PICsar2D Tutorial

1. Introduction

This tutorial is intended to provide an introduction to *PICsar2D*. It is targeted at anyone who would like to use the code for scientific purposes. Information about the algorithms implemented in the code, equations solved, and the relevance to astrophysical pulsars can be found in Belyaev (2015a) (paper 1), Belyaev (2015b) (paper 2), and Belyaev (2017) (paper 3). Certain subroutines in *PICsar2D* (e.g. the Villasenor-Buneman current deposition) were borrowed from the 3D Cartesian PIC code *TRISTAN* (Buneman 1993).

2. Setting Up the Code

After downloading the code, decompress the source files via “$ tar -xzvf PICsar2D_head.tar.gz” (dollar sign is used to indicate command line input). In the “PICsar2D_head” folder, you will find two sample Makefiles, one of which is appropriate for compiling with the GNU Fortran compiler, and the other for compiling with the Intel Fortran compiler. The Intel compiler should generate faster code than the GNU compiler because *PICsar2D* has been written to take advantage of SIMD vectorization available on Intel processors. The Intel Makefile contains compiler settings for different processor families, and using the appropriate compiler settings for a given processor family will generate the fastest executable. Depending on whether you would like to use the Intel or the GNU compiler, copy either “Makefile_ifort” or “Makefile_gfortran” to a file called “Makefile”, and execute “$ make” on the command line to compile the code.

3. Getting Started with the Code

3.1. Monopole Pulsar Test Problem

The executable “picsar2D” is set up to run a pulsar problem with a monopole magnetic field on a single core. It is good to start with the monopole test problem, because it has an analytic solution that can be used to check the simulation result (see §5 of paper 1 for more details). Run the executable on the command line via “$ ./picsar2D”. The command line output shows the timestep followed by the number of positive and negative charges in the simulation. The simulation will run for 2000 timesteps and takes a few minutes to complete, so you may want to pour yourself a cup of tea in the meantime.

Now that your tea is brewed, it’s time to start analyzing the data, which is stored in the “output” directory. Go to the “analysis” directory and execute “$ python bphi_br_plot.py”, which generates plots of the ratio between the azimuthal and radial magnetic fields, B_ϕ/B_r. The plots

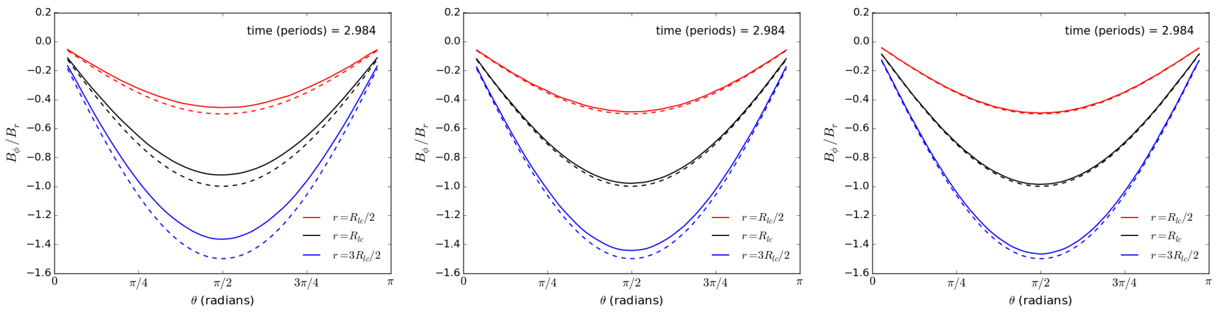

Fig. 1.— B_ϕ/B_r for the monopole test problem for “surf_inj_multi = .1”, “scfc = 1” (left panel); “surf_inj_multi = .3”, “scfc = 1” (middle panel); and “surf_inj_multi = .3”, “scfc = 2” (right panel). The dashed curves correspond to the analytical force-free solution, and the solid curves are the simulated output.

are at three different cylindrical radii, $R = R_{\rm lc}/2$, $R = R_{\rm lc}$, and $R = 3R_{\rm lc}/2$, where $R_{\rm lc}$ is the light cylinder radius. The left panel of Fig. 1 shows what you should see for the last output. The analytical solutions are the dashed lines, and the simulated solutions are the solid lines.

At the start of the simulation, B_ϕ is zero, $B_r = B_*(r_*/r)^{-2}$, and there is vacuum outside the star. When the simulation is started, charged particles are injected into the domain at the surface of the star at each timestep. As the magnetosphere fills with plasma, it approaches a force-free state. The force-free solution for the aligned monopole rotator has $B_\phi = -(R/r_*)B_r$, where R is the cylindrical radius. However, the simulation curves are shallower than the analytical curves, implying there is less current in the simulated PIC solution than in the force-free solution.

To improve the correspondence between the simulation and the analytical solution, we can increase the fraction of surface charge injected at each timestep. Go to the “source/input” directory, open the parameter file “monopole_pulsar_parameters.f”, and set “surf_inj_multi = .3”. The variable “surf_inj_multi” determines the fraction of surface charge injected per timestep, and increasing the injection rate leads to a simulation that is closer to force free. However, increasing it too much leads to overinjection and makes the injection algorithm unstable. In this case, the surface charge on the star oscillates wildly, rather than being damped to zero, and the simulated solution is much noisier.

Execute “$ make clean” followed by “$ make” to clean up and compile the source code with the increased value for the surface charge injection rate. Run the simulation again by executing “$ ./picsar2D”. The simulated solution is now closer to the analytical force-free solution (middle panel of Fig. 1), which can be seen by plotting B_ϕ/B_r in the same manner as before.

3.2. Running on Multiple Processors with MPI

The benefit of using multiple cores is that larger problem sizes can be run. To change the grid size, open "monopole_pulsar_parameters.f" and set "scfc = 2". Here, "scfc" is a scale factor for the problem, which can be used to increase the resolution in both directions by an integer multiple of the base resolution. Notice that the dimensions of the grid, N_r and N_θ, are odd. This is because N_r and N_θ correspond to the number of grid edges in each dimension, which is one more than the number of cells. To change the number of MPI domains in r and θ to two in each dimension, set "NPROC_R = 2" and "NPROC_T = 2" in "monopole_pulsar_parameters.f". If you have more cores available to you, you can set "NPROC_R" and "NPROC_T" accordingly to use the available cores and reduce the computing time.

To compile with MPI, you must include "-Dmpi" on the definitions line of the Makefile, remove the comment before "source/mpi_related_routines.f", and change the compiler to "mpif90". Recompile the code and execute "$ mpirun -np 4 picsar2D" (change the 4 to NPROC_R * NPROC_T if using a different number of processors). Because of the higher resolution, the simulation will run for twice as many timesteps and may take several times longer than before.

The outputs from individual processors are saved to the "MPI_output" folder and need to be joined together before they can be analyzed. First, however, you will need to specify the number of processors by setting "nproc_r=2" and "nproc_t=2" in "analysis/sim_pars.py" (in the else clause of the if statement). Also set "scfc = 2" in "sim_pars.py" to inform "merge_routines.py" about the new grid dimensions.

Join the files by executing "$ python merge_routines.py" in the "analysis" directory. The merged files are saved to the "output" directory and can be analyzed in the same way as before. Plotting B_ϕ/B_r again, we see that the simulation is closer to the analytical solution than before (right panel of Fig. 1), even though the value of "surf_inj_multi" has stayed constant. This is in part due to the fact that there are more timesteps per pulsar rotation period due to the Courant condition, so the surface charge is released twice as often.

4. Running a Dipole Pulsar Problem

To generate the executable for the dipole pulsar problem, change "monopole_pulsar_parameters.f" to "dipole_pulsar_parameters.f" in the Makefile and recompile. Run the dipole pulsar problem via "$ mpirun -np 4 picsar2D" (feel free to use more cores if available by changing NPROC_R and NPROC_T accordingly in "source/input/dipole_pulsar_parameters.f"). The simulation will execute for 10000 timesteps and take a comparable amount of time as the higher resolution monopole test problem on the same number of cores. When the simulation has completed, set "is_dipole = True" in "analysis/sim_pars.py" and execute "$ python merge_routines.py" to join the output files together.

The dipole pulsar problem you just ran has the same parameters as simulation A1 of paper 3 but is at half the resolution and has one fifth the particles per cell. Execute "\$ python prtl_plot.py" to display an image of the logarithm of the particles per cell at the end of the simulation (compare with Fig. 5 of paper 3). You can also use the same routine to plot the logarithm of the cell-averaged gamma factor, the pair multiplicity, and the individual contributions to the current from electrons and positrons. To display the cell-averaged gamma factor, set "ptl_type = 'ptlg' " in "prtl_plot.py", and run "\$ python prtl_plot.py" again (compare with Fig. 6 of paper 3).

Additional interesting quantities to display are the spindown luminosity and the logarithm of the magnetization, which can be plotted by executing "\$ python poynt_plot.py" and "\$ python magnetization.py", respectively (compare with Fig. 1 and Fig. 2 of paper 3). Note that it is normal for the the spindown luminosity in the PIC simulation to be elevated compared to the force-free case, in particular at low resolution (factor of 1.5x is normal for this resolution). Finally, it is informative to display the four current magnitude via "\$ python four_current_density.py" (compare with Fig. 4a of paper 2).

The trajectories of tracer particles have also been saved, but before they can be displayed, they must first be constructed via "\$ python save_trajectories.py". The trajectories can then be plotted by executing "\$ python plot_trajectories.py", which will generate an image similar to Fig. 8 of paper 3. Individual particle trajectories with color showing the gamma factor along the trajectory can be plotted via "\$ python plot_trajectory.py". The variable "ptl_type" within the "plot_trajectory.py" script is used to select whether electron or positron trajectories are displayed.

Individual particle trajectories can also be superimposed on top of a background image. For instance, executing "\$ python flds_image.py" displays the particle trajectories on top of a black and white image of the current density in the meridional plane, similar to Fig. 7 of paper 3. The script "flds_image.py" can also be used to display combinations of electric fields, magnetic fields, and grid currents (e.g. $\boldsymbol{E} \cdot \boldsymbol{B}$, $\boldsymbol{E} \cdot \boldsymbol{J}$, etc.) weighted by functions of r and θ.

The particle injection prescription at the neutron star surface can be selected via the "mode_type" parameter in "dipole_pulsar_parameters.f". Paper 3 described four simulations that each had a different prescription for surface particle injection. The dipole pulsar simulation that you ran has "mode_type=2", which corresponds to simulation A1 of paper 3. The other three simulations discussed in paper 3 correspond to "mode_type=0" (simulation D1), "mode_type=3" (simulation D2), and "mode_type=4" (simulation A2). Selecting "mode_type=1" turns off surface charge emission, but keeps surface pair production turned on.

Set "mode_type = 0" in "dipole_pulsar_parameters.f", which corresponds to only surface charge emission (no pair production) and leads to a dead electrosphere. Also set "pdsample = 10", which determines the downsampling of the particles when they are saved to file. Recompile and run the simulation again (it will run much faster than before). The particle distribution can be displayed via "\$ python prtl_image.py", and the rotation profile can be plotted via "\$ python rot_plot.py" (compare with Fig. 3 of paper 3). One can also plot the surface charge on the neutron star as a function of θ relative to the vacuum case via "$ python inj_plot.py". The surface charge is driven to zero by the surface charge injection algorithm. Time-permitting, try running different mode types and displaying the results. Active magnetosphere simulations have "mode_type = 1", "mode_type = 2", or "mode_type = 4", and dead pulsar electrosphere simulations have "mode_type = 0" or "mode_type = 3".

5. Testing the Code

5.1. Charge Conservation

The compile time option "-Dkill_step=N_STEP" sets a variable in the code which switches the function of the mover after "N_STEP" timesteps. Initially, the regular electromagnetic particle mover is used (either Boris or Vay), but after "N_STEP" timesteps, particles are forced to move radially inward with a velocity close to the speed of light. All of the particles eventually fall into the inner conductor and are deleted, after which time the divergence of the electric field is everywhere zero (to machine precision) inside the simulation domain.

Compile and run the dipole pulsar problem with the option "-Dkill_step=4750". The "mode_type" you use doesn't matter, but it is important that "kill_step" is at least slightly more than twice the run time of the simulation, so all particles are deleted from the simulation before the last data dump. After merging the outputs, use "$ python calc_div.py" to plot the divergence of the electric

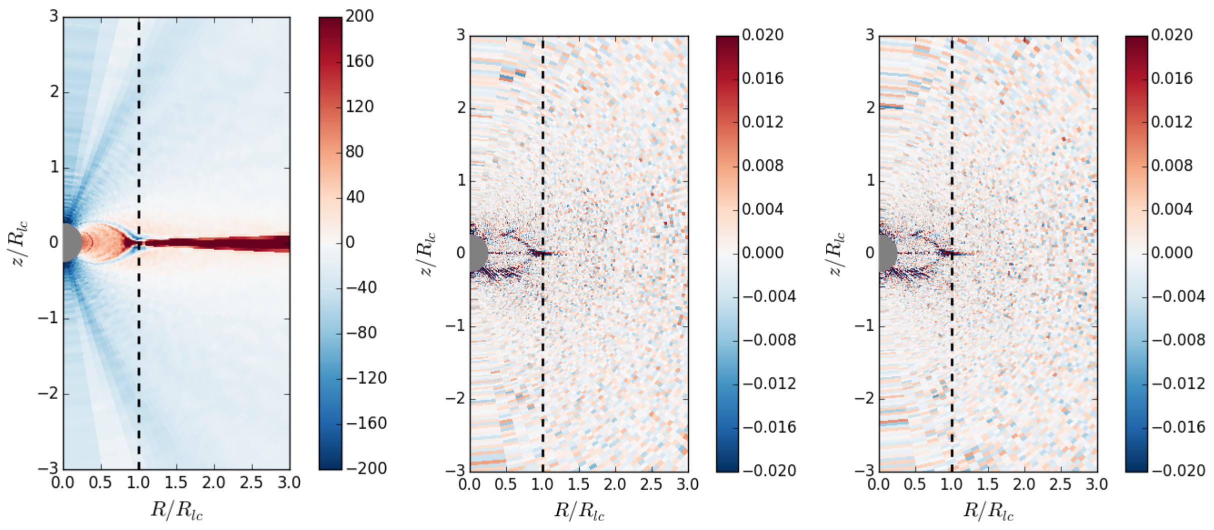

Fig. 2.— Divergence within the simulation domain for the aligned dipole rotator with "mode_type = 2" and "scfc = 1". Left panel: $t = 4000$ (Esirkepov); middle panel: $t = 10000$ (Esirkepov); right panel: $t = 10000$ (Villasenor-Buneman). Note the difference in scale for the left panel compared to the middle and right panels.

field in the simulation. Fig. 2 shows the divergence of the electric field at $t = 4000$ and at $t = 10000$ for "mode_type = 2" and "scfc = 1". It is also possible to repeat this test with Villasenor-Buneman rather than Esirkepov current deposition by changing "deposit_Esirkepov.f" to "deposit_VB.f" in the Makefile.

5.2. Additional Test Problems

Several additional test problems are included with *PICsar2D*. Each one has a parameter file in the "source/input" folder, and a file implementing problem-specific functions (e.g. initialization, particle injection, etc.) in the "source/prob" folder. The file "source/prob/init_default_parameters.f" contains default values for simulation parameters, and "source/prob/init_default_funcs.f" contains default implementations of problem-specific functions.

To compile a given test problem, it is necessary to change the names of the parameter and problem-specific functions files in the Makefile. Both the monopole and dipole pulsar problems use the same problem-specific functions: "pulsar_funcs.f" which imports routines from "psr_fields.f". Thus, it is only necessary to change the name of the parameter file in the Makefile when switching between the monopole and dipole pulsar runs.

The three additional test problems that are provided are "TM_mode", "two_charge_test", and "ExBtest". The associated parameter and problem-specific functions have the problem prefix with an underscore followed by "parameters.f" and "funcs.f" (e.g. "TM_mode_parameters.f" and "TM_mode_funcs.f"). The "ExBtest" problem also imports functions from "psr_fields.f". However, to avoid compilation errors it is necessary to remove "psr_fields.f" from the list of source files in the Makefile when compiling the "TM_mode" or "two_charge_test" problems. Sample *Python* parameter files for each test problem are provided in the "analysis/pars" folder. Simply overwrite "analysis/sim_pars.py" with the appropriate *Python* parameter file to analyze the output.

The "TM_mode" test problem simulates a transverse magnetic mode that is confined between conducting boundaries (see §4.1 of paper 1) and is used to check the second order convergence of the electromagnetic field solver. The associated analysis routine is "L2error.py", which shows an image of the error in the $\mathcal{L}_2$ norm between the analytical solution and the simulation at the end of the run.

The "ExBdrift" test problem initializes a test particle in corotation electric and magnetic fields (see §4.3 of paper 1). It tests the accuracy of the particle pusher for capturing the motion of charged particles in crossed electric and magnetic fields. The associated analysis routine is "ExBdrift_plot.py", which in its default mode displays the particle trajectory in the xy-plane. The same routine can also be used to display the test particle radius or azimuthal velocity as a function of time. This is a good test problem for showing the advantage of the Vay pusher over the Boris pusher for simulating ExB drift motion when the Larmor frequency is not resolved in time. To compile with the Boris pusher instead of the Vay pusher remove the "-Dvay" flag in the Makefile.

The “two_charge_test” test problem fires an electron and positron that start at the same location in the simulation domain in opposite directions (see §4.2 of paper 1). It is used to test the radiation absorbing boundary condition and the ability of the inner conducting boundary to hold a central charge. The associated analysis routine is “plot_inv_square.py”, which plots the radial component of the electric field at the end of the simulation compared to the analytical solution. The default version of the test assumes linear scaling in the radial direction. To compile with linear radial scaling remove the “-Dlogarithmic_r” flag in the Makefile.

REFERENCES

Belyaev, M. A. 2015, New Astronomy, 36, 37

Belyaev, M. A. 2015, MNRAS, 449, 2759

Belyaev, M. A. 2017, arXiv:1707.01598

Buneman, O., 1993, in Computer Space Plasma Physics: Simulation Techniques and Software, ed. H. Matsumoto & Y. Omura (Tokyo: Terra Scientific), 67–84