

# Software Technique in VITESS

Michael Fromme  
Arbeitsgruppe Anwendersoftware  
Hahn-Meitner-Institut Berlin

This report is an updated and more verbose treatment on VITESS software technique taken from a contribution to the 2001 workshop at HMI and updated to news of version 2.8 in 2008.

## VITESS Software

First we learn about the building blocks of VITESS and it's design principles. Then we get to know features of the VITESS graphical user interface and how to use them for new modules. Existing software tools are best used if you stick to some conventions.

To make things clear we then examine step by step how to integrate a new simulation module in a one-shows-all program example.

## *VITESS Building Blocks*

### Piped Commands

The most basic thing for VITESS programs from the beginning was that modules represent parts of the chain, from neutron source to moderators and guides up to detectors, as single programs which communicate by means of pipes, where data representing the neutron sample flow as output of one module to the input of the next module.

This makes modules fairly simple. They only have to agree on the data format for neutrons in the pipe, and could be parametrised by very individual means. In practice we have command options and parameter files to let modules know what to do.

### Parameters as Command Arguments

In the course of development it proved fruitful to have some common parameter passing syntax. Because we had no module with more than 26 parameters then it seemed sufficient to reserve a single letter for command options, and to put the adjunct argument right after with no space in between like

`-pxxx`

where `p` is the single option letter, and `xxx` is a placeholder for an arbitrary argument.

Some parameters are valid for all modules, are shown to all modules, they have the form

`--Pddd`

like here for the default parameter directory.

This is not standard for command shells, but easy to parse.

Users are not confronted by hundreds of parameters on a single command line, but use the Graphical User Interface (GUI) to generate commands with the appropriate options.

## *VITESS Design Principles*

### Multi Platform Support

You need just a C compiler like GNU `gcc` and Tcl/TK support for your computer and operating system, which means it is easy to compile and adjust VITESS for Microsoft Windows and most Unix variants. As for now VITESS is ready to be used with MS Windows and Linux (32 and 64 bit systems). It has been used with Solaris and Tru64, too.

### Easy Integration of new Simulation Modules

You may contribute own modules very easily. If your module knows how to read from standard input and write to standard output, that's all you need to use it as an external module. For tighter integration to the GUI you just enter some lines of Tcl code, and you will learn here where and why.

## **Consistency of Parameters**

With the Tcl/TK GUI you have the glue to operate your module, that is to check the consistency of parameters, control the execution of the overall pipe, save and restore parameter settings, and even visualise the output.

## **Features of the VITESS GUI**

### **Parameters**

It's up to the GUI part to provide defaults for parameters. They may be range-checked and more complex cross checks may be done here.

By now parameters are mostly of two sorts: Those which are entered and shown with the module on the main VITESS module window and those which are put aside to text parameter files. The latter are for parameters which are changed more seldom. Your C-programs read those directly, but you may edit those text files by means of the GUI, where parameter checking may be achieved easily.

Of course you may browse these files in the Windows fashion.

### **Default Parameter Directory**

Because it was too easy to mix up parameter files for different simulation runs, the GUI supports the use of a default parameter directory. Modules see a parameter `-PD:\Vites\Test` and then should read and write files relative to this directory. The GUI cares to copy files of other origin to this directory if necessary.

### **Save and Restore Settings**

VITESS saves all settings under direct control to `.gui` files. Later you may restore these settings.

You may save by the file menu, or you are asked to save if you want to load new settings or leave VITESS, and have unsaved settings.

For simulation series which need no manual interaction, you may store simulation series to `.tcl` files where specially selected parameters are varied.

### **Controlled Execution**

Simulation pipes are started by means of the GUI, which gathers and checks parameters to be transformed to a lengthy Tcl command string. The started module processes are monitored then and their activity is shown. If all modules have finished, their non-pipe output, which has been directed to temporary files, becomes shown in the output window.

You may stop or kill the command pipe at any time. To stop the pipe means to send signals to all modules to end computation. If modules are prepared for this signal, they cease execution at reasonable points and prepare their final statistics as if the stream of input neutron samples from the pipe has ceased. If this soft abort fails for some reason, processes may be killed terminally.

### **Click for Help**

To get information on parameters (meaning, valid input,..) you may just click on the parameter label. Of course you may search for these descriptions. Description also comes with HTML files for modules, where your browser gets a call when you select help from module menus.

### **Logging**

The merged output of the VITESS GUI and the non-pipe output of modules is shown in the output window and is copied to a log file, the date being part of the file name.

## **Software Tools**

### **C/C++ Compiler**

We chose GNU gcc for Unix binaries and Microsoft Visual C++ to compile Windows executables. Makefiles for both compilers are output of a Perl script `mmake.pl`, which contains all information on module dependencies.

Modules may be from other computer languages, as long they stick to the binary format of neutrons in the pipe, and know how to process arguments.

For C/C++ this is much easier, because common definitions and tool routines may be used.

## Tcl/TK

Tcl is an open source multi-platform programming shell, abstracting features of the operating system. TK brings in X-Windows and the MS Windows GUI. VITESS uses the features of the BLTWish extensions if given, but works fine with Tcl/TK version 8.4 without extensions. We use plain Tcl/TK, without any own C-extension. Because most Unix deviates have Tcl/TK installed, we distribute VITESS without this base, but for Windows we put the Scriptics compilation plus BLTWish extensions to the self-installing program. Of course this is not part of the VITESS software, and you might chose to use other or newer distributions of Tcl/TK and BLTWish.

## VITESS Conventions

Code, sources and documentation are put to specific subdirectories under the chosen installation.

## Installation Directory

the path must not contain names with blanks, do not use C:\Program Files.

install\_\*.txt tells how to install VITESS

license.txt contains terms of use.

The file vitess is the main Tcl script to be called, the first line specifies which wish is used.

### GUI

has all other \*.tcl sources which are auto-loaded

### MODULES

binary executables for modules. A Module a comes in variants

a.exe	windows
a_Linux	32 bit Intel systems
a_Linux_x86_64	64 bit, Opteron, Xeon
a_SunOS	Solaris
a_OSF1	Tru64 alpha

### SRC

has all module sources and files necessary for compilation.

### BITMAPS

auxiliary graphic files

### WWW

HTML and GIF files of the documentation

### FILES

parameter files and example directories; contains reference instrument simulations for validation of new releases

### RelNotes

release notes of VITESS releases

### TOOLS

IDL / PV-Wave routines for visualisation of neutron data  
grid\_command.pl

When installed under Unix the main Tcl/TK source vitess has to be adopted to the Tcl/TK installation path by help of the installAndTest script.

When installed by the self-extracting program generated by InnoSetup under Windows, the installation directory may contain the Tcl/TK distribution, too. These files as the subdirectories bin, demos, doc, include, and lib come from the Scriptics Tcl/TK distribution and are not part of the VITESS software itself, but are added here for convenience.

## Subversion Repository

The subversion (svn) repository at <https://www.hmi.de/svn/vitess/trunk> is a mirror of the directory tree of the VITESS installation. Developers communicate by means of subversion, where each person has his own working directory, to be committed to the repository when ready. An official release corresponds to a svn version number, like the release 2.6 to svn number 26.

Intermediate svn versions may be used by developers and interested persons, but have not undergone integration and quality tests.

Mere bug fixes between major official releases are announced, will have different svn version numbers, but just supersede the published version.

## Integrate a Module to VITESS

If you plan to add a new simulation module you probably understand the overall idea of neutrons samples in the simulation pipe, and how neutron states are coded in the C struct Neutron. You may then edit a program `example.c` to read command options, process neutrons, and write statistical output.

Assumed you work with Windows, the compiled output `example.exe` then should be copied to the MODULES directory.

Without further modification of the GUI you may then test your module as an „external command“. To use an external command with the GUI you have to specify the execution file, and all parameters as string input. Thus you may use your module immediately, but without the comfort of GUI parameter edit, check, and help.

## Program Example

In a program `example` we will comment in detail code snippets necessary for an example module `example.c`.

```
example.c
    uses controlled interrupts
general.h
    declares time,wavelength,probability as double
    and position, vector and spin as double[3]
init.h
    declares macros and utility routines of
init.c
    parses standard parameters --B --f --F --G --J --L --P --U --Z
```

## Structure of `example.c`

```
/******
/* VITESS module example */
/******
#include "init.h"
int ParA;
/* own initialisation of the module */
void OwnInit(int argc, char **argv) {
    char *arg;
    int Option;
    fprintf (LogFilePtr," \n") ;
    print_module_name("example") ;
    argv++; /* skip program name */
    while ((arg = *argv++) {
        arg++; /* skip - char */
        switch (*arg++) {
            case 'A':
                sscanf(arg, "%d", &ParA); break;
                /* ... */
        }
    }
}
```

```

void OwnCleanup() {
    /* do some final action, like logging. */
    fprintf(LogFilePtr, "\nsome logging");
}

int main(int argc, char **argv) {
    Neutron output;
    int i;
    Init(argc, argv);
    OwnInit(argc, argv);
    DECLARE_SOFTABORT
    while(ReadNeutrons() != 0)
        for(i=0; i<NumNeutGot; i++) {
            CHECK
            /* process the neutron ... */
            WriteNeutron( &output);
        }
    soft_exit:
    OwnCleanup();
    Cleanup(0.0,0.0,0.0, 0.0,0.0);
    return 0;
}

```

OwnInit should process only options which are specific for this module, others should be left for the generic Init subroutine.

ReadNeutrons is a general tool routine which sets the global variable NumNeutGot.

WriteNeutron and Cleanup are tool routines you should use. Put module specific code to OwnCleanup.

Most of your specific code will be around the „process the neutron“ comment.

All non-pipe verbose output should be printed to LogFilePtr, output to stdout is reserved for pipe neutrons, and output from different modules to stderr is mixed up to rubbish.

Macros DECLARE\_SOFTABORT and CHECK are for controlled abortion of pipe execution. CHECK should be inserted where a jump to soft\_exit is possible; computation time between successive CHECK calls should be short.

You might look at slit.c as the simplest real module to see more.

## Structure of general.h

```

#ifndef GENERAL_H
#define GENERAL_H
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

#ifdef _MSC_VER
# define M_PI 3.14159265358979323846 /* pi */
..
#endif

typedef double VectorType[3];
typedef double DoublePair[2];
typedef struct {
    TotalID      ID;
    char         Debug;
    short        Color;
    double       Time;
    double       Wavelength;
    double       Probability;
    VectorType   Position;
    VectorType   Vector;
    VectorType   Spin;
} Neutron;

```

```
#endif
```

The `GENERAL_H` `ifdef`-clause permits nested includes.  
`_MSC_VER` is defined for MS Visual / Net C++.

## Structure of `init.h`

```
#ifndef INIT_H
#define INIT_H
#include "general.h"
extern long BufferSize; /* size of the neutron input and output buffer */
extern Neutron *InputNeutrons; /* input neutron Buffer */
extern Neutron *OutputNeutrons; /* output neutron buffer */
extern long OutNeutPtr; /* points to the next free position in OutputNeutrons */
extern long NumNeutGot; /* number of neutrons read in the current batch */
extern long NumNeutRead; /* number of neutrons read in total */
extern long NumNeutWritten; /* number of neutrons written in total */
extern FILE *InputFilePtr; /* stream from which the neutrons are read */
extern FILE *OutputFilePtr; /* stream to which the neutrons are written */
extern FILE *LogFilePtr; /* stream to which things are logged*/
extern char *InputFileName; /* file to read neutrons */
extern char *OutputFileName; /* file to write neutrons */
extern char *LogFileName; /* log filename */
extern double Temp; /* Temperature of the simulation */
extern long idum; /* random number specific */
void OutputBufferFlush();
void Init(int argc, char **argv);
void Cleanup();
int ReadNeutrons();
void CopyNeutron(Neutron *source, Neutron *dest);
void WriteNeutron(Neutron *OutNeutron);
long parseLine(char *Buffer, char *ptr[]);
long ParseFileLine(FILE *In, char *ptr[]);
long LinesInFile(FILE *In);
void ReadSourceData(double* p_pCurrent, double* p_pTimeMeas);
void WriteSourceData(double p_dCurrent, double p_dTimeMeas);
void print_module_name(char name[]);

#ifdef _MSC_VER
# define CHECK /* test semaphore and go my_exit if finished */
# define DECLARE_SOFTABORT /* windows ... */
#else
# include <signal.h>
static int finish;
void abort_handler(int sig);
# define CHECK if (finish) {goto soft_exit;}
# define DECLARE_SOFTABORT signal(SIGTERM, abort_handler);
#endif

#endif
```

## GUI Integration : Editing Tcl Code

To have a smoother integration to the graphical user interface, you will edit Tcl code. You need not be a Tcl expert to do this. If you do it like it has been done for other modules, that is copy from existing code, you may do it as filling a form. Programming mistakes mostly aren't fatal, and the Tcl environment complains to give you a clue in case of an error.

Some ideas about Tcl will help. In general every text line in Tcl is a command, where command parts are separated by white space.

```
set a      5
```

sets the variable a to 5; set is the command, the number of blanks before 5 does not matter.

```
puts a + 1
```

prints „a + 1“. To print the sum 6 you need

```
puts [expr $a + 1]
```

where \$a denotes the value of variable a, and expr is the command to evaluate an arithmetic expression. The parentheses [] are to be interpreted as „evaluate what's in [] and take the result“.

Another building block is lists. The VITESS GUI relies on lists of descriptions.

```
set li {a 3}
```

defines variable li as a list of two items. The restriction „one command per line“ is lifted for lists, that is list contents between opening and closing curly brace may span many lines like in

```
set li {a
      b
      c}
```

```
set l2 [list x $li]
```

which defines variable l2 as a list of 2 items, where the second item is the nested list \$li with tree items.

```
set l2 {x $li}
```

would not work, because there is no direct variable substitution within curly braces, and the second list item of l2 would be the silly string “\$li”.

## How to add a Module like slit to the Vitess GUI

1. Declare module slit in AvailableSET, the list of known modules  
gSet slitESET ...
2. do more if your module needs special parameter files
3. Modify generateVitessCommand  
if the executable file is not slit.exe
4. Copy slit.exe from SRC/ to MODULES/
5. Put slit.html help to WWW/

You mostly edit lists to add your module and define module parameters.

gSet is a command, xxxESET is the nested list variable with almost all parameter definitions for that module. If you have a module xxx, you must provide a variable xxxESET.

If your module needs special parameter files of extension par and you want to edit these parameter files with GUI help:

1. gSet parESET for parameter files with extension par  
As exampleESET this is the list description for all parameters which may be part of a \*.par file.
2. Modify editFile  
This is more ambitious, as you write a Tcl procedure, but you may learn from others.
  1. Modify editSave  
The procedure editSave is for saving edits.
  2. Write serializeParFile  
serialising a file means writing the GUI parameters to a file or to read parameters from a file  
serializeXyzFile is the procedure to serialise data of parameter type xyz
3. Add a line in fileDialog (tools.tcl)

## Insertion of Modules `slit` and `polariser_sm` to the Module List

(`vitess.tcl`)

```
proc makeModuleSets {} {
    global AvailableSET
    set AvailableSET {
        {source {source_const_wave source_HMI source_ILL
            source_short_pulsed source_ESS source_IPNS source_ISIS source_SNS
            source_ESS_LPTS} source}
        {guide {} guide}
        ...
        {spacewindow {space slit spacewindow spacewindow_multiple grid}
            {space slit spacewindow spacewindow_multiple grid}}
        ...
        {polariser {polariser_he3 polariser_sm} {polariser_he3 polariser_sm}}
        ...
    }
}
```

`AvailableSET` is a list of module descriptions available. Because we have so many modules they are grouped like the many source modules. `spacewindow` is the name of the module group, `slit` a specific module; the second occurrence of `slit` tells which help information belongs to that module – as you can see all source modules refer to the same source help information.

Definition of Parameters for `slit` in `vitess.tcl`

```
set slitESET {
    {dist_slit float "" {"distance\n to slit [cm]" "" "" d} ge0}
    {width_slit float "" {"width [cm]" "width of rectangular slit [cm]" "" W}}
    {hite_slit float "" {"height [cm]" "height of rectangular slit [cm]" "" H}}
}
```

If you have a module `slit`, you need a Tcl list `slitESET`.

`slit` has 3 float parameters with options `-d`, `-W`, and `-H`. `dist_slit` is the internal name for the distance value which must be non-negative (`ge0`).

Parameters for `polariser_sm` are more demanding:

```
gSet polariser_smESET {
    {pfile peditablefile polariser_SM.par {"parameter\nfile" "" "" P} w pol 1}
    ...
    {}
    {x float 100 {"position\nX [cm]" "x centre position of the rectangular
geometry polariser" "" a}}
    {y float 0 {"position\nY [cm]" "y centre position of the rectangular geometry
polariser" "" b}}
    ...
}
```

Parameters may be of kind `int`, `float`, `string`, `radio` and `file`. Parameters are grouped in chunks separated by headers (`{}` or `{,This is a header“ header}`).

Each chunk shows `int`, `float`, `string`, and `file` parameters in groups of the same kind, where e.g. 3 float values are grouped per row.

The `x` parameter is a float value with default 100, labelled „position X[cm]“ with a line break after position, it has a longer help description, and corresponds to option `-a`.

Parameter `pfile` is an editable file, residing in the parameter directory, default file name is `polariserSM.par`, and the corresponding option is `-P`. The file must be writeable (`w`), it's file extension is `pol`, and this parameter must be specified (1).

Parameter names like `pfile`, `x`, `y` here are arbitrary made of letters, digits and underscores, and must be distinct for each module, but need not be different for different modules.

## General Positions and Meaning in Parameter Lists

Parameter lists should have names ending with (or at least including) ESET like singleDetectorESET.  
List items 0,1,2.. have a position-dependent meaning.  
0 depicts the first list item as Tcl counts list items 0,1,...

0 name of global variable

1 type, one of {float int string longstring select radio filename editablefile browsefile browsedir  
parfilename pareditablefile parbrowsefile moneditablefile mon2editablefile}

browse indicates entries which are selectable by a file browser

par indicates a file which must reside in the special default (parameter) directory

moneditablefile is a pareditablefile and a monitor output file of 1-dimensional data,  
where mon2editablefile is for 2 dimensional data

2 default value

3 comment list

item 0: label text

item 1: long description text (optional)

item 2: callback procedure to show long description (optional)

item 3: command option prefix string (optional)

other entries depend on type

for type `select`

4 list of pairs with {name\_appendix default\_bool}

for types `browsefile browsedir editablefile parbrowsefile pareditablefile  
moneditablefile mon2editablefile`

4 r for a readable file,

w for a valid filename

5 file extension, used to specify GUI-editable files

6 1 for mandatory

7 d for directory

for types `filename string longstring`

4 like `browsefile`

5 dummy

6 like `browsefile`

for type `radio`

4 list of selectable items

5 list of corresponding keys (may be omitted)

for types `float int`

4 min or 1 if no value 5, but a valid specification is necessary

may specify a range like `gt3` for values greater than 3, or use comparison operators are  
`eq,ne,ge,le,gt`, and `lt` like `lt100` for less than 100

5 max

for control variables additionally

6 not needed: if this is 1, then empty input is allowed

## Another Parameter Set Definition, now for Module frame

```
### Frame
###
gSet frameESET {
  {Transformation header}
  {seq radio RTM {sequence of Rotation, Translation, and Mirroring" "" S}
    {RTM RMT TRM TMR MTR MRT} {1 2 3 4 5 6}}
  ...
}
```

We include this to show a radio selection parameter, where e.g. the GUI selection RMT corresponds to parameter -S2

## Definition of parameters in \*.pol files

```
gSet polESET {
  {dx float 60 {"dimension\nX [cm]" "length of the polariser"} gt0}
  {dy float 10 {"dimension\nY [cm]" "width of the polariser"} gt0}
  {dz float 10 {"dimension\nZ [cm]" "height of the polariser"} gt0}
  {nc int 9 {"number of\nchannels" "number of channels in vertical direction"} ge1}
  {dw float 0.05 {"wall\nwidth [cm]" "width of the wall between the channels"} gt0}
  {cp float 0 {"cutoff\nprobability" "minimal probability weight transmitted"} ge0}
  ...
}
```

As you may see it is not necessary to specify an option character here, values of parameters in files are read from the file, and are not provided as GUI input value.

Parameter dx is forced to be positive here (gt0) as nc is forced to be a non-negative integer.

## Necessary Changes of Edit/Save Routines for Parameter Files

(vitess.tcl)

Editing parameter files is facilitated by just defining the Vitess parameter lists, but you must provide code to read and write the special parameter files, and add a unique parameter file extension ( `pol` in the example).

```
proc editFile {var param ext app} {
  # Edit parameters from a parameter file with extension ext.
  # $var$app is the name of the parameter file, param is a parameter for editSave.
  # This GUI generator relies on serialize${ee}File (where $ee is capitalized $ext) to read a
  # file
  # and editSave to store the results.
  ...
  switch $ext {
    chp - crs - ine - ref - san - pow - pol - iso {set special 1}
    default {
      ...
    }
  }
  ...
  if $special {
    switch $ext {
      iso {serializeIsoFile $f r $var $app}
      pol {serializePolFile $f r $var $app}
      chp {serializeChpFile $f r $var $app}
      ...
    }

    generateEntries $w.v ${ext}ESET {} $app
  } elseif {$f != "0"} {
    while {[gets $f line] >= 0} {$w.v.text insert end "$line\n"}
  }
  ...
}
```

`editFile` is called when it comes to editing a parameter file. Depending on the extension different things are to be done. You add the line `pol {serializePolFile $f r $var $app}` and provide the procedure `serializePolFile`.

`editSave` is called when a parameter file has to be stored. You again add the line `pol {serializePolFile $f r $var $app}` and provide the procedure `serializePolFile`.

```
proc editSave {var param ext app {saveAs 0}} {
...
catch {
  switch $ext {
    chp {serializeChpFile $f w $var $app}
    crs {serializeCrsFile $f w $var $app}
    san - pow {serializeSamFile $f w $var $app $ext}
    ...
    pol {serializePolFile $f w $var $app}
    default {puts $f [$w.v.text get 1.0 end]}
  }
}

proc serializePolFile {f mode var app} {
  set nlist {dx dy dz nc dw cp gx gy gz ax ay az}
  foreach l $nlist {
    upvar #0 $l$app $l
  }
  if {$mode == "r"} {
    foreach l $nlist { catch {unset $l}}
    if {$f == "0"} return
    readNumItems $f $nlist $app
  } else {
    puts $f "$dx $dy $dz\n$nc $dw $cp\n$gx $gy $gz\n$ax $ay $az"
  }
}
}
```

pol files are text files with 4 lines of 3 numbers each.

Reading is done with the help of the `readNumItems` tool routine, writing just calls the `puts` command.

GUI values are accessed by global variables with an special appendix `$app` like `dx$app`.

## ***Changes, if Module Name and Name of executable File differ***

(`comexe.tcl`)

```
### compose the VITESS command pipe string
###
proc generateVitessCommand {} {
...
for {set i 1} {$i <= $maxModule} {incr i} {
...
switch $var {
  source_HMI {set com "source$sys -S1"}
  source_ILL {set com "source$sys -S1"}
  source_short_pulsed {set com "source$sys -S2"}
  source_cws {set com "source$sys -S1"}
...
}
```

`generateVitessCommand` is the Tcl routine called when it comes to generate the overall pipe command. You need only insert a single line for your module like it is done here for `source_HMI`. As you may see it is possible to use one program `source$sys` for different modules with a silent parameter (-S in the example).

It is default behaviour to call `slit.exe` for module `slit` under Windows, so we do not add code for these simple cases.

## Changes for File Dialogue File Types

(vitess.tcl)

```
proc fileDialog {operation {ext ""} {ifile Untitled}} {  
# Type names Extension(s) Mac File Type(s)  
#-----  
set types {  
  {"All files" {*}}  
  {"X,Y ASCII files" {.dat}}  
  {"2 D Intensity files" {.out}}  
  {"chopper files" {.chp .par .dat}}  
  {"crystal description" {.crs .par .dat}}  
  {"powder sample description" {.pow .par .dat}}  
  {"sans sample description" {.san .par .dat}}  
  {"sample reflectometer description" {.ref .dat}}  
  {"inelastic sample description" {.ine .par .dat}}  
  {"elastic isotropic sample description" {.iso .par .dat}}  
  {"polarizer sm description" {.pol .par .dat}}  
  {"GUI settings" {.gui}}  
  {"Batch command files" {.bat}}  
  {"Tcl files" {.tcl}}  
}
```

If we tell the Tcl/TK browser what file type we're looking for, it restricts files shown to those of that type. Here we added that polarizer sm description files come with extensions .pol, .par or .dat.

# Speeding up Your Simulation

## *Split the Pipe*

If you vary mostly parameters of modules in the end of the pipe, you may save much time if you save the output of the first part of the pipe to a file. This file then may be read over and over again.

This file should reside on a local disk. If you have a high number of trajectories, consider to use one of the GNU random number generators instead of ran3.

## *Switch to Batch Processing*

Some simulation started on your laptop may take too long there. It is easy to switch to different hardware, only copy your instrument definition and contents of the working directory to a Linux server. You may work with the GUI there, too. but for simulations lasting days it might be better to save Tcl commands and start pipes from the command shell.

## *Use a Grid*

If you know how to start batch jobs on a Linux machine it's only a small step to use a grid of compute nodes. We give some details in the final chapter.

## *Who is to blame?*

If you have a pipe of 20 modules it may seem all modules have equal impact on the performance. If you watch a working Vitess pipe with the task manager under Windows or the top command under Linux, you will see the truth. It's one module in most cases which makes the CPU load, and that module probably is a source module, or a module which absorbs neutrons like choppers do, or a module with complicated reflection/absorption calculations like sm\_ensemble.

Reading the documentation more carefully you may find parameter settings for that module, which by taking into account some extra knowledge you may provide speed things up, or you may find that this module is inappropriate at all.

If you have the feeling that this module just wastes time for nothing, you could debug that module. Cut the pipe before that module to have a neutron input file. Asking the author of the module by email may be easier.

## *Using Pipes for Grid Execution*

When a simulation takes very long time, you may consider computer grids to do the lengthy number crunching.

The Vitess GUI will not be of much help then, because you won't sit there watching ... in the GUI's output window. Of more concern: You may not have a Tcl/Tk installation on grid nodes, and you are told to use the Grid Engine at hand.

The method is to use the GUI to generate the pipe for your instrument and do first runs interactively to sort out simple errors. Then you save the pipe with "Save as Grid Command", which generates a shell file to be used on a grid engine. The Perl script grid\_command.pl from the TOOLS directory then helps to execute the pipe on the computer grid.

## *How to use parallel Pipes*

If you start a pipe on n nodes without modification, the only good result can be that all n outcomes are identical. You probably do not want this, but to improve the statistics of the simulation, to make the sample n times as big.

The pipe output is strictly determined by pseudo random generators. We assume that you double your statistical sample if you start two pipes with different random number seeds, keeping the number of trajectories of each pipe, and merge the result appropriately.

There are obstacles in the way: If some module in the pipe uses a filename for output, this filename is used by n pipes, probably mangling output to garbage. If your pipe has some graphical output this won't do good on grid execution. "Save as Grid Command" may warn you about that, but side effects may be subtle.

The difficulty is to restrict the pipe to one output file of the last module, or to that of the module before. If only the last module generated a file, the output files of parallel pipes are to be concatenated. If the last module just consumes neutron like a detector does, the concatenated output of n pipes is to be fed to the last module in a serial step after parallel execution of n pipes.

grid\_command.pl deals with file output of the final pipe module, or if that has no output file, chops off the last module and arranges parallel execution of the modules before, and lets them write part result files. In a serial step the concatenated output then is fed to the final module.

Normally all modules of the pipe get the same random seed with the `-Z` parameter. For parallel execution of pipes this is changed, each module of each pipe gets a unique random seed by `grid_command.pl`.

## Gain

Parallel execution is of value even for single computers. With multi-CPU (m) boards of dual core CPUs it may be useful to do up to  $m * 2$  ( $m * 4$  for quad core) parallel pipes. If you want to use multi-node grids, you must provide some synchronisation of pipes, if you have the second form where the result is gathered in a final serial step.

`grid_command.pl` uses Unix commands and Perl, which are available on most grid engines. You may use the script without a grid engine on a multi-CPU PC with Linux.

It may be worth doing it with plain tcl for multi-CPU windows PCs.

## Parallel Execution Example at HMI

I'm user JOE, my working directory is `/net/home/JOE/vitess` on the Linux server `dirac` at HMI.

After proper building an instrument with VITESS using the working directory, I found my simulation does take so long I'd like to speed that up using the Grid Engine on the `dirac` cluster.

So first I create the file `/net/home/JOE/vitess/instrument.grd` using the Vitess "Save Grid Command" menu. I do this on a Linux system like `dinux4`, `dinux5` or `dirac` itself, to have the same hardware architecture, `x86_64` in this case. Contents of this file are

```
V=/net/home/JOE/vitess/MODULES
P=/net/home/JOE/vitess/FILES/MyInstrument
Z>--Z
L>--L
F>--F
$V/source_Linux_x86_64 -S1 ...
```

To submit a batch job using 4 pipes on a single node of the Grid Engine I call

```
grid_command.pl instrument.grd 1 4 /home/JOE/vitmod /home/JOE/vitwork -l cpu=8 -l vf=1G
```

I use the directory `/home/JOE` because this file system is visible on the grid nodes, too.

The script creates the directories `/home/JOE/vitmod/` and `/home/JOE/vitwork/` if they do not exist.

Because `/net/home/JOE/vitess/MODULES/` is invisible for `dirac` compute nodes, all used `*_Linux_x86_64` executables are copied to `/home/JOE/vitmod`, plus this script itself, and referenced files from `/net/home/JOE/vitess/FILES/MyInstrument` plus `instrument.grd` are copied to `/home/JOE/vitwork`.

`grid_command.pl` prepares a batch command file `/home/JOE/vitwork/job<x><time>`, where

```
x    is for number of nodes, time Unix time
-0 4 is special for single node execution, 4 pipes
```

```
#!/bin/bash
# Grid Engine Parameters
#$ -S /bin/bash
#$ -cwd
#$ -l cpu=8
#$ -l vf=1G
/home/JOE/vitmod/grid_command.pl /home/JOE/vitmod/instrument.grd -0 4 /home/JOE/vitmod /home/JOE/vitwork
```

The job is submitted internally from `grid_command.pl` with the Grid Engine submit command `qsub`.

# Contents

VITESS Software.....	1
VITESS Building Blocks.....	1
Piped Commands.....	1
Parameters as Command Arguments.....	1
VITESS Design Principles.....	1
Multi Platform Support.....	1
Easy Integration of new Simulation Modules.....	1
Consistency of Parameters.....	2
Features of the VITESS GUI.....	2
Parameters.....	2
Default Parameter Directory.....	2
Save and Restore Settings.....	2
Controlled Execution.....	2
Click for Help.....	2
Logging.....	2
Software Tools.....	2
C/C++ Compiler.....	2
Tcl/TK.....	3
VITESS Conventions.....	3
Installation Directory.....	3
Subversion Repository.....	4
Integrate a Module to VITESS.....	4
Program Example.....	4
Structure of example.c.....	4
Structure of general.h.....	5
Structure of init.h.....	6
GUI Integration : Editing Tcl Code.....	7
How to add a Module like slit to the Vitess GUI.....	7
Insertion of Modules slit and polariser_sm to the Module List.....	8
General Positions and Meaning in Parameter Lists.....	9
Necessary Changes of Edit/Save Routines for Parameter Files.....	10
Changes, if Module Name and Name of executable File differ.....	11
Changes for File Dialogue File Types .....	12
Speeding up Your Simulation.....	13
Split the Pipe.....	13
Switch to Batch Processing.....	13
Use a Grid.....	13
Who is to blame?.....	13
Using Pipes for Grid Execution.....	13
How to use parallel Pipes.....	13
Gain.....	14
Parallel Execution Example at HMI.....	14