

# Instructions for use of S-PRES example code

Joshua T. Berryman

June 8, 2011

## 1 Introduction

S-PRES is a method for accelerating simulations such that rare events are observed. It is assumed in this document that you understand the general principles of this process; if you do not then a scientific paper is bundled with this release which explains in detail [1].

## 2 Directory Structure

This release contains a directory “spres” and a directory “shi”. The spres directory contains code for accelerating simulations, and the shi directory contains simulation programs which are accelerated. “shi” is short for “Sheared Ising”, the first of the simulation programs implemented.

Both spres and shi contain “test” subdirectories. The spres/test directory contains test scripts for accelerated dynamics; the shi/test directory contains scripts for non-accelerated dynamics. The spres/test scripts can be used as examples if you want to try to accelerate dynamics of your own.

## 3 Installation

Change into the shi/src directory and type:

```
>make
```

This builds and links the “spres” library code and the “shi” objects to create an accelerated executable, “shi.exe”.

### 3.1 Testing

Change to the spres/test directory and run the script:

```
>cd ../../spres/test  
>./testAsep.bash
```

## 3.2 Parallelism

Parallelism using openMP is disabled by default; to enable this you must edit the makefiles “spres/src/Makefile” and “shi/src/Makefile”, replacing “\_PARALLEL\_FLAGS” with “PARALLEL\_FLAGS” in each one.

If you want parallelism but the parallel build doesn't work initially, then make sure that openMP is installed. This is provided by *libgomp* on my system. Note that openMP is a completely different animal to openMPI - openMP is much simpler for the user.

To select the number of threads you wish to use, set the environment variable OMP\_NUM\_THREADS and run the program as normal. The default value on your system probably gives serial behaviour.

```
>export OMP_NUM_THREADS=8
```

Thanks to Sven Dorosz for doing the parallelisation.

## 4 Implementing your own Dynamics

If you want to study something other than the Ising model using S-PRES, then you are going to have to write the simulation code yourself, or at least write a wrapper for some existing code. The spres library is written in C++; this can be linked easily with C or python programs and also with a fortran program if you have some understanding of fortran (i.e. if you are not a typical fortran programmer).

### 4.1 Accelerating a C++ simulation program

To accelerate a C++ simulation program using spres then you need to make sure that the class which runs your dynamics implements the abstract “simulation” class which is defined in spres/src/spres.h.

The file shi/src/shi.h gives an example of a class which implements simulation. To do this, firstly you need to bring the definition of the virtual class “simulation” into scope:

```
#include "../../spres/src/simulation.h"
```

Next, you need to state that the primary class of your system implements simulation:

```
class shi : public simulation {
```

The characters “: public simulation” instruct the compiler that the class shi (or whatever you want to call your class) implements simulation. Once this is stated, then if it does not implement simulation you will get error messages at compile time.

You are now ready for some programming, because you have to make sure that your simulation program implements all of the functions defined in spres/src/simulation.h. Use the shi class as an example of how to do this.

Here are instructions for each function:

- **void stepTime()** This function should run  $\tau$  timesteps of dynamics. The value of  $\tau$  is up to the simulation class; in the current implementation of spres it can be any constant integer greater than zero.

- **void reset()** If your system has some kind of “initial conditions” then this function should place that system in those initial conditions. An example is the Ising model with all spins random as used for the Kawasaki calculation in the paper [1].
- **void calcMyStats()** Calling this function should ensure that the reaction coordinate of your system and any other observables which you intend to track are up to date.
- **double get\_reactionCoordinate()** Calling this function directly after calcMyStats() should return the reaction coordinate.
- **double \*get\_derivedData(int \*count)** Calling this function directly after calcMyStats() should return the observable for your system. The formatting is as follows: if your observable is an integer with the range [0..N-1] then getDerivedData() should return an array of doubles of size N, with all values zero except for a 1.0 in the bin indexed by the value of the observable. This format is convenient for then building up a histogram. Do not worry about this until you have everything else working.
- **get\_XXXX()** This group of functions does nothing except transfer data that has been read in by your simulation program as arguments (or in a config file) to the spres code.
- **long int \*get\_iran()** This function should return a pointer to some variable controlling the state of the random number generator, which is assumed to be a long int. This should enable the future dynamics to be changed by modifying the current RNG state. Look at the class “RNG2” which is defined in shi/src/ran2.h.
- **simulation \*copySelf()** This function should return a pointer to an initialised object of the same data type as the class which calls it. Look at the function “copyShi()” in shi/src/setup.C for an example.
- **void reLoad( istream \*inStream )** This function should load a configuration from inStream; containing sufficient information to then continue a simulation (e.g. positions and velocities of particles). Look at the implementation in shi/src/inOut.C for an example.
- **unsigned int save( ostream \*outStream )** This function should write a configuration to outStream; containing sufficient information to then continue a simulation (e.g. positions and velocities of particles). It should return the number of bytes written.

## 4.2 Calling S-PRES

To run an accelerated simulation, you have to jump out of the normal dynamics program. To do this, add the following line to the top of the file which contains the main loop of your simulation program (shi/src/shi.C in the example code):

```
#include "../../spres/src/spres.h"
```

Then, a little later in the main loop, once you have read in all of your inputs and initialised your simulation, you should make a function call of the spres run method, like this:

```
{
    spres mySpres;
    mySpres.spresRun( this, pathsPerBin, numBins );
    exit( EXIT_SUCCESS );
}
```

The **this** keyword is used to pass a reference to the calling simulation class, so that spres can call-back and request iterations of the dynamics using `timeStep()` and the other functions which are implemented. The arguments “pathsPerBin” and “numBins” determine the target number of forward shots from each RC bin, and the fineness of the RC binning.

## 5 Analysing Your Data

The S-PRES code shipped has been purposely stripped of most of the probes for things like reaction flux, committor surfaces and so on in order to present a clean API to the user. What you want to observe will depend on the type of simulation that you have in mind.

An easy observable which is written by default to standard error during the run is the occupation probability of each bin. Try scrolling down the “testAsep.elog.ref” file in `spres/test` in order to see how this evolves with time. It is quite straightforward to estimate reaction flux from this information, although getting the last few significant figures for a precise calculation requires a little more work (see the paper).

## References

- [1] Joshua T Berryman and Tanja Schilling. Sampling rare events in nonequilibrium and nonstationary systems. *J Chem Phys*, 133(24):244101, Dec 2010.