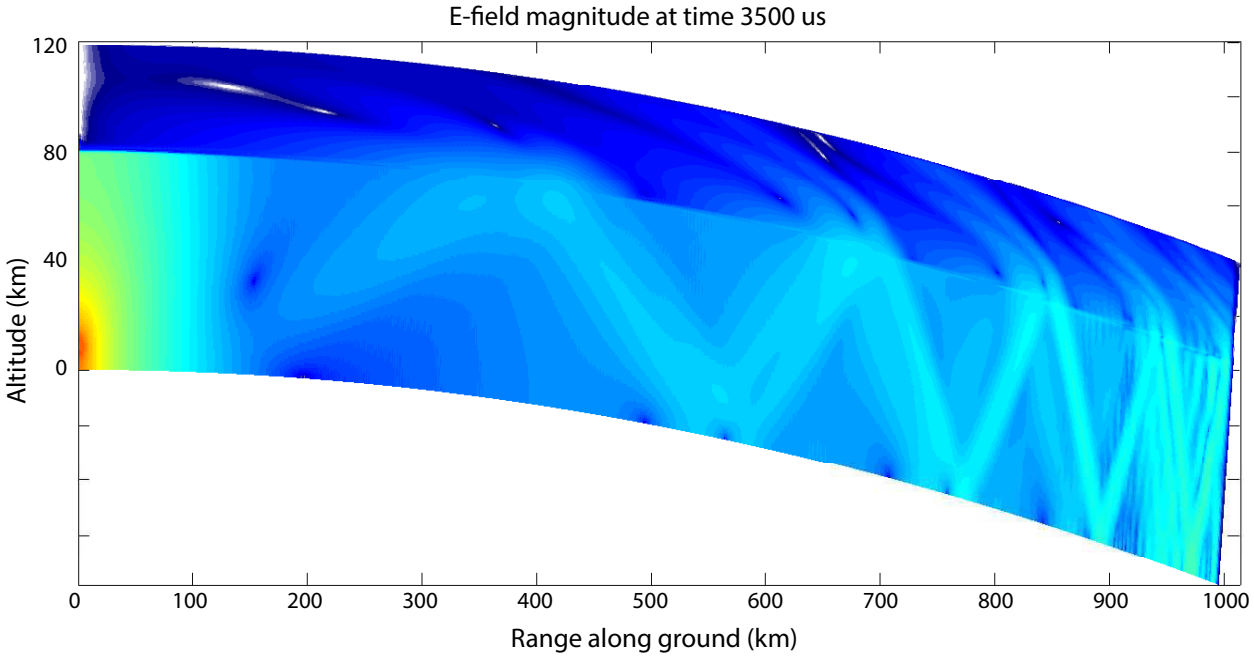
Stanford EMP Model

Installation and Operating Manual 1.0

Robert A. Marshall, Stanford University
ram80@stanford.edu

October 29, 2014



Contents

1. Explanation of Directories

1. emp1: includes the source code for the 1D EMP model, and the makefile required to compile the code. After compilation, the executable will also reside here, but you can move it if you wish.
2. emp2: includes the source code for the 2D EMP model, and the makefile required to compile the code.
3. emp3: includes the source code for the 3D EMP model, and the makefile required to compile the code.
4. setup: includes Matlab code to setup simulations in 1D, 2D, and 3D.
5. analysis: includes Matlab code to plot results and do some analysis. You'll likely have to write some of your own code!

2. Installation

Compatibility

The model is compatible with unix based systems (linux and OS X) - for Windows users, it will probably work within Cygwin, but I haven't tried it and don't plan to. Good luck.

Download

Download the source code, setup codes, and analysis codes from www.github.com/ram80unit/empcodes. The simplest way is to download a zip file from the link at lower right; users versed in git and github can fork the repository.

Install

Installation is simple: unzip the zip file into a directory from which you would like to run the model.

Required Packages

The only package required that's remotely out of the ordinary is OpenMP, which you can easily install on any linux or OS X system. Even if your computer only has one core, you'll need OpenMP for the code to compile and run correctly. (There are notes below on how to change the code if you really, really don't want to install OpenMP.)

Compile

From a terminal,
1. cd into the source directory, either emp1, emp2, or emp3 as needed
2. run "make clean" and then "make". There is no "make install" required. When you run the code it will grab the compiled executable from these directories.

You'll get errors at this point if you don't have OpenMP; otherwise it should compile error- and warning-free.

The code is compiled by default with g++ and optimized with -O4. If you wish to use a different C compiler, go for it and let me know how it goes.

If you wish to remove OpenMP

If for some reason you want to run on a single core and don't want to install OpenMP, you'll have modify the codes a bit.

1. In the makefile in each source directory, remove any references to $(CFLAGS) and $(LFLAGS).
2. In the source codes (e.g., emp1/emp1d-1.0.cpp),
    1. remove "#include <omp.h>" near the top
    2. Search for "#pragma" (e.g., line 515) and remove the ENTIRE LINE. There are quite a few of these lines.

The code should then compile and be able to run on a single core.

3. Running Simulations

You will always launch simulations (runs) by running emp1Dbatch.m, emp2Dbatch.m, or emp3Dbatch.m. A "batch" of jobs can include a vector of different inputs, or a single job if you wish. Let's first see what you need to do to run the code as downloaded, without changing any parameters.

First Run

1. Start by opening Matlab (sorry IDL and Python users). cd into the setup directory.
2. You will need to make a few changes for your system. For a 2D run:
    - In emp2Dbatch.m, change the "toprundir" in line 14; this is where all of the runs will occur and save outputs.
    - In emp2Ddefaults.m, line 4, change the path to exedir, the location of the executable.
    - In emp2Ddefaults.m, lines 105 and 106, change the name of the cluster you are using and the number of nodes (processor cores).
    - The model launches jobs using qsub, which requires a batch server running on your cluster. If you wish to run a job on a single system without a queue manager, set in.cluster = 'local' on line 105 of emp2Ddefaults.m.
    - Make the same changes to the 1D or 3D versions as needed.
3. Launch jobs by running emp2Dbatch.m in Matlab. You will get some information about the grid, time estimates (don't believe them), and a confirmation that the job was submitted.

When you submit the job locally (in.cluster = 'local'), the code will run with OpenMP on as many cores as are available; this is the default behavior for the GNU g++ compiler.

To follow the progress of the simulation, see the Diagnostics section below.

Setting Parameters

Changing simulation parameters is easy. There are two places to do this:
1. Change any of the parameters in emp2Ddefaults.m.
2. Add lines in emp2Dbatch.m that override the inputs set by emp2Ddefaults.m.

There are a few issues to be aware of when you change parameters:
1. Grid size: if you change the grid resolution (dr1, dr2, drange), it will cause the grid to grow. This will increase memory requirements and runtimes. You may also need to reduce the time step (dt, line 53) to ensure stability - it's up to you to do this! It is not dynamically updated (although the Matlab code gives you an estimate when it is run).
2. Changing the grid size also affects the numerical dispersion. Generally, you need 10-20 grid cells per the wavelength, for the shortest wavelength (highest frequency) in the simulation. The highest frequency can be limited by the fcut parameter (line 99), which will be used to low-pass filter the source input.

Details of all input parameters are discussed in Section 7.

Diagnostics

In any batch server, you can see your active jobs by running qstat from any directory.

You can watch what's happening with the simulation as it runs to get some valuable information:
1. cd into the run directory of interest.
2. open log.txt. The log file, updated while the simulation is running, will tell you some information about the run parameters, and some other useful information:
    1. Memory usage: this adds up the total RAM used by the simulation. This will let you know how big you can make your simulation.
    2. Run time: the simulation times itself up to 50 time steps, then extrapolates how long it will take to finish.
    3. abs(Er) estimates: every so often, the simulation will output the maximum Er (vertical) in the entire grid. This will tell you if things have gone unstable, or if there are other issues: If you start to see max abs(Er) = 0, or growing exponentially, that's probably bad. When this happens I usually find it's because of the instability in the boundary condition (see Section 5 below).

Note, as below in Section 4, that you can also plot the field outputs as the simulation runs, since the output files are appended throughout the simulation, rather than being written all at the end of the simulation.

4. Analysis

Plotting field evolution

The first thing to do to analyze your simulation results is to plot the 2D fields and watch them evolve in time. An animation is plotted using plot2Dresults.m, and the 1D and 3D equivalents. As before, you'll need to change the path to your run, and the run name, in lines 5 and 6 of this script.

As mentioned above, you can start doing this before the run is finished; it will simply stop at the most recent time step that was output to the files. This way you don't have to wait to the end to see if the simulation is working (or if something has gone awry!).

By default, this script plots Er (vertical component of E), S (the Poynting flux), collision frequency, electron density change, optical emissions in N2 1P, and integrated J-dot-E heating. The fields that get plotted can be changed in lines 188-207 (note that I make the plots laterally symmetric in lines 188-193).

Plotting Probe fields

In the setup for the simulation, you were given the option to place a vector of "probes" at arbitrary locations in the simulation space. These will store the six field components for all time steps at the locations marked. You can read these using getProbeFields.m and plot them with plotProbeSignals.m and the 3D equivalent (1D version coming soon!).

This script really just shows you how to read the probe output file and generate a time vector; you'll probably want to create your own script to do any real analysis. I do also create a frequency vector and plot FFTs of the probe signals, which may be of use to some users.

Plotting elve evolution

Optical emissions in four bands are plotted with plot2Delve.m and plot3Delve.m. (Note that the Elve.dat output file contains 5 bands, but we only plot 5 just to make the plot symmetric. Ultimately the N2 1P is the brightest by far!)

Note also that Elve.dat is output at the end of the simulation, so you can't plot it until the simulation is finished.

In addition to plotting an animation of the elve, these scripts will produce a time-integrated elve image (to compare to camera data), and a PIPER-photometer integrated result (to compare to PIPER data). If you don't have a PIPER instrument to compare to, you should build one! I'll give you the design!

There is another elve plotting file, plot2DelveAuger.m. This was specifically written to view elves as they were seen at the Pierre Auger Observatory by my colleague Roberto Mussa. This probably won't be of great use to the wider community, but it looks cool so you should try it!

5. Explanation of Some of the Subtleties

Stability, Dispersion, and Grid resolution

The FDTD method, in its explicit implementation, is stability limited by a minimum time step dt <= dx/c/sqrt(D), where dx is the smallest grid cell dimension, c is the speed of light and D is the number of dimensions. In an unmagnetized plasma, stability is further restricted by the maximum electron density. In a magnetized plasma, it is further restricted, but an analytical expression for the stability limit is not trivial. In practice, we can find the stability limit by trial and error. I have found that for the E and D region ionosphere (at night), a time step of 0.1 us is

sufficient for stability. (As the spatial step gets smaller though, you'll have to reduce dt by the equation above.)

Stability is different from accuracy, though. For high accuracy in the FDTD method, of course it must be stable, but in addition waves must be "well resolved". This imposes a limit on k.dx, the wavenumber times the grid cell size. You impose sources that have a maximum frequency (and thus minimum wavelength); in practice, the grid cell size dx should be at most 1/10th or 1/20th of a wavelength. Otherwise, high frequencies in the simulation will propagate at a different numerical velocity from c, and numerical dispersion will result (where different frequencies propagate at different velocities).

Things get worse in a plasma, because the wavelength changes in the medium; in fact, in the ionosphere the phase velocity becomes very slow, so the wavelength can become very short. For this reason, one often needs an even smaller grid resolution in the ionosphere. Fortunately, the EMP code is set up to use a variable vertical grid size, so you can keep larger cells in the free-space region of the lower atmosphere and refine the grid size in the ionosphere. (The grid resolution in the horizontal direction is kept constant; however, the effect on the horizontal wavelength in the ionosphere is much less pronounced.)

Maximum Altitude

In the 2D and 3D versions of the EMP model, we have successfully run simulations up to 150 km altitude. In 3D it is simply impractical to go higher for memory reasons. In 2D we have run into stability and accuracy issues due to the magnetic field and the small wavelengths that occur in the F-region of the ionosphere, requiring very small grid cells and thus very small time steps. In theory there is no inherent limitation in altitude, but in practice getting into the F region is difficult.

For this reason, and due to interest in nonlinear effects in the F-region due to lightning (F-region elves, maybe?), I created a 1D version of the model. In 1D, you can use very small grid cells and very small time steps and still run in a reasonable amount of time and memory, and simulate up to 500 km or higher. All six fields are still solved, so you can get Poynting fluxes and energy flow. The magnetic field is included and is not restricted to the vertical. However, since it's 1D, the amplitude of the input pulse does not decay as it does in the physical world. If you input a 100 V/m pulse at the ground, it will still be 100 V/m when it reaches the ionosphere. This should be kept in mind when you design 1D simulations and try to reach conclusions about real-world effects.

Instabilities in the corner (2D runs)

For 2D simulations, with a vertical magnetic field (the default), you will notice that if you run for a long time, a slow numerical instability arises in the upper right corner (i.e. opposite the source). This instability arises due to known problems with the PML boundary in a magnetized plasma. To my knowledge there is no existing algorithm that overcomes this instability. I have furthermore noticed that the problem is worse for larger ampitudes, and for low amplitudes (such as VLF transmitters) the problem doesn't arise in a reasonable amount of time.

For a detailed description of the problem, see Chevalier et al [2008]. In short, the problem arises because there is a component of the wave k-vector that points **out of** the PML, and the PML is

designed to absorb waves in the k-direction; as such, the PML actually causes the wave to **grow**. The existing solution is a) stop the simulation after the EMP reaches the corner, and/or b) make the simulation space larger (in range) than you really need.

Choice of ground method

There are three choices of ground method in the model, each with their advantages and drawbacks.

1. **PEC (perfect electrical conductor) ground**: in this method, the horizontal E components are set to zero on the ground. This forces perfect reflection, and the fields below the ground remain zero. If you don't care about the ground conductivity and its effect on dispersion and absorption of the ground wave, this is the method to use.
    1. Advantages: simplicity; no numerical dispersion
    2. Disadvantages: not physically realistic; cannot simulate effects of ground conductivity
2. "**Real" ground**: in this method, the ground is discretized by a number of grid cells (nground) with vertical resolution given by dr0 (lateral resolution is still drange). These grid cells are assigned permittivity and conductivity values from a table of values over the Earth's surface (in 2D; in 3D, a single value is used based on the latitude and longitude of the source). Maxwell's equations are solved as usual.
    1. Advantages: more physically satisfying, since the ground is being simulated with realistic parameters. You can also "see" the fields in the ground and see how they decay with depth.
    2. Disadvantages: due to the high conductivity and permittivity, the wavelength in the ground becomes very small, and it becomes hard to resolve a wavelength; this results in pretty bad numerical dispersion. You'll see this as "ringing" in the probe fields. In practice, I don't use this method anymore.
3. **SIBC (Surface Impedance Boundary Condition) method**: recently added, this method uses a Laplace-transformed analytical solution where the ground is modeled by an frequency-dependent impedance, determined by the real conductivity and permittivity values. The impedance can vary in space as above, and the Laplace transform puts it in the time domain. This method is described in detail by Oh and Schutt-Aine [1995]. I use this method by default now.
    1. Advantages: uses real permittivity and conductivity values; no numerical dispersion since waves do not propagate into the ground; simple implementation
    2. Disadvantages: not entirely physical, and a bit of a "hack" to the FDTD method; for that reason it's not totally satisfying, though the solutions are very accurate. Further, it does simulate the fields in the ground, in case you were interested in those.

Intracloud and CID sources

The code has recently been updated so that you can input any arbitrary current source, defined in altitude and time, in the 2D model. A similar update to the 3D model is forthcoming. However, the input current needs to be interpolated to the **r** vector (altitude) and time step of the simulation.

As it is, the code will run typical engineering models of the lightning return stroke current, and will add on a continuing current if you wish. To get IC sources, there is a flag (in.lightningtype) that you can set.

For CG sources, in.sourcealt sets the terminal altitude, and thus the channel length, of the current source. For an IC source, in.sourcealt sets the altitude of the center of the IC. It then also requires a channel length, given by in.chlength. The IC will be centered around in.sourcealt.

You also need to define whether the IC current pulse propagates up or down; this is managed by the sign of the return stroke speed. A positive in.rsspeed goes downwards.

For now, CIDs are simulated as ICs; in the near future I plan to implement the “bouncing wave” model of Nag et al [2012]. I also plan to add a “decaylength” parameter that will serve as the decay length for MTLE, or for MTLL within a bouncing CID.

Elve Integration

The integration of photon emissions in the EMP model to a camera on the ground is surprisingly simple. First, an array is created with the number of pixels and the number of time steps for the elve “movie”. Then, before the simulation begins, the code finds the grid cells whose photons will hit that pixel, and calculates a time delay based on distance. During simulation, as photons are produced, they are placed in the appropriate pixel bin and time bin based on the time in the simulation plus the propagation delay. Similarly, the 3D model finds the 3D grid cells that fall within the field-of-view of each pixel, and the time delay for propagation.

However, two subtleties result in significant, observable noise and “flickering” in the elve movie. First, no interpolation is done - photons are simply assigned to the nearest pixel. Second, the assignment is done from pixel to grid cell, rather than from grid cell to pixel. That means some grid cells might get skipped altogether.

The better method (which may get implemented in the future, but might not) would be to determine the pixel index of each grid cell and the time delay. Then, as the simulation runs, photons calculated at grid cell (i,j,k) get assigned to pixel m(i,j,k) with time t + delay(i,j,k). I just need to find the time to do this…

6. Special Notes on the 1D and 3D Models

1D Model Notes

1. The 1D model is cartesian and uses x as the direction of propagation (altitude).
2. The 1D model has spatial variation in the altitude direction only. This means it cannot properly decay fields as the source radiates from the ground. The input source is cast in electric field (V/m), and is input at a single grid cell only. The logical use of this model is to input a field amplitude that is expected at the base of the ionosphere (say, 20 V/m) and watch what happens as it propagates through the ionosphere. There will be considerable error due to the fact that the field does not have 1/r decay as it propagates; however, the model is still useful for looking at modes and amplitudes as the EMP propagates into the F-region.

3D Model Notes

1. In 2D, the EMP model grid is located at the “north pole” of a spherical coordinate system; hence the dimensions r and theta. In this setup the grid cells are very nearly rectangular and the same size throughout the grid. However, in 3D this wouldn’t work, so the 3D grid is shifted to the “equator”, i.e. centered around theta = pi/2. The theta direction then moves north and south, and phi moves along the equator. It is important to note that for large ranges, as the grid moves too far north or south, the grid cells get smaller in phi!
2. In the 3D grid, the ground is assumed to take a single value of permittivity and conductivity. This just simplified the programming, with the justification that you’re usually simulating a small region anyway!
3. The 3D model outputs “slices” of the fields for animation and plotting. Slices are taken through the center of the grid in each of the three planes; however, all of the slices are the “r” components of E, J and H. We do not output slices of the phi and theta components at this point. For other fields (optics, ne, etc) we output a slice only in the r-theta plane (i.e. along the meridian).
4. In 3D you have to be careful about the elve viewing direction. By default, the camera is located “north” of the center of the grid, and views to the “south”. If you are putting in inhomogeneous ionosphere or atmosphere (i.e. gravity waves) or simulating horizontal sources, this orientation is important to keep in mind.
5. In 3D you can input gravity waves! These are perturbations to the background density with wavenumbers given in each direction. (You can also do gravity waves in 2D, but they aren’t very physical.)

7. Detailed Index of Input Parameters

Note that all inputs are in MKS units: meters, seconds, etc. When in doubt, assume MKS!

Note also that while I am familiar with the use of boolean values in C, I have chosen to implement flags as type int (I denote them as “int flag” below). My tests in the C code then assume anything that’s not zero is true. This is lazy programming; deal with it.

In the order they are listed in emp2Ddefaults.m:

Input	Type	Description
inputs.exefile = '…';	string	the name of the executable file to run
inputs.exedir = '…';	string	the location of the executable file. The shell script will copy the executable from this location to the run directory.
inputs.submitjob = 1;	int flag	set this to zero just to test the setup
inputs.savefields = [1 1 1 1 1 1];	int flag x 6	a vector six zeros or ones. Set each to zero if you don't need the large output fields. In order, the files are E, J, H (fields); K (); D (); and O.
inputs.Re = 6370000;	double	radius of the Earth (or planet) in meters.
inputs.dopml_top = 1;	int flag	use (1) or turn off (0) the PML on the upper boundary. If 0, the upper boundary becomes a perfect conductor.
inputs.dopml_wall = 1;	int flag	use (1) or turn off (0) the PML on the far right boundary. If 0, this boundary becomes a perfect conductor.
inputs.doionosphere = 1;	int flag	use (1) or turn off (0) the ionosphere. Useful for studying the ground wave of sferics, radiation patterns, etc when you don't need the ionospheric interaction. Saves times (doesn't solve the J equation) and guarantees stability.
inputs.doioniz = 1;	int flag	use (1) or turn off (0) the ionization calculations, including ionization, attachment, detachment, heating, and optical emissions. Useful for long-distance propagation, when you want ionospheric reflection but don't care about the nonlinear effects.
inputs.doelve = 1;	int flag	use (1) or skip (0) the elve photon propagation to the camera, resulting in elve.dat.
inputs.dodetach = 1;	int flag	use (1) or skip (0) detachment process in particular; this flag was created to compare the results with and without it. See Marshall [2012].
inputs.numfiles = 30;	int	number of output files; they will be evenly spaced through the simulation. More files = more hard drive space but better time resolution when you go to animate it.
inputs.maxalt = 110e3;	double	maximum simulation altitude
inputs.groundmethod = 0;	int	choice of ground method. 0 is perfectly conducting (PEC) ground; 1 is SIBC method; 2 is "real" ground, using permittivity and conductivity values from a global array of values. See Section 5 for details.
inputs.dr0 = 20;	double	altitude step size below the surface (in the ground); doesn't matter if you choose 0 or 1 above.
inputs.nground = 0;	int	number of vertical grid cells in the ground; this will be set to 0 for PEC or SIBC ground methods.

Input	Type	Description
inputs.dr1 = 200;	double	altitude step size above ground
inputs.dr2 = 200;	double	altitude step size above inputs.stepalt below; use to refine the grid in the ionosphere, where the wavelength becomes smaller
inputs.drange = 200;	double	horizontal step size
inputs.stepalt = 70e3;	double	altitude at which vertical step transitions from dr1 to dr2
inputs.Trlat = 38;	double	latitude of source (Tr = transmitter; I should rename this). This and the next three parameters are used to pull permittivity and conductivity values for the ground. They are NOT used to set the ionosphere, atmosphere, or magnetic field.
inputs.Trlon = -83;	double	longitude of source
inputs.range = 250e3;	double	total simulation distance in the lateral (along-ground) direction
inputs.az = 153;	double	azimuth from source, used to get ground parameters
inputs.dt = 1e-7;	double	time step
inputs.sig = 0;	double	conductivity of air (or other medium if you wish!)
inputs.sigm = 0;	double	magnetic conductivity of air (not physical)
inputs.camdist = 500e3;	double	distance to camera (for elves)
inputs.camalt = 0;	double	altitude of camera
inputs.camelev = 9;	double	elevation angle of camera (center of FOV) in degrees
inputs.camfov = [36 18];	double x 2	camera field of view (in degrees) in left-right and up-down directions
inputs.numpixels = [128 64];	int x 2	number of camera pixels in left-right and up-down directions
inputs.cameratype = 'camera';	string	camera type; this is no longer used, since I now generate an array of camera pixels
inputs.elvesteps = 1000;	int	number of time steps in the elve.dat array. Note that photons are output at every time step of the simulation, but get slotted into an appropriate time step for the elve based on the distance from the grid cell to the camera.
inputs.dotransmitter = 0;	int flag	flag to use a VLF transmitter source (1) or lightning (0). This changes the heating model that is implemented from BOLSIG+ (for lightning) to the Rodriguez heating model (for transmitters)
inputs.txf0 = 20e3;	double	VLF transmitter frequency.
inputs.planet = 0;	int	choice of planet: (0) Earth, (1) Venus, (2) Saturn. Saturn is not yet implemented. Note: you still need to change the planet radius in inputs.Re!

Input	Type	Description
inputs.decfactor = 2;	int	decimation of 2D field outputs. For fine grid resolutions (500 m or less), the grid is very large, so the output files can be large and cumbersome. This will decimate them by this factor in each dimension; so decfactor = 2 reduces the files by 4x in 2D. It is not implemented in 1D or 3D.
inputs.probedist = 100e3;	double	no longer used
probeangle		no longer used; was used for radiation patterns
inputs.probealt = [0] * 1e3;	double vec	vector of probe altitudes. even if they are all on the ground, you need a value here for each probe
inputs.proberange = [100] * 1e3;	double vec	vector of probe ranges (distance along the ground).
inputs.lightningtype = 1;	int	lightning type: 0 is CG, 1 is IC, and 2 is a bouncing CID (not yet implemented).
inputs.I0 = 100e3;	double	peak current in Amperes
inputs.Ic = 0;	double	continuing current in Amperes
inputs.sourcealt = 20e3	double	source maximum altitude for CG, or center altitude for IC
inputs.chlength = 2e3;	double	channel length for IC lightning; ignored for CG
inputs.taur = 7e-6;	double	source current rise time
inputs.tauf = 20e-6;	double	source current fall time
inputs.rsspeed = -0.9*vp;	double	source current return stroke speed. For CG, the sign is ignored; for IC, a positive speed is downwards and a negative speed is upwards
inputs.decaytype = 1;	int	lightning decay type: 0 = TL (no decay), 1 = MTLL, 2 = MTLE, 3 = Bruce-Golde, 4 = TCS, 5 = DU, and 6 = TL with infinite return stroke speed (the old method, circa Veronis et al [1999])
inputs.fcut = 300e3;	double	cutoff frequency for source; whatever you generate will be low-pass filtered with this cutoff in a 40th order FIR filter.
inputs.Bmag = 50000e-9;	double	magnitude of Earth's magnetic field in Tesla; note that it is vertical only.
inputs.cluster = 'batchnew';	string	name of computer cluster to launch jobs on
inputs.numnodes = '8';	string	number of nodes (processor cores) to use for each job
inputs.dogwave = 0;	int flag	flag to include (1) or exclude (0) a gravity wave in the atmosphere
inputs.gwavemag = 0.5;	double	gravity wave maximum amplitude

Input	Type	Description
inputs.gwavemaxalt = 100e3;	double	altitude where gravity wave reaches maximum; it increases exponentially up to this altitude, then is help constant above this altitude
inputs.gwavekh = 2*pi/ 20e3;	double	gravity wave horizontal wavenumber 2*pi/wavelength). In the 2D code there is no vertical wavelength

A few inputs particular to the 3D version:

- inputs.sourcedirection — allows for horizontal sources! Set this flag to 0 for vertical or 1 for horizontal (not yet implemented).
- inputs.Bmag — The magnetic field in 3D is a 3-element vector, as it is no longer restricted to be vertical. It is, however, homogeneous in the 3D space.
- inputs.IRI — This will multiply the ionosphere used in the simulation by a constant factor. I used this to vary the ionospheric density and compare the results of attenuation of wave propagating through the ionosphere.
- inputs.gwavekr — vertical wavenumber for a gravity wave
- inputs.gwavekh — wavenumber in the theta direction for a gravity wave
- inputs.gwavekp — wavenumber in the phi direction for a gravity wave.

Other inputs that are modified in emp2Drun.m or emp3Drun.m:

- inputs.tsteps — the number of time steps in the simulation. This is calculated based on the maximum distance, i.e. from the source to the top right corner in 2D. The resulting time is then multiplied by a fudge factor. You'll have to modify this factor in emp2Drun.m (etc) to change the simulation duration.
- ne, nd, ni, temperature — each of these are 1D (altitude) vectors, evaluated at the grid steps, and stored as input files (i.e. ne.dat) that are read by the main code. I have a variety of functions that I use such as IRIIonosphere1.m to grab a few simple ionospheres. ne is electron density, nd is neutral density, ni is ion density (equal to electron density), and temp is temperature, which is used in the VLF transmitter heating code. I get nd and temp from the MSIS model using MSISAtmosphere1.m.
  - Note that IRIIonosphere1.m and MSISAtmosphere1.m are mostly copied-and-pasted from the web models, located at http://omniweb.gsfc.nasa.gov/vitmo/iri_vitmo.html and http://omniweb.gsfc.nasa.gov/vitmo/msis_vitmo.html. It should be easy enough to create your own such profiles to replace the ones I use.
- rates — ionization, attachment, detachment, mobility, and optical excitation rates are stored in 2D look-up tables, as a function of altitude and E-field magnitude. The values are stored at the specific altitudes in the grid, and so interpolation occurs in the main C code across E-field.
- in.Prad (transmitter only) — calculates the radiated power of the transmitter. Used to figure out what current source to input to get the desired radiated power.

A few inputs need to be modified in source code (.cpp files) — sorry for hard coding these! In a future version I'll pull them out and make them input parameters.

- The SIBC model from Oh and Schutt-Aine [1995] uses the six-parameter version, and the coefficients are hard-coded in the C files (lines 233-234 in the 2D code). It would be easy to modify to use more coefficients, but it will require code changes and re-compiling.
- The code assumes a single positive ion (NO); it is defined by its mass in line 17.
- There are a few others that are rather esoteric; if questions come up I will address them here in an updated version.