Introduction to Warp

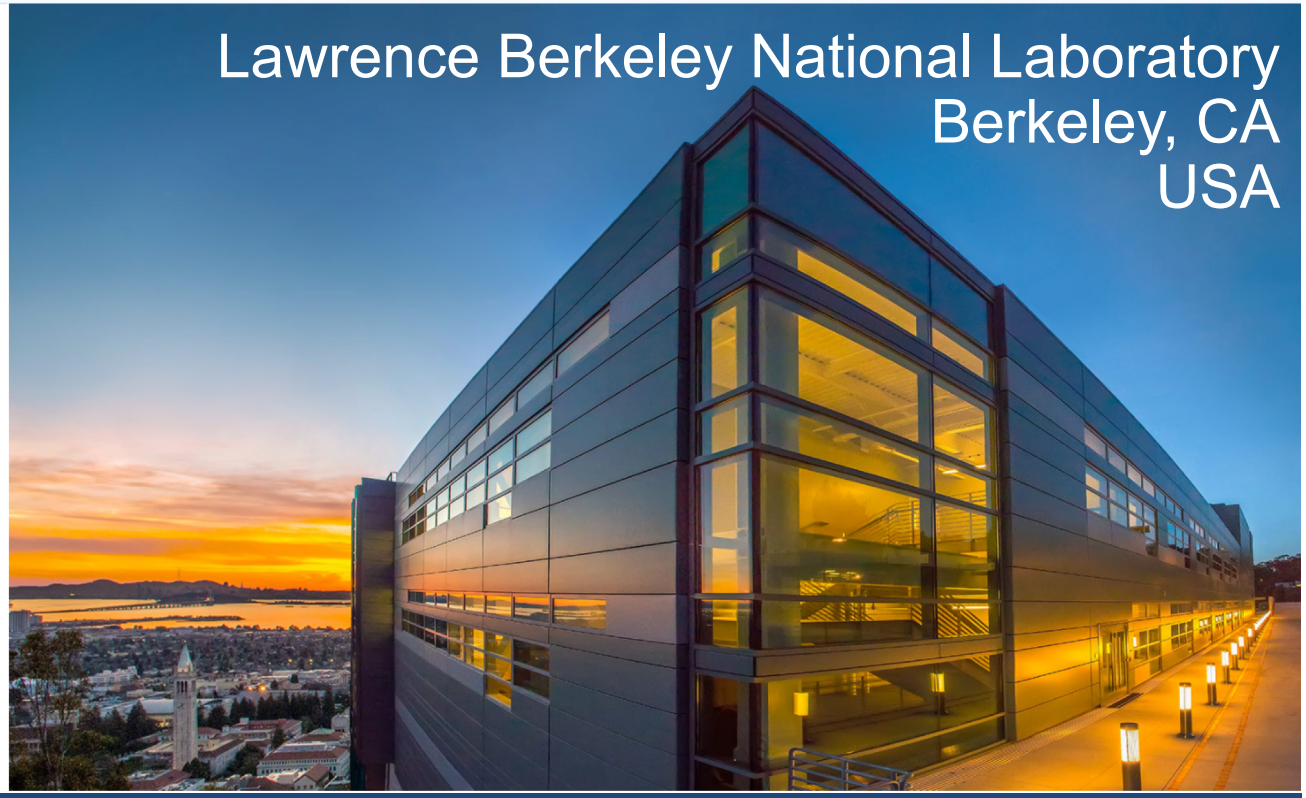
May 7-9, 2018

1st

BLAST

BERKELEY LAB ACCELERATOR SIMULATION TOOLKIT

workshop



Installing Warp

Installations instructions are given at :

`http://warp.lbl.gov/home/how-to-s/installation`

Pre-requisite:

`C and F95 compiler, python, numpy, Forthon, pygist or matplotlib`

Download Warp from bitbucket:

`https://bitbucket.org/berkeleylab/warp/downloads (get Warp_Release_4.5.tgz)`

Note: *detailed installation notes for various platforms in Warp/doc.*

Install

```
cd in the Warp/pywarp90 directory
make install
# see more details on website, including how to install in parallel
```

Update

```
git pull
cd pywarp90
# serial
make install
# parallel
make pinstall
```

Running Warp

A Warp input script is a Python script, e.g. script.py.

Running interactively:

```
% python –i script.py
```

Running non-interactively:

```
% python script.py
```

Documentation and help at the command prompt:

```
>>> warpdoc()

>>> warphelp()

>>> doc('name')   -- or --  In [2] name? with Ipython

>>> name+TAB+TAB  -- or --  In [2] name+TAB with Ipython for auto-completion
```

Warp units

Warp units are M.K.S. with the notable exception of energy, which is in eV.

A number of conversion constants are provided for convenience.

Example:

```
>>> piperad=2.*inch
>>> piperad
0.0508
```

```
inch = 0.0254    # inches to meter
nm   = 1.e-9     # nm to meter
um   = 1.e-6     # um to meter
mm   = 1.e-3     # mm to meter
cm   = 0.01      # cm to meter
ps   = 1.0e-12   # picosecond  to second
ns   = 1.0e-9    # nanosecond  to second
us   = 1.0e-6    # microsecond to second
ms   = 1.0e-3    # millisecond to second
kV   = 1.0e3     # kV to V
keV  = 1.0e3     # keV to eV
MV   = 1.0e6     # MV to V
MeV  = 1.0e6     # MeV to eV
mA   = 1.0e-3    # mA to A
uC   = 1.0e-6    # micro-Coulombs to Coulombs
nC   = 1.0e-9    # nano-Coulombs  to Coulombs
deg  = pi/180.0  # degrees to radians
```

Mathematical and Physical Constants

Those are provided to aid in the setup.

```
Mathematical Constants

pi       = 3.14159265358979323   # Pi
euler    = 0.57721566490153386   # Euler-Masceroni constant
e        = 2.718281828459045     # exp(1.)

Physical Constants

avogadro  = 6.02214129e23          # Avogadro's Number
amu       = 1.66053921e-27         # Atomic Mass Unit [kg]
clight    = 2.99792458e+8          # Speed of light in vacuum [m/s]
echarge   = 1.602176565e-19        # Proton charge [kg]
emass     = 9.10938291e-31         # Electron mass [kg]
eps0      = 8.854187817620389e-12  # Permittivity of free space [F/m]
jperev    = 1.602176462e-19        # Conversion factor, Joules per eV [J/eV]
mu0       = 1.2566370614359173e-06 # Permeability of free space [H/m]
boltzmann = 1.3806488e-23          # Boltzmann's constant [J/K]
planck    = 6.62606957e-34         # Planck's constant [J.s]
```

Example:

```
>>> pipesec=pi*piperad**2
>>> pipesec
0.008107319665559963
```

Warp script outline

Outline of basic 3D simulation script:

```
from warp import *      # Read in Warp

setup()                 # Setup graphics output

# Setup Lattice defining accelerator -- see "Lattice" How To
... python code to set variables and call lattice setup functions

# Setup initial beam -- see "Particles" How To
... python code to describe species and distribution parameters of initial beam

# Setup particle mover -- see "Particles" How To
... python code to setup particle mover including timestep etc.

# Setup simulation mesh and field solver -- see "Field Solver" How To
... python code to set mesh variables and setup fieldsolver

# Setup simulation diagnostics -- see "Diagnostics" How To
... python code to setup diagnostics

# Generate code
package("w3d"); generate()

# Advance simulation, say 1000 timesteps
step(1000)

# Save dump of run, if desired -- see Saving/Retrieving Data How To
dump()
```

Particles

Warp can handle an arbitrary number of particle species.

Predefined particle types:

- periodic table, electron, positron, proton, anti-proton, muon, anti-muon, neutron, photon.

Creating a particle species:

```
beam = Species(type=Potassium, charge_state=+1, name="Beam ions (K+)")
plasma_electrons = Species(type=Electron, name="Plasma electrons", color=red)
plasma_ions = Species(type=Hydrogen, charge_state=+1, name="Plasma ions", color=green)
```

Accessing species data (sample):

```
beam.sq       # gives the charge of the species
beam.charge   # same as beam.sq
beam.mass     # gives the mass of the species (also given by beam.sm)
beam.emitn    # normalized emittance
beam.hxrms    # time history of Xrms in the z-windows (as computed by Warp diagnostic)
```

Using species functions (sample):

```
plasma_electrons.ppzx() # plots the particles in z-x space (the particles will be red).
plasma_ions.getx()      # returns a list of all of the plasma ions x positions
plasma_ions.add_uniform_cylinder(...) # creates a cylindrical distribution of particles
```

Creating particles at initialization

Adding predefined distributions:

```
plasma_ions.add_gaussian_dist(...)     # adds Gaussian distribution
plasma_ions.add_uniform_box(...)       # adds uniform box
plasma_ions.add_uniform_cylinder(...)  # adds uniform cylinder
```

Adding user defined distributions:

```
def createmybeam():
    ... Python code to set x,y,z,vx,vy,vz
    beam.addpart(x,y,z,vx,vy,vz,...)
...
createmybeam()
```

Creating particles to be added or injected from surface

Injecting particles directly:

```
def injectplasma():
    plasma_electrons.addparticles(...)
    plasma_ions.addparticles(...)
installuserinjection(injectplasma)
```

Injecting particles from an emitting surface:

```
top.inject = 1
top.ainject = source_radius
top.binject = source_radius
w3d.l_inj_user_particles_v = true
def nonlinearsource():
    if w3d.inj_js == elec.js:
        # --- inject np particles of species elec
        # --- Create the particles on the surface
        r = source_radius*random.random(np)
        theta = 2.*pi*random.random(np)
        x = r*cos(theta); y = r*sin(theta)
        # --- Setup the injection arrays
        w3d.npgrp = np
        gchange('Setpwork3d')
        # --- Fill in the data. All have the same z-velocity, vz1.
        w3d.xt[:] = x
        w3d.yt[:] = y
        w3d.uxt[:] = 0.
        w3d.uyt[:] = 0.
        w3d.uzt[:] = vz1

installuserparticlesinjection(nonlinearsource)
```

Types of injection (top.inject)

```
0: turned off,
1: constant current,
2: space-charge limited (Child-Langmuir),
3: space-charge limited (Gauss's law)),
4: Richardson-Dushman thermionic emission
5: mixed Richardson-Dushman thermionic
and space-charge limited emission
6: user specified emission distribution
7: Taylor-Langmuir ionic emission
8: mixed Taylor-Langmuir ionic and space-
charge limited emission
```

Accessing particles data

Many functions are available to access particles data:

position: `getx(), gety(), getz()`
velocites: `getvx(), getvy(), getvz()`
momentum/mass: `getux(), getuv(), getuz()`
tranverse angles (x-velocity/z-velocity etc): `getxp(), getyp()`
gamma inverse: `getgaminv()`
particle identification number: `getpid()`
x-x' and y-y' statistical slopes of the distribution: `getxxpslope(), getyypslop()`

Examples:

```
n = beam.getn()      # returns macro-particle number
x = beam.getx()      # returns macro-particle x-coordinates
y = beam.gety()      # returns macro-particle y-coordinates
z = beam.getz()      # returns macro-particle x-coordinates
pid = beam.getpid()  # returns macro-particle identification numbers
```

Many arguments are available for downselection of particles based on positions, etc:

```
beam.getx(zl=0.,zu=0.1)  # returns x-coordinates for 0.<=z<=0.1
see documentation of selectparticles() for complete description
```

Simulation mesh

Setting the mesh extents:

```
w3d.xmmin =  -10.*mm    # x-mesh min
w3d.xmmax =   10.*mm    # x-mesh max
w3d.ymmin =  -10.*mm    # y-mesh min
w3d.ymmax =   10.*mm    # y-mesh max
w3d.zmmin = -250.*mm    # z-mesh min
w3d.zmmax =  250.*mm    # z-mesh max
```

Setting the number of meshes:

```
w3d.nx =  200  # size of x-mesh, mesh points are 0,...,nx
w3d.ny =  200  # size of y-mesh, mesh points are 0,...,ny
w3d.nz = 1000  # size of z-mesh, mesh points are 0,...,nz
```

The cell sizes are automatically computed during the call to generate():

```
>>> generate()
>>> w3d.dx, w3d.dy, w3d.dz
(0.0001, 0.0001, 0.0005)
```

Boundary conditions

Boundary conditions for the fields:

```
w3d.bound0   # lower z-mesh
w3d.boundnz  # upper z-mesh
w3d.boundxy  # transverse x-y
```

Valid values: `Dirichlet, Neumann, periodic.`
`In addition, for the Maxwell solver: openbc (uses the Perfectly Matched Layer).`

Boundary conditions for the particles:

```
top.pbound0   # lower z-mesh
top.pboundnz  # upper z-mesh
top.pboundxy  # transverse x-y
```

Valid values: `absorb, reflect, periodic.`

Field solvers

Various field solvers are available in Warp:

```
# --- multigrid Poisson solvers
fieldsolvers.multigrid.MultiGrid3D()    # 3-D geometry
fieldsolvers.multigridRZ.MultiGrid2D()  # 2-D x-z geometry
fieldsolvers.multigridRZ.MultiGridRZ()  # 2-D r-z geometry

fieldsolvers.MeshRefinement.MRBlock3D() # 3-D geometry with mesh refinement
fieldsolvers.MeshRefinement.MRblock2D() # 2-D geometry with mesh refinement

fieldsolvers.multigridRZ.MultiGrid2DDielectric() # solver in 2-D with variable
                                                 # dielectric constant (serial only)

# --- multigrid Magnetostatic solvers
fieldsolvers.magnetostaticMG.MagnetostaticMG()  # solver in 3-D geometry
fieldsolvers.MeshRefinementB.MRBlockB() # solver in 3-D geometry with mesh refinement

# --- electromagnetic Maxwell solvers
fieldsolvers.em3dsolver.EM3D() # 3-D, 2-D x-z/r-z, r-z+azimuthal decomposition
fieldsolvers.em3dsolverFFT.EM3DFFT() # spectral (FFT-based) solver in 3-D and 2-D x-z
EM3DPXR() # 3-D, 2-D x-z using optimized PICSAR library (available via PICSAR)
```

Field solvers usage

Multigrid Poisson solver:

```
solver = MultiGrid3D()
registersolver(solver)
```

Multigrid Poisson solver with mesh refinement:

```
solver = MRBlock3D()
solver.addchild(mins=...,maxs=...,refinement=[2,2,2]) # add refinement level 1
solver.children[0].addchild(mins=...,maxs=...,refinement=[2,2,2]) # add ref. level 2
registersolver(solver)
```

Electromagnetic Maxwell solver:

```
solver = EM3D()             # 3D solver
registersolver(solver)

solver = EM3D(l_2dzx=True)  # 2D XZ solver
registersolver(solver)

solver = EM3D(l_2drz=True)  # axisymmetric RZ solver
registersolver(solver)

solver = EM3D(circ_m=1)     # axisymmetric RZ solver + first azimuthal dipole mode
registersolver(solver)
```

Internal conductors

Many primitives can be combined to define internal conductors

Cylinders:

Cylinder (radius,length,theta=0.,phi=0.,...)
Zcylinder (radius,length,...)
ZCylinderOut (radius,length,...)
ZCylinderElliptic (ellipticity,radius,length,...)
ZCylinderEllipticOut (ellipticity,radius,length,...)
ZRoundedCylinder (radius,length,radius2,...)
ZRoundedCylinderOut (radius,length,radius2,...)
Xcylinder (radius,length,...)
XCylinderOut (radius,length,...)
XCylinderElliptic (ellipticity,radius,length,...)
XCylinderEllipticOut (ellipticity,radius,length,...)
Ycylinder (radius,length,...)
YCylinderOut (radius,length,...)
YCylinderElliptic (ellipticity,radius,length,...)
YCylinderEllipticOut (ellipticity,radius,length,...)
Annulus (rmin,rmax,length,theta=0.,phi=0.,...)
Zannulus (rmin,rmax,length,...)
ZAnnulusElliptic (ellipticity,rmin,rmax,length,...)

Planes/Rectangles:

Plane (z0=0.,zsign=1,theta=0.,phi=0.,...)
Xplane (x0=0.,xsign=1,...)
Yplane (y0=0.,ysign=1,...)
Zplane (z0=0.,zsign=1,...)
Box (xsize,ysize,zsize,...)

Cones:

Cone (r_zmin,r_zmax,length,theta=0.,phi=0.,...)
Cone (r_zmin,r_zmax,length,...)
ZConeOut (r_zmin,r_zmax,length,...)
ConeSlope (slope,intercept,length,theta=0.,phi=0.,...)
ZConeSlope (slope,intercept,length,...)
ZConeOutSlope (slope,intercept,length,...)

Others:

Sphere (radius,...)
ZElliptoid (ellipticity,radius,...)
ZTorus (r1,r2,...)
ZGrid (xcellsize,ycellsize,length,thickness,...)
Beamletplate (za,zb,z0,thickness,...)
CADconductor(filename,voltage,...)

Surfaces of revolution:

ZSrfrvOut (rofzfunc,zmin,zmax,...) / **XSrfrvOut** (rofxfunc,xmin,xmax,...) / **YSrfrvOut** (rofyfunc,ymin,ymax,...)
ZSrfrvIn (rofzfunc,zmin,zmax,...) / **XSrfrvIn** (rofxfunc,xmin,xmax,...) / **YSrfrvIn** (rofyfunc,ymin,ymax,...)
ZSrfrvInOut (rminofz,rmaxofz,...) / **XSrfrvInOut** (rminofx,rmaxofx,...) / **YSrfrvInOut** (rminofy,rmaxofy,...)
ZSrfrvEllipticOut (ellipticity,rofzfunc,zmin,zmax,rmax,...)
ZSrfrvEllipticIn (ellipticity,rofzfunc,zmin,zmax,rmin,...)
ZSrfrvEllipticInOut (ellipticity,rminofz,rmaxofz,zmin,zmax,...)

Help: generateconductors_doc ()

Internal conductors properties & usage

The internal conductors may have the following properties:

```
Voltage          = 0.     # conductor voltage
Condid           = 'next' # conductor ID; will be set to next available ID if 'next'
Name             = None   # conductor name
Material         ='SS'    # conductor material type (for secondaries), other options are 'Cu', 'Au'
Conductivity     = None   # conductor conductivity (for use with Maxwell solver)
Permittivity     = None   # conductor permittivity (for use with Maxwell solver)
Permeability     = None   # conductor permeability (for use with Maxwell solver)
Laccuimagecharge = False  # if True, collect history of particles charge accumulated/emitted
                          # by/from conductor, as well as image charges from nearby particles
                          # (using integral of Gauss Law around conductor)
```

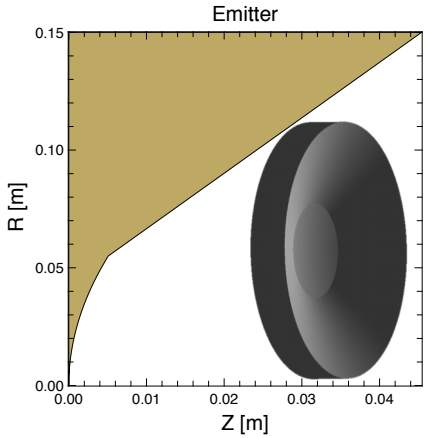
Examples:

```
# --- addition of two conductors
z1 = ZCylinder(0.1,1.0,zcent=-0.5,voltage=-1.*keV)
z2 = ZCylinder(0.12,1.0,zcent=+0.5,voltage=+1.*keV)
zz = z1 + z2
installconductor(zz)

# --- substraction of two conductors
souter = Sphere(radius=1.,voltage=1.)
sinner = Sphere(radius=0.8,voltage=souter.voltage,condid=souter.condid) # conductors must have same ID
shollow = souter - sinner

# --- intersection of two conductors
zz = ZCylinder(0.1,1.0,voltage=10.*keV)
yy = YCylinder(0.1,1.0,voltage=zz.voltage,condid=zz.condid)  # conductors must have same ID
installconductor(zz*yy)
```

Help: generateconductors_doc ()

Internal conductors as a surface of revolution

The internal conductors may have the following properties:




```
# --- Source emitter parameters
channel_radius          = 15.*cm
diode_voltage           = 93.*kV
source_radius           = 5.5*cm
source_curvature_radius = 30.*cm # --- radius of curvature of emitting surface
pierce_angle            = 67.
plate_width             = 2.5*cm # --- thickness of aperture plate
piercezlen              = 0.04

piercezlen = (channel_radius - source_radius)*tan((90.-pierce_angle)*pi/180.)
rround = plate_width/2.

# --- Outer radius of Pierce cone
rpierce = source_radius + piercezlen*tan(pierce_angle*pi/180.)

# --- Depth of curved emitting surface
sourcezlen = (source_radius**2/(source_curvature_radius + sqrt(source_curvature_radius**2 - source_radius**2)))

# --- the rsrf and zsrf specify the line in RZ describing the shape of the source and Pierce cone .
# --- The first segment is an arc, the curved emitting surface.
source = ZSrfrv(rsrf=[0., source_radius, rpierce, channel_radius, channel_radius],
                zsrf=[0., sourcezlen, sourcezlen + piercezlen, sourcezlen + piercezlen, 0.],
                zc=[source_curvature_radius, None, None, None, None],
                rc=[0., None, None, None, None],
                voltage=diode_voltage)

# --- Create emitter conductors
installconductor(source, dfill=largepos)
```

Help: generateconductors_doc ()

Lattice

The easiest and most elegant way to setup a lattice in Warp uses a MAD-like syntax.

List of lattice elements:

```
Drft()   # drift
Bend()   # bend
Dipo()   # dipole
Quad()   # quadrupole
Sext()   # sextupole
Accl()   # accelerating gap
Hele()   # hard edge multipole
Emlt()   # electric multipole
Mmlt()   # magnetic multipole
Bgrd()   # gridded B-field
Pgrd()   # gridded E-field
```

Usage:

```
qf = Quad(l=10.*cm,db= 1.5,ap=3.*cm)     # focusing magnetic quadrupole
qd = Quad(l=10.*cm,db=-1.5,ap=3.*cm)     # defocusing magnetic quadrupole
dd = Drft(l=5.*cm,ap=3.*cm)              # drift
fodo1 = qf + dd + qd + dd                # 1 FODO section
s1 = 10*fodo1                            # section 1 is a repetition of 10 fodo1
...define s2                             # define section 2
...define s3                             # define section 3
lattice = s1+s2+s3                       # the lattice is the sum of the 3 sections
madtowarp(lattice)                       # converts MAD-like lattice into Warp native format and initializes

or

getlattice(file)                         # reads MAD lattice from file, converts and initializes
                                         # (not all MAD files are readable without tweaking file or Warp reader)
```

Note: Maps elements also exist for quads, bends, drifts and RF kicks.

Help: latticedoc()

Lattice (native formulation)

The native lattice element syntax is as described here.

List of lattice elements:

```
addnewdrft(...)    # Drift element (can be used to load aperture/pipe structure in some field solvers)
addnewquad(...)    # Hard-edged or self-consistent quadrupole (magnetic or electric) focusing elements
addnewbend(...)    # Bends
addnewdipo(...)    # Hard-edged or self-consistent dipole bending elements
addnewaccl(...)    # Hard-edged accelerating gaps
addnewsext(...)    # Hard-edged sextupole elements
addnewhele(...)    # Hard-edged arbitrary electric and magnetic multipole moments
addnewemlt(...)    # Axially varying arbitrary electric multipole moments
addnewmmlt(...)    # Axially varying arbitrary magnetic multipole moments
addnewbgrd(...)    # Magnetic field specified on a two or three-dimensional grid
addnewegrd(...)    # Electric field specified on a two or three-dimensional grid
addnewpgrd(...)    # Electrostatic potential specified on a two or three-dimensional grid
```

Usage:

```
rp = 2.*cm     # aperture radius [m]
Bp = 1.        # pole tip field at aperture [Tesla]
lq  = 20.*cm   # Quadrupole hard-edge axial length [m]
ld  = 20.*cm   # Drift length between hard-edge quadrupoles [m]
dbdx = Bp/rp   # magnetic field gradient
```

```
# --- define FODO cell elements
addnewquad(zs=0.,ze=lq,db=dbdx,ap=rp)
addnewdrift(zs=lq,ze=lq+ld,ap=rp)
addnewquad(zs=lq+ld,ze=2.*lq+ld,db=-dbdx,ap=rp)
addnewdrift(zs=2.*lq+ld,ze=2.*lq+2.*ld,ap=rp)
```

```
# --- set lattice periodicity:
top.zlatstrt = 0.                  #  z of lattice start (added to element z's on generate).
top.zlatperi = 2.*(lq+ld)          #  lattice periodicity
```

```
Note: http://warp.lbl.gov/home/how-to-s/lattice/time-dependent-lattice-elements
for setting time-dependent elements.
```

Help: latticedoc()

Stepping

The main stepping is performed by calls to the function step:

```
step(nstep)
```

The user can add functions to be called before and after step:

```
@callfrombeforestep # installs next function to be called automatically before a step
def myplotfunc1():
    ...

@callfromafterstep # installs next function to be called automatically after a step
def myplotfunc2():
    ...
```

Example

```
plotfreq = 100  # --- frequency of plotting
@callfromafterstep
def myplotfunc():
    if top.it%plotfreq==0:  # --- execute only every plotfreq time steps
        beam.ppzx()         # --- plot ZX projection
        fma()               # --- save plot and clear for next plot
```

Plotting basics (using pygist)

Generic plotting subroutines: `type doc(subroutine) for list of options`

```
setup()          # setup graphics output
winon(i)         # open window number i (0<=i<10)
window(i)        # select window number I (opens window if not already)
ppgeneric()       # generic particle and fields plotting routine
pla()            # plot a graph of y vs x
limits()         #  sets plot limits in order left, right, bottom, top
ptitles()        # draw plot titles on the current frame
fma()            # frame advance
refresh()        # refresh of window for live plots
eps('filename') # output current plot in epsi file
pdf('filename') # output current plot in pdf file
```

Viewing file:

```
gist myfile.cgm # open graphic output file
gist commands:
  f (or F)  - Forward one frame          b (or B)  - Backward one frame

  10f - Forward 10 frames                F    - Forward n frames
  10b - Backward 10 frames               B    - Backward n frames
                                         10n  - Set n to 10 (10 is the default)

  r    - Redraw current frame
  g    - Goto first frame                G    - Goto last frame

  s    - Send current frame to output device (see below for more info)
  q    - Quit
```

Plotting particles

Particle projection examples:

```
beam.ppxy()    # --- projection in X-Y
beam.ppxvx()   # --- projection in X-Vx
beam.pptrace() # --- Plots X-Y, X-X', Y-Y', Y'-X' in single page
```

List of all projections:

```
beam.pprrp()      beam.ppxux()      beam.ppyvz()      beam.ppzuz()
beam.pprtp()      beam.ppxvx()      beam.ppyyp()      beam.ppzvperp()
beam.pprvr()      beam.ppxvz()      beam.ppzbx()      beam.ppzvr()
beam.pprvz()      beam.ppxxp()      beam.ppzby()      beam.ppzvtheta()
beam.pptrace()    beam.ppxy()       beam.ppzbz()      beam.ppzvx()
beam.ppvzvperp()  beam.ppybx()      beam.ppzex()      beam.ppzvy()
beam.ppxbx()      beam.ppyby()      beam.ppzey()      beam.ppzvz()
beam.ppxby()      beam.ppybz()      beam.ppzez()      beam.ppzx()
beam.ppxbz()      beam.ppyex()      beam.ppzke()      beam.ppzxp()
beam.ppxex()      beam.ppyey()      beam.ppzr()       beam.ppzxy()
beam.ppxey()      beam.ppyez()      beam.ppzrp()      beam.ppzy()
beam.ppxez()      beam.ppyuy()      beam.ppzux()      beam.ppzyp()
beam.ppxpyp()     beam.ppyvy()      beam.ppzuy()
```

History plots:

```
Many history plots (vs time or vs z) are available
histplotsdoc() # --- list of available plots
hpdoc()        # --- list of possible arguments
```

Plotting fields (electrostatic)

List of slice plots:

```
Plots contours of charge density (rho) or electrostatic potential (phi) or self E fields in
various planes.
  pcrhozy(), pcrhozx(), pcrhoxy()          # charge density
  pcphizy(), pcphizx(), pcphixy()          # scalar potential
  pcselfezy(), pcselfezx(), pcselfexy()    # self electric field
  pcjzy(), pcjzx(), pcjxy()                # current density
  pcbzy(), pcbzx(), pcbxy()                # magnetic field
  pcazy(), pcazx(), pcaxy()                # vector potential
```

Plotting fields (electromagnetic)

Main functions:

```
em = EM3D()
# --- plotting electric field components in x, y, r, theta, z
em.pfex(); em.pfey(); em.pfer(); em.pfet(); em.pfez()
# --- plotting magnetic field components in x, y, r, theta, z
em.pfbx(); em.pfby(); em.pfbr(); em.pfbt(); em.pfbz()
# --- plotting current density components in x, y, r, theta, z
em.pfjx(); em.pfjy(); em.pfjr(); em.pfjt(); em.pfjz()
# --- plotting charge density
em.pfrho()
```

Options (in addition to those of ppgeneric):

```
- l_transpose=false: flag for transpose of field
- direction=None: direction perpendicular to slice for 2-D plot of 3-D fields
                  (0='x', 1='y', 2='z')
- slice=None: slice number for 2-D plot of 3-D fields (default=middle slice)
```

Examples:

```
# --- plotting of Ez in plane Z-X
em.pfez(direction=1,l_tranpose=1)
# --- plotting of Ex in plane X-Y
em.pfex(direction=2)
```

Saving/restarting/retrieving - default

Saving/restarting Warp simulations:

```
dump()               # dumps simulation in external file
restart('filename')  # restart simulation from file
```

Saving/retrieving data in external binary files:

```
# --- saving
fout = PWpickle.PW("save.pkl")
fout.var = var
fout.close()
# --- retrieving
fin = PRpickle.PR("save.pkl")
var = fin.var
fin.close()
```

By default, binary files use Python pickle format. Other formats (e.g. hdf5) are available.

Saving in the new PIC I/O standard OpenPMD is also possible with the electromagnetic PIC solver.

Saving/restarting/retrieving/viewing - OpenPMD

New OpenPMD standard:

http://www.openpmd.org

Saving/retrieving data in external OpenPMD hdf5 files:

```
# --- saving
from warp.data_dumping.openpmd_diag import ParticleDiagnostic
from warp.data_dumping.openpmd_diag import ElectrostaticFieldDiagnostic
…
part_diag = ParticleDiagnostic(period = 100, top = top, w3d = w3d,
                               species = {species.name: species for species in listofallspecies},
                                         comm_world=comm_world, lparallel_output=False)
efield_diag = ElectrostaticFieldDiagnostic(period = 100, solver=solverE, top=top, w3d=w3d,
                                           comm_world = comm_world, write_dir='diags/hdf5')
# --- retrieving
from opmd_viewer import OpenPMDTimeSeries
ts = OpenPMDTimeSeries('./diags/hdf5/')
```

Viewing using:

- OpenPMD Jupyter notebook viewer: follow instructions at https://github.com/openPMD/openPMD-viewer
- Pyside-based GUI: follow instructions at https://ecp-warpx.github.io/visualization/pyside.html

Command line options

Main command line options:

```
-p, --decomp npx npy npz # Specify the domain decomposition when running in parallel.
                         # The npx, npy, and npz are the number of domains along the x, y,
                         # and z axes. Note that npx*npy*npz must be the same as the total
                         # number of processors.

--pnumb string # Specify the runnumber to use, instead of the automatically incremented
               # three digit number. Any arbitrary string can be specified.
```

Example when running in parallel:

```
# --- run on 64 cores, with decomposition of 4 in x, 2 in y and 8 in z, with mpi4py
mpirun –np 64 python myscript.py –p 4 2 8
```

User specified line arguments:

```
import warpoptions    # --- first import warpoptions module
# --- add arguments
warpoptions.parser.add_argument('--radius',dest='radius',type=float,default=0.)
warpoptions.parser.add_argument('--volt',dest='voltage',type=float,default=0.)
from warp import *    # --- import Warp
radius = warpoptions.options.radius
voltage = warpoptions.options.voltage

Usage: python -i myinputfile.py --radius 0.1 --volt 93000.
```