

VORPAL Reference Manual MultiFields Addendum



Release 4.0.2

Copyright © 2007-2009 Tech-X Corporation. All rights reserved.

VORPAL is copyright 1999-2002 by the University of Colorado, copyright 2002-2009 University of Colorado and Tech-X Corporation. For licensing details, please email sales@txcorp.com.

All trademarks are the property of their respective owners.

This page intentionally left blank.

Contents

25. MultiField.....	1
25.1 Field	3
25.2 FieldUpdater	5
25.2.1 Understanding Region Indices.....	8
25.3 Updater Kinds	10
25.3.1 curlUpdater.....	11
25.3.2 curlUpdaterCoordProd	13
25.3.3 deyMittraConstrainUpdater	14
25.3.4 deyMittraUpdater.....	15
25.3.5 divUpdater	16
25.3.6 divUpdaterCoordProd	17
25.3.7 dummyUpdater	18
25.3.8 edgeToNodeVec	19
25.3.9 epetraUpdater	20
25.3.10 faceToNodeVec.....	22
25.3.11 fieldBinOpUpdater.....	23
25.3.13 fieldSqrDiagUpdater	24
25.3.14 geometryUpdater	25
25.3.15 gradVecUpdater	26
25.3.16 gradVecUpdaterCoordProd	27
25.3.17 lightFrameEnvelopeUpdater	28
25.3.18 lightFrameEnvForceUpdater	29
25.3.19 linIterUpdater	30
25.3.20 linPlasDielcUpdater	32
25.3.21 m2x2VecUpdater.....	33
25.3.22 neutralBoltzmannUpdater.....	34
25.3.23 open.....	35
25.3.24 phaseShiftVecUpdater.....	36
25.3.25 poissonUpdater	37
25.3.26 setEpsilonUpdater	38

25.3.27 smooth1D	39
25.3.28 spaceScalarFieldFuncUpdater	40
25.3.29 STFuncUpdater	41
25.3.30 unaryFieldOpUpdater	42
25.3.31 yeeAmpereUpdater	43
25.3.32 yeeAmpereDielVecUpdater.....	44
25.3.33 yeeConductorUpdater.....	45
25.3.34 yeeFaradayUpdater	46
25.4 Update steps.....	46
25.5 FieldMultiUpdater	48
25.6 PmlRegion.....	49
25.7 ScalarDepositors and VectorDepositors.....	53
25.7.1 ScalarDepositor	54
25.7.2 VectorDepositor	55

25. MultiField

VORPAL has an advanced method of describing fields for simulations called MultiField. The MultiField object holds a collection of fields, field updaters, and update steps. MultiField manages the interface between the objects it contains and the model of the outside world (i.e. the domain). MultiField facilitates development and application of new update methods by separating the data (the fields) from the update algorithms (the field updaters), and makes fields easier and more versatile for the user to employ.

MultiFields are declared within `<MultiField nameOfThisMultiField>` and `</MultiField>` tags. The MultiField block contains the following blocks:

- Field
- FieldUpdater
- UpdateStep
- PmlRegion

In addition to these blocks, the Multifields accepts definitions of the following parameters:

- `dumpPeriod` (integer) -- The number of time steps per dump. MultiField will pass this on to any fields for which `dumpPeriod` is not specifically declared. A `dumpPeriod` of 0 tells VORPAL to never dump the MultiField.
- `externalFields` (string vector) -- The names of the fields that are made available by the rest of the simulation to be used in this MultiField.
- `skinUpdatersInSerial` (option) -- Whether updaters should be subjected to skin/body separation even in serial runs where there's a single domain and no performance benefit. This is helpful for debugging, however, when one wants to see how the serial run would separate an updater into skin and body updaters. The default value for this parameter is false.
- `stdoutRank` (option) -- Specifies which processor will write output messages to stdout. Note that all processors will write to their own comms file. This is useful for debugging. The default value is 0.
- `restoreTimeFromField` (string) -- Name of the field from which MultiField will get the current time when restoring from a dump. This is for cases when MultiField cannot automatically determine the proper field from which to restore.
- `kind` (string) – `lightFrameEnvelopeMultiField` (optional). Usually the correct choice is to not specify a kind of MultiField. The `lightFrameEnvelopeMultiField` is a special case which implements the Laser Envelope Model. See below.

The Multifield kind parameter can be:

- No kind parameter; this is the default. The default behavior of not specifying a kind parameter for MultiFields is usually the correct choice for simulations.
- lightFrameEnvelopeMultifield – this is a kind of MultiField that implements the Laser Envelope Model. This kind of Multifield creates special grids that have the same size and spacing as the main grid, but which move in the +x direction at the speed of light.

Fields can be discretized on Laser Envelope Model grids and interpolated to particle positions. In particular, there are two special grids that can be created:

- active grid: position is initialized to be offset from the main grid by $c*dt/2$ in the x direction
- alternate grid: initially offset by $-c*dt/2$.

The positions of both the active and the alternate grid are shifted by $c*dt$ in the +x direction for every timestep. When using the Envelope Model with particles, the Species should be set to kind=evnBoris.

This lightFrameEnvelopeMultifield kind of multiField uses the following parameters:

- activeLightFrameFields -- specifies the fields which exist on the active light frame grid
- alternateLightFrameFields -- specifies the fields which exist on the alternate light frame grid

The following sections of this chapter cover the types of blocks that can be contained within the <MultiField> block.

25.1 Field

A Field object code block is contained between the `<Field nameOfThisField>` and `</Field>` tags. Field objects take the following parameters:

- `kind` (string) -- The kind of field. Field kind values include:
 - `interior`
 - `depField`
 - `funcField`Please see below for more information on each of these kinds. If no kind is given, the default (and usually correct choice for most simulations) will be used.
- `numComponents` (integer) -- The number of field components. The default is 1. Other field component values include:
 - 1 for a scalar field,
 - 3 for a vector field,
 - 4 for edge4v vectors,
 - 9 for a tensor field.
- `offset` (string) -- The offset positions where the field values are located within a cell. Offset values include:
 - `none` (corner with lowest coordinates)
 - `center` (center of cell)
 - `edge` (component `j` is at the center of the lowest edge parallel to the `j`th direction)
 - `face` (component `j` is at the center of the lowest face perpendicular to the `j`th direction)
 - `edge4v` (for `rhoJ`, 0-component is at node, 1 is at x-center-edge, 2 at y-center-edge, 3 at z-center-edge)
- `overlap` (integer vector) -- The number of lower and upper guard cells. The guard cells are important if higher order particles are near the simulation boundaries or for communications when running in parallel. The default for a `depField` is [1 2]. Otherwise, the default is [1 1].
- `dumpPeriod` (integer) -- number of time steps between dumps. `dumpPeriod=0` means never dump; if `dumpPeriod` is omitted from the attribute set, the Field will inherit `MultiField`'s `dumpPeriod`. This is the default behavior.
- `interpolation` (string) – Defines the interpolation method used for particles. Interpolation values include:
 - `linearFromNodalFields`
 - `polynomial`
 - `esirkHalfSine`
 - `esirkGaussian`
 - `esirk1stOrder`, `esirk2ndOrder`, ..., `esirk7thOrder`.

For more information on interpolation methods, please see “Higher-Order Particles in VORPAL” in the VORPAL User Manual. The default for `offset=none` is `linearFromNodalFields`, otherwise the default is `esirk1stOrder`.

- `dumpOnly` (integer) -- If the `dumpOnly` option is set to true (1) in a Field block of kind `depField` then depositors depositing into that field will only execute at dump time. This is useful for `depFields` that are dumped but not needed for the update, such as charge density for EM PIC.

In addition to the Field object parameters, a Field code block can also contain other objects that describe attributes of the Field. Field object attribute objects include:

- BoundaryCondition (attribute block) -- a boundary condition for the field.
- InitialCondition (attribute block) -- an initial condition for the field.
- Source (attribute block) -- a source for the field. This is the same as a BoundaryCondition for the field, however may make more sense to the user when introducing a condition over a region of the domain rather than a plane or line one would expect from a boundary condition.

BoundaryCondition, InitialCondition, and Source object kinds of fields include:

- Default is no kind. If no kind parameter specified, the default (and usually the correct choice for most simulations), is that VORPAL will create a normal field whose guard cells are filled from the cells of the domain to which they belong. When a field is shifted in a moving window simulation using this method, interior cells are filled from guard cells on the same domain, and then the guard cells are filled from interior cells on other domains.
- interior -- A field whose guard cells are expected to be invalid; when the field is shifted in a moving window simulation, (newly) interior cells in a domain are filled from the (formerly) interior cells on other domains.
- depField -- A field that fills guard cells and interior cells (during messaging) by adding together the values for a cell on all domains that have that cell (either as interior or a guard cell). Particles are written into depFields.
- funcField -- A field that fills guard cells, or newly interior cells in a moving window simulation, by calculating their values from a function.

25.2 FieldUpdater

Field updaters defined within a MultiField are bracketed by `<FieldUpdater nameOfThisUpdater>` and `</FieldUpdater>` tags. FieldUpdater takes the following parameters:

- `kind` (string) -- type of updater. This parameter is required. FieldUpdater kinds include:?
- `lowerBounds` (integer vector) -- lower bounds of updater region. This parameter is required.
- `upperBounds` (integer vector) -- upper bounds of updater region. This parameter is required.
- `gridBoundary` (string) -- the name of the gridBoundary to determine which cells should be updated by this updater. The default is `universe`; all possible cells should be updated.
- `interiorness` (string) -- the method the gridBoundary uses to define “interior”; one of:
 - `cellcenter` (the default),
 - `deymittra` -- A cell is considered inside a gridBoundary if its center is inside the gridBoundary when `interiorness=cellcenter`. If `interiorness` is `deymittra`, a cell is considered to be inside a gridBoundary if any of the cell corners is inside it. Generally, one should set the `interiorness` to `deymittra` when wanting to use cut cells in the simulation.
 - `dmconstraint` A cell is considered inside a gridBoundary if its center is inside the gridBoundary when `interiorness=cellcenter`. `dmconstraint` is similar to `deymittra`, however should be used for fieldUpdaters of kind = `deyMittraConstrainUpdater`.
- `readFields` (string vector) -- names of fields this updater needs but will not alter (order is important and updater-dependent)
- `writeFields` (string vector) -- names of fields this updater will alter (order is important and updater-dependent)
- `component` (integer) – defines the updater component. The meaning of component depends on the type of updater. For a single-component updater, the component is usually is the updated field component (0 for x, 1 for y ...); for a vector-updater, it may be the first component (0 for x, y, and z). The default is 0.
- `minDim` (integer) -- if `NDIM < minDim`, this updater will not be used; that is, if `minDim=2`, it will be included in 2D and 3D simulations, but not in 1D simulations. The default is 0.
- `restoreTimeFromField` (string) -- Name of the field from which MultiField will get the current time when restoring from a dump. `restoreTimeFromField` is used for cases when MultiField cannot automatically determine the proper field from which to restore.

There are many optional component-dependent region modification flags that can be specified in the FieldUpdater code block as well:

- `expandToTopInComponentDir` (integer) -- when true: if the region upper bound in the direction of the component (mod 3) is the upper bound of the simulation, and that direction is not periodic, then the upper bound is increased by one so that the region includes the top of the simulation. The default value is 0 (indicating false).
- `contractFromBottomInComponentDir` (integer) -- when true: if the region lower bound in the direction of the component (mod 3) is the lower bound of the simulation, and that direction is not periodic, then the lower bound is increased by one so that the region excludes the bottom of the simulation. The default value is 0 (indicating false).
- `expandToTopInNonComponentDir` (integer) -- when true: if the region upper bound in a direction perpendicular to the component (mod 3) is the upper bound of the simulation, and that direction is not periodic, then the upper bound is increased by one so that the region includes

- the top of the simulation. The default is 0 (indicating false).
- `contractFromBottomInNonComponentDir` (integer) -- when true: if the region lower bound in a direction perpendicular to the component (mod 3) is the lower bound of the simulation, and that direction is not periodic, then the lower bound is increased by one so that the region excludes the bottom of the simulation. The default is 0 (indicating false).
 - `expandAboveTopInComponentDir` (integer) -- if the region upper bound in the direction of the component (mod 3) is the upper bound of the simulation or higher, and that direction is not periodic, then the upper bound is increased by this value. The default value is 0.
 - `expandBelowBottomInComponentDir` (integer) -- when true: if the region lower bound in the direction of the component (mod 3) is the lower bound of the simulation or below, and that direction is not periodic, then the lower bound is decreased by this value. The default is 0.
 - `expandAboveTopInNonComponentDir` (integer) -- when true: if the region upper bound in a direction perpendicular to the component (mod 3) is the upper bound of the simulation or higher, and that direction is not periodic, then the upper bound is increased by this value. The default is 0.
 - `expandBelowBottomInNonComponentDir` (integer) -- when true: if the region lower bound perpendicular to the direction of the component (mod 3) is the lower bound of the simulation or below, and that direction is not periodic, then the lower bound is decreased by this value. The default is 0.
 - `cellsToUpdateBelowDomain` (integer vector) -- the number of cells (in each direction, x, y, and z) the updater may update below the local domain. This parameter is used in rare cases to allow boundary-condition updaters to update areas outside of the simulation region in periodic directions, and also to make parallel algorithms more efficient. See below more explanation. The default is [0 0 0].
 - `cellsToUpdateAboveDomain` (integer vector) -- analogous to `cellsToUpdateBelowDomain`. The default is also [0 0 0].
 - `cellsToUpdateBelowDomainInComponentDir` (integer) -- if present, the component of `cellsToUpdateBelowDomain` corresponding to the updater component will be set to the given value (Note that this does not work with `vecUpdaters`.)
 - `cellsToUpdateAboveDomainInComponentDir` (integer) -- analogous to `cellsToUpdateBelowDomainInComponentDir`.
 - `cellsToUpdateBelowDomainInNonComponentDir` (integer) -- if present, the components of `cellsToUpdateBelowDomain` perpendicular to the updater component will be set to the given value (doesn't work with `vecUpdaters`)
 - `cellsToUpdateAboveDomainInNonComponentDir` (integer) -- analogous to `cellsToUpdateBelowDomainInNonComponentDir`
 - `lowerSkinDepth` (integer vector) -- a 2-component vector of integers, [xSkin ySkin zSkin], sets the lower skin depth for the domain; if omitted, VORPAL will guess what the skin depth should be. Note: either both lower- and upperSkinDepth must be used, or neither. This should be used only for parallel runs, and only for users who are comfortable with the update algorithm.
 - `upperSkinDepth` (integer vector) -- analogous to `lowerSkinDepth`, but for the upper sides of the domain.
 - `lowerSkinDepthInComponentDir` (integer) -- if present, the component of `lowerSkinDepth` corresponding to the updater component will be set to this value. The default is to omit this parameter, unless `lowerSkinDepthInNonComponentDir` is present, in which case the default value is zero.
 - `upperSkinDepthInComponentDir` (integer) -- analogous to `lowerSkinDepthInComponentDir`

- `lowerSkinDepthInNonComponentDir` (integer) -- if present, the components of `lowerSkinDepth` perpendicular to the updater component will be set to this value. The default is to omit this parameter, unless `lowerSkinDepthInComponentDir` is present, in which case the default value is zero.
- `upperSkinDepthInNonComponentDir` (integer) -- analogous to `lowerSkinDepthInNonComponentDir`
Note: the bounds are never increased or decreased in a dimension that doesn't exist (i.e., the z-bounds don't exist in a 2D simulation). Also, the field quantities in the cell with an index equal to an upper bound are not updated.

25.2.1 Understanding Region Indices

The region alteration flags facilitate description of the possibly different regions of the three components of an updater (such as a Yee Faraday updater). So far, VORPAL protocol has been to compute the magnetic field on simulation boundary faces, but not to compute the electric field on those boundaries (instead setting the electric field by a boundary condition). If the simulation has $10 \times 10 \times 10$ cells, with lower and upper bounds $[0 \ 0 \ 0]$ and $[10 \ 10 \ 10]$ then typically the bounds of updaters for the various field components (in the Yee scheme) are:

- Ex: $[0 \ 1 \ 1]$ and $[10 \ 10 \ 10]$
- Ey: $[1 \ 0 \ 1]$ and $[10 \ 10 \ 10]$
- Ez: $[1 \ 1 \ 0]$ and $[10 \ 10 \ 10]$
- Bx: $[0 \ 0 \ 0]$ and $[11 \ 10 \ 10]$
- By: $[0 \ 0 \ 0]$ and $[10 \ 11 \ 10]$
- Bz: $[0 \ 0 \ 0]$ and $[10 \ 10 \ 11]$

The actual field components that are in the simulation volume (including the boundary) are:

- Ex: $[0 \ 0 \ 0]$ and $[10 \ 11 \ 11]$
- Ey: $[0 \ 0 \ 0]$ and $[11 \ 10 \ 11]$
- Ez: $[0 \ 0 \ 0]$ and $[11 \ 11 \ 10]$
- Bx: $[0 \ 0 \ 0]$ and $[11 \ 10 \ 10]$
- By: $[0 \ 0 \ 0]$ and $[10 \ 11 \ 10]$
- Bz: $[0 \ 0 \ 0]$ and $[10 \ 10 \ 11]$

To specify the update regions above, one may specify the bounds $[0 \ 0 \ 0]$ and $[10 \ 10 \ 10]$ for all updaters, and add the flag “expandToTopInComponentDir = true” for the magnetic field updaters, and contractFromBottomInNonComponentDir for the electric field updaters. If the region does not touch the global boundary of the simulation, no changes are made.

Note on updating beyond domain: the options `cellsToUpdateBelow/AboveDomain` allow the updater to update beyond the `domainRgn`: the `domainRgn` is the region for which a particular processor is responsible (the main simulation region if running serial on one processor). However, in many cases a processor stores guard cells for a field -- cells that technically belong to another processor; the main simulation region usually also has guard cells. Usually the guard cells are updated by that other processor, and the values sent (messed) to other processors that need those values. Sometimes, when possible, it's more efficient to calculate guard cell quantities than to receive the values from another processor. This option facilitates such transactions. To understand how to use the `cellsToUpdateBelow/AboveDomain` options, it helps to understand how an updater figures out which cells to update.

1. The updater finds the largest region which it is allowed to update; usually this is the local `DomainRgn` (for example, `lowerBounds = [4 6 7]`, `upperBounds = [10 12 14]`), but it will be expanded according to the following rules:
 - (a) If the `DomainRgn` is at the bottom of the simulation in direction `d` (and `d` is not a periodic direction), the updater will be allowed to write as far below the `DomainRgn` in direction `d` as it can without going beyond field data (VORPAL may incorrectly estimate

how far this is). The top of the simulation is treated similarly. Note: boundary conditions often update field values beyond the global DomainRgn; this features allows that.

- (b) If the DomainRgn is not at the bottom of the simulation in direction d (or d is periodic), then the updater will be allowed to write below the DomainRgn by the number of cells given by (the d th element of) `cellsToUpdateBelowDomain`. For example, if `cellsToUpdateBelowDomain = [1 2 2]`, the `lowerBounds` will be adjusted to `[3 4 5]` (that is, the largest allowed region will have `lowerBounds = [3 4 5]`).
2. The updater takes the intersection of the region specified in the input file with the largest allowed region for the particular processor, found in the previous step; this is the updater's update region.
 - (a) Tricky issue: In the case that direction d is periodic, the region specified in the input file is considered to include translations of itself in direction d . For example, consider a 1D simulation with N cells. If the updater bounds are specified in the input file as 0 to N , and `cellsToUpdateBelowDomain = [1]`, then the allowed region will be -1 to N , and the updater will update cell -1 because cell -1 is equivalent to cell $N-1$, and cell $N-1$ is included in the updater bounds. In other words, the intersection of $[-1, N)$ and $[0, N)$ is $[-1, N)$. However, if the update region had been $[0, N-5)$, excluding cell $N-1$, then the updater would not update cell -1 . Because this might be confounding, and VORPAL's logic may not be perfect, this notion is implemented when the updater region (specified in the input file) is altered, before the updater is created. VORPAL will try to extend the updater bounds in periodic directions as it sees fit. In other words, $[0, N)$ would be extended to $(-1, N]$, and this would be the region that appears in MultiField's attribute set, which it writes to the standard output, as well as to the per-rank streams. However, $[0, N-5)$ would be unchanged. Then, a straightforward intersection is applied.

The updater then tries to make sure that the updater only needs field data that it can get (so that it doesn't try to go past the array, causing a segfault).

3. The component/direction-specific options, such as `cellsToExtendBelowDomainInComponentDir`, alter the updater attribute set, before the updater is created, and add, say, `cellsToExtendBelowDomain`, as appropriate.

25.3 Updater Kinds

The following are the kinds of updaters that can be used in a FieldUpdater block. They are described in detail in the following subsections.

- curlUpdater
- curlUpdaterCoordProd
- deyMittraConstrainUpdater
- deyMittraUpdater
- divUpdater
- divUpdaterCoordProd
- dummyUpdater
- edgeToNodeVec
- epetraUpdater
- faceToNodeVec
- fieldBinOpUpdater
- fieldSqrDiagUpdater
- geometryUpdater
- gradVecUpdater
- gradVecUpdaterCoordProd
- lightFrameEnvelopeUpdater
- lightFrameEnvForceUpdater
- linIterUpdater
- linPlasDielcUpdater
- m2x2VecUpdater
- neutralBoltzmannUpdater
- open
- phaseShiftVecUpdater
- poissonUpdater
- setEpsilonUpdater
- smooth1D
- spaceScalarFieldFuncUpdater
- STFuncUpdater
- unaryFieldOpUpdater
- yeeAmpereUpdater
- yeeAmpereDielVecUpdater
- yeeConductorUpdater
- yeeFaradayUpdater

25.3.1 curlUpdater

Updaters of kind=curlUpdater perform the following operation:

$$F_{c+i} += (c_0 + c_1 \Delta t) [A(\nabla \times G)_{c_g+i} + BH_{c_h+i}]$$

where F is the writeField, G is the first readField, H is an optional second readField (if missing, the coefficient B is taken to be zero), c is the updater component, [c0 c1] are the dtCoefficients, A and B are the readFieldFactors, and [cg ch] are the readFieldCompShifts (usually both zero, but see below). Updates of kind=curlUpdater take the parameters listed below.

- differencing (string) -- one of either forward or backward depending on which is desired. See below for more detail on forward and backward differencing. This parameter is required.
- dtCoefficients (real vector) -- two components [c0 c1] as defined in the equation above. The result of the updater will be multiplied by (c0 + c1 Δt), where Δt is the current time step. The default is [1. 0.].
- readFieldCompShifts (optional vector) -- [cg ch] the number of components by which to shift the iterators on the first field and the (optional) second field. e.g., if the B field is represented by components 3-5 of the field “EandB”, then to calculate curlB, one would specify readFields = [EandB] and readFieldCompShifts = [3]. The default is [0 0].
- readFieldFactors (real vector) – the coefficients A and B [A B] that multiply each readField. If there is only one readField, then there should only be one element of readFieldFactors [A].
- useVecUpdater (integer) -- if true, the updater will make just one pass through all cells, and in each cell it will either update all components or none. In general, the default component parameter of zero is ideal. In principle, however, one could use a 6-component E-B field and specify component = 3, which for the Faraday update, which would then update components [3 4 5] of the E-B-field. Also, if true, it will ignore all component-dependent region modification flags. By default, useVecUpdater is false.

Forward and backward differencing are defined as:

$$\text{forward: } (\nabla \times G)_{i,j,k,x} = \frac{G_{i,j+1,k,z} - G_{i,j,k,z}}{\Delta y} - \frac{G_{i,j,k+1,y} - G_{i,j,k,y}}{\Delta z}$$

$$\text{backward: } (\nabla \times G)_{i,j,k,x} = \frac{G_{i,j,k,z} - G_{i,j-1,k,z}}{\Delta y} - \frac{G_{i,j,k,y} - G_{i,j,k-1,y}}{\Delta z}$$

Note: the Yee Ampere (Ey) update performs the following update:

```
<FieldUpdater ampere-y>
  kind = curlUpdater
  component = 1
  differencing = backward
  writeFields = [YeeElecField]
  readFields = [YeeMagField SumRhoJ]
  readFieldFactors = [c^2 ~(-1/epsilon_0)]
  dtCoefficients = [0. 1.]
  readFieldCompShifts = [0 1]
```

</FieldUpdater>

or

$$E_1(t + \Delta t) = E_1(t) + (\Delta t)[c^2(\nabla \times B)_1 - (1/\epsilon_0)S_2]$$

in which E is the YeeElecField, B is the YeeMagField, and S is the SumRhoJ field -- ,

$$S = (\rho, J_0, J_1, J_2)$$

So, adding S2 is adding J1; therefore we have to shift the component on the SumRhoJ field so that J_{mu} gets added to E_{mu}.

25.3.2 curlUpdaterCoordProd

curlUpdaterCoordProd is a specialization of curlUpdater that should be used whenever non-uniform or non-Cartesian grids are used in the simulation. curlUpdaterCoordProd uses the same parameters as curlUpdater.

The curlUpdaterCoordProd introduces the following parameter:

- includeCylAxis (integer) – set this to true (1) if $r=0$ is included in the simulation. To obtain the correct behavior at the axis, one needs to specify two separate curlUpdaterCoordProd updaters. One with the axis and one without.

For example:

```
<FieldUpdater yeeFaraday_0>
  kind=curlUpdaterCoordProd
  useVecUpdater=1
  differencing=forward
  lowerBounds=[0 1 0]
  upperBounds=[NZ NR NPHI]
  dtCoefficients=[0.0 1.0]
  readFieldFactors=[-1.0]
  readFields=[YeeElecField]
  writeFields=[YeeMagField]
</FieldUpdater>
<FieldUpdater yeeFaraday_0_cyl>
  kind=curlUpdaterCoordProd
  useVecUpdater=1
  differencing=forward
  includeCylAxis=1
  lowerBounds=[0 0 0]
  upperBounds=[NZ 1 NPHI]
  dtCoefficients=[0.0 1.0]
  readFieldFactors=[-1.0]
  readFields=[YeeElecField]
  writeFields=[YeeMagField]
</FieldUpdater>
```

25.3.3 `deyMittraConstrainUpdater`

The `deyMittraConstrainUpdater` sets the electric field on the edges fully inside the conductor in a cut cell. This is done such that the interpolated value of the electric field at the center of the cut segment obeys the condition that the electric field parallel to the surface is zero. For cuts with two unknown edges, it also uses the constraint that the derivative of the normal electric field at the center of the cut segment is zero. By constraining the electric fields on the edges in this manner the interpolated fields will better match the correct behavior as they approach the conducting surface represented by the cut segment.

The `deyMittraConstrainUpdater` is designed purely to provide better field interpolation in the cut cells and is not meant to do any dynamics. It should be used in conjunction with the `deyMittraUpdater`.

Only one `readField` and one `writeField` are allowed. This updater expects a `readField` of an electric field with an offset of edge, and a `writeField` of an electric field with an offset of edge (same field in read and write).

25.3.4 deyMittraUpdater

The deyMittraUpdater does the Yee Faraday update for Dey Mittra (cut) cells. It updates 3 components of the magnetic field, starting with “component” and using components 0, 1, and 2 for the electric field.

Only one readField and one writeField are allowed. This updater expects a readField of an electric field with an offset of edge, and a writeField of a magnetic field with an offset of face.

25.3.5 divUpdater

Takes the divergence of the readFields and writes to the writeFields. Only one readField and writeField are allowed.

This updater introduces the following parameters:

- factor (real) – This is the factor that multiplies the end result after taking the divergence. The default is 1.
- skipFirst (integer) – Set this flag to 1 (true) when taking the divergence of J in a sumRhoJ field. A sumRhoJ field has 4 components, the first of which is rho, so the first component of this field must be skipped to get to the J components. The default is 0.
- differencing – one of forward or backward. Backward is the default and is used for the divergence of the electric field. Forward is used for divergence of magnetic field in the Yee scheme.

25.3.6 divUpdaterCoordProd

divUpdaterCoordProd is a specialization of divUpdater which should be used whenever non-uniform or non-Cartesian grids are used in the simulation. It uses the same parameters as divUpdater.

The divUpdaterCoordProd introduces the following parameter:

- includeCylAxis (integer) – set this to true (1) if $r=0$ is included in the simulation. To obtain the correct behavior at the axis, one needs to specify two separate divUpdaterCoordProd updaters. One with the axis and one without.

For example:

```
<FieldUpdater Div_k_Grad_T>
  kind=divUpdaterCoordProd
  lowerBounds=[0 1 0]
  upperBounds=[NZ NR NPHI]
  readFields=[HeatFlux]
  writeFields=[dTemp]
</FieldUpdater>
<FieldUpdater Div_k_Grad_T_axis>
  kind=divUpdaterCoordProd
  includeCylAxis=1
  lowerBounds=[0 0 0]
  upperBounds=[NZ 1 NPHI]
  readFields=[HeatFlux]
  writeFields=[dTemp]
</FieldUpdater>
```

25.3.7 dummyUpdater

The dummyUpdater is an updater that does not update values. Use of dummyUpdater may be helpful if you require specific timing and messaging of fields. In the update step, you can specify a toDtFrac such that during that update step, nothing is done to the field except for modifying its time.

25.3.8 edgeToNodeVec

Interpolates field components known at center edges along direction 0, 1, and 2 to the nodes.

This Updater uses a the parameter:

- nodeToEdge (string) -- This is set to either true or false, and tells VORPAL whether to perform the transpose operation, interpolating from nodes to edges. The default is false.

25.3.9 epetraUpdater

An epetraUpdater works by defining a matrix relationship

$$Ax = b$$

e.g.

$$\begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \begin{bmatrix} F_1 \\ G_1 \end{bmatrix} = \begin{bmatrix} F_2 \\ G_2 \end{bmatrix}$$

between two vectors (x and b) and the matrix A. This kind introduces the following parameters:

- `problemType` (string) -- One of either multiply or solve. The update is either that one starts with x and gets b (multiply) or one starts with b and gets x (solve). If multiply, the readFields should describe x and if solve, the readFields should describe b. The default is multiply.
- `readComponents` (integer vector) – The component to use of each of the readFields.
- `writeComponents` (integer vector) – The component to write of each of the writeFields.
- `readFactors` (real vector) – Optional vector describing the factors by which to multiply the readFields prior to any updating.
- `writeFactors` (real vector) – Optional vector describing the factors by which to multiply the writeFields prior to writing.
- `maxIters` (integer) – The maximum number of iterations to take for convergence to the desired residual. The default is 100.
- `solver` (string) – One of cg, gmres, cgs, tfqmr, bicgstab, or lu. These are the iterative solves available in Aztec that can be used for solving a linear system of equations. Please refer to the Trilinos documentation for further details.
- `scaling` (string) – One of none, Jacobi, row_sum, sym_diag, or sym_row_sum. Please refer to the Trilinos documentation for further details on these scalers.
- `precond` (string) – One of ml, dom_decomp_ilu, dom_decomp_ilut, neumann, ls, or jacobi. Please refer to the Trilinos documentation for further details on these preconditioners.
- `desiredResid` (real) – The desired residual. The default is 1.e-9
- `diagFac` (real) – This should be the value equivalent to the diagonal value on the matrix. `diagFac` is used to scale the equation. The default is 1.
- `boundaryAtBottomInComponentDir` (integer) – If true, the boundary is at the bottom in the component direction. E.g., do not put in a stencil for Vx along the IX=0 plane, as that will be filled in by boundary conditions.
- `boundaryAtBottomInNonComponentDir` (integer) – If true, the boundary is at the bottom in the non-component directions. E.g., do not put in a stencil for Vy and Vz along the IY=IZ=0 planes, as they will be filled in by boundary conditions.
- `boundaryAtTopInComponentDir` (integer) -- If true, the boundary is at the top in the component direction. E.g., do not put in a stencil for Vx along the IX=0 plane, as that will be filled in by boundary conditions.
- `boundaryAtTopInNonComponentDir` (integer) -- If true, the boundary is at the top in the non-component directions. E.g., do not put in a stencil for Vy and Vz along the IY=IZ=0 planes, as they will be filled in by boundary conditions.

An example for the boundary component flags for the electric field in FDTD:
 boundaryAtBottomInComponentDir = [0]
 boundaryAtBottomInNonComponentDir = [1]
 boundaryAtTopInComponentDir = [1] (Since Ex is actually outside the box)
 boundaryAtTopInNonComponentDir = [1]

To define the matrix A, you must use a MatrixFiller block. At least one MatrixFiller block is required for the epeetraUpdater. The MatrixFiller block uses the following parameter:

- kind – one of either interior or cutcell. This parameter is required.

The MatrixFiller block also must contain at least one StencilElement block. The StencilElement block defines the non-zeros in a row of the matrix. The StencilElement blocks contain the following parameters:

- minDim (integer) -- if $NDIM < minDim$, the element will not be inserted into the matrix. The default is 1.
- rowFieldIndex (integer) -- the row index of this element in the matrix A. This parameter is required.
- columnFieldIndex (integer) -- the column index of this element in the matrix A. This parameter is required.
- cellOffset (integer vector) -- [m n p], as shown below. The cellOffset is with respect to the center cell. The cellOffset allows the user to define a differencing scheme for the entire grid. This parameter is required.
- value (real) -- the value of the stencil element. This parameter is required.

An example of a 2D stencilElement is shown below with the cellOffset vectors and values in the corresponding grids.

	[0,1,0]	
cellOffset = [-1,0,0] value = -0.866	cellOffset = [0,0,0] value = 0.866	[1,0,0]
	[0,-1,0]	

25.3.10 faceToNodeVec

Interpolates field components known at center faces perpendicular to direction 0, 1, and 2 to the nodes. This updater adds a new parameter:

- nodeToFace (string) -- This is set to either true or false, and tells VORPAL whether to perform the transpose operation, interpolating from nodes to faces. The default is false.

25.3.11 fieldBinOpUpdater

The fieldBinOpUpdater applies a mathematical operation on two fields (F and G) specified through the readFields parameter and writes the result to the writeFields parameter. This updater introduces the following parameters:

- binOp (string) – The operation to apply to the fields. One of add, subtract, multiply or divide. This parameter is required. add: $(aCoeff * F_j) + (bCoeff * G_j)$, subtract: $(aCoeff * F_j) - (bCoeff * G_j)$, multiply: $(aCoeff + F_j) * (bCoeff + G_j)$, divide: $(aCoeff + F_j) / (bCoeff + G_j)$
- aCoeff (real) – The coefficient for the first field (see above). Default for add and subtract is 1.0. Default for multiply and divide is 0.0.
- bCoeff (real) – The coefficient for the second field (see above). Default for add and subtract is 1.0. Default for multiply and divide is 0.0.

25.3.13 fieldSqrDiagUpdater

An updater of the kind fieldSqrDiagUpdater computes an integral over the sum of the fields squared. The calculation is written to the standard output of the run. fieldSqrDiagUpdater is useful for measuring field energy and reflected components.

$$F = a \int (f_1^2 + f_2^2 + \dots + f_n^2) dV$$

fieldSqrDiagUpdater uses the following parameters:

- factor (real) – The factor (a) by which to multiply the integral. The default is 1.0.
- writePeriod (integer) – The frequency of which to write out the calculation. A writePeriod of 1 will write out the calculation at each time step. The default is 0.

The fields ($f_1^2 + f_2^2$ etc...) used in the integral are given by the readFields parameter. No write fields are allowed as the calculations are written to the standard output.

25.3.14 geometryUpdater

The geometryUpdater evaluates a geometric quantity (pertaining to the individual cell) and sets (or adds or multiplies) the field component to that quantity.

For updating a single component of F, the updater is straightforward; the operation is specified by the operation parameter.

For updating multiple components of F, there is some intricacy if different components of F are located at different places (e.g., for a Yee electric field, E_x and E_y are located at different places within a mesh cell). In this case, the geometricQuantity is evaluated only once (for all components); it is evaluated at the position of the component given by the component parameter of the updater and the field offset of field F. However the components of F that are updated are those given by the parameter writeComponents.

If you want to have f evaluated at the location of each component, then you must use separate STFuncUpdaters for each component (or a FieldMultiUpdater).

The geometryUpdater object uses the following parameters:

- operation (string) -- either: set $F_j=g$, add $F_j+=g$, or multiply $F_j*=g$. This parameter is required.
- geometricQuantity (string) -- the geometric quantity g (see operation) to be evaluated. Choice of: volumeFraction (the fraction of the volume in the cell which is inside the specified gridBoundary), surfaceOutwardNormal (the outward normal to the surface of the gridBoundary in a cell), surfaceArea, faceAreaFraction or edgeLineFraction.
- queryGridBoundary (string) -- the name of the gridBoundary to be used to calculate the geometricQuantity (as opposed to the gridBoundary all updaters have, which determines which cells get updated by that updater). This parameter is required.
- queryComponent (integer) -- the component of the data to be read (for the surfaceOutwardNormal, edgeLineFraction and faceAreaFraction quantities). The default is 0.
- component (integer) -- the component j of F to be updated. However, if writeComponents is not [0], the component, along with the field offset of F determines the location (in each cell) where the function f will be evaluated. The default is 0.
- writeComponents (integer vector) -- a list of components j of F that will be updated. f will be evaluated only once (per cell) for all components, and the location within each cell at which f is evaluated is determined by the 'component' parameter and the field offset of F. The default is [0].

25.3.15 gradVecUpdater

The gradVecUpdater performs the gradient of a vector field given by the readFields parameter and writes it to writeFields. gradVecUpdater uses the following parameter:

- factor (real) – The factor by which to multiply the field. Default is 1.0.
- differencing – one of forward or backward. The default is forward which does a gradient of the potential to get the electric field. Backward result gives B for a Yee cell.

25.3.16 gradVecUpdaterCoordProd

gradVecUpdaterCoordProd is a specialization of gradVecUpdater that should be used whenever non-uniform or non-Cartesian grids are used in the simulation. gradVecUpdaterCoordProd uses the same parameters as gradVecUpdater.

gradVecUpdaterCoordProd uses the following parameter:

- includeCylAxis (integer) – set this to true (1) if $r=0$ is included in the simulation. To obtain the correct behavior at the axis, one needs to specify two separate gradVecUpdaterCoordProd updaters. One with the axis and one without.

For example:

```
<FieldUpdater gradPhi>
  kind=gradVecUpdaterCoordProd
  lowerBounds=[0 1]
  upperBounds=[NZ NR]
  readFields=[phi]
  writeFields=[edgeE]
  factor = 1.0
</FieldUpdater>
<FieldUpdater gradPhi_axis>
  kind=gradVecUpdaterCoordProd
  includeCylAxis=1
  lowerBounds=[0 0]
  upperBounds=[NZ 1]
  readFields=[phi]
  writeFields=[edgeE]
  factor = 1.0
</FieldUpdater>
```

25.3.17 lightFrameEnvelopeUpdater

lightFrameEnvelopeUpdater updates the envelope fields in the laser envelope model.

lightFrameEnvelopeUpdater must be specified in a MultiField of kind lightFrameEnvelopeMultiField. The model represents the laser envelope using two fields: an "active" field, which is discretized at time $dt/2$ later than fields on the main grid, and the "alternate" field, which is $dt/2$ earlier. There is also a field which contains the plasma susceptibility, derived from the particles. Use the readFields parameter to specify the active field, alternate field, and susceptibility field, in that order. The writeFields parameter specifies the active and alternate fields. When using the envelope model with particles, the Species should be kind=evnBoris.

lightFrameEnvelopeUpdater uses the following parameters:

- omega (real) -- The angular frequency of the laser, in Hz

lightFrameEnvelopeUpdater requires the inclusion of a solver block in the updater. This block specifies parameters for the linear solver used to compute the update. In general, the possible values of solver block parameters correspond to the values of parameters of the AztecOO library used by VORPAL. Solver parameters that you may want to adjust are:

- kind (string) – One of cg, gmres, cgs, tfqmr, or bicgstab. These are the iterative solves available in Aztec that can be used for solving a linear system of equations. Please refer to the Trilinos documentation for further details.
- output (string) – One of all, none, warnings, last, or brieflast. The desired level of output.
- tolerance (real) -- The tolerance for solver convergence.
- precondition (string) – One of ml, dom_decomp_ilu, dom_decomp_ilut, neumann, ls, or jacobi.

Please refer to the Trilinos documentation for further details on these preconditioners.

25.3.18 lightFrameEnvForceUpdater

The lightFrameEnvForceUpdater computes the ponderomotive force from an envelope field specified in readFields and stores the results in the field specified in writeFields. This updater must be specified in a MultiField of kind lightFrameEnvelopeMultiField. When using the envelope model with particles, the Species should be kind=evnBoris.

Use of lightFrameEnvForceUpdater requires that you include the following parameters in the Species block:

- envelopeMultiField -- gives the user specified name of the envelope MultiField.
- envelopeFields – gives the user defined names of the active and alternate fields.
- forceField – gives the user defined name of the force field.

25.3.19 linIterUpdater

linIterUpdater is a general linear updater, based on the iterator approach. Depending on the specified parameter operation, linIterUpdater can perform the following updates, where E and F^1, \dots, F^N are fields:

$$E_i(x, y, z) = \sum_j A_j F_{c_j}^{f_j}(x + m_j \Delta x, y + n_j \Delta y, z + p_j \Delta z)$$

$$E_i(x, y, z) += \sum_j A_j F_{c_j}^{f_j}(x + m_j \Delta x, y + n_j \Delta y, z + p_j \Delta z)$$

$$E_i(x, y, z) *= \sum_j A_j F_{c_j}^{f_j}(x + m_j \Delta x, y + n_j \Delta y, z + p_j \Delta z)$$

$$E_i(x, y, z) /= \sum_j A_j F_{c_j}^{f_j}(x + m_j \Delta x, y + n_j \Delta y, z + p_j \Delta z)$$

In each case, we can describe the operation as a translation-invariant matrix A multiplied by the field F (and E is set, summed with, multiplied by, or divided by the result).

Actually, the linIterUpdater can perform an update like the above for several E simultaneously (replace E_i with E_i^f and j with r_j in the above).

linIterUpdater updates cell-by-cell, based on an internal iterator approach (hence the name “iter”). linIterUpdater evaluates the right-hand-side (above) for a cell then applies them to the left-hand-side (above) for a cell, and then proceeds to the next cell. This is important to remember if a field appears both in the Fs as well as the Es.

The matrix A is described (in the input file) in such a way as to be compatible with other matrix solvers in VORPAL. There are a few differences, however, because with the linIterUpdater, the full matrix is never created, so the linIterUpdater can do some extra operations to modify matrix elements at each time-step with negligible computation.

First, the operation can be chosen, as described above. $E += F$ could be re-written as

$$E = F^1 + F^2$$

where F^2 is F and F^1 is E. Multiplication and division are done cell-by-cell, nothing to do with matrix multiplication.

Second, the matrix coefficients may be modified at each time-step. In many updates, for instance, one prefers to multiply the coefficients by the time-step at each time-step, allowing for (usually only very slightly) varying time-steps. This operation is probably too costly to implement in matrix updaters that construct the entire matrix. (Of course, such updaters can use matrices that are not translationally invariant, which the linIterUpdater cannot.)

linIterUpdater cannot perform matrix solves, so rowFieldIndex always means writeFieldIndex (though you must use the notation rowFieldIndex), and columnFieldIndex means readFieldIndex, referring to the index of fields in the writeFields and readFields lists. For example, if readFields = [E E E F F F] and readComponents = [0 1 2 0 1 2], then columnFieldIndex = 3 refers to F0.

linIterUpdater uses the parameters:

- operation (string) -- either: set, add, multiply, or divide. This parameter is required.
- readComponents (integer vector) -- for each readField, a component; references to the jth readField will use the component specified in jth element of this vector. This parameter is required.
- writeComponents (integer vector) -- for each writeField, a component; references to the jth writeField will use the component specified in jth element of this vector. This parameter is required.
- dtCoefficients (real vector) -- 2 components [c0 c1]. The matrix coefficients will be multiplied by $(c_0+c_1 \Delta t)$, where Δt is the current time step (if c1 is not specified it is assumed to be zero). The default vector is [1.0 0.0]

You must also include within the the linIterUpdater block a FieldMatrix block to describe the matrix B. You should describe each element of the matrix within a StencilElement block. The FieldMatrix block is required.

The FieldMatrix block should contain only StencilElement blocks. StencilElement takes the following parameters:

- minDim (integer) -- if $NDIM < minDim$, the element will not be inserted into the matrix. The default is 1.
- rowFieldIndex (integer) -- the row index of this element (always the writeFieldIndex in linIterUpdater). This parameter is required.
- columnFieldIndex (integer) -- the column index of this element (always the readFieldIndex in linIterUpdater). This parameter is required.
- cellOffset (integer vector) -- [mj nj pj], as described above in the equations. This parameter is required.
- value (real) -- the actual matrix element (before being multiplied by $c_0+c_1\Delta t$, as requested by dtCoefficients). This parameter is required.

25.3.20 linPlasDielcUpdater

The linPlasDielcUpdater kind is used to describe linear plasma dielectric model for cold plasma. There are some specifics with this updater. First, readFields should always have a background magnetic field defined (B0) in the multifield block. Second, if damping is used the damping fields need to be added to the read fields.

For example:

```
<FieldUpdater plasmaDielectric>
  kind = linPlasDielcUpdater
  lowerBounds = [0 0 0]
  upperBounds = [NX1 NY1 NZ1]
  nspecies = 2
  # common mass numbers: e=0.5486e-3, He=4.00, N=14.00, Ne=20.18, Ar=39.95,Kr=83.80
  massNumbers = [83.80 0.5486e-3]
  chargeNumbers = [1.0 -1.0]
  $if INCLUDE_DAMPING==1
    includeDamping = 1
    readFields = [B0 density0ion1 density0electron nuPML nuIon1 nuElectron]
  $else
    readFields = [B0 density0ion1 density0electron]
  $endif
  writeFields = [nodalE linearJion1 linearJelectron]
</FieldUpdater>
```

In the update step, the nodalE field must be specified in the messageFields parameter. See the example below:

```
# in sequence of update steps
<UpdateStep step08>
  toDtFrac = 1.0
  messageFields = [nodalE]
  updaters = [plasmaDielectric]
</UpdateStep>
```

25.3.21 m2x2VecUpdater

M2x2VecUpdater performs a linear transform on two fields (F and G), multiplying them by a 2 x 2 matrix at every point for 3 components.

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} F \\ G \end{bmatrix} = \begin{bmatrix} aF + bG \\ cF + dG \end{bmatrix}$$

m2x2VecUpdater uses the following parameters:

- row1 (real vector) – A 2 element vector containing [a b] values
- row2 (real vector) – A 2 element vector containing [c d] values

m2x2VecUpdater requires two writeFields to specify F and G and zero readFields. A particular case is the phase-shift update (see phaseShiftVecUpdater), when the two fields are the real and imaginary components of a complex field.

25.3.22 neutralBoltzmannUpdater

neutralBoltzmannUpdater computes the electrostatic potential for a neutral Boltzmann electron fluid.

neutralBoltzmannUpdater uses the following parameters:

- n_0 (real) – The background density in $\#/m^3$. This parameter is required.
- T_0 (real) – The electron temperature in eV. This parameter is required.
- q_0 (real) – The fundamental charge. This parameter is required.
- ϕ_0 (real) – The electron potential that goes in the Boltzmann equation. This parameter is required.
- m_{ion} (real) – The mass of the ion. This parameter is required.
- $mindensity$ (real) – The lowest density allowed. This parameter is required.

Only one readField (ρ_J) and one writeField (ϕ) are allowed in this updater.

25.3.23 open

An updater of kind=open allows for outgoing waves on a Yee grid. This is only allowable as a boundary condition on the domain boundary.

This open updater introduces the following parameters:

- `velOverC` (real) – The ratio of the wave velocity to the speed of light. Can be negative to specify direction of propagation. This parameter is required.
- `normalDir` (integer) – The propagation axis of the outgoing wave. If not specified, VORPAL guesses the axis will be the first simulated direction with upper bound 1 greater than the lower bound.

25.3.24 phaseShiftVecUpdater

phaseShiftVecUpdater is a special case of the m2x2VecUpdater. phaseShiftVecUpdater performs a linear transform on two fields (F and G), multiplying them by a 2 x 2 matrix at every point for 3 components.

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} F \\ G \end{bmatrix} = \begin{bmatrix} aF + bG \\ cF + dG \end{bmatrix}$$

However, in this special case of a phase shift, F and G are the real and imaginary components of a field E, and $a=\cos(\phi)$, $b=-\sin(\phi)$, $c=\sin(\phi)$, $d=\cos(\phi)$ and $E \rightarrow e^{i\phi} E$

phaseShiftVecUpdater is intended to be used with periodic boundary conditions to create phase-shifted-PBCs or Bloch BCs. This updater is good for photonic crystals and also very good for measuring reflections from PMLs.

phaseShiftVecUpdater uses the following parameter:

- phase (real) – phi (the phase) as described above in the definitions to a,b,c, and d.

phaseShiftVecUpdater requires two (2) writeFields to specify F and G and zero readFields.

25.3.25 poissonUpdater

poissonUpdater solves Poisson's equation for phi. poissonUpdater is a less flexible version of epetraUpdater. poissonUpdater forces a stencil defined by Poisson's equation.

$$\nabla^2 \phi = -\frac{\rho}{\epsilon_0}$$

poissonUpdater uses the following parameters:

- solver (string) – One of cg, gmres, cgs, tfqmr, or bicgstab. These are the iterative solves available in Aztec that can be used for solving a linear system of equations. Please refer to the Trilinos documentation for further details.
- output (string) – One of all, none, warnings, last, or brieflast. The desired level of output.
- scaling (string) – One of none, Jacobi, row_sum, sym_diag, or sym_row_sum. Please refer to the Trilinos documentation for further details on these scalers.
- maxIters (integer) – The maximum number of iterations to take for convergence to the desired residual. Default is 100.
- factor (real) – The factor by which to divide rho.
- desiredResid (real) – The desired residual. Default is 1.e-9.

poissonUpdater expects one readField (rho) and one writeField (phi).

25.3.26 setEpsilonUpdater

setEpsilonUpdater fills the symmetric inverseEpsilon field (6 components, xx, yy, zz, xy=yx, yz=zy, zx=xz) from the given dielectric functions.

When a cell is partially filled (as determined by subsampling the functions over the cell), the dielectric is averaged appropriately over the cell. The number of samples per cell should increase as the simulation resolution increases, to maintain accuracy.

setEpsilonUpdater uses the following parameters:

- samplesPerCell (integer vector) – The number of locations (in each direction, x,y, and z) at which the dielectric is sampled in each cell. The default is [1 1 1], usually [7 7 7] or higher would be a better choice.

setEpsilonUpdater requires you to use STFunc blocks to define the dielectric constant and the surface normal. The dielectric constant is defined as the components of the dielectric tensor at any location (x,y,z). The surface normal is defined as the components of the vector normal to the dielectric interface at any location (x,y,z) that is within one cell length of the interface.

To define the dielectric constant, use STFunc blocks with a name “dielectricConstNM” where N and M represent the components [X Y Z]. A total of 6 blocks (components) are needed. It will first look for XY, YZ, and ZX, and use them if present. If, e.g., XY is missing, it will use YX instead. If, e.g., XY and YX are both present it will use XY and ignore YX. An example of a dielectricConst block is:

```
<STFunc dielectricConstXZ>  
  kind = expression  
  expression = 1.+H(R^2-(x-X0)^2-(y-Y0)^2-(z-Z0)^2)*(REL_EPS_ZZ-1)  
</STFunc>
```

To define the surface normals (the vector normal to the dielectric interface at any location (x,y,z) that is within a cell length of the interface), use an STFunc block with a name “surfNormalN” where N represents the component [X Y Z]. For example:

```
<STFunc surfNormalX>  
  kind = expression  
  expression = x-X_SPHERE  
</STFunc>
```

The setEpsilonUpdater requires one writeField (the invEpsilon field with 6 components) and zero readFields.

25.3.27 smooth1D

smooth1D updater applies a multiplicative 1D digital filter of the form $a/b/a$ across a 1D grid for the specified component of a specified field. smooth1D is typically used for a single pass smoothing of the current. Only one field component can be smoothed. This updater can be applied to a 1D, 2D or 3D grid and smoothDir will give the specific axis on which to apply it in a 1D fashion.

smooth1D uses the following parameters:

- aFac (real) – The “a” factor of the digital filter. The default is 0.25.
- bFac (real) – The “b” factor of the digital filter. The default is 0.50.
- smoothDir (integer) – The direction over which to smooth. The default is 0.
- smoothComp (integer) – The field component which requires smoothing. The default is 0.

smooth1D requires one readField with the unfiltered values and one writeField to hold the filtered values.

25.3.28 spaceScalarFieldFuncUpdater

Updaters of type `spaceScalarFieldFuncUpdater` take field `F` and field `G` and `STFunc` `f` and perform an operation, such as $F_j = f(x [y,z], G_k)$ or $F_{j+} = f(x [y,z], G_k)$ or $F_j^* = f(x [y,z], G_k)$. The `readField` component, `Gk`, is used in place of time, in the `STFunc`.

For updating a single component of `F`, the updater is straightforward; the operation is specified by the operation parameter. For updating multiple components of `F`, there is some intricacy if different components of `F` are located at different places (e.g., for a Yee electric field, `Ex` and `Ey` are located at different places within a mesh cell). In this case, the `STFunc` is evaluated only once (for all components); it is evaluated at the position of the component given by the component parameter of the updater and the field offset of field `F`. However the components of `F` that are updated are those given by the parameter `writeComponents`.

If you want to have `f` evaluated at the location of each component, then you must use separate `spaceScalarFieldFuncUpdaters` for each component (or a `FieldMultiUpdater`).

`spaceScalarFieldFuncUpdater` uses the parameters:

- `operation` (string) — either: set $F_j = f$, add $F_{j+} = f$, or multiply $F_j^* = f$; this parameter is required.
- `dtCoefficients` (real vector) — 2 components [`c0 c1`]: the function will be multiplied by $(c_0 + c_1 t)$, where t is the current time step (if `c1` is not specified it is assumed to be zero). The default is [1.0 0.0].
- `component` (integer) — the component `j` of `F` to be updated; unless `writeComponents` is not [0], in which case, the component, along with the field offset of `F` determines the location (in each cell) where the function `f` will be evaluated. The default is 0.
- `writeComponents` (integer vector) — a list of components `j` of `F` that will be updated; be careful! `f` will be evaluated only once (per cell) for all components, and the location within each cell at which `f` is evaluated is determined by the ‘component’ parameter. The default is [0].
- `readComponents` (integer vector) — a list of components of `G` used to update `F`: e.g., if `writeComponents` = [0 1 3] and `readComponents` = [2 1 3], then $F_0 = f(x, G_2)$, $F_1 = f(x, G_1)$, and $F_3 = f(x, G_3)$, where `f` is evaluated at the location of component. The default is to set this vector to the same value as `writeComponents`.

The `STFunc` block is required with `spaceScalarFieldFuncUpdater`:

- `STFunc` -- an `STFunc` used to describe `f`. This block will have other parameters depending on the kind; for instance, `kind = expression` will then take the parameter `expression = "` as well. This block is required.

25.3.29 STFuncUpdater

Updaters of kind STFuncUpdater take field F and an STFunc f and perform an operation such as $F_j=f(x,t)$ or $F_j+=f(x,t)$ or $F_j*=f(x,t)$.

For updating a single component of F, the updater is straightforward; the operation is specified by the operation parameter.

For updating multiple components of F, there is some intricacy if different components of F are located at different places (e.g., for a Yee electric field, E_x and E_y are located at different places within a mesh cell). In this case, the STFunc is evaluated only once (for all components); it is evaluated at the position of the component given by the component parameter of the updater and the field offset of field F. However the components of F that are updated are those given by the parameter writeComponents.

If you want to have f evaluated at the location of each component, then you must use separate STFuncUpdaters for each component (or a FieldMultiUpdater).

The STFuncUpdater introduces the following parameters:

- operation (string) -- either: set $F_j = f$, add $F_j += f$, or multiply $F_j *= f$; this parameter is required.
- dtCoefficients (real vector) -- 2 components [c0 c1]: the function will be multiplied by $(c0+c1 \Delta t)$, where Δt is the current time step (if c1 is not specified it is assumed to be zero). The default is [1.0 0.0]
- component (integer) -- the component j of F to be updated. However, if writeComponents is not [0], the component, along with the field offset of F determines the location (in each cell) where the function f will be evaluated. The default is 0.
- writeComponents (integer vector) -- a list of components j of F that will be updated. f will be evaluated only once (per cell) for all components, and the location within each cell at which f is evaluated is determined by the 'component' parameter and the field offset of F. The default is [0].

The STFunc block is required with an STFuncUpdater:

- STFunc -- an STFunc used to describe f. This block will have other parameters depending on the kind; for instance, kind = expression will then take the parameter ``expression = `` as well. This block is required.

25.3.30 unaryFieldOpUpdater

unaryFieldOpUpdater updates one field F from another field G, multiplied by an STFunc f.

For updating a single component of F, the updater is straightforward; the operation is specified by the operation parameter.

For updating multiple components of F, there is some intricacy if different components of F are located at different places (e.g., for a Yee electric field, E_x and E_y are located at different places within a mesh cell). In this case, the STFunc is evaluated only once (for all components); it is evaluated at the position of the component given by the component parameter of the updater and the field offset of field F. However the components of F that are updated are those given by the parameter writeComponents.

If you want to have f evaluated at the location of each component, then you must use separate STFuncUpdaters for each component (or a FieldMultiUpdater).

The unaryFieldOpUpdater introduces the following parameters:

- operation (string) -- either: set ($F_j=fG_j$), add ($F_j+=fG_j$), multiply ($F_j*=fG_j$), divide ($F_j/=fG_j$). This parameter is required.
- dtCoefficients (real vector) -- 2 components [$c_0 c_1$]. The function f will be multiplied by $(c_0+c_1 \Delta t)$, where Δt is the current time step (if c_1 is not specified it is assumed to be zero). The default vector is [1.0 0.0]
- component (integer) – the component j of F to be updated. However, if writeComponents is not [0], the component, along with the field offset of F determines the location (in each cell) where the function f will be evaluated. The default is 0.
- writeComponents (integer vector) -- a list of components j of F that will be updated. f will be evaluated only once (per cell) for all components, and the location within each cell at which f is evaluated is determined by the 'component' parameter and the field offset of F. The default is [0].
- readComponents (integer vector) -- a list of components of G used to update F. e.g., if writeComponents = [0 1 3] and readComponents = [2 1 3], then $F_0=fG_2$, $F_1=fG_1$ and $F_3=fG_3$, where f is evaluated at the location of component. The default is to set this vector to the same value as writeComponents.
- BumpReadIter (integer vector) – the number of cells (in each simulated dimension) to bump the read-field iterators to their starting offset. The default is [0 0 0].

The STFunc block is required with a unaryFieldOpUpdater:

- STFunc -- an STFunc used to describe f. This block will have other parameters depending on the kind; for instance, kind = expression will then take the parameter ``expression = `` as well. This block is required.

25.3.31 yeeAmpereUpdater

yeeAmpereUpdater updates the electric field component E_j (on center-edge j) according to standard second-order Yee leapfrog algorithm, using B (on faces) and ρJ (a 4-vector, with the last three components $-j$ used for the update)

The yeeAmpereUpdater kind uses the following parameter:

- useVecUpdater (integer) -- if true, the updater will make just one pass through all cells, and in each cell it will either update all components or none. In general, the default component parameter of zero is ideal. In principle, however, one could use a 6-component E-B field and specify component = 3, which for the Faraday update, which would then update components [3 4 5] of the E-B-field. Also, if true, it will ignore all component-dependent region modification flags. If not present, useVecUpdater is considered to be false.

25.3.32 yeeAmpereDielVecUpdater

yeeAmpereDielVecUpdater updates all three (3) components of the electric field according to Ampere's Law, but takes into account the inverse dielectric tensor given by the 6-component invEpsilonField. You can set the inverse epsilon field with the setEpsilonUpdater.

yeeAmpereDielVecUpdater expects the following parameters:

- readFields = [yeeB SumRhoJ invEpsilon] where
 - yeeB is the user-defined yee magnetic field
 - SumRhoJ is the user defined 4 component field including rho and J
 - invEpsilon is a 6-component symmetric epsilon field, {invEps_xx, invEps_yy, invEps_zz, invEps_yz, invEps_zx, invEps_xy} defined with the setEpsilonUpdater.
- writeFields = [yeeE] where yeeE is the user defined yee electric field

Details of the yeeAmpereDielVecUpdater update can be found in the publication:

G R Werner and J R Cary, "A stable FDTD algorithm for non-diagonal, anisotropic dielectrics," J Comput Phys 226, 1085 (2007).

25.3.33 yeeConductorUpdater

Updaters of kind=yeeConductorUpdater solve the equations:

If Ampere:

$$\varepsilon_0 \frac{\partial E_i}{\partial t} = \left(\frac{\nabla \times B}{\mu_0} \right)_i - J_{e,i} - \sigma_{e,i} E_i$$

If Faraday:

$$\mu_0 \frac{\partial B_i}{\partial t} = -\mu_0 (\nabla \times E)_i - J_{m,i} - \sigma_{m,i} B_i$$

eeConductorUpdater uses the parameters:

- emType (string) -- either: faraday or ampere; if ampere, expects writeFields to be the yee electric field and readFields to be the yee magnetic field. readFields can also contain the optional field rhoJ as the standard electric charge/current field; faraday reverses E and B, and if the option rhoJ field is used, it would represent magnetic charge/current. This parameter is required.
- sigma(W) (real) -- specifies the conductivity profile. For Ampere, one should specify σ/ε_0 and for Faraday, one should specify σ/μ_0 ; both have units of frequency. Must either specify sigma, or sigmaX and sigmaY and sigmaZ. This parameter can be specified either as constants given by the parameters sigma(W) or by using an STFunc block. This parameter is required.

Here is an example of an STFunc sigma:

```
<STFunc sigma>  
  kind = expression  
  expression = SIGMA_MAX * ((x - LX_ABS_L)/LX_ABS)^2  
</STFunc>
```

If yeeConductorUpdater finds both a parameter and an STFunc for sigma(W), it throws an exception. If yeeConductorUpdater doesn't find sigmaW then it looks for sigma. Specifying an isotropic conductivity makes computation faster, and specifying a uniform conductivity makes computation much faster.

25.3.34 yeeFaradayUpdater

Updates the magnetic field component B (on center-face j) according to standard second-order Yee leapfrog algorithm, using electric field E (on center edges)

yeeFaradayUpdater uses the following parameter:

- useVecUpdater (integer) -- if true, the updater will make just one pass through all cells, and in each cell it will either update all components or none. In general, the default component parameter of zero is ideal. In principle, however, one could use a 6-component E-B field and specify component = 3, which for the Faraday update, which would then update components [3 4 5] of the E-B-field. Also, if true, it will ignore all component-dependent region modification flags. If not present, useVecUpdater is considered to be false.

25.4 Update steps

MultiFields use UpdateStep and InitialUpdateStep blocks to define how fields are updated. InitialUpdateStep defines an update step to be performed at time = 0; This will also be run after a restore if the alsoAfterRestore flag is set to true. UpdateStep blocks are used for every subsequent update.

Steps to be done every time step are indicated with <UpdateStep nameOfThisStep> and </UpdateStep> brackets. Note, however, that steps are by default ordered as they appear in the input file! However, the order may be overruled with the parameter “updateStepOrder = [step1 step2 ...]” if the user desires.

Update steps take the following parameters:

- updaters (string vector) -- Names of updaters to be executed during this step. This parameter is required.
- messageFields (string vector) – The name of the field to be messaged. Parallel processing often requires that domain boundary fields be messaged between processors. Only one field can be specified. Typically it is one of the updater's writeFields. The dummyUpdater can be used if more than one writeFields needs to be messaged. Periodic boundary conditions will also be applied to only this field.
- toDtFrac (float) -- see below. Either updates the updaters to the same time, or the half way time. This parameter is required.

Steps to be done only at time=0 are indicated with <InitialUpdateStep nameOfThisStep> and </InitialUpdateStep> brackets.

- updaters (string vector) -- Names of updaters to be executed during this step. This parameter is required.
- messageFields (string vector) – The name of the field to be messaged. Parallel processing often requires that domain boundary fields be messaged between processors. Only one field can be specified. Typically it is one of the updater's writeFields. The dummyUpdater can be used if more than one writeFields needs to be messaged. Periodic boundary conditions will also be applied to only this field.
- alsoAfterRestore (option) -- (true/false) specifies whether to perform this update step when restoring from a dump. This is handy, for example, for nodal fields, which can be easily

computed from the Yee fields. The default is false.

InitialUpdateSteps do not use toDtFrac; they are updated with time 0 at initialization; and after restore (if alsoAfterRestore=true), they are updated with the current MultiField time.

A note on toDtFrac: When MultiField is at t_{n-1} and is told to update itself to time t_n (update(t_n)), it will call update(t') for all updaters in the update step, where

$$t' = t_{n-1} (1 - \text{toDtFrac}) + t_n \cdot \text{toDtFrac}$$

In other words, toDtFrac=1 tells MultiField to update the updaters (in the update step) to its same time; whereas, toDtFrac=0.5 updates the updaters to the half-way time. Note that toDtFrac does not refer to a time step, but to an absolute time.

Example: Suppose an updater is in only one update step, with toDtFrac=0.5. During the first update, that updater will be updated by only a half-time step; however, all subsequent updates will be by a whole time step.

Note: FieldUpdaters may appear in more than one update step. It is common in EM simulations with particles to perform the B-update with toDtFrac=0.5 then the E-update with toDtFrac=1, and then the B-update with toDtFrac=1.

Field-particle-overlap: UpdateSteps are divided into two groups: the first group contains all the update steps before the first update step with an updater that requires the field “rhoJ” (here the name must be “rhoJ” exactly); remaining updaters are put in the second group.

Important: the update order for updaters within the same update step is sometimes difficult to predict, because of skin/body separation. The skin updates will all be done in the order in which the updaters are listed, and the same for the body updates. However, some the regions covered by the skin and body updates may differ for different updaters. For example, some updaters cannot be separated: consider a skinnable updater u_1 and a non-skinnable updater u_2 . If “updaters = [u_1 u_2]” then u_1 will be done before u_2 on the skin cells of u_1 , while u_2 will be done before u_1 on the body cells of u_1 (because the entire u_2 update is treated as a skin update). Two updaters can both be skinnable but have different skin and body regions.

When two updaters are of the same kind, and read and write the same fields then they should be done in the order listed.

25.5 FieldMultiUpdater

You can specify a FieldMultiUpdater as a shortcut to specifying multiple FieldUpdaters, FieldMultiUpdater creates as many FieldUpdaters as elements in the “components” parameter. FieldMultiUpdater with components = [0 1 2] creates three FieldUpdaters, which therefore make three passes (one each) through all cells (first pass updates only comp 0, etc.). The cells updated in each pass may differ due to any specified gridBoundary and/or any region-adjusting parameters used (see FieldUpdater).

The FieldMultiUpdater takes the same parameters as the FieldUpdater, with the exception that the component parameter is changed to components (plural):

- components (integer vector) -- The components that should be updated. This parameter is required.

Parameters specified in the FieldMultiUpdater are passed untouched to the new FieldUpdater. The parameter “components” is reduced to one element, the “component” of the FieldUpdater.

When using FieldMultiUpdater it is common to use the component-dependant region-adjusting modification flags as discussed in earlier in this document.

25.6 PmlRegion

PmlRegion is used to describe where (if any) PMLs exist in the simulation. For every PmlRegion, the region will be divided into slabs; for each slab two FieldMultiUpdaters will be created, a Faraday PML updater, and an Ampere PML updater (each with 3 components). The lowerBounds and upperBounds will be given to the FieldMultiUpdaters, as well as the PML “sigma” functions.

The PmlRegion uses the following parameters:

- faradayUpdaterName (string) -- Name of the Yee Faraday updater. (The PML Faraday updater will be put in the same update step.) This parameter is required.
- ampereUpdaterName (string) -- Name of the Yee Ampere updater. This parameter is required.
- eFieldName (string) -- Name of the Yee E field. This parameter is required.
- bFieldName (string) -- Name of the Yee B field. This parameter is required.
- eAuxFieldName (string) -- Name of the PML auxiliary E field. The Field attribute will be created automatically if it needed. The default for this parameter is pmlEAuxField.
- bAuxFieldName (string) -- Name of the PML auxiliary B field. The Field attribute will be created automatically if it needed. The default for this parameter is pmlBAuxField.
- rhoJFieldName (string) -- Name of the density-current field (current at components 1, 2, 3, density at 0)
- rhoJmagFieldName (string) -- name of the density-current field associated with “magnetic” current (untested)
- useUniPml (option) -- whether to use the more efficient uniaxial PML algorithm, for which only one sigmaW is non-zero, when possible. This parameter defaults to true.
- sigmaForm (string) -- A function of “w”, meant to be a form for producing sigmaWn, the conductivity profile. “w” is replaced by x, y, or z, depending on W given by sigmaFactorW. It is multiplied by sigmaFactorW.
- sigmaFormW (string) -- (W is one of {X,Y,Z}). A function of “w”, meant to be a form for producing sigmaWn, the conductivity profile. “w” is replaced by x, y, or z, depending on W given by sigmaFactorW. A string that is multiplied by sigmaFactorW.
- sigmaFactorW (real vector) -- (W is one of {X,Y,Z}). The components of sigmaFactor [sigmaFactorX1 sigmaFactorX2] are used to multiply the expressions of sigmaFormX or sigmaForm for the X1 and X2 slabs. This parameter defaults to [1.0 1.0].

An STFunc block can be used to specify the sigma function directly. The name of the block must correspond to sigmaW or sigmaWn where W is X,Y,Z and n is 0 or 1 (lower or upper). See below for examples.

Sigma may be specified through a combination of sigmaForm, sigmaFormW and STFunc blocks.

The PmlRegion block requires the use of two Region blocks to give the inner and out bounds of the PML region. The inner region describes the region inside PML (usually the region describing the normal Maxwell update; can/may extend beyond the outer region). The outer region describes the region bounding PML (which will be slightly altered for each component). These blocks have the following parameters:

- lowerBounds (integer vector) -- lower bounds of the region. This parameter is required.

- upperBounds (integer vector) -- upper bounds of the region. This parameter is required.

For example in two dimensions the following code block specifies a PML 5 cells thick on the left and right sides of the inner region, and 10 cells thick above and below the inner region. The region will be divided into several slabs (rectangular regions) and updaters will be created for each region.

```
<Region outer>
  lowerBounds = [0 0]
  upperbounds = [110 220]
</Region>
<Region inner>
  lowerBounds = [5 10]
  upperbounds = [105 210]
</Region>
```

As a rule of thumb, you should make the outer region the outer boundary of the PMLs, and the inner region the normal Maxwell update area; generally this will give the correct result. In typical simulations, the inner region may be specified to be the normal Yee update region, and the outer region to be the outer boundary of the PML. This rule applies also when the PML is only a single slab, or is only slabs on the right and left. Before setting itself up, the PML considers the inner region to be the intersection of the outer region with the input-file-specified inner region. Specifying the inner slab might seem a bit tricky in some cases. In VORPAL a PML slab can have a direction; and the simulation gets this direction from the relation between the inner and outer slabs. In the above, for instance, the slab from [105 10] to [110 210], on the right side of the simulation, attenuates waves only in the x direction; the PML algorithm can be computed more efficiently for it than for the upper-right corner [105 0] to [110 10]. Because of this, you must be careful while specifying the inner region when a PML does not surround it completely. For instance, a PML slab just on the right hand side would be described as follows (the inner slab is degenerate in the x direction, and on the left side of the outer slab, indicating that the wave will enter the PML from the left side):

```
<Region outer>
  lowerBounds = [105 0]
  upperbounds = [110 220]
</Region>
<Region inner>
  lowerBounds = [105 0]
  upperbounds = [105 220]
</Region>
```

Or, the following example will do just as well:

```
<Region outer>
  lowerBounds = [105 0]
  upperbounds = [110 220]
</Region>
<Region inner>
  lowerBounds = [0 0]
```

```

    upperbounds = [105 220]
</Region>

```

The conductivity (“sigma”) is an important parameter for the PML. In VORPAL, sigma is actually the electric conductivity divided by ϵ_0 and the magnetic conductivity divided by μ_0 , with units of c/ℓ (1/s). Make sure the conductivity is zero where the inside and the outside regions meet-- where a wave enters the PML region -- and increase the PML in the direction normal to the PML interface (the corners require special treatment, increasing in more than one direction).

As a result, VORPAL needs to use a conductivity function for each direction: -x (left slab), +x (the right slab), -y (bottom slab), +y (top slab), -z (back slab), +z (top slab). The conductivity for each of these slabs is specified by an STFunc block named sigmaX1, sigmaX2, sigmaY1, sigmaY2, sigmaZ1, and sigmaZ2, respectively. For example,

```

<STFunc sigmaX2>
    kind = expression
    expression = 1.0 * c / DX * ( (x - 105*DX)/(5*DX) )\^{ }2
</STFunc>

```

This function, is zero at x-grid number 105, and increases as x increases. (Above, c is meant to be the speed of light and DX the cell-length in the x direction.) Similarly, you could specify

```

<STFunc sigmaY1>
    kind = expression
    expression = 1.5 * c / DY * ( (210*DX - y)/(10*DX) )\^{ }2
</STFunc>

```

You only need the above, for example, when there is a “bottom” slab. sigmaXn should depend only on x, and likewise for other directions; however, it may depend on x, y, z, and t, though this is not recommended.

VORPAL determines which conductivity functions are needed by which slab. For instance, a “side” slab (as opposed to a corner) on the right side needs only sigmaX2. The upper-right corner, however, needs both sigmaX2 and sigmaY2. If the inner region is not correctly specified, VORPAL may use the wrong conductivity functions.

Because so often you will need to use a similar conductivity profile for different directions, VORPAL offers a shortcut. If you specify sigmaForm, VORPAL can make a sigmaWn. For example,

```

sigmaForm = 1.5 * c/DX * w\^{ }2
sigmaFactorX = [ 2.0 0.5 ]

```

can be turned into

```

<STFunc sigmaX1>
    kind = expression
    expression = 2.0 * (1.5 * c/ DX * ( (5*DX - x)/(5*DX) )\^{ }2 )
</STFunc>

```

```

<STFunc sigmaX2>
  kind = expression
  expression = 0.5 * (1.5 * c/ DX * ( (x - 105*DX)/(5*DX) )\^{ }2 )
</STFunc>

```

VORPAL calculates the real coordinates of the PML edge and the PML thickness, and replaces w with a linear function of x (or whatever the normal direction is) that goes from 0 at the edge to 1 at the end; VORPAL then multiplies it by the appropriate factor given by the paramVec \sigmaFactorX .

Should you want to have different functional forms in the X, Y, and Z directions, you may similarly specify a \sigmaFormW where “W” is X, Y, or Z. Again, \sigmaFormX (e.g.) should be a function of the dummy variable “w” (“w” is recognized as the dummy variable if it is neither preceded nor followed by another letter, number, or underscore). Again, \sigmaFactorX is used to allow different factors for $\sigmaX1$ and $\sigmaX2$.

VORPAL first looks for an STFunc σX (this is what the PML updater wants in the end). If there isn't one in the PmlRegion input block, VORPAL looks for an STFunc $\sigmaX1$. If VORPAL doesn't find an STFunc $\sigmaX1$, it looks for a \sigmaFormX , from which it can create $\sigmaX1$, using \sigmaFactorX if present. Otherwise, VORPAL looks for a \sigmaForm , from which it can create $\sigmaX1$, using \sigmaFactorX at present. You can mix all three methods. The resulting $\sigmaX1$ is renamed σX and put in the resulting PML MultiUpdater.

In the end, VORPAL will give an error if a PML updater does not have the σWn STFuncs that it needs.

Note: UniPmlUpdaters are created for the “side” slabs, and PMLUpdaters are created for the corner regions.

The slabs specified in PmlRegion will be expanded and contracted to and from the simulation boundary, depending on the component and emType (Faraday or ampere).

25.7 ScalarDepositors and VectorDepositors

ScalarDepositors and VectorDepositors are described in the following sections.

25.7.1 ScalarDepositor

ScalarDepositors are a new, more flexible way of depositing charge from charged particles in a simulation into depFields, instead of the classical way of depositing charge into the zeroth component of a SumRhoJ field. Note that these should not be used in MultiField block of kind = emMultiField, or errors occur in the simulation.

ScalarDepositor uses the following parameters:

- kind (string) -- The kind of deposition algorithm: choices are areaWeighting (the default), areaWeightingCP (for coordProdGrid), esirk1stOrder, esirk2ndOrder, ..., esirk7thOrder.
- depField (string) -- The scalar depfield defined in a MultiField block that will contain the deposited charge.

An example of an input block is below.

```
<ScalarDepositor chargeDep>  
  kind = areaWeighting  
  depField = myEmField.rho  
</ScalarDepositor>
```

This ScalarDepositor should be referenced in a Species block as

```
<Species electrons>  
  kind = relBorisDF  
  chargeDeps = [ chargeDep ]  
  .  
  .  
  .  
</Species>
```

25.7.2 VectorDepositor

VectorDepositors are a new, more flexible way of depositing current from charged particles in a simulation into depFields, instead of the classical way of depositing current into the last three components of a SumRhoJ field. With VectorDepositors a user can simply employ a VectorDepositor and a three-vector J in an EM PIC simulation instead of using the four- vector SumRhoJ as has been classically used, thus saving on memory. Note that these should not be used in MultiField block of kind = emMultiField, or errors occur in the simulation.

VectorDepositor uses the following parameters:

- kind (string) -- The kind of deposition algorithm: choices are areaWeighting (the default), areaWeightingCP (for coordProdGrid), esirk1stOrder, esirk2ndOrder, ..., esirk7thOrder
- depField (string) -- The vector depfield defined in a MultiField block that will contain the deposited current.

An example of an input block is below.

```
<VectorDepositor~currDep>  
  kind = areaWeighting  
  depField = myEmField.J  
</VectorDepositor>
```

This VectorDepositor should be referenced in a Species block as

```
<Species electrons>  
  kind = relBorisDF  
  currDeps = [ currDep ]  
  .  
  .  
  .  
</Species>
```