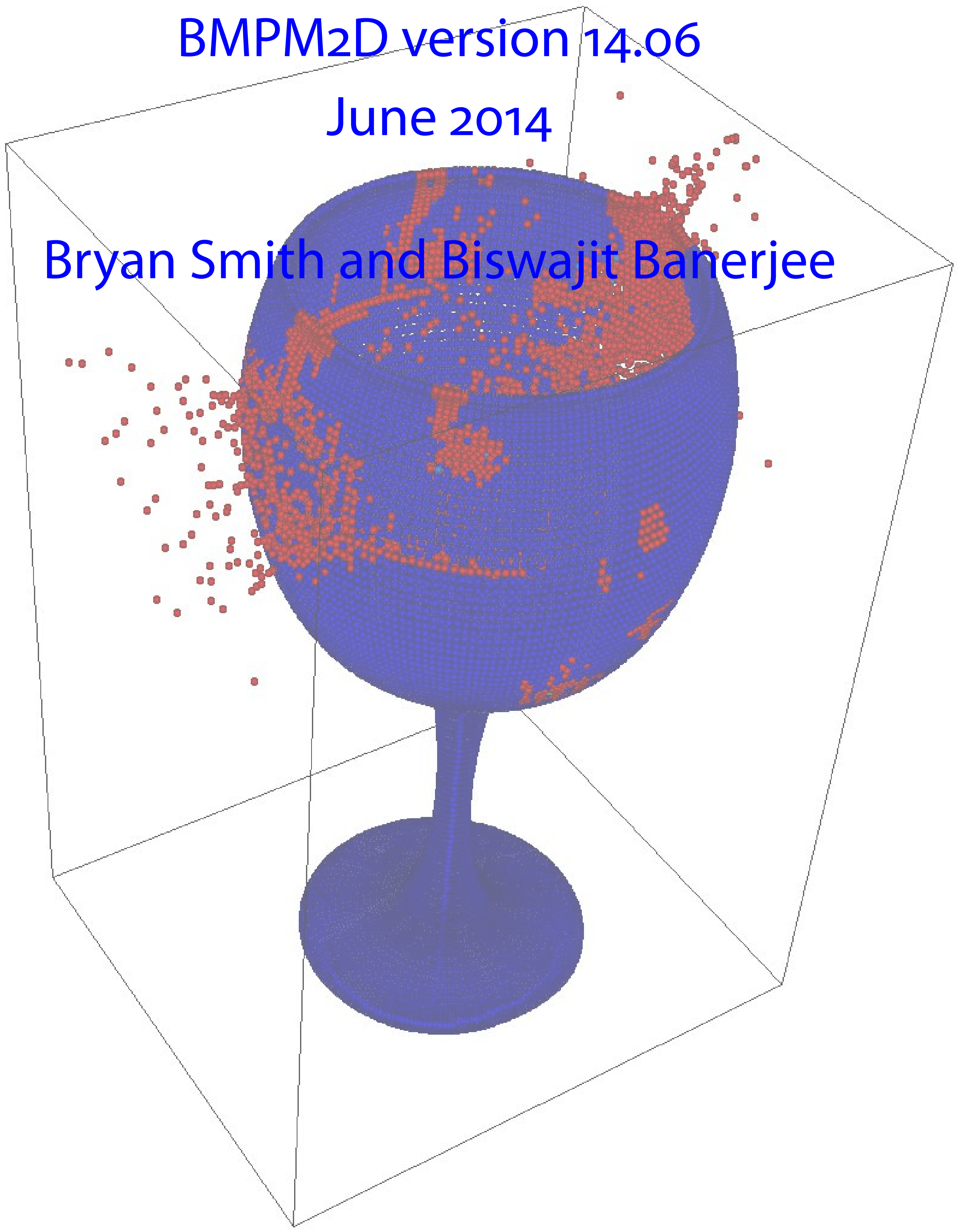
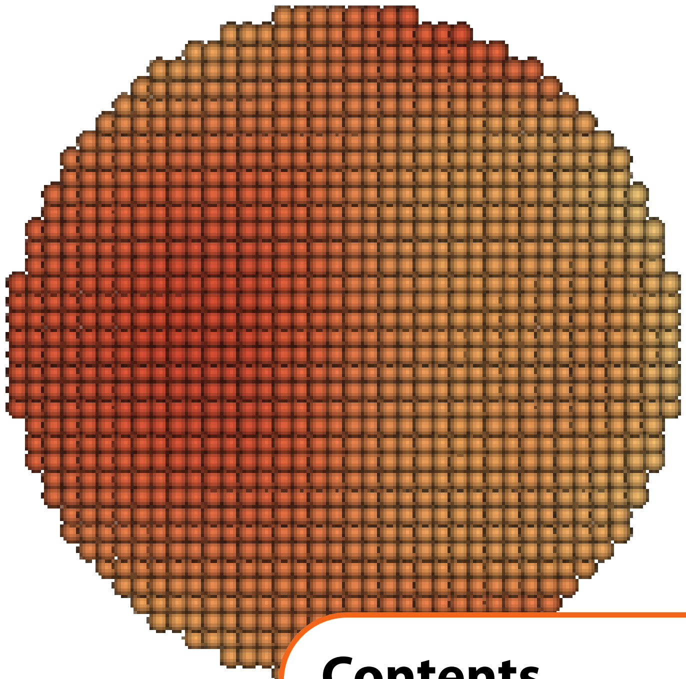
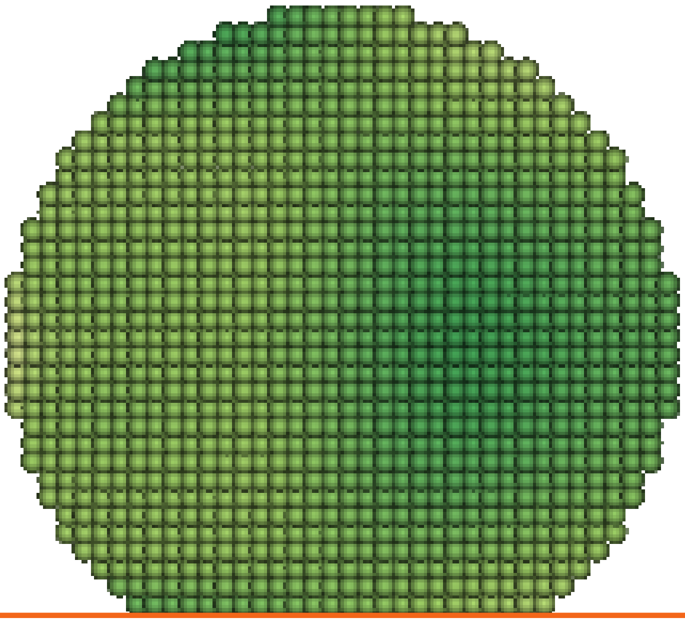
Understanding the BMPM Python Code

BMPM2D version 14.06

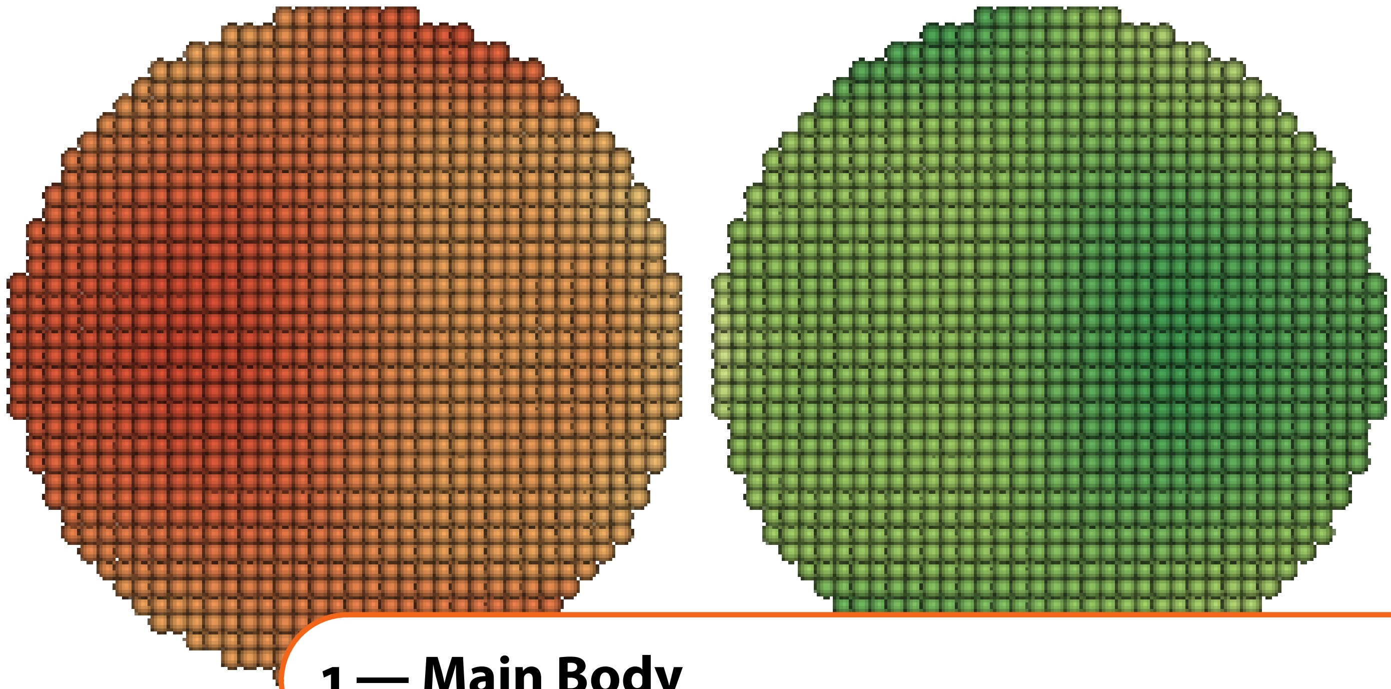
June 2014

Bryan Smith and Biswajit Banerjee



Contents



1 — Main Body

At the first part of the program all the needed packages, modules and classes like numpy(a package which adds support for large, multi-dimensional arrays and matrices), time(a module which provides various functions to manipulate time values) and a couple of written modules (which are gathered in another module named *mpm_imports* for convenience) should be called.
ex_two_contact.py program consists of three parts defined by four functions. The first one, initializes the velocity function. Second one initializes simulation, the third one is for time stepping which cause the movement of your object in your program. And the last one which runs the whole program.

1.1 Importing Modules

Here you have to import all the needed classes modules and packages:

```
import numpy as np
```

By this you import the numpy package with a name which is easy for you to call in your program. It is commom to import it as *np*. Numpy is a fundamental package for scientific computing with Python.

```
import time
```

This module provides various time-related functions.

```
from copy import deepcopy as copy
```

This module provides generic copy operations. This module consist of two types of copy commands: Shallow copy → *copy.copy(x)* and Deep copy → *copy.deepcopy(x)*.
here we have just imported the deep copy. The program will recognize the hard copy command by the name of *copy*

```
from mpm_imports import *
```

mpm_imports is a module which is written to import all the other related classes and modules. Having this module makes it easy to access the required classes. So whenever you want to use a new class or module you need to import it in this module with a suitable name and call the name wherever needed in your program.In order to get access to all the written commands in this module, we put * at the end.

1.2 def initVel(x)

This function defines the initial velocity for the objects in your program.

```
def initVel(x):
    dv = 1.
    if (x[1]+x[0] > 2.):
        dv = -1.
    return dv * 0.1 * np.array([1.,1.])
```

dv determines the direction of movement. dv=1 is for the object moving to the right and dv=-1 is for the object moving in opposite direction.

In this program we have two objects. The points of the first object are located in the left half of your patch domain and the points of the other object are in the other half. As we will explain later on, your patch domain is a 2*2 square so the x and y components of the center point of this domain is equal to one. Then all the points whose x- and y- components confirm with the condition: x[1]+x[0]¿2 are the points from the right half of the domain and have opposite velocity.
The magnitude of the velocity is determined as ”0.1”

1.3 def init(useCython)

```
outputName= 'two'
outputDir = 'test_data/two_contact'
```

Using these two we will make a cirectory and the file name for saving our data. For more information go to **Chapter** 3 → ”datawarehouse”/ ”saveutil”

In this part all the constants of the program have been introduced within 3 sections named as:

1. Domain constants → All the needed information for building the patch domain, the number of cells, the size of cells and the thickness, the initial velocity and so on can be brought here.

   ```
   x0 = np.array([0.0,0.0]);                  # Bottom left corner
   x1 = np.array([2.,2.])                     # Top right corner
   nN = np.array([40,40])                     # Number of cells
   nG = 2                                     # Number of ghost nodes
   thick = 0.1                                # Domain thickness
   ppe = 4                                    # Particles per element
   ```

2. Material Properties → Like the Modulus(Young Modulus), Poisson's ratio and the density of your object gathered in a dictionary named as ”matProps1”

   ```
   E1 = 1.0e3;    nu1 = 0.3;    rho1 = 1.0e3;
   vWave1 = np.sqrt( E1/rho1 )
   matProps1 = {'modulus':E1, 'poisson':nu1, 'density':rho1 }
   matModelName = 'planeStrainNeoHookean'
   ```

3. Time Constants → like the initial and final time, time interval and so on can be determined here.

   ```
   t0 = 0.0                                        # Initial Time
   CFL = 0.4                                       # CFL Condition
   dt = min((x1-x0)/nN) * CFL / vWave1;            # Time interval
   tf = 10.                                        # Final time
   outputInterval = 0.05                           # Output interval
   ```

Now having all the above information we can build up three important objects in order to get access to information and functions of each impotant classes:

1. `dw=Dw(outputDir,outpuName,outputInterval)`

   By this we would be able to initialize arrays and then add and save our particles in each iteration of time

2. `patch=Patch(x0,x1,nN,nG,t0,tf,dt,thick)`

   Using this object we prepare the ground(patch) in which our particles would be created and move. Moreover, we create the grids with which the interpolation between particle and grids happens.

3. `shape=Shape(useCython)`

   using this object we define shape function/its derivative/the number of supporting nodes and finally update the node contributions.

Having prepared the ground now we can create our object(s) or material(s).

mats is a list of your objects or materials based on their material ID (*dwis=[1,2]*). Here since we have two balls, we have added these two balls' properties to this list considering their material IDs. These material IDs make difference between the points of the two balls. So the points of the balls will never get mixed.

```
mats = []
dwis = [1,2]
mats.append(Material( matProps1, matModelName, dwis[0], shape, useCython ))
mats.append(Material( matProps1, matModelName, dwis[1], shape, useCython ))
```

Moreover, for each one of the objects we create a grid:

```
dw.createGrid(dwis[0],patch)
dw.createGrid(dwis[1],patch)
```

This happens in "datawarehouse" → **Chapter** 3

Knowing a couple of information of your object (for example the centres of circles and their radii) you can create your own object using a module named "geomutils":

```
center1 = np.array([0.75,0.75])
center2 = np.array([1.25,1.25])
radii = np.array([0.0,0.2])
density = matProps1['density']
px1, vol1 = geomutils.fillAnnulus( center1, radii, ppe, patch )
px2, vol2 = geomutils.fillAnnulus( center2, radii, ppe, patch )
```

In *ex_two_contact* we need to create two circles. So defining two centres(center1 and center2), their radii and the density we make two circles by calling "geomutils.fillAnnulus". The position and volume of these particles would be saved as *px1*, *vol1* for the first object and *px2*, *vol2* for the second object . (→ **Chapter** 10) We add these information about the particles to the datawarehouse → **Chapter** 3 :

```
dw.addParticles( dwis[0], px1, vol1, density, shape.nSupport )
dw.addParticles( dwis[1], px2, vol2, density, shape.nSupport )
```

Here we initialize the *contacts* matrix:

```
contacts = []
contacts.append( Contact(dwis) )
```

Now, we have the points (particles of our objects) and their properties sorted and saved in px(for position), pm(for mass) and pVol(for volume) in "datawarehouse". We have also initialized the needed variables (like: momentum, velocity increment, position increment, external forces and so on) to zero matrices so in the continue you would be able to add to them. → dw.addParticles (→ **Chapter** 3).For each particle now we need to find the node contributions. The contribution of each particle to the grid node can be evaluated by calculating the nodal shape functions (S) and getting the weighting functions and other needed data → mpm.updateMats (→ **Chapter** 11) :

```
mpm.updateMats( dw, patch, mats )
```

Knowing the initial velocity and the mass of the particles we can find their initial momentums and save them in (pw). → matLis[ ].setVelocity (→ **Chapter** 5):

```
mats[0].setVelocity( dw,  initVel )
mats[1].setVelocity( dw,  initVel )
```

At the end, running this function prints the ”dt” as the time increment and returns all the saved information in datawarehouse, patch, mats and contacts for furthure inquiry:

```
print 'dt = ' + str(patch.dt)
return (dw, patch, mats, contacts )
```

1.4 def stepTime

Here we keep moving our objects (particles) while they are still in the patch domain and before the end of the time (t ¡ tf). And in every each time step we save all the data of particles (position and velocity) and grid nodes (position and acceleration) in order to export your points with the variables (velocity, acceleration, material ID and a function of shape function(S)) at each point via the VTK.

```
def stepTime( dw, patch, mats, contacts ):
    # Advance through time
    tbegin = time.time()  //the time of starting iteration
    mpmData = dict()  //make a dictionary for saving data
    try:
        while( (patch.t < patch.tf) and patch.allInPatch(dw.get('
           px',1)) ):
            mpm."timeAdvance"( dw, patch, mats, contacts )
            if "dw.checkSave"(patch.dt): mpmData[dw.t] = copy(dw)
            "dw.saveData"( patch.dt, mats )
    except JacobianError:
        print 'Negative Jacobian'

    tend = time.time()  //the time of finishing iteration
    print (str(dw.idx) + ' iterations in: ' + readTime(tend-
       tbegin)
            + ' t=' + str(patch.t) )

    return mpmData
```

mpm.timeAdvance

At each time step, the program runs the ”mpm2d” module. So along side with ”Material” class the particles will move by integrating the particles data to the grid, finding the acceleration and new velocity and interpolating back the grid data on the particles. And at last we increase the time by (dt).

Check for Save & save

Here we check if the time is proper for saving the data at the exact time. If it is ”True” so the data will be coppied to the `mpm.Data` dictionary and saved in a VTK file.

```
if "dw.checkSave"(patch.dt): mpmData[dw.t] = copy(dw)
            dw.saveData( patch.dt, mats )
```

When the time is finished or the points go out of the patch domain the program exists the "while" loop and the ending time will be saved as `tend`.
As the return, this function gives the data gathered in `mpmData` and saved as VTK files.

1.5 def run

Here you first get your initial information from "init" function:

```
dw, patch, mats, contacts = init( output, useCython )
```

Then you update and save all the new data at each time step:

```
mpmData = stepTime( dw, patch, mats, contacts )
```

As the export data you will get the "mpmData".

In order to run this program, you first need to go to the directory where this program is saved and import it in **ipython**. Then type: *programname**.run()***:

```
>>> ipython
  >>> import ex_two_contact
  >>> data= ex_two_contact.run()
```

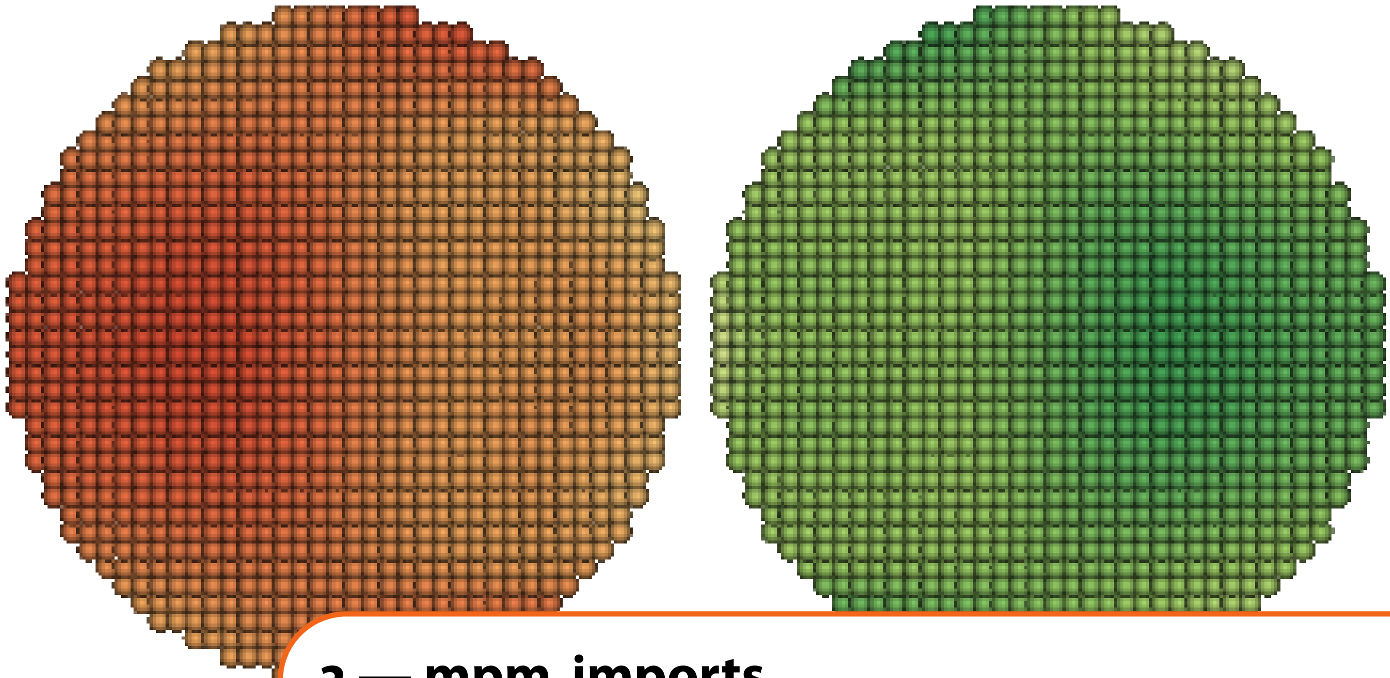
All the data will be saved in a directory named: "test_data/two_contact". In order to visualise this program, go to this new directory and list all the data:

```
>>> ~/.../test_data/two_contact$ ls
```

and open **visit** program in the same path.

```
>>> ~/.../test_data/two_contact$ visit
```

Now all the data is gathered there and you just need to open them and by adjusting color and other adjustments and pushing the *play* key you can see your program runs.



2 — mpm_imports

1.
   ```
   import sys
   sys.path.append('../')
   ```
   By this command you add ../ before your directory path which refers to filename in parent directory.

2.
   ```
   import numpy as np
   ```
   NumPy is an extension to the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large library of high-level mathematical functions to operate on these arrays. It has been imported to the whole program with the name of "np".

3.
   ```
   from src.datawarehouse import DataWarehouse as Dw
   ```
   Datawarehouse is a class in which you mainly hold all the data regarding your particles and grid nodes and at last save all the data as input data in VTK file format so therefore be visualized using an appropriate visualization packages like *visIt*. (→ **Chapter** 3)

4.
   ```
   from src.patch import Patch
   ```
   Patch class mainly creates the domain in which our objects can be created and move around. In here you make sure that as long as your points are in the patch domain, your program runs. (→ **Chapter** 4)

5.
   ```
   from src.material import Material
   ```
   ```
   from src.material import JacobianError
   ```
   "Materal" class is a place for bringing particles with the contributions of nodes (as weighting function) along with all the particle properties (such as mass, momentum and so on). After finding the

internal and external forces we interpolate all the point properties to the grid nodes. So we can compute the velocity and acceleration of the grid nodes. After interpolating again the grid nodes' properties to the particle points we would be able to find the position and velocity increments of the particles as well as the deformation. So at the end we can update the particles' position and momentum and the deformation for the next iteration. (→ **Chapter** 5)

6. `from src.simplecontact import FreeContact as Contact`

   simplecontact class is where you can check when the contact between your objects has occurred. (→ **Chapter** 6)

7. `from src.boundcond import BoundaryCondition as Bc`

   Here we set boundary condition.

8. `from src.mpmutils import readableTime as readTime`

   ”mpmutils” basically helps in moving the data between particles and the grid. (→ **Chapter** 8)

9. `from src.shape2 import GIMP as Shape`

   shape2 first initializes the shape functions and their derivatives and then by calling another module (: **gimp**) it updates the shape functions and their derivatives so consequently the weighting function and the gradient of the shape function can be achieved. By these weighting functions and their gradients we can find the contribution of each particle to the grid nodes in *mpmutils*. (→ **Chapter** 9 & **Chapter** 8)

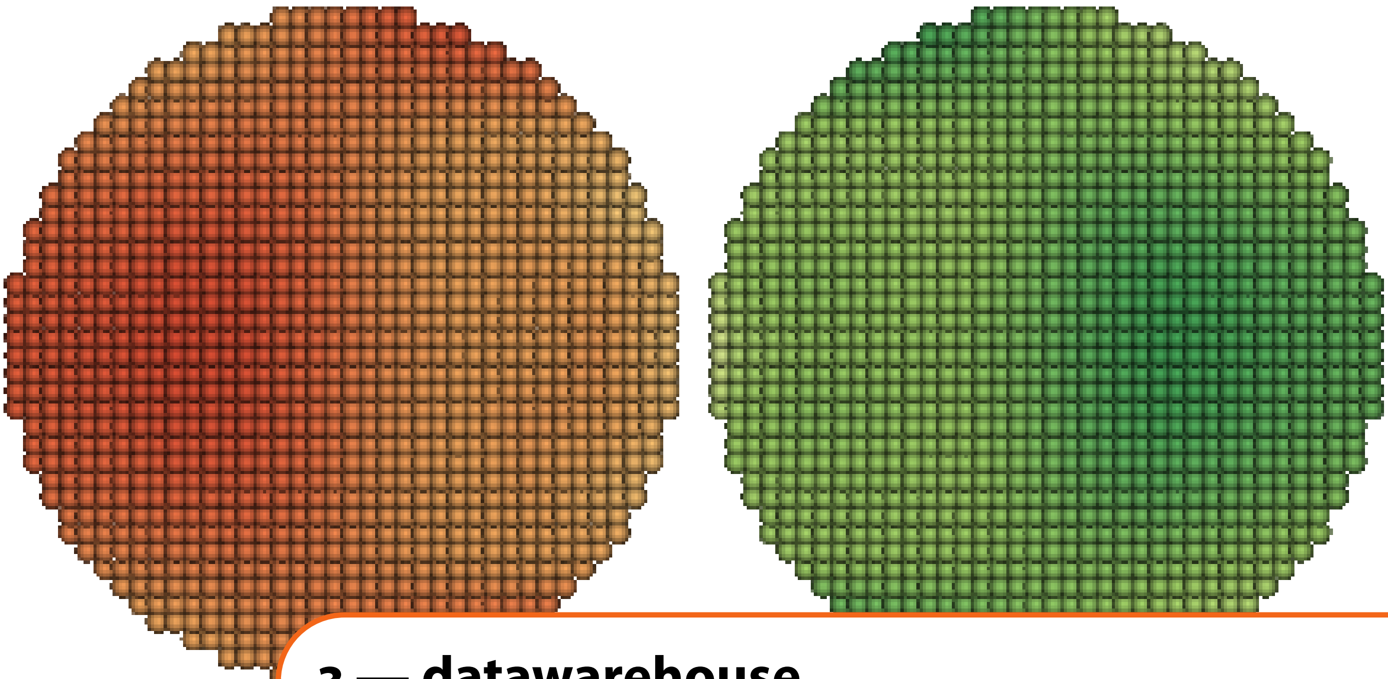
10. `from src import geomutils`

    Here we basically build our objects. (→ **Chapter** 10)

11. `import src.mpm2d as mpm`

    mpm2d module alongside with the *Material* class and *mpmutils* gives us the materials' movements in each time step. (→ **Chapter** 11 & **Chapter** 5 & **Chapter** 8)

12.
```
try:
    from src.shape2_c import GIMP as Shape_c
except Exception:
    Shape_c = Shape
```



3 — datawarehouse

In this class you mainly hold all the data regarding your particles and grid nodes and at last save all the data as input data for VTK to visualize your particles.

First you have to import all the needed packages, classes and modules:

```
import numpy as np
import collections
from itertools import izip
import os
from saveutil import SaveUtil
from evtk.hl import pointsToVTK
from copy import deepcopy as copy
```

3.1 __init__

This __init__ function is known as constructor which means all the methods defined in this function will be automatically run as soon as an object assigned to this class.
Note that in order for other methods and functions in the class to run we have to call them in the program after creating the object: (object. name of the method).

Here we can define and initialize some variables and determine the type of them.

```
def __init__(self, ddir, fname, dt, idx=0, t=0. ):
"self.dw = dict()"
self.out_idx = idx                    # Index for output file
self.idx = idx                            # Iteration index
self.t = t                                # Initial time
self.dt = dt
self.fname = ddir + '/' + fname           # Output file name
self.nzeros = 4                   # Number of digits in filename
self.saveUtil = "SaveUtil"(dt,self.fname)  # Saving Utility Class

  "try:  os.mkdir( ddir )"
  "except Exception:  pass"
```

`self.dw = dict()` make a dictionary type of data. Using dictionary data type is more convenient and much faster than using lists.

The `SaveUtil` class will be explained in the next section.

`self.fname = ddir + '/' + fname` gives us a file directory + the file name in which our data is going to be saved. (→ exp: `test_data/two_contact/two` which `test_data/two_contact/` is the file directory and `two` is the file name.)

os.mkdir

```
try:  os.mkdir( ddir )
except Exception:  pass
```

By this we make a directory in which we are going to save our points and nodes data.

3.2 def saveData & def checkSave

Here we first save all the data at each desirable time step so we can export them by VTK and then increase time by (dt).

```
def saveData( self, dt, matlist ):
  if self."checkSave"( dt ):
      self.out_idx = self."saveUtil.saveData"( self.out_idx,
         matlist, self )
  self.t += dt
  self.idx += 1
```

def checkSave

This function checks for the time step at which we like to save the data. Without this fuction all the data would be saved which is too much specially for very small time steps. So by this we narrow down the number of savings:

```
def checkSave( self, dt ):
  dr = self.t/self.dt
  dt0 = self.dt * min( dr-np.floor(dr), np.ceil(dr)-dr )
  return dt0 < dt/2.
```

3.2.1 SaveUtil

`SaveUtil` is a class defined for saving the data in VTK format.

```
from evtk.hl import pointsToVTK
```

We need to import a function named: "`pointsToVTK`" from evtk.hl in order to save all the points data as files in VTK format.

def __init__

```
def __init__(self, dt, fname):
  self.dt = dt                                   # Output interval
  self.fname = fname                             # Output file name
  self.nzeros = 4                              # Number of digits in
     filename
```

This function gets `dt` and file name (`fname`) as the inputs and initialize the ouptput interval, output directory/filename, and the number of digits in the filename.

def saveData(self, idx, matlist, dw)

By getting the index for output (`idx`), material IDS (`matlist`) and the information in `dw`, this function first save the data existing in `dw` and then increase the ouput-index by one:

```
def saveData( self, idx, matlist, dw ):
  self.save( matlist, idx, dw )
  return idx + 1
```

def save(self, matlist, idx, dw)

In this function first we define the file-name in which the following data is going to be saved:

```
def save( self, matlist, idx, dw ):
  fName = self.fname + str(idx).zfill(self.nzeros)
```

str(idx).zfill(self.nzeros) pads string: (`str(idx)`=0 for the first iteration) on the left with zeros to fill width: (`self.nzeros` which is equal to 4 in this program).

```
for idx=0 --> str(idx).zfill(self.nzeros) = 0000
for idx=1 --> str(idx).zfill(self.nzeros) = 0001
for idx=2 --> str(idx).zfill(self.nzeros) = 0002
for idx=3 --> str(idx).zfill(self.nzeros) = 0003
.
.
.
```

So at each iteration we save the points data in this directory with its own file name. For example, at the first iteration the "idx" is "0" so the file name made by the `fName` variable is: `test_data/two_contact/two0000`. For the next iteration the "idx" is "1" and the file name is: `test_data/two_contact/two0001` and so on.

save as VTK file format

Now that we have made the places for points and nodes to be saved , we need to save the data itself in a way that can be used and visualised with common packages like visIt and so on. To do so we go through the following steps but first let's explain how we generate VTK files with Python because it is where we save the particles in file(s) in order to visualise them by a visualization package later on. EVTK (Export VTK) package allows exporting data to binary VTK files for visualization and data analysis with any of the visualization packages that support VTK files, e.g. VisIt. So at the beginning of "saveutil" we imported pointsToVTK from evtk.hl so therefore we can use it here to export particle simulation data.
Here we need to define all the x, y and z components of the points and and scalar variables (like velocity and materialID) at each point:

```
x = [];    y = [];    v = [];    ms = [];
  matid = []

  for mat in matlist:
      dwi = mat.dwi
      px,pv,pVS,pVol = dw.getMult( ['px','pxI','pVS','pVol'], dwi
          ) //--> gets the data from datawarehouse
      nn = len(pVol)
      matid += [dwi]*nn
```

```
        x += list(px[:,0])

        y += list(px[:,1])

        vx = pv[:,0]
        vy = pv[:,1]
        v += list( np.sqrt( vx*vx + vy*vy ) * np.sign(vx) )
        //signe(vx) gives the direction of motion to the balls.
           They are moving towards each other so one is moving in
           the opposite direction to the other one.
        pS = [pVS[ii]/pVol[ii] for ii in range(nn)]
        ms += [vonMises(pp) for pp in pS]

    x = np.array(x)
    y = np.array(y)
    z = np.zeros(x.shape)
    v = np.array(v)
    ms = np.array(ms)
    matid = np.array(matid)
```

*vonMises is function defined at the very first part of the ”saveutil” class:

```
def vonMises( S ):
    return np.sqrt( S[0,0]*S[0,0] - S[0,0]*S[1,1] + S[1,1]*S[1,1]
        +
                    3.*S[1,0]*S[0,1] )
```

So we can export a set of points at each time iteration as a VTK-XML file format. All the data points at each iteration are saved in a single file with the extension of (.vtu) → (UnstructuredGrid (.vtu) fh Serial vtkUnstructuredGrid (unstructured).)

```
vsdat = {"vonMises":ms, "v":v, "mat":matid}        //--> it is a
   dictionary for keeping the scalar variables for evtk.
pointsToVTK(fName, x, y, z, data = vsdat)
```

3.3 def addParticles(self, dwi, pX, pVol, density, shSize):

With this function we can import all the data regarding our particles and save them in ”datawarehous” (like Px as particle position, pm as particle mass, and so on). More over here we initialize other values which we need throughout the program (like pw as particle momentum, etc.)

```
 def addParticles( self, dwi, pX, pVol, density, shSize ):
  npt = len(pX)
  labels = ['pw','pvI','pxI','pfe','pGv','pVS']
  shapes = [(npt,2),(npt,2),(npt,2),(npt,2),(npt,2,2),(npt,2,2)]

  // Add initial position, position, volume, mass, and node
     contributions
  self."add"( 'pX',    dwi, pX )
  self."add"( 'px',    dwi, pX )
  self."add"( 'pVol',  dwi, pVol*np.ones(npt) )
  self."add"( 'pm',    dwi, pVol*density*np.ones((npt,2)) )
  self."add"( 'cIdx',  dwi, np.zeros((npt,shSize),dtype=np.int) )
  self."add"( 'cW',    dwi, np.zeros((npt,shSize)) )
  self."add"( 'cGrad', dwi, np.zeros((npt,shSize,2)) )

  // Add variables contained in "labels"
  for (lbl,shp) in izip( labels, shapes ):          //"izip"
     couples each label with its 'shapes' value
      self."add"( lbl, dwi, np.zeros(shp) )

  // Create and add initial deformation (identity matrix)
  pF = np.zeros((npt,2,2))
  for pFi in pF:
      pFi += np.eye(2)
  self."add"( 'pF', dwi, pF )
```

By getting some information like material ID `dwi`, particle position `pX`, particle volume `pVol`, Particle density `density` and number of supporting nodes `shSize` we can make dictionary data type of each one of the above variables using the `add` function.

3.4 def add(self, label, dwi, val)

Here we first look for the variable with its label. If any dictionary data type exists with the same label (by checking the length of the out-put of the `get` function), we will add the new data to the older one (using the `append` function). But if there was no such a data in datawarehouse, we will initialize it using the `init` function:

```
def add( self, label, dwi, val ):
  if len(self."get"(label,dwi)): self."append"( label, dwi, val )
  else: self."init"( label, dwi, val )
```

3.5 def get(self, label, dwi):

This function basically check for the data by its label. If there was a dictionary data type with the same label, it returns the data. But if there was not such a data it returns an empty list:

```
 def get( self, label, dwi ):
  try:
      return self.dw[label,dwi]
  except Exception:
      return []
```

3.6 def append(self, label, dwi, val):

This function basically add the new data to the older one as a new row:

```
 def append( self, label, dwi, val ):
  self.dw[label,dwi] = np.append( self.dw[label,dwi], "toArray(
     val)", axis=0 )
```

3.7 def init(self, label, dwi, val):

This function basically initialize the new data to its value and make a dictionary out of it:

```
 def init( self , label , dwi , val ):
  self.dw[label,dwi] = "toArray(val)"
```

3.8 def toArray(val):

This function makes sure that the type of the value is an "array":

```
 def toArray( val ):
    if type(val) is np.ndarray:
  return val
    else:
  return np.array(val)
```

3.9 def getMult(self, labels, dwi):

This function makes a list of any input varible (as labels):

```
  def getMult( self , labels , dwi ):
  output = []
  for label in labels:
      output.append( self.get( label , dwi ) )
  return output
```

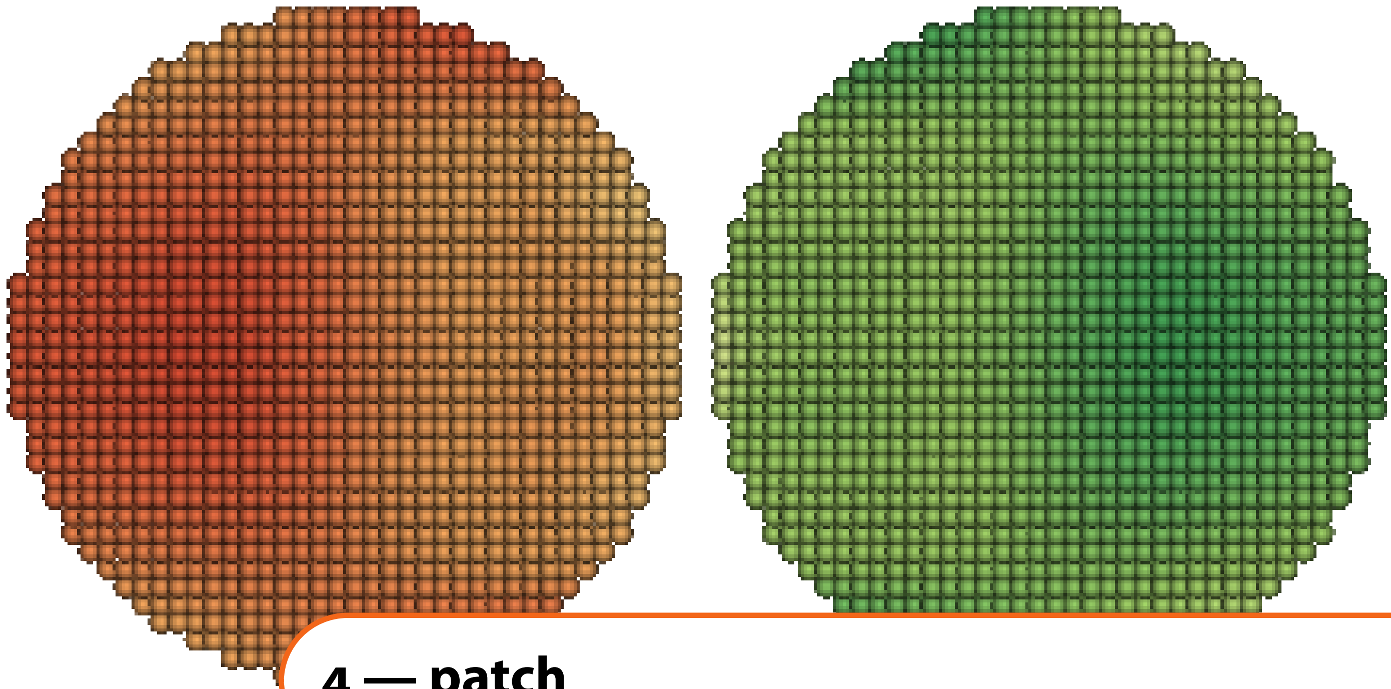
3.10 def createGrid & def zeroGrid

So far we have added and saved the particle information. In order to add the grid information we define two more functions:

```
  def createGrid( self , dwi , patch ):
  gx = patch.initGrid()
  self.dw['gx',dwi] = toArray(gx)
  self."zeroGrid"(dwi)
```

With `createGrid` we get the grid position which is created in "patch" class (**Chapter** 8) and add it to the "datawarehouse" as a dictionary data type.
`zeroGrid` function on the other hand, initialize all the other grid information to zero.

```
def zeroGrid( self , dwi ):
  gx = self.get('gx',dwi)
  labels = ['gm','gv','gw','ga','gfe','gfi','gGm']
  for label in labels:
      self.init( label , dwi , np.zeros(gx.shape) )
```



4 — patch

This class is mainly a computational domain which create the domain in which our objects can be created and move around.

4.1 __init__

We define the domain variables here. All the needed variables are sorted here. The domain is created with two points (X0 and X1). The value for these points are given in the ”Domain Constants” of the main program (ex_two_contact.py).

```
def __init__(self,X0,X1,Nc,nGhost,t0,tf,dt,th):
        dim = 2
        self.X0 = X0                     # Bottom corner of patch
           domain
        self.X1 = X1                     # Top corner of patch domain
        self.Nc = Nc+1+2*nGhost          # Vector of node counts
        self.thick = th                  # Thickness
        self.nGhost = nGhost             # Number of Ghost nodes
        self.dX = (X1-X0)/(Nc)           # Cell size
        self.t = t0                      # Time
        self.tf = tf                     # Final time
        self.dt = dt                     # Time increment
        self.it = 0                      # Timestep
        self.tol = 1.e-15                # Global tolerance
        self.bcs = []
```

4.2 initGrid(self)

Here we initialize the position of grid nodes. This will be done by a function: ” initGrid ”. This function gives us the x-component and y-component of the grid nodes within the domain and adds the nodes to node positions(”gx” matrix) in ”datawarehouse” class.

```
def initGrid(self):
        dg = self.nGhost*self.dX
        x = np.linspace( self.X0[0]-dg[0], self.X1[0]+dg[0], self
           .Nc[0] )
        y = np.linspace( self.X0[1]-dg[1], self.X1[1]+dg[1], self
           .Nc[1] )
        xx, yy = np.meshgrid( x, y )
        gx = np.append(xx.reshape(xx.size,1), yy.reshape(yy.size
           ,1), axis=1)

        return gx
```

4.3 allInpatch & inpatch

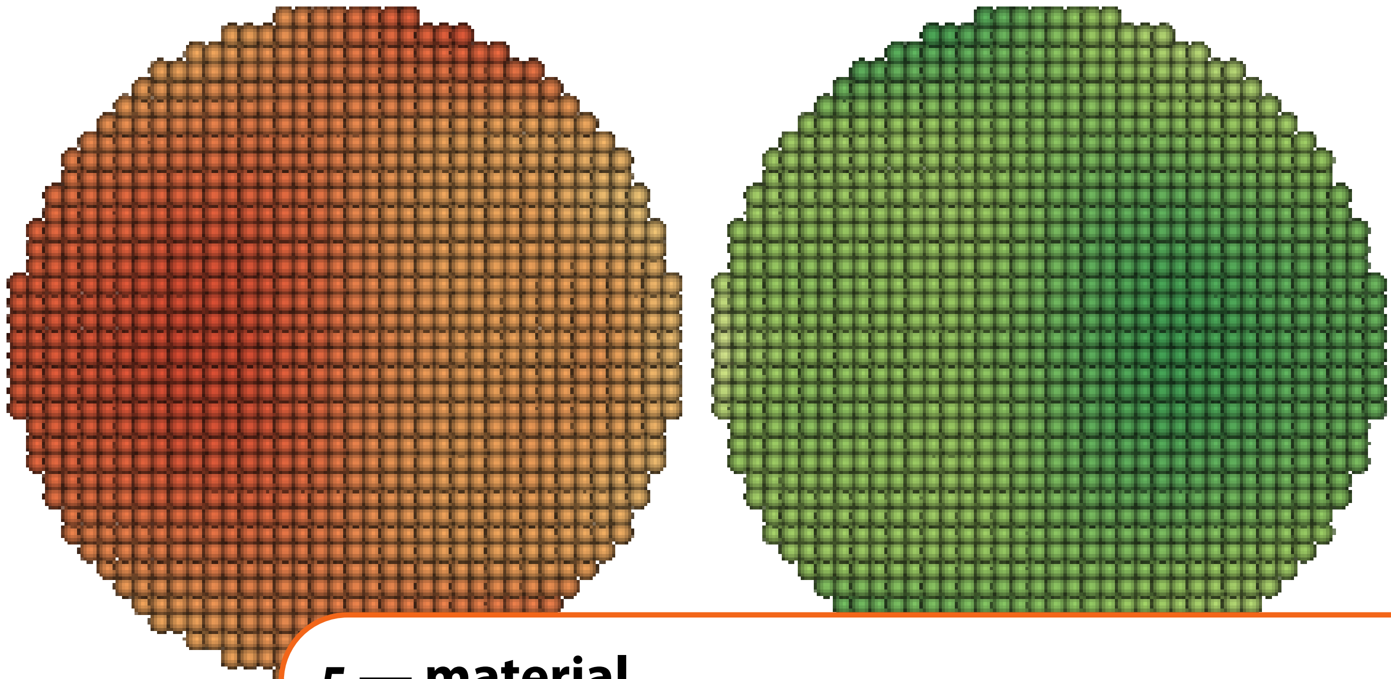
Two functions: `allInPatch` and `inPatch` make sure that the program runs as long as we have points in the patch domain:

```
def allInPatch( self, pts ):
        for pt in pts:
            if not self.inPatch( pt ):
                return False
        return True

 def inPatch( self, pt ):
        if (pt[0] < self.X0[0]) or (pt[1] <self.X0[1]):
            return False
        if (pt[0] > self.X1[0]) or (pt[1] >self.X1[1]):
            return False
        return True
```

4.4 stepTime

This function increases time by a time increment (`dt`) and the Timestep ID by 1:

```
def stepTime( self ):
        self.t += self.dt
        self.it += 1
```



5 — material

”Materal” class is a place for bringing particles with the contributions of nodes (as weighting function) along with all the particle properties (such as mass, momentum and so on). Here we set the External and Internal forces for particles. Using ”mpmutils” module we send all the particle data to the grid. After that we can calculate the grid velocity and acceleration. At last we can find the velocity and position increment for particles by interpolation the grid data to particles.

5.1 __init__

All the initial data in this part would be set as soon as we make an object from this class:

```
def __init__(self, props, model, dwi, shape, useCython=True):
        self.props = props
        self.dwi = dwi
        self.shape = shape

        try:
            self.ignoreNegJ = props['ignoreNegJ']
        except Exception:
            self.ignoreNegJ = False

        if useCython:
            self.util = util_c
            self.mmodel = mmodel_c
        else:
            self.util = util
            self.mmodel = mmodel

        self.mm = self.mmodel.MaterialModel( model, props )
```

5.2 def updateContributions(self, dw, patch):

This function take the particles and using the **shape2/gimp2** classe updates their contributions to nodes.

```
  def updateContributions( self, dw, patch ):
        dw.zeroGrid( self.dwi )
        self.shape.updateContribList( dw, patch, self.dwi )
```

5.3 def setVelocity(self, dw, v):

This function brings particles' mass and along with their velocity it finds the momentum.

```
def setVelocity( self, dw, v ):
        pw,pm,px = dw.getMult( ['pw','pm','px'], self.dwi )

        for (ii,pxi,pmi) in izip(count(),px,pm):
            if isfunction(v):                  //--> 'isfunction'
               checks whether the object is a function or not.
                pw[ii] = v(pxi) * pmi
            else:
                pw[ii] = v * pmi
```

5.4 def setExternalLoad, def setExternalAcceleration & def applyExternalLoads

First [setExternalLoad] sets the external force (if there is any) as "pfe" and by using `applyExternalLoads` this external force will be moved to the grids: "gfe"

```
def setExternalLoad( self, dw, fe ):
        pfe = dw.get( 'pfe', self.dwi )
        for pfei in pfe:
            pfei = fe


    def setExternalAcceleration( self, dw, acc ):
        pfe,pm = dw.getMult( ['pfe','pm'], self.dwi )
        pfe = acc * pm


    def applyExternalLoads( self, dw, patch ):
        # Apply external loads to each material
        cIdx,cW = dw.getMult( ['cIdx','cW'], self.dwi )

        pp = dw.get( 'pfe', self.dwi )                        #
           External force
        gg = dw.get( 'gfe', self.dwi )
        self.util.integrate( cIdx, cW, pp, gg )
```

5.5 interpolateParticlesToGrid(self, dw, patch):

This function brings the mass and momentum arrays of the particles and integrates these particles values to the grid (pm-¿gm and pw-¿gw) using the weighting functions (`cW`):

```
def interpolateParticlesToGrid( self, dw, patch ):
        # Interpolate particle mass and momentum to the grid
        cIdx,cW = dw.getMult( ['cIdx','cW'], self.dwi )

        pp = dw.get( 'pm', self.dwi )                              #
           Mass
        gg = dw.get( 'gm', self.dwi)
        self.util.integrate( cIdx, cW, pp, gg )

        pp = dw.get( 'pw', self.dwi )                              #
           Momentum
        gg = dw.get( 'gw', self.dwi )
        self.util.integrate( cIdx, cW, pp, gg )
```

Note that all the movements from particles to grid and vice versa happen in "mpmutils" module (→ (**Chapter** 8)).

5.6 computeStressTenso & computeInternalForce

First we get the stress tensor from **MaterialModel** class and along with the volume tensor we find "stress * deformed volume" as pvs.

```
def computeStressTensor( self, dw, patch ):
        pf  = dw.get( 'pF', self.dwi )                    #
           Deformation Gradient
        pvs = dw.get( 'pVS', self.dwi )                   # Volume *
           Stress
        pv  = dw.get( 'pVol', self.dwi )                  # Volume

        for (ii,pfi,pvi) in izip(count(),pf,pv):
            S,Ja = self."mm.getStress( pfi )"      # Get stress
               and det(pf)  //It is calculated in '
               materialmodel2d'
            pvs[ii] = S * pvi * Ja                 # Stress *
               deformed volume
            if not self.ignoreNegJ:
                if Ja < 0:  raise JacobianError('
                   computeStressTensor','Neg J')
```

Next by sending the "pvs" to the grid, the grid internal force can be found ("gfi").

```
def computeInternalForce( self, dw, patch ):
        # Compute internal body forces - integrate divergence of
           stress to grid
        cIdx,cGrad = dw.getMult( ['cIdx','cGrad'], self.dwi )

        pp = dw.get( 'pVS', self.dwi )                              #
            Stress*Volume
        gg = dw.get( 'gfi', self.dwi)
        self."util.divergence"( cIdx, cGrad, pp, gg )    //The
           movement from particles to the grids happens in '
           mpmutils' class.
```

5.6.1 MaterialModel

- **__init__**

```
def __init__(self, modelName, props):
        self.modelName = modelName                    # Selects
           Material Model
        self.props = props
```

- **getStress**
  This function gets the stress and Jacobian from the function whose name is the same az the modelName introduced in "ex_two_contact"→ "Material Properties" as "matNodelName":

```
def getStress( self, F ):
       model = getattr( self, self.modelName )
       S,Ja = model(self.props, F);
       return (S,Ja)
```

- For the `matModelName`= "planeStrainNEohookean" as introduced in "ex_two_contact" our function for finding the stress is:

```
def planeStrainNeoHookean( props, F ):
       # Props - poisson, E
       I2 = F*0.
       I2[0,0] = I2[1,1] = 1.
       v = props['poisson']
       E = props['modulus']
       l = E * v / ((1.+v)*(1.-2.*v))
       m = 0.5 * E / (1.+v)
       Ja = F[0,0]*F[1,1] - F[1,0]*F[0,1]
       S = I2*l*np.log(Ja)/Ja + m/Ja * (np.dot(F, F.T) - I2)

       return (S,Ja)
```

5.7 def computeAndIntegrateAcceleration(self, dw, patch, tol):

Now that we have obtained the grid mass (gm), grid momentum (gw), internal and external forces (`gfi` and `gfe`) we can find the velocity, acceleration and update the velocity (gv, ga):

```
def computeAndIntegrateAcceleration( self, dw, patch, tol ):
       # Integrate grid acceleration
       dwi = self.dwi
       a_leap = 1. - (patch.it==0) * 0.5                   #
          Initializes leap-frog

       gm = dw.get( 'gm', dwi )                            # Mass
       gw = dw.get( 'gw', dwi )                            # Momentum
       gfi = dw.get( 'gfi', dwi )                          # Internal
          Force
       gfe = dw.get( 'gfe', dwi )                          # External
          Force
       gv = dw.get( 'gv', dwi )                            # Velocity
       ga = dw.get( 'ga', dwi )
```

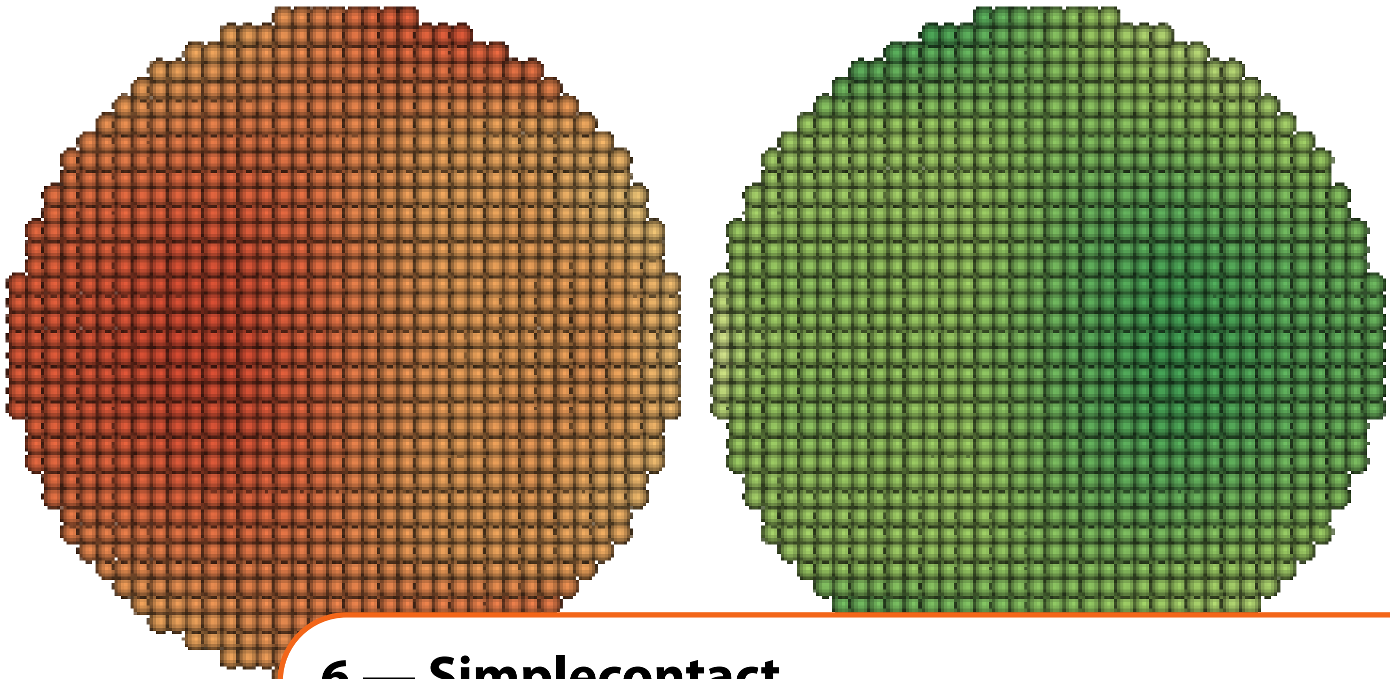
```
        gm[:] += tol
        gv[:] = gw/gm
        ga[:] = a_leap * (gfe+gfi)/gm
        gv[:] += ga*patch.dt
```

5.8 def interpolateToParticlesAndUpdate(self, dw, patch):

Here we interpolate the grid values (acceleration ga and velocity gv) to the particle and find position and velocity increment for the particles (pxI and pvI respectively) and particle velocity gradient. Finally from these data we can find the new updated particle position, particle momentum and deformation gradient:

```
def interpolateToParticlesAndUpdate( self, dw, patch ):
        dwi = self.dwi
        cIdx,cW,cGrad = dw.getMult( ['cIdx','cW','cGrad'], self.
           dwi )

        pvI = dw.get( 'pvI', dwi )
        pxI = dw.get( 'pxI', dwi )
        pGv = dw.get( 'pGv', dwi )
        ga  = dw.get( 'ga', dwi )
        gv  = dw.get( 'gv', dwi )

        self."util.interpolate"( cIdx, cW, pvI, ga )   //-->
           Happens in 'mpmutils'
        self."util.interpolate"( cIdx, cW, pxI, gv )   //-->
           Happens in 'mpmutils'
        self."util.gradient"( cIdx, cGrad, pGv, gv )   //-->
           Happens in 'mpmutils'

        px = dw.get( 'px', dwi )
        pw = dw.get( 'pw', dwi )
        pm = dw.get( 'pm', dwi )
        pF = dw.get( 'pF', dwi )

        pw += pvI * pm * patch.dt
        px += pxI * patch.dt

        self."util.dotAdd( pF, pGv*patch.dt )"                #
           pF += (pGv*dt).pF //--> Happens in 'mpmutils'
```

Note that all the movements from particles to grid and vice versa happen in ”mpmutils” module (→ (**Chapter** 8)).



6 — Simplecontact

”simplecontact” consists of a superclass: ”Simpleclass” and two subclasses: ”FreeContact” and ”FrictionlessContact” which we used ”FreeContact” in the ”ex_two_contact.py”.

6.1 SimpleContact

- **def __init__:**
  Here we initialize the material ID to the values given in the main body of our program. Moreover we initialize the list of nodes sharing a common grid to an empty list.

```
def __init__( self , dwis ):
        self.dwis = dwis
        self.nodes = []
```

- **def findIntersection**

```
 def findIntersection( self , dw ):
        self."findIntersectionSimple"( dw )
```

- **def findIntersectionSimple**
  Here we check whether the contact happened or not. If our two objects come close enough that some particles from each one of them share the same grid with each other, we say that there is a contact. At this time when we have the contact, we add the common nodes to the node list:

```
 gm0 = dw.get('gm',self.dwis[0])
 gm1 = dw.get('gm',self.dwis[1])
 self.nodes = np.where( (gm0>0)*(gm1>0) == True )[0]
```

6.2 FreeContact(SimpleContact)

This is a subclass(child class) of the above super(parent) class so it inherits the superclass properties and adds some more definitions to them.

- **def __init__(self, dwis):**

```
 def __init__( self, dwis ):
        SimpleContact.__init__(self, dwis)
```

- **exchMomentumInterpolated**:
  This function checks if any intersection nodes exist and if the condition is true, using a function named: `exchVals` it will exchange the desired values (gm, gw) of each object with each other:

```
def exchMomentumInterpolated( self, dw ):
        self.findIntersection( dw )
        if self.nodes.any():
            self."exchVals"( 'gm', dw )
            self."exchVals"( 'gw', dw )
```

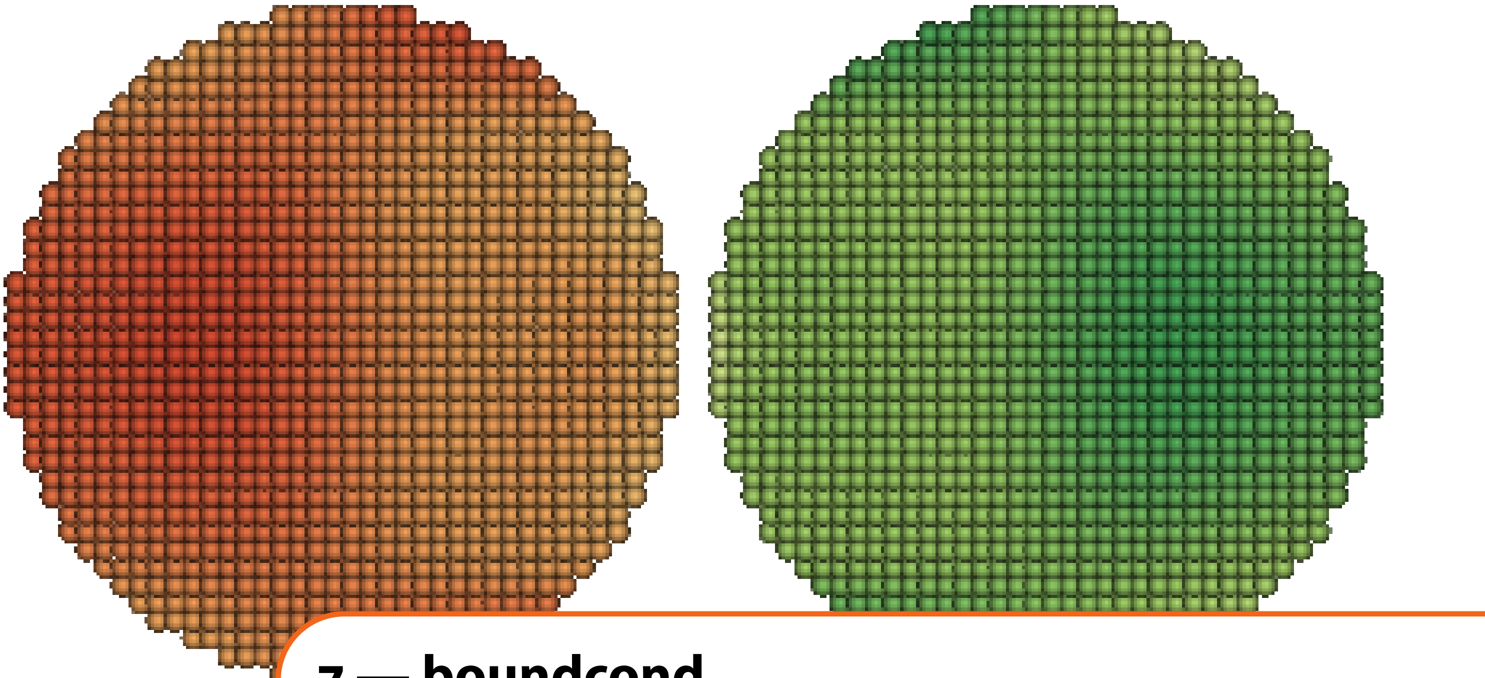
- **def exchVals** This function get the values which are going to be exchanged by the names of: `g0` (values from the object #1) and `g1`(values from the object #2). Then it adds the values from object #2 to the values from object #1. At the end it assigns the result to object #2 as well. So at the end the desired values for particles of the two objects are the same and is equal to the summation of the values obtained from particles of the two objects separately:

```
def exchVals( self, lbl, dw ):
        g0 = dw.get(lbl,self.dwis[0])   \\values from the
           object #1
        g1 = dw.get(lbl,self.dwis[1])   \\values from the
           object #2

        g0[self.nodes] += g1[self.nodes]  \\summation of the
           values
        g1[self.nodes] = g0[self.nodes]    \\same result
           assigned to both objects
```

- **def exchForceInterpolated** This function looks for any contacts. So if it finds any contact it exchanges the internal forces between the objects:

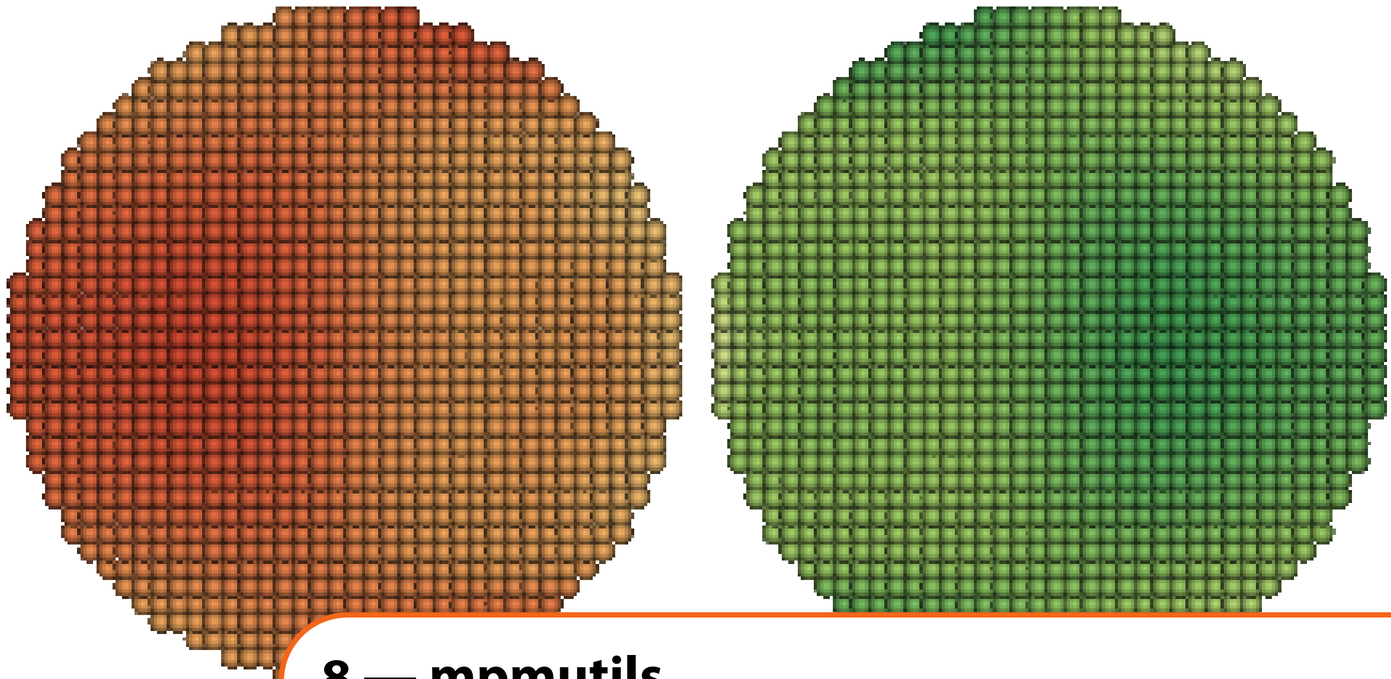
```
def exchForceInterpolated( self, dw ):
        if self.nodes.any():
            self.exchVals( 'gfi', dw )
```

Here we again used `exchVals` to exchange the values.



7 — boundcond

```
import numpy as np


#
   ================================================================================

class BoundaryCondition:
    def __init__(self, bc_type, bc_val, bc_var, fun ):
        # Set boundary condition - bc_type = 'X' or 'Y'
        # bc_val = value of x or y where condition is applied
        # bc_var is nodal variable to set
        # fun is function - takes a point as input
        self.bc_type = bc_type
        self.bc_val = bc_val
        self.bc_var = bc_var
        self.fun = fun

    def setBoundCond( self, dw, patch, tol ):
        if( self.bc_type == 'X' ):
            self.bcX( dw, patch, tol )
        else:
            self.bcY( dw, patch, tol )

    def bcX( self, dw, patch, tol ):
        #  Set boundary condition on line x=val
        gg = dw.getData( self.bc_var )
        for ii in range(len(dw.gx)):
            if( np.abs(dw.gx[ii][0]-self.bc_val) < tol ):
                gg[ii] = self.fun( dw.gx[ii] )

    def bcY( self, dw, patch, tol ):
        #  Set boundary condition on line y=val
        gg = dw.getData( self.bc_var )
        for ii in range(len(dw.gx)):
            if( np.abs(dw.gx[ii][1]-self.bc_val) < tol ):
                gg[ii] = self.fun( dw.gx[ii] )
```



8 — mpmutils

”mpmutils” basically helps in moving the data between particles and the grid:

- **integrate**:
  Integrates particle values to grid (p-¿g) using weighting function.

```
def integrate( cIdx, cW, pp, gg ):
    # Integrate particle values to grid (p->g)
    for (ppi,idx,w) in izip(pp,cIdx,cW):
        gg[idx] += ppi * w[:,np.newaxis]
    return gg
```

- **interpolate**:
  Interpolates grid values to particles (g-¿p) using weighting function.

```
def interpolate( cIdx, cW, pp, gg ):
    # Interpolate grid values to particles pp
    for (ii,idx,w) in izip(count(),cIdx,cW):
        pp[ii] = np.sum( gg[idx] * w[:,np.newaxis], 0 )
    return pp
```

- **gradient**:
  Interpolates grid value to particle gradient (gv-¿pGv) using gradient of weighting function.

```
def gradient( cIdx, cGrad, pp, gg ):
    # Interpolate a gradient
    for (ii,idxi,gradi) in izip(count(),cIdx,cGrad):
        pp[ii] = [0]
        for (idx,grad) in izip( idxi, gradi ):
            gR = np.reshape( gg[idx], (2,1) )
            cg = np.reshape( grad, (1,2) )
            pp[ii] += np.dot( gR, cg )
    return pp
```

- **divergence**:
  Sends divergence of particle field to the grid (pVs-¿gfi) using gradient of weighting function.

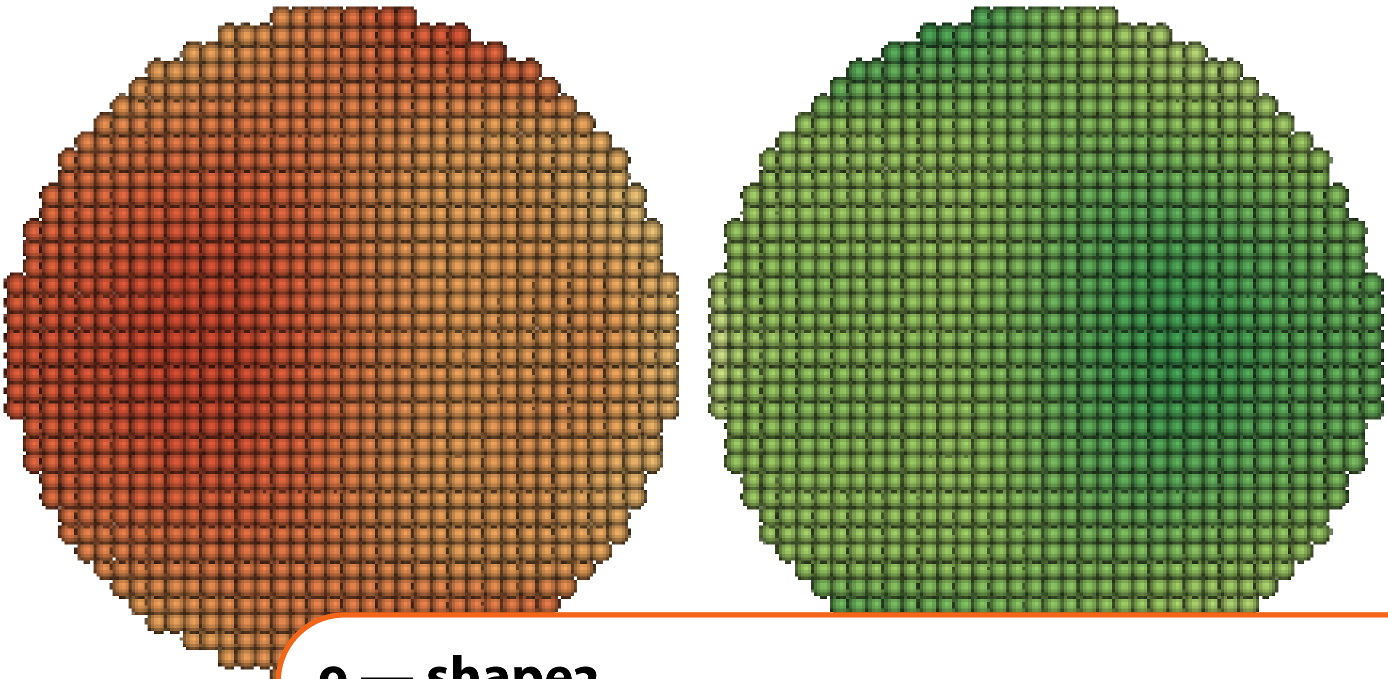
```
def divergence( cIdx, cGrad, pp, gg ):
    # Send divergence of particle field to the grid
    for (ppi,idxi,gradi) in izip(pp,cIdx,cGrad):
        for idx,grad in izip( idxi, gradi ):
            cg = np.reshape( grad, (2,1) )
            gg[idx] -= np.reshape( np.dot( ppi, cg ), 2 )
    return gg
```

- **dotAdd**:
  Is a function for adding a matrix multiplication to a data matrix (used in "Material" for calculating the deformation gradient)

```
def dotAdd( pp, qq ):
    # return pp += qq dot pp
    for (ii,ppi,qqi) in izip(count(),pp,qq):
        pp[ii] += np.dot( qqi, ppi )
```

- **readableTime**

```
def readableTime( tt ):
    attrs = ['years', 'months', 'days', 'hours', 'minutes', '
       seconds']
    h_read = lambda delta: ['%d %s' % (getattr(delta, attr),
                   getattr(delta, attr) > 1 and attr or attr
                      [:-1])
    for attr in attrs if getattr(delta, attr)]
    tm = h_read(reldelta(seconds=tt))

    st = ''
    for t in tm:
        st += t + ', '
    return st[:-2]
```



9 — shape2

”shape2” consists of one class: ”Shape” and one subclass: ”GIPM”.
In the Shape class we just initialize the shape functions and their derivatives.

```
class Shape:
    #  Shape functions - compute nodal contributions to particle
       values
    def __init__(self):
  self.dim = 2;
  self.S = np.zeros([self.dim,1])     # Value of Shape function
  self.G = np.zeros([self.dim,1])     # Value of Shape function
     derivative
```

The subclass which contains all the functions and methods of the super(parent) class just adds some other methods.

```
class GIMP(Shape):
    def __init__(self, useCython=True):
  self.nSupport = 9
  self.nGhost = 2
  Shape.__init__(self)

  if useCython:
      self.gimp = gimp2_c
  else:
      self.gimp = gimp2

    def updateContribList( self, dw, patch, dwi ):
  self.gimp.updateContribList( dw, patch, dwi )
```

updateContribList

With this function we find the nodes to which we have to find and update the contribution of each particle. Through this function we calculate `S` =”shape function”, `G` =”shape function derivative”, `cW` =”weighting function”, `cGrad` =”gradient of weighting function” Not to mention that this function itself is an assigning to a function in ”gimp2 ” module.

9.1 gimp2

Here you Find a list of nodes contribution of each one of your particles of an object. Then shape functions and their derivatives would be built up based on the information in this module.

9.1.1 def updateContribList(dw, patch, dwi):

- nx is the number of your nodes in x direction.

```
nx = patch.Nc[0]
```

- h is the cell size

```
h = patch.dX
```

- dxdy is a Length-to-Width Ratio:

```
dxdy = h[::-1]/h
```

- idxs is a list of indices of 9 nodes around the particular particle to all which the particle contributes. 0 is the index of the lower left node. 1 and 2 are nodes on the right hand side of the "0". nx in the index of the node above the "0" and so on.

```
idxs = [0,1,2,nx,nx+1,nx+2,2*nx,2*nx+1,2*nx+2]
```

- S and G are initialized at zero and their size are the same as the size of vector "h".

```
S = np.zeros(h.size)
G = np.zeros(h.size)
```

- We bring all the needed data from "datawarehouse":

```
cIdx,cW,cGrad = dw.getMult( ['cIdx','cW','cGrad'], dwi )
px,gx,pVol,pF = dw.getMult( ['px','gx','pVol','pF'], dwi )
```

- l

```
l = np.sqrt(pVol[ii]/(4.*patch.thick*dxdy)) * np.diag(pF[ii])
```

- cc
  Next step is finding the index of these nodes in the domain so we need to find cc which is the number of nodes in the domain before the lower left node of the block:"0". cc is calculated in **getCell** function.

```
for ii in range(len(pVol)):
        cc = getCell( patch, px[ii] )
```

9.1.2 def getCell(patch, pos):

- x_sc gives us the number of nodes before the particle (in x and y directions)

```
x_sc = (pos - patch.X0)/patch.dX + patch.nGhost
```

- idx actually rounds the number in "x_sc" down to the nearest integer.

```
idx = np.floor(x_sc)
```

- By calculating the rem and all the if conditions we make sure that for all the particles in the block, the nodes would be counted from the lower left node of the block. So for example if we have two different particles in the middle of the block and one is in the lower left part of the middle point and the other is on the top right side of the middle point, the nodes which contribute to these two particles would be the same and would be counted from the lower left node of the block.

```
    rem = (x_sc - 1.*idx) >= 0.5
    ii = idx[0] if rem[0] else idx[0]-1
    jj = idx[1] if rem[1] else idx[1]-1
```

- As `return` we get the whole number of the nodes in the domain and before the lower left node of the block.

```
    return int(jj * patch.Nc[0] + ii)
```

Go back to **updateContribList**: Having the ”cc” and the ”idxs” we can find the real index of all the contributing nodes in the domain → idx

```
for jj in range(9):
            idx = idxs[jj] + cc
            r = px[ii] - gx[idx]
```

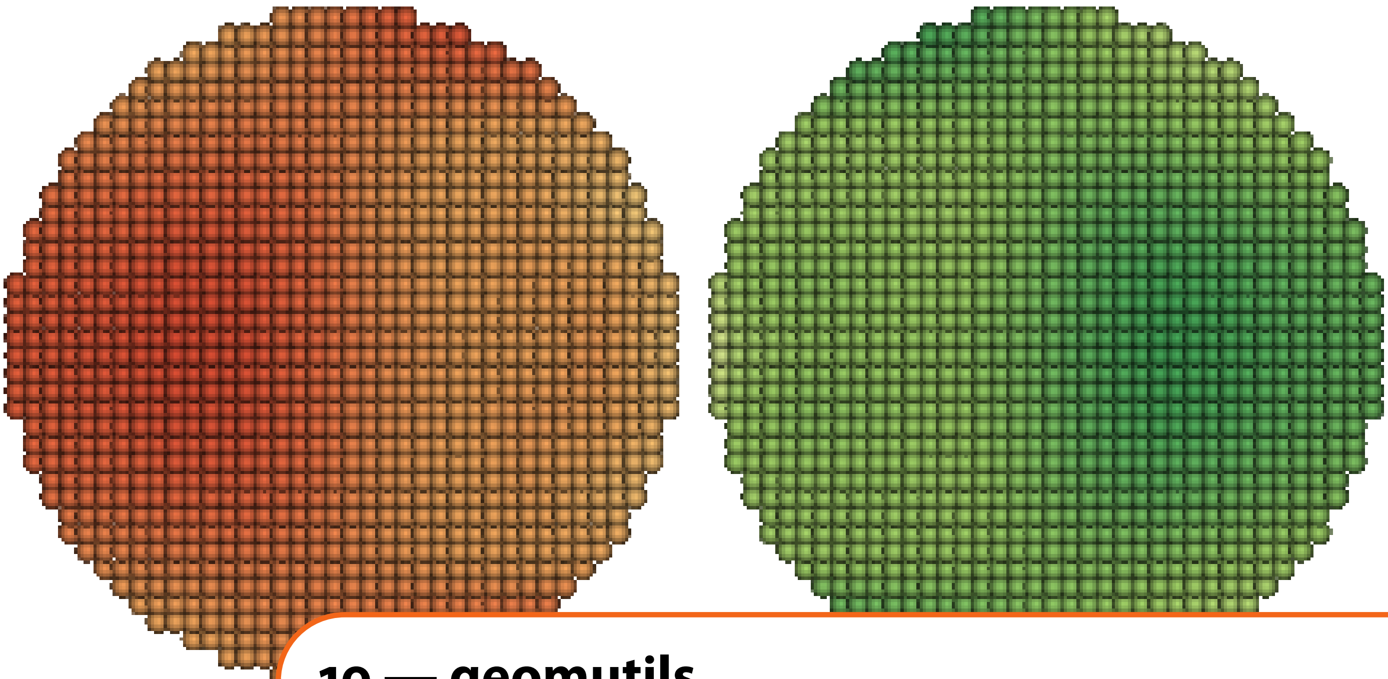
9.1.3 uSG(x, h, l)

Now having h, l and r, shape function(`S`) and shape function derivative(`G`) can be constructed in uSG

```
def uSG( x, h, l ):
    r = abs(x)
    sgnx = cmp(x,-x)

    if (r<l):       S = 1. - (r*r+l*l) / (2.*h*l);  G = -x/(h*l)
    elif(r<h-l):    S = 1. - r/h;                   G = -sgnx/h
    elif(r<h+l):    S = (h+l-r)*(h+l-r) / (4.*h*l); G = (h+l-r) /
       (-2.*sgnx*h*l)
    else:           S = G = 0.
    return (S,G)
```

and consequently weighting function(`cW`) and gradient of shape function(`cGrad`) can be found and added to the ”datawarehouse”

```
 for kk in range(len(r)):
               S[kk],G[kk] = uSG( r[kk], h[kk], l[kk] )

cIdx[ii][jj] = idx
cW[ii][jj] = S[0]*S[1]
cGrad[ii][jj] = G * S[::-1]
```



10 — geomutils

Here we basically build our objects:

- **fillRectangle** → builds a rectangular of particles

```
def fillRectangle( pt1, pt2, ppe, patch ):
    nn = "pCeil"( (pt2-pt1) / (patch.dX/ppe) )
    ps = (pt2-pt1)/nn
    vol = patch.thick * ps[0] * ps[1]
    parts = []
    for jj in range(int(nn[1])):
        for ii in range(int(nn[0])):
            ns = np.array([ii+0.5,jj+0.5])
            pt = pt1 + ps*ns
            if patch.inPatch( pt ):
                parts.append( pt )

    parts = np.array(parts)
    return (parts, vol)
```

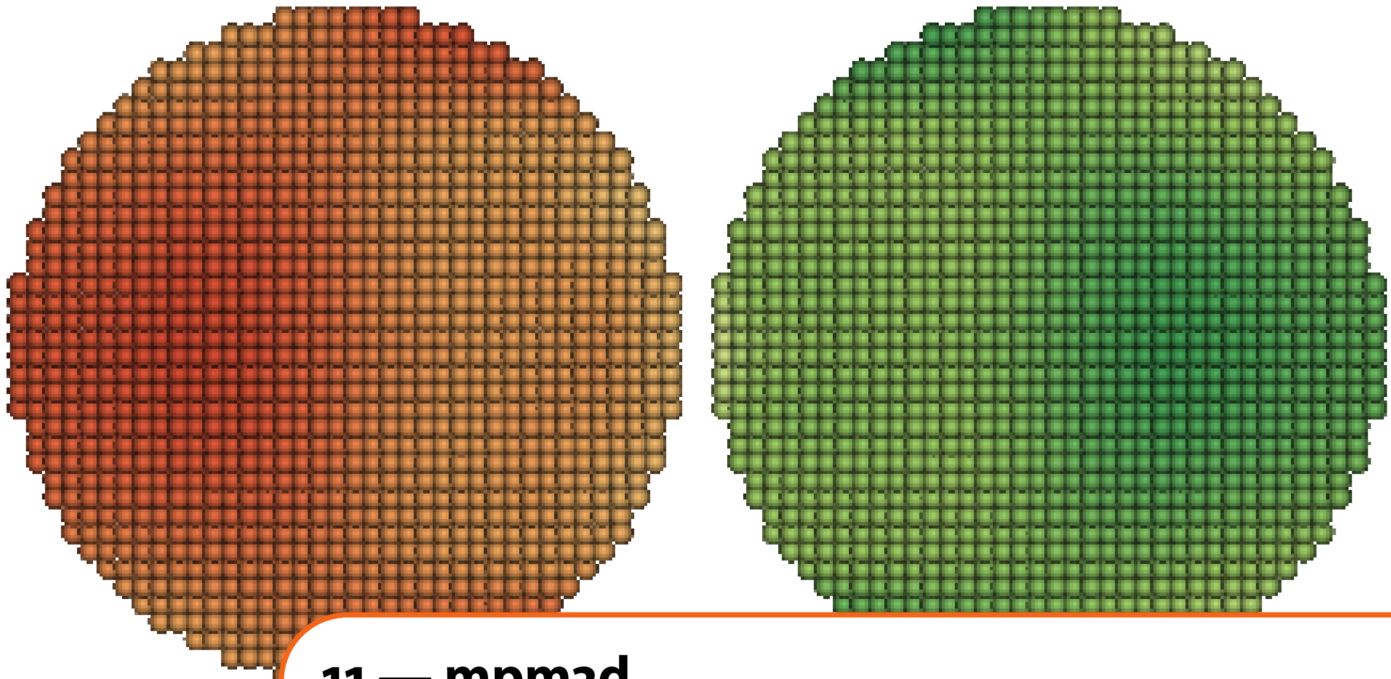
- **fillAnnulus** → builds a circle (or an annulus if r[0] is not zero) of particles

```
def fillAnnulus( pt1, r, ppe, patch ):
    nn = "pCeil"( 2*r[1] / (patch.dX/ppe) )
    ps = 2.0*r[1]/nn
    vol = patch.thick * ps[0] * ps[1]
    parts = []
    for jj in range(int(nn[1])):
        for ii in range(int(nn[0])):
            ns = np.array([ii+0.5,jj+0.5])
            pt = pt1 - r[1] + ps*ns
            if patch.inPatch( pt ):
                if ( r[0] <= np.linalg.norm( pt1 - pt ) <= r
                    [1] ):
                     parts.append(pt)

    parts = np.array(parts)
```

```
        return( parts , vol )
```

pCeil is a function which makes sure that no division by zero happens. In this function first we define a variable as ”tol” and then get the ceiling value of ”(the input variable - tol)” which is the smallest integer not less than ”(the input variable - tol)”:

```
def pCeil( x ):
    tol = 1.e-14
    return np.ceil(x-tol)
```



11 — mpm2d

This module alongside with the "Material" class and "mpmutils" gives us the materials' movements in each time step.

timeAdvance

is a function that call all the needed functions in the right order

```
def timeAdvance( dw, patch, mats, contacts=[] ):
    # Advance timestep
    updateMats( dw, patch, mats )
    applyExternalLoads( dw, patch, mats )
    interpolateParticlesToGrid( dw, patch, mats )
    exchMomentumInterpolated( dw, contacts )
    computeStressTensor( dw, patch, mats )
    computeInternalForce( dw, patch, mats )
    exchForceInterpolated( dw, contacts )
    computeAndIntegrateAcceleration( dw, patch, mats )
    exchMomentumIntegrated( dw, contacts )
    setGridBoundaryConditions( dw, patch )
    interpolateToParticlesAndUpdate( dw, patch, mats )
```

so therefore we can:

- Update the materials (particles) and get the node contributions → `updateMats`

  ```
  def updateMats( dw, patch, mats ):
      for mat in mats:
          mat.updateContributions(dw, patch)
  ```

- Apply the external forces (if any) and interpolate it on the grid as `gfe` → `applyExternalLoads`

  ```
  def applyExternalLoads( dw, patch, mats ):
      # Apply external loads to each material
      for mat in mats:
          mat.applyExternalLoads( dw, patch )
  ```

- Interpolate particles' mass and momentum to the grids using a weighting function to find the momentum and mass of the grid nodes: `gw` & `gm` → `interpolateParticlesToGrid`

```
def interpolateParticlesToGrid( dw, patch, mats ):
    # Interpolate particle mass and momentum to the grid
    for mat in mats:
        mat.interpolateParticlesToGrid( dw, patch )
```

- Throughout the program we have to check if there is any contact between the objects. Here is where we can check whether the contact has occurred or not for every particles of each objects with the material ID. This function is an assignment to a function in the ”contact” class (**Chapter** 6). When a contact is observed the gm (grid node mass) and gw (grid node momentum) would be added to gether and the result is the new mass (gm) and momentum(gw) of the grids for all the particles of both objects → `exchmomentumInterpolated`

```
def exchMomentumInterpolated( dw, contacts ):
    # Exchange Interpolated Momentum
    for contact in contacts:
        contact.exchMomentumInterpolated( dw )
```

- Compute stress tensor (`computeStressTensor`) so after that we can:
- Compute internal forces on the grid nodes `gfi`→ `computeInternalForce`

```
def computeStressTensor( dw, patch, mats ):
    # Compute Material Stress Tensors
    for mat in mats:
        mat.computeStressTensor( dw, patch )

def computeInternalForce( dw, patch, mats ):
    # Compute internal body forces
    for mat in mats:
        mat.computeInternalForce( dw, patch )
```

- Here check if the contact has occurred, the internal force for the particles of each object should be added together. The new internal force for both of the particles of the objects is the same and equals to the summation of the `gfi` from each (**Chapter** 6). → `exchForceInterpolated`

```
def exchForceInterpolated( dw, contacts ):
    # Exchange Interpolated Momentum
    for contact in contacts:
        contact.exchForceInterpolated( dw )
```

- Compute the acceleration and new nodal velocity with the internal force `gfi`, external force `gfe`, mass gm and momentum gw → `computeAndIntegrateAcceleration`

```
def computeAndIntegrateAcceleration( dw, patch, mats ):
    # Integrate grid acceleration
    for mat in mats:
        mat.computeAndIntegrateAcceleration( dw, patch, patch
           .tol )
```

- Check again for the contact → `exchmomentumIntegrated`

```
def exchMomentumIntegrated( dw, contacts ):
    # Exchange Interpolated Momentum
    for contact in contacts:
        contact.exchMomentumInterpolated( dw )
```

- Set the the boundary condition for the grid → `setGridBoundaryConditions`

```
def setGridBoundaryConditions( dw, patch ):
    # Set boundary conditions
    for bc in patch.bcs:
        bc.setBoundCond( dw, patch, patch.tol )
```

- And at last we can interpolate the grid acceleration and velocity to the particles to get the velocity increment and position increment. So now we can find the new position and new momentum for the particles → `interpolatetoParticleAndUpdate`

```
def interpolateToParticlesAndUpdate( dw, patch, mats ):
    # Interpolate velocity/accel/deformation vals to
       Particles and Move
    for mat in mats:
        mat.interpolateToParticlesAndUpdate( dw, patch )
```

- At the end we increase the time-step to start the next iteration in the `while` loop of the ”stepTime” function in ”ex_two_contact.py ”:

```
patch.stepTime()
```

All the above mentioned functions work basically by instantiating the related methods from ”Material” class (**Chapter** 5) and ”Simplecontact” (**Chapter** 6)

Bibliography