SAND2020-10623C

SIMPIC: A Simple Particle-in-Cell Code

B.T. Yee, M.M. Hopkins

Abstract

Particle methods are appealing tools for modeling the behavior of plasmas for several reasons. Their implementation can be extremely simple both in terms of the physical models and the numerical methods. However, despite this simplicity they are capable of reproducing complex plasma dynamics. From a pedagogical perspective, they provide an appreciation for the connection between the microscopic view of a plasma and its characteristic collective effects. The primary downsides of particle methods are their computational cost and the stochastic nature of their output. This talk will review the foundational principles of the particle-in-cell (PIC) method and basic numerical methods required for its implementation. From there, some of the basic constraints for accurate solutions will be presented, followed by a hands-on demonstration of using a PIC code to simulate the two-stream instability. The results from this simulation will then be compared to the analysis of the instability and the fluid results. By the end of this talk, attendees can expect to learn the essential components of PIC codes, important considerations in interpreting their results, and publicly available resources.

Description

This program is a simple particle-in-cell code designed to accompany the “Plasma Physics Fundamentals” workshop at the 73rd Annual Gaseous Electronics Conference. It uses common methods and models that are mostly documented in the book “Plasma Physics via Computer Simulation” by Birdsall and Langdon. More specifically, it is a 1D1V electrostatic PIC code in a bounded domain which is solved using the leapfrog and finite difference methods. Given its intended use for pedagogy it is written with a greater emphasis on readability and brevity rather than speed. For the same reasons, the code is written in Python for its portability and simple syntax. Despite this, the code performs reasonably fast for particle populations up to 100k, sufficient to replicate important kinetic effects in plasma sheaths.

Funding Statement

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy’s National Nuclear Security Administration under contract DE-NA0003525.

Appendix: Code

simpic

```
#!/usr/bin/env python3

def cprint(s, b, **kwargs):
    """
    Conditionally print a string.

    Keyword arguments:
    s -- string
    b -- boolean
    """
    if b:
        print(s, **kwargs)

def save_grid_data(savedir, step, steps, arr):
    """
    Save data presumed to be on a grid.

    Keyword arguments:
    name -- base name of quantity being saved
    step -- current timestep
    steps -- total number of timesteps
    arr -- array to save
    """
    if not os.path.exists(savedir):
        os.makedirs(savedir)
    order = int(np.ceil(np.log10(steps)))
    fmt = '%%.0%ii' % order
    fn = os.path.join(savedir, fmt % step)
    return np.save(fn, arr)

if __name__ == "__main__":
    import argparse
    import configparser
    import os
    import shutil
    import warnings

    import numpy as np

    from lib.particlelist import ParticleList

    # constants
    e = 1.602e-19  # C, elementary charge
    me = 9.109e-31  # kg, electron mass
    k = 1.381e-23  # J/K, Boltzmann constant
    epsilon_0 = 8.854e-12  # F/m, permittivity of free space
    from math import pi
```

```
parser = argparse.ArgumentParser(
        description='A simple particle-in-cell code.')
parser.add_argument(
        'input_deck', type=str, help='Input deck specifying simulation '
        'physics')
args = parser.parse_args()

# read in settings
config = configparser.ConfigParser()
config.read(args.input_deck)
name = config.get('simulation', 'name')

# initialize the random number generator
from numpy.random import default_rng
rng = default_rng(config.getint('simulation', 'seed', fallback=1234))

# check if output directory already exists
outputdir = os.path.join(config.get(
        'simulation', 'outputdir', fallback='output'), name)
if os.path.exists(outputdir):
    while True:
        action = input(
                'Existing results found: [o]verwrite, [d]elete, [a]bort? ')
        if action == 'o':
            break
        elif action == 'd':
            shutil.rmtree(outputdir)
            os.makedirs(outputdir)
            break
        elif action == 'a':
            import sys
            sys.exit()
else:
    os.makedirs(outputdir)

# copy config file to output directory for posterity
tail = os.path.split(args.input_deck)[1]
try:
    shutil.copy(args.input_deck, os.path.join(outputdir, tail))
except shutil.SameFileError:
    pass

# derived quantities
cells = config.getint('simulation', 'cells')
dt = config.getfloat('simulation', 'dt')
steps = config.getint('simulation', 'steps')
xi = config.getfloat('physical', 'xi')
xf = config.getfloat('physical', 'xf')
nodes = cells + 1  # number of nodes
ti = 0  # s, start time of simulation
tf = dt * steps  # s, end time of simulation
```

```
dt = (tf - ti) / steps  # s, timestep
dx = (xf - xi) / cells  # m, spatial step

# Do a preliminary check on whether the solution will be valid
if config.getboolean('simulation', 'check_validity', fallback=False):
    Te = config.getfloat('simulation', 'nominal_temperature')
    n0 = config.getfloat('simulation', 'nominal_density')
    solution_valid = True
    print('Checking solution criteria')
    v_cfl = dx / dt
    v_th = np.sqrt(k * Te / me)
    print('\tCFL condition')
    print('\t\tdx/dt = %e m/s' % v_cfl)
    print('\t\tv_th = %e m/s' % v_th)
    cfl_met = v_cfl > v_th
    print('\t\tdx/dt > v_th = %s' % cfl_met)
    lambda_D = np.sqrt(epsilon_0 * k * Te / (n0 * e**2))
    print('\tDebye length resolved')
    print('\t\tlambda_D = %e m' % lambda_D)
    print('\t\tdx = %e m' % dx)
    debye_met = dx < (lambda_D / 2)
    print('\t\tdx < lambda_D / 2 = %s' % debye_met)
    lambda_D = np.sqrt(epsilon_0 * k * Te / (n0 * e**2))
    print('\tPlasma frequency resolved')
    f_pe = 2 * pi * np.sqrt(n0 * e**2 / (me * epsilon_0))
    print('\t\tf_pe = %e Hz' % f_pe)
    print('\t\t1/dt = %e Hz' % (1 / dt))
    freq_met = (1/dt) > 2 * f_pe
    print('\t\t1 / dt > 2f_pe = %s' % freq_met)
    if not all((cfl_met, debye_met, freq_met)):
        raise ValueError('Simulation parameters will not resolve plasma behavior.')

# generate the requested particle species
print('Generating requested species')
species = []
for section in config.sections():
    if 'species: ' in section:
        new_species = ParticleList(config, section, rng)
        try:
            N = config.getint(section, 'number')
        except configparser.NoOptionError:
            N = int((xf - xi) * config.getfloat(
                    section, 'density', fallback=0) / new_species.w)
        print('\tSampling %i %ss' % (N, new_species.name))
        new_species.sample_particles(N)
        species.append(new_species)

for s in species:
    particledir = os.path.join(outputdir, 'particles', s.name)
    try:
        os.makedirs(particledir)
```

```
    except IOError:
        # already exists in the case of overwrite
        pass
    s.save_descriptor(outputdir)

print('Discretizing space and time...')
# define spatial grid
spatial_grid = np.linspace(xi, xf, num=nodes)  # m, array of nodes
np.savetxt(os.path.join(outputdir, 'spatial_grid.dat'), spatial_grid)

# define temporal grid
temporal_grid = np.linspace(ti, tf, num=steps)  # s, array of timesteps
np.savetxt(os.path.join(outputdir, 'temporal_grid.dat'), temporal_grid)

# setup LHS of finite difference version of Poisson's equation
# precompute inverse to save on solution time
print('Setting up difference equations...')
A = np.eye(nodes - 2, k=-1) - 2*np.eye(nodes - 2) + np.eye(nodes - 2, k=1)
Ainv = np.linalg.inv(A)

# allocate memory
rho = np.zeros((len(species), nodes))
phi = np.zeros(nodes)
E = np.zeros(nodes)
E_mapped = [None] * len(species)
remainders = [0] * len(species)  # for repopulation
Vi = eval('lambda t:' + config.get('physical', 'Vi'))
Vf = eval('lambda t:' + config.get('physical', 'Vf'))

print('\tInitial field solve')
# map each list of particles to grid and get their charge
for j, s in enumerate(species):
    f = ((s.x - xi) % dx) / dx  # fractions to allocate to left node
    left_rho, _ = np.histogram(
            s.x, bins=spatial_grid, weights=(f * s.q * s.w / dx))
    rho[j, :-1] += left_rho
    right_rho, _= np.histogram(
            s.x, bins=spatial_grid, weights=((1 - f) * s.q * s.w / dx))
    rho[j, 1:] += right_rho
rho[:, 0], rho[:, -1] = 0, 0  # no charge is stored on the walls
total_rho = rho.sum(axis=0)  # get the total charge density
# calculate the potentials using center differencing
b = -dx**2 * total_rho[1:-1] / epsilon_0  # RHS of Poisson's equation
b[0] -= Vi(0)  # correct for applied bias
b[-1] -= Vf(0)
phi[1:-1] = np.dot(Ainv, b)
phi[0], phi[-1] = Vi(0), Vf(0)  # determined by boundary conditions
# calculate the fields in a similar manner
E[0] = -(phi[1] - phi[0]) / dx
E[1:-1] = (phi[:-2] - phi[2:]) / (2 * dx)
E[-1] = -(phi[-1] - phi[-2]) / dx
```

```
# map fields back to particles
for j, s in enumerate(species):
    E_mapped[j] = np.interp(s.x, spatial_grid, E)

# solution loop
print('Stepping through time:')
import time
start = time.time()
before = start
announce_delay = config.getfloat('simulation', 'announce_delay', fallback=1)
grid_diagnostic_step = config.getint('simulation', 'grid_diagnostic_step')
particle_diagnostic_step = config.getint(
        'simulation', 'particle_diagnostic_step')
for i, t in enumerate(temporal_grid):
    # rate limit writing to command line
    now = time.time()
    if now - before > announce_delay:
        announce_step = True
        before = now
    else:
        announce_step = False
    cprint('\tt = %e s' % t, announce_step)

    # pre-sort particle to speed up access
    for j, s in enumerate(species):
        s.sort()

    # run diagnostics if needed
    # insert diagnostics here to get time-aligned values
    if not i % grid_diagnostic_step and not grid_diagnostic_step == 0:
        cprint('\t\tRunning diagnostics', announce_step)
        for j, s in enumerate(species):
            T = np.zeros(nodes)
            u = np.zeros(nodes)
            f = ((s.x - xi) % dx) / dx  # fractions to allocate to left node
            # get particle density from prior charge mapping
            N = rho[j, :] * dx / s.q
            # calculate drift velocity
            uweight = s.w * s.v
            left_u, _ = np.histogram(
                    s.x, bins=spatial_grid, weights=(f * uweight))
            u[:-1] = left_u
            right_u, _= np.histogram(
                    s.x, bins=spatial_grid, weights=((1 - f) * uweight))
            u[1:] += right_u
            u[0], u[-1] = 0, 0
            with np.errstate(invalid='ignore'):
                u = np.nan_to_num(u/N, posinf=0, neginf=0)
            # calculate temperature
            indices = (s.x // dx).astype(int)
            Tweight = s.w * (s.v - u[indices])**2
```

```
        left_T, _ = np.histogram(
                s.x, bins=spatial_grid, weights=(f * Tweight))
        T[:-1] = left_T
        right_T, _= np.histogram(
                s.x, bins=spatial_grid, weights=((1 - f) * Tweight))
        T[1:] += right_T
        T[0], T[-1] = 0, 0
        with np.errstate(invalid='ignore'):
            T = np.nan_to_num(s.m * T / N / k / 2, posinf=0, neginf=0)
        save_grid_data(
                os.path.join(outputdir, 'densities', s.name),
                i, steps, rho[j, :] / s.q)
        save_grid_data(
                os.path.join(outputdir, 'velocities', s.name),
                i, steps, u)
        save_grid_data(
                os.path.join(outputdir, 'temperatures', s.name),
                i, steps, T)
    save_grid_data(os.path.join(outputdir, 'electric_field'), i, steps, E)
    save_grid_data(os.path.join(outputdir, 'electric_potential'), i, steps, phi)
    save_grid_data(os.path.join(outputdir, 'charge'), i, steps, total_rho)
if not i % particle_diagnostic_step and not particle_diagnostic_step == 0:
    for s in species:
        s.save_data(os.path.join(outputdir, 'particles', s.name), i, steps)

# first half-step in velocity for leapfrog method
cprint('\t\tHalf velocity update', announce_step)
for j, s in enumerate(species):
    s.v += (s.q * E_mapped[j] / s.m) * dt/2

cprint('\t\tFull position update', announce_step)
for s in species:
    s.x += s.v * dt

# optionally heat particles
for j, s in enumerate(species):
    if s.heating > 0.0:
        cprint('\t\tHeating %s' % s.name, announce_step)
        s.heat_particles(dt, rng)

# check for particle absorption or reflection
cprint('\t\tChecking particle crossings', announce_step)
for s in species:
    absorbed = np.where((s.x < xi) + (s.x > xf))[0]
    cprint('\t\t\t%i %s exited, ' % (len(absorbed), s.name), announce_step, end='')
    if s.resample:
        cprint('Resampling...', announce_step)
        s.resample_particles(absorbed)
    else:
        cprint('Removing...', announce_step)
        s.remove(absorbed)
```

```
cprint('\t\tInserting new particles', announce_step)
for j, s in enumerate(species):
    remainders[j] += s.repopulate * dt / s.w
    num_new, remainders[j] = divmod(remainders[j], 1)
    num_new = int(num_new)
    cprint('\t\t\t%i computational %ss' % (num_new, s.name), announce_step)
    s.sample_particles(num_new)

if announce_step:
    for s in species:
        print('\t\t%i %ss' % (len(s.x), s.name))

# map each list of particles to grid and get their charge
cprint('\t\tMapping particles to grid', announce_step)
rho[:, :] = 0  # zero out previous results
for j, s in enumerate(species):
    f = ((s.x - xi) % dx) / dx  # fractions to allocate to right node
    left_rho, _ = np.histogram(
            s.x, bins=spatial_grid, weights=(f * s.q * s.w / dx))
    rho[j, :-1] += left_rho
    right_rho, _= np.histogram(
            s.x, bins=spatial_grid, weights=((1 - f) * s.q * s.w / dx))
    rho[j, 1:] += right_rho
rho[:, 0], rho[:, -1] = 0, 0  # no charge is stored on the walls
total_rho = rho.sum(axis=0)  # get the total charge density

# calculate the potentials using center differencing
cprint('\t\tCalculating potentials', announce_step)
b = -dx**2 * total_rho[1:-1] / epsilon_0  # RHS of Poisson's equation
b[0] -= Vi(t)  # correct for applied bias
b[-1] -= Vf(t)
phi[1:-1] = np.dot(Ainv, b)
phi[0], phi[-1] = Vi(t), Vf(t)  # determined by boundary conditions

# calculate the fields in a similar manner
cprint('\t\tCalculating fields', announce_step)
E[0] = -(phi[1] - phi[0]) / dx
E[1:-1] = (phi[:-2] - phi[2:]) / (2 * dx)
E[-1] = -(phi[-1] - phi[-2]) / dx

# map fields back to particles
cprint('\t\tMapping fields to particles', announce_step)
for j, s in enumerate(species):
    E_mapped[j] = np.interp(s.x, spatial_grid, E)

# push particle positions, and finish the velocity update
cprint('\t\tFinish the velocity update', announce_step)
for j, s in enumerate(species):
    s.v += (s.q * E_mapped[j] / s.m) * dt/2
```

```
    elapsed = time.time() - start
    per_step = elapsed / steps
    print('Finished in %e s for an average of %e s per step' % (elapsed, per_step))
```

particlelist.py

```
import os

import numpy as np

e = 1.602e-19  # C, elementary charge
me = 9.109e-31  # kg, electron mass
k = 1.381e-23  # J/K, Boltzmann constant
epsilon_0 = 8.854e-12  # F/m, permittivity of free space


class ParticleList:
    def __init__(self, config, section, rng):
        """
        Initialize a list of particles sharing certain characteristics.

        Keyword arguments:
        config -- configparser object to read from
        section -- section with species information
        """
        self.name = section[9:]
        self.q = config.getfloat(section, 'charge')
        self.m = config.getfloat(section, 'mass')
        self.w = config.getfloat(section, 'weight')
        self.repopulate = config.getfloat(section, 'repopulation_rate', fallback=0)
        self.resample = config.getboolean(section, 'resample', fallback=False)
        self.heating = config.getfloat(section, 'heating_rate', fallback=0)

        xdist = config.get(section, 'spatial_distribution')
        if xdist == 'uniform':
            xi = config.getfloat('physical', 'xi')
            xf = config.getfloat('physical', 'xf')
            def fx(num):
                return rng.uniform(xi, xf, num)
        elif xdist == 'point':
            x0 = config.getfloat(section, 'x0')
            def fx(num):
                return np.ones(num) * x0
        else:
            raise NotImplementedError(
                    '%s is not a valid spatial distribution.' % xdist)
        vdist = config.get(section, 'velocity_distribution')
        if vdist == 'uniform':
            vmin = config.get(section, 'vmin')
            vmax = config.get(section, 'vmax')
            def fv(num):
                return rng.uniform(vmin, vmax, num)
        elif vdist == 'maxwellian':
            vd = config.getfloat(section, 'drift')
            T = config.getfloat(section, 'temperature')
            sigma = (k * T / self.m)**0.5
            def fv(num):
                return rng.normal(vd, sigma, num)
        elif vdist == 'constant':
            v0 = config.getfloat(section, 'v0')
            def fv(num):
                return np.ones(num) * v0
```

```
    else:
        raise NotImplementedError(
                '%s is not a valid velocity distribution.' % vdist)
    self.fx = fx
    self.fv = fv

    # arrays holding individual particle information
    self.x = np.array([])
    self.v = np.array([])

def sort(self):
    ind = np.argsort(self.x) # sort from low to high, reduces memory misses later
    self.x = self.x[ind]
    self.v = self.v[ind]

def add_particle(self, x, v):
    """
    Add individual particles to the list.

    Keyword arguments:
    x -- position (m)
    v -- velocity (m/s)
    w -- weight
    """
    self.x = np.hstack((self.x, x))
    self.v = np.hstack((self.v, v))

def remove(self, indices):
    """
    Remove specificed particles.

    Keyword arguments:
    indices -- iterable of particle indices
    """
    self.x = np.delete(self.x, indices)
    self.v = np.delete(self.v, indices)

def resample_particles(self, indices):
    """
    Resample the position and velocity for given particle indices.

    Keyword arguments:
    indices -- iterable of particle indices
    """
    self.x[indices] = self.fx(len(indices))
    self.v[indices] = self.fv(len(indices))

def sample_particles(self, num):
    """
    Add new particles by sampling their distribution functions.

    Keyword arguments:
    num -- number of particles to sample
    """
    self.x = np.hstack((self.x, self.fx(num)))
    self.v = np.hstack((self.v, self.fv(num)))

def heat_particles(self, dt, rng):
    """
    Heat particles the particles in this list.
```

```
    Keyword arguments:
    energy -- mean energy in joules to add to particles
    """
    energy = self.heating * dt
    dv = rng.normal(0, (2*energy/self.m)**2, len(self.v))
    self.v += dv

def save_descriptor(self, d):
    """
    Save basic particle information.

    Keyword arguments:
    d -- directory
    """
    fn = 'species_%s.txt' % self.name
    with open(os.path.join(d, fn), mode='w') as f:
        f.write('name: %s\nmass: %.5e\ncharge: %.5e\nweight: %.5e'
                % (self.name, self.m, self.q, self.w))

def save_data(self, d, step, steps, filetype='binary'):
    """
    Save particle data in specified format

    Keyword arguments:
    fn -- filename
    filetype -- format to save as
    """
    output = np.vstack((self.x, self.v)).T
    order = int(np.ceil(np.log10(steps)))
    fmt = '%%.0%ii' % order
    fn = os.path.join(d, fmt % step)
    if not os.path.exists(d):
        os.makedirs(d)
    if filetype == 'binary':
        np.save(fn, output)
    elif filetype == 'ascii':
        header = 'x(m),v(m/s)'
        np.savetxt(fn, output, fmt='%.5e', delimiter=',', header=header)
```