SimParm:
A simple, flexible and collaborative configuration framework for interactive and batch simulation software

Kevin Pulo
APAC National Facility
ANU Supercomputing Facility
Australian National University
Acton, ACT, 0200, Australia
kevin.pulo@anu.edu.au

Abstract

The configuration of parameters in simulation software is an often overlooked aspect of the development process. SimParm is a C++ framework that alleviates the burden of managing configuration parameters from software developers. It has been designed to be simple, easy to use and flexible, both when defining parameters and using them in the simulation. Plain text configuration files are supported, as well as overriding values on the command line. SimParm allows interactive real-time adjustment of parameters during the simulation — when running locally and remotely. Furthermore, multiple users can adjust parameters, allowing collaborative exploration of the parameter space. This helps users to determine suitable parameter values for unfamiliar datasets — even when the dataset is too large to run be run on the local workstation. This paper describes the design and usage of SimParm, and includes an example application of a simple mass-spring simulation of a triangular mesh.

Keywords— **configuration, framework, simulation, parameters, batch, interactive, remote interactivity, collaboration, HPC, C++, ConfigSet, ConfigEntry, twiddler, multiplexer, named pipe, fifo, geomslab, geomview, OOGL**

1 Introduction

In the area of developing software simulations for use in HPC, the main interest of the software developer is usually the mechanics and algorithms of the simulation itself, as well as the scientific results and insights that can be obtained from running the simulation. As a result, one aspect of simulation code that can easily be overlooked is the configuration of the simulation software. This is particularly true for codes that are smaller, more experimental and those just commencing development. This is a situation that is frequently found when simulation software is developed by a single, small research groups. These developers are focused on the simulation itself, rather than a good way to set the values of parameters in the software.

In the worst case, parameter values are hard-coded into the source code of the simulation software. This means that the code must be recompiled each time a user wants to change a parameter value. This consumes additional time and space, and is more confusing when reviewing results, running several simulations concurrently on the same system, and so on. The benefit of this method, though, is that the software developer need not concern themselves with issues such as the reading and parsing of values from the command line, configuration files and the like.

At the other end of the spectrum, developers may implement sophisticated systems for managing runtime configuration (such as a custom configuration file format), or they might employ other libraries to do this task (such as an XML parsing library like Expat[2]). In the former case, the developer may spend significant amounts of time writing code to support the configuration system, and then maintaining that code when the configuration needs of the simulation change. In the latter case, the developer must learn how to use the external library, which often achieve generality by being overly complicated for the simple task of assigning runtime values to simulation parameters. In both of these cases, the clarity and simplicity of the actual simulation code is often compromised by the configuration code.

One good example of support for configuration that is provided in a toolkit is in the Message Passing Environment Utilities (MPEU)[5], distributed as part of the Model Coupling Toolkit (MCT)[6, 7]. However, this software has a focus on message-passing applications, thus limiting its generality, and is written for Fortran90, and thus is not suitable for C++ applications.

For example, a common occurance is the need to add a new configuration parameter that had not earlier been considered. If the configuration system requires considerable work to add this new parameter, then the developer has more incentive to simply "fudge" the system, for example by hard-coding the parameter "temporarily", or by repurposing existing parameters in non-obvious ways. The configuration system should allow new parameters to be easily defined in a single master location.

Furthermore, choosing suitable and appropriate simulation parameters for a particular input dataset can be difficult. This is particularly true when the dataset is unfamiliar or very large. One common reason for this is that even when configuration systems are adequate, the *modus operandi* of the simulation is to run potentially large and long HPC jobs, which are under the control of a batch queueing system. Initial parameters are chosen with very little guidance, and feedback on the choice of parameters does not occur until after the simulation has run for some time. At this point, the parameters can be evaluated, adjusted, and the process repeated. This whole procedure is time-consuming, and can be frustrating for the researcher who would prefer a quicker and easier way of "honing in" on suitable configuration parameter values.

This paper presents SimParm, a C++ framework which alleviates the burden of managing configuration parameters from simulation software developers. SimParm addresses all of the aforementioned problems, and has been designed to meet the following requirements:

- Simple and easy to use in simulation code. Code that uses SimParm is as easy to read as code that has hard-coded static configuration parameters.
- Configuration parameters are easily and simply defined in a single location in the code.
- New parameter types are easily added to the SimParm code by using object-oriented inheritance.
- Configuration parameters are statically typed but dynamically accessible. This provides the convenience usually associated with dynamically typed systems or interpreted languages, but without the performance penalties and with the benefits of type-safety.
- Configuration file format is simple ASCII, making it human readable and editable. Multiple configuration files can be read in, which allows the segregation of common options into logical groups. For example, parameters specifying input and output data files can be in a separate configuration file to those specifying timesteps, epsilon cutoff values, and so on.
- Supports interactive real-time adjustment of simulation parameters while the simulation is running. This is possible not only when the simulation is running on the local machine, but also when it is running on remote computing infrastructure (such as a batch-queue controlled HPC facility). This helps simulation users to quickly and easily identify suitable parameter values for new or unfamiliar datasets.
- Supports collaborative investigation of simulation parameters, by allowing multiple researchers to connect to a single simulation running on HPC facilities. As above, this helps to facilitate the determination of suitable parameter values.

Section 5 presents the example application of SimParm to a simple mass-spring physical simulation of an unstructured triangular mesh.

2 Code integration

This section describes how SimParm is used inside simulation code. This is addressed in two main aspects: first, how configuration parameters are defined, and second, how these parameters are actually used by the simulation.

2.1 ConfigSets

The fundamental class in SimParm is `ConfigSet`, which is conceptually a collection of `ConfigEntry` objects. Each ConfigEntry object defines a particular simulation parameter. In order to define the set of parameters that comprise a particular simulation, a child class of `ConfigSet` is created, with `ConfigEntry` objects as members. For example, the following class definition for `ConfigSetRelax` has two double-precision floating point parameters named `timestep` and `epsilon`, and an unsigned long parameter named `max_t`.

```
#include "ConfigSet.hh"
class ConfigSetRelax
    : public ConfigSet {
public:
  ConfigEntryDouble timestep;
  ConfigEntryUnsignedLong max_t;
  ConfigEntryDouble epsilon;
  ConfigSetRelax();
};
```

It is also possible to create a hierarchy of ConfigSet classes, with each defining the necessary ConfigEntry objects. C++'s multiple inheritance means that this hierarchy need not take the form of a tree. For example, the following `ConfigSetRunControl` class defines three boolean parameters named `running`, `finished` and `autorestart`, and `ConfigSetApplication` has eight members in total; the three inherited from `ConfigSetRelax`, the three from `ConfigSetRunControl`, and two string parameters of its own.

```
class ConfigSetRunControl
    : public ConfigSet {
public:
  ConfigEntryBool running;
  ConfigEntryBool finished;
  ConfigEntryBool autorestart;
  ConfigSetRunControl();
};

class ConfigSetApplication
    : public ConfigSetRelax,
      public ConfigSetRunControl {
public:
  ConfigEntryString input_filename;
  ConfigEntryString output_filename;
  ConfigSetApplication();
};
```

The definition for the `ConfigSetRelax` constructor might look as follows:

```
ConfigSetRelax::ConfigSetRelax()
: ConfigSet() {
  timestep.setName("timestep");
  timestep.setDesc("Relax timestep");
  timestep = 1.0;
  timestep.setIncrement(0.1);
  timestep.setMin(0.0);
  register_entry(&timestep);
  max_t.setName("max_t");
  max_t.setDesc("Relax max timesteps");
  max_t = 1000000;
  max_t.setIncrement(10000);
  max_t.setMin(0);
  register_entry(&max_t);

  epsilon.setName("epsilon");
  epsilon.setDesc("Relax epsilon");
  epsilon = 0.1;
  epsilon.setIncrement(0.01);
  epsilon.setMin(0.0);
  register_entry(&epsilon);
}
```

This defines the string name and description for each ConfigEntry, as well as a default value. The increment and minimum values are advisory, and are only used interactively, see Section 4. Finally, a pointer to each entry is passed to the `register_entry()` member function, which informs the ConfigSet of the presence of each member.

Whenever a new configuration parameter is required, it only needs to be added to the relevant ConfigSet class definition and constructor.

In order to use the parameters from within the simulation code, an instance of the ConfigSet child class must be created. Typically this is done by declaring a static variable named `config` in the file containing `main()`, for example:

```
static ConfigSetApplication config;
```

The value of a parameter is obtained by using the function operator, that is, `operator()`, for example:

```
double e = config.epsilon();
```

The value of a parameter is set by using the assignment operator with the underlying type, that is, `operator=`, as shown above in the ConfigSet constructors.

The following example code shows how easily ConfigEntries are used. It uses the `ConfigSetRelax` parameters to control a relaxation `for` loop. The `max_t` parameter specifies the maximum number of loop iterations, `timestep` specifies the amount of time δt by which to advance the simulation each time step, and `epsilon` specifies a convergence criteria. If the timestep is determined to be too large, then it is reduced by a factor of 10%.

```
for (unsigned long t = 0;
     t < config.max_t(); t++) {
  computeForces();
  actionForces(config.timestep());
  if (errorsum < config.epsilon())
    break;
  if (errorHasWorsened())
    config.timestep *= 0.9;
}
```

In the case of multiple inheritance, namespace clashes can be resolved by explicitly resolving the members in question, for example, `config.ConfigSetRelax::epsilon()` compared to `config.ConfigSetApplication::epsilon()`.

2.2 ConfigEntrys

As shown above, each configuration parameter is represented by a single ConfigEntry object. The `ConfigEntry` class is abstract, and the functionality for each type of parameter is provided by child classes. The following parameter types are currently supported:

- `ConfigEntryDouble` and `ConfigEntryFloat` represent double-precision and single-precision floating-point values, respectively.
- `ConfigEntryInt`, `ConfigEntryUnsignedInt`, `ConfigEntryLong` and `ConfigEntryUnsignedLong` (and so on for shorts and long longs) represent the appropriate signed or unsigned integer value.
- `ConfigEntryBool` represents a boolean value.
- `ConfigEntryString` represents a string value.
- `ConfigEntryChoice` represents an enumerated type, that is, where the value may only be one of a fixed set of possible choices.

The floating-point, integer and boolean based parameter types are straightforward and self-evident. The string parameter type uses the standard C++ string type, which allows access to the underlying character array via the `c_str()` member function. However, `ConfigEntryChoice` requires more explanation. It requires an `enum` to be defined, usually in a separate namespace, and then each of the values to be passed to the `addChoice()` member function with a description. For example, the following describes possible actions for the simulation to take on convergence:

```
namespace Convergence {
  enum {
    nothing,
    exit_program,
    reduce_timestep
  };
}
```

with the following member added to `ConfigSetRelax`:

```
ConfigEntryChoice convergence;
```

and the following added to the constructor:

```
convergence.setName("convergence");
convergence.setDesc("Convergence action");
convergence.addChoice(Convergence::
    nothing, "nothing", "Nothing");
convergence.addChoice(Convergence::
    exit_program, "exit_program",
    "Exit program");
convergence.addChoice(Convergence::
    reduce_timestep, "reduce_timestep",
    "Reduce timestep");
convergence = Convergence::nothing;
register_entry(&convergence);
```

The convergence `if` statement would then be modified as follows:

```
if (errorsum < config.epsilon())
  switch(config.convergence()) {
    case Convergence::nothing:
      break;
    case Convergence::exit_program:
      exit(0);
    case Convergence::reduce_timestep:
      config.timestep *= 0.9;
      break;
  }
```

This is an example of the simplicity of adding a new configuration parameter to existing code. Even for the relatively sophisticated case of an enumerated type, both the definition of the parameter and its use in the simulation code are simple and straightforward. In particular, the simulation code is no harder to read. It is easy to conceive how another configuration option might be added to control the action of the simulation when the error worsens. This ease of configuration means that developers are actually encouraged to write software that is highly configurable, as opposed to perceiving it as a chore.

There are two additional types that are more useful for interactive usage: `ConfigEntryTrigger` and `ConfigEntryDivider`. These will be discussed in Section 4.

Adding a new configuration parameter type (for example, `ConfigEntryComplex` to represent complex numbers) simply requires creating a child class of `ConfigEntry`. This child class must allocate storage for the value of the parameter, implement functions to get and set the value of the parameter, and implement the necessary virtual functions to support input/output of the parameter. As above, getting and setting the value is typically done using operator overloading.

3 Configuration files

Configuration files are a simple ASCII text representation of a collection of configuration entries. The format is one entry per line, of the form:

```
<name> = <value>
```

where `<name>` is the name of entry that was passed to the `setName()` member function (with no embedded whitespace), and `<value>` is the type-specific ASCII representation of the parameter value. For ease of parsing, the whitespace around the = is significant. Lines beginning with a hash # are ignored as comments, as are lines that are wholly whitespace. For the numeric entry types, the value format is self-evident. For the boolean entry type, the value may be integer (0 for false, anything else for true), or it may be the string “true” or “false”. For the string entry type, the value is all the characters until a newline is reached (no quotes are required). For the choice entry type, the value may be either the integer index in the `enum`, or the corresponding string name passed to the `addChoice()` member function. Generally, the latter is preferred, in order to maximise the readability of the configuration file. An example of a configuration file for the above ConfigSet might be:

```
timestep = 1.5
max_t = 5000
epsilon = 0.001
running = true
finished = false
autorestart = true
input_filename = input.dat
output_filename = output.dat
convergence = exit_program
```

Reading a configuration file is a simple matter of using the overloaded input operator (that is, `operator>>`) on the `ConfigSet` class. For example, a simulation code may do the following:

```
static ConfigSetApplication config;
int main(int argc, char *argv[]) {
  ifstream config_file("config.cfg");
  if (config_file) {
    config_file >> config;
    config_file.close();
  }

  // ...
}
```

Although of course, typically the name of the configuration file will not be hard-coded, rather, it is passed on the command line when invoking the simulation. Furthermore, it is a simple matter to read in as many configuration files as are specified on the command line. This allows the segregation of configuration parameters into logical groupings. For example, a certain collection of configuration files might only assign values to relaxation parameters, while another collection of files assign the values of filenames to use. This allows researchers to easily substitute the values of any arbitrary group of parameters, without affecting the others, and so without needing to create many different configuration files for each combination of parameters. The code to read multiple files is as follows:

```
static ConfigSetApplication config;
int main(int argc, char *argv[]) {
  for (int i = 1; i < argc; i++) {
    ifstream config_file(argv[i]);
    if (config_file) {
      config_file >> config;
      if ( ! config_file.eof())
        cerr << "Error reading "
```

```
             << argv[i] << endl;
      config_file.close();
    }
  }

  ostream &log = cerr;
  ofstream logfile(config.logfile());
  if (logfile)
    log = logfile;
  log << "Config values are:" << endl;
  log << config;

  // ...
}
```

Note also that the values of the configuration parameters are output using the overloaded output operator (that is, `operator<<`) of `ConfigSet`, as a convenient means to record the parameters used for that invocation of the simulation. The output format is the same as the input format described above. In this case, the values are output to a logfile, where the name of the logfile to use is determined by the configuration parameter named `logfile`. If this is logfile name is not specified in any of the configuration files that are input, then the default value specified in `ConfigSetApplication` will be used. If the logfile is unable to be opened, then the standard error stream will be used instead.

It is also possible to read configuration entries directly from the command line arguments. In this case, the intention would be to briefly override some values from the configuration file(s) with temporary replacement values, without being required to create and edit a new configuration file. The `ConfigSet::readConfig(int argc, char *argv[])` member function does this, along with the code above to read in configuration files. Command line arguments are assumed to be configuration files, until an argument matching "`-`" is found, after which the arguments are interpreted as individual configuration entries. The individual entries may be enclosed in shell quotes or not. For example, the following code:

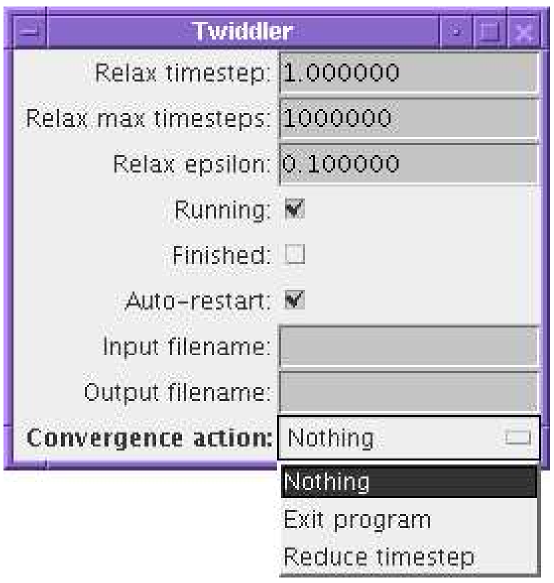
```
static ConfigSetApplication config;
int main(int argc, char *argv[]) {
  config.readConfig(argc, argv);

  // ...
}
```

would allow the program to be invoked as

```
sim simple.cfg relax-conservative.cfg \
        -- "epsilon = 0.01" \
        output_filename = test1.out
sim simple.cfg relax-conservative.cfg \
        -- "epsilon = 0.001" \
        output_filename = test2.out
```

which would take the "simple" parameters, along with the "conservative" relaxation parameters, and then specifically override the relaxation epsilon with test values of 0.01 and 0.001, saving the output into different test files.



Relax timestep: 1.000000
Relax max timesteps: 1000000
Relax epsilon: 0.100000
Running: ✓
Finished: ☐
Auto-restart: ✓
Input filename:
Output filename:
Convergence action: Nothing

Figure 1: An example of a Twiddler interface.

4 Interactive use

SimParm has an interactive module known as the *Twiddler* which can easily interface with any simulation written using ConfigSets. Twiddler is written in Java, and provides an interface for viewing and changing the values of the simulation parameters.

Figure 1 shows an example view of a Twiddler for the earlier ConfigSet examples. Presently, the configuration entries are simply displayed in a vertical list (in the order they are passed to `register_entry()`), although future work will allow for more advanced display options. Numeric parameters may be edited with the keyboard, or may also be incremented or decremented rapidly using the mouse wheel. This allows, for example, the magnitude of forces in the simulation to be increased gradually, or for convergence critereon to be gradually decreased. String parameters may be edited as expected, boolean parameters are represented by checkboxes, and choice parameters by drop-down choice boxes.

Communication between the simulation and Twiddler is performed using standard pipes, for example, with Unix named pipes (FIFOs), or by connecting the standard input and output of the simulation, to the standard output and input of the Twiddler[1]. Following is an example of connecting the simulation `sim` and the twiddler using a combination of standard input and output, and named pipes.

```
$ mkfifo stdin stdout
$ sim config.cfg < stdin > stdout &
$ twiddler < stdout > stdin
```

Alternatively, if the *twinpipe* package[8] is installed, the named pipes can be eliminated and the standard input and output of both processes joined directly:

```
$ twinpipe "twiddler" "sim config.cfg"
```

[1]Some versions of C++ lack the ability to force named pipes to be completely unbuffered, as such, connecting the standard input and output is preferable in these cases

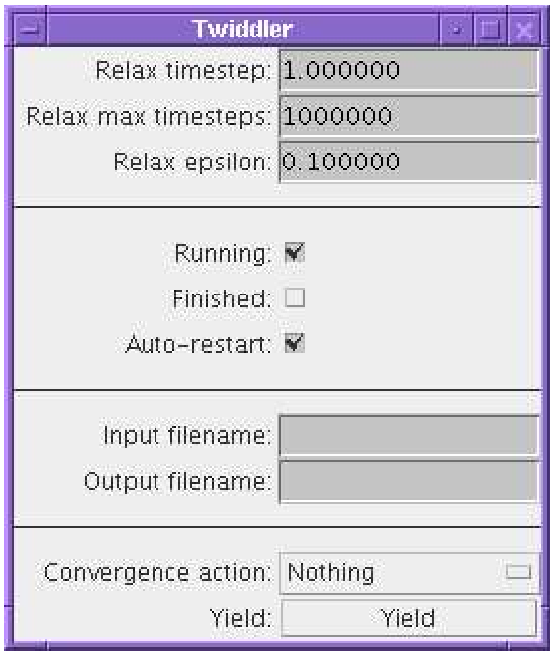


Figure 2: The Twiddler interface of Figure 1, but with the addition of `ConfigEntryDivider`s and a `ConfigEntryTrigger`.

There are two `ConfigEntry` types that are more specifically aimed at interactive use:

- `ConfigEntryTrigger` represents a "trigger" value, that is, a parameter which may be "triggered" to activate a certain function.
- `ConfigEntryDivider` represents a horizontal dividing line, useful for visually separating groups of entries.

Figure 2 shows an example of these two types. Triggers are used in simulation code by testing if they have been triggered with the `triggered()` member function or by casting to a `bool` (for example, with the statement `if (config.yield)`). If so, after the necessary action has been performed the Trigger object must be "untriggered" by calling the `untrigger()` member function. The simulation code can activate the trigger simply by calling the `trigger()` member function.

To use this feature, the `ConfigSet::setIO(istream*, ostream*)` member function is called. For example, to use standard input and standard output, `config.setIO(&cin, &cout);`. This will output the definition of the ConfigSet in question to the provided output, allowing the twiddler to determine what configuration parameters are present, their names, descriptions, initial values and so on. The format is again simple line-based ASCII text. The first line indicates the number of entries present. Each subsequent entry then takes the form:

```
type <type>
name <name>
desc <description>
viewable <true/false>
editable <true/false>
outputOnChange <true/false>
<type-specific-options>
end
```

Where *<type>* is a string representing the type of the entry (for example, `double`, `unsigned_long`, `bool` or `choice`), *<name>* and *<description>* are the string name and description of the entry (set in the ConfigSet constructor), and `viewable`, `editable` and `outputOnChange` are boolean options for each entry. The `viewable` option indicates whether or not the entry should be displayed by the twiddler, thereby allowing the developer to hide certain configuration parameters from the display if they so desire. The `editable` option indicates whether or not the entry may be changed by the twiddler user. This allows the developer to mark certain configuration parameters as "read-only", for example, the name of an input data file that has already been read in might be of interest to the twiddler user (so they know which dataset they are working with), but interactively changing the value of the filename later in the simulation does not make sense. The `outputOnChange` option indicates if the value of the parameter will be automatically updated in the twiddler display when it changes in the simulation. The *<type-specific-options>* are a set of options that are specific to each ConfigEntry type. They follow the same basic format as the preceeding options. For example, the numeric entry types each have the following format:

```
value <value>
increment <increment>
hasmin <true/false>
min <minimum value>
hasmax <true/false>
max <maximum value>
```

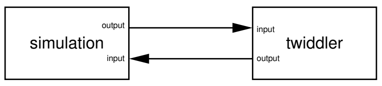
Where the `min` and `max` lines are only present if `hasmin` and `hasmax` are `true`, respectively. The *<value>* is the initial value of the entry, and *<increment>* is the amount by which the value should increase or decrease when the mouse wheel is scrolled up or down. If *<hasmin>* is true, then *<min>* is the lowest possible value that the entry may be set to, and similarly for *<hasmax>* and *<max>*.

Following this definition, whenever the value of a configuration parameter changes, and that parameter has the `outputOnChange` option set (which is the default), a line will be output of the form

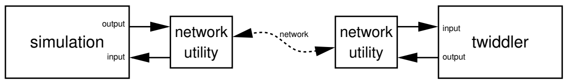
```
set <name> = <value>
```

that is, the format is the same as the configuration file format, except with the leading word `set`. The twiddler reads these lines as they are output, and updates the value of the corresponding entry in the display.

Similarly, when the user changes the value of an entry in the twiddler, a line of the same format will be output



(a) Local



(b) Remote

Figure 3: Connecting the simulation and twiddler locally, and connecting them remotely across a network.

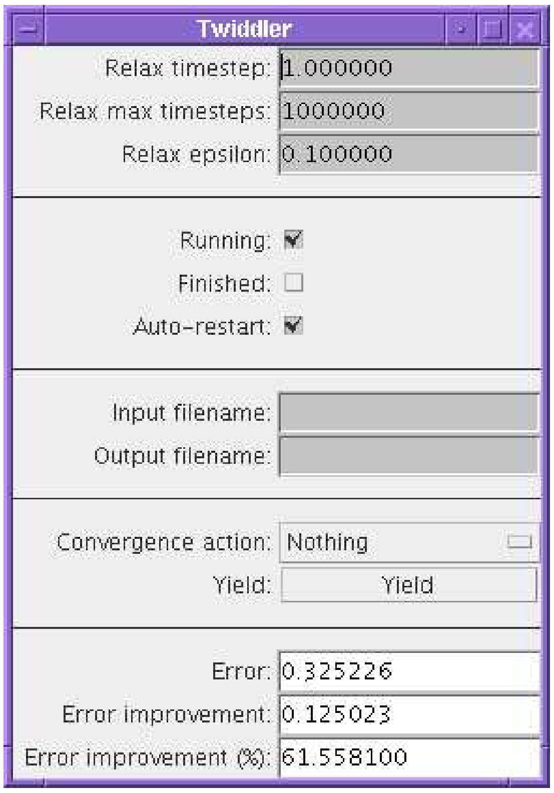
by the twiddler. However, the simulation must read in these lines and update the values of the ConfigEntry objects accordingly. This is achieved by periodically calling the `ConfigSet::serviceInput()` member function. This is typically performed within the main loop of the simulation, so that interactive parameter changes are detected and their values updated in a timely fashion. Of course, there is a tradeoff between the delay in updating parameters, and the amount of CPU time spent checking for updates (which would usually be considered wasted overhead). Checking for updates every 0.1 to 1 seconds is generally considered to be adequate for most purposes.

The simple nature of the communication between the simulation and twiddler allow for two additional and powerful features to be implemented in an easy and straightforward manner. These are remote interactivity and collaborative use, which are discussed in the following subsections.

4.1 Remote interactivity

The bi-directional data channel between the simulation and twiddler is usually achieved by named pipes, but this does not necessarily have to be the case. It would also be possible to use C++ and Java networking libraries to open a TCP/IP connection between the simulation and twiddler, and then use these sockets for the interactive communication. However, the abundance of networking utilities available mean that it is usually easier to leave the simulation and twiddler using named pipes or standard input/output, and then use such a networking utility to connect these pipes or streams. Figure 3 shows this concept.

One common utility that is frequently used to connect pipes to networks is *netcat*[3], however, this still requires choosing an appropriate TCP/IP port, ensuring that the port is not blocked by firewalls, and so on. The easiest solution is usually to use the *ssh* program to connect the simulation and twiddler. This very simply allows the simulation to run on a higher performance machine, while the twiddler runs on the user's desktop workstation. For example, if the simulation named `sim` used standard input and output for interactive communication, then it could be run on the machine named `remotehost`, under local interactive control as follows:


Twiddler

Relax timestep:	1.000000
Relax max timesteps:	1000000
Relax epsilon:	0.100000
Running:	✓
Finished:	☐
Auto-restart:	✓
Input filename:	
Output filename:	
Convergence action:	Nothing
Yield:	Yield
Error:	0.325226
Error improvement:	0.125023
Error improvement (%):	61.558100

Figure 4: The addition of read-only monitoring "parameters".

```
$ twinpipe "twiddler" \
   "ssh user@remotehost sim config.cfg"
```

The only difference between this and the earlier example of running the simulation with the twiddler is the inclusion of the ssh command to transport the standard input and output of the simulation from the remote machine to the local one. This makes running the simulation interactively very easy, even when the simulation is running on a remote high performance machine. It is also possible to use ssh's port forwarding features, combined with netcat, to tunnel the interactive communication across the network.

Interactivity is very useful for helping to determine appropriate initial values to use for the simulation configuration parameters. However, for large datasets the simulation is unlikely to run adequately on the local desktop workstation (for example, it may be too slow, or may require large amounts of memory or disk). Thus, remote interactivity is very important for allowing users to interactively explore the parameter space of datasets that are too large to run locally.

Interactive exploration of the parameter space requires feedback on the state of the simulation. One way to do this would be to have this information written to standard error, so that the user can view it in the terminal window alongside the twiddler. This is not convenient if there are a large number of values to be monitored. A better system is to create read-only ConfigEntry objects for each of the values to monitor, and then set these configuration "parameters" whenever the corresponding value changes in the simulation. This will cause the display in the twiddler to update the values as the simulation progresses, allowing the user to simply monitor the twiddler window to adjust parameters and evaluate the simulation. Figure 4 shows an example of a twiddler window with additional monitoring values at the bottom. Section 5 describes an example where this concept is extended, allowing the actual output of the simulation to be monitored in real time, along with the values in the twiddler.

4.2 Collaborative use

Remote interactivity is useful, but the requirement that the twiddler be started at the same time as the simulation limits its flexibility. In particular, it does not work well with batch queueing systems. This is because the time at which the batch job will be started is unknown, and also on such systems it is usually not possible to ssh directly into batch-controlled compute nodes. Although some batch queueing systems, such as the ANUPBS system used at the APAC National Facility, allow interactive access to compute nodes through *interactive batch jobs*[1], such systems are not commonly available. Furthermore, if the twiddler is closed before the simulation ends, then the simulation will abort with a broken pipe error.

A better approach is to allow the twiddler to connect to a simulation that has already been started. This connection could occur at any time during the simulation and the twiddler could be closed and the simulation continue. Furthermore, collaboration can be facilitated by allowing multiple twiddlers to connect to a single running simulation. This allows multiple users at potentially disparate geographic locations to work together to find good parameters for the simulation. When a value is changed in one twiddler, that parameter is then updated in all other connected twiddlers (as well as the simulation). Similarly, the values of "output parameters" in the simulation are updated in all of the twiddlers, so that each user can see the state of the simulation. This form of collaboration is similar to collaborative visualisation and collaborative problem solving environments[4]. It works best when the collaborating users are also able to communicate via standard means, such as telephone, teleconferencing or videoconferencing.

The pipe-based design of SimParm makes this easy. Instead of connecting the interactive I/O of the simulation directly to the twiddler, it is instead connected to a *multiplexer*. The multiplexer monitors the current directory for the creation of named pipes matching `.control-*.in` or `.control-*.out`. When one is found, it is opened, added to the collection of input or output pipes, and then deleted from the filesystem. Lines output by the simulation are then output to each output pipe, and lines received from an input pipe are output to the simulation and all output pipes. Thus, the multiplexer is effectively broadcasting messages between the simulation and the connected twiddlers. This situation is illustrated in Figure 5.

The only additional consideration is when a new output pipe is opened, the multiplexer requests the current ConfigSet definition from the simulation by sending the line "`request definition`". The simulation then outputs the ConfigSet definition (as described in Section 4), and the multiplexer outputs the definition to the new output pipe only.

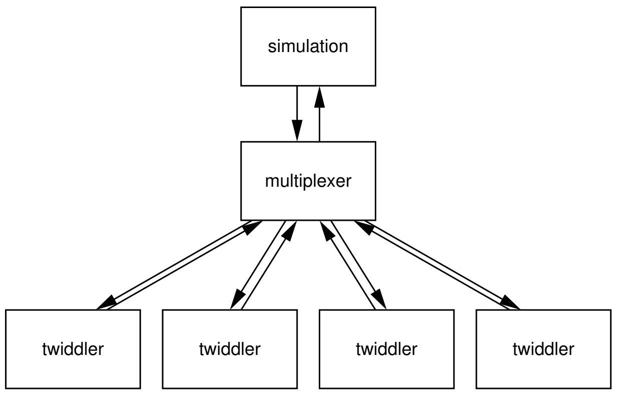


Figure 5: Using the multiplexer to connect the simulation to multiple twiddlers.

The multiplexer is currently a separate program written in python, due to limitations in the disabling of buffering in some C++ stream implementations. Thus, it is currently invoked as:

```
$ twinpipe "multiplex" "sim config.cfg" &
```

Then `twiddler` is invoked with the `--pipe` command line option, to indicate that it should create the necessary named pipes and use them for interactive I/O.

```
$ twiddler --pipe
```

To attach a local twiddler to a remote simulation is a simple variation on the invocations already seen. For example:

```
$ twinpipe "twiddler" "ssh
     user@remote simconnect <target>"
```

where `<target>` is the directory containing the running simulation, and `simconnect` is a simple shell script as follows:

```
#!/bin/sh
cd "$1"
base=.control-$$
mkfifo $base.out $base.in
cat < $base.out > $base.in
```

This method still requires the ability to ssh or rsh into the location where the simulation is running. Staff at the APAC National Facility are developing a `qsh` program, which is a replacement for rsh that will allow users to obtain a shell inside their batch job (running under the ANUPBS batch queueing system). If access to a shell where the simulation is running is truly impossible, then an alternative version of the multiplexer may be written which uses TCP/IP network sockets instead of named pipes. The twiddler could still be used as-is in this case, using a program such as `netcat`, for example:

```
$ twinpipe "twiddler" "nc remote port"
```

A future version of the multiplexer may be written in C, and integrated directly into the ConfigSet object in the simulation. This would be written using the standard Unix convention of `fork()`, `pipe()` and `dup2()` to create a separate process that has its standard input and output attached to the standard output and input of the simulation. This would be integrate the twinpipe functionality, thereby making it unnecessary here.

5 Sample simulation: geomslab

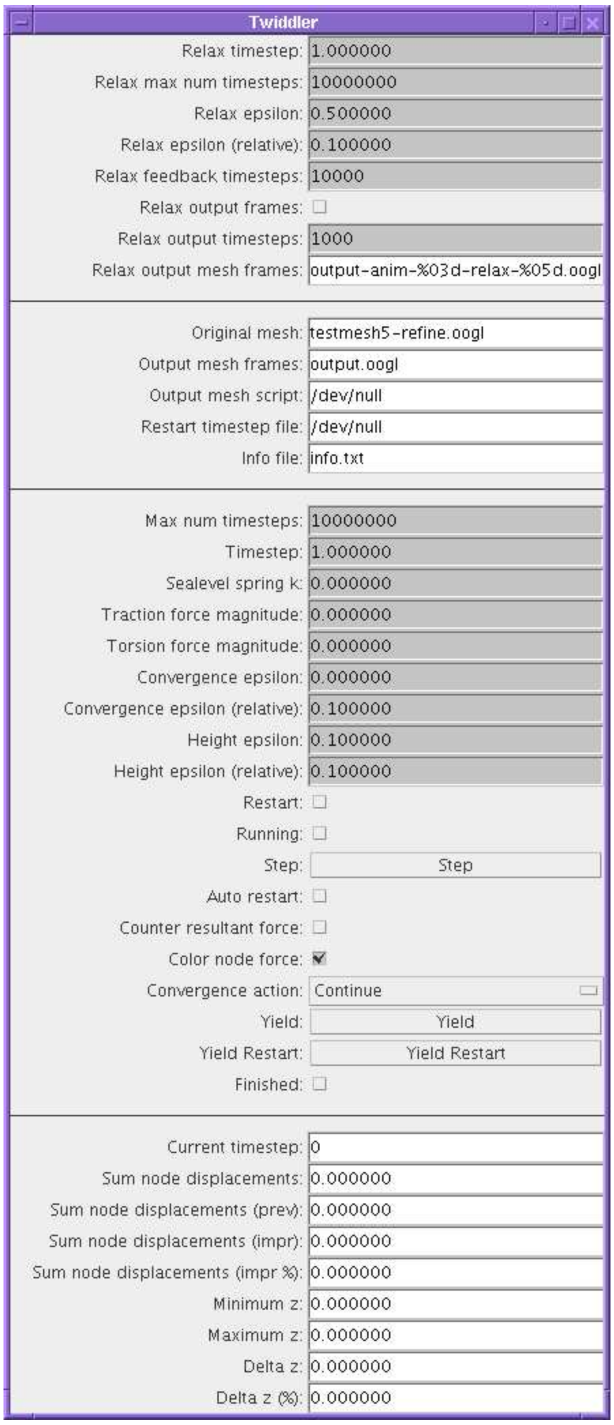
SimParm has been applied to *geomslab*, a simple mass-spring physical simulation of an unstructured triangular mesh. Figure 6 shows the twiddler display for geomslab.

This application uses triangular mesh surfaces in 3-space, as such, the *geomview* package is used to display these 3D meshes. The *animate* module for geomview allows the animation of the mesh over time, which helps to show how the mesh changes throughout the simulation. This is superior to a generated animation in a raster format, such as Quicktime or MPEG, because it allows the user to rotate, zoom, translate and otherwise transform the view of the model as the animation is progressing. However, this strategy requires the mesh frames to be output to individual files, and then animated at the conclusion of the simulation.

The extensible nature of geomview module system means that it is possible to write a simple module which would take a stream of geometry files (in geomview's native *OOGL* format), and display them directly in geomview. In fact, the module is a simple bash shell script, which reads OOGL files from a named pipe, and then outputs them on standard output with the required geomview commands to update the display.

```
#!/bin/bash
echo -n "(geometry example "
echo "{ : streamgeom })"
tmpfile="/tmp/stream-$$-$RANDOM"
trap "rm -f $tmpfile" 0
while :; do
  cat "$1" > "$tmpfile"
  echo -n "(read geometry "
  echo "{ define streamgeom "
  cat "$tmpfile"
  echo "})"
done
```

This allows the simulation to output to the named pipe in exactly the same way as it would ordinarily output to individual files, but have the output appear directly in the geomview window. This stream of OOGL output can also easily be forwarded over the network using ssh, in the same way as the above examples for the twiddler. When this is combined with SimParm and its twiddler, it allows the simulation to run on the HPC compute infrastructure, whilst the user can see the exact state of the simulation and control all of the parameters. This is a very powerful



Relax timestep:	1.000000
Relax max num timesteps:	10000000
Relax epsilon:	0.500000
Relax epsilon (relative):	0.100000
Relax feedback timesteps:	10000
Relax output frames:	☐
Relax output timesteps:	1000
Relax output mesh frames:	output-anim-%03d-relax-%05d.oogl
Original mesh:	testmesh5-refine.oogl
Output mesh frames:	output.oogl
Output mesh script:	/dev/null
Restart timestep file:	/dev/null
Info file:	info.txt
Max num timesteps:	10000000
Timestep:	1.000000
Sealevel spring k:	0.000000
Traction force magnitude:	0.000000
Torsion force magnitude:	0.000000
Convergence epsilon:	0.000000
Convergence epsilon (relative):	0.100000
Height epsilon:	0.100000
Height epsilon (relative):	0.100000
Restart:	☐
Running:	☐
Step:	Step
Auto restart:	☐
Counter resultant force:	☐
Color node force:	✓
Convergence action:	Continue
Yield:	Yield
Yield Restart:	Yield Restart
Finished:	☐
Current timestep:	0
Sum node displacements:	0.000000
Sum node displacements (prev):	0.000000
Sum node displacements (impr):	0.000000
Sum node displacements (impr %):	0.000000
Minimum z:	0.000000
Maximum z:	0.000000
Delta z:	0.000000
Delta z (%):	0.000000

Figure 6: The twiddler display for geomslab.

and flexible system, which allows the one simulation code to support a wide range of batch and interactive use cases.

An example of the flexibility of SimParm is the *gnuplot*-compatible output that is created by geomslab. It is very easy to create a `vector` collection of pointers to ConfigEntry objects to be output. The first line output is a comment (that is, it starts with #) containing the tab-separated names of the ConfigEntries in the vector. Then, each timestep the ConfigEntries are output tab-separated on a single line. The first parameter output is the current timestep, and the subsequent parameters are a mix of output parameters and input parameters. This very easily allows the user to generate plots in gnuplot showing the values of the parameters over the course of the simulation, including when and how input parameters were varied. Future work could extend this to also transmit this plot information over pipes, which would allow the plot to be viewed in gnuplot throughout the course of the simulation in the same way that the mesh can be viewed in geomview during the simulation. This is a simple example of how SimParm can easily be extended to support a wide range of uses.

An animation file in Shockwave Flash format, showing the system in action, can be found at `http://www.kev.pulo.com.au/geomslab/movie/`. This animation shows a typical interactive session with a small test mesh. The right side of the display shows the geomview window containing the coloured mesh, and the left side shows a partial view of the twiddler, including the more important parameters and the output parameters. The actual application is more responsive than the animation, due to the nature of the recording (and that the simulation and twiddler were running on the same single-cpu machine).

6 Conclusion

This paper has presented SimParm, a C++ configuration framework aimed at supporting simulation software in both batch and interactive HPC environments. SimParm provides parameters that are easy to work with at all levels, including the C++ simulation code, plain text configuration files and interactively on local and remote machines. Configuration parameters are quick and easy to add to the software, which promotes software that is highly configurable and helps to avoid developers considering configuration issues as a chore.

SimParm is flexible in the ways it can be used, not merely limited to those shown here. The same code that uses SimParm can support both unattended batch jobs, through configuration files, and can also support interactive exploration of the simulation parameter space. In this interactive mode, the user is able to directly control the simulation parameters, even if the simulation is running on a different machine. Furthermore, multiple users in different locations can all connect to a single running simulation and control its parameters, allowing a collaborative exploration of the parameters. This helps users to determine suitable parameter values for use in configuration files for batch jobs, particularly for datasets that are new or unfamiliar.

Much of this flexibility is due to communication based around pipes. An example application of SimParm has shown that this pipe-based method can even be extended to showing the actual 3D output of the simulation in real-time as the simulation progresses.

References

[1] APAC National Facility. APAC National Facility Local Userguide. `http://nf.apac.edu.au/facilities/userguide/localguide.php#pbsinteractive`.

[2] Fred L. Drake Jr. and James Clark and et al. Expat XML parsing software package. `http://expat.sourceforge.net/`.

[3] "Hobbit" <hobbit@avian.org>. Netcat 1.10 software package. `http://www.vulnwatch.org/netcat/`.

[4] Ian J. Grimstead and David W. Walker and Nick J. Avis. Collaborative Visualization: A Review and Taxonomy. In *Proceedings of the 2005 Ninth International Symposium on Distributed Simulation and Real-Time Applications*, pages 61–69. IEEE, 2005.

[5] J. Walter Larson. The Message-Passing Environment Utilities Tutorial. `http://nf.apac.edu.au/training/MPEU_COURSE/mpeu-tutorial.pdf`.

[6] Jay Larson and Robert Jacob and Everest Ong. The Model Coupling Toolkit: A New Fortran90 Toolkit for Building Multiphysics Parallel Coupled Models. *International Journal of High Performance Computing Applications*, 19(3):277–292, 2005. `http://www-unix.mcs.anl.gov/mct/`.

[7] Robert Jacob and Jay Larson and Everest Ong. MxN Communication and Parallel Interpolation in CCSM3 Using the Model Coupling Toolkit. *International Journal of High Performance Computing Applications*, 19(3):293–307, 2005.

[8] Joel Yliluoma. Twinpipe software package. `http://bisqwit.iki.fi/source/twinpipe.html`.