
VORPAL User Guide

Release 5.0

Tech-X Corporation

September 20, 2011

CONTENTS

1	Introduction	1
1.1	Setup and Installation	1
2	Basic Concepts	5
2.1	Basic Concepts for All Simulations	5
2.2	Cells and Dimensions	5
2.3	Pre File Syntax	10
2.4	Output Files	12
2.5	Particle Data	12
3	Input File Basics	13
3.1	Python Token Evaluator (txpp.py)	13
3.2	Basic Features	15
3.3	Symbol Definition	15
3.4	Scoping Rules	16
3.5	Expression Evaluation	16
3.6	Using Macros in Input Files	16
3.7	Macro Parameters	17
3.8	Macro Overloading	18
3.9	Defining Functions Using Macros	18
3.10	More About Parameters	19
3.11	Importing Files	19
3.12	Flow Control and Repetition	20
3.13	Conditionals	20
3.14	Repetition	20
3.15	Recursion	21
3.16	Symbol Definition on the Command Line	21
3.17	Requires	22
3.18	String Concatenation	22
3.19	VORPAL Macros	22
3.20	adimacros.mac Macros	24
3.21	applyPML Macro	25
3.22	deymittra.mac Macros	26
3.23	deyMitraYeeInterpol Macro	26
3.24	dmAdiUpdaters Macro	27
3.25	dmYee Macro	28
3.26	Yee Macro	28
3.27	Geometry Macro	29
3.28	mathphys	37

4	Simulations	39
4.1	Running VorpalComposer in Parallel	39
4.2	Running VORPAL from the Command Line	40
4.3	HDF5 Format VORPAL Output Files	44
4.4	Advanced Topics in Electrostatic Simulation	49
4.5	Advanced Topics in Electromagnetic Simulation	51
4.6	Advanced Topics in Particle Simulation	53
4.7	Monte Carlo Interactions Package	54
5	GPU Computing	59
5.1	GPU Computing	59
6	Troubleshooting	61
6.1	Troubleshooting Electrostatic Simulations	61
6.2	Troubleshooting Electromagnetic Simulations	62
7	References	63
7.1	Bibliography	63
7.2	Glossary	63
8	Copyright and Trademark Information	65
9	Indices and tables	67

INTRODUCTION

1.1 Setup and Installation

Supported operating systems and required software programs are listed below. The default directory structure used for the software instructions in this document are also described in this section.

1.1.1 Supported Operating Systems

VORPAL is supported on:

- Linux
- Macintosh OS X
- Windows XP and above

Both serial execution and parallel execution are supported on these operating systems. You can obtain support for other computing platforms and operating systems via Tech-X Professional Consulting Services.

1.1.2 Software Utilized by VORPAL

VorpalComposer

VorpalComposer is a graphical user interface you can use to create input files for use with the VORPAL plasma simulation code; VORPAL is run from within the VorpalComposer. VorpalComposer is included with the VORPAL distribution. The VorpalComposer editor and validator have built-in functions and graphical components that help you to create well-formed input files. Several example input files, ranging in complexity from beginning to advanced, are included with VorpalComposer. New VORPAL users can use these examples as templates. Advanced VORPAL users can use VorpalComposer to validate the syntax of their own input files, whether their files have been created using VorpalComposer or by using a text editor.

The Visualization tool in VorpalComposer is a simulation data visualization application based on [VisIt](#).

VORPAL

VORPAL runs as both a serial code for and as a parallel code for multi-processor / multi-core systems that support MPI. VORPAL is embedded within VorpalComposer. No user installation is required.

Python

Python is an open-source, interpreted scripting language managed by the Python Software Foundation. For more information about Python, see: <http://www.python.org/>

VORPAL uses Python to process input files, allowing users to set up simulations with math functions, variable substitutions, and macros.

VORPAL uses its own embedded version of the Python interpreter to pre-process input files and execute any Python code in an input file. No user installation of Python is required.

1.1.3 Optional Software

MPI

Message Passing Interface (MPI) is an application programming interface (API) for communicating between processes that execute in parallel. MPI is not required if you are only running a serial version of VORPAL.

VORPAL uses its own embedded version of MPI to allow VORPAL to make use of multicore processors and clusters. No user installation of MPI is required.

VORPAL supports two implementations of MPI:

- OpenMPI, available at <http://www.open-mpi.org/>
- Vendor-specific MPI implementation from Microsoft on machines running Windows XP / Vista / 7, available at <http://www.microsoft.com/>

VorpalView

VorpalView is a deprecated feature as of VORPAL 5.0.

VorpalView is written in IDL and runs on the IDL virtual machine. The IDL virtual machine, which is available at no cost for Linux, Windows, Mac OS X, and other common UNIX variants, is included in the VORPAL distribution.

The VorpalView program is in the file vorpalview.sav. VorpalView is included with the VORPAL product distribution.

1.1.4 VORPAL Installation Instructions

Windows VORPAL Software Distribution

The VORPAL distribution package for Windows, both 32-bit and 64-bit, is a self-extracting executable that includes:

- VORPAL physics simulation software
- VorpalComposer, the graphical user interface to VORPAL
- VorpalView, a complimentary (and optional to install) free-standing visualization tool
- VorpalView documentation (optional to install)

Windows VORPAL Software Installation

On Windows, invoke the installer by double clicking on it. The default installation path is:

For 32-bit application on 32-bit Windows:

`C:\Program Files\Tech-X\Vorpal 5.0`

For 32-bit application on 64-bit Windows:

`C:\Program Files (x86)\Tech-X\Vorpal 5.0`

For 64-bit application on 64-bit Windows:

`C:\Program Files\Tech-X (Win64)\Vorpal 5.0`

To open the VORPAL software, go to the Start Menu, click on the Tech-X folder, click on Vorpal 5.0, then click on VorpalComposer.

Mac OS X VORPAL Software Distribution

The VORPAL distribution package for Mac OS X is a .dmg installer that includes:

- VORPAL physics simulation software
- VorpalComposer, the graphical user interface to VORPAL
- VorpalView, a complimentary (and optional to install) free-standing visualization tool
- VorpalView documentation (optional to install)

Mac OS X VORPAL Software Installation

On Mac OS X, invoke the installer by double clicking on it. Drag the VorpalComposer icon into your Applications folder (visible in the installer window). Double-click on the VorpalComposer icon to run VORPAL. This default installation path is:

`/Applications/VorpalComposer.app`

Linux VORPAL Software Distribution

The VORPAL distribution package for Linux is a gzipped tarball that includes:

- VORPAL physics simulation software
- VorpalComposer, the graphical user interface to VORPAL
- VorpalView, a complimentary (and optional to install) free-standing visualization tool
- VorpalView documentation (optional to install)

Linux VORPAL Software Installation

On Linux, unpack the gzipped tarball in the directory in which you wish to install VORPAL. A typical location would be

`/usr/local/VorpalComposer`

To run VORPAL, execute the command

`./VorpalComposer.sh`

from within the installation directory.

1.1.5 VORPAL Documentation

VORPAL documentation is accessible from within the VorpCom interface, as well as online at the Tech-X web site (<http://vorpal.txcorp.com/>).

Learning VORPAL by Example

Learning VORPAL by Example provides numerous tutorials for both beginning and advanced VORPAL users.

VORPAL User Guide

The *VORPAL User Guide* contains comprehensive VORPAL documentation, including directions for running VORPAL from the command line, and describes support resources. When you are ready to create your own simulation, consult this manual for in depth information about VORPAL features.

VORPAL Reference Manual

The *VORPAL Reference Manual* provides a quick reference to VORPAL input file code element syntax. The reference lists each VORPAL code element, its parameters, and a sample code snippet demonstrating use of that code element in context of a VORPAL input file. VorpCom provides ready access to the contents of the VORPAL Reference Manual so that you can check the syntax of your code.

VorpView User Guide

For instructions about using VorpView, consult the *VorpView User Guide*. The *VorpView User Guide* is bundled with the VorpView distribution. VorpView is included in your VORPAL distribution.

BASIC CONCEPTS

2.1 Basic Concepts for All Simulations

This section presents concepts about VORPAL that you should understand before creating and running VORPAL simulations. Taking the time to examine these concepts before reading about the simulation process will make the simulation procedures simple to understand and tutorial lessons straightforward to follow.

2.1.1 Simulation Grid

Every object in a VORPAL simulation interacts with a 3D grid, in either Cartesian or cylindrical coordinates. Fields are represented on the grid, and positions relative to the grid determine which particles interact through collisions and how fields are interpolated to the particles.

2.2 Cells and Dimensions

This section discusses basic concepts involved in setting up VORPAL's simulation grid. Understanding simulation grid setup is a prerequisite to defining a grid in a VORPAL pre file as described in *the-simulation-process* from *Learning VORPAL By Example* as well as the simulation tutorials from the same document. For the purposes of this guide, all examples are represented in Cartesian coordinates.

The grid consists of:

- cells
- set of dimensions, or lengths, along the x, y, and z axes

The number of cells in the grid determines the resolution of the simulation.

This document's illustrations are based on a right-handed coordinate system. In this coordinate system, VORPAL lays out its axes as follows:

- X = left to right
- Y = front to back
- Z = bottom to top

The example simulations in this document refer to lower and upper ends of the simulation, where the X, Y, Z, coordinate mapping references in explanations are:

- Left X-lower
- Right: X-upper

- Front: Y-lower
- Back: Y-upper
- Bottom: Z-lower
- Top: Z-upper

All units in VORPAL are expressed in SI (meters, kilograms, and seconds).

Note: VORPAL ignores higher dimension information when it is provided for a lower dimension (that is, when 3D information is provided for 1D or 2D), simulations.

You can specify the location of the grid's start point: the lower left, front corner of the grid (that is, the position at which $x = 0$ cells, $y = 0$ cells, and $z = 0$ cells). If you do not specify the start point's location, VORPAL assigns the start point a location of $x = 0$ m, $y = 0$ m, and $z = 0$ m.

Note: Tech-X recommends that you start with a low-resolution simulation and make sure that it runs to completion without any problems. Then increase the resolution as needed.

Let's examine a 3D Cartesian grid for a simple electromagnetic simulation in which:

- The grid has 40 cells along the x axis, 20 cells along the y axis, and 20 cells along the z axis.
- The lengths of the grid are $X = 1.6e-5$ m, $Y = 5.e-5$ m, and $Z = 5.e-5$ m.
- The start point is located at $X = 0$ m, $Y = -2.5e-5$ m, and $Z = -2.5e-5$ m.
- The dimensions of each cell are $4.e-7$ m along the x axis, $2.5e-6$ m along the y axis, and $2.5e-6$ m along the z axis.

The *figure* of the Cartesian grid showing cells and dimensions shows the 3D Cartesian grid for this example. The lower bounds of the grid are at $x = 0$ cells, $y = 0$ cells, and $z = 0$ cells. The upper bounds are at $x = 40$ cells, $y = 20$ cells, and $z = 20$ cells. The user-defined cells are in red. The user-defined dimensions are in blue. The gray cuboid encloses the cells in the grid.

Note: The cells in the figure's Cartesian grid showing cells and dimensions and Cartesian grid extended by VORPAL are not drawn to scale. The dimensions of each cell along the y and z axes are actually 6.25 times larger than the dimension of the cell along the x axis.

VORPAL uses guard cells, which are cells located outside the user-defined simulation grid, for parallel processing and other computational purposes. Charges cannot be deposited in guard cells, but you can use guard cells when you describe boundary conditions. VORPAL extends the user-defined 3D Cartesian grid by:

- One guard cell at the lower end of the x axis and two guard cells at the upper end
- One guard cell at the lower end of the y axis and two guard cells at the upper end
- One guard cell at the lower end of the z axis and two guard cells at the upper end

The user-defined Cartesian grid with 40 cells along the x axis, 20 cells along the y axis, and 20 cells along the z axis is extended by VORPAL to 43 cells along the x axis, 23 cells along the y axis, and 23 cells along the z axis. The lower bounds of the grid extended by VORPAL are at $x = -1$ cells, $y = -1$ cells, and $z = -1$ cells. The upper bounds are at $x = 42$ cells, $y = 22$ cells, and $z = 22$ cells.

The *figure* of the Cartesian grid extended by VORPAL shows how VORPAL adds additional guard cells to extend the user-defined grid. The user-defined cells are in red. The user-defined dimensions are in blue. The guard cells that VORPAL adds to extend the grid are in green. The gray cuboid now encloses both the user-defined cells and the guard cells added by VORPAL.

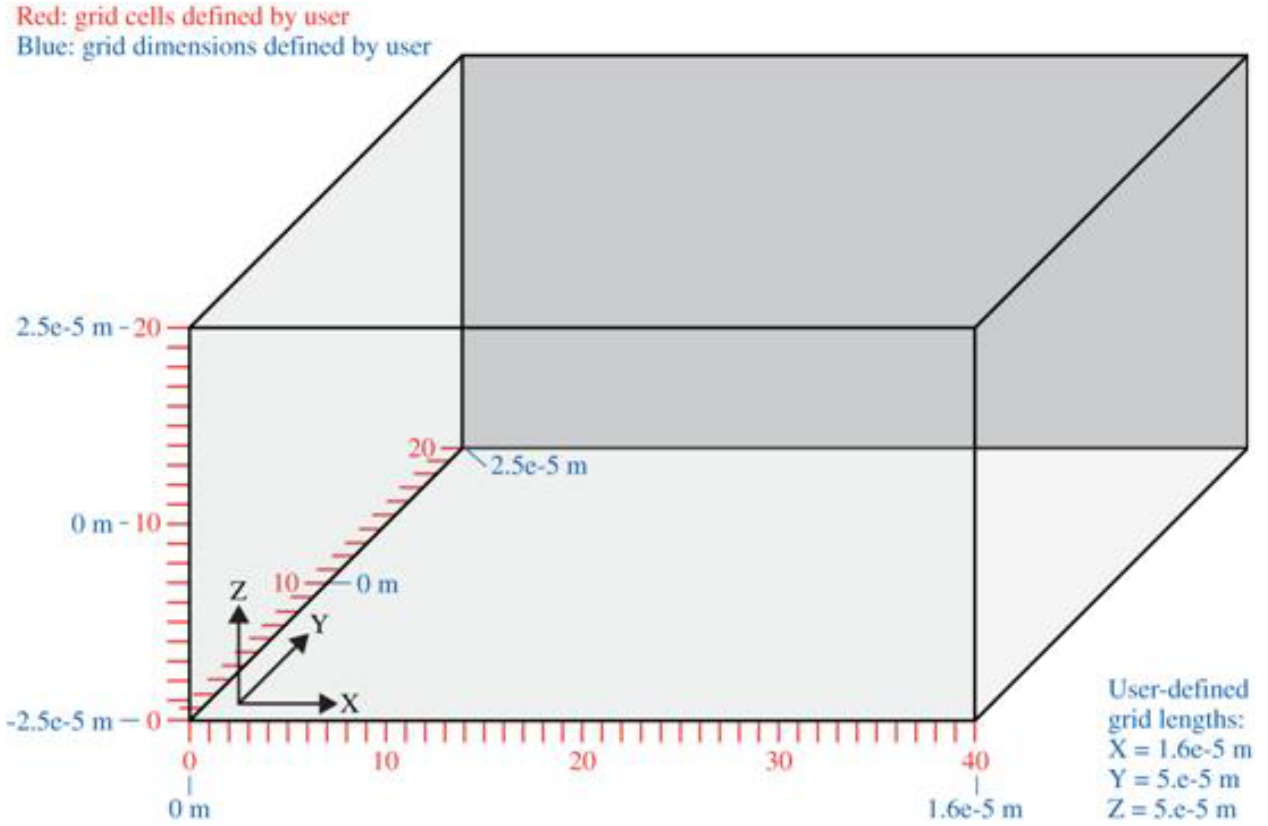


Figure 2.1: Cartesian grid showing cells and dimensions

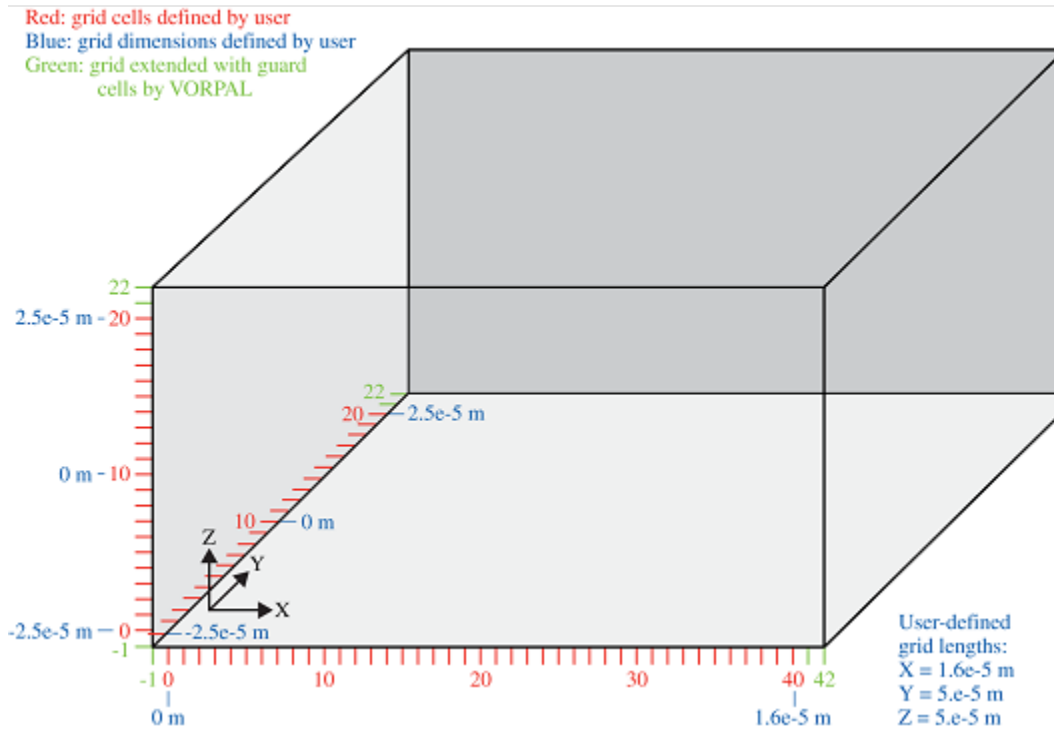


Figure 2.2: Cartesian grid extended by VORPAL

2.2.1 Electric and Magnetic Fields

On a Yee mesh, electric fields are located on the edges of grid cells, and magnetic fields are located on the faces of the cell surfaces as shown in the *figure*.

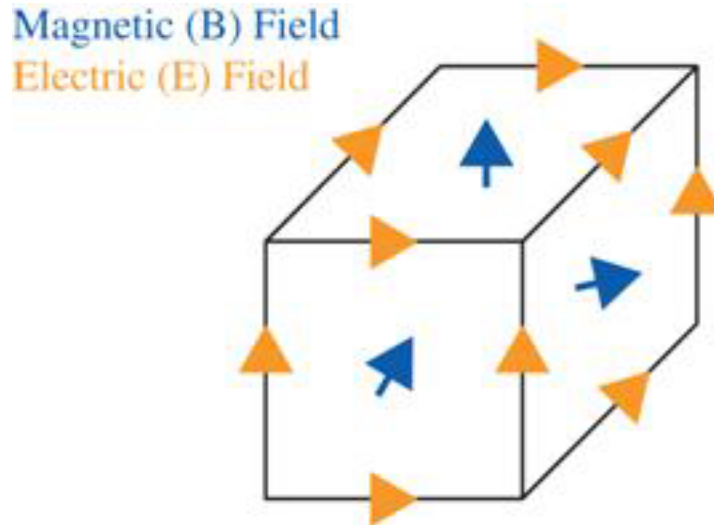


Figure 2.3: Electric and magnetic fields on a grid cell

2.2.2 Electromagnetic and Electrostatic Boundaries

To describe an electromagnetic or electrostatic boundary, we specify the lower bounds and upper bounds of the cells. Let's examine this concept on a 2D user-defined grid with 10 cells along the x axis and 5 cells along the y axis. In the below *figure* observe the top portion of the 2D extended grid showing lower bounds and upper bounds for electric charges. The user-defined grid cells are in red. The guard cells added by VORPAL are in green.

We want to define the bounds for electric charges in a row of cells at the bottom of the grid. The lower bounds are at $'0, 0'$. The upper bounds are at $'10, 1'$.

We also want to define the bounds for electric charges in a row of cells at the top of the grid. The lower bounds are at $'0, 5'$. The upper bounds are at $'10, 6'$.

The lower portion of the figure, 2D extended grid showing lower bounds and upper bounds for electric charges highlights the cells enclosed in the bounds for the electric charges in both rows of cells.

Note: The coordinates for defining upper and lower bounds extend from -1,-1 to 10,6. For electromagnetic and electrostatic boundaries, these coordinates do not include the last row of guard cells: the guard cells at the upper end of the x axis and the upper end of the y axis. (On a 3D grid, the coordinates do not include the row of guard cells at the upper end of the z axis.) For particle boundaries, the coordinates do include the last row of guard cells. We will discuss particle boundaries in more detail in the *es-tutorial-intro* and *em-tutorial-intro*.

For more information about the simulation grid, see the *VORPAL-Reference-Manual*.

Now that we have examined the VORPAL simulation grid, we are ready to examine the syntax used to define a simulation model on the grid. We begin with an overview of a pre file, which is the basic input file for a VORPAL simulation.

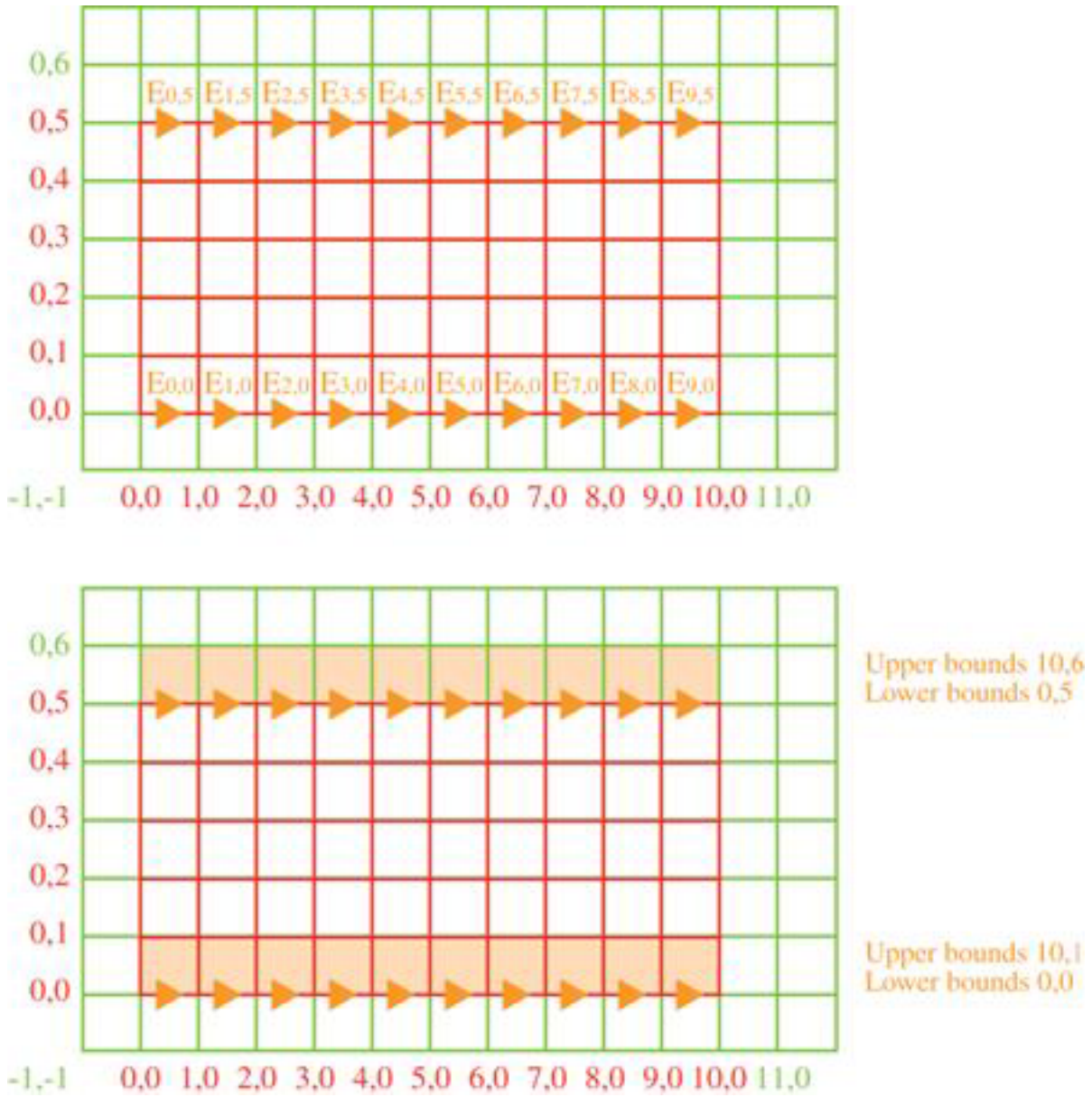


Figure 2.4: 2D extended grid showing lower bounds and upper bounds for electric charges

2.3 Pre File Syntax

The most important part of the VORPAL simulation process, which we will examine in *the-simulation-process* is creating a pre file. You define the simulation and its variables in the pre file, which has a .pre file suffix.

This section discusses the syntax used in pre files.

A pre file consists of:

- Comments
- Variables
- Top-level simulation parameters
- Parameters and vectors of parameters organized into input blocks

2.3.1 Comments

You can enter comments in either of two ways:

- Following a pound sign (#) either on a new line or a continuation of a current line
- Between the opening and closing comment tags `<Comment>` `</Comment>`

Note: Tech-X recommends that you always update your comments when you make changes to a pre file. The reasoning behind a change may become unclear if you do not provide comments that explain why you made the change. Pre files with old, out-of-date comments are difficult to work with.

2.3.2 Variables

Each line defining a variable begins with a dollar sign (\$).

2.3.3 Parameters

Parameters can be integers, floating-point numbers, or text strings.

The format of the parameter value determines the type of parameter. For example:

- `x = 10` indicates an integer
- `x = 10.0` indicates a floating-point number
- `x = ten` indicates a text string

Some parameters accept any text string (within reason). Other parameters accept only a choice of text strings.

If VORPAL can parse a value, such as '42', as an integer, it will do so. If VORPAL cannot parse the value as an integer, it will attempt to parse it as a floating-point number – for example, any of the following:

```
42.  
3.14159  
1.60217646e-19
```

If VORPAL cannot parse the value as either an integer or a floating-point number, it will parse the value as a string of text, for example, either of the following:

4o. (4 and lowercase O) or
40 (4 and uppercase O).

Use a decimal point to specify a floating point number. You must write floating-point numbers with a decimal point so that VORPAL will not interpret them as integers. If you want to assign an integer value to a floating-point parameter, make sure you write it as ‘3’. (with a decimal point ‘.’) rather than only the numeral ‘3’ (without a decimal point). If you write the number as an integer (without the decimal point), VORPAL will interpret it as an integer. This will likely produce unexpected results.

Check that you have correctly defined parameter values. If you incorrectly define a parameter that has a default value, VORPAL will use the default value and give you unsatisfactory results. If you incorrectly define a parameter that does not have a default value, VORPAL may crash, fail to compute the physics of the simulation, or ignore the incorrectly defined parameter and give you unsatisfactory results.

Do not specify a parameter twice. If you do, VORPAL will use the second occurrence of the parameter in the input file produced from the pre file. Although VORPAL allows you to specify parameters and input blocks in many different sequences, if you follow the recommendations in this user guide, you should not have a problem with specifying parameters twice.

2.3.4 Vectors of Parameters

Vectors of parameters are enclosed by brackets ‘[]’ with white space used as separators. For example:

- ‘x = [10 10 10]’ indicates a vector of integers
- ‘x = [10. 10. 10.]’ indicates a vector of floats

2.3.5 Input Blocks

Input blocks are used to create simulation objects. The block is enclosed by opening and closing tags such as:

```
<Grid globalGrid>
.
.
.
</Grid>
```

The tag determines:

- **object type:** indicated by an initial capital letter, for example, ‘Grid’
- **object name:** indicated by an initial lowercase letter, for example, ‘globalGrid’

You use the object name to refer to the object in other input blocks. For example, in the input block for a particle object, you may refer to the name of the electromagnetic field object.

Input blocks can be nested. For example, input blocks for boundary conditions are nested within the input block for an electromagnetic field.

If you have not already examined *the-simulation-process* from *Learning VORPAL By Example*, then now is the time to do so. Otherwise, we move on next to programming terminology used through VORPAL that is necessary to understand before delving further into the *VORPAL User Guide*.

2.4 Output Files

All data produced by VORPAL is in MKS units and output using the HDF5 data file format. These output files are dumped as defined by the user; they can be written at the end of a simulation, for example, or every n steps to create time series that can be used to see how a system evolves over time. Additionally, these output files can be used to restart a run and continue from a given point; for example, if a user has run a simulation for 1000 time steps and wishes to see how the simulation progresses if run for another 1000.

Execution of VORPAL leads to the generation of the data from the simulation in the form of HDF5 ‘.h5’ files. There are separate executables for serial and parallel runs. This section describes how VORPAL is run in serial and parallel and the data files it generates. Not all methods of parallel execution can be covered, as there are many different ways of invoking an MPI-based, parallel application.

2.5 Particle Data

VORPAL stores six quantities for particles in the HDF5 data files: position in x , y , and z , and relativistic velocity in x , y , and z .

$$x, y, z, \frac{p_x}{m_0}, \frac{p_y}{m_0}, \frac{p_z}{m_0}$$

Relativistic velocity comes from the momentum of the particle:

$$p = mv = m_0v\gamma$$

where gamma is:

$$\gamma = \frac{1}{\sqrt{1 - \frac{v^2}{c^2}}}$$

Gamma can be obtained directly from the values stored by VORPAL with some algebra:

$$\frac{p}{m_0} = \gamma v = u; \gamma = \sqrt{1 - \frac{u^2}{c^2}}$$

Energy, then, would be:

$$E = (\gamma - 1)m_0c^2$$

INPUT FILE BASICS

Input Parser

3.1 Python Token Evaluator (txpp.py)

The Python pre-parser has the following features:

- It accepts a file, conventionally with suffix `.pre`, for processing.
- Lines in that file that start with the character `'$'` are processed by the pre-parser.
- Those lines are sent through the python interpreter to for evaluation
- The resulting values are replaced and written to a new file with suffix, `.in`

For example, suppose one has a file, `myfile.pre`, containing,

```
$ LIGHTSPEED = 2.9979e8
$ LX = 1.e-6
$ NX = 20
$ DX = LX/NX
$ DT = DX/LIGHTSPEED
<Grid thegrid>
    numCells = [NX]
    lengths = [LX]
</Grid>
dt = DT
```

Execution of:

```
<txpp.py directory>/txpp.py --prefile=myfile.pre
```

produces a file, `myfile.in` that contains:

```
#$ LIGHTSPEED = 2.9979e8
# --> LIGHTSPEED = 299790000.0
#$ LX = 1.e-6
# --> LX = 9.9999999999999995e-07
#$ NX = 20
# --> NX = 20
#$ DX = LX/NX
# --> DX = 4.9999999999999998e-08
#$ DT = DX/LIGHTSPEED
# --> DT = 1.6678341505720671e-16
```

```
<Grid thegrid>
  numCells=[20]
  lengths=[9.9999999999999995e-07]
</Grid>
```

```
dt=1.6678341505720671e-16
```

This mechanism facilitates modifying files to change systems size, resolution, or other parameters while keeping requisite mathematical relationships intact.

The preparser imports math, so one can include statements such as:

```
$ PI = math.pi
```

and then use the variable PI in the pre file. In addition, the replace occurs for commented lines as well, so the `myfile.pre` could have contained the line,

```
# dx = DX
```

and then `myfile.in` would have contained the line:

```
# dx = 4.9999999999999998e-08
```

This is useful for printing out intermediate values for, e.g., debugging.

The pre files can be made self executing by added the stanza:

```
#!/bin/sh
$VPUTILSDIR/txpp.py --prefile=$0 $*
exit $?
```

to the top, where `'VPUTILSDIR'` is an environment variable that gives the directory of the preparser. The preparser then knows to skip lines up to `exit` before processing the file. In addition, the value of any variable named `'NDIM'` defined in the pre file can be modified on the command line with the directive `'-ndim = 2'`, for example, to have all occurrences of `NDIM` in the file replace by `'2'` instead of the value defined in the file. This enables writing only a single pre file for simulations of multiple dimensionalities when the differences in the file follow from the value of `NDIM` alone.

3.1.1 Interpretation of Parameters and Values

The Python pre-parser simplifies setup of complex simulations by providing an abstraction mechanism. With the pre-processor's macro features, you can write parameterized input files and encapsulate input file fragments into libraries for later use. By using a macro (described in detail later) you can easily reuse the same code in more than one place. Rather than repeating the code, you just call the single macro to be substituted for that code each place you would like to use it.

The type of value expected by VORPAL for each parameter and variable is identified in parentheses next to the name of that parameter or variable in the description of each VORPAL feature or block. The VORPAL input parser distinguishes between types for integers, floating-point numbers, and strings based on input format. Expressions are parsed for values following the order:

- integer
- floating-point
- string (text)
- vector

integer If a symbol can be parsed as an integer, VORPAL assumes that integer is the correct type.

Example of an integer:

```
'42'
```

floating point If the integer format comparison fails, the software tries to parse the symbol as a floating-point number. Floating-point numbers must be written with a decimal point so they will not be interpreted as integer parameters.

Examples of floating-point numbers:

```
'3.14159'
```

```
'1.60217646e-19'
```

To assign an integer value to a floating-point variable, make sure it is written with a decimal point, otherwise, the variable will be interpreted as an integer, and this will likely produce unexpected results.

Example of correct integer value format for usage with a floating-point variable:

```
'3.'
```

Example of incorrect integer value format for usage with a floating-point variable:

```
'3'
```

string If the integer and subsequent floating-point comparisons both fail VORPAL interprets the variable as string of text characters rather than a number.

vector When parsing a vector type of variable, VORPAL uses the same logic and order of analysis as for non-vector variables. For example, VORPAL would interpret the following example as a vector having three integer components:

```
'lowerBounds = [-1 -1 -1]'
```

Input File Elements:

3.2 Basic Features

The VORPAL pre-parser, 't_xpp', is a *top-down translator*. This means 't_xpp' processes a given input file on a line-by-line basis, making substitutions and calculations as they are encountered.

3.3 Symbol Definition

In VORPAL, symbols are defined by assignment, similar to many other programming languages. For example, to define a given symbol with an expression, the syntax is:

```
$SYMBOL = EXP
```

where 'SYMBOL' is the name of the symbol and 'EXP' is any valid expression.

The expression 'EXP' is a valid expression. See the section *Expression Evaluation* for details.

The pre-parser will not try to substitute a symbol on the left hand side of an equal sign '='. For example, the following code snippet:

```
$charge = 1.6e-19
charge = charge
```

results in:

```
charge = 1.6e-19
```

3.4 Scoping Rules

Symbols in txpp are *scoped*. This means that the effect of a symbol's definition is confined to the macro or block in which that symbol is defined. Whenever txpp enters a macro or a new input file block, it enters a new scope.

In the case in which 'SYMBOL' is defined in multiple scopes, txpp ignores the previously defined 'SYMBOL' for the duration of the current scope. In the case in which 'SYMBOL' is defined more than once in the current scope, the new value overrides the previous value defined in the current scope.

This scope is closed once txpp leaves the block or macro. That is, the symbol's definition no longer has an effect once txpp has used the symbol's value in the macro or block where it was defined and then proceeded to a different block or macro. Scoping allows the next block or macro to be free to redefine the value of the symbol for its own purposes.

3.5 Expression Evaluation

Txpp evaluates expressions by interpreting them as Python expressions. Python expressions are composed of tokens. A token is a single element of an expression, such as a constant, identifier, or operation. The pre-parser breaks the expression string into individual tokens then performs recursive substitution on each token. Once a token is no longer found to be substitutable, the pre-parser tries to evaluate it as a Python expression. The result of this evaluation will then be used as the value of this token. All the token values are then concatenated and again evaluated as a Python expression. This result will then be assigned to the symbol.

Tokenizing, the act of breaking a string into tokens, is performed following the lexical rules of Python. This means that white spaces are used to delimit tokens, but are otherwise entirely ignored.

Note: A string within matched quotes is treated as a single token with the matching quotes removed.

The input files generated by txpp are sensitive to white spaces; as a result, txpp has to re-introduce white spaces in the translation process. By default, tokens are joined without any white spaces. However, if both tokens are of type string, then a white space is introduced. Also, tokens inside an array (delineated by '[' and ']') are delimited by a white space.

See the Python documentation on the official Python website at <http://www.python.org> for more information about Python expressions.

Macros:

3.6 Using Macros in Input Files

A macro is a mechanism to abstract complex input files sequences into (parameterized) tokens. In its simplest form, a macro provides a way to substitute a code snippet from an input file:

```
<macro snippet>
  line1
  line2
  line3
</macro>
```

In this example, every occurrence of the code named snippet in the input file will now be replaced by the three lines defined between the '`<macro>`' and '`</macro>`' tags.

For example, you could define a macro to set up a laser pulse like this:

```
<macro myLaser>
  yeeEmField
  <BoundaryCondition LaserPulseBC>
  ... some regular boundary conditions ...
</BoundaryCondition>
</macro>
```

You could then call your myLaser macro within the input file like this:

```
<EmField exampleField>
  kind = myLaser
</EmField>
```

VORPAL will expand the input file use of your macro into:

```
<EmField exampleField>
  kind = yeeEmField
  <BoundaryCondition LaserPulseBC>
  ... some regular boundary conditions ...
</BoundaryCondition>
</EmField>
```

3.7 Macro Parameters

Macros can take parameters, allowing variables to be passed into and used by the macro. Parameters are listed in parentheses after the macro name in the macro declaration, as in this example:

```
<macro box(lx, ly, lz, ux, uy, uz)>
  lowerBounds = [lx ly lz]
  upperBounds = [ux uy uz]
</macro>
```

Once a macro is defined, it can be used by calling it and providing values or symbols for the parameters. The macro will substitute the parameter values into the body provided. Calling the example above with parameters defined like this:

```
$ NX = 10
$ NY = 20
$ NZ = 30
box(0, 0, 0, NX, NY/2, NZ)
```

will create the following code fragment in the input file:

```
lowerBounds = [0 0 0]
upperBounds = [10 10 30]
```

Note: The parameter substitution happened in the scope of the caller. Parameters do not have scope outside of the macro in which they are defined.

3.8 Macro Overloading

As with symbols, macros can be overloaded within a scope. The particular instance of a macro that is used is determined by the number of parameters provided at the time of instantiation. This enables the user to write macros with different levels of parameterization:

```
<macro circle(x0, y0, r)>
    r^2 - ((x-x0)^2 + (y-y0)^2)
</macro>
<macro circle(r)>
    circle(0, 0, r)
</macro>
```

Looking in the example above, whenever the macro `circle` is used with a single parameter, it creates a circle around the origin; if you use the macro with 3 parameters, you can specify the center of the circle.

The macro substitution does not occur until the macro instantiation is actually made. This means that you do not have to define the 3-parameter `circle` prior to defining the 1-parameter `circle`, even though the 1-parameter `circle` refers to the 3-parameter `circle`. It is only necessary that the first time the 1-parameter `circle` is instantiated, that 3-parameter `circle` has already been defined, otherwise you will receive an error.

3.9 Defining Functions Using Macros

Macros can be particularly useful for defining complex mathematical expressions, such as defining functions in `STFunc` blocks with `'kind = expression'`.

Consider a macro that should simplify the setup of a Gaussian. One could define the following macro:

```
<macro badGauss(A, x, sigma)>
    A * exp(-x^2/sigma)
</macro>
```

While this is a legitimate macro, an instantiation of the macro via:

```
badGauss(A0+5, x-3, 2*sigma)
```

will result in:

```
A0+5*exp(-x+3^3/2*sigma)
```

which is probably not the expected result. One alternative is to put parentheses around the parameters whenever they are used in the macro.

```
<macro betterGauss(A, x, sigma)>
    ((A) * exp(-(x)^2/(sigma)))
</macro>
```

This will ensure that the expressions in parameters will not cause any unexpected side effects. The downside of this approach, however, is that the macro text is hard to read due to all the parentheses. To overcome this issue, `txpp` provides a mechanism to automatically introduce the parentheses around arguments by using a function block

```
<function goodGauss(A, x, sigma)>
    A * exp(-x^2/sigma)
</function>
```

The previous example will produce the same output as the `badGauss` macro, but without requiring the additional parentheses in the macro text.

3.10 More About Parameters

In the previous examples, parameters were always single tokens or simple expressions. However, the pre-parser allows you to pass parameters that span multiple lines. This can be particularly useful for writing larger macros. An example of multiple line parameter passing would be defining a general particle source. This example below shows a macro defining a general species:

```
<macro ions(name,charge,extra)>
  <Species name>
  kind = relBoris
  emField = emSum
  charge = charge extra
  <ParticleSink leftAbsorber>
    kind = absorber
    lowerBounds = [-1 -1 -1]
    upperBounds = [ 0 NY1 NZ1]
  </ParticleSink>
</Species>
</macro>
```

The parameter extra can be an arbitrary string such as:

```
ions(species1, 1.6e-19, ''mass = 1e-28'')
```

or it can be an empty string, if no additional information is needed:

```
ions(species2, 1.6e-19, ''')
```

In addition, you can add entire input file blocks to this parameter. Assume we have a macro called loader, defined as follows:

```
<macro loader(ionDens)>
  <ParticleSource ptcl_loader>
    kind = randDensSrc
    lowerBounds = [ -0.05 -0.05 -0.2]
    upperBounds = [ 0.05 0.05 0.2]
    density = ionDens
    vbar = [0. 0. 0.]
    vsig = [V_ion_rms V_ion_rms V_ion_rms ]
    <STFunc loadProb>
      kind = expression
      expression = H(0.1 - sqrt(x*x + y*y))
    </STFunc>
  </ParticleSource>
</macro>
```

Using this macro with the ions macro defined previously, we can now create an ion species with a source via a single line:

```
ions(species3, 1.6e-19, loader(1e18))
```

3.11 Importing Files

Txpp allows input files to be split into individual files, thus enabling macros to be encapsulated into separate libraries. For example, physical constant definitions or commonly-used geometry setups can be stored in files that can then be used by many VORPAL simulations. Input files can be nested to arbitrary depth.

Files are imported via the import keyword:

```
$ import FILENAME
```

where FILENAME represents the name of the file to be included. txpp applies the standard rules for token substitution to any tokens after the import token. Quotes around the filename are optional and, computed filenames are possible.

3.12 Flow Control and Repetition

The VORPAL pre-parser includes both flow control and conditional statements, similar to other scripting languages. These features allow the user a great deal of flexibility when creating input files.

3.13 Conditionals

A conditional takes either the form:

```
$ if COND
    ...
$ endif
```

or

```
$ if COND
    ...
$ else
    ...
$ endif
```

Conditionals can be arbitrarily nested. All the tokens following the 'if' token are interpreted following the expression evaluation procedure (see above) and if they evaluate to true, the text following the if statement is inserted into the output. If the conditional statement evaluates to false, the text after the 'else' is inserted (if present). Note that 'true' and 'false' in pre-parser macros are evaluated by Python – in addition to evaluating conditional statements such as 'x == 1', other tokens can be evaluated. The most common use of this is using '0' for false and '1' for true. Empty strings are also evaluated to false. For more detailed information, consult the Python documentation.

3.13.1 Example Conditional Statements

```
$ if NDIM == 2
$   dt = 1/(c * sqrt(1/dx^2+1/dy^2))
$ else
$   dt = 1/(c * sqrt(1/dx^2 + 1/dy^2 + 1/dz^2))
$ endif
```

3.14 Repetition

For repeated execution, txpp provides while loops; these take the form:

```
$ while COND
    .
    .
```

```

.
$ endwhile

```

which repeatedly inserts the loop body into the output. For example, to create 10 stacked circles using the circle macro from above, you could use:

```

$ n = 10
$ while n > 0 circle(n)
$   n = n - 1
$ endwhile

```

3.15 Recursion

Macros can be called recursively. E.g. the following computes the Fibonacci numbers:

```

<macro fib(a)>
  $ if a < 2
    a
  $ else
    fib(a-1)+fib(a-2)
  $ endif
</macro>
fib(7)

```

Note: There is nothing preventing you from creating infinitely recursive macros; if terminal conditions are not given for the recursion, infinite loops can occur.

3.16 Symbol Definition on the Command Line

txpp allows symbols to be defined on the command line. These definitions override any symbol definitions in the outer-most (global) scope. This allows you to set a default value inside an input file that can then be overridden on the command line if needed.

For example, if the following is in the outermost scope of the input file (outside of any blocks or macros):

```

$ X = 3
X

```

Then this will result in a line containing '3' in the output. However, if you were to invoke txpp via:

```
txpp.py -DX=4
```

then this will result in a line '4'.

However, if you were to define 'X' inside a block (not in the global scope), such as:

```

<block foo>
  $ X = 3
  X
</block>

```

then 'X' will always be '3', no matter what value for 'X' is specified on the command line.

3.17 Requires

When writing reusable macros, best practices compel macro authors to help ensure that the user can be prevented from making obvious mistakes. One such mechanism is the `requires` directive, which terminates translation if one or more symbols are not defined at the time. This allows users to write macros that depend on symbols that are not passed as parameters. For example, the following code snippet will not be processed if the symbol 'NDIM' has not been previously defined:

```
<macro circle(r)>
  $requires NDIM
  $if NDIM == 2 r^2 - x^2 - y^2
  $endif
  $if NDIM == 3 r^2 - x^2 - y^2 - z^2
  $endif
</macro>
```

3.18 String Concatenation

One task that is encountered often when creating VORPAL simulations is naming groups of similar blocks, e.g. similar species. Macros can allow us to concatenate strings to make this process more clean and simple. However, based on the white-spacing rules, strings will always be concatenated with a space between them. For example,

```
$a = hello
$b = world
a b
will result in
hello world
```

However, we can get around this rule to get the desired output with the following:

```
<macro concat(a, b)>
  $ tmp = 'a tmp b'
</macro>
```

Now when calling

```
concat(hello, world)
```

the result will be:

```
helloworld
```

The first line appends a single quote to 'a' and stores the result in 'tmp'. The next line then puts the token 'a' together with the token 'b'. As they are now no longer two strings; they will be concatenated without a space. The final evaluation of the resulting string then removes the quotes around it, resulting in the desired output.

VORPAL Pre-Defined Macros:

3.19 VORPAL Macros

VORPAL contains a number of pre-defined macros that are used throughout the example input files available through the VorpCom interface. You may find the VORPAL macros to be helpful in your own simulations. The VORPAL macros, including those listed in the VORPAL Reference manual, are accessible through the Macros tab in VorpCom as depicted in the illustration below.

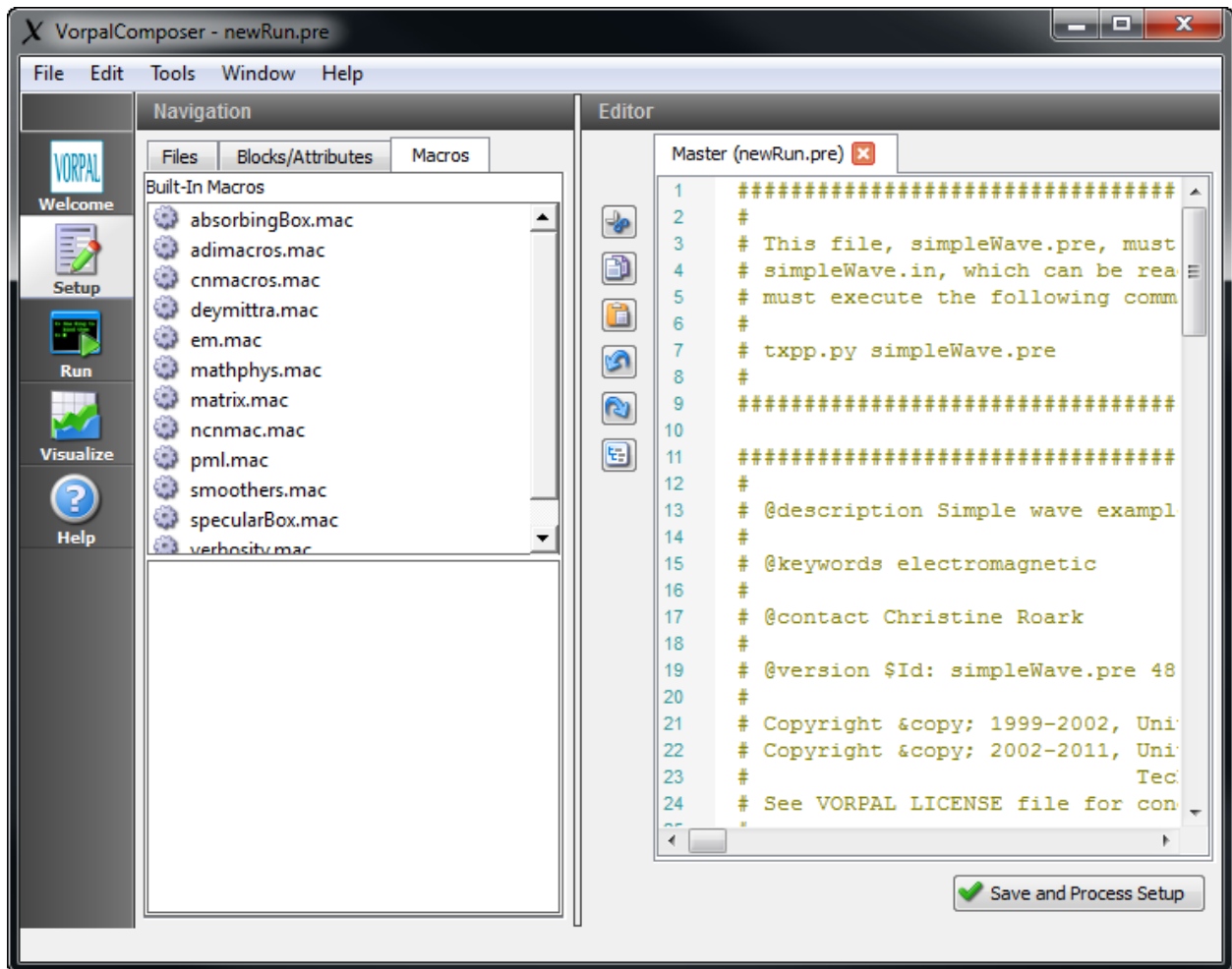


Figure 3.1: VorpComposer Macro Tab

Detailed descriptions of the macros and their parameters are available in the *VORPAL Reference Manual*.

3.20 adimacros.mac Macros

adimacros: Alternating Direction Implicit (ADI) macros are used with the VORPAL implicit Solver. The `adimacros.mac` macros are for use with the VORPAL Multifield feature. Macros from `adimacros.mac` that VORPAL users may find helpful include:

```
:option `adiUpdaters`:  
    for full ADI update.  
  
:option: `dmAdiUpdaters`:  
    for full ADI update with domains requiring GridBoundary.
```

3.20.1 adiUpdaters

adiUpdaters (name, efld, bfld): defines all of the operators for the full ADI update. The `adiUpdaters` macro is located in the file `adimacros.mac`.

Names start with 'p' or 'm' corresponding to P matrix and M matrix, respectively. Names end with the direction of the ADI. `mult` or `solv` refers to application or inversion of the matrix. The divergence preserving update applies these in the order:

- `msolv`
- `pmult`
- (add in current)
- `psolv`
- `mmult`

The curl steady state update applies these in the order:

- `mmult`
- `pmult`
- (add in current)
- `psolv`
- `msolv`

`adiUpdaters` macro parameters include:

- `name`: name assigned to the collection.
- `efld`: electric field
- `bfld`: magnetic field

3.20.2 dmAdiUpdaters

dmAdiUpdaters (name, efld, bfld, bndry, inttype): defines all of the operators for the full ADI update for domains requiring `GridBoundaries`. This allows users to simulation EM with implicit methods for domains that are circles, spheres, or cavities, for example. The `dmAdiUpdaters` macro is located in the file `adimacros.mac`.

Names start with 'p' or 'm' corresponding to P matrix or M matrix, respectively. Names end with the direction of the ADI. mult or solv refers to application or inversion of the matrix. The divergence preserving update applies these in the order:

- msolv
- pmult
- (add in current)
- psolv
- mmult

The curl steady state update applies these in the order:

- mmult
- pmult
- (add in current)
- psolv
- msolv

dmAdiUpdaters macro parameters include:

- name: name assigned to the collection.
- efld: electric field.
- bfld: magnetic field.
- bndry: name of the grid boundary.
- inttype: the interiorness type; this should usually be deymittra.

3.21 applyPML Macro

applyPML (elecfield, magfield, nufieldname, nxl, nyl, nzl, nxu, nyu, nzu, pmlidir, pmlsign, startval, delta, namebn, nameen, namein) applies a PML damping to the given region. The applyPML macro is found in the file pml.mac. You must use update step names that interleave properly with the Yee updater you are using. For the Yee macro in the em.mac file, the updates are:

- 'step1.0': first half B update
- 'step2.0': full E update
- 'step3.0': the second half B update

Therefore,

- 'step0.5'
- 'step1.5'
- 'step2.5'
- 'step3.5'

are good choices for the four update names. To ensure that the steps occur in the correct order, you must add the following line to the Multifield block:

updateStepOrder = [step0.5 step1.0 step1.5 step2.0 step2.5 step3.0 step3.5]

3.21.1 applyPML Macro Parameters

elecfield: name of the electric field.

magfield: name of the magnetic field.

nufieldname: name of the field to store nu profile.

nxl: lower bound in x.

nyl: lower bound in y.

nzl: lower bound in z.

nxu: upper bound in x.

nyu: upper bound in y.

nzu: upper bound in z.

pmdir: one of 0, 1, or 2 for the x, y, or z direction of the PML, respectively.

pmlsign: sign of the outgoing direction.

startval: any offset of the grid start position in the direction of the PML (XSTART, YSTART, ZSTART).

delta: the grid size in the PML direction.

namebn: name for the bfield numerator damping update.

nameen: name for the efield numerator damping update.

nameed: name for the efield denominator damping update.

namebd: name for the bfield denominator damping update.

3.22 deymittra.mac Macros

deymittra.mac: useful macros from deymittra.mac include:

dMYee: for defining fields, updaters, and steps for the Dey-Mittra cut cell method on a Yee mesh (not to be used with implicit solvers).

deyMittraYee: for defining fields, updaters, and steps for the Dey-Mittra cut cell method on a Yee mesh using constrained fields in the conductors.

3.23 deyMittraYeeInterpol Macro

deyMittraYeeInterpol (elecfield, magfield, rhojfield, gridbdry, nx, ny, nz): defines the fields, updaters and update steps for the Dey-Mittra cut cell method on a Yee mesh, including the new constrained fields in the conductors needed for improved interpolation. The dMittraYeeInterpol macro is located in the file `deymittra.mac`.

Since this macro defines updates, be aware of the order in case boundary updaters are to be added:

'step1': first half B update
 'step2': full E update
 'step2.5': constrained field updater
 'step3': second half B update

3.23.1 deyMittraYeelInterpol Macro Parameters

elecfield: name of the electric field.
magfield: name of the magnetic field.
rhojfield: name of the charge density and current field.
gridbdry: name of the grid boundary.
nx: number of cells in x direction.
ny: number of cells in y direction.
nz: number of cells in z direction.

3.24 dmAdiUpdaters Macro

dmAdiUpdaters (name, efld, bfld, bdry, inttype): defines all of the operators for the full ADI update. The dmAdiUpdaters macro is located in the file `adimacros.mac`.

Names start with 'p' or 'm' corresponding to P matrix or M matrix, respectively. Names end with the direction of the ADI. `mult` or `solv` refers to application or inversion of the matrix. The divergence preserving update applies these in the order:

- `msolv`
- `pmult`
- (add in current)
- `psolv`
- `mmult`

The curl steady state update applies these in the order:

- `mmult`
- `pmult`
- (add in current)
- `psolv`
- `msolv`

3.24.1 dmAdiUpdaters macro parameters

name: name assigned to the collection.
efld: electric field.
bfld: magnetic field.

bdry: name of the grid boundary.

inttype: the interiorness type; this should usually be `deymittra`.

3.25 dmYee Macro

dmYee (***elecfield***, ***magfield***, ***gridbdry***, ***nx***, ***ny***, ***nz***): defines the fields, updaters and update steps for the Dey-Mittra cut cell method on a Yee mesh. This macro also sets up interpolation to use the Yee field (edge for E and face for B). The dmYee macro is located in the file `deymittra.mac`.

Since this macro defines updates, be aware of the order in case boundary updaters are to be added:

- ‘`step1.0`’: the first half B update.
- ‘`step2.0`’: the full E update
- ‘`step3.0`’: the second half B update

The dmYee macro is compatible with the **applyPML macro** from `pml.mac`.

3.25.1 dmYee Macro Parameters

elecfield: name of the electric field.

magfield: name of the magnetic field.

gridbdry: name of the grid boundary.

nx: number of cells in x direction.

ny: number of cells in y direction.

nz: number of cells in z direction.

3.26 Yee Macro

Yee (***elecfield***, ***magfield***, ***nx***, ***ny***, ***nz***): defines the fields, updaters and update steps for the regular update method on a Yee mesh. This macro also sets up interpolation to use the Yee field (edge for E and face for B). The Yee macro is located in the file `em.mac`.

Since this macro defines updates, be aware of the order in case boundary updaters are to be added:

‘`step1.0`’: first half B update.

‘`step2.0`’: full E update.

‘`step3.0`’: second half B update.

‘`step4.1`’: nodal E update.

‘`step4.2`’: nodal B update.

‘`init_step1.1`’: nodal E restore.

‘`init_step1.2`’: nodal B restore.

3.26.1 Yee Macro Parameters

- elecfield:** name of the electric field.
- magfield:** name of the magnetic field.
- nx:** number of cells in x direction.
- ny:** number of cells in y direction.
- nz:** number of cells in z direction.

3.27 Geometry Macro

3.27.1 Introduction

A very powerful tool exists in VORPAL to create complex shapes for simulation. Simply import the geometry.mac file and a large number of primitive shapes are available to combine into complex shapes. This file also contains macros to import .stl and python-defined shapes.

3.27.2 Filling/Voiding Introduction

Begin forming the geometry macro by using the default that it is void, and then filling it with the shapes specified. To insure a void starting environment use the command, `resetGeoToVoid()`.

Alternately you can begin forming the geometry by filling the entire region with material and then removing the shapes specified. The filled starting environment is created with `resetGeoToFill(universe)`. Note that a name, in this case *universe* must be given for a filled environment.

Filling and Voiding operations act on the previously defined operations. This means that the order of shape filling and voiding is important. This is expanded upon further in Filling/Voiding advanced and demonstrated in the ridged-Waveguide macro of Geometry_Demo.pre

3.27.3 Grid Boundaries

To finalize the geometry as a grid boundary the command,

```
saveGeoToGridBoundary (gridBoundaryName, DMFRAC)
```

must be used. If a reusable complex shape is desired, the shape may be constructed as a macro, with the argument of the macro listed as the shape name. This is shown below, and in the geometry_demo file.

```
<macro ridgedWaveguide ()>
voidGeoExpression(waveguideBox, geoBoxP(x,y,z,.5,.4,.15))
fillGeoExpression(ridge_2, geoBoxP(x-.25,y,z,.1,.15,.15))
fillGeoExpression(ridge_1, geoBoxP(x-.25,y-.25,z,.1,.15,.15))
fillGeoExpression(ridge_5, geoBoxP(x,y-.25,z,.1,.15,.15))
fillGeoExpression(ridge_6, geoBoxP(x,y,z,.1,.15,.15))
# Note the exact same results could be created by interchanging all fill and void commands
</macro>
ridgedWaveguide ()
saveGeoToGridBoundary(waveguide, DMFRAC)
```

Figure 3.2: Geometry macro function calls

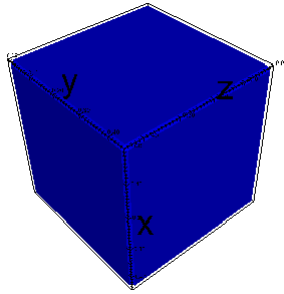


Figure 3.3: geoBoxP (x,y,z,LXO,LYO,LZO)

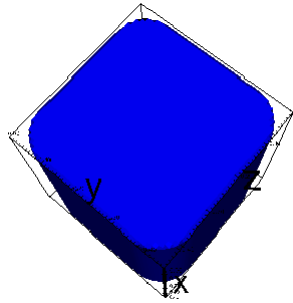


Figure 3.4: geoRndRectangleSlabXP (x,y,z,LZO,LYO,LZO,RND_RADIUS)

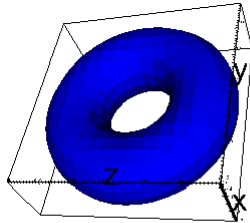


Figure 3.5: geoTorusX (x,y,z,INNER_RADIUS,RADIUS)

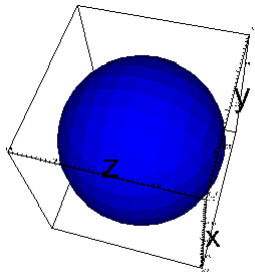


Figure 3.6: geoSphere (x,y,z,RADIUS)

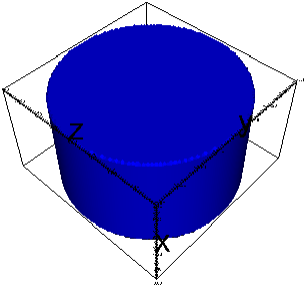


Figure 3.7: geoCylinderXP (x,y,z,RADIUS,LXO)

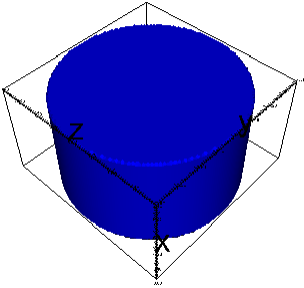


Figure 3.8: geoPipeXP (x,y,z,INNER_RADIUS,RADIUS,LXO)

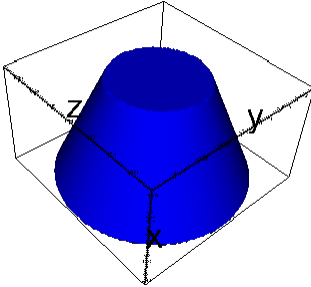


Figure 3.9: geoConeXP (x,y,z,INNER_RADIUS,RADIUS,LXO)

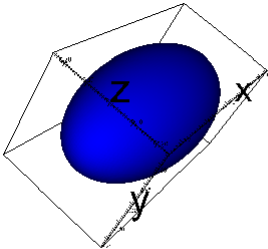


Figure 3.10: geoEllipsoid (x,y,z,A,B,C)

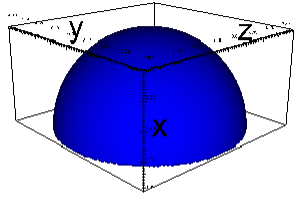


Figure 3.11: `geoHemiSphereXP (x,y,z,RADIUS)`

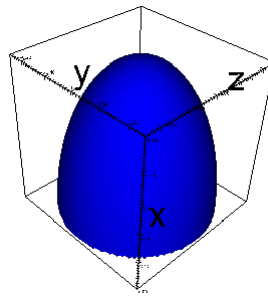


Figure 3.12: `geoHemiEllipsoidXP (x,y,z,A,B,C)`

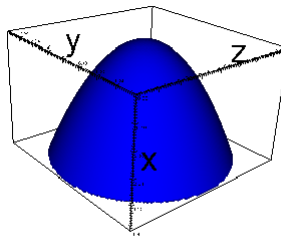


Figure 3.13: `geoParaboloidXP (x,y,z,xVertex,RADIUS)`

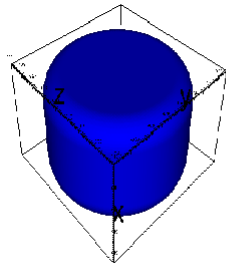


Figure 3.14: `geoRndCylinderXP (x,y,z,RADIUS,LXO,RND_RADIUS)`

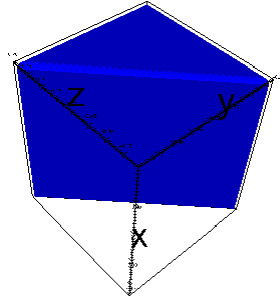


Figure 3.15: geoTriangleSlabXP ($x,y,z,LXO,ya,za,yb,zb,yc,zc$)

3.27.4 Shape Primitives

The geometry macro contains several built in shapes. These are as follows

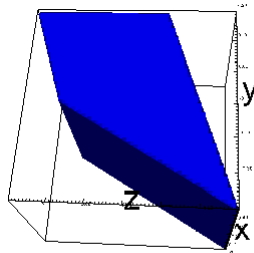


Figure 3.16: geoQuadrilateralSlabXP ($x,y,z,LXO,ya,za,yb,zb,yc,zc,yd,zd$)

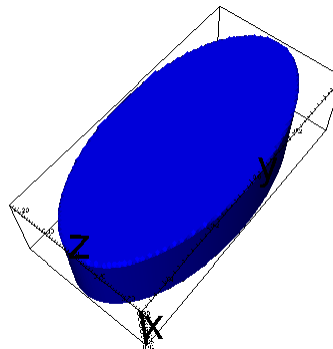


Figure 3.17: geoEllipticalCylinderXP (x,y,z,A,B,LXO)

The accompanying example [geometry demo](#) showcases each of these shapes, and what must be specified to create each one. In general, shapes in the geometry macro are created in the Y-Z plane, and then extruded through the X axis. For example a cylinder is circular in the Y-Z plane, and has its length in the X direction.

Note that in the demonstration file common variable names are used for each shape to show how the parameters relate to one another.

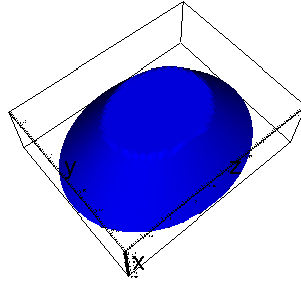


Figure 3.18: `geoEllipticalConeXP (x,y,z,A,B,lowerScale,upperScale,LXO)`

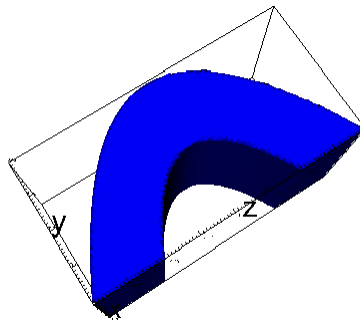


Figure 3.19: `geoBiParabolicSlabXP (x,y,z,LXO,yVertexIn,yVertexOut,halfHeightZIn,halfHeightZOut)`

3.27.5 Moving Shapes

In general, the primitive shapes place points of rotation symmetry, such as the center of a circle at the origin. Rectilinear shapes begin at the origin and extend in the X-direction.

To move a shape in the X,Y and Z direction, the arguments of the function must be adjusted. For example to move a box 3 meters in the positive X direction the input would be `geoBox(x-3,y,z,5,5,5)`. It would remain 5 meters long in the X direction. Adding instead of subtracting will cause a shift in the negative X direction.

3.27.6 Rotating Shapes

It is common to need to rotate a shape's orientation. This is most common in shapes such as cylinders, or tori. All shapes in the geometry macro have their axial direction in the X axis. To set the Y axis as the axial direction, simply exchange the variables Y and X. For example `geoCylinder(y,x,z,.3,.8)`

3.27.7 Advanced Filling and Voiding

By using a sequence of fills and voids, complex shapes can be quickly created and tested. Here we illustrate this with the ridged waveguide. We start with a filled environment. Next we void out a rectangular waveguide, and then using fills to create the ridges. This is demonstrated in the example file.

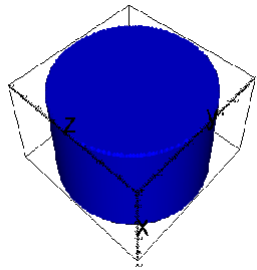


Figure 3.20: Default Rotation Cylinder

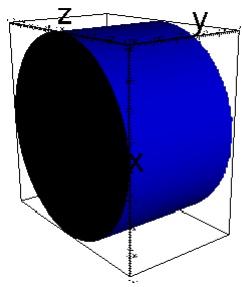


Figure 3.21: Rotated Cylinder

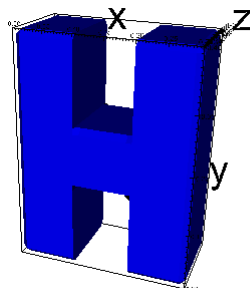


Figure 3.22: Ridged Wavguide Made with Filling and Voiding

3.27.8 Importing Objects From STL Files

STL files are a common format that many CAD programs can export to. It is possible to use the geometry macro to input shapes from a .stl file directly. This is done in the same manner as creating a shape, however a different command is used within the macro.

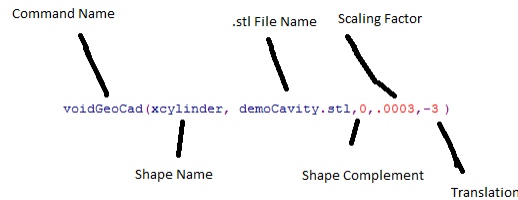


Figure 3.23: Geometry macro call to import STL geometry

The command name shows that a void will be created from an imported cad file. The shape name can be used later in the program to refer to this specific piece. The .stl file must be in the same folder as the .pre and .mac files. The shape complement is 0 (default) to refer to the inside of the STL shape, and 1 to refer to the outside of the STL shape. The scaling factor can be used to adjust for units, for example centimeters to meters. The translation factor will move the shape. CAD shapes can be used together with primitive shapes.

3.27.9 Importing a Python File

The geometry macro contains the ability to import a python-defined shape into the VORPAL simulation environment. This is done with the command shown below.

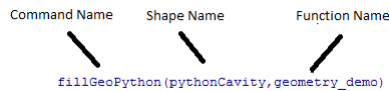


Figure 3.24: Geometry macro call to import python based geometry

The python file itself must have the same name as the PRE file name, and contain the specified function. The function should evaluate to unity inside the shape and zero outside the shape.

3.27.10 Shape Primitive Creation

Sometimes it may be desired to create a new shape primitive. The basis of all shape primitives is the Heaviside function.

In the demonstration file, the function `hollowSphere` specifies a sphere in which the sphere only extends between a specified inner and outer radius. The macro `shapeCreation` then calls and creates this shape.

3.27.11 Tips/Tricks

When building complex shapes it tends to be easier to initially center the shapes around the origin, and then move them into place. This way mirroring or rotating a shape around any of the axis is easy and understandable. It is also recommended when starting out to create a large domain to create shapes in. This way if an error in the translation or size is made it is possible that it will still appear in the domain, making the error easier to spot. Increasing the number of cells helps sharpen corners and smooths rounded shapes.

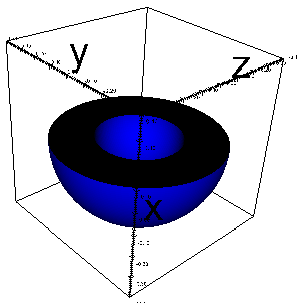


Figure 3.25: myHollowSphere (x,y,z,rInner,rOuter)

3.28 mathphys

mathphys: In many of the input file samples that are supplied with VORPAL in the VorpComposer examples you will see macros from `mathphys.mac` invoked. Macros available in `mathphys.mac` define a series of physical and mathematical constants that are commonly used in VORPAL simulations. Refer to the `mathphys.mac` file under ‘Macros’ in VorpComposer to see which constants are available.

SIMULATIONS

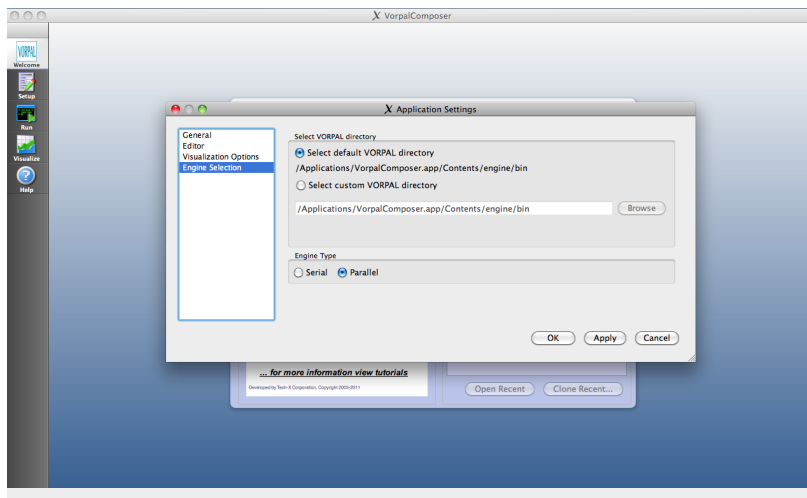
VORPAL in depth:

4.1 Running VorpCom in Parallel

VorpCom defaults to running your simulations in serial. If you are running on a local system with multiple cores, you can run your simulation in parallel as multiple processes.

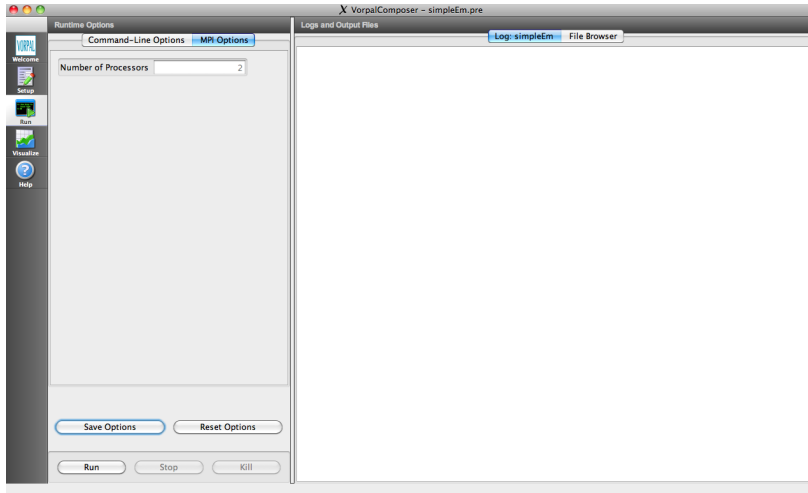
4.1.1 Setting the Engine

In order to run in parallel, you must first change the engine that VorpCom is using. To do this, go to *VorpCom* -> *Preferences* on Mac OS X, or on Linux/Windows go to *Tools* -> *Settings*. This will bring up a pop-up menu. Choose *Engine Selection* from the left menu, and under *Engine Type* switch to **parallel**.



4.1.2 Defining the Number of Processors

Once you have saved and processed the file from the *Setup* window, switch to the *Run* window to define the number of processes to run. In the upper- right portion of the *Runtime Options* pane, there is a tab for the *MPI Options*. Here you can define the number to as few or many processes you want to run.



Now change any command line options as desired or run as usual by pressing the *Run* button.

4.2 Running VORPAL from the Command Line

There are 2 main steps necessary when running from the command line.

- Pre-processing the input file. VORPAL takes a file with the extension ‘.in’ as input. When using the command line invocation of VORPAL, you must first run the Python pre-processor, txpp.py, on your ‘.pre’ input file to produce the ‘.in’ file.
- Running the VORPAL executable. Both the serial and parallel VORPAL executables require you to specify the input file (‘.in’) as well as any optional arguments.

4.2.1 PATH Definitions

The following definitions will be used for the remainder of this section.

On Mac:

```
<VORPAL_BIN_DIR>=/Applications/VorpComposer.app/Contents/engine/bin
<VORPAL_LIB_DIR>=/Applications/VorpComposer.app/Contents/engine/lib
<VORPAL_SHARE_DIR>=/Applications/VorpComposer.app/Contents/engine/share
```

On Linux (assuming you have chosen /usr/local/VorpComposer as your default installation directory):

```
<VORPAL_BIN_DIR>=/usr/local/VorpComposer/Contents/engine/bin
<VORPAL_LIB_DIR>=/usr/local/VorpComposer/Contents/engine/lib
<VORPAL_SHARE_DIR>=/usr/local/VorpComposer/Contents/engine/share
```

On Windows (e.g. 64-bit)

```
<VORPAL_BIN_DIR>=C:\Program Files\Tech-X (Win64)\VorpComposer 5.0\Contents\engine\bin
<VORPAL_SHARE_DIR>=C:\Program Files\Tech-X (Win64)\VorpComposer 5.0\Contents\engine\share
```

4.2.2 Process the .pre Input File into a .in Input File

To run the pre-processor, the Python executable must be available from your command prompt or shell path. The PYTHONPATH must also be declared.

If you want to use any macros that are not already included in your `.pre` file, then you must use the `--import` option, otherwise you can skip this option.

Use the `--outfile` option to define the name of the file to be produced by the pre-processor, which then becomes the input file formatted for use by VORPAL. Therefore, by convention the `.in` input file for VORPAL has the same filename as the `.pre` input file to the pre-processor, but with the suffix of `.in`.

Pre-Processing Steps

1. Change (`cd`) to the directory to where the `.pre` file is located before invoking **Python**.
2. Issue the command to run the **Python** pre-processor.

The command lines for running the **Python** pre-processor on Windows is:

```
set PYTHONPATH="<VORPAL_BIN_DIR>\Lib\site-packages"
cd <MY_PRE_FILE_DIR>
<VORPAL_BIN_DIR>\python.exe <VORPAL_BIN_DIR>\txpp.py --import=<VORPAL_SHARE_DIR>\macros \
--prefile=inputFile.pre --outfile=inputFile.in
```

The command lines for running the **Python** pre-processor on Linux is:

```
export PYTHONPATH=<VORPAL_LIB_DIR>/python2.6/site-packages
cd <MY_PRE_FILE_DIR>
<VORPAL_BIN_DIR>/python <VORPAL_BIN_DIR>/txpp.py --import=<VORPAL_SHARE_DIR>/macros \
--prefile=inputFile.pre --outfile=inputFile.in
```

On Mac OS X, use the system-installed python (e.g. `/usr/bin/python`). The command lines for running the **Python** pre-processor on Mac OS X is:

```
export PYTHONPATH=<VORPAL_LIB_DIR>/python2.6/site-packages
cd <MY_PRE_FILE_DIR>
python <VORPAL_BIN_DIR>/txpp.py --import=<VORPAL_SHARE_DIR>/macros \
--prefile=inputFile.pre --outfile=inputFile.in
```

4.2.3 Running the VORPAL Executable

Your VORPAL distribution package contains two executable programs for running VORPAL, one for serial computations (`vorpalsr`) and another for parallel computations (`vorpal`). Both versions of the VORPAL executables are located in the `'VORPAL_BIN_DIR'` directory.

Command Line Features

If VORPAL is run from the command line, input file and runtime options are specified as flags.

Order of Parameter Precedence

If a parameter is both set within the input file and specified on the command line, the command line parameter value takes precedence. The command line override enables you to configure an input file with default values while exploring alternative parameter settings from the command line. From the command line, you can quickly change simulation run lengths, dimensionality, output timing, etc.

Examples of Running VORPAL from the Command Line

In these examples, it is assumed that you are either in the directory in which the **vorpals** is installed or you have added the appropriate directory to your shell path.

Command Line Options

To use multiple options, the command line syntax is:

```
./vorpals -i filename [-o prefix_name] [-dim num] [-dt fnum] [-rnum] \
[-sd] [-nd] [-nc] [-oc]
```

in which ‘./vorpals’ is used to run a serial computation. See *Serial Computation* for details about serial computation. See the *Parallel Computation* for details of command line invocation with parallel computation scripts.

Commonly used options that you can specify on the command line include:

-i filename Read input from file named *filename*.

For example:

```
./vorpals -i simpleEs.in
```

-o prefix_name Base names of output files on the text string *prefix_name*.

For example, if you want output files named ‘newSimpleEs’ rather than ‘simpleEs’, use:

```
./vorpals -i simpleEs.in -o newSimpleEs
```

-dim num Run simulation using *num* spatial dimensions. This option overrides the *dimension* parameter specified in the input file.

For example, if you want to run ‘simpleEs’ in 2D rather than 3D, use:

```
./vorpals -i simpleEs.in -dim 2
```

-dt fnum Use time step of size *fnum*. This option overrides the *dt* parameter defined in the .pre file.

For example, if you want to run ‘simpleEs’ with the timestep duration 9.9e-12 seconds, use:

```
./vorpals -i simpleEs.in -dt 9.9e-12
```

-n num Run the simulation for *num* time steps. This option overrides the *nsteps* parameter.

For example, if you want to run ‘simpleEs’ with 50 time steps rather than 10, use:

```
./vorpals -i simpleEs.in -n 50
```

-d num Dump data every *num* time steps. This option overrides the *dumpperiodicity* parameter.

For example, to run ‘simpleEs’ and dump output after every 5 time steps, use:

```
./vorpals -i simpleEs.in -d 5
```

-r num Restart VORPAL from dump *num*.

For example, if you want to restart simpleEsSteps using the output dumped at time step 50, use:

```
./vorpals -i simpleEsSteps.in -r 50
```

-sd Dump data at start of simulation. This option is useful for debugging purposes. It lets you see whether VORPAL used the data you wanted it to use at the start of the simulation.

- nd** Disable data dumping.
- nc** Redirect all of the ‘*_comms_*.txt’ output to /dev/null.
- oc [rank]** Suppress the creation of all comms text files except for that file from the specified [rank] process. This flag is overridden by the **-nc** flag, above.

4.2.4 Serial Computation

The VORPAL executable for use in serial computation is named **vorpalser**. Except as noted, the explanations and tutorials within the *VORPAL User Guide* and *Learning VORPAL by Example* demonstrate VORPAL usage for serial computations. Here is an example of VORPAL command line invocation using an input file named `myfile.in` and specifying 1000 time steps, outputting the result data (dumping) every thousand steps. By default, the output files for this example would be named using the format `myfile.out`.

```
./vorpalser -i myfile.in -n 1000 -d 1000
```

Note: When running VORPAL via VorpComposer, command line options are not directly available, however ‘-i’ and ‘-o’ command line options described in this document are implicit; that is, these options are automatically invoked when running VorpComposer.

4.2.5 Parallel Computation

The VORPAL executable for use in parallel computation is named **vorpal**. This section explains use of the VORPAL executable program for parallel computations.

VORPAL for parallel computations requires the Message Passing Interface (MPI). For information about *MPI* for use with VORPAL, see *Running VORPAL with mpiexec*.

To run parallel computations in VORPAL, you must pre-process your `.pre` input file to produce a `.in` input file that can be submitted to VORPAL for simulation. See directions for processing a `.pre` file and running VORPAL from the command line.

Output Files Generated by Individual Processors during Parallel Computations

In contrast to VORPAL for serial computation, which creates a single output file, VORPAL for parallel computation creates multiple output files. Each individual processor from the parallel run sends comments to a different output file. A parallel computation output file’s name includes a label that identifies the number of the processor that generated that file.

For example, running VORPAL for serial computations to process the `simpleEs.in` file creates a single text file named `simpleEs_comms_0.txt`.

Whereas, running VORPAL for parallel computations on two processors to process the `simpleEs.in` file creates the two text files:

- `simpleEs_comms_0_1.txt`
- `simpleEs_comms_0_2.txt`

in which the final ‘_1’ and ‘_2’ before the file name suffix indicate the number of the processor.

Parallel Computation Scripts

Running VORPAL with **MPI** or Parallel Queuing Systems requires use of different shell scripts to enable invocation of the VORPAL executable as discussed in the following sections.

Running VORPAL with mpiexec

The program 'mpiexec' is used for command-line parallel computations using VORPAL. 'mpiexec' is distributed with VORPAL. VORPAL for parallel computations can be run from the command line:

```
<VORPAL_BIN_DIR>/mpiexec -np <#> <VORPAL_BIN_DIR>/vorpall -i filename
```

in which '<#>' is the number of processors, **vorpall** is the executable program for parallel computations, and 'filename' is the name of the VORPAL input file.

Following 'mpiexec', but before '<VORPAL_BIN_DIR>', you can specify a variety of 'mpiexec' options. For more information about 'mpiexec', including the complete list of options, see a man page or other documentation for 'mpiexec'.

Following **vorpall**, you can specify a variety of VORPAL options. For a list of commonly used options, see *Command Line Features*.

VORPAL automatically adjusts its decomposition to match the number of processors it is given.

Running VORPAL with Parallel Queuing Systems

Parallel queuing systems, such as LoadLeveler and PBS, require the submission of a shell script with embedded comments that the systems interpret. Here is an example of a basic shell script for a PBS-based system:

```
#PBS -N NDS_vorpall
#PBS -l nodes=2:ppn=2
<VORPAL_BIN_DIR>/mpiexec -np 4 <VORPAL_BIN_DIR>/vorpall -i vaclaunch.in -n 250 -d 50
```

Running VORPAL in Parallel under Windows

To run VORPAL in parallel on a Windows system, bring up a DOS window. From the command prompt,

```
<VORPAL_BIN_DIR>\mpiexec.exe -np 2 <VORPAL_BIN_DIR>\vorpall.exe -i input_file
```

4.3 HDF5 Format VORPAL Output Files

Hierarchical Data Format Version 5 (HDF5) is a library and file format for storing graphical and numerical data and for transferring that data between computers. VORPAL outputs data in HDF5 format.

The Hierarchical Data Format was developed by the National Center for Supercomputing Applications at the University of Illinois at Urbana-Champaign. For more information about HDF5, please see the web page at: <http://hdfgroup.org/HDF5>

4.3.1 HDF5 Files

VORPAL outputs data in HDF5 format files that have '.h5' suffixes.

VORPAL produces one HDF5 file for each field or species at each dump time. For example, if the simulation parameter ‘nsteps = 100’ and the simulation parameter ‘dumpPeriodicity = 10’, VORPAL dumps data 10 times during the simulation and outputs a total of 10 HDF5 files for each field or species while running the simulation.

For more information about HDF5, see: <http://hdfgroup.org/HDF5>

4.3.2 Change the Names of Output Files

If you want to change the names of the output files, which include the ‘.h5’ files, you can specify the ‘-o’ output option when you run VORPAL.

For example, you want to replace `simpleEs` with `simpleEsTest1` in the names of the `simpleEs` simulation’s output files. Run VORPAL from the command line using this command:

```
vorpals -i simpleEs.in -o simpleEsTest1
```

The output files will be:

- `simpleEsTest1_all_1.txt`
- `simpleEsTest1_comms_0.txt`
- `simpleEsTest1_completed.txt`
- `simpleEsTest1_dumpedobjs_0.txt`
- `simpleEsTest1_electrons_1.h5`
- `simpleEsTest1_Globals_1.h5`
- `simpleEsTest1_SumRhoJ_1.h5`
- `simpleEsTest1_universe_1.h5`
- `simpleEsTest1_YeeStaticElecField_1.h5`

4.3.3 Display the Content of .h5 Files

The `h5dump` utility converts the binary data in ‘.h5’ files into human-readable ASCII data in ‘.txt’ files.

The `h5dump` utility is available for all the platforms on which VORPAL runs. You can download the utility from: <http://hdfgroup.org/HDF5>

The basic command is:

```
h5dump -o output_file_name.txt your_h5_file.h5
```

To convert the ‘`simpleEs_electrons_1.h5`’ to ASCII text format, use this command:

```
h5dump -o simpleEs_electrons_1.txt simpleEs_electrons_1.h5
```

For information about HDF5 and the `h5dump` utility, see the *HDF5 User’s Guide* and *HDF5 Reference Manual*, which are available at: <http://www.hdfgroup.org/HDF5/doc/index.html>

4.3.4 General Structure of VORPAL Simulation Output .h5 Files

For each type of output file below, main data entries within that output file are displayed as a list of fields at the same level within the list. For those data fields within an output file that contain one or more subcategories of data, subcategories appear in an indented list below the main data category to which the subcategories apply.

Type of Output File: Globals

```
globalGridGlobal
runInfo
time
```

Type of Output File: SumRhoJ

```
SumRhoJ
  int array [NX NY NZ 4]
  SumRhoJ[0] = Rho, charge density
  SumRhoJ[1] = Jx
  SumRhoJ[2] = Jy
  SumRhoJ[3] = Jz
derivedVariables
globalGridGlobal
globalGridGlobalLimits
runInfo
time
```

Type of Output File: comboEmField

```
comboEmField
  int array [NX NY NZ 6]
  comboEmField[0] = Ex
  comboEmField[1] = Ey
  comboEmField[2] = Ez
  comboEmField[3] = Bx
  comboEmField[4] = By
  comboEmField[5] = Bz
globalGridGlobal
globalGridGlobalLimits
runInfo
time
```

Type of Output Files: yeeEmField – YeeElecField

```
YeeElecField
  int array [NX NY NZ 3]
  YeeElecField[0] = Ex
  YeeElecField[1] = Ey
  YeeElecField[2] = Ez
derivedVariables
globalGridGlobal
globalGridGlobalLimits
runInfo
time
```

Type of Output Files: yeeEmField – YeeMagField

```
YeeMagField
  int array [NX NY NZ 3]
  YeeMagField[0] = Bx
  YeeMagField[1] = By
  YeeMagField[2] = Bz
derivedVariables
globalGridGlobal
globalGridGlobalLimits
runInfo
time
```

Type of Output Files: emMultiField – ElecMultiField

```

ElecMultiField
  int array [NX NY NZ 3]
  ElecMultiField[0] = Ex
  ElecMultiField[1] = Ey
  ElecMultiField[2] = Ez
globalGridGlobal
globalGridGlobalLimits
runInfo
time

```

Type of Output Files: emMultiField – MagMultiField

```

MagMultiField
  int array [NX NY NZ 3]
  MagMultiField[0] = Bx
  MagMultiField[1] = By
  MagMultiField[2] = Bz
globalGridGlobal
globalGridGlobalLimits
runInfo
time

```

Type of Output File: GridBoundary name

```

name
  int array [NX NY NZ 2]
  name[0] = true (1) or false (0) is the lower left front corner inside or not?
  name[1] = true (1) or false (0) is the cell center inside or not?
nameLargeBndryFaces
nameLargeFaceFracs
nameSmallBndryFaces
nameSmallFaceFracs
nameStairStepBndryEdges
nameStairStepBndryFaces
globalGridGlobal
globalGridGlobalLimits
poly
runInfo
time

```

Type of Output File: universe

```

globalGridGlobal
globalGridGlobalLimits
poly
runInfo
time
universe
  int array [NX NY NZ 2]

```

Type of Output File: history

```

runInfo
historyName1
historyName2

```

These `:option:'historyName'` arrays contain time data pertaining to the type of history chosen in the input file.

Type of Output File: species

4.3.5 Columns in Particle Simulation .h5 Output Files

Below is a table displaying how the columns in particle simulation ‘.h5’ output files for various species kinds correspond to the columns that can be seen in the ‘.h5’ file when the file is opened using a tool such as **HDFView**. HDFView may be downloaded for free from the HDF Group at <http://www.hdfgroup.org/hdf-java-html/hdfview/>. HDFView distributions are available for 32-bit and 64-bit Linux, Mac, and Windows platforms.

Table 4.1: Legend

NDIM	Number of Dimensions: 1, 2, or 3	
	Dimension Notation	
1D	2D	3D
x	x y	x y z
1D	1D,[2D]	1D,[2D],[3D]]
	Cylindrical Coordinates	
Polar	u0, u1, u2	r, phi, z
Cylindrical	u0, u1, u2	z, r, phi
Tubular	u0, u1, u2	phi, z, r
	Momentum/Mass Notation Convention	
	momentum/mass = gamma*v = P	
gamma*vx:Px	gamma*vy: Py	gamma*vz: Pz

Columns in .h5 in Particle Simulation Output Files

Species	Number of Columns	Comma-separated Columns
cmplxRelBorisDF	5 + NDIM	x,[y,[z]]Px, Py, Pz, real weight, imaginary weight
envBoris	5 + NDIM	x,[y,[z]]Px, Py, Pz, tag, weight
freeRel	3 + NDIM	x,[y,[z]]Px, Py, Pz
freeRelVW	4 + NDIM	x,[y,[z]]Px, Py, Pz, weight
noMove	3 + NDIM	x,[y,[z]]Px, Py, Pz
noMoveVW	4 + NDIM	x,[y,[z]]Px, Py, Pz, weight
nonRelBoris	3 + NDIM	x,[y,[z]]Px, Py, Pz
nonRelBorisCell	3 + NDIM	x,[y,[z]]Px, Py, Pz
nonRelBorisNoDrift	3 + NDIM	x,[y,[z]]Px, Py, Pz
nonRelEsCell	3 + NDIM	x,[y,[z]]Px, Py, Pz
relBoris	3 + NDIM	x,[y,[z]]Px, Py, Pz
relBorisBallisticVW	4 + NDIM	x,[y,[z]]Px, Py, Pz, weight
relBorisCellVW	4 + NDIM	x,[y,[z]]Px, Py, Pz, weight
relBorisCyl	3 + NDIM	u0,[u1,[u2,]]P0, P1, P2 (see Legend)
relBorisDF	4 + NDIM	x,[y,[z]]Px, Py, Pz, weight
relBorisEffMassExtd	2 + 3(NDIM always 3)	x, y, z, Px, Py, Pz, valley index, weight (always 1)
relBorisFuncVW	4 + NDIM	x,[y,[z]]Px, Py, Pz, weight
relBorisTagged	4 + NDIM	x,[y,[z]]Px, Py, Pz, tag
relBorisVW	4 + NDIM	x,[y,[z]]Px, Py, Pz, weight
relBorisVWTagged	5 + NDIM	x,[y,[z]]Px, Py, Pz, tag, weight

4.3.6 HDFView Particle Simulation .h5 Output File Illustration

Below is an **HDFView** display of a VORPAL species ‘.h5’ output file. The annotated columns within the illustrated output file correspond to those presented in the table Columns in ‘.h5’ in Particle Simulation Output Files. HDFView may be downloaded for free from the HDF Group at:

<http://www.hdfgroup.org/hdf-java-html/hdfview/>

	column 0 = x	column 1 = y	column 2 = z	column 3 = Px	column 4 = Py	column 5 = Pz	column 6 = weight
0	-0.0157799	-0.0294280	-0.0067586	-5.5887878	-2.3778187	-5875626.1	1.4450893
1	-0.0098049	-0.0165341	0.0195369	-420.1638.0	-1426265.4	341447.50	6.3760314
2	-0.0087244	-0.0169444	0.0200418	-2.5225648	-4850094.2	6851803.9	1.0894884
3	-0.0118060	-0.0064300	0.0299319	-6.8308721	-7253392.0	9530509.7	2.9598999
4	-0.0182294	0.0166187	-0.0237532	-4.0239392	7473651.6	-1.3012772	7.2752569
5	-0.0119538	0.0158877	0.0196579	-2.4252586	2491561.2	1332087.1	5.0884887
6	-0.0194465	0.0197522	0.0195454	-1.4209478	3490519.1	-2903718.8	1.6339286
7	-0.0191773	0.0288810	-0.0060669	-4.3453048	1.3899456	-3618288.2	5.2850813
8	-0.0190443	0.0256082	0.0044288	-3.2301030	1.0445447	-834138.84	1.7075190
9	-0.0096326	0.0228216	0.0198359	-7.6883733	2.9816529	1.9470096	3.5222526
10	-0.0142185	0.0350007	-0.0083154	-7.6379928	3.9493348	-1.0607862	1.6007987
11	0.0054198	-0.0348406	-0.0347100	-1.0119441	-1.0845906	-9.1778277	1.2742728

This is an example of a .h5 particle simulation output file for the VORPAL species kind reIBorisVW. If you look at the table Columns in .h5 in Particle Simulation Output Files you can see how reIBorisVW maps to the layout of this file.

To understand the layout of any particular .h5 output file for a species, you must know the number of dimensions for the simulation. In this example, a 3D simulation was performed, so the number of columns is 4 + NDim, where the number of dimensions is 3, so there are 7 columns: x, y, z, Px, Py, Pz, weight

Px, Py, Pz = momentum-per-mass = gamma*Velocity
where gamma is the relativistic factor

Figure 4.1: HDFView display of a VORPAL species .h5 output file.

4.4 Advanced Topics in Electrostatic Simulation

Advanced electrostatic simulation topics include information that is not introduced in the *es-tutorial-intro*.

4.4.1 Electrostatic Solvers and Pre-Conditioners

Tech-X offers the following advice on choosing an electrostatic solver, tolerance, and a pre-conditioner.

4.4.2 Choosing an Electrostatic Solver

Due to the way the boundary conditions are implemented in the electrostatic solver, the matrix in the resulting linear system is non-symmetric (except for the fully periodic case). Therefore, you should select a solver that can deal with non-symmetric matrices. The multi-grid pre-conditioned ‘bicgstab’ solver tends to yield good convergence.

When choosing a solver for periodic problems, the key is to choose a solver that uses at least one boundary condition to set the potential (define ground). Alternatively, you can use a multigrid preconditioner with the ‘bicgstab’ solver.

4.4.3 Choosing the Tolerance

Many of the parameters in the electrostatic solver are highly problem dependent. The tolerance, which is a measure of the residual reduction compared to the initial residual, is the stopping criterion for the iterative solver. Values of 10^{-5} often are sufficient for giving meaningful results. If the solver does not converge, increase the tolerance. If the resulting potentials miss a lot of small scale structure, reduce the tolerance.

4.4.4 Choosing a Pre-Conditioner

Pre-conditioners transform the linear system used in the Poisson solver into systems with more favorable convergence behavior. For a simple fully periodic system, a pre-conditioner may not be necessary. For most other cases, using a

pre-conditioner significantly improves the convergence behavior. Multi-grid pre-conditioning tends to yield the best convergence behavior. (You can see an example of multi-grid pre-conditioning in *es-tutorial-lesson-5-modify-the-electrostatic-solver*.)

For a comparison of different pre-conditioners and solvers that you can use for your electrostatic simulations, see P. Messmer and D.L. Bruhwiler, **A parallel electrostatic solver for the VORPAL code**, *Computer Physics Communications*, vol. 164, pp. 118-121, 2004.

4.4.5 Putting Mathematical Expressions in a Pre File

You can put mathematical expressions directly in a pre file's input blocks by encapsulating them between dollar signs '\$ \$'. When you convert the pre file to an input file, `txpp.py` evaluates the expression within the dollar signs.

In the `simpleEs.pre` file, one of the variable definitions is: `'$NXM1 = NX - 1'`, and several boundary condition input blocks use the 'NXM1' variable in the vector defining the lower bounds. For example: `'[NXM1 0 0]'`. You can use a mathematical expression in place of the 'NXM1' variable. For example, you could define the vector as follows: `'[$ NX -1$ 0 0]'`.

Multiplying NX by 2 would also work: as shown here: `'[$2 * NX - 1$ 0 0]'`.

4.4.6 Saving the Potential to a File

By default, VORPAL does not write out the potential to an '.h5' file. In 1D and 2D, you can save the potential to the array typically reserved for the z-component of the electric field by setting the electrostatic solver parameter:

```
'dumpPotential = true'
```

This feature is for convenient diagnosis and debugging; it has the side effect of giving any particles unphysical values for the z component of velocity. In the presence of a magnetic field with an x or y component, this unphysical velocity can couple back into a 2D simulation.

Note: Production runs should not use `'dumpPotential = true'`.

4.4.7 Symmetry Boundaries

A Neumann boundary can act as a symmetry boundary. For instance, placing Neumann boundaries appropriately on three faces of a cube will turn the simulation into a symmetric simulation of 1/8th of a larger cube.

4.4.8 Fully Periodic Systems

You can model a system that is periodic in all directions, that is, which has no boundaries. In a fully periodic system, however, the overall charge must be zero as demonstrated in this equation:

$$\int_x^{x+L} \frac{\partial E}{\partial x} = E(x+L) - E(x) = \frac{1}{\epsilon} \int_x^{x+L} \rho dx = \frac{L}{\epsilon} \langle \rho \rangle \quad (4.1)$$

Zero net charge can be enforced by specifying the electrostatic solver parameter `'enforceZeroNetCharge = true'`. To do so, remove the comment symbol '#' in front of `'enforceZeroNetCharge = true'`, which is near the top of the electrostatic solver input block in the `simpleEs.pre` file.

For more information, see Birdsall and Langdon, *Plasma Physics via Computer Simulation*. [1]

4.4.9 Secondary Electrons

To learn how to add secondary electrons, see the example file `secElecs.pre` that is accessible through VorpCom-poser.

4.4.10 Sputtering

To learn how to add sputtering, see the example file `sputter.pre` that is accessible through the VorpCom-poser.

4.5 Advanced Topics in Electromagnetic Simulation

Advanced electromagnetic simulation topics include information that is not introduced in the *em-tutorial-intro*.

4.5.1 Modeling a Time-Dependent or Space-Dependent Value on a Boundary

An electromagnetic boundary condition may use time signals or spatial profiles. For example, the `simpleEm.pre` file's 'xLowerWaveLauncher' boundary condition input block uses a spatial profile: 'kind = planeWavePulse'.

The *function* parameter in a boundary condition input block also may be an expression, and the expression may be any standard function of space and time.

The `simpleWave.pre` file uses the 'kind = expression' technique.

```
<BoundaryCondition xLowerWaveLauncher>
  # Value given by function
  kind = variable
  lowerBounds = [0 0 0] # Lower limits
  upperBounds = [1 NY NZ] # Upper limits
  components = [1] # Ey polarized
  <STFunc function>
    kind = expression
    expression = 1.e6\*sin(4.71e14\*t)
  </STFunc>
</BoundaryCondition>
```

For the complete content of the `simpleWave.pre` file, see *simplewavepre*.

4.5.2 Creating a Circularly Polarized Pulse

The steps to create a circularly polarized pulse are:

1. In the electromagnetic boundary condition input block, change the components parameter vector to launch both a y and z component: 'components = [1 2]'
2. **Give the amplitudes parameter vector a value for each component:** 'amplitudes = [AMPLITUDE AMPLITUDE]'
3. Give the phases parameter a value for each component. To make a circularly polarized pulse, the phases for the two components should be $\pi/2$ radians (90 degrees) apart: 'phases = [0. 1.57]'

The *widths* parameter controls the spatial extent of the wave.

See the example file `coldrelramp.pre` that is accessible through VorpCom-poser.

4.5.3 Launching a Pulse of Arbitrary Temporal Profile

You can use the expression parser for this type of simulation. See the example file `bathtubAntenna.pre` that is accessible through VorpalComposer demonstrates how to use a space-time expression to drive a current.

4.5.4 Launching a Wave from X-Upper Boundary of Simulation

The steps to change a wave launcher from the x-lower to the x-upper boundary follow:

1. Change the lower and upper bounds of the wave launcher boundary condition to the x-upper end of the grid.

Note: The lower bounds of a boundary condition at the x-upper end of the grid are in the physical domain. See the description of the x-upper conductor in *em-tutorial-lesson-1-boundary-conditions-for-electromagnetic-simulations*.

2. Adjust the x-upper conducting boundary to zero out only the y component.
3. Adjust the x-lower conducting boundary to zero out both y and z components. Be sure to change both the amplitudes and components vector.
4. Comment out the *em-tutorial-1-moving-window*. The moving window can move only in a positive direction (to the x-upper end of the simulation, to the y-upper end, or to the z-upper end); it cannot move to a negative direction (to the x-lower end, to the y-lower end, or to the z-lower end).

Without the moving window, the simulation occurs in the lab frame. This means the pulse will strike the x-lower boundary and reflect back after roughly fifty steps for the given time step and simulation size. To avoid reflections, you need to include an absorbing boundary at the x-lower end. For more information, see *Absorbing a Wave*.

4.5.5 Absorbing a Wave

To use an absorbing boundary condition, see the example file `pmlSlab.pre` that is accessible through VorpalComposer.

4.5.6 Simulating a Non-Vacuum

To simulate a non-vacuum, see the example file `phCrystal.pre`, accessible through VorpalComposer, which demonstrates how to use a dielectric.

4.5.7 Additional Types of Electromagnetic Boundary Conditions

In addition to wave launchers and conductors, VORPAL supports other types of electromagnetic boundary conditions. An example is the resistive wall (`'kind = resistiveWall'`). For more information, see the *VORPAL Reference Manual*.

4.5.8 Defining Multiple Electromagnetic Fields

Multiple electromagnetic fields can co-exist in a simulation. For more information, see the *VORPAL Reference Manual*.

4.6 Advanced Topics in Particle Simulation

Advanced particle simulation topics include information that is not introduced in the Beginner Tutorials lessons.

4.6.1 Particle Tracking

Since VORPAL is a particle-in-cell code, the position of a specific particle in the particle data structure will vary as the simulation proceeds due to sorting and the creation and/or deletion of particles. Before you can track a particle, you must identify the particle by tagging it.

Note: To track an individual particle, you must use a kind of particle species that enables tagging of individual particles. Creating a species with the kind parameter set to one of the following kinds enables you to create tagged particles that you can then use to generate particle position data for all tags (indicating up to some maximum tag value) by using History.

- *relBorisTagged*
 - *relBorisVWTagged*
 - *relBorisVWScale*
-

Before particles can be tracked they must be tagged. There are three ways to tag particles.

- By setting the tags manually with the *manualSrc*.
- By using the tagGen **STFunc** in the *funcVelGen* **VelocityGenerator**. See *velocity-generator-details*.
- By using the *fieldScaleVelGen* **VelocityGenerator**. This should only be used with *relBorisVWScale* particles.

If the tags are set manually then the user must insure that the tags are unique or errors will occur tracking the particles.

Example of Using manualSrc to Tag Particles

```
<Species electrons>
  kind = relBorisTagged
  # constants defined previously
  charge = ELECCHARGE mass = ELECMASS
  # Place a single particle in the macroparticle
  emField = constMag numPtclsInMacro = 1.
  #
  # If the manual source is used to create
  # tagged particles, the tag must be a unique
  # integer or the particle tracking will not
  # work.
  #
  <ParticleSource SingleParticleSrc>
    applyPeriod = 0
    kind = manualSrc
    # manual particle loaders always require
    # AT LEAST 3 velocity components regardless
    # of NDIM; in this case, NDIM=2
    p1 = [R_GYRO 0. 0. V_GYRO V_Z 0.]
    p2 = [0. R_GYRO -V_GYRO 0. V_Z 1.]
    p3 = [-R_GYRO 0. 0. -V_GYRO V_Z 2.]
    p4 = [0. -R_GYRO V_GYRO 0. V_Z 3.]
```

```

    </ParticleSource>
</Species>

```

Example of Using tagGen to Tag Particles

```

<VelocityGenerator velGen>
  kind = funcVelGen

  # This sets the tags to be unique integers so they
  # can be tracked by the tagged particle tracking
  # history
  <STFunc component3>
    kind = tagGen
  </STFunc>
</VelocityGenerator>

```

After you have tagged those particles you would like to track using you can record the particles position and/or velocities (or other internal variables such as the weight) by using the *speciesTrackTag* **History**.

Example History Block Used to Record Data from Tagged Particles

```

# Tagged particle trajectory history.
<History trajectory>
  kind = speciesTrackTag
  # Any particle with a tag greater than or equal to
  # the maximum tag will not be tracked.
  maximumTag = 4
  xComponents = all
  species = electrons
</History>

```

4.7 Monte Carlo Interactions Package

Monte Carlo Interactions package: allows the user to model random interactions that involve different objects in the simulation, such as Species, Fluids, and Fields. In a VORPAL simulation, macroparticles of one species may interact with macroparticles of another species, such as in a collision like impact ionization. Alternately, macroparticles of a species can interact with a Fluid or a Field. There are some random interactions that involve only fluids and fields, such as field ionization of a neutral gas, where a fluid represents the neutral gas.

There are two kinds of objects used by the Monte Carlo Interactions package: the incidentSelector object and the interaction objects themselves. To help remain clear as we describe these objects, we will define some terminology that will be used in this chapter:

- *interaction*: The italicized word **interaction** is used to describe the specific kind of interaction used in a simulation, such as impact ionization with an impacting species colliding with a neutral fluid background gas.
- *incident*: The italicized word **incident** is used to describe a particular example of an interaction event. In the impact ionization macroparticle N of the impacting species colliding with the neutral fluid in cell (i,j,k) in which macroparticle N is located. There are two general kinds of incidents in a given VORPAL simulation: possible incidents and selected incidents. Possible incidents are incidents that can potentially occur, based on the interaction algorithm and a probability for occurring. Selected incidents are incidents that have been selected to be performed, a subset of the list of possible incidents.

incidentSelector: object in VORPAL that determines the selected incidents from the list of possible incidents. There are various algorithms for doing this, with each method best suited for different simulation situations. For any given list of interactions in a VORPAL simulation, there can be only one incidentSelector. By separating the interactions and the incidentSelector, we can distinguish between the interaction physics and the Monte Carlo algorithm.

The next sections describe how the Monte Carlo Interaction package works in general and how to use it in VORPAL simulations.

4.7.1 Resolution Issues with Monte Carlo Interactions

Like all simulation, resolution is very important for accurate modeling of the desired physics. For Monte Carlo interactions, resolution issues can be slightly different than for other simulations. Like the simulation of other many other processes, it is important to resolve the physical distributions of the interacting particles as well as the temporal evolution of the distributions. However, for Monte Carlo interactions, additional demands must be satisfied to properly resolve the physics.

First, consider the issue of spatially resolving the interaction physics. The most obvious aspect of this issue is making sure that the number of macroparticles in the simulation is large enough to smoothly resolve the spatial distributions of the species. Additionally, however, random interactions between particles occur with non-negligible probability only when the particles are in close proximity. To avoid checking the

N^2

distances between

N

interacting macroparticles, the VORPAL Monte Carlo Interactions package limits interactions to those between macroparticles within the same cell. This reduces the number of possible interactions to

$N_{\text{cells}}(N_{\text{ppc}})^2$

where

N_{cells}

is the number of cells in the simulation and

N_{ppc}

is the number of particles per cell. Hence,

$N = N_{\text{ppc}}N_{\text{cells}}$

and

$N^2 = (N_{\text{cells}})^2(N_{\text{ppc}})^2 \gg N_{\text{cells}}(N_{\text{ppc}})^2$

Spatial resolution of the interactions, then, requires using a grid that properly resolves physical distributions of the particles as well.

Temporal resolution is also of crucial importance for accurate modeling of the interactions. Temporal evolution of the particles distributions is obviously part of this, but temporal resolution in random interactions involves more than just the physical motion of the particles themselves. Each Monte Carlo interaction has an intrinsic time scale set by the physics of the interaction itself. In other words, the probability for an interaction event to occur can be written as

$P = dt/T_i$

where

dt

is the simulation time step and

T_i

is the natural time scale associated with the interaction itself. The fundamental probability for an interaction event to occur between two macroparticles in a given cell with volume V , in a time-step

dt

is

$$P = N_1 N_2 \sigma(v) v dt / (N_x V)$$

where v is the relative velocity between the two macroparticles,

N_1

and

N_2

are the numbers of physical particles per macroparticle for each colliding species, and

N_x

is the number of physical particles per macroparticle in the final-state species. This implies that the natural time scale associated with this numerical process is:

$$T_i = N_x V / (N_1 N_2 \sigma(v) v).$$

If the simulation time step

dt

is not small enough to resolve the natural interaction time scale, then inaccurate statistics will result. As one can see from the form of the fundamental time scale, however, it is possible to tune the natural time scale for an interaction by changing the number of fundamental particles per macroparticle in the involved species. However, it should be noted by the user that if a species participates in multiple interactions, it could be challenging to properly resolve all of the different natural time scales in a single simulation.

The last issue of resolution comes from the need to accurately sample the velocity distributions of the interacting particles. Since the probability for an interaction event to occur non-trivially depends on the particle velocities, and since a single macroparticle samples only one point in velocity space, accurate statistics may only be achieved in some simulations with many macroparticles per cell, such that the velocity distributions of the participating species are well sampled within each cell.

Together, these three issues of resolution can lead to very demanding conditions. It is not uncommon for the spatial and velocity resolution demands to lead to large numbers of cells in the simulation as well as large numbers of macroparticles per cell. Temporal resolution demands can lead to very small time steps and very long simulations. The user should address all of these issues when the simulation is being designed.

4.7.2 Monte Carlo Interactions in VORPAL

The Monte Carlo Interactions package is used in VORPAL via the MonteCarloInteractions block. Inside the MonteCarloInteractions block, the user must specify the kind of incidentSelector to use with the selector attribute. Then, within the MonteCarloInteractions block, the user must sequentially add all of the individual Interaction blocks for each interaction to be simulated. Details on each interaction can be found in the VORPAL Reference Manual.

Example MonteCarloInteraction Block Used to Simulate Field Ionization

In this example 'CsNeutralGas' is the name of a neutral gas fluid block, 'electrons' is the name of the species representing the electrons, 'Cs1' is the name of the species representing the ionized gas.

```
<MonteCarloInteractions myMC>

  <IncidentSelector myselector>
    kind = nullOnlySelector
  </IncidentSelector>

  <NullInteraction FluidIonizationCs>
    kind = nullFieldIonization
    input      = CsNeutralGas
    electrons  = electrons
    ions       = Cs1
    polarizationFlag = 1
    frequency  = 1.e15
  </NullInteraction>
</MonteCarloInteractions>
```


GPU COMPUTING

5.1 GPU Computing

GPU Computing: VORPAL supports a limited subset of features for doing computation on NVIDIA GPUs (Graphical Processing Units). This subset is currently restricted to pure Electromagnetic simulations (no particles). In order to use these features, your system must have the NVIDIA CUDA 4.0 toolkit installed. Provided certain criteria are met in terms of simulation size, GPU-accelerated pure Electromagnetic simulation can achieve substantial speedup over parallel (multi-core) CPU simulations. Moreover, this benefit can be quite significant if your simulation requires complex space-time transformations of the Electric and Magnetic fields or your simulation has complex driver source functions for the current fields.

In the current release, VORPAL only supports three-dimensional (3D) pure Electromagnetic simulations. In future releases, we may consider supporting 2D and 1D simulations if significant benefit can be attained. In order to get efficient computations, we discuss a few key simulation considerations that impact performance.

5.1.1 Domain Size Considerations

Due to the nature of the underlying data structure and its effect on the performance of the computations, one needs to pay special attention to the simulation size. In particular, one typically wants to set the number of cells (including guard/halo cells) in the z-direction to a multiple of 64 (128) in double (single) precision for the NVIDIA C2070 (Fermi) architecture. For example, one might choose a simulation with 62 (126) cells in the z-direction. The additional guard cell above and below the physical domain yields 64 (128) total cells in the z-direction. By default, VORPAL uses a data structure for the GPU that ensures aligned memory accesses (through padding). This is essential for achieving high memory bandwidth on the device. However, insisting that domains have a multiple of 64 (128) cells in the z-direction forces all threads in a block to be active. This also ensures efficient utilization of device resources.

However, one must take care not to make simulations too big. Currently, there is a limit of 1024 cells in the z-direction. Although simulation size are essentially unbounded in x and y. One should be aware that sufficient device memory is necessary to fit all of the data for the fields on the device (that is, we do not swap data between the host and the device). In order to determine the maximum simulation size, we use the formula:

$$N_f \times N_{cells} \times Precision < \text{Device Global Memory}$$

$$N_f = \text{Number of Fields (including components)}$$

$$N_{cells} = \text{Total Number of Simulation Cells Per Field (including Guard Cells)}$$

$$Precision = 8(4) \text{ for double (single) precision}$$

As an example consider a cavity simulation with a current source driver, to be executed in double precision on an NVIDIA C2070 GPU. This simulation has 5, 3-component fields for E, B, J, L, and A-L and A are the Dey-Mittra Geometry descriptors which are represented as 3-component fields on the GPU. A C2070 has ~6 GB Device Memory, however in practice, we have found that roughly 5GB or less is accessible in a computation. Then.

$$15 \times 8 \times (4.e7cells) \approx 4.5GB$$

Thus, about 40 million cells (including guard/halo cells) is the upper limit for the size of the simulation that can fit onto a single C2070 for double precision.

To learn more details on the capabilities of your device, run VORPAL with the command line flag

```
-gpuinfo
```

For the C2070, the following will be printed to standard out of your simulation.

```
GPU_Initialize::cudaGetDeviceCount() detects 1 devices available to rank 0
GPU_Initialize::cudaSetDevice() setting local GPU device #0 on rank 0
GPU_Initialize::deviceProperties for device #0 are :
    name=Tesla C2070
    compute capability=2.0
    totalGlobalMem=5.24921 GB
    number of (pitched) cells in the z-direction (fastest varying) for double (single) precision will
```

The last print statement is critical since the number of pitched cells can change between different GPU models. Thus, users should inspect this diagnostic and then restructure simulation parameters accordingly.

5.1.2 Parallel GPU Simulations

In the event that your simulation size exceeds the memory on a single device or you want to accelerate your computation with a multi-GPU simulation, VORPAL offers GPU capabilities that are supported on parallel-GPU machines, i.e. systems or clusters with multiple GPUs. Since every system configuration is different with regard to the number of GPUs per node, it is difficult to give an exact recipe of how to run your simulations in parallel. It useful to illustrate with two examples. On a single node desktop machine with two GPUs, simply using `'mpirun -np 2'` will ensure that both GPUs are used in a parallel simulation. On a machine such as DIRAC at NERSC, where many of the nodes have 1 GPU per node (8 cores), one would run with 1 core per node on N nodes to do an N GPU simulation.

In general, there are strategies for maximizing the performance of a parallel GPU simulation. In particular, one wants to avoid general domain decomposition as this can have an adverse effect on the performance. Ideally, one should decompose only in the x-direction as this is the slowest varying dimension in the data structure.

Example Simulations

Several example simulations are being distributed with this release. These examples show how to modify your input files to take advantage of GPUs. Currently, Electromagnetic computations can only be accessed through the pure Multifield syntax. Thus, EmField blocks of type `yeeEmField` or `yeeStaticElecField` do not support GPU capabilities. However, input files using the *multifield* syntax can be modified in a straightforward manner. Although some structural changes are required to use the Dey-Mittra algorithm or STFunc blocks, many of the other changes are simply keyword changes in *field* and *fieldupdater* blocks.

TROUBLESHOOTING

Debugging the input file:

6.1 Troubleshooting Electrostatic Simulations

This chapter discusses common problems that you may encounter as you modify the `simpleEs` example and develop your own electrostatic simulation.

6.1.1 The Simulation Does Not Finish Properly

The most common cause of crashes is improperly set up particle boundaries. The particle boundaries must completely surround the space in which particles are loaded. Otherwise particles will drift out of the grid and try to reference fields that do not exist. This leads to a VORPAL segmentation fault.

One possible reason for an electrostatic simulation not finishing properly is that a particle has crossed more than one cell in a time step. This could allow the particle to pass through a particle sink without being absorbed.

One solution is to reduce the duration of the time step.

Another solution is to limit the number of cells a particle can cross in one time step by artificially reducing the velocity of high speed particles. For an example of this feature, see the parameter `'maxcellxing = 1'` in the particle species input block in the `simpleEs.pre` file section, *Particle Definition*.

The Definition of the Particle Species Is Incorrect

The following Species input block is not defined correctly:

```
<Species electrons>
  verbosity = 0
  kind = nonRelBoris
  emField = myZeroField
  ...
</Species>
```

The problem: The input block does not specify mass and charge.

The result: VORPAL runs normally with no complaints. The default mass and charge are those of a positron.

6.1.2 The Electrostatic Solver Does Not Converge

If the electrostatic solver does not converge, this often indicates a problem with the setup. The matrix can become singular in a fully periodic system. For solutions, see the section *Fully Periodic Systems*.

6.2 Troubleshooting Electromagnetic Simulations

This section discusses common problems that you may encounter as you modify the simpleEm example and develop your own electromagnetic simulation.

6.2.1 The Simulation Does Not Finish Properly

The most common cause of crashes is improperly set up particle boundaries. The particle boundaries must completely surround the space in which particles are loaded. Otherwise particles will drift out of the grid and try to reference fields that do not exist. This leads to a VORPAL segmentation fault.

A common problem with electromagnetic simulation is the Courant condition. The Courant condition states roughly that the time step must be small enough that a light wave cannot cross more than one cell in a single time step. For more information, see *courant-condition*.

6.2.2 The Output Shows an Unexpected High-Frequency or Checkerboard Pattern

These patterns can be symptoms of an instability resulting from violating the Courant condition. Roughly stated, the Courant condition states that the time step must be small enough that a light wave cannot cross more than one cell in a single time step. For more information, see *courant-condition*.

REFERENCES

Further reading and terminology definitions:

7.1 Bibliography

- [1] Birdsall, C. K., and Langdon, A. B., *Plasma Physics Via Computer Simulation*. Adam Hilger, 1991.tocpartBibliography
- [2] Courant, R., Friedrichs, K., and Lewy, H., “On the partial difference equations of mathematical physics,” *IBM Journal*, p. 215, March 1967. This article is a publication in English of an article originally published in German: Über die partiellen Differenzgleichungen der mathematischen Physik, *Mathematische Annalen*, vol. 100, no. 1, pp. 32-74, 1928.
- [3] Hockney, R.W., and Eastwood, J. W., *Computer Simulation Using Particles*. Adam Hilger, 1988.
- [4] Kutasi, K., and Donko, Z., “Hybrid model of a plane-parallel hollow-cathode discharge,” *Journal of Physics D: Applied Physics*, vol. 33, pp. 1081-1089, 2000.
- [5] Messmer, P., and Bruhwiler, D.L., “A parallel electrostatic solver for the VORPAL code,” *Computer Physics Communications*, vol. 164, pp. 118-121, 2004.
- [6] Nieter, C., and Cary, J. R., “VORPAL: a versatile plasma simulation code,” *Journal of Computational Physics*, vol. 196, pp. 448-473, 2004.
- [7] Reiser, M., *Theory and Design of Charged Particle Beams*. Wiley-Interscience, 1994.
- [8] Villasenor, J., and Buneman, O., “Rigorous charge conservation for local electromagnetic-field solvers,” *Computer Physics Communications*, vol. 69, p. 306-316, 1992.
- [9] Xiang, N., Cary, J. R., Barnes, D. C., and Carlsson, J. A., “Low-noise electromagnetic particle-in-cell simulation of electron Bernstein waves,” *Physics of Plasmas*, vol. 13, pp. 062111.1-062111.13, 2006.
- [10] Yee, K. S., “Numerical solution of initial boundary value problems involving Maxwell’s equations in isotropic media,” *IEEE Transaction of Antennas and Propagation*, vol. 14, p. 302-307, 1966.

7.2 Glossary

domain The rectangular Cartesian grid. The physical domain is the grid specified by a user. The extended domain is the grid with guard cells added by VORPAL

extended domain See domain.

FDTD Finite-difference time-domain. The FDTD method is a technique for solving problems in electromagnetics.

float A floating-point number.

guard cell A cell located outside the user-defined simulation grid that VORPAL adds for parallel processing and other computational purposes. Charges cannot be deposited in guard cells, but you can use guard cells when you describe boundary conditions.

HDF5 Hierarchical Data Format Version 5. A library and file format, developed by the National Center for Supercomputing Applications at the University of Illinois at Urbana-Champaign, for storing graphical and numerical data and for transferring that data between computers. VORPAL outputs data in *HDF5 Format VORPAL Output Files*.

input block In VORPAL, an input block is an object consisting of parameters. Input blocks can be nested within other input blocks. For example, input blocks for boundary conditions are nested within the input block for an electromagnetic field.

input file A VORPAL simulation file, which has a .in suffix. Users produce an input file by running a pre file through a pre-parser. VORPAL processes the input file.

MPI Message Passing Interface. An application programming interface (API) for communicating between processes executing in parallel.

multi-grid pre-conditioner A pre-conditioner that enables a solver to use a hierarchy of grids to solve a partial differential equation problem. The multi-grid pre-conditioner applies the results from coarse grids to accelerate the convergence on the finest grid.

parameter In VORPAL, a parameter is a variable value (integer, floating-point number, or text string) that users define to create a simulation.

parse To divide input into parts and determine the meaning of each part.

physical domain See domain.

pre file A VORPAL simulation file, which has a .pre suffix. Users define a simulation and its variables in a pre file and then run the pre file through a pre-parser to produce an input file.

pre-conditioner An algorithm that works with an electrostatic solver to transfer an original linear system matrix into a matrix that has better convergence behavior.

Python An open-source, interpreted scripting language managed by the Python Software Foundation.

SI units The International System of Units ('Le Systeme International d'Unites'), which has seven base units: meter, kilogram, second, ampere, kelvin, mole, and candela.

solver An algorithm that calculates the results of electrostatic problems.

TxPhysics A cross-platform library of computational modules, provided by Tech-X Corporation, for modeling charged particles.

COPYRIGHT AND TRADEMARK INFORMATION

VORPAL © 2002-2011 University of Colorado and Tech-X Corporation. All rights reserved.

VORPAL © 1999-2002 University of Colorado. All rights reserved.

VORPAL © 2008-2011 Tech-X Corporation. All rights reserved.

For VORPAL licensing details please email sales@txcorp.com.

All trademarks are the property of their respective owners.

Redistribution of all VORPAL simulation input file code block examples from the VORPAL 5.0 Document Set, including the VORPAL User Guide, VORPAL Reference Manual, and Learning VORPAL by Example, is allowed provided that this copyright statement is also included with the redistribution.

INDICES AND TABLES

- *genindex*
- *search*