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
src/basics/MonteCarlo
                        |- genRanNum.cpp        # Random number generator
                        |- genUniDstrb.cpp      # Generate uniform distribution
                        |- genNormDstrb.cpp     # Generate normal distribution
                        |- genCosDstrb.cpp      # Generate cosine distribution

src/solvers/poisson
                        |- possion1d.cpp        # 1D Poisson solver
                        |- possion2d.cpp        # 2D Poisson solver

src/modules/common
                        |- mesh.cpp     # mesh class with material assigned
                        |- particle.cpp # particle class
                        |- calcSurfNorm.cpp     # Calculate surface normal
                        |- findFloatCell.cpp    # Find float cells

src/modules/feature
                        |- feature.cpp  # 'main' func for feature model
                        |- featureMesh.cpp      # mesh class for feature model
                        |- react.cpp    # reaction class
                        |- reflect.cpp  # reflection class
```

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 genRanNum(seed)
        return ranNumList
```

```
                # seed: int, seed for random number generator
                # ranNumList: a list of generated random numbers
```

Another core module for particle-based models is, as its name states, particle module, which determines the properties and actions of a particle.

```
Class Particle()
-- attribute --
        self.name: str
        self.type: str, one of('Eon', 'Ion', 'Neut')
        self.charge: float, unit in C
        self.mass: float, unit in AMU
        self.position: float array(3), unit in m
        self.speed: float, unit in m/s
        or self.erg: float, unit in eV
        or self.vel: float array(3) unit in m/s
        self.uvec: float array(3), unitless, normalized velocity
        or self.ang: float array(2), theta and phi
        self.accl: float array(3), unit in m/s**2, acceleration
        self.isAlive: bool, state of particle
-- method --
        self.initParticle()
        self.initPosition()
        self.initSpeed(imode)
        self.initEnergy(imode)
                # imode: str, determines the distribution such as
                # Uniform, Normal and Cosine.
        self.move(dL, imode='Space')
                # dL: float, unit in m, spacestep for a move
                # imode: str
                # move in a spacestep, with no field,
                # that is self.accl = (0.0, 0.0, 0.0), used in Feature Model.
        or self.move(dt, EF, imode='Normal')
                # dt: float, unit in s, timestep for a move
                # EF: float array(3), unit in V/m, E-Field
                # move in timestep, with E-field, 1st order accuracy.
                # used in EEDF Model, or Feature Model with E-field on.
        or self.move(dt, EF, imode='leapfrog')
                # dt: float, unit in s, timestep for a move
                # EF: float array(3), unit in V/m, E-Field
                # move in timestep, with time-varying E-field or B-field,
                # 2nd order accurary, able to track the oscillating motion.
        self.checkBndy(domain, imode):
                # domain: float array(6)
                # imode: str, 'Rflective' or 'Periodic'
```

self.move() is the key method for a particle. Depending on the field, best algorithm needs to be chosen for required accurary and speed.

self.move() can be designed as a interface, where the real move function is called from /basics or /common

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

Feature Model mesh class is designed as

```
Class Mesh()
-- attribute --
        self.name: str
        self.domain: float array(6), unit in m, (top, bottom, left, right, front, back
        self.res: float array(3), unit in m, resolution in structured mesh
        self.ngrid: int array(3), num of nodes in structured mesh
        self.x, self.z, self.y: 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.initMesh()
        self.readMesh()
        self.addShape(ishape='Rectangle', 'Triangle' or 'Circle')
        or self.addRectangle(), self.addTriangle() and self.addCircle()
        self.find_surf(): assign values to self.surf
        self.saveMesh()
```

This 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 FeatMesh(Mesh): FeatMesh inheritates all attributes and methods from Mesh
-- attribute --
        self.surfNorm: list of float array(3) store all surface normal along the surfa
-- method --
        self.checkHit(Particle.position)
                # 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.calcSurfNorm(global or local)
                # calculate the surface normal either globally or for a given node
        self.updateMat(index, newMat)
```

```
                # change the mat at index to newMat
                # to protect the self.mat, we define a method to explicitly
                # change the material
        self.findFloatChell()
                # find the floating cells which are detached from surface
                # the algorithm is not determined yet
        self.dropFloatCell(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