EPOCH Workshop Overview

The aims of the Workshop are:

- After the two days you should be able to setup and run EPOCH on a problem of real importance to your research.
- You should also be in a position to use and understand the manual.
- You should learn about PIC codes in general.
- You should understand more about the pitfalls of trying to do LPI studies with PIC.
- Advice on how to run EPOCH and setup software on your home computers.
- Give advise to the EPOCH team on new features for the code.

Warwick EPOCH Personnel:

- *Tony Arber* – PI on EPOCH project at Warwick.
- *Keith Bennett* – PDRA and senior EPOCH developer.

Resources:

- All machines, and exercises, are *linux* based.
- EPOCH is a Fortan90 program which uses MPI for parallelization.
- You will always need both F90 and MPI to compile and run the code even on one processor.
- MPI on a Windows computer is not easy. Use *linux* or a Mac.

Workstation Usage

You can use the workstations for simple 1D tests and looking at the code.

Ultra-simple getting EPOCH guide!

These instructions should work in your host institute if you have *svn.*

1. Login to workstation using *guest* account.
2. Open a terminal.
3. Open a web browser and navigate to CCPForege at the URL http://ccpforge.cse.rl.ac.uk.
4. Login to your CCPForge account – register if you don't have one yet!
5. Find the EPOCH project under 'Projects'
6. Go to 'SVN' tab on left.
7. Click on 'Access info' and copy the *svn* command line into your terminal.
8. Add /trunk to the end so that you have something like. "svn checkout --username <*user*> http://ccpforge.cse.rl.ac.uk/svn/epoch/trunk Epoch". The final "Epoch" here is the name of the directory on your local machine.
9. Now hit return and wait for the project to download.

You will now have a directory called 'epoch'. Inside this directory will be three EPOCH sub-directories *epoch1d, epoch2d* and *epoch3d* and some MatLab and VisiT files. Change directory into the *epoch1d* directory and start working through the 'Getting Started with EPOCH' guide.

Running the codes

Single core job: > echo Data | mpiexec –n 1 bin/epoch1d

Four core parallel job: > echo Data | mpiexec –n 4 bin/epoch2d

Note: If you don't have *svn* on your home computer you can always download a tar file of *epoch* when you return to your lab. This you get from the 'Releases' tab on the EPOCH CCPForge webpage. However I recommend you get, and learn, *svn* and join the 21st century.

Getting Started with EPOCH

Compiling the code

The first thing you must do is to compile the code. This is done using the UNIX "make" command. This command reads a file called "Makefile" and uses the instructions in this file to generate all the steps required for compiling the code. Most of this is done automatically and the only part which typically needs changing are the instructions for which compiler to use and what compiler flags it accepts. The Makefiles supplied as part of the EPOCH source code contain sections for most commonly used compilers so it is usually unnecessary to actually edit these files. Usually you can compile just by passing the name of the compiler on the command line.

To compile the 1D version of the code, first change to the correct directory by typing "cd Epoch/epoch1d". The compiler used on most desktop machines is "gfortran", so you can compile the code by typing "make COMPILER=gfortran". Alternatively, if you type "make COMPILER=gfortran –j4" then the code will be compiled in parallel using 4 processors. If you wish, you can save yourself a bit of typing by editing the Makefile and adding the line "COMPILER=gfortran" at the top of the file. Then the command would just be "make –j4".

The most commonly used compiler on clusters these days is the Intel FORTRAN compiler. This is the default one used in the Makefile so for such machines you need only type "make" without editing the file.

You should rarely need to edit the Makefile more than this. Occasionally, you may need to change fundamental behavior of the code by changing the list of flags in the "DEFINES" entry. This is documented in the User manual.

Running the code

In your home directory there is a subdirectory called "EXAMPLES". This contains the example decks we will be working through.
When EPOCH is run it asks for the name of a directory to use. It will look in this directory for a file with the name "input.deck" containing the

problem setup. Any output performed by the code will also be written into this directory.
To work through the examples, you must copy an input deck from "~/EXAMPLES" to the directory you want EPOCH to use and rename the file "input.deck". Throughout this guide we will assume that you use the directory named "Data".

A Basic EM-Field Simulation

Our first example problem will be a simple 1D domain with a laser. This should give you a simple introduction to the input deck and visualization of 1D datasets.

Begin by copying the "01-1d_laser.deck" file from the EXAMPLES directory into the "Data" directory using the command:

cp ~/EXAMPLES/01-1d_laser.deck Data/input.deck

Open the input deck with an editor to view its contents.
Eg. "gedit Data/input.deck"

This is the simplest possible input deck. The file is divided into blocks which are surrounded by "begin:*blocktype*" and "end:*blocktype*" lines. There are currently ten different blocktypes. The most basic input deck requires only two.

The first block is the "control" block. This is used for specifying the domain size and resolution and the length of time to run the simulation. There are also some global simulation parameters that can be specified in this block which will be introduced later.
Within the block, each parameter is specified as a "*name = value" pair.*

The parameters are as follows. "nx" specifies the number of grid points in the x-direction (since this is a 1D code, the grid is only defined in the x-direction). "x_min" and "x_max" give the minimum and maximum grid locations measured in meters. Since most plasma simulations are measured in microns, there is a "micron" multiplication factor for convenience. There are also multiplication factors for "milli" through to "atto". Finally, the simulation time is specified using "t_end" measured in seconds.

There are also commented lines in the deck. Any text following the "#" character is ignored. The character may appear anywhere on a line, so in the following example:
t_end = 50 #* femto

The value of “t_end” will be set to 50 seconds, since “#* femto” is ignored.

The other required block is the “boundaries” block. This contains one entry for each boundary, specifying what boundary condition to apply. For the 1D code there are two boundaries: “bc_x_min” and “bc_x_max”. The deck currently has both of these set to use open boundary conditions.

To run the code type:
echo Data | mpiexec –n 4 ./bin/epoch1d

This will run epoch1d in parallel using 4 processors. It will use the directory named “Data” for all its output and will read the file “Data/input.deck” to obtain the simulation setup.

This simulation is rather dull. It is just a grid with zero electromagnetic field and it generates no data files. After running the program, two files are generated in the “Data” directory. The “deck.status” file contains the results from the deck parsing routines and is only useful for debugging. The “epoch1d.dat” file contains a terse one line header with the code name, version information and time the job started followed by a list of output dumps generated during the run.

Status information about the running job can be requested by uncommenting the “stdout_frequency” line in the “control” block. This is achieved by using a text editor to remove the “#” character and saving the file.

Adding a Laser

We will now edit this input deck to add a laser source to the left hand boundary and dump some output files.

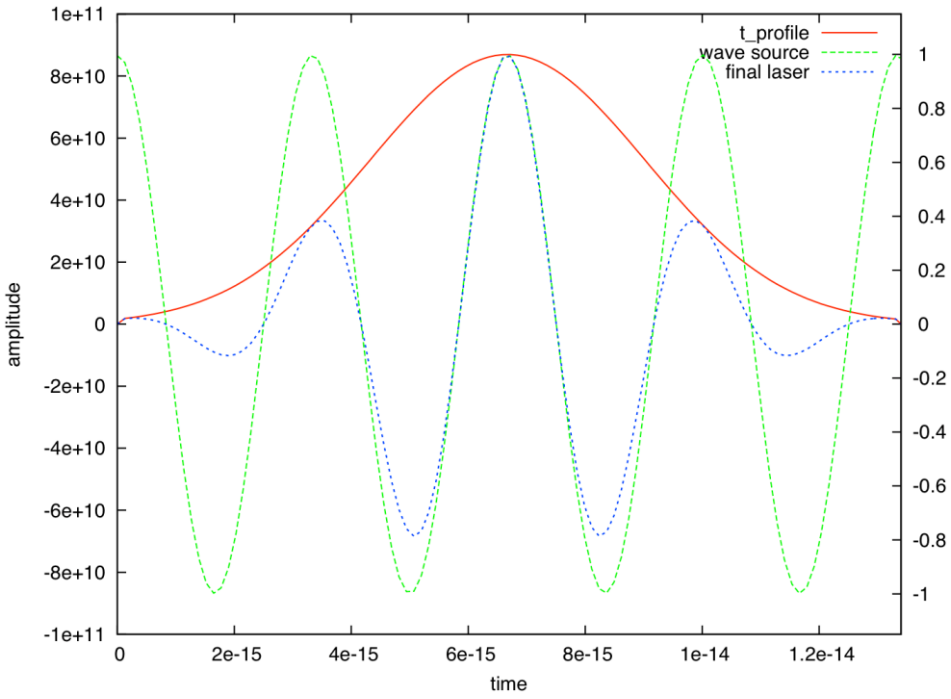
1. Open the "Data/input.deck" file with an editor.
2. Add a "#" comment character to the beginning of the first "bc_x_min" line in the "boundaries" block.
3. Uncomment the line "bc_x_min = simple_laser"
4. Uncomment the remaining lines in the file.

The change to the "boundaries" block instructs the code to add a laser source to the left-hand boundary.

The Laser Block

We then require a new block, named "laser", to set up the laser source. The parameters in this block do the following:

- boundary – Specifies the boundary on which to attach this laser source
- intensity_w_cm2 – Specifies the intensity of the laser in Watts / cm^2
- lambda – Gives the wavelength of the laser in meters. We have used the multiplication factor "micron" for readability
- phase – Specifies the phase shift of the laser.
- t_profile – This parameter is used to modify the amplitude of the laser over time. It is usually used to ramp a laser up or down gradually. The left-hand side will be a function of time, usually ranging between zero and one.
- t_end – The time at which to switch off the laser.



These parameters are mostly self-explanatory. The "t_profile" parameter is best explained using an example. The figure above shows the result of using a guassian time profile. The red line shows the value of "t_profile" over time. This starts at a value close to zero, ramps up to one and then ramps back down to zero. The green line shows the amplitude of the laser when "t_profile" has not been specified. Note that the function would normally be a sine wave, but this has been shifted by pi/2 because the "phase" parameter was used. The blue line shows the laser amplitude generated when the "t_profile" guassian profile is applied.

The Output Block

The final addition is the "output" block. We will cover this in more detail later. For now, it is sufficient to know that this is the block which controls the generation of data output. The parameters used in this case are:

- dt_snapshot – This specifies the simulation time between each output dump
- grid – This controls when to dump the simulation grid. The value of "always" means that the grid will be output whenever there is a new output dump generated.
- ey – The controls when to dump the y-component of the electric field.

Visualising the data

Now that we have generated some data we need to plot it. The data is written to a self-describing file format called SDF. This has been developed for use by several codes maintained at the University of Warwick. There are routines for reading the data from within IDL, VisIt, MatLab and Python.

Loading the data into IDL/GDL

First, we will load the data into IDL/GDL. The desktop machines have GDL installed – the GNU Data Language, which is a free implementation of IDL. It doesn't have all the feature of IDL but the core routines and syntax are identical.

Type "gdl Start.pro" and GDL will start up and load the SDF reading library.

To view the data contained in a file, type

- data = getstruct(7,'Data',/variables)

Here, "7" is the snapshot number. It can be any number between 0 and 9999. The second parameter specifies the directory which holds the data files. If it is omitted then the directory named "Data" is used by default. "/variables" instructs the reader to print a summary of the file contents to screen.

To load the data and assign the result to a structure named "data", just issue the following command:

- data = getstruct(7,/varname)

Here, "/varname" is any of the variables listed by the previous command. This will just read the "varname" variable into the data structure. However, it is usually easiest just to omit the "/varname" flag. If it is omitted then the entire contents of the file is read.

The "getstruct" command returns a hierarchical data structure. The contents of this structure can be viewed with the following command:

- help,data,/struct

For the current example the result of this command is the following:

```
GDL> help,data,/struct
```

```
** Structure <2a0ddd8>, 6 tags, length=5096, data length=5080,
refs=1:
   FILENAME        STRING    'Data/0007.sdf'
   TIMESTEP        LONG                 185
   TIME            DOUBLE       2.3386179e-14
   EY              STRUCT    -> <Anonymous> Array[1]
   GRID            STRUCT    -> <Anonymous> Array[1]
   X               DOUBLE    Array[200]
```

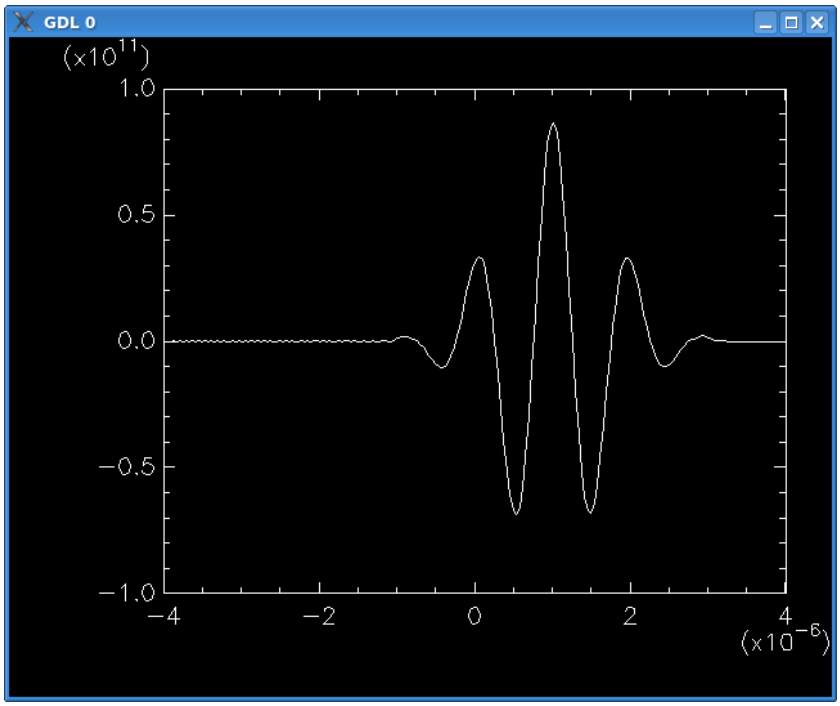
The first few entries are fairly self-explanatory. The sixth item is a 1D array containing the cell-centred grid positions. The fourth item is a structure containing a 1D array of Ey at these positions. This structure can be queried in the same way as “data” :

```
GDL> help,data.ey,/struct
** Structure <2a051a8>, 2 tags, length=1696, data length=1696,
refs=2:
   METADATA        STRUCT    -> <Anonymous> Array[1]
   DATA            DOUBLE    Array[200]
```

The raw data is contained in the “data” entry. The fifth entry, “GRID” is a structure which contains :

```
GDL> help,data.grid,/struct
** Structure <2a0d7a8>, 5 tags, length=1768, data length=1756,
refs=2:
   METADATA        STRUCT    -> <Anonymous> Array[1]
   X               DOUBLE    Array[201]
   LABELS          STRING    Array[1]
   UNITS           STRING    Array[1]
   NPTS            LONG      Array[1]
```

This is the node-centred grid along with its metadata. The cell-centred array shown previously is derived from this.



The above plot can be generated by issuing the following command:

- plot,data.x,data.ey.data

There are more examples on using idl/gdl in the manual.

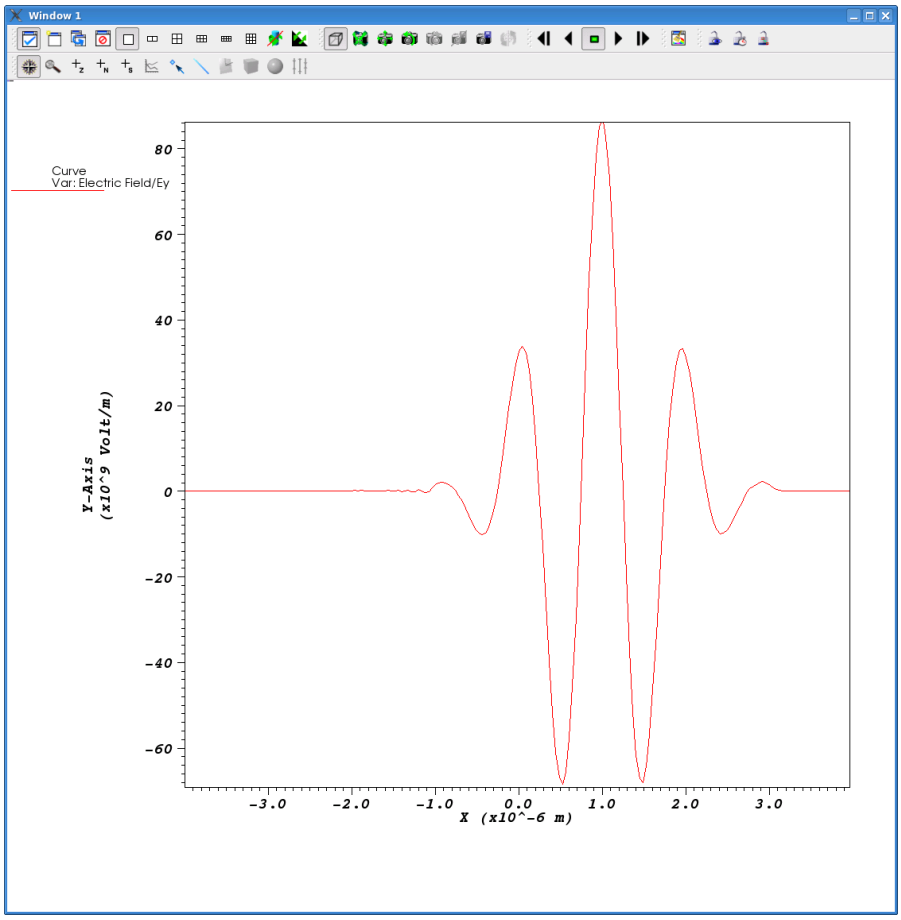
Loading the data into VisIt

EPOCH comes with an SDF reader plugin for the VisIt parallel visualization tool. In order to use it, you must first compile the reader to match the version of VisIt installed on your system. To do this, first ensure that the "visit" command is in your path. This is the case if typing "visit" on the command line launches the VisIt application. Once you have this setup, you should be able to type "make visit" from one of the epoch{1,2,3}d directories. You will need to re-do this each time a new version of VisIt is installed on your system.

Launch the VisIt application by typing "visit" on the command line. A useful shortcut is to type "visit –o Data/0000.sdf". This will launch VisIt and open the specified data file on startup. Alternatively, you can browse for the file to open using the "Open" button. All the SDF files in a directory will be grouped together with a green "DB" icon and the name "*.sdf database".

You can then plot a quantity by pressing the "Add" button, selecting the type of plot and the variable to use for the plot. When the plot has been selected, press the "Draw" button to render it to screen. The following plot was generated by selecting "Add->Curve->Electric Field->Ey". Some of the plot properties were adjusted to make it look nicer.

More details on using VisIt are in the Manual. We recommend that you learn VisIt – it's free and powerful.



Loading data into MatLab

The EPOCH distribution also comes with a set of reader routines for the MatLab plotting utility. The routines themselves are contained in the "Epoch/Matlab" directory. It is first necessary to add this directory to your search path. One simple way of doing this is to use the menu item "File->Set Path" and then "Add Folder" to select the location of the "Matlab" folder. To make this change permanent you have to use the "Save" button. Unfortunately, on many systems this will not work as it tries to change global settings which will not be permitted on a multi-user

setup. On Unix systems (including OS X), the change can be made permanent by using the "$MATLABPATH" environment variable. For example in bash this would be 'export MATLABPATH="Epoch/Matlab" ' which you can add to your .bashrc file.

To load the data from an SDF file, type the following at the MatLab prompt:

- data=GetDataSDF('Data/0007.sdf');

The "data" variable will now contain a data structure similar to that obtained with the IDL reader. You can explore the contents of the structure using MatLab's built-in variable editor. To plot Ey, you can browse to "data.Electric_Field.Ey". The structure member "data.Electric_Field.Ey.data" contains the 1D array with Ey values. Right-clicking on it gives a range of options, including "plot".
Alternatively, from the command prompt you can type

```
>> x=data.Electric_Field.Ey.grid.x;
>> xc=(x(1:end-1) + x(2:end))/2;
>> plot(xc,data.Electric_Field.Ey.data);
```

The first two lines set up a cell-centred grid using the node-centred grid data. In the future, this work will be automatically done by the reader.

A 2D Laser

Next, we will take a look at the 2-dimensional version of the code.

- Change to the epoch2d directory: "cd ~/Epoch/epoch2d"
- Type "make –j4" to compile the code.
- Copy the next example input deck into the Data directory:
  "cp ~/EXAMPLES/02-2d_laser.deck Data/input.deck"
- Run with "echo Data | mpirun –np 4 ./bin/epoch2d"

This deck is very similar to the 1D version that we have just looked at. It contains the necessary modifications for adding a new dimension and some additions to the laser block for driving a laser at an angle.

The "control" block now contains "ny" which specifies the number of grid points in the y-direction. Notice that we are using the value "nx" to set "ny". As soon as "nx" has been assigned it becomes available as a constant for use as part of a value.
We must also provide the minimum and maximum grid positions in the y-direction using "y_min", "y_max". Like "nx", the values "x_min" and "x_max" are available for use once they have been assigned.

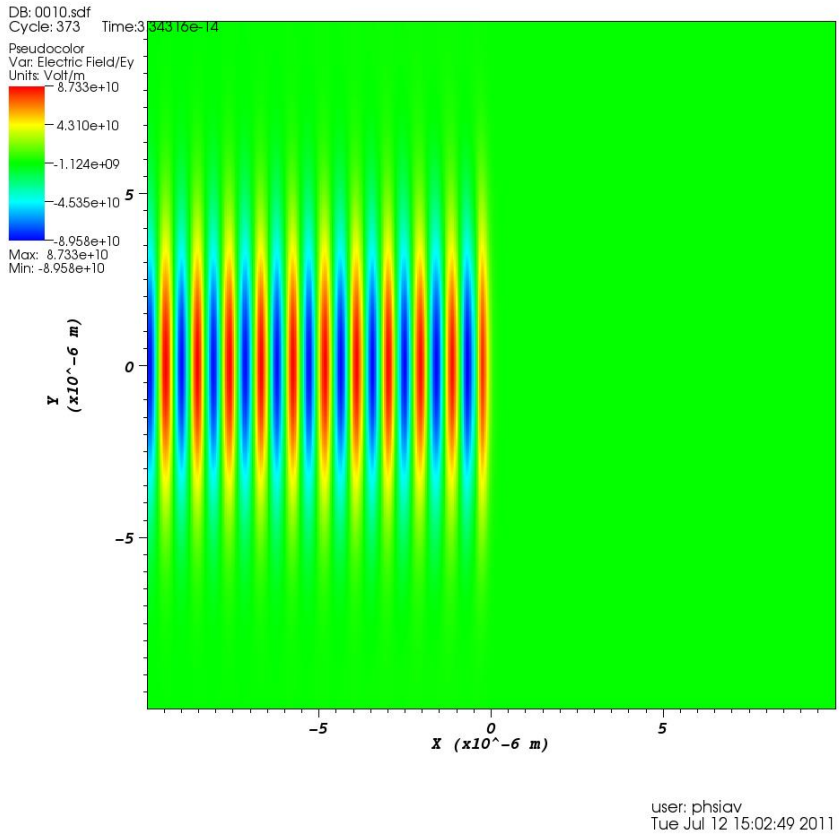
In the "boundaries" block we must include boundary conditions for the lower and upper boundaries in the y-direction, "bc_y_min", "bc_y_max". These have both been set to "periodic" so that the field at the top of the domain wraps around to the bottom of the domain.

Next, we introduce a new block type, "constant". This block defines named variables which can be arbitrary mathematical expressions. Once defined, these can be used on the left-hand side of name-value pairs in the same way we used "nx", "x_min", etc. in the "control" block. This facility can greatly aid the construction and maintenance of complex input decks.

The "laser" block is similar to that given in the 1D version except that there is now a "profile" parameter. In a similar manner to "t_profile" this is a function ranging between 0 and 1 which is multiplied by the wave amplitude to give a modified laser profile. The only difference is that this is a function of space rather than time. When applied to a laser attached

to “x_min” or “x_max” it is a function of Y, defined at all points along the boundary. When the laser is attached to “y_min” or “y_max”, it is a function of X.

Finally, the output block has been modified so that it outputs all electromagnetic field components.



The result of plotting “Add->Pseudocolor->Electric Field->Ey” in VisIt is shown above.

The laser block also contains a commented-out “phase” entry. Unlike in the 1D version seen previously, this is a function of Y, like the “profile” parameter. Uncommenting this line and re-running the deck will generate a laser driven at an angle to the boundary. The mathematical details explaining why this works are explained in more detail in the User Manual. By making the value of “theta” a function of Y, it is also possible to produce a focused laser. This is left as an exercise for the reader!

Specifying Particle Species

In this example we will finally introduces some particles into the PIC code! The deck is for the 1D version of the code, so change back to the epoch1d directory and copy ~/EXAMPLES/03-1d_two_stream.deck to Data/input.deck and run the code.

The control block has one new parameter. “npart” gives the total number of PIC particles to use in the simulation.

The input deck contains a new block type, “species”, which is used for populating the domain with particles. Every species block must contain a “name” parameter. This is used to identify the particle species in other sections of the input deck and is also used for naming variables in the output dumps. The next parameter is “charge” which gives the charge on each particle in terms of elementary charge units. “mass” is specified in units of electron mass. “frac” is the fraction of the total number of PIC particles (npart) to assign to this species. Both of the blocks in this deck use “frac = 0.5”, so there will be 1600 particles of each species. The next parameter, “temp”, sets the average temperature of the particle species in Kelvin. Alternatively, you can use “temp_ev” to specify the temperature in electronvolts. Particles are assigned an initial momentum corresponding to a Maxwell-Boltzmann distribution for this temperature. It is defined across the entire problem domain, so in 1D it is a function of X, in 2D a function of X and Y, and in 3D a function of X, Y and Z. “density” sets the number density across the problem domain. The code is set to use per-particle weights in the default Makefile. With this option, the pseudoparticles are distributed evenly across the domain. Then the weight of each pseudoparticle is adjusted so that it matches the number density specified in the “density” parameter. The alternative option is to disable per-particle weighting. In this case, the weight of each pseudoparticle is the same and the particles are placed on the grid so that they match the number density at the start of the simulation.
Finally, we have a “drift_x” parameter. This is also defined across the entire problem domain and is used to give the particles an average momentum drift in the x-direction. There are similar “drift_y” and “drift_z” parameters.

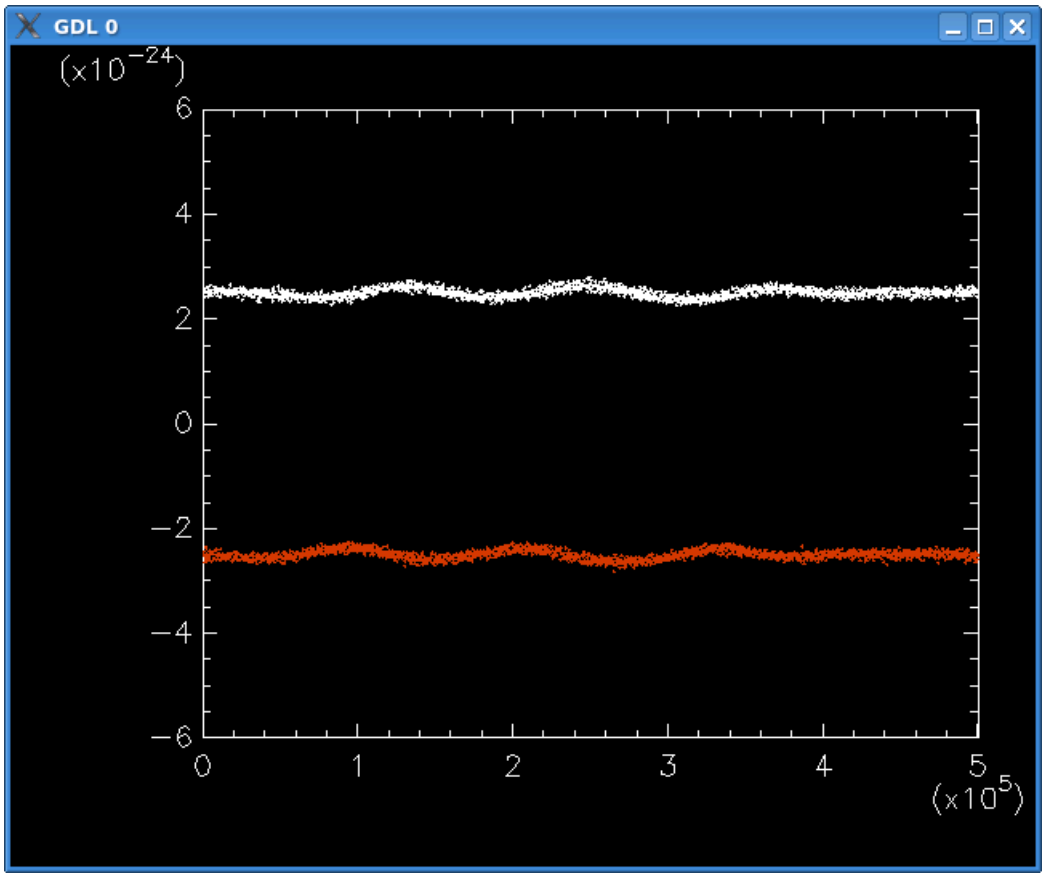
This deck has been designed to simulate a two-stream instability, so it has to groups of particles which are identical in every respect except that one set is drifting in the opposite direction to the other.

In the output block we have added a couple of parameters for outputting particle data. The first parameter, "particles", outputs the grid on which the particles are defined. There are two different types of variable in EPOCH: particle variables and grid-based variables. The grid-based variables are like the electromagnetic field components we have seen previously. The domain is divided into a regular Cartesian mesh and the grid-based variables are defined at either a node or cell-centre of each point in this mesh. Particle variables, on the other hand, are associated with each of the pseudoparticles. These PIC particles move independently of the Cartesian mesh and can be located anywhere in the problem domain. The "particles" parameter requests that the coordinates of each particle are written to file. This information is required in order to plot any of the particle variables. The next parameter is "px" which writes the momentum of each particle.

To plot this with GDL, type the following:

```
gdl Start.pro
GDL> data=getstruct(30)
GDL> plot,data.grid_right.x,data.px_right,psym=3,$
  yrange=[-6e-24,6e-24]
GDL> oplot,data.grid_left.x,data.px_left,psym=3,color=150
```

This will give a plot such as the following:



Here we have plotted the x-component of particle momentum as a function of x-position at a time when the instability is just starting to form. The “psym=3” option to the plot routine tells GDL to plot each data point as a dot and not to join the dots up.

The Output Block

Tony Arber

The contents of the output block can be much more complicated than the examples shown so far. Here, we will cover the options in a little more depth.

EPOCH currently has three different types of output dump. So far, we have only been using the "normal" dump type. The next type of dump is the "full" dump. To request this type of dump, you add the parameter "full_dumps_every" which is set to an integer. If this was set equal to "10" then after every 9 dump files written, the 10^{th} dump would be a "full" dump. This hierarchy exists so that some variables can be written at frequent intervals whilst large variables such as particle data are written only occasionally.
The third dump type is the "restart" dump. This contains all the variables required in order to restart a simulation, which includes all the field variables along with particle positions, weights and momentum components. In a similar manner to full dumps, the output frequency is specified using the "restart_dumps_every" parameter.

So far, we have given all the variable parameters a value of "always" so that they will always be dumped to file. There are three other values which can be used to specify when a dump will occur. "never" indicates that a variable should never be dumped to file. This is the default used for all output variables which are not specified in the output block. The value of "full" indicates that a variable should be written at full dumps. "restart" means it is written into restart dumps.

There are a few output variables which are grid-based quantities derived by summing over properties for all the particles contained within each cell on the mesh. These are "ekbar", "mass_density", "charge_density", "number_density" and "temperature". To find more details about these variables, consult the output block section of the User manual.

Other Laser-Plasma Example Decks

Now that you have a basic understanding of how the input decks work, you should be able to work through the remaining example decks by referring to the User manual for a description of any new parameters used. Several experienced users of the code will be available throughout the duration of the workshop, so if you want help with anything please don't hesitate to ask.
The decks are:

01-1d_laser.deck Described in notes above
02-2d_laser.deck Described in notes above
03-1d_two_stream.deck Described in notes above
04-1d_two_stream_io.deck This is the same as the previous deck but with the addition of more sophisticated output diagnostics
05-2d_moving_window.deck This deck contains an example of firing a laser into a plasma and then using the moving window facility to track the wave front as it moves beyond the edge of the original domain.
06-2d_ramp.deck This deck contains an example of firing a laser at a plasma with a ramped density profile.
07-1d_heating.deck This deck contains a setup for investigating the anomalous heating of a plasma that occurs for purely resolved systems.

Remote Visualisation with *VisIt*

If the local workstation you are using isn't big enough for your test problems you may also use a your host institutes HPC cluster.

Remote Visualisation with VisIt

Most large simulations are carried out on a remotely located machine. Often this machine is located many miles away, perhaps even in a different country. Viewing data on remote systems can be awkward and poor network speeds can often make it nearly impossible.
The VisIt visualisation tool solves this problem by using a client-server model. The program which reads, processes and renders the data is completely separated from the program which displays the results on the screen. It is therefore possible to run VisIt on your local machine and look at data located on a different machine. The method of setting this up varies depending on the configuration of the remote machine so we will not go into details here. However, the desktop machines have been setup to be able to view data located on remote clusters so you can try it out.

In the VisIt control window, click the "Open" button which launches a file browser window. The first entry is called "Host" and contains a drop-down list of all configure remote machines.

If you want to know more about how to set up remote visualisation in VisIt, you can ask one of the Warwick staff members.

When viewing data across a slow network connection, there is one more useful thing to know. VisIt has two methods of drawing plots generated on a remote machine. The first method is to construct the polygons used in drawing the plot on the remote machine and send them across the network. The local machine then turns these into a plot image. This makes manipulating the figure very fast (zooming, rotating, etc), since all the polygons that generate the image are on the local machine. However, if there are a lot of polygons then they can be slow to transfer across the network. They can also use up a lot of memory. For these cases, the alternative is to render the image on the remote machine and just transfer the image across the network. The downside of this approach is

that whenever you manipulate the plot, it must be re-drawn on the remote machine and then transferred across the network again. The options controlling this behaviour are to be found under “Options->Rendering” in the “Advanced” tab. The feature is called “scalable rendering”.

Collisions in *EPOCH*

EPOCH now contains a collision routine based on the technique outlined in Sentoku & Kemp [J Comput Phys, 229, 4591 (2010)]

Collisions are enabled using the output block named "collisions" which accepts the following three parameters.

use_collisions - This is a logical flag which determines whether or not to call the collision routine.
If omitted, the default is "true" if any of the frequency factors are non-zero (see below) and "false" otherwise.

coulomb_log - This may either be set to a real value, specifying the Coulomb logarithm to use when scattering the particles or to the special value "auto". If "auto" is used then the routine will calculate a value based on the properties of the two particles being scattered.
If omitted, the default value is "auto".

collide - This sets up a symmetric square matrix of size nspecies*nspecies containing the collision frequency factors to use between particle species. The element (s1,s2) gives the frequency factor used when colliding species s1 with species s2.
If the factor is less than zero, no collisions are performed.
If it is equal to one, collisions are performed normally.
For any value between zero and one, the collisions are performed using a frequency multiplied by the given factor.
If "collide" has a value of "all" then all elements of the matrix are set to one. If it has a value of "none" then al elements are set to minus one.
If the syntax "species1 species2 <value>" is used, then the (species1,species2) element of the matrix is set to the factor "<value>". This may either be a real number, or the special value "on" or "off". The "collide" parameter may be used multiple times.
The default value is "all" (ie. all elements of the matrix

are set to one).

For example:

```
begin:collisions
  use_collisions = T
  coulomb_log = auto
  collide = all
  collide = spec1 spec2 off
  collide = spec2 spec3 0.1
end:collisions
```

With this block, collisions are turned on and the Coulomb logarithm is automatically calculated. All values of the frequency array are set to one except (spec1,spec2) is set to minus one (and also (spec2,spec1)) and (spec2,spec3) is set to 0.1

Ionisation in *EPOCH*

EPOCH includes field ionization which can be activated by defining “field_ionisation = T” in the control block along with ionisation energies and an electron for the ionising species in one of the species blocks. This is done via the species block in the "ionisation_energies" and "electron_species" parameter respectively. "ionisation_energies" should be given as a list in joules, and "electron_species" should be the name of the species to be used as the electron species. For example, ionising carbon species might appear in the input deck as:

```
begin:species
  charge=0.0
  mass=1837.2
  name=carbon
  ionisation_energies=(11.26*ev,24.38*ev,47.89*ev,64.49*ev,392.1*ev,
490.0*ev)
  electron_species=electron
  rho=den_gas
end:species

begin:species
  charge=-1.0
  mass=1.0
  name=electron
  rho=0.0
end:species
```

It is possible to define different electron species for each ionisation level, which is particularly useful in monitoring specific ionisation levels. If we wished to monitor the fourth ionisation level of carbon in the above example, the above example might appear:

```
begin:species
  charge=0.0
  mass=1837.2
  name=carbon
```

```
  ionisation_energies=(11.26*ev,24.38*ev,47.89*ev,64.49*ev,392.1*ev,
490.0*ev)
  electron_species=(electron,electron,electron,fourth,electron,electron)
  rho=den_gas
end:species

begin:species
  charge=-1.0
  mass=1.0
  name=electron
  rho=0.0
end:species

begin:species
  charge=-1.0
  mass=1.0
  name=fourth
  rho=0.0
end:species
```

Field ionisation consists of three distinct regimes; multiphoton in which ionisation is best described as absorption of multiple photons, tunneling in which deformation of the atomic coulomb potential is the dominant factor, and barrier suppression ionisation in which the electric field is strong enough for an electron to escape classically. It is possible to turn off multiphoton or barrier suppression ionisation through the input deck by adding "use_multiphoton=F" and/or "use_bsi=F" to the control block.

QED Effects in *EPOCH*

EPOCH has recently been extended to include some quantum electrodynamic effects that are important for high intensity (>5 x 10^{21} W/cm^2) lasers. The two processes that are included are

- Gamma ray production by QED corrected synchrotron emission (Also called magnetic bremsstrahlung or nonlinear Compton scattering).
- Electron positron pair production by the Breit-Wheeler process from these gamma ray photons.

For more information on the theory see Duclous et al. PPCF **53** (2011) 015009.

Simulating the QED effects increases EPOCH's memory requirements and so the code has to be compiled with the correct compilation options to turn the module on. To turn the module on, open "Makefile" in an editor and find the line which begins "DEFINES = ". Make sure that somewhere on the right hand side of the = line appears the text "-DPHOTONS". If it isn't there then add it. Then type "make clean" and then "make" to rebuild the code with QED support.

Once the code is built with QED support, actually turning on QED for a specific simulation requires the addition of a new block into the input deck. This block is simply called "qed" and starts with the usual "begin:qed" and "end:qed" markers of the other blocks. The parameters which can go into the block are

- use_qed - Turns QED on or off. If you don't want QED effects at all then compile the code without the "-DPHOTONS" lines in the makefile.
- qed_start_time - Specifies the time after which QED effects should be turned on. For example you can turn off the routines until a laser has crossed the vacuum region in front of the target.
- produce_photons - Specifies whether you're interested in the photons generated by synchrotron emission. If this is F then the radiation reaction force is calculated but the properties of the emitted photons are not tracked.

- photon_energy_min - Minimum energy of produced photons. Radiation reaction is calculated for photons of all energies, but photons with energy below this cutoff are not tracked.
- photon_dynamics - If F then photons are generated, but their motion through the domain is not simulated and they stay where they were generated. Photon motion is often less interesting than photon generation unless you want to simulate pair production. In these cases set this to F.
- produce_pairs - Whether or not to simulate the process of pair generation from gamma ray photons. Both produce_photons and photon_dynamics must be T for this to work.
- qed_table_location - EPOCH's QED routines use lookup tables to calculate gamma ray emission and pair production. If you want to use tables in a different location from the default put the location in this parameter.

QED also requires that the code now know which species are electrons, positrons and photons. Rather than try to do this automatically the user has to specify the type of a species. This is done by using a single "identify" tag in a species block. To specify an electron the block in the deck would look like

```
begin:species
  name=electron
  frac=0.5
  rho=7.7e29
  identify:electron
end:species
```

Once the identity of a species is set then the code automatically assigns mass and charge states for the species. At present, the user cannot override these. Possible identities are

- electron : A normal electron species. All species of electrons in the simulation must be identified in this way or they will not generate photons.

- positron : A normal positron species. All species of positron in the simulation must be identified in this way or they will not generate photons.
- photon : A normal photon species. One species of this type is needed for photon production to work. If multiple species are present then generated photons will appear in the first species of this type.
- bw_electron : The electron species for pair production. If a species of this type exists then electrons from the pair production module will be created in this species. If no species of this type is specified then pair electrons will be generated in the first electron species.
- bw_positron : The positron species for pair production. If a species of this type exists then positrons from the pair production module will be created in this species. If no species of this type is specified then pair positrons will be generated in the first positron species.

A species should be identified only once, so a "bw_electron" species does not need to also be identified as an "electron" species. If the code is running with "produce_photons=T" then a photon species must be created by user and identified. If the code is running with "produce_pairs=T" then the code must specify at least one electron (or bw_electron) species and one positron (or bw_positron) species. The code will fail to run if the needed species are not specified.

Other Useful Info

Bug reports, feature requests and questions

All questions and requests after the workshop should be posted on the CCPForge EPOCH project web page.

The VisIt programme

The VisIt programme is free. It can be downloaded from https://wci.llnl.gov/codes/visit/

There are many pre-compiled binaries so this ought to be easy. If you have any problems post a question on the CCPForge EPOCH project.

GDL not IDL

If you don't have IDL, or don't want to pay for it!, then the free GDL is available from http://gnudatalanguage.sourceforge.net/

Updating EPOCH

To update to the latest version of EPOCH simple cd into your Epoch directory and enter 'svn update'. This will work fine provided you haven't edited any of the Fortran source code. If you have edited the source code then you need to learn svn.

Getting Old Copies of EPOCH

You can also checkout an old version of EPOCH, you may want to get the version used 18 months ago to reproduce some previous simulations exactly for example. In this case it is best to open a new copy of EPOCH. If you wanted the version from 10 February 2010 for example you would enter "svn checkout –r {2010-02-10} --username <*user*> http://ccpforge.cse.rl.ac.uk/svn/epoch/trunk Epoch-Old". Where the -r flag is used to specify the date and the whole thing is copied to Epoch-Old