Langmuir Project Design

Wei Tian

12/18/2020

The Langmuir project is designed for ...

The Langmuir project file structure can be seen as below,

```
Langmuir Project File Structure
packages/Lib/
        |- Poisson1D_Solver.py  # 1D Poisson solver
        |- Poisson2D_Solver.py  # 2D Poisson solver

packages/Mesh/
        |- Geom2D.py    # 2D geometry generator
        |- Mesh2D.py    # 2D mesh generator

packages/Model/Common
        |- Particle.py  # class for single particle, 3D by default
        |- Multi_Particle.py    # class for multi-particles, supporting vector operati
        |- Particle_Mover.py    # functions to move particle
        |- Field.py     # class for field info and solver

packages/Model/Feature2D
        |- Feature2D_main.py    # 'main' func for feature model
        |- Feature2D_mesh.py    # class for feature mesh, inherited from mesh2d.py
        |- Feature2D_rct.py     # class for reaction
        |- Feature2D_rflct.py   # class for reflection

packages/Model/Sheath2D
        |- Sheath2D_main.py     # 'main' func for sheath model
        |- Sheath2D_coll.py     # class for collision

packages/Model/Reactor2D
        |- Reactor2D_main.py    # 'main' func for reactor model
        |- Reactor2D_transp.py  # solver for plasma transport
        |- Reactor2D_eerg.py    # solver for electron energy equation
        |- Reactor2D_eedf.py    # solver for electron energy distribution
        |- Reactor2D_field.py   # solver for field
        |- Reactor2D_rct.py     # class for reaction
```

```
database/
        |- Species.csv  # database for species info
        |- Cross_Section.csv    # database for species info

database/valid
        |- reaction_xxx.yaml    # validated reaction set for xxx

utest/
        |- Particle_test.py     # test for Particle.py
        |- Multi_Particle_test.py       # test for Multi_Particle.py
        |- Particle_Mover_test.py       # test for Particle_Mover.py

stest/
        |- xxx_test.py  # test for xxx

run/Feature2D
        |- Feature2D_run.py     # prepare input and call main()
        |- mesh.npz     # mesh for feature model
        |- feature.yaml # input for feature model
        |- reaction.csv # reaction set

run/Sheath2D
        |- Sheath2D_run.py      # prepare input and call main()
        |- sheath.yaml  # input for sheath model
        |- collision.py # collision set

doc/
        |- Langmuir_Doc.lyx     # lyx for Langmuir pyhsics
        |- Langmuir_Doc.pdf     # pdf for Langmuir pyhsics
        |- Langmuir_Doc.tex     # LaTex for Langmuir pyhsics
        |- Langmuir_Structure.lyx       # lyx for Langmuir code structure
        |- Langmuir_Structure.pdf       # pdf for Langmuir code structure
        |- Langmuir_Structure.tex       # LaTex for Langmuir code structure
```

This project design will focus on the modules.

The project design is based on the modeling of the physics and cannot go beyond it. In order to well design the project structure, the models need to be analyzed in details.

Let's see the particle-based Monte Carlo models first.

The most frequently called function in a Monte Carlo model is random number generator. In order to repeat the simulation identically, the random number generator should be able to produce identical pseudo-random numbers for each run. The random number generator will be placed in /basics.

```
Function Poisson2d_Solver(input)
        return solution
```

```
                # input:
                # solution:
```

Class PARTICLE() is defined to contain all information of single created particle and simple update and move methods. All particle-based models use Particle() for basic functions. If advanced functions are needed, they are defined within the model, unless sharable.

```
Class PARTICLE():
-- attribute --
        self.name: str
        self.type: str, one of('Eon', 'Ion', 'Neut')
        self.charge: int, unit in Element Charge
        self.mass: float, unit in AMU
        self.posn: float array(3), in (x, z, y), unit in m
        self.vel: float array(3), in (x, z, y), unit in m/s
                # velocity can be also defined as
                # speed + uvec, or erg + angle
        self.isAlive: bool, state of particle
-- method --
        self.update_posn(posn):
                # posn: float array(3), unit in m
        self.update_vel(vel):
                # vel: float array(3), unit in m/s
        self.update_state(state)
                # state: bool, state of particle
        self.move_in_space(dL):
                # dL: float, unit in m, spacestep for a move
                # move in a space, with no field
        self.move_in_time(dt):
                # dt: float, unit in s, timestep for a move
                # move in time, with no field
        self.vel2speed():
                # speed: float, unit in m/s, speed of the particle
                # uvec: float array(3), normalized velocity
                return speed, uvec
        self.vel2erg():
                # erg: float, unit in eV, energy of the particle
                return erg, uvec
        self.vel2ang():
                # ang: float array(2), unit in degree, angle of the velocity
                return ang
```

There are two kinds of attributes: fixed and variable. Fixed attributes, *name, type, charge and mass*, are protected after the particle creation. Variable attributes, *posn, vel and isAlive*, can be updated through predefined methods, *update_posn() update_vel() and update_state()*. These predefined method() should

check the consistency of the input type. PARTICLE() is defined by default in 3D. When it interacts with 2D geometry or mesh, attention should be paid to the dimension match. All 3D dimension is defined with the sequency of (x, z, y) and 2D dimension as (x, z). Moving a particle is the core for particle tracing. Depending on an individual model, moving a particle could require information from field or mesh. It makes sense to move the moving function out of the class PARTICLE() to an independent class or file called PARTICLE_MOVER(), which will be introduced next. Within the class PARTICLE(), only simple move is supported. "Simple move" means no addtional information is required, except dL and dt. In class PARTICLE(), velocity is used for sufficient information. However, for certain applications in the PARTICLE_MOVE() or even in the "simple move", speed or energy, unit vector or angle, are commnly used instead of velocity. To meet these goals, conversion functions are defined in PARTICLE(). More general conversions between velocity, speed and energy, uvec and angle, can be defined in packages/Lib or packages/Model/Common as well.

Class PARTICLE() supports only one sinlge particle, while class MULTI_PARTICLE() supports the creation and track of multi particles at once.

```
Class MULTI_PARTICLE():
-- attribute --
        self.num: int, num of particles
        self.name: str array(N)
        self.type: str array(N), chosen in ('Eon', 'Ion', 'Neut')
        self.charge: int array(N), unit in Element Charge
        self.mass: float array(N), unit in AMU
        self.posn: float array(3, N), in ((x, z, y), N), unit in m
        self.vel: float array(3, N), in ((x, z, y), N), unit in m/s
                # velocity can be also defined as
                # speed + uvec, or erg + angle
        self.isAlive: bool array(N), state of particle
-- method --
        self.update_posn(posn):
                # posn: float array(3, N), unit in m
        self.update_vel(vel, N):
                # vel: float array(3, N), unit in m/s
        self.update_state(state):
                # state: bool array(N), state of particle
        self.move_in_space(dL):
                # dL: float, unit in m, spacestep for a move
                # move in a space, with no field
        self.move_in_time(dt):
                # dt: float, unit in s, timestep for a move
                # move in time, with no field
        self.vel2speed():
                # speed: float array(N), unit in m/s, speed of the particle
                # uvec: float array(3, N), normalized velocity
                return speed, uvec
```

```
        self.vel2erg():
                # erg: float array(N), unit in eV, energy of the particle
                return erg, uvec
        self.vel2ang():
                # ang: float array(2, N), unit in degree, angle of the velocity
                return ang
```

Geometry and mesh generators are defined under packages, since they do not depend any physics or models. There are quite a few commercial or open-source mesh generators available. Within the Langmuir project, the geometry and mesh generator aim only to provide rectangular domain with material assignment and structured mesh. In the future, it will support the import or readin of internally used mesh. The basic 2D geometry and mesh design are shown below,

```
Class GOEM2D():
-- attribute --
        self.name: str, name of the geom
        self.dim: int, dim of the geom, fixed to 2
        self.isCly: bool, symmetry of the geom
        self.domain: float array(4), domain in (left, right, top, bottom)
        self.shape_list: obj array(N), record list of shape
-- method --
        self.add_domain(domain):
                # domain: float array(4), domain in (left, right, top, bottom)
        self.add_shape(shape):
                # shape: obj, defined in SHAPE2D()
        self.inDomain(posn):
                # posn: float array(2), position of tested point
                # isInside: bool, posn is inside the domain or not
                return isInside
        self.get_mater(posn):
                # posn: float array(2), position of tested point
                # mater: str, mater name of the tested point
                return mater

Class SHAPE2D():
-- attribute --
        self.name: str, name of the geom
        self.dim: int, dim of the geom, fixed to 2
        self.domain: float array(), depends on the shape, Rect, Tri, or Circ
        self.mater: str, mater of the shape
-- method --
        self.isInside(posn):
                # posn: float array(2), position of tested point
                # isInside: bool, posn is inside the domain or not
                return isInside
```

```
Class MESH2D():
-- attribute --
        self.name: str, name of the mesh
        self.dim: int, dim of the geom, fixed to 2
        self.domain: float array(4), domain in (left, right, top, bottom)
        self.res: float array(2), unit in m, resolution in structured mesh
        self.ngrid: int array(2), num of nodes in structured mesh
        self.x, self.z: float array(m,n), unit in m, coordinates in axis x, z
        self.mat: int array(m,n), material number, vacuum = 0 by default
        self.matDict: dictionary to map material number to material name
        self.surf: int array(m,n), indicator for surface nodes
                # 0: non-surf node
                # 1: surf node in material
                # -1: surf node in vacuum
-- method --
        self.import_geom(geom):
                # geom: obj, defined in GEOM2D()
        self.gen_mesh(ngrid):
                # ngrid: int array(2), num of nodes in structured mesh
                # generate mesh according to geom
        self.assign_mat():
                # gen matrix containing mater info
                # loop all mesh nodes and get the mater from geom
        self.read_mesh(filename):
                # filename: str, file name for saved mesh
        self.find_surf():
                # assign values to self.surf
        self.save_mesh(filename):
                # filename: str, file name for saved mesh
```

Feature Model needs intensive information for mesh, while Sheath Model and EEDF Model do not.

Feature Model mesh class is designed asThis mesh class can be shared by all mesh-based models, such as Feature Model and Fluid Model.

This mesh class will support read-in function so that it enables restart capability for Sheath Model.

In Feature Model, mesh evolution is the key so that mesh needs more methods. Let's create a FeatMesh class to inheritate generic Mesh class.

```
Class FEATURE2D_MESH(MESH): inheritates all attributes and methods from MESH2D
-- attribute --
        self.surf_norm: float array(2, N) store all surface normal along the surface
-- method --
        self.check_hit(posn):
                # the position of the particle is passed to checkHit()
                # position in continuous space is mapped to meshgrid
```

```
                # position --> index
                # check the material of the index
                # if self.mat[index] != 0, it is a hit
        self.calc_surf_norm(global or local)
                # calculate the surface normal either globally or for a given node
        self.update_mat(index, newMat)
                # change the mat at index to newMat
                # to protect the self.mat, we define a method to explicitly
                # change the material
        self.find_float_cell()
                # find the floating cells which are detached from surface
                # the algorithm is not determined yet
        self.drop_float_cell(imode='Remove' or 'Drop')
                # process floating cells
                # imode='Remove': Remove floating cells, change mat to zero
                # imode='Drop': Drop floating cells to bottom, like deposition
```

checkHit() is placed under FeatMesh() class instead of Particle() because checkHit() is more associated to Mesh than Particle.

self.calcSurfNorm() and self.findFloatCell() can be designed as a interface, where the real functions are called from /basics or /common

There are two classes specific to Feature Model, Reflection and Reaction.

When a hit occurs, the program first check the reaction. If a reaction occurs, the mesh material will be updated; otherwise, the particle is going to reflect.

Reaction class can be seen as blow.

```
Class Reaction()
-- attribute --
        self.index: int array(3), index of hit node
-- method --
        self.readReaction(fname)
                # fname: str, filename for chemical reaction
                # Read in reaction files and
                # create a reaction list with associated probability
        self.getParticle(Particle)
                # get information from Particle
                # name, erg, uvec
        self.getMesh(FeatMesh)
                # get information from FeatMesh
                # mat, matDict, surfNorm
        self.determineReaction(erg, angle, surfNorm)
                # explanied with an example
                # Cl+ + Si_ --> Etch : p1(erg, angle, surfNorm)
                # Cl+ + Si_ --> SiCl_ : p2(erg, angle, surfNorm)
                # Cl+ + Si_ --> SiCl : p3(erg, angle, surfNorm)
```

```
                # Cl+ + Si_ --> Reflect : p4(erg, angle, surfNorm)
                # normalize pi as pi/(p1 + p2 + p3 + p4)
                # roll a dice and determine which reaction to happen
                # return the serial number of the reaction
        self.makeReaction(index)
                # index: int, the serial number of the reaction in reaction list
                # change Particle.isAlive to False
                # call updateMat()
                # if the reaction is a type of "Byproduct": initParticle()
```

Reflection class can be seen as below.

```
Class Reflection()
-- attribute --
        self.index: int array(3), index of hit node
-- method --
        self.getParticle(Particle)
                # get information from Particle
                # name, erg, uvec
        self.getMesh(FeatMesh)
                # get information from FeatMesh
                # mat, matDict, surfNorm        self.reflect()
        self.updateAngle(diffThld, specThld)
                # diffThld: threshold of diffusive reflection
                # specThld: threshold of specular reflection
                # return the anlge or uvec after reflection and
                #       the energy after reflection
                # if erg < diffThld: diffusive reflection
                # if diffThld < erg < specThld: mixed reflection
                #       anlge is determined by the combination of
                #       diffusive and specular reflections
                #       angleMixed =
                #       [(erg - diffThld)*angleSpec + (erg - specThld)*angleDiff]
                #       /(specThld - diffThld)
                # if erg > specThld: specular reflection
        self.updateEnergy(thermThld, thermErg, facErgLoss)
                # thermThld: threshold of falling to thermErg
                # thermErg: thermal energy
                #       thermErg must be less than thermThld
                # facErgLoss: factor of energy loss due to each collision
                # if erg < thermThld: Particle.erg = thermErg
                # if erg > thermThld: Particle.erg = facErgLoss*Particle.erg
        self.postMove(numStep):
                # numStep: int, the number of steps to move after reflection
                # if the reflection angle is close to the surface (angle is
```

```
            # almost 90 degrees w.r.t. surface normal), the particle
            # travel a long distance within a material before going out.
            # the postMove will move the particle by steps without
            # checkHit() until it gets out of the material
```

aaa