SAND2022-16514

SANDIA REPORT

SAND2019-DRAFT
Printed November 15, 2022

Sandia National Laboratories

EMPIRE User Manual

M. T. Bettencourt, K. L. Cartwright, E. C. Cyr, N. Hamlin E. Love, W. McDoniel, D. A. O. McGregor, S. Miller, C. H. Moore, R. P. Pawlowski, E. G. Phillips, T. D. Pointon, G. A. Radke, N. A. Roberds, N. V. Roberts, S. Shields, M. S. Swan, C. D. Turner

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185
Livermore, California 94550

ABSTRACT

This is the user manual for EMPIRE, a simulation code for electromagnetics and plasma physics.

ACKNOWLEDGMENT

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

2. QUICK START

2.1. Running a Simulation

If you already have an executable and your environment is set up, then running a simulation can be as easy as:

```
% /path/to/EMPIRE_PIC.EXE --i=input.yaml
```

This is the most basic invocation and should be avoided as it assumes that all available processors should be used.

For finer control of how the simulation is executed, it is necessary to pass in commands to control Kokkos, the library that is the basis of the parallel data structures. For single-machine parallelization, it is sufficient to call it with

```
% /path/to/EMPIRE_PIC.EXE --kokkos-threads=3 --i=input.yaml
```

This will run EMPIRE with 3 threads.

When running with OpenMP, it is not uncommon for EMPIRE to complain about not having environmental variables properly set and will make suggestions about what and how to set them. Often, setting the following variables will be sufficient:

```
% export OMP_PROC_BIND=spread
% export OMP_PLACES=threads
```

2.2. Dissecting an Input File

This section discusses the basics of the input file and covers a handful of common errors when creating your own input.

2.2.1. What is YAML?

YAML is a recursive acronym standing for *YAML Ain't Markup Language* and is the markup language for input files for EMPIRE. YAML was developed to cover many of the same use cases as XML, but to be more human-readable and clean. YAML uses indentation to manage sections of code, colons to delimit keys and values, and the pound sign # to denote comments.

Here is an example of a YAML file that contains information about dog breeds.

```
# This is a comment
dog breeds:
    # This is a comment
    Chihuahua:
        average weight: 5 lbs # This is a comment
        description: The smallest breed of dog.
    Dobermann:
     average weight: 75 lbs
     description: |
        Dobermann Pinschers were originally developed
        by Karl Friedrich Louis Dobermann.  # This is not a comment
          # This is not a comment

        Dobermanns were bred to be guard dogs.
       # This is a comment and the end of the multi-line string
```

The root block of the tree is `dog breeds` and is at the lowest level of the file. Notice that whitespace in block names, keys, and values throughout the file is perfectly valid. Each element of `dog breeds` must be indented to the same level using spaces, not tabs. The amount of indenting is inconsequential so long as it is at least one space and is consistent within the block (here, the elements are `Chihuahua` and `Dobermann` and the block is `dog breeds`). As an example, the key-value pairs within both the `Chihuahua` block and the `Dobermann` block are indented uniformly within each block, but there is no requirement for consistency between blocks.

Multi-line strings are done by either inserting a pipe | or a greater-than sign >. When the pipe is used, YAML keeps the newlines whereas with the greater-than sign it gobbles the newlines. In either case, YAML expects the multi-line string to continue until it sees a line with less indentation than the first line of the multi-line string. This can cause unexpected behavior if comments are used in or around the multi-line string. This is demonstrated with attempted comments in the `Dobermann:description` multi-line string.

2.2.2. Basic Electrostatic EMPIRE Input

The example input file below contains a basic electrostatic example with a single stationary particle at the origin of a 2D space. To understand the input as a whole, we will discuss each block individually.

```
Electrostatic Example:

  Physics:
    Fields:
      Electrostatic0:
        Regions: [eblock-0_0]

  Mesh:
```

```
  Inline:
    Type: Quad
    Elements: [     32,      32 ]
    Blocks:   [      1,       1 ]
    Start:    [   -1.0,    -1.0 ]
    End:      [    1.0,     1.0 ]

Initial Conditions:
  Andromeda:
    Regions: eblock-0_0
    Zero:
      Fields: [E_Field, ES_POTENTIAL]

Boundary Conditions:
  Guanine:
    Dirichlet:
      Sideset: left
      Field:   ES_POTENTIAL
      Value: 0.0
      Element Block: eblock-0_0
  Cytosine:
    Dirichlet:
      Sideset: right
      Field:   ES_POTENTIAL
      Value: 0.0
      Element Block: eblock-0_0
  Adenine:
    Dirichlet:
      Sideset: top
      Field:   ES_POTENTIAL
      Value: 0.0
      Element Block: eblock-0_0
  Thymine:
    Dirichlet:
      Sideset: bottom
      Field:   ES_POTENTIAL
      Value: 0.0
      Element Block: eblock-0_0

Time Stepping:
  Final Time: 1.e-20
  Number of Timesteps: 10

Mesh History Diagnostics:
  Solver_Fields:
    Solver Field:
```

```
  Mesh History Outputs:
    SingleParticle.exo:
      Diagnostics:
        - Solver_Fields

  Initial Particle Conditions:
    Spherical Cow:
      Point:
        Position: 0., 0.
        Type: e-
        Weight: 1e8

  Particle Dumps:
    SingleParticle.h5part:
```

2.2.2.1. Simulation Root

```
Electrostatic Example:
```

This first line is an arbitrary placeholder that acts as the root level for the blocks that define the simulation. Usually it is set to the name of the simulation, but it is not used inside of EMPIRE.

2.2.2.2. Sublists separated by physics

Every block is divided into sublists according to which physics the options in it are relative to. For example, anything relating to electromagnetics or electrostatics would be listed under the `Fields` sublist, anything relating to particles would be under the `Particles` sublist, and anything relating to fluids would be under the `Fluids` sublist. Input for pure EMPIRE runs will only see the `Fields` sublists because no particles or fluid are involved in these problems.

2.2.2.3. Physics Definition

```
  Physics:
    Fields:
      Electrostatic0:
        Regions: [eblock-0_0]
```

This block defines the physics desired in the problem. The first line here is the `Fields` sublist delimiter, showing that everything under it is related to electromagnetic or electrostatic fields in some way. Under the `Fields` sublist there are a number of options that can be defined, along with the `Electrostatic` and `Electromagnetic` equations sublists. Details of the Physics block can be found in Chapter 5

2.2.2.4. Inline Mesh Generator

```
Mesh:
  Inline:
    Type: Quad
    Elements:   [     32,      32 ]
    Blocks:     [      1,       1 ]
    Start:      [   -1.0,    -1.0 ]
    End:        [    1.0,     1.0 ]
    Processors: [      4,       2 ]
```

While EMPIRE can import meshes in various formats, the easiest way to get a simulation set up is to use the built-in inline mesh generator. This example creates a 2D mesh comprised of a single block (from which `eblock-0_0` comes from in the section above) with an X- and Y-extent from -1 meter to +1 meter. Detailed descriptions of mesh options can be found in Chapter 4

2.2.2.5. Initial Conditions

```
Initial Conditions:
  Andromeda:
    Regions: eblock-0_0
    Zero:
      Fields: [E_Field, ES_POTENTIAL]
```

The `Initial Conditions` block may contain many elements with arbitrary, unique names, where each child sets the initial conditions for a single block of the mesh. In this example, the arbitrary, unique name for the block defining the initial conditions in `eblock-0_0` was chosen the be `Andromeda`, the name of the closest galaxy to the Milky Way. Inside the `Andromeda` block, it must define the `Region` key with the name of the particular mesh block and then define the fields in that region. Constant fields (as opposed to fields defined by runtime compiled (RTC) functions) should be defined in a `Constant` block. Details of the various options are listed in 7

2.2.2.6. Boundary Conditions

```
Boundary Conditions:
  Guanine:
    Dirichlet:
      Sideset: left
      Field:   ES_POTENTIAL
      Value: 0.0
      Element Block: eblock-0_0
  Cytosine:
    Dirichlet:
      Sideset: right
      Field:   ES_POTENTIAL
```

```
          Value: 0.0
          Element Block: eblock-0_0
      Adenine:
        Dirichlet:
          Sideset: top
          Field:    ES_POTENTIAL
          Value: 0.0
          Element Block: eblock-0_0
      Thymine:
        Dirichlet:
          Sideset: bottom
          Field:    ES_POTENTIAL
          Value: 0.0
          Element Block: eblock-0_0
```

Each child in the `Boundary Conditions` block can take an arbitrary, unique name. For this example, the names are the four DNA nucleobases. Their children must have a specific name depending on the type of boundary condition that is desired (here, it's `Dirichlet`). The children of the specific type of boundary condition requires all of its children be set and named correctly. The remaining usage is discernable from similar usage and descriptions in previously discussed sections. Details of the various boundary condition options are provided in Chapter 8.

2.2.2.7. Time Stepping

```
  Time Stepping:
    Final Time: 1.e-20
    Number of Timesteps: 10
```

The `Time Stepping` block is required but only has two elements which are both optional. All simulations start at time zero and proceed until the time given by `Final Time`. If the `Final Time` element is missing, a default value of zero is used. The simulation will advance up to the `Final Time` in the number of timesteps equal to what is given in `Number of Timesteps`. If that is missing, a default value of zero is used. Detailed descriptions of the Time Stepping options can be found in Chapter 6.

2.2.2.8. Field Output

```
  Mesh History Diagnostics:
    Solver_Fields:
      Solver Field:

  Mesh History Outputs:
    SingleParticle.exo:
      Diagnostics:
        - Solver_Fields
```

The `Mesh History Diagnostics` and `Mesh History Outputs` blocks are optional. Details on diagnostics and output can be found in Chapter 12.

3. USER DATA

The User Data section has miscellaneous settings not involved with the actual physics of the problem

```
User Data:
  Time Barriers: [bool]
  Timing File Name: [string]
  Dump Timers: [Stride sublist]
  Kokkos Timers: [string]
```

`Time Barriers` Puts barriers in key points and then reports the time waiting at the barrier, useful for determining if the run is not load balanced.

`Timing File Name` Name of file to dump timing information.

`Dump Timers` Provide a stride sublist to dump timing at interval defined by the stride.

`Kokkos Timers` String which defines what kokkos stuff to dump in the integrated timer file or memory file.

- `empty` only regions are dumps
- `None` means nothing is added
- `Parallel Functions` parallel for, scan and reduce are dumped
- `Views` allocation of views are timed.
- `Memory` allocations and deallocations are dumped to a file and size information
- `All` all of the above are done

4. MESH

The mesh block specifies the mesh the problem is run on. Currently EMPIRE supports either simplex (i.e. triangle or tetrahedron) or tensor product (i.e. quadrilateral and hexahedron) elements. EMPIRE does not support meshes with more than one type of element.

EMPIRE offers three options – externally provided exodusII files (typically denoted with `.g` or `.gen` when used for input), a very simple inline mesher capable of meshing brick-like domains with either uniform tensor elements or simplices, or pamgen which provides tensor product meshes for a variety of shapes and supporting multi-block. Note that pamgen does not provide the capability to cut a tensor product mesh into a simplex mesh.

4.1. Mesh Options

4.1.1. Exodus Mesh

An exodus mesh can be read in using the keyword Exodus as follows.

```
  Mesh:
    Exodus:
      File: [string]
```

`File` name of the exodus file being read in.

For parallel runs the mesh does not need to be decomposed and Zoltan's Recursive Inertial Bisection (RIB) will be used. If a decomposition matching the MPI parameters is available, this will be used instead.

Expert option - control decomposition method. First determine what methods are available

```
$ io_info --config
....
Supported decomposition methods:
        LINEAR, BLOCK, CYCLIC, RANDOM, RCB, RIB, HSFC
```

Then you can set the decomposition method as follows:

```
export IOSS_PROPERTIES=DECOMPOSITION_METHOD=RCB
```

4.1.2. Inline Mesh

```
Mesh:
  Inline:
    Type:        [string]
    Elements:    [   [int]   ,  [ int]   (optional, [int]    ) ]
    Blocks:      [   [int]   ,  [int]    (optional, [int]    ) ]
    Start:       [   [double],  [double] (optional, [double]) ]
    End:         [   [double],  [double] (optional, [double]) ]
    Processors:  [   [int]   ,  [int]    (optional, [int]    ) ]
```

`Type` requires a value of either `Quad` or `Hex` when the simulation is either 2D or 3D, respectively.

`Elements` list states the number of elements in the X-direction, Y-direction, and (if applicable) the Z-direction.

`Blocks` list states the number of blocks in the X-direction, Y-direction, and (if applicable) the Z-direction with the block numbering starting at zero. Each block will have the number of elements specified in `Elements`. The first block created is named `eblock-0_0` for 2D or `eblock-0_0_0` for 3D.

`Start` list contains the lower limits of the mesh extent for the X-, Y-, and (if applicable) Z-directions.

`End` list contains the upper limits of the mesh extent for the X-, Y-, and (if applicable) Z-directions.

`Processors` list contains the user specified distribution of processors. The length of this list must match the dimension of the mesh, and the product of all the entries must equal the number of processors for the run. If the `Processors` block is excluded then the default is a one dimensional decomposition in the direction with the most elements.

`Elements`, `Blocks`, `Start`, and `End` keys each require a list value of the same length as the number of dimensions.The mesh that is created has some sideset names already set: `left` and `right` for low and high X-faces, `bottom` and `top` for low and high Y-faces, and `back` and `front` for low and high Z-faces (if applicable).

4.1.3. Pamgen Mesh

4.2. Electromagnetic and Electrostatic Accuracy

EMPIRE's support of high order basis functions is currently limited. As such, the mesh determines the expected order of accuracy in space for electromagnetics. On simplex meshes we expect the electromagnetics to be first order accurate. On uniform tensor product meshes (e.g. squares or cubes) one can expect second order accuracy in the electric field. Electrostatics are expected to always exhibit second order accuracy in the voltage.

5. PHYSICS

5.1. Fields

The following yaml excerpt shows all possible options to be defined in the `Fields` sublist along with their default values (if the options are not specified, the following defaults will be used):

```
Physics:
  Fields:
    Classic Interface:          [bool]    (default false)
    Electric Field Push:        [string]  (default by physics)
    Nodal E Field Correction:   [bool]    (default false)
    Initial Poisson Solve:      [bool]    (default false)
    Implicit Particle Update:   [bool]    (default false)
    Explicit Field Update:      [bool]    (default false)
    Basis Order:                [int]     (default 1)
    Pinning:                    [int]     (default do nothing)
    Electromagnetic:            [sublist] (special)
    Electrostatic:              [sublist] (special)
```

The compatibility of these options is listed in Table 5-1.

Keyword	EM	ES
`Classic Interface`	✓	✓
`Electric Field Push`	✓	✓
`Nodal E Field Correction`	✓	✓
`Implicit Particle Update`	✓	✓
`Explicit Field Update`	✓	X
`Basis Order`	X	✓
`Pinning`	X	✓
`Electromagnetic`	✓	X
`Electrostatic`	X	✓

Table 5-1. Compatibility of Fields Physics options by type: Electromagnetics (EM) and Electrostatics (ES). X represents incompatible while ✓represents compatibility.

5.1.1. Classic Interface

The `Classic Interface` option forces the code to use the original, almost deprecated, solver interface. At this point, the only feature supported in the classic interface, and not the newer, more optimized interfaces, is second order electrostatics.

5.1.2. Electric Field Push

The `Electric Field Push` option has three choices: `Lumped Nodal`, `Consistent Nodal`, and `Edge Based`. `Edge Based` is the default for electromagnetics, and `Lumped Nodal` is the default for electrostatics.

5.1.3. Nodal E Field Correction

The `Nodal E Field Correction` option is only applicable if `Electric Field Push` is set to one of the two nodal options. This option uses Gauss' law to correct the nodal E values on the boundaries to be second order accurate.

5.1.4. Initial Poisson Solve

The `Initial Poisson Solve` option is used if the initial particle load does not have a pointwise zero charge density. This option allows EMPIRE to do an initial electrostatic solve to compute the E field associated with the initial charge density and sets it as the initial condition. This option should eliminate ghost charge associated with initial fill of particles.

5.1.5. Implicit Particle Update

The `Implicit Particle Update` option chooses whether the particles are updated implicitly or not. Note that the algorithm for implicit pic is different by physics type.

5.1.6. Explicit Field Update

The `Explicit Field Update` option chooses whether the fields are updated explicitly or not.

5.1.7. Basis Order

The `Basis Order` option is an integer that specifies the order of the basis functions used in the electromagnetics or electrostatics solver. At the moment, only basis orders 1 and 2 are supported for electrostatics, and only basis order 1 is supported for electromagnetics.

5.1.8. Pinning

The `Pinning` option only applies to electrostatic problems. This option is an int that specifies the index of the node to use as a reference value for the entire problem. There is no default here, if no pinning is specified the option is left empty. Pinning should be used when solving an electrostatic problem where a voltage is not specified on the boundary.

5.1.9. Electromagnetic and Electrostatic Blocks

The remaining possible inputs are the `Electrostatic` and `Electromagnetic` sublists. Each sublist represents a set of regions with shared material parameters (permittivity, permeability, and conductivity) for electrostatic or electromagnetic field solve. Specifying the material parameters are optional, and if nothing is specified in the input the defaults will be used. The `Regions` parameter is a list (surrounded by brackets, comma-separated) of blocks to apply the physics and parameters specified in the list to. The following yaml excerpts show all possible options for an `Electrostatic` or `Electromagnetic` sublist, and the relevant defaults:

```
Physics:
  Fields:
    ...
    Electromagnetic_Unique_Suffix:
      Regions:                [string] (required)
      Relative Permittivity:  [double] (default 1.0)
      Relative Permeability:  [double] (default 1.0)
      Conductivity:           [double] (default 0.0)
      MaterialModel:          [sublist] (special)
```

```
Physics:
  ...
  Fields:
    Electrostatic_Unique_Suffix:
      Regions:                [string] (required)
      Relative Permittivity:  [double] (default 1.0)
```

An arbitrary number of these sublists are allowed as long as they have unique names and only one of `Electrostatic` and `Electromagnetic` are used at a time. Each one of these sublists must be uniquely named using the proper keyword (Electrostatic or Electromagnetic) as a prefix for the name, with some unique suffix. Even if only one of these sublists are described, an arbitrary suffix must accompany it. For example, `Electrostatic0` along with `Electrostatic box` are allowed, but `Electrostatic0` along with `Electromagnetic0`, or just `Electrostatic` are not.

`Regions` is a list of elements governed by the same material parameters.

`Relative Permittivity` determines the permittivity of the block. Presently only constant values by block are supported. The permittivity is determined by the dimensionless relative permittivity ε_r by

$$\varepsilon = \varepsilon_r \varepsilon_0 \tag{5.1}$$

where ε_0 is the vacuum permittivity.

`Relative Permeability` determines the permeability of the block. Presently only constant values by block are supported. The permeability is determined by the dimensionless relative permittivity μ_r by

$$\mu = \mu_r \mu_0 \tag{5.2}$$

where μ_0 is the vacuum permeability.

`Conductivity` determines the electrical conductivity of the block. Presently only constant values by block are supported. The conductivity is specified supplying a value in the units of S/m (Ω^{-1}/m).

`MaterialModel` is a sublist that specifies properties for specific material models. More details about the possible material models and how to specify them can be found in 5.1.11. If no material model is set, then a simple dielectric material given by

$$D = \varepsilon_r \varepsilon_0 E \tag{5.3}$$

$$H = \frac{1}{\mu_r \mu_0} B \tag{5.4}$$

$$J = \sigma E \tag{5.5}$$

is assumed with parameters determined by `Relative Permittivity`, `Relative Permeability`, and `Conductivity`.

5.1.10. Zero-Dimensional Block

Instead of any `Electrostatic` or `Electromagnetic` sublists, a `Zero-Dimensional` sublist can be defined. Instead of setting parameters for the field solve, the purpose of this is to allow the user to prescribe spatially-uniform but potentially time-varying fields for use in zero-dimensional simulations. Use of this sublist also has effects on `EMPIRE-PIC.exe` that support zero-dimensional simulation. In particular it disables particle movement. Particle velocities will still be updated according to the local fields and collisions, but their positions will not update according to their velocities, enabling multiple independent realizations to be run in separate elements. Particle boundary conditions are not disabled, and in general should create particles which stack up on the domain boundary. Because collisions are agnostic as to particle position within elements this can be used for certain quasi-0D problems. The following yaml excerpt shows all possible options for a `Zero-Dimensional` sublist:

```
Physics:
  Fields:
    ...
  Zero-Dimensional_Unique_Suffix:
    Regions:                 [string] (required)
    Initial E Field:         [double x3] (default [0.0, 0.0, 0.0])
    Initial B Field:         [double x3]  (default [0.0, 0.0, 0.0])
    Final E Field:           [double x3]  (default [Initial E Field])
    Final B Field:           [double x3]  (default [Initial B Field])
    Field Rise Time:         [double] (default -1.0)
```

The sublist should have a suffix, and use of more than one `Zero-Dimensional` sublist or use of a `Zero-Dimensional` with `Electrostatic` or `Electromagnetic` sublists is not supported.

`Regions` is a list of elements governed by the same material parameters.

`Initial E Field` is a vector giving the initial value of the electric field (V/m).

`Initial B Field` is a vector giving the initial value of the magnetic field (T).

`Final E Field` is a vector giving the final value of the electric field (V/m).

`Final B Field` is a vector giving the final value of the magnetic field (T).

`Field Rise Time` is the time over which the fields will linearly adjust from their initial values to their final values (s).

`Regions` is the only required parameter, and by default there will be no fields. If an initial field is set with no corresponding final field, then the field will be constant in time. If a final field is set, then `Field Rise Time` must be set to a non-negative number. The field will linearly adjust from its initial value to its final value over the rise time, and will be constant at its final value thereafter. If any field parameters are set here then field boundary conditions will be overridden.

5.1.11. Material Models

For electromagnetic physics blocks it is possible to specify a material model to exist on those same element blocks. Generically the material models will be of the form

$$D = \varepsilon E + P(E) \tag{5.6}$$

$$H = \mu^{-1}(B - M(B)) \tag{5.7}$$

$$J = \sigma E + J(E,B). \tag{5.8}$$

Only one material model can be specified for each electromagnetic block, and the material model will apply to the same set of element blocks, set in `Regions`, as the electromagnetic block. If no material model is set, then the material parameters are determined solely by the `Relative Permittivity`, `Relative Permeability`, and `Conductivity`. If the same material model with different parameters is desired on different blocks, then separate electromagnetic blocks must be specified. The current, polarization, and magnetization can be visualized using the Quadrature Data diagnostic, see Section 12.1.2.8.

```
Physics:
  Fields:
    ...
    Electromagnetic_Unique_Suffix:
      Regions:                          [string] (required)
      Relative Permittivity:            [double] (default 1.0)
      Relative Permeability:            [double] (default 1.0)
      Conductivity:                     [double] (default 0.0)
      MaterialModelName_Unique_Suffix:  [sublist] (special)
```

5.1.11.1. *Isotropic Cold Plasma*

The Isotropic Cold Plasma is a current material model given by

$$\dot{J} = \varepsilon \omega_p^2 E - \nu_c J, \tag{5.9}$$

where ω_p is the plasma angular frequency and ν_c is a collision constant. Both of these parameters can vary spatially, but should be constant in time. The permittivity ε is determined by the `Relative Permittivity` of the electromagnetic block. The initial plasma current can be set using the sublist `Current Initial Condition`. The parameters ω_p and ν_c are set using the sublists `Omega Plasma` and `Nu Collision`. Presently the initial conditions and parameters can be set using RTC functions or with file inputs. An RTC specifying the current initial conditions should set the variables **jx**, **jy**, and **jz**. For two dimensional problems **jz** can be omitted. The variables **omega_plasma** and **nu_collision** should be set in their respective function strings. Files to set the parameters generally come from reading CGNS files. The parameters ω_p and ν_c can be visualized with the Quadrature Data diagnostic, see section 12.1.2.8. The plasma model differential equation has the option to be stepped using an implicit method, Crank Nicolson, or with an explicit method, Heun method/SSPERK22. The default is to use the implicit method.

```
Physics:
  Fields:
    ...
    Electromagnetic_Unique_Suffix:
      Relative Permittivity: [double] (default 0.0)
      ...
      Isotropic Cold Plasma_Unique_Suffix:
        Explicit Timestepping: [bool] (default false)
        Current Initial Condition:
          Function : [string] (default empty)
          File Name: [string] (default empty)
        Omega Plasma:
          Function : [string] (default empty)
          File Name: [string] (default empty)
        Nu Collision:
          Function : [string] (default empty)
          File Name: [string] (default empty)
```

6. TIME STEPPING

6.1. General Inputs

There are several basic options for the Time Stepping block which are used regardless of physics. You need two of the following three options.

```
Time Stepping:
  Final Time:           [double] (required 2 of 3)
  Number of Timesteps: [int]    (required 2 of 3)
  Timestep Size:        [double] (required 2 of 3)
```

EMPIRE presently requires a start time of 0. Number of time steps determines the timestep size taken by the method.

6.2. Fields Timestepping

6.2.1. Tempus Path

The Tempus path for electromagnetics is engaged as follows

```
Time Stepping:
  ...
  Fields:
    Use Tempus: true (required)
    Tempus Field Stepper: [string] (required)
```

The supported Tempus Field Stepper options are given below.

`Friedman` A second order, single stage, dissipative method with an adjustable damping parameter. This method has a third order dissipation which can smooth out under-resolved high frequency noise. The stability of the method is determined by the damping parameter, typically denoted as θ. The method is θ is A-stable for $\theta \in [0,2]$ and L-stable for $\theta = 1$. Typically $\theta \leq 0.2$ is chosen. The default value is $\theta = 0.1$.

`RK Implicit Midpoint` A second order, single stage, energy conserving option which requires all frequencies to be resolved or else results in oscillations.

`Backward Euler` A first order, single stage, L-stable method.

`SDIRK 2 Stage 2nd order` A second order, two stage, L-stable method. Note that the two stages will make each EM time step roughly twice as expensive as Implicit Midpoint, Friedman, or Backward Euler.

The integrators `Friedman` and `RK Implicit Midpoint` are the preferred options for the Tempus path. `RK Implicit Midpoint` should be used in situations where the frequency content is both linearly bounded and well resolved. `Friedman` should be used if either of those assumptions is violated. Both `Backward Euler` and `SDIRK 2 Stage 2nd order` are not recommended for typical use cases. If `Use Tempus` is set to true, EMPIRE defaults to the Tempus implementation of Friedman.

Additional time integrators can be accessed by setting the Tempus Field Stepper per Tempus' documentation. These options are not tested and therefore considered unsupported.

The Friedman damping parameter can be adjusted as follows:

```
Time Stepping:
  ...
  Fields:
    Use Tempus: true
    Tempus Field Stepper: Friedman
    Friedman Theta: [double] (optional) (default 0.1)
```

6.2.2. Non-Tempus Path

The default time integrator for Electromagnetics is Crank-Nicolson implemented without Tempus. This can be explicitly requested by adding

```
Time Stepping:
  ...
  Fields:
    Use Tempus: false (default false)
```

to the block. Crank-Nicolson is theoretically equivalent to the Tempus' `RK Implicit Midpoint` integrator. A non-Tempus Friedman can also be activated by setting `Non-Tempus Friedman` to true. For example:

```
Time Stepping:
  ...
  Fields:
    Use Tempus:          false
    Non-Tempus Friedman: true
    Friedman Theta:      [double] (optional) (default 0.1)
```

7. INITIAL CONDITIONS

7.1. Fields

Initial conditions, Fields has a top level default `Regions`.

```
Initial Conditions:
  Fields:
    Regions: [element block array] (default empty)
    ...
```

This top level `Regions` will be the default for all child blocks beneath it. If you overspecify initial conditionss – i.e. declare multiple initial conditions for a given field in overlapping Regions the parser will error out.

7.1.1. Zero Conditions

Zero initial conditions are the default conditions for **E** and **B** fieds. If a `Regions` is specified at the Fields level then by default Zero blocks for `E_Field` and `B_Field` are populated with that Field's level region data. To explicitly call a zero condition use the block

```
Initial Conditions:
  Fields:
    Zero:
      Fields: [arry of field names or single field name] (required)
      Regions: [element block array] (required)
```

The `Field` input for Zero conditions accepts an single string or an array. The entries must be `E_Field` or `B_Field`

7.1.2. RTC Conditions

To specify arbitrary initial conditions use the RTC block.

```
Initial Conditions:
  Fields:
    RTC:
      Regions: [element block array]   (see below)
      Field: [see below] (required)
      Function: [RTC Function]       (required)
  ...
```

`Regions`: A valid array of element block string is required.

`Field`: To set an electric field condition use `E_Field_Vector`. To set a magnetic flux condition, in 3D use `B_Field_Vector` and in 2D use `B_Field`.

`Function`: This RTC provides has the standard spatial coordinates (`xin`,`yin`,`zin`). Function values should match the above field name. Entries `E_Field_Vector` and `B_Field_Vector` are 3D arrays. Entry `B_Field` is a scalar.

7.1.3. Electrostatics

Electrostatics does not have a notion of initial conditions as the electric field responds quasi-statically to a given charge.

8. BOUNDARY CONDITIONS

8.1. Electromagnetic Boundary Conditions

This section describes electromagnetic boundary conditions which are available in EMPIRE. Certain boundary conditions are mutually exclusive while others can be used in conjunction with another boundary condition. These relationships are summarized in Table 8-1.

	PMC	$\mathbf{E}\times\mathbf{n}$	$\mathbf{H}\times\mathbf{n}$	Z	WL	TL	$\mathbf{B}\cdot\mathbf{n}$
PEC	X	X	X	X	X	X	✓
PMC		X	X	X	X	X	✓
$\mathbf{E}\times\mathbf{n}$			X	X	X	X	✓
$\mathbf{H}\times\mathbf{n}$				✓	X	X	✓
Z					X	X	✓
WL						X	✓
TL							X

Table 8-1. Compatibility of Electromagnetic Boundary Conditions. Entries marked with an (X) on this table are mutually exclusive. Entries marked with (✓) can have multiple boundary conditions prescribed. Here PEC – Perfect Electric Conductor, PMC – Perfect Magnetic Conductor, Z – impedance, WL – TEM Wave Launch, TL – Transmission Line.

8.1.1. Perfect Magnetic Conductor

Perfect magnetic conductor (PMC) boundary conditions are the natural boundary condition for EMPIRE's electromagnetic discretization. If no other boundary condition is specified then the method will enforce

$$\mathbf{H}\times\mathbf{n}=\mathbf{0} \tag{8.1}$$

on that boundary. A typical theoretical description of a PMC includes the condition $\mathbf{D}\cdot\mathbf{n}=0$. This is not directly enforced by our discretization. Instead we enforce the involution condition

$$\frac{\partial}{\partial t}\mathbf{D}\cdot\mathbf{n}+\mathbf{J}\cdot\mathbf{n}=0. \tag{8.2}$$

Thus $\mathbf{D}\cdot\mathbf{n}=0$ can be guaranteed for all time only if $\mathbf{D}(0)\cdot\mathbf{n}=0$ and if $\mathbf{J}(t)\cdot\mathbf{n}=0$ for all time. In order to enforce a PMC condition the following options need to be set:

```
Boundary Conditions:
  Fields:
    ...
    PMC Unique_PMC_Name_1:
      Sidesets:  [sideset ID string<, ...>] (required)
      Regions:   [region name<, ...>]       (optional)
    ...
```

`Unique_PMC_Name_1` The boundary condition requires a unique identifier
`Sidesets` List of sidesets to which the condition will be applied (brackets required)
`Regions` Masks the boundary condition to a set of regions if a sideset spans more than one boundary (brackets required)

8.1.2. Perfect Electric Conductor

Perfect electric conducting boundary conditions are typically used to model a metal wall. The mathematical formulation of the PEC is to enforce

$$\mathbf{E} \times \mathbf{n} = \mathbf{0} \tag{8.3}$$

on the boundary. In order to enforce a PEC condition the following options need to be set:

```
Boundary Conditions:
  Fields:
    ...
    PEC Unique_PEC_Name_1:
      Sidesets:  [sideset ID string<, ...>] (required)
      Regions:   [region name<, ...>]       (optional)
    ...
```

`Unique_PEC_Name_1` The boundary condition requires a unique identifier
`Sidesets` List of sidesets to which the condition will be applied (brackets required)
`Regions` Masks the boundary condition to a set of regions if a sideset spans more than one boundary (brackets required)

Similar to the PMC, for the PEC it is typical to wonder if $\mathbf{B} \cdot \mathbf{n} = 0$. This is enforced through an involution condition

$$\frac{\partial}{\partial t}\mathbf{B} \cdot \mathbf{n} = 0. \tag{8.4}$$

Thus to guarantee $\mathbf{B} \cdot \mathbf{n} = 0$ on a PEC boundary, initial conditions must be chosen consistent with this constraint.

8.1.3. Arbitrary $\mathbf{E} \times \mathbf{n}$

Arbitrary tangential **E** boundary conditions can be enforced as follows:

```
Boundary Conditions:
  Fields:
    ...
    E Tangent Unique_ETAN_Name_1:
      Sidesets:  [sideset ID string<, ...>] (required)
      Regions:   [region name<, ...>]       (optional)
      Function:  [RTC Function string]      (required)
    ...
```

`Unique_ETAN_Name_1` The boundary condition requires a unique identifier
`Sidesets` List of sidesets to which the condition will be applied (brackets required)
`Regions` Masks the boundary condition to a set of regions if a sideset spans more than one boundary (brackets required)
`Function` Determines the `E_Field` imposed on the boundary. This standard RTC provides `xin`,`yin`, and, if applicable, `zin` as well as `time`.

This enforces the condition

$$\mathbf{E} \times \mathbf{n} = \mathbf{E}_{\text{EXT}} \times \mathbf{n} \tag{8.5}$$

on the boundary. Here $\mathbf{E}_{\text{EXT}}$ is the data the user provides through the RTC function. Note that any normal components of the electric field will not be set by this condition. This boundary condition is essential and may be referred to in literature as a “hard source."

The RTC string is a C function which sets the array `E_Field` as a function of `time, xin, yin, zin`. The array `E_Field` requires two entries in 2D and three entries in 3D. An example follows:

```
        Function: |
          double r2 = xin*xin+yin*yin+zin*zin;
          double a = 11.8/(log(4.0)-log(2.0));
          E_Field[0] = -a*xin/r2;
          E_Field[1] = -a*yin/r2;
          E_Field[2] = 0.0;
```

8.1.4. Arbitrary $\mathbf{H} \times \mathbf{n}$

An arbitrary tangential **H** boundary condition can be imposed as follows:

```
Boundary Conditions:
  Fields:
    ...
    H Tangent Unique_HTAN_Name_1:
      Sidesets:  [sideset ID string<, ...>] (required)
      Regions:   [region name<, ...>]       (optional)
      Function:  [RTC Function string]      (required)
  ...
```

`Unique_HTAN_Name_1` The boundary condition requires a unique identifier

`Sidesets` List of sidesets to which the condition will be applied (brackets required)

`Regions` Masks the boundary condition to a set of regions if a sideset spans more than one boundary (brackets required)

`Function` Determines the `H_Field` imposed on the boundary. This standard RTC provides `xin`,`yin`, and, if applicable, `zin` as well as `time`.

This enforces the condition

$$\mathbf{H} \times \mathbf{n} = \mathbf{H}_{\mathrm{EXT}} \times \mathbf{n} \tag{8.6}$$

on the boundary. Here $\mathbf{H}_{\mathrm{EXT}}$ is the data the user provides through the RTC function. Note that any normal components of the imposed magnetic field will not be set by this condition. This boundary condition is sometimes called a “soft source" in the literature.

The RTC string is a C function which sets the array `H_Field` as a function of `time, xin, yin, zin`. The array `H_Field` requires two entries in 2D and three entries in 3D. An example follows:

```
        Function: |
          double r2 = xin*xin+yin*yin+zin*zin;
          double a = 11.8/(log(4.0)-log(2.0));
          H_Field[0] = -a*xin/r2;
          H_Field[1] = -a*yin/r2;
          H_Field[2] = 0.0;
```

8.1.5. Arbitrary $\mathbf{B} \cdot \mathbf{n}$

The normal component of the magnetic flux density can be imposed as follows:

```
Boundary Conditions:
  Fields:
    ...
    B Normal Unique_BFLUX_Name_1:
      Sidesets:  [sideset ID string<, ...>] (required)
      Regions:   [region name<, ...>]       (optional)
      Function:  [RTC Function string]      (required)
    ...
```

`Unique_BFLUX_Name_1` The boundary condition requires a unique identifier

`Sidesets` List of sidesets to which the condition will be applied (brackets required)

`Regions` Masks the boundary condition to a set of regions if a sideset spans more than one boundary (brackets required)

`Function` Determines the `B_Field` imposed on the boundary. This standard RTC provides `xin`,`yin`, and, if applicable, `zin` as well as `time`.

This enforces the condition

$$\mathbf{B} \cdot \mathbf{n} = \mathbf{B}_{\mathrm{EXT}} \cdot \mathbf{n} \tag{8.7}$$

on the boundary. Here $\mathbf{B}_{\text{EXT}}$ is the data the user provides through the RTC function. Note that any tangential components of the imposed magnetic field will not be set by this condition.

The RTC string is a C function which sets the array `B_Field` as a function of `time, xin, yin, zin`. As this boundary condition sets `B_Field` on boundary faces, it does nothing in 2D. The array `B_Field` requires three entries in 3D. An example follows:

```
      Function: |
        double r2 = xin*xin+yin*yin+zin*zin;
        double a = 11.8/(log(4.0)-log(2.0));
        B_Field[0] = -a*xin/r2;
        B_Field[1] = -a*yin/r2;
        B_Field[2] = 0.0;
```

This boundary condition should be used with EXTREME CAUTION. Under most circumstances the normal magnetic flux is a degree of freedom! If the problem is initialized with a magneto-static field and the boundary values of field must be kept fixed, then this may be an appropriate boundary condition.

This boundary condition can be used in conjunction with an $\mathbf{E} \times \mathbf{n}$ boundary condition. It can be used in conjunction with an $\mathbf{H} \times \mathbf{n}$ condition but great care must be taken to make sure the two constraints are consistent.

8.1.6. Impedance

EMPIRE includes a simple impedance boundary condition. This boundary condition functions only in electromagnetics. This model truncates the domain by a dielectric interface. This truncation is only first order accurate. If one selects the impedance of the boundary to match the impedance of the domain then this boundary condition is a simple open or absorbing boundary condition for electromagnetic fields. This boundary condition enforces

$$\mathbf{H} \times \mathbf{n} = Z^{-1}\mathbf{n} \times (\mathbf{E} \times \mathbf{n}). \tag{8.8}$$

Here Z is the impedance on the boundary given by

$$Z = Z_r\sqrt{\frac{\mu_0}{\varepsilon_0}} = \sqrt{\frac{\mu_r\mu_0}{\varepsilon_r\varepsilon_0}} \tag{8.9}$$

where Z_r is the relative impedance.

```
Boundary Conditions:
  Fields:
    ...
    Impedance Unique_IBC_Name_1:
      Sideset:              [sideset ID string] (required)
      Relative Impedance:   [double]            (default = 1.0)
    ...
```

`Unique_IBC_Name_1` The boundary condition requires a unique identifier

`Sideset` Sideset to which the condition will be applied (brackets required)
`Relative Impedance` Determines the relative impedance at the boundary

Note that this boundary condition can be used to truncate a load for a powerflow simulation. However, at present we do not support dynamic impedance on these boundaries.

The impedance condition can be used in conjunction with a $\mathbf{H} \times \mathbf{n}$ condition to impose what is known in the literature as a Robin or mixed condition:

$$\mathbf{H} \times \mathbf{n} = \mathbf{H}_{\text{EXT}} \times \mathbf{n} + Z^{-1}\mathbf{n} \times (\mathbf{E} \times \mathbf{n}) \tag{8.10}$$

This can be done by imposing both an arbitrary $\mathbf{H} \times \mathbf{n}$ condition and an impedance condition on the same boundary. An example is given as follows:

```
Boundary Conditions:
  Fields:
    ...
    Impedance Robin_Impedance_1:
      Sideset: sideset_101
      Relative Impedance: 2.3
    H Tangent Robin_Source_1:
      Sidesets: [sideset_101]
      Function: |
        t0 = 1e-9;
        tscale = 1.e10;
        iscale = 2.e6;
        targ = tscale*(time-t0)*(time-t0);
        theta = atan2(xin,yin);
        H_Field[0] = -sin(theta)*iscale*exp(targ);
        H_Field[1] = cos(theta)*iscale*exp(targ);
        H_Field[2] = 0.;
    ...
```

8.1.7. TEM Wave Launch

This boundary condition sets a TEM mode on a sideset given voltages on two nodesets. That is, given a $d-1$ dimensional sideset Γ and two $d-2$ dimensional nodesets N_1 and N_2 with corresponding voltages V_1 and V_2, we solve the surface Laplace equation

$$-\Delta_\Gamma \phi = 0, \qquad \phi|_{N_1} = V_1, \quad \phi|_{N_2} = V_2 \tag{8.11}$$

for ϕ. V_1 and V_2 are assumed to be constant on the entire nodeset, but may be functions of time. Then we set the Dirichlet condition

$$E|_\Gamma = -\nabla_\Gamma \phi. \tag{8.12}$$

This boundary condition has two forms using either a constant voltage or an RTC:

```
Boundary Conditions:
  Fields:
    ...
    Wave Launch Constant Unique_WaveLaunch_Constant_Name:
      Sideset:              [sideset ID string]      (required)
      Nodeset 1:            [nodeset ID string]      (required)
      Nodeset 2:            [nodeset ID string]      (required)
      Voltage 1:            [double]                 (required)
      Voltage 2:            [double]                 (required)
    Wave Launch RTC Unique_WaveLaunch_RTC_Name:
      Sideset:              [sideset ID string]      (required)
      Nodeset 1:            [nodeset ID string]      (required)
      Nodeset 2:            [nodeset ID string]      (required)
      Voltage 1:            [RTC function]           (required)
      Voltage 2:            [RTC function]           (required)
    ...
```

`Unique_WaveLaunch_Constant_Name_1` The boundary condition requires a unique identifier

`Sideset` Sideset to which the condition will be applied (corresponding to Γ above)

`Nodeset 1` and `Nodeset 2` are strings specifying nodesets on the sideset where Dirichlet conditions are applied to the surface Laplace equation (corresponding to N_1 and N_2)

`Voltage 1` and `Voltage 2` define the voltages on these nodesets (V_1 and V_2 above)

Each voltage may be set as either a double to set it to a fixed value for all time or as a string defining an RTC function. The RTC function is a C function that defines the variable `voltage` and can be a function of `time`, for example

```
      Voltage 1: |
        voltage = sin(time);
```

8.1.8. Transmission Line Coupling

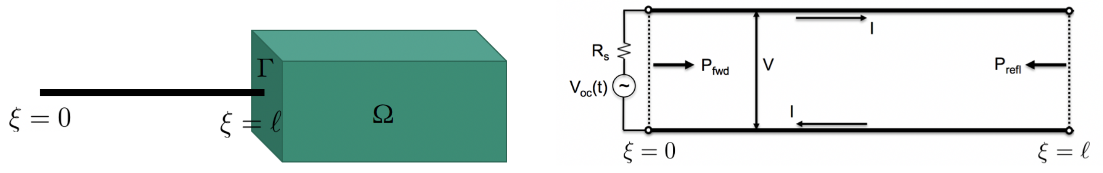

Figure 8-1. Schematics of transmission line coupling. Left: A 1D transmission line of length ℓ coupled to the electromagnetic domain Ω through sideset Γ. Right: Detail of the transmission line model showing the voltage source and resistor and $\xi = 0$.

EMPIRE can couple a 1D transmission line model to a full 2D or 3D electromagnetics simulation through a sideset. This feature is only available in electromagnetics simulations, not electrostatics. There is no limit on the number of transmission lines EMPIRE can couple to, but each transmission line must be coupled to a unique sideset. The transmission line requires a discretization of the 1D domain as well, and is assumed to be driven by a voltage source through a resistor (see Figure 10-1).

The details of the transmission line model are described in the Circuit Network block, see Chapter 10 for details. In order to enable this coupling boundary condition an `EM Coupling` node must be added to the circuit network. If you declare one of these nodes then no other boundary conditions can be enforced on the coupling sideset.

8.1.9. Periodic Boundary Conditions

Periodic boundary conditions are supported for rectangular and wedge meshes. The input file requires two sideset names and the direction for coordinate matching. Periodic boundary conditions will match and merge the electromagnetic degrees of freedom for the geometric entities (nodes, edges and faces) of the two sidesets using a coordinate search. The user can specify an optional relative tolerance for the coordinate matching algorithm (defaults to 1.0e-8). If the coordinates fail to match, adjusting the tolerance in the matching is a good place to start debugging. For a rectangular mesh, the periodic faces must be axis aligned. The face orientation is given by an axis direction. An example input file is:

```
Boundary Conditions:
  Fields:
    ...
    Periodic Unique_Periodic_Name_1:
        Sidesets: [surface_1,surface_2]
        Direction: x
        Coordinate Matching Tolerance: 1.0e-12
    Periodic Unique_Periodic_Name_2:
        Sidesets: [surface_3,surface_4]
        Direction: y
        Coordinate Matching Tolerance: 1.0e-10
    Periodic Unique_Periodic_Name_3:
        Sidesets: [surface_5,surface_6]
        Direction: z
    ...
```

The direction `x` specifies that the planes with normals in the x direction (y-z planes) on the rectangle will be matched. Valid directions for a rectangular mesh are `x`, `y` and `z`. For a wedge mesh, the axis of rotation is the z-axis and the symmetry plane splitting the wedge in half must be the x-z plane. To enable a wedge periodic condition, use `wx` for the direction. **IMPORTANT**: The wedge must subtend an arc which is centered about the **POSITIVE** x-axis otherwise particles will get stuck when they hit the periodic boundary.

```
Boundary Conditions:
  Fields:
    ...
    Periodic Unique_Periodic_Name_1:
        Sidesets: [surface_1,surface_2]
        Direction: wx
    ...
```

The topological entities (nodes, edges and faces) associated with the sidesets must match exactly, however the interior entities of the cells attached to the periodic surfaces do not have to match. Mesh construction must be very careful in matching the side entities. The Panzer inline mesh generators are guaranteed to match. Cubit and Pamgen meshing tools require careful construction. For example, using a paving algorithm on the surfaces of a square can look to match by visual inspection but in reality may have small differences that cause the matcher to fail. The periodic wedge tests in EMPIRE have examples for Exodus and Pamgen that create perfectly aligned surfaces and are the recommended procedure for mesh construction.

Since the periodic field merges mesh entities in the global numbering, ALL degrees of freedom (DOF) in the field solve will be periodic. You cannot specify that only certain DOFs are periodic.

Note that visualizing periodic meshes can sometimes look confusing on coarse meshes since the output for edge and face bases are stored as cell centered quantities. The interior entities of cells on the periodic boundary are not required to be symmetric leading to an unaligned cell centers. An example is shown in Figure 8-2 for a 2D tri mesh. While the side entities (nodes and edge) for

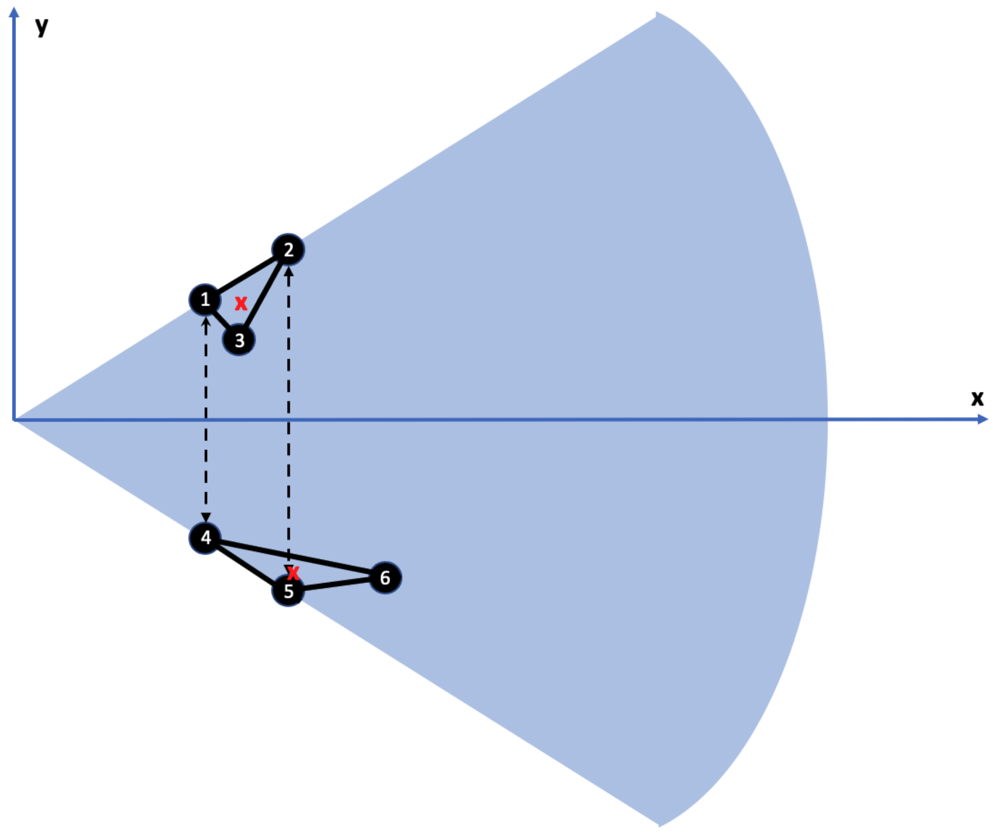

Figure 8-2. Periodic sideset with non-matching interior cells. The center of mass denoted by red x are not aligned so output of cell centered values is not symmetric.

the two elements match (e.g. nodes 1 and 2 match 4 and 5 respectively) the center of mass for the elements are offset in the x direction. Viewing a periodic solution in Paraview will show different solution values for the matching cells. As a mesh is refined, this discrepancy diminishes.

8.2. Electrostatic Boundary Conditions

8.2.1. No Surface Charge

No surface charge conditions natural condition for the electrostatic (Poisson) equations. Thus if nothing else is specified on a sideset,

$$\varepsilon \mathbf{E} \cdot \mathbf{n} = 0 \tag{8.13}$$

will be enforced on the boundary. If only natural conditions are imposed in the problem then the `Pinning` option in the `Physics:  Fields:` should be set.

```
Boundary Conditions:
  Fields:
    ...
    Ground Unique_Ground_Name_1:
      Sidesets: [sideset ID string<, ...>] (required)
      Regions:  [region name<, ...>]       (optional)
    ...
```

`Unique_Ground_Name_1` The boundary condition requires a unique identifier
`Sidesets` List of sidesets to which the condition will be applied (brackets required)
`Regions` Masks the boundary condition to a set of regions if a sideset spans more than one boundary (brackets required)

8.2.2. Fixed Voltage

A simple fixed voltage can be imposed as follows:

```
Boundary Conditions:
  Fields:
    ...
    Potential Constant Unique_Voltage_Name_1:
      Sidesets: [sideset ID string<, ...>] (required)
      Regions:  [region name<, ...>]       (optional)
      Value:    [double]                   (required)
    ...
```

`Unique_Voltage_Name_1` The boundary condition requires a unique identifier
`Sidesets` List of sidesets to which the condition will be applied (brackets required)
`Regions` Masks the boundary condition to a set of regions if a sideset spans more than one boundary (brackets required)
`Value` Determines the voltage imposed on the boundary

8.2.3. Arbitrary Voltage

```
Boundary Conditions:
  Fields:
    ...
    Potential RTC Unique_Voltage_Name_1:
      Sidesets: [sideset ID string<, ...>] (required)
      Regions:  [region name<, ...>]       (optional)
      Function: [RTC Function string]      (required)
    ...
```

`Unique_Voltage_Name_1` The boundary condition requires a unique identifier

`Sidesets` List of sidesets to which the condition will be applied (brackets required)

`Regions` Masks the boundary condition to a set of regions if a sideset spans more than one boundary (brackets required)

`Function` Determines the voltage imposed on the boundary. This standard RTC provides `xin`,`yin`, and, if applicable, `zin` as well as `time`.

9. CURRENT SOURCE

9.1. Current Sources

The Current Source block is used to specify an external source.

```
Current Source:
  Regions: [element block array] (default empty)
  ...
```

This top level `Regions` is used to populate default Zero current source blocks if there are element blocks unspecified by the user. If you over specify element blocks the parser will error out. If you are using a mesh with multiple blocks conditions need to be specified on every block. As such it is recommended that this top level `Regions` be populated with all of the element blocks in the mesh.

As a warning the Current source specified in this block should be tangentially continuous to avoid serial-parallel inconsistencies and reduction in convergence order.

9.2. Zero Conditions

```
Current Source
  Zero:
    Regions: [element block array] (required)
```

This denotes no source in a given region. To specify multiple zero source the list name `Zero` must be appended with a unique suffix.

9.3. RTC Conditions

This specifies arbitrary current source with an RTC.

```
Current Source:
  RTC:
    Regions: [element block array]
    Function:  [RTC Function]
```

To specify multiple RTC sources in a given problem each the sublist name `RTC` must be given a unique suffix.

`Regions`: A valid array of element block strings is required.

RTC: This is a run time compiler function. The user has access to spatial variables (xin,yin,zin) and a time time. The user must specify an array named CURRENT which has dimension appropriate to the prblem. E.g.

```
 double alpha = 7.;
CURRENT[0] = 0.;
CURRENT[1] = sin(alpha*xin-time);
CURRENT[2] = 0.;
```

10. CIRCUIT NETWORK

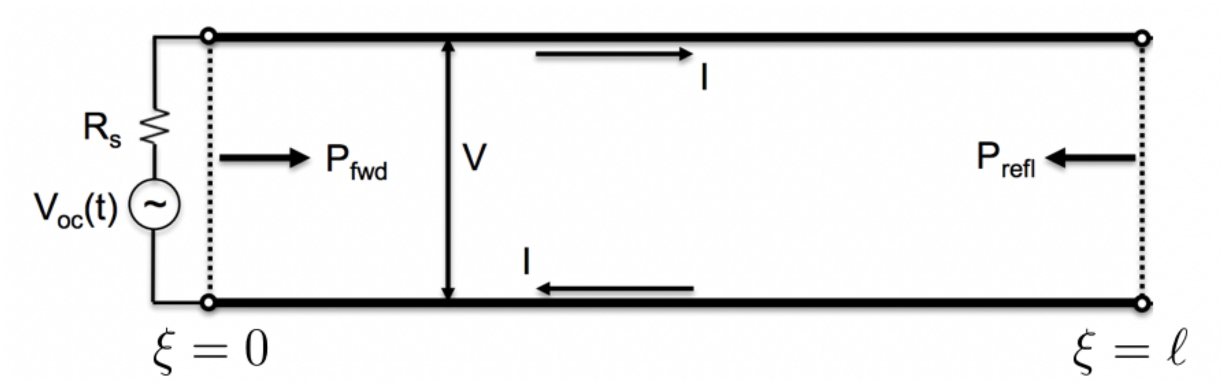

Figure 10-1. A basic schematic of a single conductor transmission line. We call this a single conductor as we assume the bottom conductor is grounded. Our convention for labelling the ens of the cable is that $\xi = 0$ is referred to as the left and and $\xi = \ell$ is the right end.

EMPIRE provides the ability to model a circuit network whose edges are modeled as transmissions lines and whose nodes are various circuit elements. For EM, PIC, and HF problems this requires EMTL coupling on through a sideset. Presently this mode requires a "single conductor" transmission line similar to 10-1.

```
Circuit Network:
  Transmission Lines:
    Unique TL Group Name 1:
      Mode:                        [TEM or TM]              (default=TEM)
      Sideset:                     [sideset ID string]      (see below)
      Surface Mesh:                [surface file name]      (see below)
      Conductors:                  [array of nodeset IDs]   (see below)
      Ground:                      [nodeset ID string]      (see below)
      Parameters File:             [filename string]        (see below)
      Matrix Market:               [filename string]        (see below)
      Capacitance Per Meter:       [double]                 (see below)
      Inductance Per Meter:        [double]                 (see below)
      Wave Number:                 [double]                 (default=0.0)
      Length:                      [double]                 (see below)
      Number of Cells:             [int]                    (see below)
      Names:                       [array of TL names]      (required)
      Relative Permittivity:       [double]                 (default = 1.0)
      Relative Permeability:       [double]                 (default = 1.0)
```

```
        Conductivity:              [double]                 (default = 0.0)
        Voltage Initial Condition: [RTC function string]  (defaults to 0)
        Current Initial Condition: [RTC function string]  (defaults to 0)
        Normal Voltage Initial Condition: [RTC function string ] (defaults to 0)
      ...
    Nodes:
      EM Coupling 1:
        Type:                EM Coupling
        Transmission Lines: [array of TL names]    (required)
        Sideset:            [sideset ID string]    (reguired)
        Conductors:         [array of nodeset IDs] (see below)
        Ground:             [nodeset ID string]    (see below)
      Open Circuit BC 1:
        Type:                      Open Circuit Source
        Transmission Line:    [TL name string]   (required)
        Resistance:                [double]              (defaults to matched)
        Voltage Source Function: [RTC function string] (see below)
        Voltage Source File:     [dat file string]     (see below)
      Kirchhoff Junction 1:
        Type:          Kirchhoff Junction
        Input Lines: [array of TL names] (required)
        Output Line: [array of TL names] (required)
    Solver Parameters:
      Voltage Threshold: [double] (optional)
      Linear Tolerance:  [double] (default = 1.0e-16)
```

The `Circuit Network` block is broken up into three sections: `Transmission Lines` which defines the transmission line edges of the network, `Nodes` which defines various types of nodes on the network, and `Solver Parameters` which determines how the circuit network system is solved algorithmically.

10.1. Transmission Lines

Each transmission lines group defines the parameters of a set of transmission lines. Transmission lines have several possible models

1. **TEM:** The transverse electromagnetic model is the default in EMPIRE. TEM modes have a no normal voltage or current components. TEM modes are only supported on geometries with separated conductors – e.g. parallel plates or coaxial geometries. TEM wave support arbitrary frequencies and have an impedance independent of frequency.

2. **TM:** The transverse magnetic model are generalization of the TEM mode. TM modes have a normal voltage component. TM modes can be supported on more general geometries, e.g. a waveguide or coaxial geometry. TM modes are dispersive with a high-pass dispersion relationship $CL\omega^2 = k^2 + k_n^2$ and a frequency dependent impedance $Z(\omega) = Z_\infty \frac{\sqrt{\omega^2-(ck)^2}}{\omega}$

where $Z_\infty = (\frac{L}{C})^{1/2}$ is the infinite frequency impedance. Here k is a modal wave number with the mode and k_n is the a wave number associated with data along the transmission line.

Otherwise one transmission line is generated. The `Names` parameter requires a list of names for the resulting transmission lines of the correct size. There are several ways to specify parameters on a transmission line:

1. **Sideset Parameterization:** This option takes a sideset on the main EM mesh, and sets a number of nodesets on that sideset as conductors. One of these conductors must be grounded for well-posedness. The C and L parameters on the transmission line are obtained by solving Poisson systems on the sideset. `Relative Permittivity` and `Relative Permeability` will scale the resulting values of C and L. A `Conductivity` along the line may be optionally added. The transmission line `Length` and `Number of Cells` in its discretization are required. Take for example:

```
Circuit Network:
  Transmission Lines:
    Sideset TL Group:
      Sideset:                    sideset_1
      Conductors:                 [nodeset_1, nodest_2, nodeset_3]
      Ground:                     nodeset_1
      Length:                     1.0
      Number of Cells:            100
      Names:                      [TL_1, TL_2]
```

   This example will generate two transmission lines from sideset_1 associated with conductors nodeset_2 and nodeset_3, each of length 1.0 with 100 cells.

2. **Parameters from a Table:** A spatially varying transmission line can be defined by is tabular file specified by `Parameters File`. The required format of this model is specific to the Mode specified The rows of this file correspond to different sections of the transmission line.

   For the TEM mode the user must specify a length, number of cells, capacitance, inductance, and conductivity. The conductivity is optional. For example,

```
# length  num_cells  C_l   L_l  G_l
  0.5     50         0.25  1.0  0.0
  0.5     40         1.0   1.0  0.0
```

   would define a transmission line of length 1.0 with two sections, each of length 0.5. The capacitance per unit length in the first section is 0.25, and in the second section it is 1.0. The discretization edge length dx in the first section is 0.01 in the first section and 0.0125 in the second section. An example input block would be:

```
Circuit Network:
  Transmission Lines:
    Parameters File Group:
      Parameters File:            tl_params.dat
      Names:                      [TL_1]
```

For a TM mode the user must specify length, number of cells, capacitance, inductance, wave number, and conductivity. Conductivity is the only optional parameter

```
# length  num_cells  C_l   L_l  k_l  G_l
  0.5     50         0.25  1.0  1.0  0.0
  0.5     40         1.0   1.0  2.0  0.0
```

3. **Hard-coded Parameters:** The user can alternatively specify uniform values of C, L, and if necessary k over the line with `Capacitance Per Meter`, `Inductance Per Meter`, and `Wave Number`. For TM Mode a non-zero k is required. For TEM Mode a non-zero k will result in an error. The parameters `Length` and `Number of Cells` are also required in this case. Take for example:

```
Circuit Network:
  Transmission Lines:
    Sideset TL Group:
      Capacitance Per Meter:      1.0e-8
      Inductance Per Meter:       1.0e-10
      Wave Number:                1.0e2
      Length:                     1.0
      Number of Cells:            100
      Names:                      [TL_1]
```

4. **From a Matrix Market File:** You can characterize a transmission line using matrix market which output from a Cross Section Mesh option. This can be done with the following inputs

```
Circuit Network:
  Transmission Lines:
    Cross-Section_TL_Group:
      Matrix Market: Cross-Section_TL_Group.matrix.market
      Conductors: [ground, conductor0, conductor1]
      Ground: ground
      Length: 0.3
      Number of Cells: 100
```

The name of the group do not have to match the name of the matrix market file.

Additional optional parameters are available to set initial conditions on the transmission lines. `Voltage Initial Condition` and `Current Initial Condition` are optional RTC functions that can define the initial voltage and current in the transmission line. These should be C functions that define the variables `V` and `I` respectively and can be functions of `xin`, which corresponds to ξ in Figure 10-1. If not supplied, the initial condition will be set to zero.

10.2. Nodes

EMPIRE currently supports the following node types:

1. **EM Coupling:** This node type is used to connect a set of transmission lines to a sideset of the main EM mesh. Each transmission line is coupled through a conductor on the sideset, and a ground must also be supplied.

2. **Open Circuit Source:** This option applies a boundary condition at the end of a transmission line of the form

$$V_{OC}(t) - V(0,t) = R_s I(0,t) \quad \text{or} \quad V(\ell,t) - V_{OC}(t) = R_s I(\ell,t). \tag{10.1}$$

The difference in sign between right and left endpoints guarentees a sensible energy estimate for the system. Here R_s is the resistance at the transmission line source (R_s in Figure 10-1) and V_{OC} is the voltage at the source. The parameter `Resistance` is used to define R_s. If this parameter is not specified, the source resistance will be set to be impedance matched for the transmission line, i.e. $R_s = \sqrt{L/C}$. When an analytic solution is know for both V and I, the boundary condition relation can be used to define the open circuit voltage V_{OC}. When driving with a know voltage, if the source is impedance matched to the transmission line, the driving voltage at the source needs to be doubled to deliver the expected voltage to the EM domain. `Voltage Source Function` is a string defining an RTC function for the source voltage, V_{OC}. This should be a C function that defines the variable `Voc` and can be a function of `time`, for example

```
      Voltage Source Function: |
        Voc = sin(time);
```

Alternatively, the voltage can be defined via a tabular data file with the parameter `Voltage Source File`. This parameter is a string supplying the name of the data file.

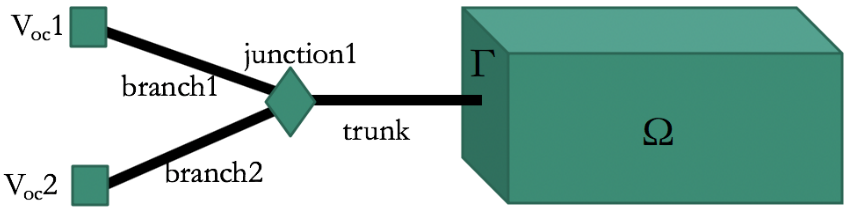

Figure 10-2. Schematic of transmission line coupling including a junction.

3. **Kirchhoff Junction:** This option supports simple junctions of transmission lines that are governed by Kirchhoff's laws. It takes a set of `Input Lines` and `Output Lines` and enforces that the sum of currents entering equals the sum of currents leaving and that the voltages at the junction are all equal. For example, the junction in Figure 10-2 would be defined by

```
  Circuit Network:
    Transmission Lines:
      Trunk Group:
        Names:         [trunk]
        Sideset:       [Gamma]
        Conductors:    [cathode_nodeset, anode_nodeset]
        Ground:        cathode_nodeset
        ...
      Branch1 Group:
        Names:            [branch1]
        Parameters File: branch1_params.dat
        ...
      Branch2 Group:
        Names:            [branch2]
        Parameters File: branch2_params.dat
        ...
    Nodes:
      EM Coupling:
        Type:                 EM Coupling
        Transmission Lines:  [trunk]
        Sideset:             [Gamma]
        Conductors:          [cathode_nodeset, anode_nodeset]
        Ground:              cathode_nodeset
      Open Circuit BC 1:
        Type:                  Open Circuit Source
        Transmission Line:    branch1
        Voltage Source File: VOC1.dat
      Open Circuit BC 2:
        Type:                  Open Circuit Source
        Transmission Line:    branch2
        Voltage Source File: VOC2.dat
      Kirchoff junction1:
        Type:          Kirchhoff Junction
        Input Lines: [branch1, branch2]
        Output Line: [trunk]
```

10.3. Solver Parameters

Within `Solver Parameters`, `Voltage Threshold` sets a minimum voltage at the TL interface before which the EM solve will not be performed. This option is useful when a simulation has zero initial conditions in the EM domain and it takes a large number of timesteps for the TL dynamics to influence the electromagnetics. Then we save computation time by skip unnecessary EM solves against essentially zero righthand sides while the early TL dynamics evolve. The EM solves will not start until the shared voltage between the EM and TL domains reaches a minimum magnitude of `Voltage Threshold`.

`Linear Tolerance` specifies the convergence criteria of the transmission line linear solver.

11. SOLVER PARAMETERS

```
Solver Parameters:
  Maximum Iterations:           [int]      (default see description)
  Tolerance:                    [double]   (default 1e-8)
  Preconditioner Type:          [string]   (default Testing Multigrid)
  Num Blocks:                   [int]      (default 200)
  Linear Solver Output:         [bool]     (default true)
  Linear Solver Output File:    [string]   (default solver.log)
  Multigrid Output:             [bool]     (default true)
  Multigrid Output File:        [string]   (default multigrid.log)
  Linear Sovle xml:             [filename] (optional)
  Nonlinear Maximum Iterations: [int] (default 2)
  Nonlinear Relative Tolerance: [double] (1e-8)
  Nonlinear Absolute Tolerance: [double] (1e-12)
  Nonlinear Solver Output File: [str] (default EM\_newton.log)
  Nonlinear Solver Failure:     [str] (default Warn)
```

`Maximum Iterations` Maximum iterations is determined by `Preconditioner Type`. If `Multigrid` is used the default is 200, if `Jacobi` is selected then 1000.

`Tolerance` The minimum accepted relative residual for the iterative solver.

`Preconditioner Type` EMPIRE supports the following preconditioner paths:

`Testing Multigrid` An Algebraic Multigrid approach that is best suited for small (fewer than 1 million element) problems, such as those in EMPIRE's ctest suite. The refMaxwell multigrid method is employed which reformulated Maxwell's equations to solve two auxiliary sub-problems. The particular parameters for this option can be found in the file `EMPIRE/xml_files/em_multigrid_classic.xml`. These parameters use multi-threaded Gauss-Seidal smoothing and coarsen to a 2500 DOF coarse mesh on which a direct solve is applied.

`At Scale Multigrid` Algebraic Multigrid with parameters tuned for performance at large scale. When compiled with CUDA, the parameters in the file `EMPIRE/xml_files/em_multigrid_cuda.xml` are appended to the `Testing Multigrid` parameters to obtain the `At Scale Multigrid` parameters. Otherwise, the parameters in `EMPIRE/xml_files/em_multigrid.xml` are appended. In both cases, the mesh is coarsened only once and the coarse problem is approximately solved with a Chebyshev iteration. The CUDA version is tuned for use on GPUs.

`Jacobi` A simple diagonal scaling.

`Default` This will swap between `Testing Multigrid` for very small problems, `Jacobi` for medium sized problems and `At Scale Multigrid` for large problems. These definitions of problem size are in a state of flux and will be documented later.

`Num Blocks` Maximum number of solver blocks.

`Linear Solver Output` This option controls whether linear solve data is written to a log file. As it defaults to true, this need only be specified if no output is desired. Note that linear solver output can be helpful in debugging failed jobs and is thus always useful to produce.

`Linear Solver Output File` The name of the file to write linear solver output to.

`Multigrid Output` This option controls whether multigrid setup data is written to a log file when a multigrid preconditioner is used. As it defaults to true, this need only be specified if no output is desired. Note that multigrid output can be helpful in debugging failed jobs and is thus always useful to produce.

`Multigrid Output File` The name of the file to write multigrid output to.

`Linear Solver xml` More complex custom settings can be set by providing a MuLue xml file. The formatting of this file is documented in the MueLu documentation. It is the design principal of EMPIRE that fine tuning the linear solve is uncessary. This option is a back door and is intentionally obtuse. This option should only be used to fix corner cases or as a work around; thus the development team will likely provide the user if necessary.

`Nonlinear Maximum Iterations` Maximum number of nonlinear iterations allowed at each time step.

`Nonlinear Relative Tolerance` The minimum relative residual accepted by the nonlinear solver.

`Nonlinear Absolute Tolerance` The minimum absolute residual accepted by the nonlinear solver. This may need to be tuned for problems achieving steady state.

`Nonlinear Solver Output File` File name for an output file describing the performance of the nonlinear solver.

`Nonlinear Solver Failure` Selects behavior of the solver when residual tolerance is not reached in the maximum number of iterations. Options are `Error` which halts the code and `Warn` which prints warning to the screen and output file.

If the requested residual is not achieved within the maximum iterations then EMPIRE will exit and report an error.

The nonlinear solver toleraence is given as the sum of the noliinear relative and absolute tolerance.

12. DIAGNOSTICS AND OUTPUT

In order to generate diagnostic output to the screen or to files, the user must specify both a *History Diagnostics* block and a *History Outputs* block. A *History Diagnostics* block has the same structure for time and mesh histories, differing only by top level name (i.e. **Time History Diagnostics** vs. **Mesh History Diagnostics**). In every diagnostic block, we have the option to include a scalar multiplier for the diagnostic. This can be used to change the units of the diagnostic, or to scale the value by a wedge factor.

```
Time History Diagnostics:
  [Name for diagnostic 1]:
    [Diagnostic type for diagnostic 1]:
      Multiplier: [float, default 1.0]
      [Block specific to the diagnostic type]
  [Name for group A]:
    [Name for diagnostic 2]:
      [Diagnostic type for diagnostic 2]:
        Multiplier: [float, default 1.0]
        [Block specific to the diagnostic type]
    [Name for diagnostic 3]:
      [Diagnostic type for diagnostic 3]:
        Multiplier: [float, default 1.0]
        [Block specific to the diagnostic type]
```

Now we specify an output block which will output diagnostics 1 and 2 to the screen, output diagnostics 1 and 3 to the file output.txt and all diagnostics under "group A" to file group-A.txt. Note that two different syntaxes are accepted for the field **Diagnostics** (this is a YAML list). Users may group diagnostics together under an arbitrary group name in the **Time History Diagnostics** and **Mesh History Diagnostics** blocks. Groups of diagnostics may be output using only the group name. Groups of groups up to an arbitrary recursion depth are possible, so that a group of groups may be output to file by specifying only the top level group name to the **Diagnostics** entry in the history output block.

```
Time History Outputs:
  Screen:
    Diagnostics: [diagnostic 1, diagnostic 2]
  output.txt:
    Diagnostics:
      - diagnostic 1
      - diagnostic 3
    Field Width: [int, default is an automatically computed width]
    Field Precision: [int, default 6]
```

```
      Separator: [string, default (space)]
      Enable Banner: [bool, default true]
      Print Version: [bool, default true]
      Echo Input: [bool, default true]
      Stride:
        ...
      Averaging:
        Window: [string, default Current]
        Size: [int, default 1]
        Factor: [float]
        Stride:
          ...
    group-A.txt:
      Diagnostics: [group A]
```

The entries parsed from each output file in the **Time History Outputs** block are as follows:

`Field Width`: Width, in number of characters, of the printed fields.

`Field Precision`: Number of significant digits to print for each field.

`Separator`: Separator used between fields.

`Enable Banner`: Specifies whether a descriptive banner having the diagnostic names is printed at the top of the output file.

`Print Version`: Specifies whether the code version is printed in the header.

`Echo Input`: Specifies whether to echo the entire input deck into the header.

`Stride`: A stride block which describes the intervals at which output is printed.

`Averaging`: An averaging block which describes the averaging to be applied to the diagnostic output. Note that averaging is not applied to certain diagnostics (e.g. Timestep).

`Window`: The averaging algorithm to be used. Accepts "Sliding", "Continuous", "Single" and "Current".

`Size`: Only used for "Sliding" averaging.

`Factor`: Only used for "Continuous" averaging. Must fall in the range (0,1].

The structure of a **Mesh History Outputs** block is slightly different:

```
  Mesh History Outputs:
    file1.exo:
      Diagnostics: [mesh diagnostic 1, mesh diagnostic 2]
    file2.exo:
      Diagnostics:
        - mesh diagnostic 1
        - mesh diagnostic 3
      Blocks: [string, default All]
      Stride:
        ...
      Averaging:
        Window: [string, default Current]
        Size: [int, default 1]
        Factor: [float]
        Stride:
          ...
```

Other than **Blocks**, The meaning of the entries under each output file are the same as for those in a **Time History Outputs**. For mesh histories, "Screen" does not have a special meaning as a file.

`Blocks`: Blocks over which to output diagnostics.

12.1. Electromagnetic Diagnostics

12.1.1. Time History Diagnostics

12.1.1.1. Field At Point

This diagnostic allows the user to output the time history of the value of a field at a point (x,y,z) in the mesh. For a 2D mesh, the z-coordinate has no influence.

```
  Field At Point:
    Field: [string, default E]
    Point: [string, default "0.0, 0.0, 0.0"]
    Projection: [string, default "0.0, 0.0, 0.0"]
    Use Nodal Fields: [bool, default true]
```

`Field`: Field to sample. May be either a solver field or a mesh history diagnostic.

`Use Nodal Fields`: If set to true will use projected electric and magnetic fields, otherwise samples shape functions directly.

`Point`: Point x,y,z at which to sample field

`Projection`: A vector parallel to the normal along which to project the field for output. If (0,0,0) is specified and `Field` is a solver field, then the x, y and z components will be output separately. If (0,0,0) is specified and `Field` is the name of a mesh diagnostic (representing either a scalar or vector field), then the magnitude of that field is output.

12.1.1.2. Line Integral

This diagnostic allows the user to output the time history of the line integral of a field through a set of points in the mesh or around a circular loop in the mesh. Note that a mesh history field can be specified so that one could, for example, compute the line integral over particle density. Note that this diagnostic will output a file *diagnosticName_points.h5part* which holds the points at which the integrand is evaluated.

To compute a line integral along line segments connecting a set of points:

```
Line Integral:
  Field: [string, default E]
  Points: [list of strings]
  Num Points: [int, default 2]
  Use Nodal Fields: [bool, default false]
```

`Field`: Field to integrate. May be either a solver field or a mesh history diagnostic.

`Points`: List of points that define the path.

`Num Points`: Number of integration points for each line segment.

`Use Nodal Fields`: If set to true will use projected electric and magnetic fields, otherwise samples shape functions directly.

To compute a line integral along a circular loop:

```
Line Integral:
  Field: [string, default E]
  Circle Center: [string]
  Circle Radius: [float]
  Circle Normal: [string]
  Num Points: [int]
  Use Nodal Fields: [bool, default false]
```

`Circle Center`: Center of a circle to be used for the line integral.

`Circle Radius`: Radius of the circle to be used for the line integral.

`Circle Normal`: Normal to the plane of a circle to be used for the line integral.

`Num Points`: Number of integration points for the circle.

12.1.1.3. Surface Temperature At Point

Outputs the value of the surface temperature, as computed by the specified **Surface Temperature** mesh history diagnostic, near a point (x,y,z) in the mesh. The point must be specified in an element which owns a face belonging to a sideset that is being processed by the **Surface Temperature**. If more than one face of the element is associated with a temperature then, in the current implementation, the face for output will be randomly selected.

```
Surface Temperature At Point:
  Point: [string, default "0.0, 0.0, 0.0"]
  Surface Temperature Diagnostic Name: [string]
```

`Point`: Point x,y,z near which to sample temperature

`Surface Temperature Diagnostic Name`: Name of the "Surface Temperature" mesh diagnostic

12.1.1.4. Field Energy

Outputs the specified, integrated energy quantity.

```
Field Energy:
  Quantity: [string]
```

`Quantity`: Quantity to output.

The following quantities are currently available,

- Electric Energy: $\frac{\varepsilon}{2}\int \vec{E}\cdot\vec{E}\;dV$
- Magnetic Energy: $\frac{1}{2\mu}\int \vec{B}\cdot\vec{B}\;dV$
- Electromagnetic Energy: $\frac{\varepsilon}{2}\int \vec{E}\cdot\vec{E}\;dV + \frac{1}{2\mu}\int \vec{B}\cdot\vec{B}\;dV$
- Joule Heating: $\int \vec{J}\cdot\vec{E}\;dV$
- Electrostatic Potential Energy: $\frac{1}{2}\int \rho\phi\;dV$

12.1.1.5. Poynting Flux

Computes the Poynting flux, $\int_S \frac{\vec{E}\times\vec{B}}{\mu_0}\cdot\hat{n}\;dS$, over the specified surface S, where mu_0 is the vacuum permeability in SI units. Currently Monte-Carlo integration is used. Note that this diagnostic will output a file *diagnosticName_points.h5part* which holds the points at which the integrand is evaluated. For 3D simulations use the following,

```
  Poynting Flux:
    Resolution: [int, default 2000]
    Geometry: Rectangle
    Normal Vector: [string, default "0.0, 0.0, 0.0"]
    Tangent Vector 1: [string, default "0.0, 0.0, 0.0"]
    Center: [string, default "0.0, 0.0, 0.0"]
    Width (Direction 1): [double, default 0.0]
    Height (Direction 2): [double, default 0.0]
    Use Nodal Fields: [bool, default false]
```

For 2D simulations, "Line" Geometry must be used,

```
  Poynting Flux:
    Resolution: [int, default 2000]
    Geometry: Line
    Normal Vector: [string, default "0.0, 0.0, 0.0"]
    Point 1: [string, default "0.0, 0.0, 0.0"]
    Point 2: [string, default "0.0, 0.0, 0.0"]
    Use Nodal Fields: [bool, default false]
```

A brief description of all of the **Poynting Flux** entries follows,

`Resolution`: Number of sampling points

`Geometry`: Geometry type for flux surface (Rectangle or Line)

`Normal Vector`: Vector normal to the rectangle

`Tangent Vector 1`: Vector tangent to the rectangle (Rectangle geometry only)

`Center`: Point at center of rectangle (Rectangle geometry only)

`Width (Direction 1)`: Width of rectangle in direction 1 (Rectangle geometry only)

`Height (Direction 2)`: Width of rectangle in direction 2 (Rectangle geometry only)

`Point 1`: Point at beginning of line (Line geometry only)

`Point 2`: Point at end of line (Line geometry only)

`Use Nodal Fields`: If set to true will use projected electric and magnetic fields, otherwise samples shape functions directly.

12.1.1.6. Transmission Line

This diagnostic allows the user to print time histories of voltages and currents within a given transmission line coupled to the electromagnetic domain. Currently, one can choose to sample the voltage or current at either the source of the transmission line or at the boundary where the transmission line couples to the EM domain. The syntax for this diagnostic is as follows:

```
Transmission Line:
  Sideset:  [sideset ID string]  (required)
  Field:    [Current or Voltage] (defaults to Voltage)
  Location: [Source or Boundary] (defaults to Boundary)
```

`Sideset` here is a string defining the sideset to which the transmission line of interest is coupled. This is a unique identifier for the transmission line. `Field` is a string defining what field to sample and can be only `Current` or `Voltage`, the degrees of freedom in the transmission line model. `Location` is a string defining where to sample the field. Setting `Location` to `Source` gives the value of the field at the transmission line source, the end of the transmission line farthest from the EM domain. Setting `Location` to `Boundary` gives the value of the field at the end of the transmission line that couples to the EM domain.

12.1.1.7. Memory Highwater

Outputs the minimum, average and maximum amount of memory (in MB) used by EMPIRE on the compute nodes.

```
Memory Highwater:
```

12.1.1.8. Norm Diagnostic

This diagnostic allows the user to evaluate a norm on any Mesh History Diagnostic. Norms include L_2/root-mean-square (RMS), L_1, and L_∞ as defined by

$$|v|_2 = RMS(v) = \sqrt{\sum_i v_i^2} \tag{12.1}$$

$$|v|_1 = \sum_i |v_i| \tag{12.2}$$

$$|v|_\infty = \max(|v|) \tag{12.3}$$

The input deck takes the form

```
Norm Diagnostic:
  Mesh Diagnostic Name: [string, default ""]
  Norm Type: [string, default "L1"]
```

`Mesh Diagnostic Name`: Mesh history diagnostic to read data from.

`Norm Type`: Type of norm. Options are: L1, L2, RMS, Linf

12.1.1.9. Time and Solver Diagnostics

The following diagnostics are computed by default and can be requested directly in the `Time History Outputs` list:

`Timestep`: The size of the timstep Δt

`Simulation_time`: The time t reached in the simulation

`Elapsed_time`: The wall clock time elapsed

`Iteration_count`: The number of iterations required for linear solver convergence

`Achieved_tolerance`: The tolerance reached by the linear solver

As an example,

```
Time History Outputs:
  Screen:
    Diagnostics: [Iteration_count]
```

would print iteration counts to screen without requiring any `Time History Diagnostics` being defined.

12.1.2. Mesh History Diagnostics

12.1.2.1. Solver Field

This diagnostic outputs all CG electromagnetic/electrostatic solver fields.

```
Solver Field:
  Fields: [string, default All]
  Sampling: [string, default Default]
```

`Fields`: Specify which solver fields are to be output; if "All" is present, dump all solver fields.

`Sampling`: Specify what processing to apply to solver field data before outputting: takes Nodal, Centered, Average or Default.

12.1.2.2. DG Field

This diagnostic outputs all DG fields (often associated with fluids). DG fields can be written either as the cell-centered average or as individual Degrees of Freedom. The individual Degrees of Freeedom are written as Exodus element variables with the DoF ordinal as a suffix. The DoF ordering matches that given by the Intrepid2 basis classes.

This diagnostic can also create Exodus Information Records that identify the Intrepid2 basis and the names of the DG fields written.

To visualize the DG fields using the Paraview Finite Element Field Distributor filter (available starting in Paraview 5.11.0), set `Sampling` to `None` or `Default` and set `Create Basis Info Records` to `true`.

```
DG Field:
  Fields: [string, default All]
  Sampling: [string, default Default]
  Create Basis Info Records: [bool, default false]
```

`Fields`: Specify which solver fields are to be output

`Sampling`: Specify what processing to apply to DG field data before outputting: None, Average or Default

`Create Basis Info Records`: If set to true, Exodus Information Records will be created

---

12.1.2.3. Edge Field

This diagnostic outputs edge fields (HCurl coefficients). Edge fields can be written to either an Exodus edge block or to an Exodus element block. When written to an edge block, there is one coefficient for each shared edge in the mesh and the orientation map is not applied to the coefficients. When written to an Exodus element block, an element variable is created for each edge of each element and the orientation map is applied to the coefficients. The element variables are suffixed with the edge number as described by the Intrepid2 basis class.

This diagnostic can also write the orientation map to an Exodus element block using element variables.

This diagnostic can also write Exodus Information Records that identify the Intrepid2 basis and the names of the edge fields written.

To visualize the edge fields using the Paraview Finite Element Field Distributor filter (available starting in Paraview 5.11.0), set `Write Oriented Coefficients` to `true` and set `Create Basis Info Records`to `true`.

```
Edge Field:
  Fields: [string, default E_Field]
  Create Basis Info Records: [bool, default false]
  Write Edge Block: [bool, default false]
  Write Orientation Map: [bool, default false]
  Write Oriented Coefficients: [bool, default false]
```

`Fields`: Specify which solver fields are to be output

`Create Basis Info Records`: If set to true, Exodus Information Records will be created

`Write Edge Block`: If set to true, write edge coefficients to an Exodus edge block

`Write Orientation Map`: If set to true, write the orientation map to an Exodus element block

`Write Oriented Coefficients`: If set to true, write oriented edge coefficients to an Exodus element block

12.1.2.4. Face Field

This diagnostic outputs face fields (HDiv coefficients). Face fields can be written to either an Exodus face block or to an Exodus element block. When written to a face block, there is one coefficient for each shared face in the mesh and the orientation map is not applied to the coefficients. When written to an Exodus element block, an element variable is created for each face of each element and the orientation map is applied to the coefficients. The element variables are suffixed with the face number as described by the Intrepid2 basis class.

This diagnostic can also write the orientation map to an Exodus element block using element variables.

This diagnostic can also write Exodus Information Records that identify the Intrepid2 basis and the names of the face fields written.

To visualize the face fields using the Paraview Finite Element Field Distributor filter (available starting in Paraview 5.11.0), set `Write Oriented Coefficients` to `true` and set `Create Basis Info Records`to `true`.

```
Face Field:
  Fields: [string, default B_Field]
  Create Basis Info Records: [bool, default false]
  Write Face Block: [bool, default false]
  Write Orientation Map: [bool, default false]
  Write Oriented Coefficients: [bool, default false]
```

`Fields`: Specify which solver fields are to be output

`Create Basis Info Records`: If set to true, Exodus Information Records will be created

`Write Face Block`: If set to true, write face coefficients to an Exodus face block

`Write Orientation Map`: If set to true, write the orientation map to an Exodus element block

`Write Oriented Coefficients`: If set to true, write oriented face coefficients to an Exodus element block

---

12.1.2.5. D Dot N

This diagnostic computes $\mathbf{D} \cdot \mathbf{n}$ on the boundary and outputs it on the mesh. The diagnostic is based on the equation

$$\int_\Omega \rho\phi \, dx + \int_{\partial\Omega} (\mathbf{D} \cdot \mathbf{n})\phi \, ds = \int_\Omega \rho_c \phi \, dx + \int_\Omega \mathbf{D} \cdot \nabla\phi \, dx \tag{12.4}$$

for each nodal test function ϕ. The diagnostic first evaluates the volume integrals on the righthand side. It then restricts this to nodes on the boundary and applies a bounary mass matrix solve to obtain $\mathbf{D} \cdot \mathbf{n}$ at nodes on the boundary. This mass matrix can be lumped, making the inverse a diagonal scaling, or consistent, requiring an iterative solve. This diagnostic is evaluated only on nodes, as it is inaccurate anywhere else.

```
D Dot N:
  Lumped: [bool, default true]
```

`Lumped`: Used a lumped mass matrix for the mass matrix solve

12.1.2.6. Div D Minus Rho

This diagnostic computes $\nabla \cdot \mathbf{D} - \rho$ and outputs it on the mesh. When not using "integration by parts" (the default behavior), the quantity is evaluated in a straightforward manner at cell centers. A mass matrix inversion is required to evaluate the charge density at cell centers in this calculation. Solver and preconditioner options for the mass matrix inversion are accessible to the user in the input deck. When using the "integration by parts" path, the diagnostic is computed by first evaluating

$$\int_\Omega (\nabla \cdot \mathbf{D} - \rho)\phi \, dx = -\int_\Omega \mathbf{D} \cdot \nabla\phi \, dx - \int_\Omega \rho\phi \, dx + \int_{\partial\Omega} (\mathbf{D} \cdot \mathbf{n})\phi \, ds \tag{12.5}$$

for each nodal test function ϕ and then applying a mass matrix inverse to remove the integral. This mass matrix can be lumped, making the inverse a diagonal scaling, or consistent, requiring a solve. If a "lumped" mass matrix is used, the solver and preconditioner parameters do not apply.

```
Div D Minus Rho:
  Sampling: [string, default Default]
  Lumped: [bool, default false]
  Use Integration By Parts: [bool, default false]
  Boundary Element Values Are NaN: [bool, default true]
  Preconditioner: [string, default R-ILU(K)]
  Preconditioner XML: [string, default ""]
  Solver Tolerance: [double, default 1.0e-16]
  Maximum Iterations: [int, default 100]
```

`Sampling`: Specify what processing to apply to solver field data before outputting: takes Nodal, Centered, Average or Default.

`Lumped`: Used a lumped mass matrix for the mass matrix solve

`Use Integration By Parts`: Specify the method for computing the quantity

`Boundary Element Values Are NaN`: Set the diagnostic value to NaN at boundaries

`Preconditioner`: Preconditioner for the mass matrix inversion if a lumped mass matrix is not used: takes R-ILU(K), Jacobi or None

`Preconditioner XML`: File specifying the Ifpack2 RILUK or RELAXATION preconditioner parameters

`Solver Tolerance`: Solver tolerance for the mass matrix inversion

`Maximum Iterations`: Solver max iterations for the mass matrix inversion

12.1.2.7. *Collision Rate*

This diagnostic computes quantities which are related to the number of collisions in each mesh cell,

$$r_i = \frac{1}{f g_i} \sum_{j=1}^{N_i} w_{i,j}, \tag{12.6}$$

where N_i is the number of computational collision events in mesh cell i. The quantities $w_{i,j}$, f and g_i are defined according to the user specified options as described below,

```
 Collision Rate:
   [name of collision rate quantity 1]:
       Reaction: [string]
       Scale by Volume: [bool, default true]
       Scale by Time: [bool, default true]
       Scale by Weight: [bool, default true]
   [name of collision rate quantity 2]:
       ...
   [name of collision rate quantity 3]:
       ...
```

`Reaction`: Reaction name.

`Scale by Volume`: If true, g_i is the volume of the i^{th} mesh cell, otherwise it is unity.

`Scale by Time`: If true, f is the timestep Δt, otherwise it is unity.

`Scale by Weight`: If true, sum the real # of collisions ($w_{i,j}$ is the weight associated with the j$^{\text{th}}$ collision in the i$^{\text{th}}$ mesh cell), otherwise sum the computational # of collisions ($w_{i,j} = 1$) for output.

12.1.2.8. Quadrature Data

This diagnostic outputs data stored at quadrature points onto a mesh. The data on quadrature points is averaged over each element and is output to exodus as a cell quantity. Presently only material models store data on quadrature points that is available to this diagnostic. The fields that are available are `current`, `omega_plasma`, and `nu_collision`. If a field is selected that applies to multiple material models, then all will be output to the same field in exodus. Since the material models are on different element blocks, they will not overlap.

```
 Quadrature Data:
   Fields: [list of strings, required]
```

`Fields`: List of fields to output, needs to regex match names.

12.1.2.9. RTC

The runtime-compiler (RTC) diagnostic is used mainly for testing purposes. The goal of the operator is to supply an analytic evaluation of an arbitrary function as a mesh quantity.

This diagnostic outputs all solver fields.

```
 RTC:
   Function: [string, default ""]
   Location: [string, default "Cell Center"]
```

`Function`: Source code (C) for function evaluation. Output field name is “output”. Inputs include time, xin, yin, zin, as well as various constants (e.g. PI).

`Location`: Specify where the function is located on the mesh. Currently only “Cell Center” is supported.

12.1.2.10. Far Field

The Far Field diagnostic computes the near to far field transformation. The user specifies a set of frequencies, **Frequencies**, for which the far fields will be computed. Then, equivalent sources (Love’s Equivalance Principle) for each frequency are determined over the specified sideset, **Integration Sideset Name**, over the course of the entire simulation. Note that a region, **Nearfield Region**, must be specified in order to disambiguate the direction of the surface normal on the specified sideset. If the domain is split into two regions, one containing all of the electric currents which radiate and one which contains only vacuum, then the former region should be specified. At the end of the simulation, the far fields are computed by integrating over the equivalent sources.

sideset, **Integration Sideset Name**,

```
Far Field:
  Frequencies: {farfield_frequencies}
  Integration Sideset Name: [string]
  Nearfield Region: [string]
  Far Field Grid Type: [string, default "Arc"]
  Point 1: [string]
  Point 2: [string]
  X Axis: [string, default "1.0, 0.0, 0.0"]
  Y Axis: [string, default "0.0, 1.0, 0.0"]
  Z Axis: [string, default "0.0, 0.0, 1.0"]
  Num Points: [int, only use with "Arc"]
  Num Points Theta: [int, only use with "Angular Area"]
  Num Points Phi: [int, only use with "Angular Area"]
```

`Integration Sideset Name`: Name of the sideset on which to compute the equivalent sources

`Nearfield Region`: An exodus mesh block which is adjacent to the integration sideset

`Frequencies`: List of frequencies for which to compute the far fields

`Far Field Grid Type`: Accepts "Arc" or "Angular Area"

`Num Points`: Used only when **Far Field Grid Type** is "Arc"

`Num Points Theta`: Used only when **Far Field Grid Type** is "Angular Area"

`Num Points Phi`: Used only when **Far Field Grid Type** is "Angular Area"

`Point 1`: Point (theta1, phi1) for specifying the farfield grid

`Point 2`: Point (theta2, phi2) for specifying the farfield grid

`X Axis`: Specifies the x-axis, which determines the orientation of the spherical coordinate system that the far fields refer to

`Y Axis`: Specifies the y-axis, which determines the orientation of the spherical coordinate system that the far fields refer to

`Z Axis`: Specifies the z-axis, which determines the orientation of the spherical coordinate system that the far fields refer to

12.1.2.11. Far Field Transient

The Far Field Transient diagnostic computes the near to far field transformation in the time domain for a single far field point. Equivalent sources (Love's Equivalance Principle) are determined over the specified sideset, **Integration Sideset Name**. The value of the sources are temporally interpolated, using a polynomial interpolation of order **Interpolation Order**, to the retarded time and the contribution to the far field value at the far field time is added. At the end of the calculation, the far fields are known and the values are output to file named according to the diagnostic name. Note that a region, **Nearfield Region**, must be specified in order to disambiguate the direction of the surface normal on the specified sideset. If the domain is split into two regions, one containing all of the electric currents which radiate and one which contains only vacuum, then the former region should be specified. The output is computed assuming that the fields on the near field integration surface have been zero for all time before the first simulation step. Output is printed only for far fields times which are fully causual within the simulation (given the aforementioned assumption). Note that, in order to get the correct field amplitudes for a far field point a distance r from the origin, the result must be scaled by a factor $1/r$.

```
Far Field:
  Integration Sideset Name: [string]
  Nearfield Region: [string]
  Far Field Point: [string]
  X Axis: [string, default "1.0, 0.0, 0.0"]
  Y Axis: [string, default "0.0, 1.0, 0.0"]
  Z Axis: [string, default "0.0, 0.0, 1.0"]
  Interpolation Order: [int]
  Ensure Full Causality: [bool, default True]
```

`Integration Sideset Name`: Name of the sideset on which to compute the equivalent sources

`Nearfield Region`: An exodus mesh block which is adjacent to the integration sideset

`Far Field Point`: Point (theta, phi) at which the time-domain far field will be computed

`Interpolation Order`: Order of the polynomial interpolation for temporally interpolating the EMPIRE solution fields to the retarded time.

`Ensure Full Causality`: If true, restricts the output to the range where the assumption of E and H reaching zero on the integration surface by the end of the simulation does not hold.

`X Axis`: Specifies the x-axis, which determines the orientation of the spherical coordinate system that the far fields refer to

`Y Axis`: Specifies the y-axis, which determines the orientation of the spherical coordinate system that the far fields refer to

`Z Axis`: Specifies the z-axis, which determines the orientation of the spherical coordinate system that the far fields refer to

12.1.2.12. Transmission Line Solver Field

This mesh diagnostic will create a 1D exodus file for outputting spatially varying currents and voltages. It only works if there is a circuit network block. All transmission lines outputs are present throught the entire 1D mesh. The name of the transmission line is a prefix for the current and voltages. A conductor number is listed as a suffix.

13. RESTART

There are several basic options for the restart block which are used regardless of physics.

```
Restart:
   Write Checkpoint:
     Stride:
       Stride: [...] (no default)
     Number of Checkpoints: [int] (default 2)
     Name: [string] (default empire)
   Enable Restart: [bool] (default false)
   Restart from Checkpoint: [int] (default -1)
```

Empire enables restart by writing *checkpoint* binary files to disk, and subsequently reading them back in upon restart.

`Write Checkpoint` Block determining checkpoint restart file write options.

> `Stride`: A stride block which describes the intervals at which a restart checkpoint file is written to disk. The user must supply this block and define the stride through either the `Stride` or `Stride Time` parameters. The user is strongly encouraged to be judicious in choosing the dump frequency.
>
> `Number of Checkpoints`: The maximum number of checkpoint restart files to store on disk. Older dump files are deleted to make room for new ones as the simulation progresses.
>
> `Name`: The base name of the restart checkpoint files. Files will have the names *basename.dumpNumber.mpiRank*.

`Enable Restart`: Start clean or start from a restart dump. If there are no checkpoint files present, the simulation starts clean from time 0.

`Restart from Checkpoint`: The dump index of the restart checkpoint files to start from. The default of -1 means to start from the last (most recent) dump. The dump index starts from zero(0).

Standard Caveats:

1. Restart is not guaranteed to be backwards compatible. Older restart files may not work with newer executables.

2. Restart is not guaranteed to be cross-platform compatible. Restart files from filesystem A may not work on filesystem B.

3. Restart is not guaranteed to be compiler independent. Restart files generated by a gnu/gcc executable may not work with an intel or clang executable.
4. Restart is not independent of the number of mpi cores. A simulation run on 64 cores cannot be restarted on a different number of cores.
5. Restart is not independent of the parallel domain decomposition. The parallel domain decomposition must not change upon restart.

The user is strongly encouraged to *run a restart using exactly the same environment as the original run*. This implies the same binary executable running on the same hardware, the same mpi, the same number of cores, and the same filesystem. Mixing and matching these is not supported.

Here is a concrete example:

```
Restart:
  Write Checkpoint:
    Stride:
      Stride: 10
    Number of Checkpoints: 3
    Name: AlfvenWave
  Enable Restart: true
  Restart from Checkpoint: 3
```

This means:

- Write a restart dump every 10 cycles.
- Save a maximum of 3 restart dumps on disk.
- Restart files have the naming convention *AlfvenWave.dumpNumber.mpiRank*.
- Restart if checkpoint files are present.
- Restart from checkpoint dump files with dumpNumber = 3.

14. GENERAL UTILITIES

EMPIRE has some general utilities which are used throughout the input deck. This chapter gives the format for these utility features. They all have the format of:

```
Utility Name:
  Key: Value
  Key: Value
  Sublist:
    Key: Value
```

Obviously the keys and values are specific to the utility. These are often subsections to other input parameters like output specification.

14.1. Geometry

Some features of the code use geometric primitives to define regions of influence. There are several geometric primitives available to the user. The following sections will describe these primitives. Other attributes can be combined with a geometric primitive which define what happens inside that geometry.

Geometric primitives can be sampled randomly, meaning, points can be drawn randomly within the primitive. Because of this, not all shapes uniformly project from three-dimensions into two-dimensions. An example would be a spherical shape. Random distributions in three-dimensions will have a higher probability of being near the origin of the projected circle.

14.1.1. All

This defines the full domain of the problem, it is the simplest of all the geometric primitives

```
All:
```

14.1.2. Point

A “Point” represents a delta function in space. It only requires a single value to be defined, the position as follows.

```
Point:
  Position: x, y, x, position of the point (no default)
```

14.1.3. Sphere

A “Sphere” can be defined by the center point and a radius. It is a three dimensional shape, and does not always reduce to circle in two-dimensions. The definition of the “Sphere” is as follows.

```
Sphere:
  Center: x, y, x, position of the center (no default)
  Radius: r, radius of the sphere (no default)
```

14.1.4. Block

A “Block” represents a rectangular region of space which is coordinate aligned. It is defined by the lower left and upper right corners.

```
Block:
  Lower: x, y, x, position of the lower left corner (no default)
  Upper: x, y, x, position of the upper right corner (no default)
```

14.2. Stride

The input block “Stride” provides the analyst the ability to control the frequency that something happens. This something can be quite general, but outputting results is one natural use for such a feature. This stride can be given in units of simulation time or in units of time steps. If some of the specification is given in units of simulation time, all the inputs must be in simulation, and similarly for time steps.

The input block for “Stride” is as follows.

```
Stride:
  Stride: Number of steps between events (default=1)
  Start Stride: First time-step evaluated to true (default=0)
  Stop Stride: Last time-step evaluated to true (default=Inf)
```

— or —

```
Stride:
  Stride Time: Number of simulation seconds between events (default=0.0)
  Start Time: First simulation time evaluated to true (default=0.0)
  Stop Time: Last simulation time evaluated to true (default=Inf)
```

With an empty stride definition one can see that you would trigger the event that stride controls every time-step. First and last steps can be treated specially, one can enable these specifically as seen in this example.

```
Stride:
  Force First: true
  Force Last: true
  Start Stride: 10
  Stop Stride: 20
```

One would get the first and the last events set to true, then each time-step between [10, 20] will also evaluate to true.

Now the behavior when one uses a “Time Stride” that does not align with a time step size is that the time between the events will be **at least** the “Time Stride” provided. That means, if one has a time step of 1.0 and a “Time Stride” of 1.1, you will effectively be true every other time step. If one wants fine control of the frequency then it is suggested that one uses “Stride” rather than “Time Stride”.

14.3. Tabular Data

Many of the inputs can be controlled via a tabular data file. This allows one to specify the data either inline, or in a file written to disk. In the case of the data coming from a file, the data file has the format of an index which is a floating point number, followed by a number of values for that particular index.

```
# comments
#blanks are o.k.

index value1 value2 value3
# commas are o.k. as well
index, value1, value2, value3
# even mixtures
index value1, value2, value3
#it will error out if there is not the same number of columns
```

The number of values must be the same for all rows in the table and match what the code expects for a particular section.

The input deck can reference a “Tabular Data” section by just referencing the file or the user has the ability to inline the table directly into the code. Currently, the inline only allows for tables of one width. The format for specifying “Tabular Data” is as follows.

```
My Tabular Data:
  File Name: Name of file containing data (no default)
  Interpolation Type: Type of interpolation between indices
                      (default=Linear)
  Pre Extrapolation: How to handle data prior to first index
                     (default=Truncate)
  Post Extrapolation: How to handle data after the last index
                      (default=Truncate)
```

“Interpolation Type”, controls how one interpolates between two valid indices, there are only two options currently, “Linear” or “Nearest”. Once one has stepped outside the bounds of the indices, extrapolation needs to be done. Here we have two forms of extrapolation, “Truncate” and “Extend”. If one selects “Truncate” the value assumed outside the range of valid indices are assumed to be 0.0. “Extend” assumes that the first, or last value are extended for all points outside the indices.

As mentioned previously the table values can be supplied either inline or from a file. The inline option has the restriction of only allowing one column of data besides the index column. This usage replaces the “File Name” flag as follows

```
My Inlined Tabular Data:
  Data:
    Index: Value
    Index: Value
...
```

All the other options are also available as defined previously.

14.4. RTC Functions

Many features in EMPIRE allow the specification of some quantity with a runtime-compiled (RTC) function. RTC functions have a syntax similar to the C language and several features are supported including conditional statements. This functinality is taken from the Pamgen finite element library. At time of writing, documentation on the features supported by RTC functions can be found at
https://github.com/trilinos/Trilinos/blob/master/packages/pamgen/rtcompiler/rtclang.tex

APPENDIX A. Documentation of Python Utilities

The EMPIRE team has developed a workflow for running and analyzing tests that heavily relies on tools written in Python. This appendix contains the documentation for public functions (“docstrings”) if the module or function has one. These docstrings are written to help people understand the call signature of the function, what it does, as well as what is returned.

A.1. “empire” module

No docstring found.

A.1.1. empire.ArchiveResults()

No docstring found.

A.1.2. empire.Timer()

```
This class can be used in a 'with' statement to automatically time
whatever is in the 'with' statement and save that to the dictionary
that is passed in.

For example, this code:

db = {}
with Timer('spam', db):
    print("Going to sleep")
    time.sleep(1.234)
print(db)

Produces this output:

**** Start Timer 'spam'
Going to sleep
**** End Timer 'spam'
{'Timer: spam': 1.2343378067016602}
```

A.1.3. empire.allocate_resources()

```
allocate_resources() is a function designed to make running EMPIRE
on different platforms easier by seamlessly taking care of sockets,
nodes, threads, and MPI ranks.

Parameters
----------
resource_type : str, optional
    The resource type for the run. Must be either "cpu" or "gpu".
    Default is "cpu".

n_allocated_resources : int, optional
    The total number of resources that are available to run on.
    Default is `1`.

resource_ids : List[int], optional
    The IDs of resources. Only used when `resource_type` is "gpu"
    and when the host we're running on is not a batch system. The
    length of the list must be equal to `n_allocated_resources`.
    Default is None.

n_threads : int, optional
    The desired number of threads per parallel process
    (OMP_NUM_THREADS), default is `1`.

n_ranks_per_socket : None or int, optional
    The number of parallel ranks per socket. If `None`, will set to
    (resources per socket // n_threads). Default is `None`.

n_sockets_per_node : None or int, optional
    If the machine you are running on is not recognized, you can
    define this value manually. If `None` the function will look
    up the hardware environment. Default is `None`.

n_cpus_per_socket : None or int, optional
    If the machine you are running on is not recognized, you can
    define this value manually. If `None` the function will look
    up the hardware environment. Default is `None`.

n_gpus_per_socket : None or int, optional
    If the machine you are running on is not recognized, you can
    define this value manually. If `None` the function will look
    up the hardware environment. Default is `None`.

allow_undersubscribe : bool, optional
```

```
    If False, cause an error if the number of cores used is less
    than `n_allocated_cores`. When `n_ranks_per_socket` is None
    and the function determines allocation, this is set to `True`.
    Default is `False`.

allow_oversubscribe : bool, optional
    If False, cause an error if the number of cores used is more
    than `n_allocated_resources`. Default is `False`

Returns
-------
hwenv : dict
    The hardware environment returned as a dict with the following:
        `allocated_resources` : int
            equal to `n_allocated_resources`
        `nodes` : int or None
            Number of nodes used (if applicable) else `None`
        `sockets` : int or None
            Number of sockets used (if applicable) else `None`
        `ranks` : int
            Number of parallel ranks
        `ranks_per_socket` : int or None
            Number of parallel ranks per socket (if applicable) else `None`
        `threads_per_rank` : int
            Equal to n_threads
        `used_resources` : int
            Equal to the total number of cores used

Raises
------
TypeError
    If `n_allocated_resources` is not an int
    If `resource_type` is not a str
    If `resource_type` is not either 'cpu' or 'gpu'
    If `resource_ids` is not a list of ints or None
    If `len(resource_ids)` is not equal to `n_allocated_resources`
    If `n_threads` is not an int
    If `n_cpus_per_socket` is not an int or None
    If `n_gpus_per_socket` is not an int or None
    If `n_sockets_per_node` is not an int or None
    If `allow_undersubscribe` is not a boolean
    If `allow_oversubscribe` is not a boolean
Exception
    If resources are undersubscribed and allow_undersubscribe=False
    If resources are oversubscribed and allow_oversubscribe=False
    For HPC usage, if n_cpus_per_socket or n_sockets per node are
        None and the system in not recognized.
    For HPC usage, if n_gpus_per_socket or n_sockets per node are
        None and the system in not recognized.
```

A.1.4. empire.empire_history_file_statistics()

```
Merges columns from one or more EMPIRE history files into a statistical
representation with means and standard deviations.

Parameters
----------
ifiles : list of str or pathlib.Path
    The files to be read in and processed.

ofile: str or pathlib.Path or None, optional
    Filename of the file to be written. Will overwrite existing files.
    If `None`, then the output file will not be written.
    Default is `None`.

columns : Dict[str, str] or None, optional
    If None assume all columns except `index_column` are to be merged
    (assuming each file in `ifiles` has the same columns). If a dict,
    must be of the format ("key", "regex") where it will create columns
    called "{key}_mean" and "{key}_stddev" and process all columns that
    match the accompanying regex. Default is `None`.

index_column : str, optional
    The column to be copied over and to be used to verify that the
    columns can be merged. All files must have this column and it
    must be the same in all files. Default is "Simulation_time".

ddof : int, optional
    Means Delta Degrees of Freedom. The divisor used in stddev
    calculations is ``N - ddof``, where ``N`` represents the number of
    elements. Default is 1 (sample stddev).

Returns
-------
OrderedDict[str, numpy.array]
    Returns the computed output data

Raises
------
Exception
    if no input files are given (empty list)
    if columns are of uneven lengths in the same file
    if columns are of uneven lengths between files
    if any file is missing the `index_column`
    if not all index columns are the same
    if `columns` is not a dict or None
    if any keys or values of `columns` are not strings
    if any value of `columns` is not a valid regex
    if duplicate output column names are requested
    if there aren't at least two columns to perform analysis on
```

A.1.5. empire.load_test_results()

```

Load the dictionary saved by the function save_test_results().
```

A.1.6. empire.read_exodus()

```

read_exodus() is yet another python implementation of an exodus
reader. It is intended for use in plasma physics codes where the
mesh does not change with time.

Parameters
----------
exo_f : str
    The name of the exodus file to read. If reading parallel exodus
    dumps, the base exodus file name.

var_list : list of str
    A list of all the field names that you want output.

cycle : int
    An index to a list of the cycles in the file. May use python
    index conventions (`-1` for last, `-2` for penultimate, etc)

var_type : str ("NODE" or "CELL"), optional
    Which type of field is being queries. Default is `NODE`.

processor_count : int or, optional
    The number of parallel exodus files there are in the batch.
    Default is `0` (serial exodus dump).

Returns
-------
data : dict of numpy.array
    A dictionary that looks similar to the following and will
    always have 'coordx', 'coordy', 'coordz', and one of
    'node_num_map' or 'elem_num_map' (depending on `var_type`)::
        {
         "coordx": np.array([...]),
         "coordy": np.array([...]),
         "coordz": np.array([...]),
         "node_num_map": np.array([...]),
         "var1": np.array([...]),
         ...
        }

Raises
------
Exception
    If expected exodus files do not exist
    If var_type is not either "NODE" or "CELL"
    If decoding strings from the exodus file fails
    If name in `var_list` does not exist in exodus file
```

A.1.7. empire.read_h5part()

```

This function reads an h5part file and returns the specified step
as a dictionary.

Parameters
----------
ifile : string
    Name of the input file to process.

step : int
    The index of the step you want to access. Python indexing is
    allowed (e.g. 0 is the first, -1 is the last step).

var_list : list of strings, optional
    The list of variables that you want to output. Default is
    to output only coordinates. Note: the (x,y,z) coordinates are
    always returned and they are called (coordx,coordy,coordz).

Returns
-------
data : dict of numpy.array
    A dictionary that looks similar to the following and will
    always have 'coordx', 'coordy', and 'coordz':
        {
         "coordx": np.array([...]),
         "coordy": np.array([...]),
         "coordz": np.array([...]),
         "var1": np.array([...]),
        ...
      }
```

A.1.8. empire.render_jinja2()

```
This function is a wrapper around the jinja2 module and streamlines
the processing of templates. It also implements a way for variables
to be exported from the template.

This function also automatically loads the EMPIRE constants into
the namespace of every file rendered.

Parameters
----------
ifile : str
    The file name or path to the jinja2 template to process.

ofile : str or None, optional
    The file name or path where the output should be written.
    Default is `None` (print to stdout).

preload : dict, optional
    Dictionary of values to make available to the template rendering
    engine. Default is `None` (no extra variables).

Returns
-------
Dict
    A dictionary of all the exported variables and their values.
```

A.1.9. empire.run_aprepro()

```
Runs an executable built by Trilinos.

Parameters
----------
cmd : str
    The string that you would expect to pass to the executable
    EXCLUDING the executable name.

cwd : str or None, optional
    The desired working directory that you wish to execute this in.
    Default is `None` (your current working directory).

```

```
file_out : file-like object (e.g. supports ``.write())
    File-like object to save stdout to. Default is None.

allow_errors : bool
    Allow aprepro to not fail when errors are encountered.
    Default is `False` (Includes --errors_fatal).

Returns
-------
None

Raises
------
Exception
    If install_info.json is not found (which it uses to find where
        the executable is installed). Likely means that you are
        using the EMPIRE source and not a build.
    If the executable doesn't exist where it is expected.
    If the executable is not executable.
    If the executable returns with a non-zero return code.
```

A.1.10. empire.run_cable()

```

This function runs EMPIRE-Cable. It accepts an input file and will
assemble the run command and execute it.

Parameters
----------
ifile : string
    Name of the input file to run

extra_args: string, optional
    Extra arguments to pass to the EMPIRE executable. Default is "".

*** NOTE: Accepts all options avaiable to `allocate_resources()`
***       and will use the values returned to run or you may
***       manually define the following:

ranks : int, optional
    Number of MPI ranks to use. Default defined by
    `allocate_resources()`.

threads_per_rank : int, optional
    Number of threads per process, overrides OMP_NUM_THREADS.
    Default defined by `allocate_resources()`.

```

```
ranks_per_socket: int or None, optional
    If running on a HPC system, number of MPI ranks per socket.
    Default defined by `allocate_resources()`.

resource_ids: (list of int) or None, optional
    A list of Kokkos device IDs to be used for GPU/accelerator
    runs. Length must be equal to `ranks`.  Default is `None`, for
    no GPU/accelerator in the run.

resource_type : str, optional
    Must be set to either "cpu" or "gpu". Default is "cpu".

load_balancing : bool, optional
    Turns on load balancing by setting EMPIRE_VT="1". Default is `False`.

write_script : bool, optional
    Writes out a file called "run_empire.sh" with the command used to
    run EMPIRE. Default is `True`.

Returns
-------
retcode : int
    The return code of the spawned process.

Raises
------
Exception
    If module `script_util` is not loaded.
    If `ranks` is negative
IOError
    If desired EMPIRE executable is not found.
```

A.1.11. empire.run_cubit()

```
This function calls cubit on an input journal file. It doesn't
return anything as I don't think cubit ever returns with a non-zero
return code.

Parameters
----------
jou : str
    Name of the file for Cubit to run.

cubit_version : str or None, optional
    The cubit version you want to run with. If None, run with the
    first Cubit found. Default is "15.5".
```

```
Returns
-------
None

Raises
------
Exception
    If the file in `jou` is not found
    If the desired version of cubit isn't found
    If the cubit executable isn't executable
    If the call to Cubit yielded a nonzero return code
```

A.1.12. empire.run_decomp()

```
Runs an executable built by Trilinos.

Parameters
----------
cmd : str
    The string that you would expect to pass to the executable
    EXCLUDING the executable name.

cwd : str or None, optional
    The desired working directory that you wish to execute this in.
    Default is `None` (your current working directory).

file_out : file-like object (e.g. supports ``.write())
    File-like object to save stdout to. Default is None.

allow_errors : bool
    Allow aprepro to not fail when errors are encountered.
    Default is `False` (Includes --errors_fatal).

Returns
-------
None

Raises
------
Exception
    If install_info.json is not found (which it uses to find where
        the executable is installed). Likely means that you are
        using the EMPIRE source and not a build.
    If the executable doesn't exist where it is expected.
    If the executable is not executable.
    If the executable returns with a non-zero return code.
```

A.1.13. empire.run_em()

```

This function runs EMPIRE-EM. It accepts an input file and will
assemble the run command and execute it.

Parameters
----------
ifile : string
    Name of the input file to run

extra_args: string, optional
    Extra arguments to pass to the EMPIRE executable. Default is "".

*** NOTE: Accepts all options avaiable to `allocate_resources()`
***       and will use the values returned to run or you may
***       manually define the following:

ranks : int, optional
    Number of MPI ranks to use. Default defined by
    `allocate_resources()`.

threads_per_rank : int, optional
    Number of threads per process, overrides OMP_NUM_THREADS.
    Default defined by `allocate_resources()`.

ranks_per_socket: int or None, optional
    If running on a HPC system, number of MPI ranks per socket.
    Default defined by `allocate_resources()`.

resource_ids: (list of int) or None, optional
    A list of Kokkos device IDs to be used for GPU/accelerator
    runs. Length must be equal to `ranks`.  Default is `None`, for
    no GPU/accelerator in the run.

resource_type : str, optional
    Must be set to either "cpu" or "gpu". Default is "cpu".

load_balancing : bool, optional
    Turns on load balancing by setting EMPIRE_VT="1". Default is `False`.

write_script : bool, optional
    Writes out a file called "run_empire.sh" with the command used to
    run EMPIRE. Default is `True`.

```

```
Returns
-------
retcode : int
    The return code of the spawned process.

Raises
------
Exception
    If module `script_util` is not loaded.
    If `ranks` is negative
IOError
    If desired EMPIRE executable is not found.
```

A.1.14. empire.run_epu()

```

Runs an executable built by Trilinos.

Parameters
----------
cmd : str
    The string that you would expect to pass to the executable
    EXCLUDING the executable name.

cwd : str or None, optional
    The desired working directory that you wish to execute this in.
    Default is `None` (your current working directory).

file_out : file-like object (e.g. supports ``.write())
    File-like object to save stdout to. Default is None.

allow_errors : bool
    Allow aprepro to not fail when errors are encountered.
    Default is `False` (Includes --errors_fatal).

Returns
-------
None

Raises
------
Exception
    If install_info.json is not found (which it uses to find where
        the executable is installed). Likely means that you are
        using the EMPIRE source and not a build.
    If the executable doesn't exist where it is expected.
    If the executable is not executable.
    If the executable returns with a non-zero return code.
```

A.1.15. empire.run_epup()

```

Runs an executable built by Trilinos.

Parameters
----------
cmd : str
    The string that you would expect to pass to the executable
    EXCLUDING the executable name.

cwd : str or None, optional
    The desired working directory that you wish to execute this in.
    Default is `None` (your current working directory).

file_out : file-like object (e.g. supports ``.write())
    File-like object to save stdout to. Default is None.

allow_errors : bool
    Allow aprepro to not fail when errors are encountered.
    Default is `False` (Includes --errors_fatal).

Returns
-------
None

Raises
------
Exception
    If install_info.json is not found (which it uses to find where
        the executable is installed). Likely means that you are
        using the EMPIRE source and not a build.
    If the executable doesn't exist where it is expected.
    If the executable is not executable.
    If the executable returns with a non-zero return code.
```

A.1.16. empire.run_exodiff()

```

Runs an executable built by Trilinos.

Parameters
----------
cmd : str
```

```
    The string that you would expect to pass to the executable
    EXCLUDING the executable name.

cwd : str or None, optional
    The desired working directory that you wish to execute this in.
    Default is `None` (your current working directory).

file_out : file-like object (e.g. supports ``.write())
    File-like object to save stdout to. Default is None.

allow_errors : bool
    Allow aprepro to not fail when errors are encountered.
    Default is `False` (Includes --errors_fatal).

Returns
-------
None

Raises
------
Exception
    If install_info.json is not found (which it uses to find where
        the executable is installed). Likely means that you are
        using the EMPIRE source and not a build.
    If the executable doesn't exist where it is expected.
    If the executable is not executable.
    If the executable returns with a non-zero return code.
```

A.1.17. empire.run_fluid()

```
This function runs EMPIRE-Fluid. It accepts an input file and will
assemble the run command and execute it.

Parameters
----------
ifile : string
    Name of the input file to run

extra_args: string, optional
    Extra arguments to pass to the EMPIRE executable. Default is "".

*** NOTE: Accepts all options avaiable to `allocate_resources()`
***       and will use the values returned to run or you may
***       manually define the following:

ranks : int, optional
```

```
    Number of MPI ranks to use. Default defined by
    `allocate_resources()`.

threads_per_rank : int, optional
    Number of threads per process, overrides OMP_NUM_THREADS.
    Default defined by `allocate_resources()`.

ranks_per_socket: int or None, optional
    If running on a HPC system, number of MPI ranks per socket.
    Default defined by `allocate_resources()`.

resource_ids: (list of int) or None, optional
    A list of Kokkos device IDs to be used for GPU/accelerator
    runs. Length must be equal to `ranks`.  Default is `None`, for
    no GPU/accelerator in the run.

resource_type : str, optional
    Must be set to either "cpu" or "gpu". Default is "cpu".

load_balancing : bool, optional
    Turns on load balancing by setting EMPIRE_VT="1". Default is `False`.

write_script : bool, optional
    Writes out a file called "run_empire.sh" with the command used to
    run EMPIRE. Default is `True`.

Returns
-------
retcode : int
    The return code of the spawned process.

Raises
------
Exception
    If module `script_util` is not loaded.
    If `ranks` is negative
IOError
    If desired EMPIRE executable is not found.
```

A.1.18. empire.run_hybrid()

```
This function runs EMPIRE-Hybrid. It accepts an input file and will
assemble the run command and execute it.

Parameters
----------
```

```
ifile : string
    Name of the input file to run

extra_args: string, optional
    Extra arguments to pass to the EMPIRE executable. Default is "".

*** NOTE: Accepts all options avaiable to `allocate_resources()`
***       and will use the values returned to run or you may
***       manually define the following:

ranks : int, optional
    Number of MPI ranks to use. Default defined by
    `allocate_resources()`.

threads_per_rank : int, optional
    Number of threads per process, overrides OMP_NUM_THREADS.
    Default defined by `allocate_resources()`.

ranks_per_socket: int or None, optional
    If running on a HPC system, number of MPI ranks per socket.
    Default defined by `allocate_resources()`.

resource_ids: (list of int) or None, optional
    A list of Kokkos device IDs to be used for GPU/accelerator
    runs. Length must be equal to `ranks`.  Default is `None`, for
    no GPU/accelerator in the run.

resource_type : str, optional
    Must be set to either "cpu" or "gpu". Default is "cpu".

load_balancing : bool, optional
    Turns on load balancing by setting EMPIRE_VT="1". Default is `False`.

write_script : bool, optional
    Writes out a file called "run_empire.sh" with the command used to
    run EMPIRE. Default is `True`.

Returns
-------
retcode : int
    The return code of the spawned process.

Raises
------
Exception
    If module `script_util` is not loaded.
    If `ranks` is negative
```

```
IOError
    If desired EMPIRE executable is not found.
```

A.1.19. empire.run_itspff2hdf5()

```
This function will run the EMPIRE TPL 'itspff2hdf5'. If you really want or
need to specify, you can set the 'cwd' for running the executable.

Parameters
----------
cmd : str
    The string that you would expect to pass to itspff2hdf5 EXCLUDING
    the initial 'itspff2hdf5'.

cwd : str or None, optional
    The desired working directory that you wish to execute this in.
    Default is `None` (your current working directory).

Returns
-------
None

Raises
------
Exception
    If install_info.json is not found (which it uses to find where
        itspff2hdf5 is installed). Likely means that you are using the
        EMPIRE source and not a build.
    If itspff2hdf5 doesn't exist where it is expected.
    If itspff2hdf5 is not executable.
    If itspff2hdf5 returns with a non-zero return code.
```

A.1.20. empire.run_pic()

```
This function runs EMPIRE-PIC. It accepts an input file and will
assemble the run command and execute it.

Parameters
----------
ifile : string
    Name of the input file to run

extra_args: string, optional
    Extra arguments to pass to the EMPIRE executable. Default is "".
```

```
*** NOTE: Accepts all options avaiable to `allocate_resources()`
***       and will use the values returned to run or you may
***       manually define the following:

ranks : int, optional
    Number of MPI ranks to use. Default defined by
    `allocate_resources()`.

threads_per_rank : int, optional
    Number of threads per process, overrides OMP_NUM_THREADS.
    Default defined by `allocate_resources()`.

ranks_per_socket: int or None, optional
    If running on a HPC system, number of MPI ranks per socket.
    Default defined by `allocate_resources()`.

resource_ids: (list of int) or None, optional
    A list of Kokkos device IDs to be used for GPU/accelerator
    runs. Length must be equal to `ranks`.  Default is `None`, for
    no GPU/accelerator in the run.

resource_type : str, optional
    Must be set to either "cpu" or "gpu". Default is "cpu".

load_balancing : bool, optional
    Turns on load balancing by setting EMPIRE_VT="1". Default is `False`.

write_script : bool, optional
    Writes out a file called "run_empire.sh" with the command used to
    run EMPIRE. Default is `True`.

Returns
-------
retcode : int
    The return code of the spawned process.

Raises
------
Exception
    If module `script_util` is not loaded.
    If `ranks` is negative
IOError
    If desired EMPIRE executable is not found.
```

DISTRIBUTION

Hardcopy—External

Number of Copies	Name(s)	Company Name and Company Mailing Address

Hardcopy—Internal

Number of Copies	Name	Org.	Mailstop
1	Richard M. J. Kramer	1351	1152

Email—Internal [redacted]

Name	Org.	Sandia Email Address
Technical Library	01177	libref@sandia.gov