U.S. Particle Accelerator School
Education in Beam Physics and Accelerator Technology

Self-Consistent Simulations of Beam and Plasma Systems

Steven M. Lund, Jean-Luc Vay, Rémi Lehe and Daniel Winklehner
Colorado State U., Ft. Collins, CO, 13-17 June, 2016

W2. Introduction to the Warp framework

Jean-Luc Vay
Lawrence Berkeley National Laboratory
(with many contributions from D. Grote & S. Lund)

Outline

- Intro
- Installing & running Warp
- Units and variables
- Warp script outline
- Particles
- Simulation mesh
- Boundary conditions
- Fields solvers
- Internal conductors & lattice
- Stepping
- Diagnostics and plots
- Saving/retrieving data
- Command line options

Warp is a very large code and we will only present a snapshot of the possibilities. More will be presented in the examples.

History

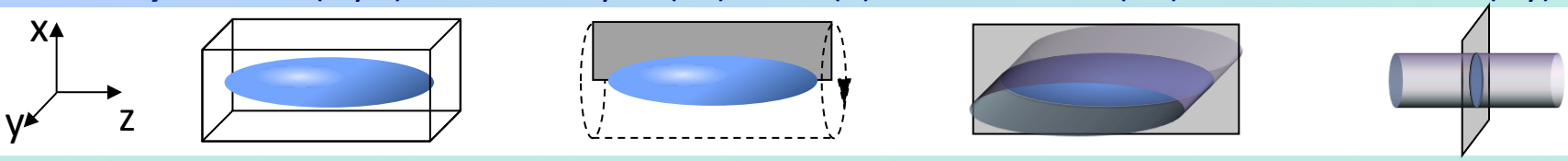
- 1989: code started by A. Friedman; electrostatic PIC with 3D Poisson solver in square pipe geometry using sine transforms
  - with inputs and contributions from I. Haber.
  - name Warp chosen to denote speed, later also denotes "warped" Fresnet-Serret coordinates used in bends.
- 1991: D. Grote joins and becomes main developer, contributing more general 3D Poisson solver and beam loading module, followed by countless contributions since.
- 1995: S. Lund start contributing, in particular beam loading module using various distributions, such as waterbag, Gaussian, and thermal.
- Late 1990's: R. Kishek begins using Warp to model experiment at UMD, and in undergraduate and graduate classes. He contributes some code, including various diagnostics used at UMD and UMER, and standardized machine description for UMER.
- 2000: D. Grote develops Forthon, transitioning Warp from Basis to Python, and parallelizes Warp using MPI (replacing previous PVM parallelization).
- 2000: J.-L. Vay joins the development team and contributes the EM solver, ES and EM AMR, boosted frame, Lorentz invariant particle pusher, etc.
- 2015: R. Lehe joins the development team and contributes the EM Circ solver, OpenPMD and hdf5 IO, mpi4py; also develops spectral EM circ stand-alone module FBPIC (coupling to Warp in development).
- 2016: RadiaSoft's open source Docker and Vagrant containers make WARP available in the cloud
  -> available for beta test via http://beta.sirepo.com/warp

History

- Other contributors include:
  - Ron Cohen: drift-Lorentz mover, other fixes.
  - Bill Sharp: Circe module.
  - Michiel deHoon: Hermes module.
  - Sven Chilton: generalized KV envelope solver (with S. Lund).
  - Arun Persaud: making Warp PEP8 compliant, various fixes and updates, also adding capability to download parameters and diagnostic output from NDCX-II run database, automatically run Warp, and overlay results with experimental data, thereby facilitating machine optimization.
  - Henri Vincenti: fixes to Circ, OpenPMD, and EM solver; development of optimized PICSAR kernel with tiling, OpenMP and advanced vectorization.
  - Manuel Kirchen: mpi4py interface, boosted particle diagnostics and FBPIC module.
  - Patrick Lee: boosted particle diagnostics, other fixes.
  - Irene Dornmair: EM initialization for relativistic beams.

Warp: A Particle-In-Cell framework for accelerator modeling

- **Geometry:** 3-D (x,y,z) | axisym. (r,z) + FFT(θ) | 2-D (x,z) | 2-D (x,y)

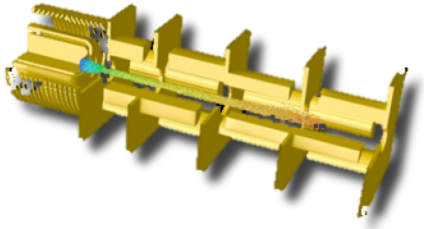


- **Reference frame:** **lab** | **moving-window** | **Lorentz boosted**

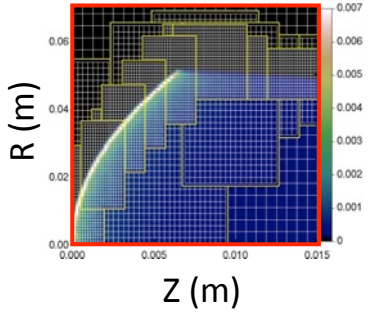
z | z-vt | $\gamma(z-vt)$; $\gamma(t-vz/c^2)$

- **Field solvers**
  - electrostatic/magnetostatic - FFT, multigrid; AMR; implicit; cut-cell boundaries

Versatile conductor generator accommodates complicated structures



Automatic meshing around ion beam source emitter

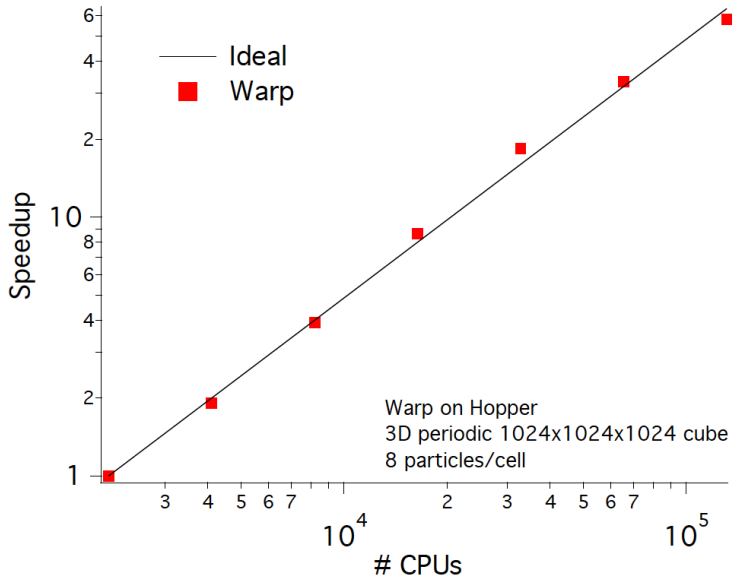


  - Fully electromagnetic – Yee/nodal mesh, arbitrary order, spectral, PML, MR
- **Accelerator lattice:** general; non-paraxial; can read MAD files
  - solenoids, dipoles, quads, sextupoles, linear maps, arbitrary fields, acceleration.
- **Particle emission & collisions**
  - particle emission: space charge limited, thermionic, hybrid, arbitrary,
  - secondary e- emission (Posinst), ion-impact electron emission (Txphysics) & gas emission,
  - Monte Carlo collisions: ionization, capture, charge exchange.

Warp is parallel, combining modern and efficient programming languages

- **Parallellization:** MPI (1, 2 and 3D domain decomposition)

Parallel strong scaling of Warp 3D PIC-EM solver on Franklin supercomputer (NERSC)



- **Python and FORTRAN*:** "steerable," input decks are programs

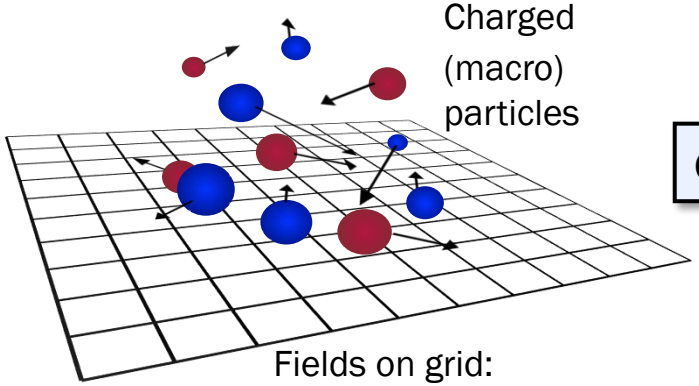
```
From warp import *          <-- Imports Warp modules and routines in memory
…
nx = ny = nz = 32           <-- Sets # of grid cells
dt = 0.5*dz/vbeam           <-- Sets time step
…
initialize()                <-- Initializes internal FORTRAN arrays
step(zmax/(dt*vbeam))       <-- Pushes particles for N time steps with FORTRAN routines
…
```

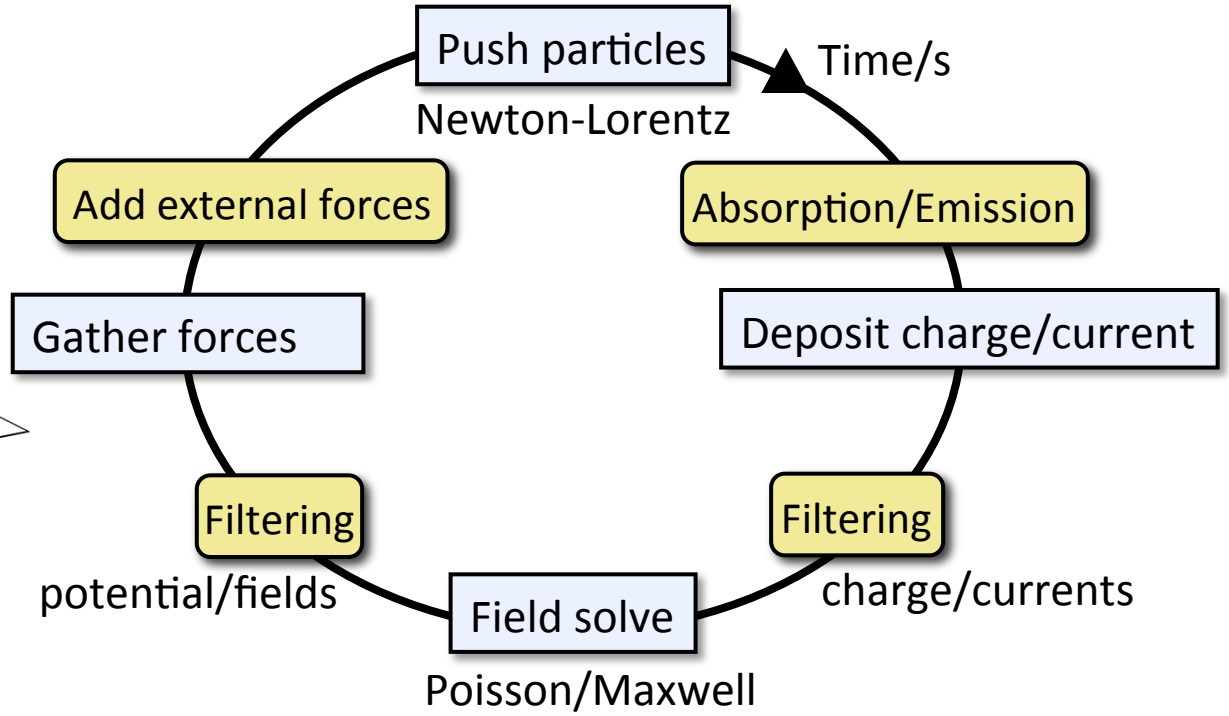
*http://hifweb.lbl.gov/Forthon (wrapper supports FORTRAN90 derived types) – dpgrote@lbl.gov

6

Warp's standard PIC loop

Particle-In-Cell (PIC)





+ external forces (accelerator lattice elements),

+ absorption/emission (injection, loss at walls, secondary emission, ionization, etc),

+ filtering (charge,currents and/or potential,fields).

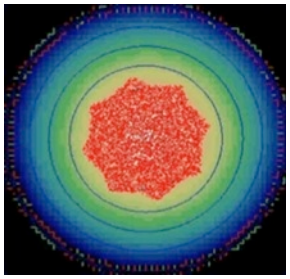
Warp's versatile programmability enables great adaptability

Standard PIC

Laboratory frame

Example:

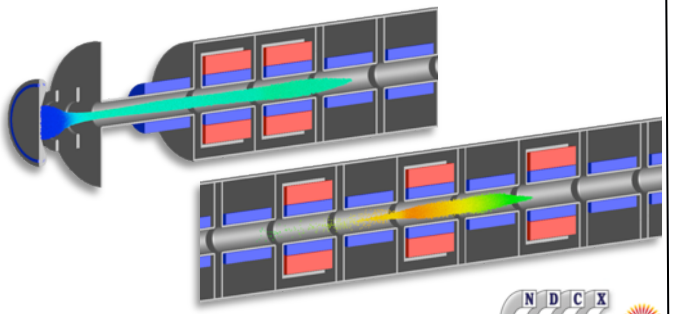
Alpha anti-H trap



Moving window

Example:

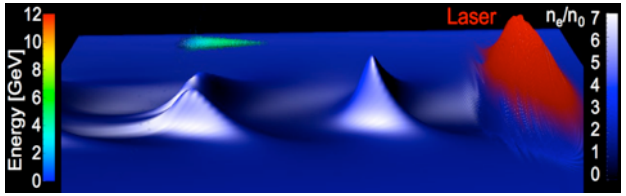
Beam generation and transport



Lorentz Boosted frame

Example:

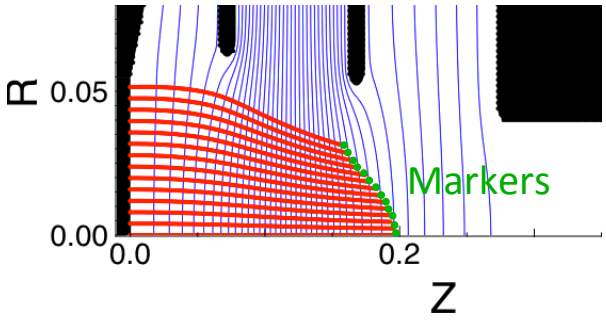
Laser plasma acceleration



Non-standard PIC

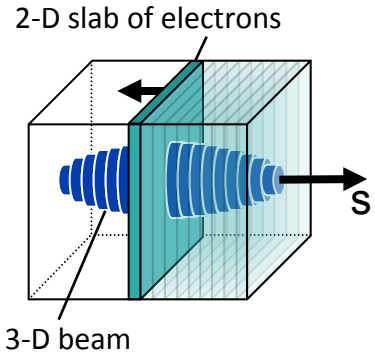
Steady flow

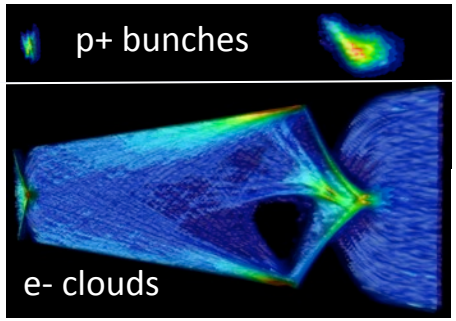
Example: Injector design



Quasi-static

Example: electron cloud studies

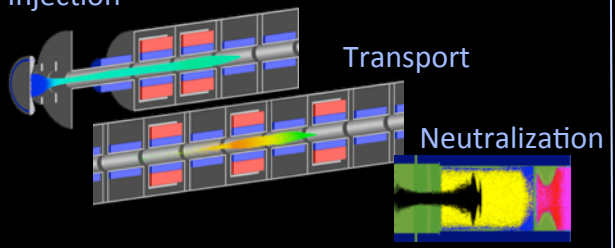




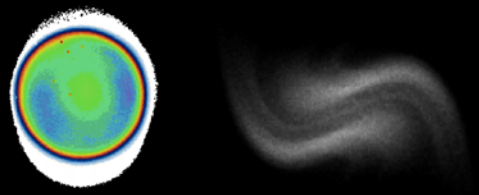
SPS - CERN

Sample applications

Space charge dominated beams

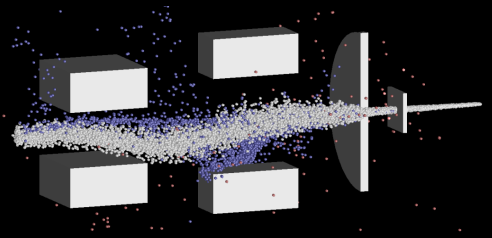


Beam dynamics in rings



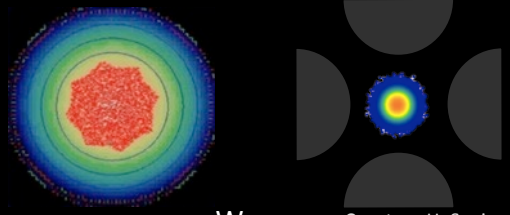
UMER

Multi-charge state beams



LEBT – Project X

Traps



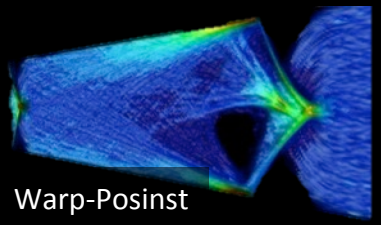
Warp

Courtesy H. Sugimoto

Alpha anti-H trap

Paul trap

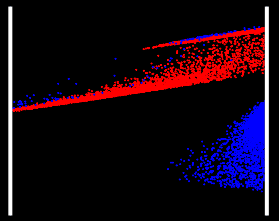
Electron cloud effects



Warp-Posinst

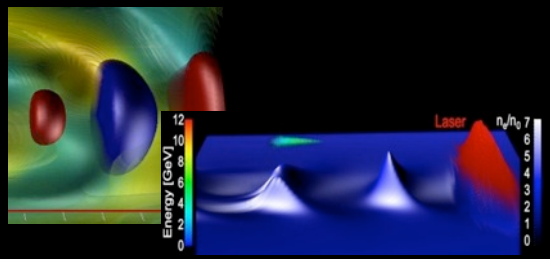
SPS

Multi-pacting



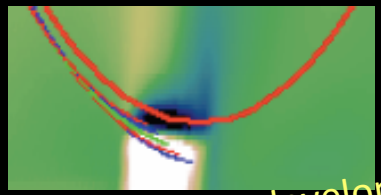
"Ping-Pong" effect

Plasma acceleration



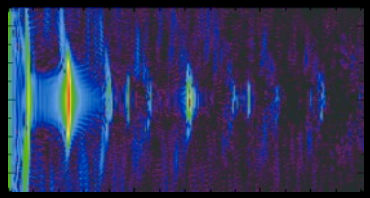
BELLA, FACET, …

3D Coherent Synchrotron Radiation



in development

Free Electron Lasers



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
download Warp_Release_4.4.tgz
```

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

# 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 injected

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)  # 3D 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)
```

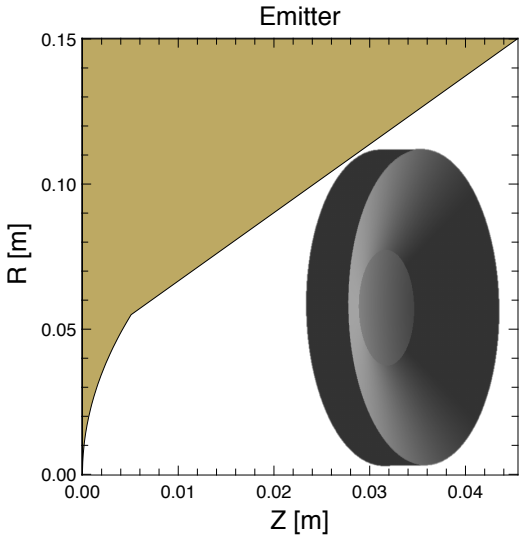
Examples:

```
# --- 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*fodo                             # 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

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)
pgeneric()       # 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 all projections:

```
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 deensity 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

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.

Command line options

Saving/restarting Warp simulations:

```
-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 7.0 -voltage 93000.
```

More infos

Warp website: http://warp.lbl.gov

Publications:

A. Friedman, R. H. Cohen, D. P. Grote, S. M. Lund, W. M. Sharp, J.-L. Vay, I. Haber, and R. A. Kishek
"Computational Methods in the Warp Code Framework for Kinetic Simulations of Particle Beams and Plasmas"
IEEE Trans. Plasma Sci. vol. 42, no. 5, p. 1321 (2014)
http://dx.doi.org/10.1109/TPS.2014.2308546.

J.-L. Vay, D. P. Grote, R. H. Cohen, A. Friedman
"Novel Methods in the Particle-In-Cell Accelerator Code-Framework Warp"
Comput. Sci. Disc. 5 014019 (2012)
http://dx.doi.org/10.1088/1749-4699/5/1/014019.

David P. Grote, Alex Friedman, Jean-Luc Vay, and Irving Haber
"The WARP Code: Modeling High Intensity Ion Beams"
AIP Conf. Proc. 749, pp. 55-58 (2004)
http://dx.doi.org/10.1063/1.1893366.

Alex Friedman, David P. Grote, and Irving Haber
"Three-dimensional particle simulation of heavy-ion fusion beams"
Phys. Fluids B 4, 2203 (1992)
http://dx.doi.org/10.1063/1.860024.

More at http://warp.lbl.gov/home/publications.