 5.2.3 [Switches], page 13. Note that flipping a switch, also e.g. `true -> true`, might have side effects. (There is presently no way to query Gnans about the setting of a switch from the command line interface.)

`store` *variable*
`unstore` *variable*

Marks *variable* (which has to be an externally visible variable, either an internal variable, or a variable in the current system) to be store or not to be stored respectively. If the switch `append trajectories` is true, this has as side effect that the accumulated trajectories will be written to disk.

There is presently no possibility to call scripts from the command line interface.

The reader might have noted that there is a considerable amount of "syntactical sugar" in the above commands. The logic behind the use of the double quotes '`"`' is this: Double quotes are used around *strings*, and a *string* is anything not recognized by the command language itself (keywords and variables). Filenames, Gnans' switches, and names for algorithms and formats are *strings*.

7.3 Discussion

The scripting language of Gnans is what I am least satisfied with. Initially, the command language was made deliberately inconvenient (lots of syntactic sugar, no command line editing etc.), since "Gnans was a program with a GUI!", and the scripting language was for writing scripts in. Also, the GUI can sometimes be very impractical, for example when you really have to "search" for a parameter in a big panel or menu, regardless of the fact that you know exactly well its name.

Experience has also showed that the design with "dumb control language" (i.e. no control structures etc.) was not a good design. Implementing control structures in another language such as e.g. Perl has shown to be error-prone, tedious, and very non-intuitive. In particular, there is no easy way to get a value *from* Gnans. The absence of control structures is a consequence of having the Yacc (Bison actually) parser directly executing all commands as it is parsed. A redesign of this aspect of Gnans should probably be based upon the "embedded tool control language" `tcl`. A problem with using `tcl` in this context is that the syntax looks more like Lisp than the control language for an interactive mathematical program, related to the Matlab tradition, so a pre-parsing is necessary (which probably saves some of my code from the garbage can!)

To have to type double quotes in a command such as '`set algorithm = "Lsode"`' is awkward and unnatural.

8 Plotting

Gnans is not a plotting program. A previous version had no plot routines at all, motivated by the fact that there are plot programs like sand in the sea around, also free of charge, and my time was better spent doing something else. Further, in a Unix/X Windows environment, there is no need for implementing everything in one program; you instead use several different tools ("the Unix philosophy"). However, it turned out that generating data in one program an visualizing it in another was both tedious and error-prone — generating new data while plotting the old.... The plot routines are there for completeness and convenience. Suggestions for substantial enhancing of the plot routines will therefore be treated rather step-motherly — unless someone donates the code!

Presently, the `plot` command can only be called from the GUI. However, there are several reasons why plot commands should be callable from the script language. This will be incorporated in a later version.

8.1 Concepts

A *plotter* is a pop-up window depicting one or several *user visible quantities* (see Chapter 9 [System Language], page 23) as dependent variables against another such, the independent variable. The dependent variables are plotted against the vertical axis, the independent against the horizontal. (For example, the most common use is to show some state variables as a function of the simulation time. In this case, the system time is the independent variable. Selecting a state variable as independent variable produces so-called phase portraits.) (Exactly) one horizontal and one or several vertical variable thus defines a plotter.

Pressing the `Plot` button in the GUI pops up a *plotter controller*, which is a panel with two push-down menus, labeled `Horizontal` and `Vertical` respectively. With these, the user selects one independent variable, and one or many dependent variables. Finally, pressing the `Plot` button makes the *plotter* appear.

You can have an unlimited amount of plot control panels active. Each of them may have an unlimited number of plotter "children" (at least if you have unlimited virtual memory!). Gnans does not recycle plotter windows. Instead it pop up new ones, and the user can select to pop down, iconify, or just leave the old ones, according to taste, screen space, and memory. (This design was inspired by Maple V.) Note, however, that plotters for which the corresponding data is no longer in memory can no longer be manipulated.

8.2 Interactive Commands for Plot Manipulation

Once realized, the selection of the variables is fixed, but the user has certain, limited possibilities to manipulate the plot:

`Scales` This brings up a pull-down menu that allows the selection between linear and logarithmic axis. Titles like '`Lin-Log`' means that the horizontal axes is linear, while the vertical is logarithmic. Of course, logarithmic scales are possible only if the corresponding data is positive.

`Point Plot/Line Plot`
: This is a toggle which allows the selection between *point plot* and *line plot*. In *point plot*, data points are plotted as colored pixels, in *line plot* lines are drawn, linearly interpolating. Default is line plot for continuous time systems and point plot for discrete time systems.

`Zoom/Unzoom`
: This is another toggle button which switches *zoom-mode* on and off: With zoom-mode on, you mark a rectangle with the left mouse button, and, when the button is released, the marked area will be zoomed in to fill the whole plot area. Zooming in on an area of the plot may make logarithmic axes possible, even if it was not before. Unzooming may therefore reset the axes to linear.

`Grid/No Grid`
`Subgrid/No Subgrid`
: These enables the grid and sub-grid respectively. You cannot have sub-grid without grid.

`Print`
: Pressing this button generates a Postscript file in the current working directory. (You will have to issue, e.g., an `lpr` command yourself.) The Postscript file will get a name such as *system*`.gnans.`*xx*`.`*yy*, where *system* is the name of the currently active system (see Chapter 9 [System Language], page 23, *xx* is the process id of the Gnans program, and *yy* is the sequence number. To make the file into an encapsulated PostScript file, remove the `showpage` command at the end of the file. Note: The generated PostScript reflects the look on your screen, and is thus dependent e.g. on the current size of the window: A larger window for example gives denser scales on the axes.

On a color display, Gnans will use colored curves. On a black and white display (actually, if the display has at most two bit planes) different line types will be used instead. When generating PostScript, displayed colors is momentarily changed to use different line types instead, to generate sensible black and white PostScript instead.[1]

When a new integration is started, or when a new system is loaded, Gnans no longer has the data corresponding to the plot in memory. Therefore, it can no longer be manipulated (including window system operations such as rescaling!). All buttons but the `Done` button will thus be grayed out.

[1] Yes, I have heard about color PostScript and color printers, but...

9 The System Language

This chapter describes the *System Definition Language* of Gnans. In the Gnans' philosophy, we distinguish very clearly between *the dynamical systems* and the *experiments to be performed on it*.[1] Keeping these concepts apart both improves the logical structure of the simulations, and enhances code re-usability.

9.1 General

A *system description* describes the dynamical system to be studied. The system description language is not a normal programming language. A file containing a system is to be called a "system definition" or "system declaration", *not* "program" or "Gnans program". It consists of *declarations* and *equations*, where *equation* has its mathematical meaning, and is not to be confused with assignment statements in programming languages. (Strictly speaking there is the difference that the left hand side and the right hand sides are not exchangeable: those who write their differential equation $f(x) = dx/dt$ may have problems...) The equations describing the systems does not have to be entered in any particular order; you should use the same order as you would if writing a paper. The equations are automatically sorted by the translator . Time invariant equations are computed only once per integration. Quantities not needed for updating the states are computed only when necessary, i.e. when a computed point is stored.

In contrast to some other simulation languages, Gnans does not make any attempt to be smart about your system description. You have to declare *every* state, variable, parameter, etc. It simulates systems of first order differential- or difference equations. It does not make any attempts to conclude that, e.g. things occurring in a derivatives must be states, or to introduce additional states to rewrite higher order equations as systems of first order differential equations.

Recall the definitions in Chapter 3 [Concepts of Gnans], page 7 regarding *system author* and *user*. Although in practice often the same person, it makes conceptual sense to separate them: The system author writes system descriptions, mathematical models; the user performs the very experiments. Analogously to a programmer, the system author may want to hide some details to the user, while presenting him/her with others. When we say "the user can change" we will mean that (s)he can do that without recompiling, or without even looking at the system code.

The Gnans system description language has been designed to be a reasonably simple and logical language for a formal specification of a dynamical system. It does not represent the highest level of user friendliness. No attempts has been made to, e.g., make the parser understand everything an intelligent person would have understood (like forgotten semicolons). It can be pointed out that it would be quite a sensible project to write different, user friendly front-ends to the Gnans system file: a graphical system editor, in which the user selects elements like integrators, adders, multiplexors, low-pass filters, PI controllers, Wiener processes, water turbines, diesel engines, etc., connected together with the mouse, and which as output generates a

---

[1] However, some examples in Chapter 13 [Examples], page 37 will show that it can sometime make sense for the user to violate this.

Gnans system is quite conceivable. Correspondingly, for those who consider it the ultimate in user friendliness to let the user answer questions, it would be quite a simple task to write a simple program to generate a system descriptions from "Enter the dimension of the state space"-type of questions.

A Simnon user would complain about "only one system file", he or she would like to compose a system out of several subsystems, connected by one connecting system. If all subsystems are continuous time systems, or if all subsystems are discrete time, it is easy to see that using only one system description is not a restriction on what you can do with the language. When hybrid time systems are implemented, it will not be a restriction for mixed systems either. It may be a restriction on the conceptual level, though.

A formal grammar of the system description language is given in Appendix F [Grammar Of The System Description Language], page 66.

9.2 Concepts

Gnans is always case sensitive. The name '`A`' is completely different from '`a`', as is natural for a mathematician, an engineer, or a C programmer, but highly annoying for a Fortran or Pascal programmer.

All reserved words in the Gnans system language are written in CAPITALS.

A *user visible quantity* is something the user can access, change (provided that it makes sense), and which may have a documentation associated with it. The user may also store it for points of the trajectory. This is information the system author makes public to the user.

For identifiers ("names") the same rules applies as in C++: an identifier consists of sequence of (Ascii) alpha-numerical characters, the first of which is a character (the underscore '`_`' is counted as character). Unfortunately, this means that ISO 8859-1 characters are not permitted. This stupid and unnatural restriction is due to the fact that the resulting C++ file is passed through a standard C++ compiler. (However, 8 bit characters are permitted in the system name.) Names starting with '`gnans_`' should not be used, since it may conflict with internally generated names. Names starting with an underscore '`_`' may lead to problems with the linker.

An identifier used for a state, parameter etc. *may not be used for any other purpose*, including local variables and formal parameters in raw C-code. (This is due to a design flaw, which should be possible to fix.)

Comments may be written anywhere. The syntax is as in C++: Either enclosed between '`/*`' and '`*/`', or from '`//`' and to end-of-line.

The system file is passed through the m4 macro processor before the translator sees it. You can thus define macros, use include files, etc. However, note that there are some dangers involved in using macros: It can be error prone, since the macro processor does not know anything about the language (use parentheses around arguments and the resulting expression!). The syntax will often look bizarre, and it is simple to write very unclear code. Error messages will refer to the code *after* macro expansion, which is confusing. In modern software engineering, most uses of macros is frowned upon.

9.3 Structure of a Gnans System

A Gnans system file consists of a declaration section followed by one time block. The declaration section consists of a number of declarations. Every declaration is ended by a semicolon. The first declaration has to be a declaration of the system, like

`CONTINUOUS TIME SYSTEM` *system_name*`;`

where *system_name* may contain also the 8-bit characters corresponding to *letters* in the ISO 8859-1 (Latin-1) character set.

Presently, Gnans handles `CONTINUOUS TIME SYSTEM`s and `DISCRETE TIME SYSTEM`s. A discrete time system is a difference equation, a dynamical system with the set of times being the set of the integers. Future versions will handle `HYBRID TIME SYSTEM`s and `DAS SYSTEM`s (differential algebraic systems), and possibly also differential equations of some special structure.

After the system header, as above, follow a number of declarations of states, parameters, functions, integer states, noise, and Wiener processes. There has to be exactly one time declaration.

9.4 Declarations

States are components of the state vector in the sense of a dynamical system. They are user visible. The user may change its initial value, but not the state itself. They are declared with the command `STATE`. The syntax is

`STATE` *name* `=` *initial_condition*`;`

or

`STATE` *name* `=` *initial_condition* `"`*documentation_string*`";`

where *name* is the name of the state as the user sees it, *initial_condition* is the default initial condition (a real constant), and the optional *documentation_string* is the system authors message to the user. (Don't take the word "documentation" too literally: Feel free to put anything sensible here. For example, it can be pretty handy to have the exact look of the equations available to the user.)

A typical declaration can be

```
STATE omega = 3 "Angular velocity in rad/s";
```

where `omega` is the name of the state (the entity to be declared), 3 is the default initial condition (which the user can change). The documentation string is there for convenience. It is not read by any program, just by humans.

Parameters are constants during an integration. They are user visible, and may thus be changed between integrations. Syntax is

`PARAMETER =` *default_value* `"`*documentation_string*`"`

where `"`*documentation˙string*`"` is optional. (A documentation string is always optional, we shall therefore not mention it for the rest of the chapter.)

Functions are quantities depending (statically) on other quantities in the systems. (The name is possibly badly chosen, they correspond neither to the mathematics nor the computer science meaning of the word.) The are user visible, but may not be changed (which would not make sense). The declaration syntax is

`FUNCTION` *name* `"`*documentation_string*`";`

Variables are like functions, except for that they are not user visible. They correspond to the C data type `double`. Syntax is

`VARIABLE` *name*`;`

i.e. no documentation allowed.

Integer states are simply integers. They are not externally visible. They are declared as

```
INTEGER STATE name;
```

Uniform white noise and *normal white noise* only makes sense in discrete time systems. These are simply a sequence of independent, identically distributed (i.i.d.) (pseudo-) random variables from the U(0,1) or N(0,1) distribution, respectively. Declaration:

```
NORMAL WHITE NOISE name = seed "documentation-string";
```

and

```
UNIFORM WHITE NOISE name = seed "documentation_string";
```

respectively.

The *seed* is used as default seed for the corresponding random number generator. Repeated runs with the same seed will produce exactly the same sequence. Setting the seed = 0 will have the effect that the seed will be derived from the system clock[2], thereby introducing a certain unpredictability, or an automatic way of selecting new seeds. If a noise source is marked for storage, the seed will be stored, not the actual noise. To save the noise realization, introduce a `FUNCTION` which is equal to the value of the noise. All noise sources are independent both in the statistical and “real” sense. They all have their own private seed. The random number generators are treated in detail in See Chapter 11 [Random Number Generators], page 33.

As opposed to other simulation programs, *Wiener processes* are first class citizens. A Wiener process is declared by

```
WIENER PROCESS name = seed "documentation_string";
```

For seeds, the same rules apply as for uniform and normal white noise. “Storing” a Wiener process stores the seed, not the actual realization.[3]

The translator checks the equations to sort them, and to check that all symbols make sense. If using raw C code, it may thus be necessary to tell the translator that a certain name should be considered as resolved. The declaration is

```
C-IDENTIFIER name;
```

This declares *name* as a quantity the translator should consider as well known. It does not correspond to a user visible variable.

C preprocessor commands (defined as lines starting with a `#` character, extending to the end of the line) are passed unprocessed to the output.

For all systems, there is only one “time”, to be declared with the declaration `TIME`. This is mandatory even in time invariant, continuous or discrete time systems. Syntax is

```
TIME name;
```

No documentation is allowed.

After the standard declaration, but before the time blocks, there may be a section of raw C++ code. Example:

```
C-DECLARATION
{ /* Arbitrary C++ code */ }
```

---

[2] Presently, this depends on the time in *seconds*, so that starting more than one run within the same second will yield the same seed. This may be a problem if using scripts and the runs are shorter than one second.

[3] To study the actual realization, introduce a new state like this: Assume that you have declared ‘`WIENER PROCESS W = 12345`’. You then introduce the state ‘`STATE B = 0`’ and the (trivial) stochastic differential equation (in the ‘`AT TIME t`’-block) ‘`d(B) = d(W)`’.

It should be enclosed within a pair of matching braces.

9.5 Time Blocks

A time block describes the equations which are in effect during the named time. Dynamical equations are all in the `AT TIME t`-block (assuming that the system time is called `t`). The equations in a time block are, if possible, sorted in an order so they can be sequentially executed. If the equations contain a circular dependence, the sorting will fail. Gnans can (at least presently) not solve algebraic loops. (It can be claimed that in this case, the user has not written complete and consistent equations for the systems.) Also other sanity tests (like that every derivative is assigned to exactly once) are also performed.

Differential equations are written like

`d(`*state*`) = `*rhs*`;`

Here `d( )` denotes the differential operator, and `d(x)` should here be interpreted as dx/dt. On the right hand side, almost any scalar C++ expression is allowed: The exponential operator, denoted either '`**`' or '`^`', has been added to the C++ syntax. Further, the sequencing operator (the comma) is not allowed, since an expression sequence can hardly be an "equation". The equation, which may run over several lines, is terminated with a semicolon.

For discrete time systems, i.e. difference equations, the dynamic equation takes the form

`q(x) = f(x);`

Here we think of `q` as the forward shift operator: `q(x(t)) = x(t+1)`.

A stochastic differential equation is entered as diffusion equation. The precise syntax is unfortunately quite complicated in order to parse "everything that makes sense". The basic equation is

`d(`*state*`) = `*drift*`*d(t) + `*noise-gain*`*d(`*Wiener_process*`);`

where *drift* and *noise-gain* are scalar expression. The first term can be left out, corresponding to *drift* $= 0$. If *noise-gain* (*drift*) $= 1$, then '*noise-gain* `*`' ('*drift* `*`') can be left out. The '`+`' character can be replaced by a '`-`' character. There may be many noise terms. (Wow!) To summarize: There is an (optional) drift term, ending with '`d(t)`', and one or more noise terms, ending with '`d(`*Wiener_process*`)`'. This corresponds to a stochastic differential equation of the type $dx = f(x)\,dt + g(x)\,dw$. The reader undoubtly asks: Is this to be interpreted in the sense of Itô or Stratonovich? The answer is: It depends on the algorithm chosen, it is free to select either interpretation. Presently, all algorithms for stochastic differential equations produce the Itô solution, but there is no reason to hard-solder this decision into the *syntax* of the language.

A time block may also contain a block of arbitrary C++ code. It should occur first in the time block. It will be executed before the equations. There is presently no possibilities to put raw C++ code *after* the equations are executed. (However, one way to circumvent this would be to use function calls, e.g. of the type '`d(z) = f(z) + function();`' where function always returns `(double) 0`.)

Declarations of states etc. are also allowed in the time block, as long as a quantity is declared before it is referenced to. This feature is necessary for certain machine generated system descriptions (e.g. using macros). It is not the recommended practice to take advantage of it in manually written system descriptions.

Four *times* are predefined: `START`, `STOP`, `STORE`, and `ACCEPTED`. The exact meaning are as follows: `AT TIME START` contains equations and/or C-code to be executed before the main integration. Time invariant equations belong here. However, the sorting mechanism detects time invariant equations, and automatically puts them in the `AT TIME START` block. For example, files can be opened in this block. If an error is detected (for example, opening of some files failed), the C statement '`return false`' should be executed. This will stop the simulation. Before this, the user should be presented with a "nice" message using the function '`gnans_error(severity s, const char *fmt,...)`' with first argument being `non_fatal`, the second argument a format string in the sense of `printf`, following an arbitrary number of arguments. An error message will be formatted from these arguments just as `printf`. (See the example in Section 13.2 [Iterated Function System], page 37.) This block is not called in "continuation runs", i.e. by the `continue` command. Analogously, `AT TIME STOP` contains instructions to be executed at the termination of an integration, typically closing of open files, or calling destructors. `AT TIME ACCEPTED` contains computations to be made on points which are "accepted": known to be on the trajectory (recall: the integration algorithm (in particular if of Runge–Kutta type) may make many "trial evaluation" of the `AT TIME t` block, and these need not correspond to point on the computed trajectory). Therefore, all computations in this block must not contain side effects. If computing e.g. statistics of the solution, here is where to put it. `AT TIME STORE` is called every time before something is stored: Functions introduced to "study things", such as the norm of the full state. Note that the sorter is smart enough to take non-dynamic equations from the `AT TIME t` section to the `AT TIME STORE` section.

Dynamical equations (differential equations and difference equations) are only allowed in the `AT TIME t` block. Algebraic equations may occur only in the `AT TIME t` and the `AT TIME START` block.

9.6 Performance Tips

To write code that executes as fast as possible, try to break up the computations in time varying and time invariant equations. If you write

```
d(x) = expensive_function(constant_argument) * x;
```

then '`expensive_function(constant_argument)`' will be computed every step in the integration. By instead writing

```
d(x) = thingy * x;
thingy = expensive_function(constant_argument);
```

the translator will recognize the second equation as time invariant, and move it to the `AT TIME START` section, so that it will be computed only once.[4]

The same argument applies to the `AT TIME STORE`-block. Make sure that the translator can move as many computations as possible to the `AT TIME STORE` block!

See Chapter 10 [System Compilation], page 30 on how to check the output from the translator.

[4] There is a possibility that a future version will take care of this automatically.

9.7 Discussion

The system language is quite consistent in its rigidity and redundancy. It may be an advantage to force the system author to make clear to himself exactly what are the states, and what quantities are memoryless functions of the other quantities. Also, to stress the state space description, rejecting higher order equations may have pedagogical advantages. The alternative, to accept equations on the form, say, '`d(d(x)) =` *something*' means that the translator implicitly introduces states.

For very simple systems, such as the Lorenz equation, the system description language with all its declarations feels somewhat redundant and verbose. For larger examples, on the other hand, the "unstructured" nature of the system description language is annoying. For example, some states are often completely uninteresting, and it would be desirable to hide them to the user. Particularly if using the GUI, a large number of user visible variables very often leads to lost overview.

A more object oriented modeling language would have been desirable. A more modest improvement would be to allow vector and matrices in the systems.

The Gnans language was designed so that it could be written not only by humans, but also by machines. I.e. you can think of a number of different front-ends, generating a Gnans system from, e.g. a graphical system editor, or even from an "Enter the dimension of the state space"-type dialogue. An interesting possibility would be to define an object-oriented language ("Gnans++"?) and a translator that translates this language to Gnans' system language.

A subsystem structure can be implemented using a macro preprocessor. The next edition of this manual will show some examples using the m4 macro processor.

10 System Compilation

Gnans works with the pre-processor model for a simulation program: The system description is translated into a compiled language (C++), which is subsequently compiled. It differs from older programs using the same principle by using *dynamic loading*, i.e. the thus created compiled module is dynamically loaded into the executing program. This chapter will describe the procedure in detail.

Ideally, the user should not need to read this chapter (at least the user who never ever makes mistakes!). You may select not to read it until you encounter problems with some of your system files. The purpose of the chapter is to provide a background on the translation/compilation/linking process. It should help the user diagnose his/her mistakes, and (since Gnans is unfortunately not guaranteed to be perfect) to help diagnose bugs in Gnans *or in its environment*. (It should be pointed out that Gnans is a program that works more intimately with the operating system environment than most other programs. If something does not work on your system, it may be that either Gnans *or* you system is to blaim. In particular, this goes for include files and the linker.)

10.1 Make

The exact steps involved in the creation of the '`.so`' file are as follows: The task is carried out by the Unix program `make`, using a '`Makefile`' automatically generated during the installation of Gnans. The make program avoids unnecessary work, where "unnecessary" is considered to be when the target already exists, and is younger than the source. The steps are as follows:

When trying to create the '`.so`' file, Gnans first checks if there is one already in the same directory, and if it is younger than the system file. If so, that one is used instead of creating it anew, which saves some seconds. (If the creation succeeds, no other files are left in the current working directory.) Since the '`.so`' file is fairly small (typically < 50kB) it is probably a good idea not to delete it.[1]

1. First the system file is processed by the m4 macro processor and subsequently by the Gnans translator, which is a separate program named `gnanslator`. A C++ file will be generated. If the system file does not obey the syntax of the system description language, see next section, the translation will fail, with an error which hopefully is understandable.
2. The generated C++ file is compiled. The optimization level used can be controlled with the environment variable `GNANSCOMPILECFLAGS`, see Chapter 12 [Customization], page 35. Unless UNSAFE status is returned from the translator, if this step fails it is to be considered as a bug in Gnans or in its environment. However, warnings about implicitly defined functions are to be taken seriously, and will indicate a real

[1] However, it is not guaranteed to be compatible between different Gnans versions, or different operating systems. You should therefore delete all '`.so`' files when the Gnans version has changed, or when they were generated under another operating system release. Possibly a future version will check this automatically — but the present one doesn't.

error. They indicate that the system called some function the compiler did not know about, say that you wrote '`sine(x)`', instead of '`sin(x)`'.

3. A sharable object, the '`.so`' file, is generated using the `ld` program. (If this step fails, it is probably to be considered as installation error.)

See Section 14.1.1 [Buggy Sun Loader], page 47 for a problems associated with some versions of the dynamic linker in SunOS.

10.2 Fixing syntax errors

Just as when writing programs in any programming language, or text for any text formatter, you *will* make mistakes that needs to be found and fixed. Unfortunately, it can also not be excluded that you will encounter bugs, in which case you are kindly asked to report reproducible failures.

If the translation/compilation/linking failed, you will get error messages. First, you have to see if the error came from translator: if the error messages are prepended by '`gnanslator:`' they are. If an input file does not conform to the syntax (for example: misspelled keywords, unbalanced parentheses, or forgotten semicolons), you will get error messages. These are called *syntactical errors*. If you get a syntactical error, the translator will tell you where it found the error. (The translator does not attempt error recovery, but just gives up after the first error.) If you desire, you can run the translator outside of Gnans, typically with a command such as

```
/usr/local/lib/gnans/gnanslator system.gnans
```

or

```
m4 system.gnans | /usr/local/lib/gnans/gnanslator
```

if you are using m4 commands.[2] (The error messages thus generated can be parsed by the `next-error` command in GNU Emacs (bound to `M-``).) There is also a debug option in the `gnanslator` program, see the code (in file '`gnanslator.cc`') for the exact meaning. (You should probably have some background in compiler technique to take advantage of this.)

You will also get error messages for doing some other thing which are not allowed, such as leaving some variable unassigned, or assigning them twice. See Chapter 9 [System Language], page 23. In this case, pointing to a particular line would in general be meaningless.

If the translation succeeds, the translator informs the user how sure it is concerning the correctness of the program: If no raw C++ code and no unknown function calls were encountered, it considers the program as *safe*, all subsequent steps of the creation and loading should succeed. If calls to functions not known to the translator are found, *semi-safe* state is returned, the compilation following should succeed, but errors may occur during the loading phase. By *unsafe* status, either the compilation or the loading may fail. It can be noted that the translator knows the normal mathematical functions such as '`sin`', and checks its usage, instead of consider it as semi-safe. It will detect if you call '`sin`' with two arguments or '`atan2`' with one.

If the translation succeeded with safe or semi-safe status, the compilation should succeed. If not, you have found a bug in Gnans (or its environment), which should be reported. Note that the translator puts line numbers cor-

---

[2] More precisely, the command is '`$GNANSLIB/gnanslator`', see Section 12.1 [Environment Variables], page 35.

responding to the system description file in the C++ file it generates, so that `M-`` or symbolic debuggers will point to the '`.gnans`' file, not the C++ file.

Similarly, if *safe* status is returned, the linking phase should work, otherwise it is a bug.

11 Random Number Generators

Underlying the "high-level" concepts *white noise* and *Wiener processes* are the *(pseudo) random number generators (RNG)*. A (pseudo) random number generator is a (completely deterministic) dynamical system, which generates a sequence of real numbers (in the sense of computer arithmetics) having statistical properties close to a sequence of independent stochastic variables from a desired distribution. The sequence is completely determined from a user-supplied *seed*, which here is a non-negative integer less or equal to $2^{31} - 1 = 2147483647$.

The seeds of the random number generators can be accessed and changed from the GUI or from the CLI just like any other variable. Note that the seeds corresponding to Wiener processes get names with '`d_`' prepended to the name of the Wiener process. For example, if your Wiener process is called `W`, the seed variable will be called `d_W`[1]. Also note that storing a noise source or a Wiener process does not stores its realization, just the seed (which is a constant). To store the realization, you must introduce another state.

All noise sources are independent of one another, both in statistical sense, and in the common sense. (However, be sure to give them different seeds, otherwise the same noise sequence will be generated, which will of course severely violate assumptions on independence of the noises.) For example, if adding another noise source to the system, the old ones will still generate exactly the same sequences as before.

For solving stochastic differential equations with different step size, but with the same noise trajectory some extra machinery is necessary. This is achieved by setting the variable `wienertick` to a positive value. By selecting `stepsize` to be an integer multiple n of `wienertick`, at each step n random numbers are added, and after suitable normalization, the sum is used as the Wiener increment.

The rest of this chapter can be skipped on the first reading.

11.1 Internals

Gnans uses a modification of the random number routines `random`, found on all modern Unix system. It is derived from the Berkeley sources. It consists of a nonlinear feedback Fibonacci generator. Its state consists of 63 32-bit numbers. The feedback polynomial is $x^{63} + x + 1$. The period is greater than 2^{69}.[2]

This generator generates an integer in the interval $[0, 2^{31} - 1]$ which is normalized to produce a uniform distributed random number. Normally distributed random number is generated by the inverse transform method, using a rational approximation of the normal distribution. See the code in '`random.h`' and '`random.cc`' for the exact algorithm used.

[1] The logic behind is that the random number "is" the infinitesimal increment of the Wiener process, not the Wiener process itself.

[2] On a Sparcstation IPX, this RNG generates 1 million random numbers in roughly one second. This means that the cycle would not repeat itself until after 19 million years.

11.2 States

As opposed to most other random number generators, the *seed* and the *state* in these RNG are different. The state is truly the state of a dynamical system, while the seed is a way to provide the user with a convenient way of initializing the RNG in different ways. Entering 63 bytes for a seed would not be feasible! However, this raises the need to be able to store the state, e.g. to be able to continue the simulation. A certain number (presently 32[3]) of states can be saved for each RNG.

To save the state of a RNG, use the command

`save_state(`*name*`, `*store_index*`)`

where *name* is the name of the RNG, and *store_index* is an expression, which when casted to integer, has a value between 0 and 31. There is also an alternative form,

`save_state(`*name*`)`

where Gnans selects the store number itself. In both these forms, the number of the stored states is returned. You would normally use the latter call in the form *foobar* = `save_state(`*name*`)`. `save_state` is to be thought of as a function (with side effect), not as a command, but since expressions are valid as commands, the difference is minute.

To reset the state of a RNG to a state previously saved with `save_state` the call

`restore_state(`*name*`, `*store_index*`)`

is used. An error will occur if no state has been saved for the RNG named *name* with number *store_index*.

The `start` command resets the initial values of both the states (in the normal sense) and the RNGs, according to their seeds, while the `continue` command resets neither. The command `continue_init` resets the states, but not the RNGs. See Section 13.4 [Pullback], page 41 for an example of its use.

11.3 Discussion

These RNGs are very popular in the Unix community, where they are available with the name `random`. In particular, it is preferred over the older routine `rand`. Simple tests I made did not reveal any problems. In any case, it would be good to have alternative RNG at hand.

The generator is quite fast: 1 million $U(0,1)$ variables was generated in 0.9 seconds, and 1 million $N(0,1)$ variables in 6 seconds on a Sparcstation IPX.

[3] This is a compilation constant.

12 Customization

Without re-compilation, Gnans can be customized by using environment variables, resources, and a startup file. This chapter describes how.

12.1 Environment Variables

Gnans uses the environment variable `GNANSLIB`. This points to the root of a small library tree, containing different support files necessary for Gnans to work. A standard location for this is '`/usr/local/lib/gnans`'. This environment variable exists mainly to be able to install Gnans in another directory from which it was compiled. If possible, please try to compile Gnans with the correct directory, instead of setting this. If you install the Gnans library in another location, you have to make sure that the dynamic loader finds the dynamic library ('`libgnanslib.so`' or '`libgnanslib.so.1.3`'), for example by manipulating with `LD_LIBRARY_PATH`, see '`man ld`'. It would probably be a good idea to write a shell script wrapper for this.

If the environment variable `AFMPATH` is defined, it is used as a path to search for `afm`-files (Postscript font metrics files). These are only needed for generating PostScript files from the plots. (However, Gnans quits quite ungracefully if a requested `afm`-file is not found.)

The optimization level can be controlled by the environment variable `GNANSCOMPILECFLAGS`. Basically, heavier optimization trades execution time for compile time (and memory). Setting `GNANSCOMPILECFLAGS` will override `CFLAGS` in '`$GNANSLIB/Makefile.gnanscompile`'. (Note that the `-fpic` option is normally required.) In my experience, when using *gcc*, heavier optimization than `-O1` does not improve performance. Note: since an environment variable is used, you have to set this *before* you invoke the program. Also, read the log file '`gnans.log`' for debugging.

12.2 Resources

As all X toolkit-based programs, there is a large number of *resources* which can customize the way Gnans behaves. The reader is assumed to be familiar with X concepts such as widgets and their customization with resources. Due to the `editres` protocol and program, the user can interactively find out the widget hierarchy using this program, and experiment with different settings of the resources. For this reason, a documentation of the widget hierarchy has been omitted. The app-defaults file can also be consulted, and possibly changed. In this way, it is easy to, e.g., label all buttons in German, if anyone would desire to do so.

There are a few other resources which govern the way Gnans behaves, which is not directly tied to the GUI. These resources, and their default values are (unless changed in the local installation):

```
starttime:                          0
stoptime:                           100
timestep:                           0
storeincrement:                     0
wienertick:                         0
relerror:                           0.0001
abserror:                           0.001
floating_point_format:              ===> %.8g\n
```

```
default_store_format:                   Matlab
default_algorithm:                      LSODE
default_stochastic_algorithm:           Euler-Maruyama
default_stochastic_timestep:            0.01
cmd_prompt:                             [Gnans]
do_logging:                             True
```

Each resource is believed to be self-explanatory, possibly with the exception of `floating_point_format`: This is a format string (in the sense of the C subroutine `printf`(3)) used for the printing of numbers in the command line interface.

Each resource has a corresponding `class`, whose name is formed by capitalizing the first letter of all words, like `Storeincrement` and `Cmd_Prompt`.

Note that resources are not available if running Gnans without X.

12.3 Startup File

At start, Gnans will search for a file named '`.gnans`' in your home directory. If found, it is read or executed. This can be used e.g. to change default values, among many other things. See Section 4.2 [Start-up Procedure], page 9.

13 Examples

In this chapter, we show some examples of nice Gnans files. The emphasis is more on demonstrating certain techniques, than on the pedagogical side. We will comment only very sparsely on the examples.

The reader should have access to all files in his Gnans installation. The reader is encouraged to run the examples himself/herself.

13.1 The Henon Attractor

A simple discrete time system. Run for, say, 10000 steps and plot `y` as a function of `x` using point plot. See Figure 2!

```
DISCRETE TIME SYSTEM henon;
STATE x = 1.16986;
STATE y = -0.0850863;
PARAMETER alpha = 1.4;
PARAMETER beta  = 0.3;
TIME t;

AT TIME t:
q(x) = y + 1 - alpha*sqr(x);
q(y) = beta*x;
```

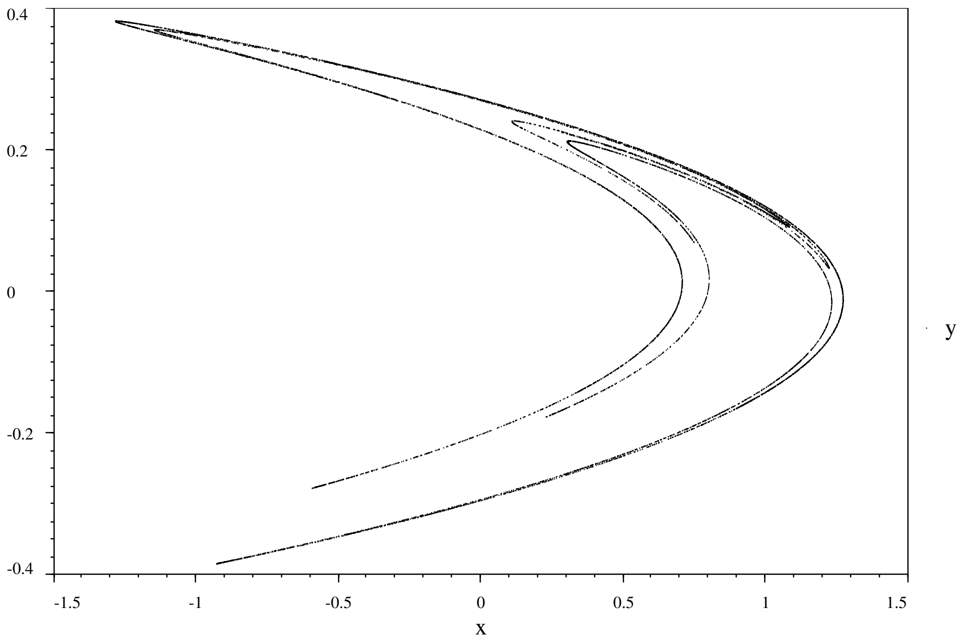

Figure 2. The Henon attractor.

13.2 Iterated Function System

This example illustrates the use of blocks of raw C++ code. It is a so-called iterated function system (IFS), parameters are taken from Barnsley's book. Since an IFS is not naturally describable by a set of difference equations, invoking raw C++ code is here natural.

Run the system for, say 100 000 steps, and plot `x2` as a function of `x1` using a point plot. The result is shown in Figure 3.

Note the error check in the `AT TIME START` block.

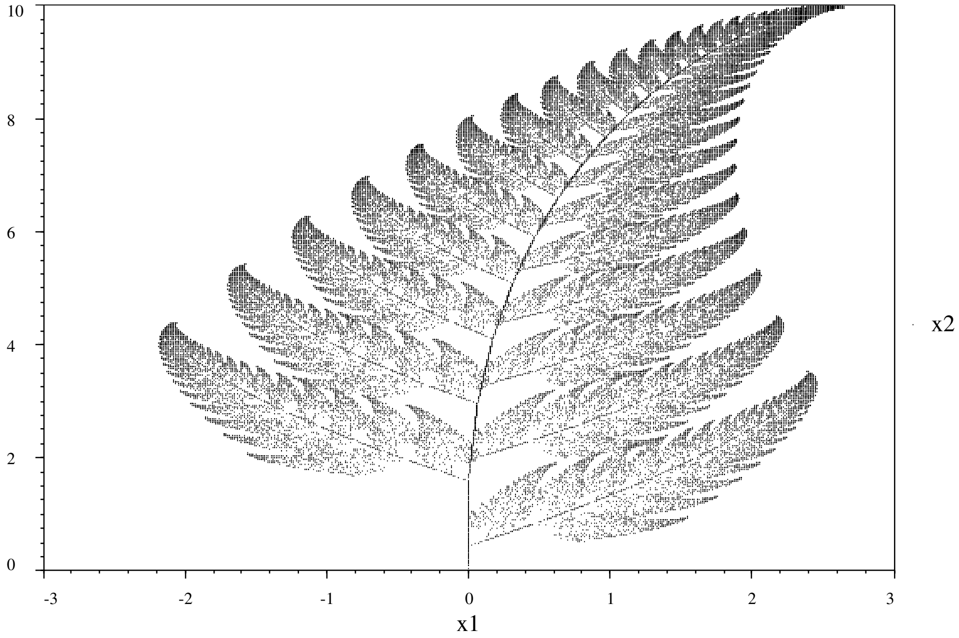

Figure 3. Iterated Function System.

```
DISCRETE TIME SYSTEM ifs;
STATE x1 = 1;
STATE x2 = 0;
TIME t;

PARAMETER p1 = 0.01;
PARAMETER p2 = 0.85;
PARAMETER p3 = 0.07;
PARAMETER p4 = 0.07;

PARAMETER a1 = 0;
PARAMETER a2 = 0.85;
PARAMETER a3 = 0.2;
PARAMETER a4 = -0.15;

PARAMETER b1 = 0;
PARAMETER b2 = 0.04;
PARAMETER b3 = -0.26;
PARAMETER b4 = 0.28;

PARAMETER c1 = 0;
PARAMETER c2 = -0.04;
PARAMETER c3 = 0.23;
PARAMETER c4 = 0.26;

PARAMETER d1 = 0.16;
PARAMETER d2 = 0.85;
PARAMETER d3 = 0.22;
PARAMETER d4 = 0.24;

PARAMETER e1 = 0;
PARAMETER e2 = 0;
PARAMETER e3 = 0;
PARAMETER e4 = 0;

PARAMETER f1 = 0;
```

```
PARAMETER f2 = 1.6;
PARAMETER f3 = 1.6;
PARAMETER f4 = 0.44;

C-IDENTIFIER a;  // Name of arrays, in the sense of C
C-IDENTIFIER b;
C-IDENTIFIER c;
C-IDENTIFIER e;
C-IDENTIFIER f;
C-IDENTIFIER i;

UNIFORM WHITE NOISE omega = 171713;

C-DECLARATION
double a[4], b[4], c[4], d[4], e[4], f[4];
int i;  // Index for the selected mapping

AT TIME START:
{
// If the user entered non-sensical ``probabilities'', we just
// refuse to run
if (fabs(p1 + p2 + p3 + p4 - 1) > 0.0000001) {
    gnans_error(non_fatal,"The probabilities do not sum up to 1");
    return false;
}

a[0] = a1;
a[1] = a2;
a[2] = a3;
a[3] = a4;

b[0] = b1;
b[1] = b2;
b[2] = b3;
b[3] = b4;

c[0] = c1;
c[1] = c2;
c[2] = c3;
c[3] = c4;

d[0] = d1;
d[1] = d2;
d[2] = d3;
d[3] = d4;

e[0] = e1;
e[1] = e2;
e[2] = e3;
e[3] = e4;

f[0] = f1;
f[1] = f2;
f[2] = f3;
f[3] = f4;
}
AT TIME t:
{
// i should get the value j with probability p{j-1} (j = 0,1,2,3)
i = (omega < p1)  ? 0 :
    (omega < p1+p2)  ? 1 :
    (omega < p1+p2+p3)  ? 2 : 3;
```

```
// Use the i'th affine mapping
q(x1) = a[i]*x1 + b[i]*x2 + e[i];
q(x2) = c[i]*x1 + d[i]*x2 + f[i];
```

13.3 The van der Pol–Duffing oscillator

This example contains a stochastic differential equation. Since Gnans parses only stochastic differential equations written as diffusions, the equations has to be slightly rewritten from the form of "noisy coefficients".

Also note the use of the "documentation string". For the first state, we use it as a scratch pad, to have the exact look of the equation handy for the user. Also note the use of a newline within the documentation string.

Figure 4 shows a typical trajectory, generated with `sigma3` = 1.

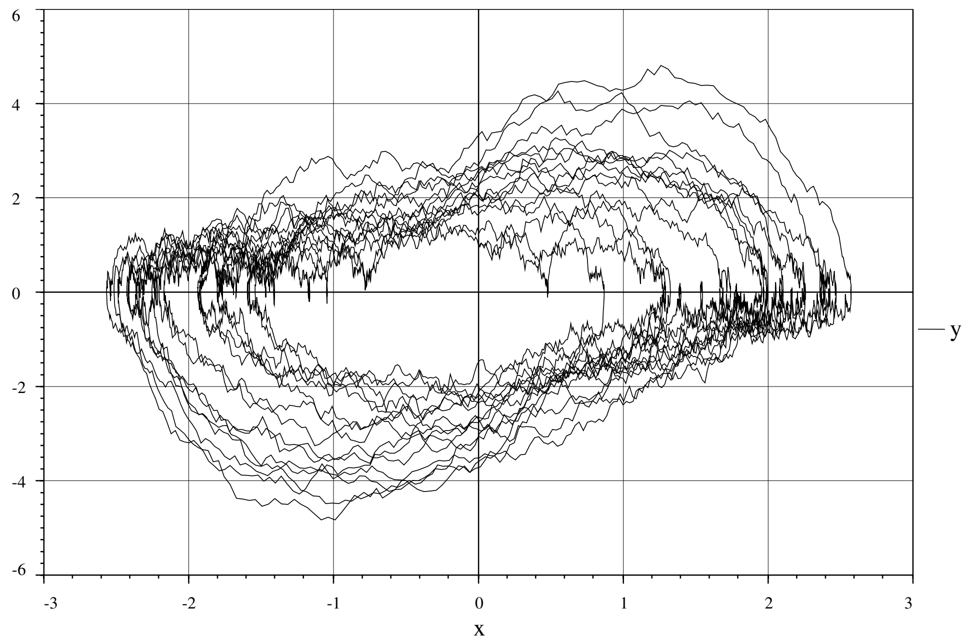

Figure 4. Trajectory of the van der Pol–Duffing oscillator.

```
CONTINUOUS TIME SYSTEM vdpol_duffing;

STATE x = 0 "d2x/dt2 = (alpha + sigma1*dxi1)*x + (beta + sigma2*dxi2)*y +\n\
        gamma*x^3 + delta*x^2*y + sigma3*dxi3";
STATE y = 1             "dx/dt";
PARAMETER alpha = -1;
PARAMETER beta = 1;
PARAMETER gamma = 0;
PARAMETER delta = -1;
WIENER PROCESS xi1 = 1          "Stochastic perturbation of alpha";
WIENER PROCESS xi2 = 997        "Stochastic perturbation of beta";
WIENER PROCESS xi3 = 171713     "Noise acting additively";
PARAMETER sigma1 = 0 "Standard deviation for the perturbation of alpha";
PARAMETER sigma2 = 0 "Standard deviation for the perturbation of beta";
PARAMETER sigma3 = 0 "Standard deviation for the additive perturbation";
TIME t;

AT TIME t:
d(x) = y;
```

```
d(y) = (alpha*x + beta*y + gamma*cube(x) + delta*sqr(x)*y)*d(t)
     + sigma1*x*d(xi1)
     + sigma2*y*d(xi2)
     + sigma3*d(xi3);
```

13.4 Pullback

This section contains a script in the Gnans scripting language. It assumes that the van der Pol–Duffing oscillator of the previous section has been loaded. It studies the trajectories of a stochastic differential equation when the initial condition is approaching minus infinity, while holding the noise trajectory fixed. For this, we first run the system from `t` $= -100$ to -75, and save the RNG state in save number 1. We then `continue` to `t` $= -50$, and save the RNG state in save number 2. We repeat to -25, and save in number 3. To present all trajectories in one plot, we switch the switch `append trajectories` on. We now reset the RNG state to the one corresponding to `t` $= -25$, and run from `t` $= -25$ to 0. The desired "Start" command is here `continue_init` which resets the initial values of the states, but not of the RNG. We repeat from `t` $= -50$, -75, and -100. Finally, by switching `append trajectories` off, the data is written to disk.

Figure 5 shows the result.

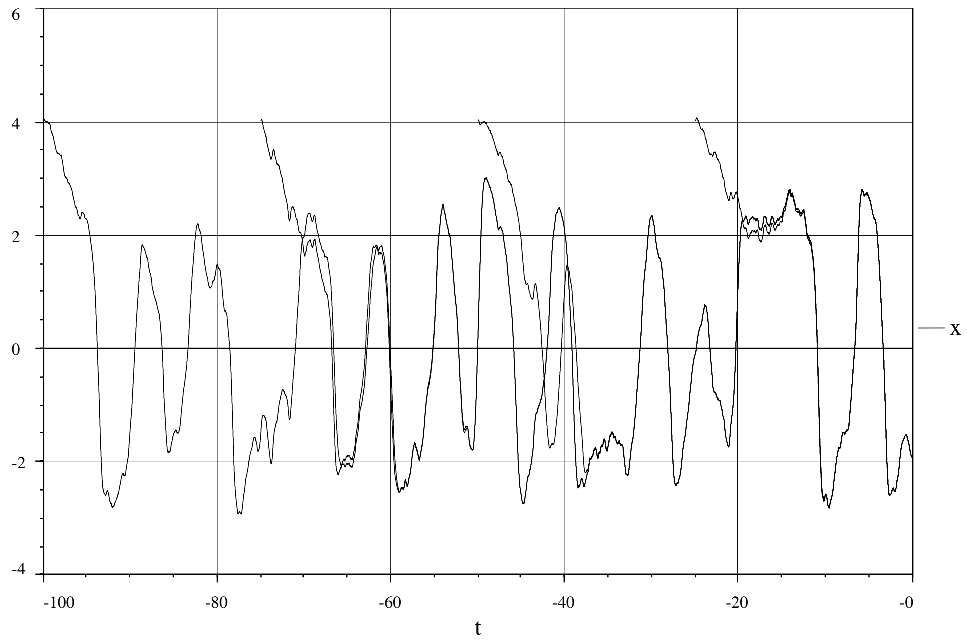

Figure 5. Pullback as described in text.

```
x@ = 4;

sigma3 = 2;
starttime = -100;
stoptime = -75;
run
save_state(d_xi3,1);

stoptime = -50;
continue
```

```
save_state(d_xi3,2);

stoptime = -25;
continue
save_state(d_xi3,3);

stoptime = 0;

switch "append trajectories" on
restore_state(d_xi3,3)
starttime = -25;
continue_init

restore_state(d_xi3,2)
starttime = -50;
continue_init

restore_state(d_xi3,1)
starttime = -75;
continue_init

starttime = -100;
run

switch "append trajectories" off
```

13.5 Feigenbaum Diagram

The logistic equation,

$$x(t+1) \;=\; 4\alpha\; x(t)(1-x(t))$$

is coded in the following simple Gnans system:

```
DISCRETE TIME SYSTEM logistic;
TIME t;
STATE x = 0.99 "x_{k+1} = 4*alpha*x*(1-x)";
PARAMETER alpha = 0.99 "The bifurcation parameter, [0,1]";
VARIABLE alpha_4;

AT TIME t:
alpha_4 = 4*alpha;
q(x) = alpha_4*x*(1-x);
```

(Note the `VARIABLE alpha_4`, cf. Section 9.6 [Performance Tips], page 28.) We would like to study this dynamical system when the parameter α varies between 0.25 and 1. To this end, we certainly would not like to write a script without employing some sort of control structure, as in Section 13.4 [Pullback], page 41. We select the programming language Perl to write a small program which writes the desired commands on standard output. (The reader is encouraged to run the program by itself, piping the output through his/her favorite pager.)

The main part of the program is a loop over α. For every α, we set the initial condition, and run for 100 steps without storing anything (`storeincrement = Infinity` and `store first value` is false), to reach "stationarity". Then we start storing the results by setting `storeincrement` to 0. To create the plot with the somewhat unusual structure, we set `append trajectories` true, and store the parameter α, despite the fact that it is constant during a run.

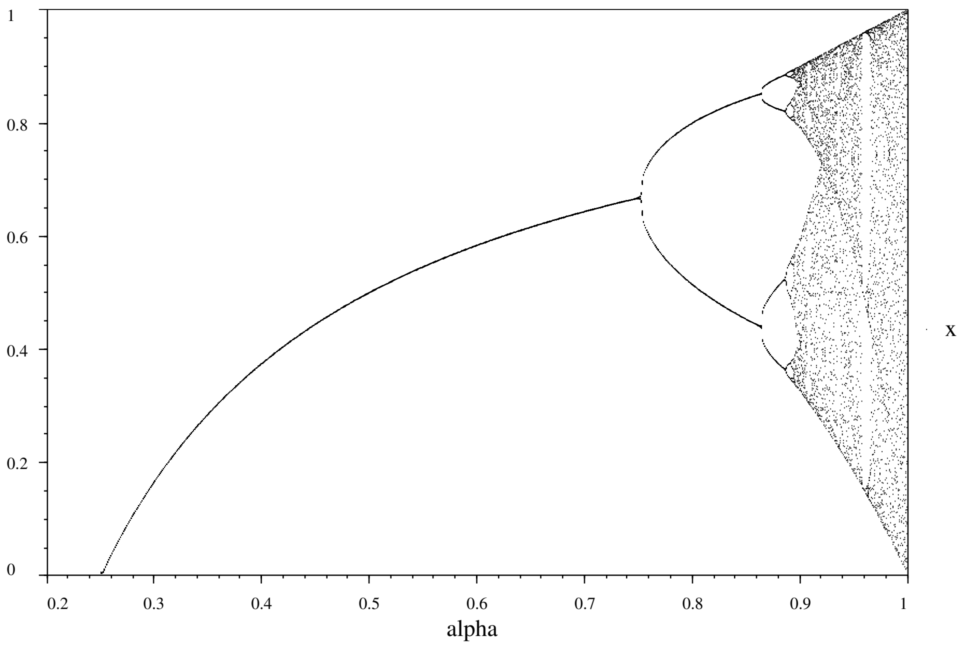

Figure 6. Bifurcation Diagram

Plot using point plot with α on the horizontal axes and `x` on the vertical. See Figure 6.

```
#!/usr/local/bin/perl
$step = 0.001;
$start = 0.25;

print '
switch "store first value" off
store alpha
unstore t
switch "append trajectories" on', "\n";

for ($alpha = $start; $alpha < 1; $alpha += $step) {
    printf "alpha = %g;storeincrement = Infinity;stoptime=100;x@=x;run;
            storeincrement=0;stoptime=150; continue\n", $alpha;
}
print 'switch "append trajectories" off', "\n"
```

A final comment: a specialized program written for the purpose would certainly run faster and with less memory requirements. However, it would probably take much longer to write than the solution using the high-level program solving tool Gnans. Recall, the relevant measure is how fast you have the results in your hand, thus roughly adding programming time and CPU time.

13.6 Histograms

The example in this section illustrates the use of the `hist` class. It is exactly the same stochastic differential equation as in Section 13.3 [van der Pol–Duffing], page 40. Note that some of the arguments in the `open` call are `PARAMETERS` in the sense of Gnans.

Running this system generates a file which has to be plotted with a plotting program. In Figure 7, such a data files has been plotted with MATLAB.

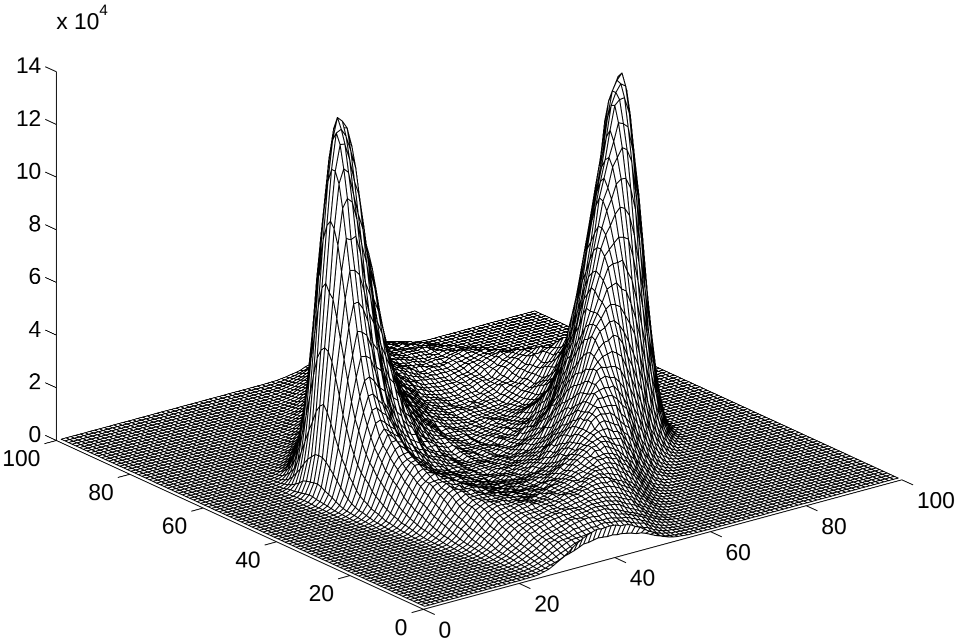

Figure 7. Histogram as described in text.

```
CONTINUOUS TIME SYSTEM vdpol_duffing;
STATE x = 0 "d2x/dt2 = (alpha + sigma1*dxi1)*x + (beta + sigma2*dxi2)*y +\n\
             gamma*x^3 + delta*x^2*y + sigma3*dxi3";
STATE y = 1 "dx/dt";
PARAMETER alpha = -1;
PARAMETER beta = 1;
PARAMETER gamma = 0;
PARAMETER delta = -1;
WIENER PROCESS xi1 = 1          "Stochastic perturbation of alpha";
WIENER PROCESS xi2 = 997        "Stochastic perturbation of beta";
WIENER PROCESS xi3 = 171713     "Noise acting additively";
PARAMETER sigma1 = 0 "Standard deviation for the perturbation of alpha";
PARAMETER sigma2 = 0 "Standard deviation for the perturbation of beta";
PARAMETER sigma3 = 0 "Standard deviation for the additive perturbation";
PARAMETER max_val = 4;
PARAMETER xbins = 50;
PARAMETER ybins = 50;
TIME t;

C-DECLARATION
histo histogram;

AT TIME START:
{
  histogram.open(-max_val, max_val, -max_val, max_val, xbins, ybins,
                 false, "histo", "histo.mat");
}

AT TIME t:
d(x) = y;
```

```
d(y) = (alpha*x + beta*y + gamma*cube(x) + delta*sqr(x)*y)*d(t)
     + sigma1*x*d(xi1)
     + sigma2*y*d(xi2)
     + sigma3*d(xi3);

AT TIME ACCEPTED:
{histogram.update(x, y);}

AT TIME STOP:
{histogram.close();}
```

13.7 Gnans as Plotting Program

Recall that two of the important principles in Gnans are: Gnans is not a plotting program, and its system consists of equations, not C code. The example presented in this section violates both these principles... However, it is believed instead to show the power and flexibility of the design. We are presenting a "discrete time dynamical system" with no dynamics, or equations at all. Instead, it reads from a disk file, and assigns its only `FUNCTION` to the value read.

When end-of-file is encountered, it calls the `terminate` function of the Gnans library, See Section E.1 [Library Functions], page 64.

The code below uses C++ streams. It is not know if it works with other compilers and C++ libraries than `gcc` and `libg++`. Writing the file with the old-fashioned `stdio` functions instead of streams would of course also be possible.

```
// Using Gnans as a plotter of off-line data, in this case one variable,
// contained in the Ascii file DATA.
DISCRETE TIME SYSTEM plot1;
TIME t;
FUNCTION x;

#include <fstream.h>

C-DECLARATION
ifstream *f;
double next_x;                  // Just to read ahead, not to get the last
                                // point twice.

AT TIME START:
{
  f = new ifstream("DATA");     // Constructor opens file
  if (!*f) {
    gnans_error(non_fatal, "Could not open data file");
    return false;
  }
  *f >> next_x;
}
// For the translator to accept the file, x has to have SOME equation
// SOMEWHERE. Therefore, we give x a trivial equation in the AT TIME START
// block.
x = 0;

AT TIME t:
{
  terminate(f->eof());
  x = next_x;
  *f >> next_x;
}
```

```
AT TIME STOP:
{
  delete f;       // The destructor closes the file
}
```

14 Odds And Ends

This chapter contains some stuff that did not fit in logically anywhere else.

14.1 Bugs and known Problems

14.1.1 Dynamic Unloading in Some Versions of SunOS is Broken.

Note for some SunOS version: On some versions of SUN-OS, the dynamic loader has a problem leading to a program crash the *second* time you load a system. SunOS 4.1.1B has this problem, 4.1.2 and 4.1.3 do not.[1] One SPARC-Classic clone I have seen, running a cloned 4.1.3 version, also had this problem. (There is a compile option that circumvents the problem (`BrokenDlclose`) by disabling unloading. The price for this is that if you load a system with the same name as a system that previously has been loaded, changes will not take effects.)

14.2 Memory Management

Gnans differs from other simulation programs in the sense that is was initially designed for computers with lots of virtual memory. Classical simulation programs write the trajectory to disk during the simulation. Instead, Gnans keeps the whole trajectories in memory, and writes them to disk only at the end of a run. (Actually, the trajectories are also written when selecting another store format, or when flipping the switch `append trajectories` to false.)

The natural question is of course: Isn't this a very wasteful way to treat (virtual) memory? The memory requirement grows linearly with the number of points computed. Experience has showed me that it is almost never a problem. It it too large to fit in memory, then it is likely to be too large to write to disk, too. It can also be argued that many programs with which you would like to post-process or visualize you data, read in all data in memory.

If you generate enormous amounts of data, it is also questionable if just writing them to disk is the best thing to do. Don't generate truck-loads of gigantic data files unless you really need it! You should always *think*, and possibly reformulate the problem. The default setting of `storeincrement` is 0, meaning that everything should be stored. Possibly this is more than you need? To only store the final point, put `storeincrement = Infinity`.[2]

Originally, I planned to have the possibility to write the trajectories during the run, but since I noted that I really did not need it, I never implemented it. Nevertheless, it would be a nice thing to be able to generate more data than fits in the memory. A solution should probably be based on the `mmap` system call.

[1] Users of 4.1.1 can probably avoid the problem by replacing '`/usr/lib/ld.so`' by the 4.1.2 version.

[2] '`Infinity`' is parsed on some systems, but not others (dependent on the implementation of the system C subroutine `sscanf`).

Another nice thing would be to be able to give back allocated memory to the operating system, like in GNU Emacs version 19. Also, the GNU `mmap` based `malloc` library would allow to implement a snapshot/undump feature (saving the complete state of the program for later retrieval), which would be a useful addition.

14.3 Signals

Signals is a fundamental concept in the Unix operating system. You can send a process (Gnans in our case) a signal either (from the CLI) by pressing a key such as `^C`, which sends Gnans the interrupt signal (`INT`), or with the Unix command `kill` (the name is historical and obsolete, since signals can do more than just killing).

Gnans does smart things with some signals:

`INT` (This signal corresponds to `^C` from the keyboard.) The current integration and, if applicable, current script, is stopped, just as if the `Stop` button was pressed.

`USR1` The `USR1` signal calls the `display` command.

`TERM` Gnans quits on the `TERM` signal.

`STOP`

`CONT` (`STOP` corresponds to the key `^Z`.) This signal has its default behavior, i.e. the current process is stopped, for possible restart by the `CONT` signal, e.g. by the Unix shell commands `fg` and `bg`.

IEEE floating point exceptions are handled as follows: *inexact*, *division by 0*, and *underflow* are ignored, while *overflow* and *invalid operand* aborts the current run and script, but not the program. The last point on the trajectory is discarded. (This behavior applies to the Sun version only, and is likely to change.)

Fatal signals like `SEGV` are trapped, and if one such is received, Gnans saves whatever it can, re-installs the default signal handler for that signal, sends itself the signal again, leading to its death. (Idea and some code stolen from GNU Emacs.)

Messages coming from signal handlers are displayed on standard output, not in pretty pop-ups, since I am not aware of a safe way to do this (cf. the X intrinsics FAQ).

14.4 Logging

If the switch `logging` is true, all actions by the user is logged to the file '`gnans.log`' in the current directory. This has turned out to be very useful for asking questions like "Exactly how did I do that?". The `display` command also writes its result to the log file.

Normally, commands executed from scripts are not logged, even if `logging` is true. By setting the switch `log script commands` true, also these commands are being logged.

By setting the `debug` level, or the `development` switch, some more possibilities for logging is offered.

14.5 Exit Codes

If exiting due to an error, Gnans returns a non-zero exit code. This is equal to the error severity as defined in the file 'Enums.h'.

14.6 The 'gstore.wiw' file

Some file formats does not contain information on the names of the quantities stored. For example, the Ascii store format stores the data in the form of a number of lines, each column corresponding to a particular stored variable. It is a non-trivial problem to find which variable corresponds to which column. To assist the user in this, the file 'gstore.wiw' (where 'wiw' means "What Is What") is generated, containing information on when the names of the variable, in the same order in which they are stored in the store file.

Even better is to use a store format which contains the name information.

14.7 Gnans without dynamic loading

On some systems, Gnans may have been incompletely ported, i.e. the dynamic loading of systems do not work. In order to concentrate the effort (both programming and documenting) on the fully functional Gnans, this case is treated step-motherly. If you only have access to the crippled Gnans, the command to invoke Gnans has the argument mandatory (see Section 4.1 [Invocation], page 8). The 'gnans' command will then translate and compile the system, and link it to a rather large executable, which will then be executed. The 'System' button will be missing, as well as the `system` command in the script language.

Appendix A Command Line Editing

This appendix is authored by Brian Fox, and is Copyright © 1988 Free Software Foundation, Inc. It is covered by a copyright identical to Gnans. Except for adding these four sentences, and correcting a few misspellings, no changes have been make to it. (Note that the arrow keys (`up`, `down`, `left`, and `right` can be used instead of `C-P`, `C-N`, `C-B`, and `C-F`, respectively.) — BM.

This text describes GNU's command line editing interface.

A.1 Introduction to Line Editing

The following paragraphs describe the notation we use to represent keystrokes.

The text `C-K` is read as 'Control-K' and describes the character produced when the Control key is depressed and the `K` key is struck.

The text `M-K` is read as 'Meta-K' and describes the character produced when the meta key (if you have one) is depressed, and the `K` key is struck. If you do not have a meta key, the identical keystroke can be generated by typing `ESC` *first*, and then typing `K`. Either process is known as *metafying* the `K` key.

The text `M-C-K` is read as 'Meta-Control-k' and describes the character produced by *metafying* `C-K`.

In addition, several keys have their own names. Specifically, `DEL`, `ESC`, `LFD`, `SPC`, `RET`, and `TAB` all stand for themselves when seen in this text, or in an init file (see Section A.3 [Readline Init File], page 52, for more info).

A.2 Readline Interaction

Often during an interactive session you type in a long line of text, only to notice that the first word on the line is misspelled. The Readline library gives you a set of commands for manipulating the text as you type it in, allowing you to just fix your typo, and not forcing you to retype the majority of the line. Using these editing commands, you move the cursor to the place that needs correction, and delete or insert the text of the corrections. Then, when you are satisfied with the line, you simply press `RETURN`. You do not have to be at the end of the line to press `RETURN`; the entire line is accepted regardless of the location of the cursor within the line.

A.2.1 Readline Bare Essentials

In order to enter characters into the line, simply type them. The typed character appears where the cursor was, and then the cursor moves one space to the right. If you mistype a character, you can use `DEL` to back up, and delete the mistyped character.

Sometimes you may miss typing a character that you wanted to type, and not notice your error until you have typed several other characters. In that case, you can type `C-B` to move the cursor to the left, and then correct your mistake. Afterwards, you can move the cursor to the right with `C-F`.

When you add text in the middle of a line, you will notice that characters to the right of the cursor get 'pushed over' to make room for the text that you have inserted. Likewise, when you delete text behind the cursor, characters to the right of the cursor get 'pulled back' to fill in the blank space created by the removal of the text. A list of the basic bare essentials for editing the text of an input line follows.

`C-B` Move back one character.

`C-F` Move forward one character.

`DEL` Delete the character to the left of the cursor.

`C-D` Delete the character underneath the cursor.

Printing characters
Insert itself into the line at the cursor.

`C-_` Undo the last thing that you did. You can undo all the way back to an empty line.

A.2.2 Readline Movement Commands

The above table describes the most basic possible keystrokes that you need in order to do editing of the input line. For your convenience, many other commands have been added in addition to `C-B`, `C-F`, `C-D`, and `DEL`. Here are some commands for moving more rapidly about the line.

`C-A` Move to the start of the line.

`C-E` Move to the end of the line.

`M-F` Move forward a word.

`M-B` Move backward a word.

`C-L` Clear the screen, reprinting the current line at the top.

Notice how `C-F` moves forward a character, while `M-F` moves forward a word. It is a loose convention that control keystrokes operate on characters while meta keystrokes operate on words.

A.2.3 Readline Killing Commands

Killing text means to delete the text from the line, but to save it away for later use, usually by *yanking* it back into the line. If the description for a command says that it 'kills' text, then you can be sure that you can get the text back in a different (or the same) place later.

Here is the list of commands for killing text.

`C-K` Kill the text from the current cursor position to the end of the line.

`M-D` Kill from the cursor to the end of the current word, or if between words, to the end of the next word.

`M-DEL` Kill from the cursor to the start of the previous word, or if between words, to the start of the previous word.

`C-W` Kill from the cursor to the previous whitespace. This is different than `M-DEL` because the word boundaries differ.

And, here is how to *yank* the text back into the line. Yanking is

`C-Y` Yank the most recently killed text back into the buffer at the cursor.

`M-Y` Rotate the kill-ring, and yank the new top. You can only do this if the prior command is `C-Y` or `M-Y`.

When you use a kill command, the text is saved in a *kill-ring*. Any number of consecutive kills save all of the killed text together, so that when you yank it back, you get it in one clean sweep. The kill ring is not line specific; the text that you killed on a previously typed line is available to be yanked back later, when you are typing another line.

A.2.4 Readline Arguments

You can pass numeric arguments to Readline commands. Sometimes the argument acts as a repeat count, other times it is the *sign* of the argument that is significant. If you pass a negative argument to a command which normally acts in a forward direction, that command will act in a backward direction. For example, to kill text back to the start of the line, you might type `M-- C-K`.

The general way to pass numeric arguments to a command is to type meta digits before the command. If the first 'digit' you type is a minus sign (`-`), then the sign of the argument will be negative. Once you have typed one meta digit to get the argument started, you can type the remainder of the digits, and then the command. For example, to give the `C-D` command an argument of 10, you could type `M-1 0 C-D`.

A.3 Readline Init File

Although the Readline library comes with a set of Emacs-like keybindings, it is possible that you would like to use a different set of keybindings. You can customize programs that use Readline by putting commands in an *init* file in your home directory. The name of this file is '`~/.inputrc`'.

When a program which uses the Readline library starts up, the '`~/.inputrc`' file is read, and the keybindings are set.

In addition, the `C-X C-R` command re-reads this init file, thus incorporating any changes that you might have made to it.

A.3.1 Readline Init Syntax

There are only four constructs allowed in the '`~/.inputrc`' file:

Variable Settings
: You can change the state of a few variables in Readline. You do this by using the `set` command within the init file. Here is how you would specify that you wish to use Vi line editing commands:

```
set editing-mode vi
```

Right now, there are only a few variables which can be set; so few in fact, that we just iterate them here:

`editing-mode`
: The `editing-mode` variable controls which editing mode you are using. By default, GNU Readline starts up in Emacs editing mode, where the keystrokes are most similar to Emacs. This variable can either be set to `emacs` or `vi`.

`horizontal-scroll-mode`
: This variable can either be set to `On` or `Off`. Setting it to `On` means that the text of the lines that you edit will scroll horizontally on a single screen line when they are larger than the width of the screen, instead of wrapping onto a new screen line. By default, this variable is set to `Off`.

`mark-modified-lines`
: This variable when set to `On`, says to display an asterisk ('`*`') at the starts of history lines which have been modified. This variable is off by default.

`prefer-visible-bell`
: If this variable is set to `On` it means to use a visible bell if one is available, rather than simply ringing the terminal bell. By default, the value is `Off`.

Key Bindings

The syntax for controlling keybindings in the '`~/.inputrc`' file is simple. First you have to know the *name* of the command that you want to change. The following pages contain tables of the command name, the default keybinding, and a short description of what the command does.

Once you know the name of the command, simply place the name of the key you wish to bind the command to, a colon, and then the name of the command on a line in the '`~/.inputrc`' file. The name of the key can be expressed in different ways, depending on which is most comfortable for you.

keyname: *function-name* or *macro*
: *keyname* is the name of a key spelled out in English. For example:

```
Control-u: universal-argument
Meta-Rubout: backward-kill-word
Control-o: ">&output"
```

In the above example, `C-U` is bound to the function `universal-argument`, and `C-O` is bound to run the macro expressed on the right hand side (that is, to insert the text '`>&output`' into the line).

"*keyseq*": *function-name* or *macro*
: *keyseq* differs from *keyname* above in that strings denoting an entire key sequence can be specified. Simply place the key sequence in double quotes. GNU Emacs style key escapes can be used, as in the following example:

```
"\C-u": universal-argument
"\C-x\C-r": re-read-init-file
"\e[11~": "Function Key 1"
```

In the above example, `C-U` is bound to the function `universal-argument` (just as it was in the first example), `C-X C-R` is bound to the function `re-read-init-file`, and `ESC [ 1 1 ~` is bound to insert the text '`Function Key 1`'.

A.3.1.1 Commands For Moving

`beginning-of-line (C-A)`
: Move to the start of the current line.

`end-of-line (C-E)`
: Move to the end of the line.

`forward-char (C-F)`
: Move forward a character.

`backward-char (C-B)`
: Move back a character.

`forward-word (M-F)`
: Move forward to the end of the next word.

`backward-word (M-B)`
: Move back to the start of this, or the previous, word.

`clear-screen (C-L)`
: Clear the screen leaving the current line at the top of the screen.

A.3.1.2 Commands For Manipulating The History

`accept-line (Newline, Return)`
: Accept the line regardless of where the cursor is. If this line is non-empty, add it to the history list. If this line was a history line, then restore the history line to its original state.

`previous-history (C-P)`
: Move 'up' through the history list.

`next-history (C-N)`
: Move 'down' through the history list.

`beginning-of-history (M-<)`
: Move to the first line in the history.

`end-of-history (M->)`
: Move to the end of the input history, i.e., the line you are entering!

`reverse-search-history (C-R)`
: Search backward starting at the current line and moving 'up' through the history as necessary. This is an incremental search.

`forward-search-history (C-S)`
: Search forward starting at the current line and moving 'down' through the the history as necessary.

A.3.1.3 Commands For Changing Text

`delete-char (C-D)`
: Delete the character under the cursor. If the cursor is at the beginning of the line, and there are no characters in the line, and the last character typed was not `C-D`, then return EOF.

`backward-delete-char (Rubout)`
: Delete the character behind the cursor. A numeric arg says to kill the characters instead of deleting them.

`quoted-insert (C-Q, C-V)`
: Add the next character that you type to the line verbatim. This is how to insert things like `C-Q` for example.

`tab-insert (M-TAB)`
: Insert a tab character.

`self-insert (a, b, A, 1, !, ...)`
: Insert yourself.

`transpose-chars (C-T)`
: Drag the character before point forward over the character at point. Point moves forward as well. If point is at the end of the line, then transpose the two characters before point. Negative args don't work.

`transpose-words (M-T)`
: Drag the word behind the cursor past the word in front of the cursor moving the cursor over that word as well.

`upcase-word (M-U)`
: Uppercase all letters in the current (or following) word. With a negative argument, do the previous word, but do not move point.

`downcase-word (M-L)`
: Lowercase all letters in the current (or following) word. With a negative argument, do the previous word, but do not move point.

`capitalize-word (M-C)`
: Uppercase the first letter in the current (or following) word. With a negative argument, do the previous word, but do not move point.

A.3.1.4 Killing And Yanking

`kill-line (C-K)`
: Kill the text from the current cursor position to the end of the line.

`backward-kill-line ()`
: Kill backward to the beginning of the line. This is normally unbound.

`kill-word (M-D)`
: Kill from the cursor to the end of the current word, or if between words, to the end of the next word.

`backward-kill-word (M-DEL)`
: Kill the word behind the cursor.

`unix-line-discard (C-U)`
: Do what `C-U` used to do in Unix line input. We save the killed text on the kill-ring, though.

`unix-word-rubout (C-W)`
: Do what `C-W` used to do in Unix line input. The killed text is saved on the kill-ring. This is different than backward-kill-word because the word boundaries differ.

`yank (C-Y)`
: Yank the top of the kill ring into the buffer at point.

`yank-pop (M-Y)`
: Rotate the kill-ring, and yank the new top. You can only do this if the prior command is yank or yank-pop.

A.3.1.5 Specifying Numeric Arguments

`digit-argument (M-0, M-1, ... M--)`
: Add this digit to the argument already accumulating, or start a new argument. `M--` starts a negative argument.

`universal-argument ()`
: Do what `C-U` does in emacs. By default, this is not bound.

A.3.1.6 Letting Readline Type For You

`complete (TAB)`
: Attempt to do completion on the text before point. This is implementation defined. Generally, if you are typing a filename argument, you can do filename completion; if you are typing a command, you can do command completion, if you are typing in a symbol to GDB, you can do symbol name completion, if you are typing in a variable to Bash, you can do variable name completion...

`possible-completions (M-?)`
: List the possible completions of the text before point.

A.3.1.7 Some Miscellaneous Commands

`re-read-init-file (C-X C-R)`
: Read in the contents of your '`~/.inputrc`' file, and incorporate any bindings found there.

`abort (C-G)`
: Stop running the current editing command.

`prefix-meta (ESC)`
: Make the next character that you type be metafied. This is for people without a meta key. Typing `ESC F` is equivalent to typing `M-F`.

`undo (C-_)`
: Incremental undo, separately remembered for each line.

`revert-line (M-R)`
: Undo all changes made to this line. This is like typing the 'undo' command enough times to get back to the beginning.

A.3.2 Readline Vi Mode

While the Readline library does not have a full set of Vi editing functions, it does contain enough to allow simple editing of the line.

In order to switch interactively between Emacs and Vi editing modes, use the command `M-C-J` (toggle-editing-mode).

When you enter a line in Vi mode, you are already placed in 'insertion' mode, as if you had typed an 'i'. Pressing `ESC` switches you into 'edit' mode, where you can edit the text of the line with the standard Vi movement keys, move to previous history lines with 'k', and following lines with 'j', and so forth.

Appendix B C++ For Gnans System Authors

No standard programming knowledge is necessary to Gnans system descriptions (at least if you do not include raw code in them). However, in order to enter the equations in a machine understandable form, they have to conform to a certain syntax. In order not to invent a completely new language (saves both learning effort for those already familiar with C++, and, even more importantly, saves me documentation effort!), the syntax has been kept as close to the C++ language as possible. A word about the "++" in "C++": if the reader so prefers, he or she can just think of C instead.

For most normal differential and difference equation, it should be possible to just enter the expressions in a standard mathematical syntax. Just append a semicolon (;) to the end. Standard mathematical functions are available with their usual name (log, cos, atan, etc...).

A useful construction is the ternary ? : operator. A good example of its use may be

```
step = t < t0 ? 0 : 1;
```

This expression states that `step` has the value 0 if `t < t0`, else the value 1.

Appendix C Algorithms

This appendix lists the available integration routines.

C.1 Smart ODE Algorithms

There is a number of “smart” integration algorithms in Gnans. They automatically select step size and other parameters on the basis of the user supplied `abserror` and `relerror`.

`LSODE`

This is the March 30, 1987 version of
Lsode: Livermore Solver for Ordinary Differential Equations.

Lsode is a package based on the Gear and Gearb packages, and on the October 23, 1978 version of the tentative Odepack user interface standard, with minor modifications.

Reference:
Alan C. Hindmarsh, Odepack, A Systematized Collection Of Ode Solvers, in Scientific Computing, R. S. Stepleman et al. (eds.), North-Holland, Amsterdam, 1983, pp. 55-64.

Author and contact: Alan C. Hindmarsh,
Computing And Mathematics Research Div., L-316
Lawrence Livermore National Laboratory
Livermore, CA 94550.

Comment: This appears to be a very good general purpose algorithm. Also with completely free step length, produces points close enough for plotted trajectories to look reasonably smooth. This is the default algorithm. I have put a lot of effort on having its error message (the `xerrwv` routines, written in “portable Fortran” — YECCH!!!) Gnans-ified.

As default, `lsode` is used with `methodflag` 10 (see Section 5.2.2 [Internal Parameters], page 13), corresponding to non-stiff integration. If `methodflag` is set to 22, a stiff integration method is used instead. See the code (in file ‘`dlsode.f`’) for more documentation.

`Ode`

Adams method IVP from Shampine and Gordon book.

Uses a modified divided difference form of the Adams Pece formulas and local extrapolation. It adjusts the order and step size to control the local error per unit step in a generalized sense. Normally each call to step advances the solution one step in the direction of tout. For reasons of efficiency de integrates beyond tout internally, though never beyond t+10*(tout-t), and calls intrp to interpolate the solution at tout.

This code is completely explained and documented in the text:

Computer Solution Of Ordinary Differential Equations: The Initial Value Problem By L. F. Shampine and M. K. Gordon.;

`RKF45`

Watts and Shampine Runge-Kutta Fehlberg 45 IVP solver

Fehlberg fourth-fifth order Runge-Kutta method

Written by H. A. Watts and L. F. Shampine
Sandia Laboratories
Albuquerque, New Mexico

RKF45 is primarily designed to solve non-stiff and mildly stiff differential equations when derivative evaluations are inexpensive. RKF45 should generally not be used when the user is demanding high accuracy.

RKF45 uses the Runge-Kutta-Fehlberg (4,5) method described in the reference
E. Fehlberg, Low-Order Classical Runge-Kutta Formulas With Stepsize Control, NASA TR R-315

The performance of RKF45 is illustrated in the reference

L. F. Shampine, H. A. Watts, S. Davenport, Solving Non-Stiff Ordinary
Differential Equations–The State Of The Art,
Sandia Laboratories Report SAND75-0182,
(To appear in SIAM Review.)

C.2 Dumb ODE algorithms

For special cases, for example in teaching, it may be desirable to have access also to dumb integration algorithms. By "dumb" algorithm we mean an integration algorithm without error control or step size selection.

`Euler` The explicit Euler method with fixed step size.

`Rkfix` Fourth order explicit Runge-Kutta with fixed step-size.

C.3 Integration Routines for Stochastic Differential Equations.

"Smart" algorithms for SDEs presently do not exist, and is a subject for ongoing research. Gnans contains only some simple, dumb algorithms.

`Euler-Maruyama`
This is Euler's explicit method for stochastic differential equations, also called Euler-Maruyama.

`Newton` Explicit, asymptotically efficient, stochastic Runge-Kutta type method. for SDE's with only one noise source. Converges with strong order 1.0 to the Itô solution. Fixed step-size, no error control.

Reference:
Nigel J. Newton: "Asymptotically Efficient Runge–Kutta Methods For A Class of Itô and Stratonovich Equations", SIAM J. Appl. Math. Vol 51, No 2, pp. 542-567, April 1991

Appendix D File Formats & Interfacing with other programs

This section contains a short guide to using some plot programs. VERY short examples are given. For a full documentation, see the respective programs.

D.1 Table Over File Formats

The following formats are supported:

`MATLAB`
: Store in a file which can be directly read by MATLAB (preserving variable names). File extension: '`.mat`'

`Maple`
: Store in a file which can be directly read by Maple (preserving variable names). Requires Maple V Release 2. File extension: '`.maple`'

`Ascii`
: Store in an Ascii text-file, one row per data point. File extension: '`.ascii`'

`Ascii (noninterlaced)`
: Store in an Ascii text-file. The variables are stored in non-interleaved format, i.e. after one another. File extension: '`.ascii`'

`Raw/double`
: Store in a raw (machine dependent) file as doubles. File extension: '`.bin`'

`Raw/double (noninterlaced)`
: Store in a raw (machine dependent) file as doubles. The variables are stored in non-interleaved format, i.e. after one another. File extension: '`.bin`'

`Raw/float`
: Store in a raw (machine dependent) file as floats. File extension: '`.bin`'.

`Raw/float (noninterlaced)`
: Store in a raw (machine dependent) file as floats. The variables are stored in non-interleaved format, i.e. after one another. File extension: '`.bin`'.

`Null`
: Don't store anything at all.

D.2 Gnuplot

Gnuplot is available free of charge from e.g. your favorite ftp achieve. Possibly the main drawback by using Gnuplot as plot engine for Gnans is that it only reads Ascii files. Recall, in general it takes much longer to write Ascii data than to generate it. Ok, with that said, to use Gnuplot, select the "Ascii" store format. Make sure you know what variables reside in what columns of the store file (this information is contained in the '`.wiw`' (What Is What) file generated simultaneously by Gnans. It is in the same directory as the store file and named '`gstore.wiw`'. You can now plot, e.g., the variable in column 2 against the variable in column 4 by the command

```
plot 'gstore.ascii' using 2:4 with lines
```

D.3 Khoros (xprism2 or xprism3)

Khoros is a gigantic package for image processing, visualization, signal processing etc. etc. It is available free of charge from `pprg.eece.unm.edu`. We are here only considering using the plotting programs `xprism2` (which creates two dimensional plots) and `xprism3` (which produces three dimensional plots) as stand alone plotting engines. The following is a very simple (almost naive) use of these rather powerful programs.

Select "Float" as store format. Be sure to store exactly two (three) variables for creating a two (three) dimensional trajectory. (Don't forget that the time is saved per default, which is not what you want if you are asking for a phase plane.) Start `xprism2` (`xprism3`) in the background without any arguments. Click `Plot` in `xprism2` (`xprism3`), fill in the name of the data file '`gstore.bin`', select the Float raw data type, click `Plot file`, and answer the question. (If the plot looks like a number of points, rather far apart and connected by straight lines, then you have probably stored the wrong number of variables (exactly two for `xprism2` and three for `xprism3`).)

D.4 MATLAB

MATLAB is a commercial program, sold by Mathworks. Gnans can create MATLAB files in the native MATLAB format. These can very quickly be read into MATLAB with the load command (`load gstore`), and the trajectories are available to MATLAB as column vectors, with their correct names. This makes it very practical to just write `plot(t,y)`, meaning plot '`y`' as a function of '`t`'. A possible pitfall is that MATLAB is reading the variables from disk with the load command, and plotting them from memory, so it is easy to generate new data, while still plotting the old one, and wonder why it did not change! Therefore, it is a good idea to make it a rule to give the load and the plot command together on the command line like

```
load gstore; plot(t,y)
```

The implementation is not derived from code copyrighted by Mathworks, nor does it depend on MATLAB libraries.

D.5 Maple

By selecting the "Maple" format, Gnans will generate files which can be read directly by Maple as vectors with their correct names. Requires Maple V release 2. However, for large data set, this is *slowwww*.

D.6 Octave, RLaB

RLaB and Octave are programs in the Matlab tradition. Just as Gnans, they are freely available under the GNU General Public License. Both uses Gnuplot for as child process for plotting. Both seems to be unable to read binary data, and instead uses their own store format (different and undocumented).

Drivers for these formats are presently not implemented, but this can be done very easily.

If your favorite program has another format, please feel free to implement a driver for that program, and send it to me for possible inclusion.

Appendix E The Gnans Library

Gnans comes with a library implementing some support functions and classes. It is expected to grow in the future.

E.1 Library Functions

`abs`

`sqr`

`cube` These are functions of a single argument, having the obvious definition. They exists in overloaded versions, both for double and integer arguments. They are inlined (provided that your compiler supports it).

`terminate`

This defines a function '`double terminate(int term, int throw_away_last_point = 0)`' which will, as a side effect, will interrupt the integration if its first argument is not 0. If the optional second argument is non-zero, the last point of the trajectories will be discarded.

`chop` This function, declared '`void chop(double &x, int digits, int truncate = 0)`', rounds its first argument to `digits` significant digits. If optional third argument is non-zero, truncates instead of rounds. Note: this function has been implemented without efficiency concerns (it computes logarithms, for example), since it is normally not a sane thing to voluntarily throw away precision.

E.2 Library Classes

This section assumes that the reader is familiar at least with some of the terminology of object oriented programming.

E.2.1 Delays

The delay class is somewhat unfinished, so I select not to document it.

E.2.2 Table

There is a `table` class under construction. This is for functions of one variable, which consist of finitely many pairs of numbers, with linear interpolation in between.

E.2.3 Histograms

A *histogram* is something that counts occupancy times in different intervals in a one- or two-dimensional subspace of the state space. The class name is `histo`.

Strictly speaking, this strongly violates the Gnans' philosophy of distinguishing between the dynamical system, the experiments to be performed on it, and the subsequent data analysis. It should be considered as a dynamical system that simultaneously is computing statistics on itself, which may be practical. In particular, only one pass over the data is made.

In the `AT TIME START` block, the member function `open` should be called. There are two overloaded versions of the `open` function. The version '`void open(double xmin, double xmax, int xbins, boolean include_faroff, char *matrixname, char *filename)`' initializes a one-dimensional histogram. The interval `[xmin, xmax]` is decomposed into `xbins` number of bins. The purpose of the histogram is to count the occupancy times in these different bins. If `include_faroff` is `true`, values less than `xmin` or larger than `xmax` will be counted in the leftmost and rightmost bin respectively, otherwise they are ignored. The result of the computation is a vector with `xbin` entries. The text string `matrixname` is the name this vector will be stored under, provided that the selected format supports named variables. Finally, the `filename` is the name of the file to which the result is stored. As opposed to the normal Gnans save routines, an ending is not automatically provided.[1] If instead the histogram is opened with the other open-function, '`open(double xmin, double xmax, double ymin, double ymax, int xbins, int ybins, boolean include_faroff, char *matrixname, char *filename)`', the histogram will be two dimensional instead. Here `ymin` and `ymax` correspond to the minimum and maximum in the second dimension. The other arguments are the same.

"Counting" takes place when the `update` function is called. Care should be taken on where to place this call. If placed in the `AT TIME t` section, the results will most likely not carry any significance at all, since integration algorithms are free to call this section anytime they want, and with any arguments they want. (Calls in that section must not contain any side effects.) Instead, the call to the `update` function should probably be put in the `AT TIME ACCEPTED` section (see Section 9.5 [Time Blocks], page 27). The call *histo*`.update(`*var*`)` increments the counter corresponding to the bin *var* happened to belong to by one. The version for two-dimensional histograms is *histo*`.update(`*var1*`,` *var2*`)`.

Finally, in the `AT TIME STOP` block, the member function `close` is called, without arguments. This has the effect that the result is written to disk.

An example is given in Section E.2.3 [HistogramsExample], page 64.

[1] This should probably be changed.

Appendix F Grammar Of The System Description Language

Presently, the following two sections are just ripped out of the Yacc grammars using the `exyacc` program described in the Perl book (page 279ff). A future version should be more cleaned up.

F.1 List of terminal symbols

`DIGITS` a sequence of digits.

`INT` an integer

`FLOAT` a floating point number

`ID` an identifier ("name") according to the same rules as in C

`ID8` like `ID`, but ISO-8859-1 *letters* are also allowed.

`STRING` a sequence of arbitrary characters (except '`"`'), enclosed by double quotes '`"`'.

`CONTINUOUS`
`DISCRETE`

`TIME`

`SYSTEM`

`STATE`
`INTEGERSTATE`
`PARAMETER`
`FUNCTION`

`VARIABLE`

`UNIFORM`

`NORMAL`

`WHITE`

`NOISE`

`WIENER`

`PROCESS`
`C_IDENTIFIER`
`CPPCOMMAND`
`ATTIME`

These correspond to identical (or almost identical) keywords, except for `CPPCOMMAND`, which is a command for the C Preprocessor.

`EQ`

`NE`

`LE`

`GE`

`OR`

`AND`

`LL`

`GG`

`PLUSPLUS`
`MINUSMINUS`
: These correspond to obvious binary and unary operators.

`C_BLOCK` A block of raw C++ code, enclosed in a pair of matching braces.

`INTEGER` The keyword `int`.

`DOUBLE` The keyword `double`.

`COMMENT` A comment in C or C++ syntax.

F.2 Grammar

```
system:         pass_thrus system_decl decls time_block time_blocks
;

decls:
                | decls decl
;

decl:           component_decl
                | time_decl
                | pass_thru
;

time_blocks:
                | time_blocks time_block
;

time_block:     time_header C_block equations
;

time_header:    ATTIME ID ':'
;

C_block:
                | C_BLOCK
;

equations:
                | equations equation
;

equation:       diff_eqn
                | stoch_diff_eqn
                | difference_eqn
                | alg_eqn
                | component_decl
                | pass_thru
```

```
;


diff_eqn:        'd' '(' ID ')' '=' exp ';'
;

difference_eqn: 'q' '(' ID ')' '=' exp ';'
;

alg_eqn:         ID '=' exp ';'
;

stoch_diff_eqn: 'd' '(' ID ')' '=' diffusion diffusions ';'
;

diffusions:
                | diffusions '+' diffusion
                | diffusions minus_diffusion
;

diffusion:      '0'
                | multiplier 'd' '(' ID ')'
;

minus_diffusion:  '-' multiplier 'd' '(' ID ')'
;

multiplier:
                | exp '*'
;
```

```
exp:            FLOAT
                | INT
                | ID
                | int_cast exp %prec UMINUS
                | double_cast exp %prec UMINUS
                | fname '(' ')'
                | fname '(' exp ')'
                | fname '(' exp ',' exp ')'
                | fname '(' exp ',' exp ',' arglist ')'
                | ID bracket_args
                | exp '+' exp
                | exp '-' exp
                | exp '*' exp
                | exp '/' exp
                | exp '%' exp
                | PLUSPLUS ID
                | MINUSMINUS ID
                | ID PLUSPLUS
                | ID MINUSMINUS
                | '-' exp  %prec UMINUS
                | '+' exp  %prec UPLUS
                | '!' exp
                | '~' exp
                | exp '^' exp
                | '(' exp ')'
                | exp LL exp
                | exp GG exp
                | exp EQ exp
                | exp NE exp
                | exp LE exp
                | exp GE exp
                | exp '<' exp
                | exp '>' exp
                | exp '&' exp
                | exp '|' exp
                | exp AND exp
                | exp OR exp
                | exp '?' exp ':' exp
;

fname:          ID
                | fname '.' ID
;

arglist:        exp
                | arglist ',' exp
;

int_cast:       '(' INTEGER ')'
;

double_cast:    '(' DOUBLE ')'
;

bracket_args:   '[' exp ']'
                | bracket_args '[' exp ']'
;

time_decl:      TIME ID ';'
;
```

```
system_decl:
                  continuous_system
                | discrete_system
;

latin1id:       ID
                | ID8
;

continuous_system:
                CONTINUOUS TIME SYSTEM latin1id ';'
;

discrete_system:
                DISCRETE TIME SYSTEM latin1id ';'
;


component_decl:
                  state_decl
                | int_state_decl
                | par_decl
                | func_decl
                | var_decl
                | u_noise_decl
                | n_noise_decl
                | wiener_decl
                | id_decl
;

state_decl:
                scalar_state_decl

;

scalar_state_decl:      STATE ID '=' real string ';'
;


int_state_decl: INTEGERSTATE ID '=' INT string ';'
;

par_decl:         scalar_par_decl
;

scalar_par_decl:        PARAMETER ID '=' real string ';'
;


func_decl:      FUNCTION ID string ';'
;

var_decl:       VARIABLE ID ';'
;

id_decl:        C_IDENTIFIER ID ';'
;

u_noise_decl:   UNIFORM WHITE NOISE ID '=' INT string ';'
;

n_noise_decl:   NORMAL WHITE NOISE ID '=' INT string ';'
;
```

```
wiener_decl:    WIENER PROCESS ID '=' INT string ';'
;

string:
                | STRING
;

real:           FLOAT
                | '-' FLOAT
                | INT
                | '-' INT
;
pass_thrus:
                | pass_thrus pass_thru
;
pass_thru:      ';'
                | COMMENT
;
```

Appendix G Grammar Of The Script Language

G.1 List of Terminal Symbols in the Script Language

The terminal symbols are believed to be obvious, so we omit the list.

G.2 Grammar

```
input:
        | input stmt
;

stmt:
          '\n'
        | ';'
        | exp ';'
        | exp '\n'
        | cmd '\n'
        | cmd ';'
        | error '\n'
;

cmd:      START
        | CONTINUE
        | CONTINUE_INIT
        | HELP
        | QUIT
        | SYSTEM STRING
        | DISPLAY display_unit
        | SET ALGORITHM '=' STRING
        | SET FORMAT '=' STRING
        | SWITCH STRING ON
        | SWITCH STRING OFF
        | STORE VAR
        | UNSTORE VAR
        | RESTORE_STATE '(' VAR ',' exp ')'
;

display_unit:
        | STATES
        | VARIABLES
        | PARAMETERS
        | FUNCTIONS
        | TIMES
        | INTERNALS
        | RESOURCES
        | RANDOMS
;
```

```
exp:      NUM
        | VAR
        | VAR '=' exp
        | VAR PLUSE exp
        | VAR MINUSE exp
        | VAR TIMESE exp
        | VAR SLASHE exp
        | VAR PERCENTE exp
        | FNCT '(' exp ')'
        | exp '+' exp
        | exp '-' exp
        | exp '*' exp
        | exp '/' exp
        | exp '%' exp
        | PLUSPLUS VAR
        | MINUSMINUS VAR
        | VAR PLUSPLUS
        | VAR MINUSMINUS
        | '-' exp  %prec UMINUS
        | '+' exp  %prec UPLUS
        | '!' exp
        | '~' exp
        | exp '^' exp
        | '(' exp ')'
        | exp LL exp
        | exp GG exp
        | exp EQ exp
        | exp NE exp
        | exp LE exp
        | exp GE exp
        | exp '<' exp
        | exp '>' exp
        | exp '&' exp
        | exp '|' exp
        | exp AND exp
        | exp OR exp
        | exp '?' exp ':' exp
        | SAVE_STATE '(' VAR ')'
        | SAVE_STATE '(' VAR ',' exp ')'
;
```

Short Contents

Table of Contents