Electrostatics: Field Calculation Using Brute Force and Fast Multipole Method

Abhikalp Shekhar

1 Introduction

This is a self motivated project to develop a python library for computing electric potential and field for an arbitrary charge distribution. The brute force method is $O(N^2)$ where N is the number of charges. An approximate algorithm, for the N-body system, is implemented for speedup. The fast method is $O(NLog(N))$

2 Methodology

Here we calculate the Coulomb potential and electric field using electrostatic potential and field for each pair.

$$V = c\sum_{i=1}^{N}\sum_{j\neq i}^{N}\frac{q_i q_j}{|r_i - r_j|} \tag{1}$$

Electric field is computed as gradient of potential where the differentiation is done numerically.

$$\vec{E} = -grad(V) = -\left(\frac{\partial V}{\partial x}\vec{i} + \frac{\partial V}{\partial y}\vec{j} + \frac{\partial V}{\partial z}\vec{k}\right) \tag{2}$$

2.1 Fast Monopole Method (FMM)

When the number of charges is large, the brute force method would become very slow. The special case of 2D charge distribution implemented here is based on 'Barnes-Hut' algorithm. Here the charges are distributed in a rectangular grid. The grid can be partitioned recursively in every iteration in four squares and the partitioning stops when every square has $O(1)$ charges. The potential between far off charges can be computed by approximating the charge clusters as point charges placed at the center of mass of each cluster. The algorithm generates as Quad Tree which has 4 children (NE, NW, SE, SW) and each child can be either a leaf node or can have children.

3 Results

3.1 Electric Monopole

Here we show the electric potential and electric field of a monopole. It can be seen that the field is flowing radially outward and potential changes like $\frac{1}{r}$ where $\vec{r}$ is the position vector.

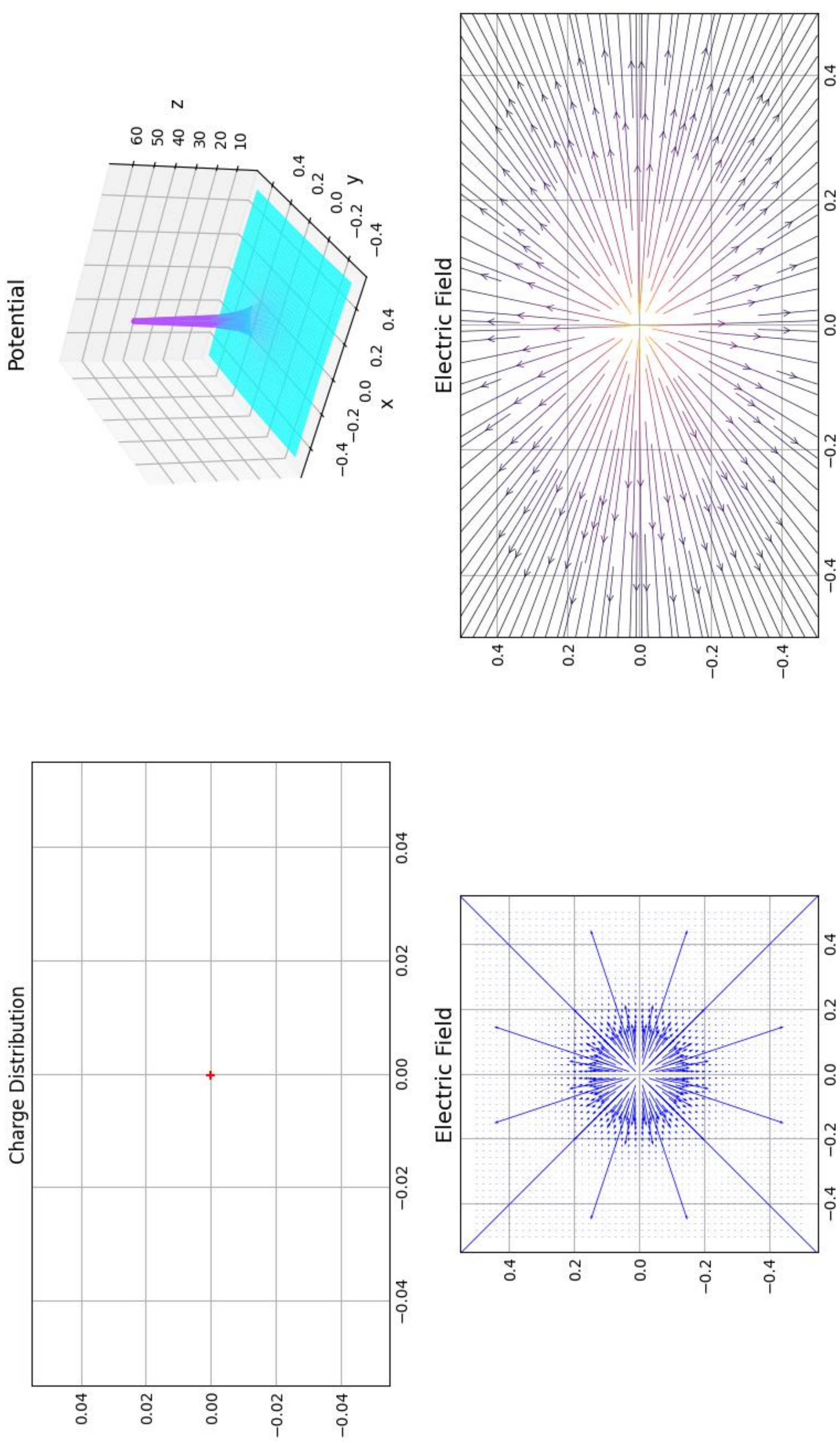

Figure 1:   Monopole: Electric Potential and Electric Field

3.2 Dipoles

Here we show the electric potential and electric field of a dipole. Here the potential changes like $\frac{1}{r^2}$.

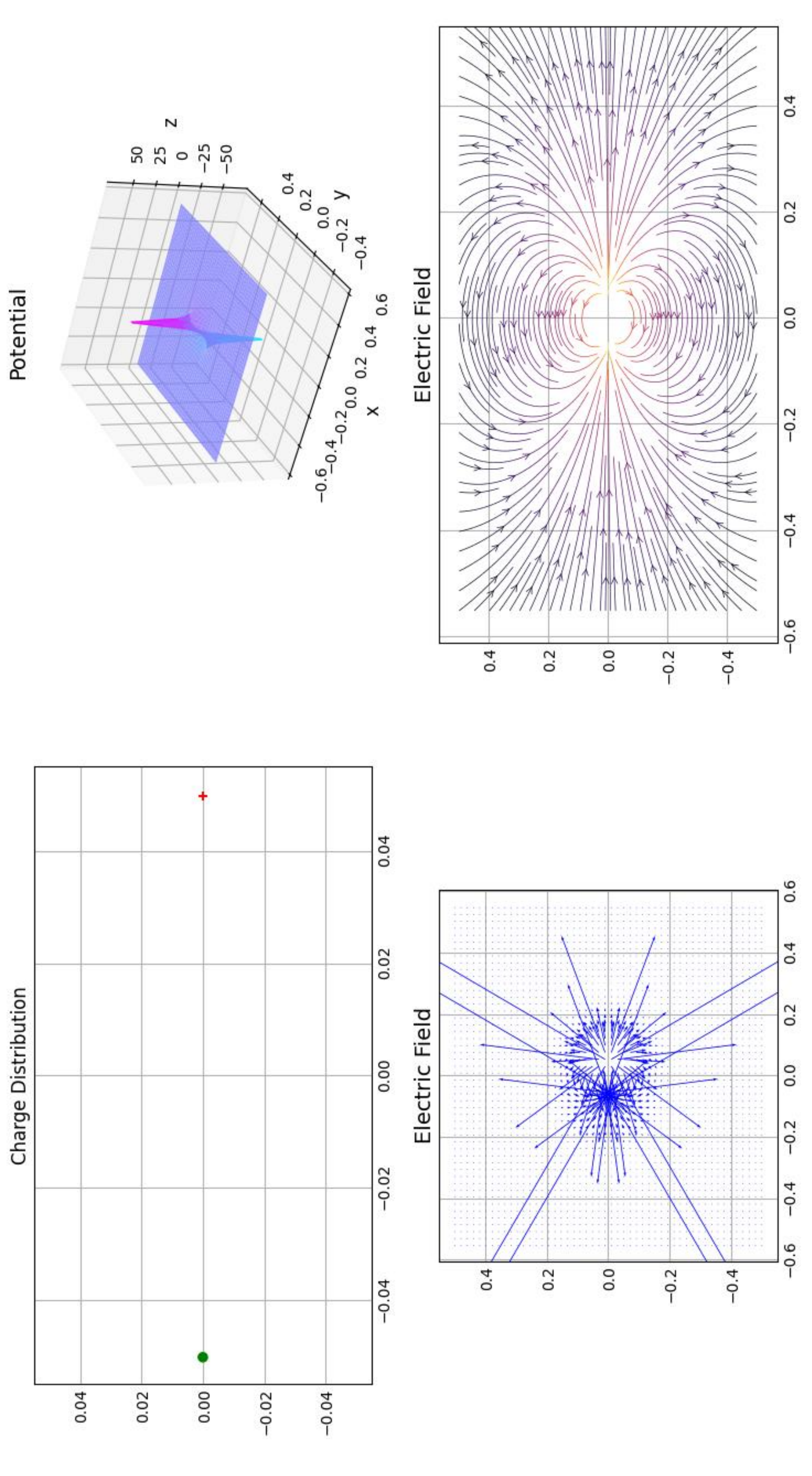

Figure 2: Dipole: Electric Potential and Electric Field

3.3 Quad-Pole

Here we show the electric potential and electric field of a quad-pole. Here the potential changes like $\frac{1}{r^3}$.

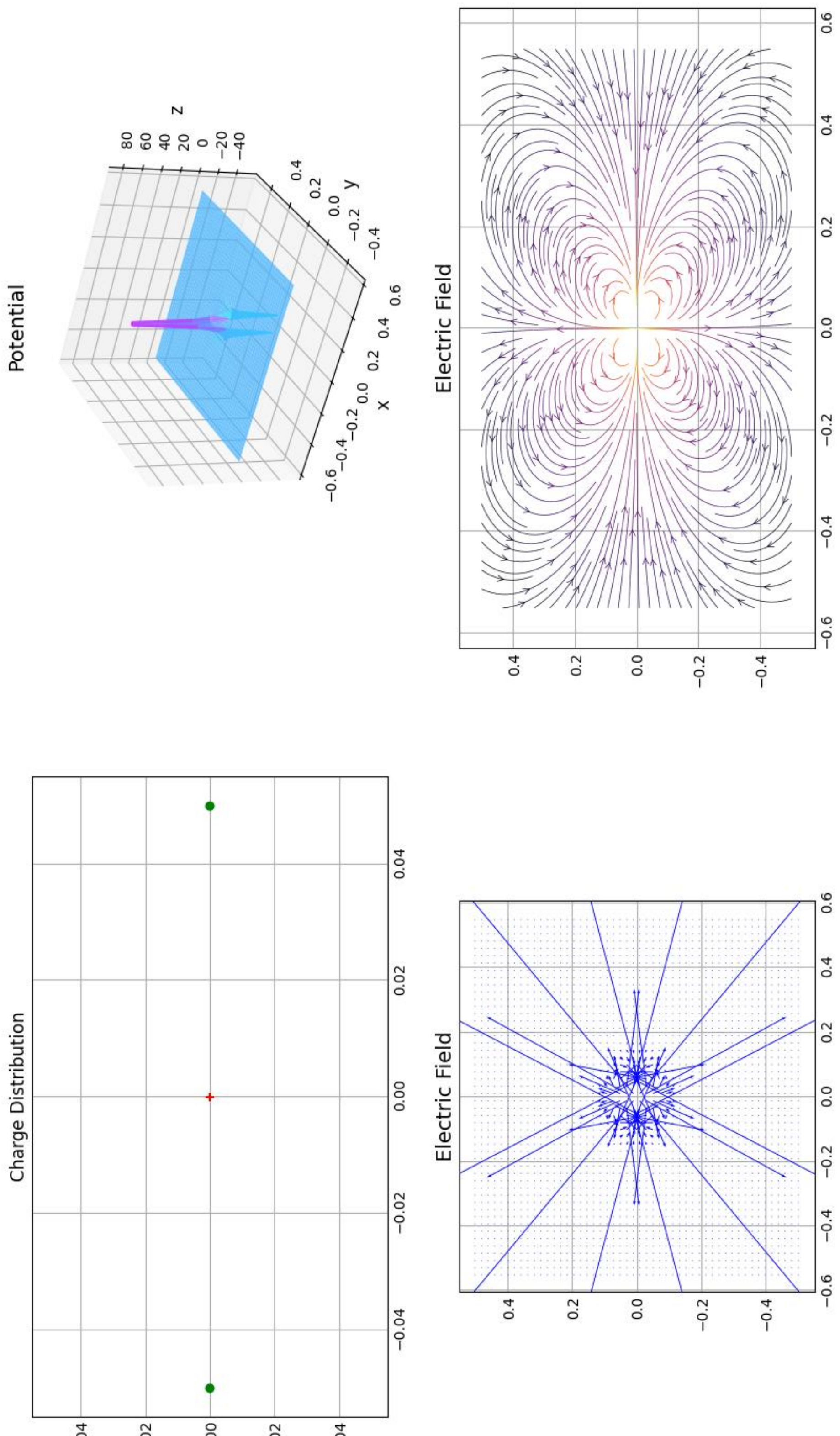

Figure 3: Quad-Pole: Electric Potential and Electric Field

3.4 Lattice Charge Distribution

Here we show the electric potential and field of a lattice with alternate plus and minus charge placed equidistantly.

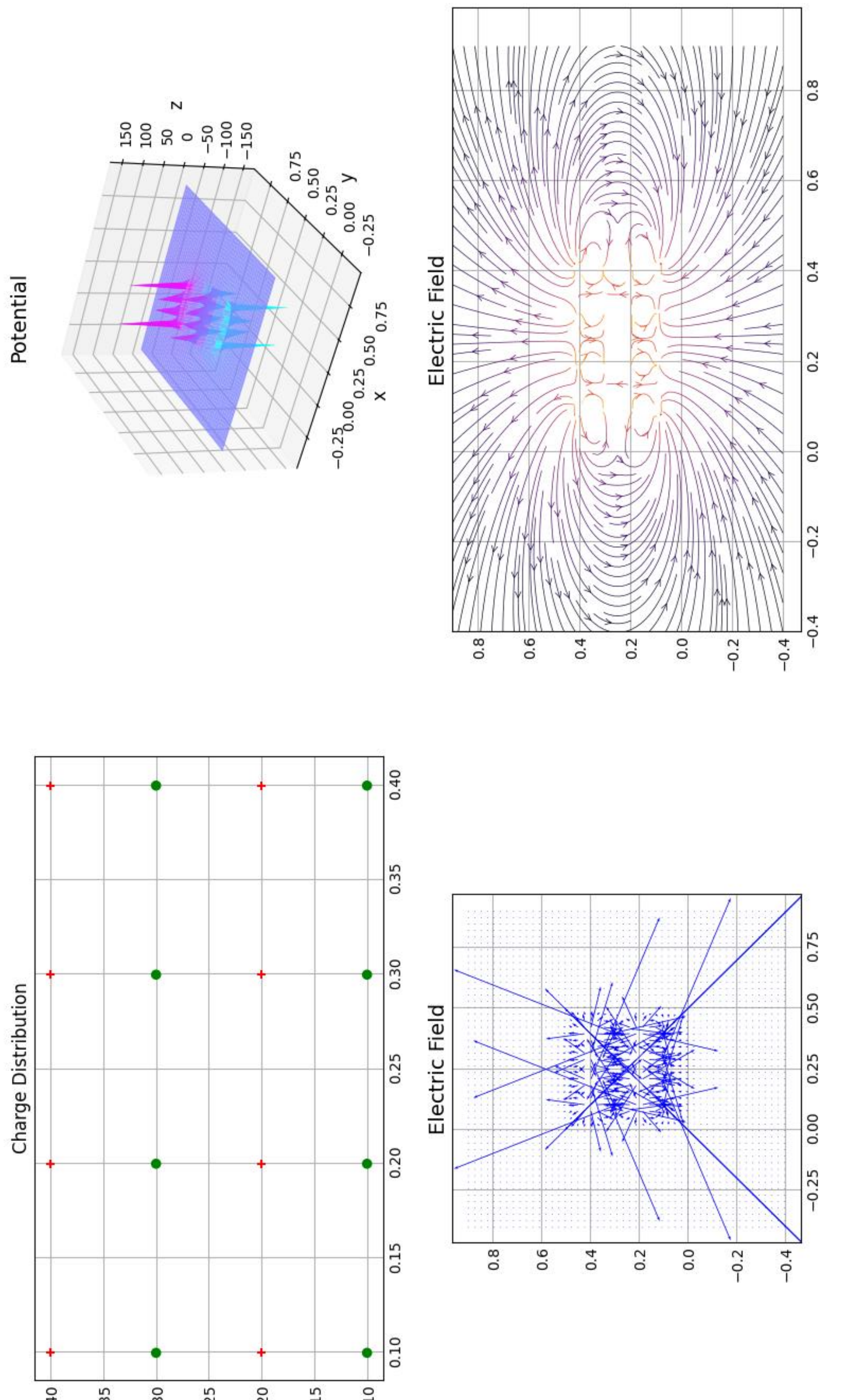

Figure 4: Lattice: Electric Potential and Electric Field

3.5 Arbitrary Charge Distribution

Here we show the electric potential and field of a uniformly random charge distribution.

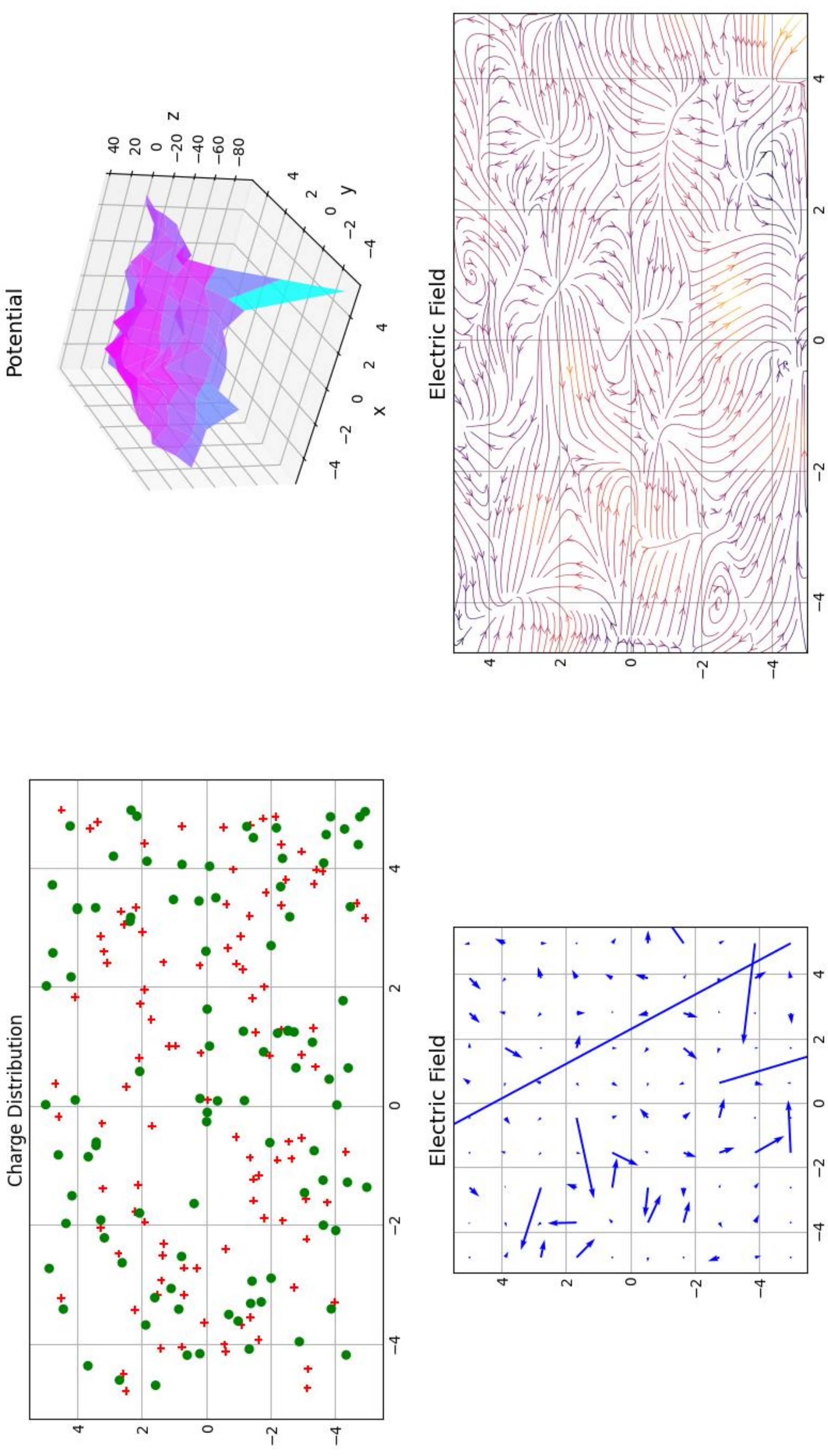

Figure 5: Random Charge Distribution: Electric Potential and Electric Field

3.6 Partitioning for FMM

Here we show the output of Quad-Tree partitioning.

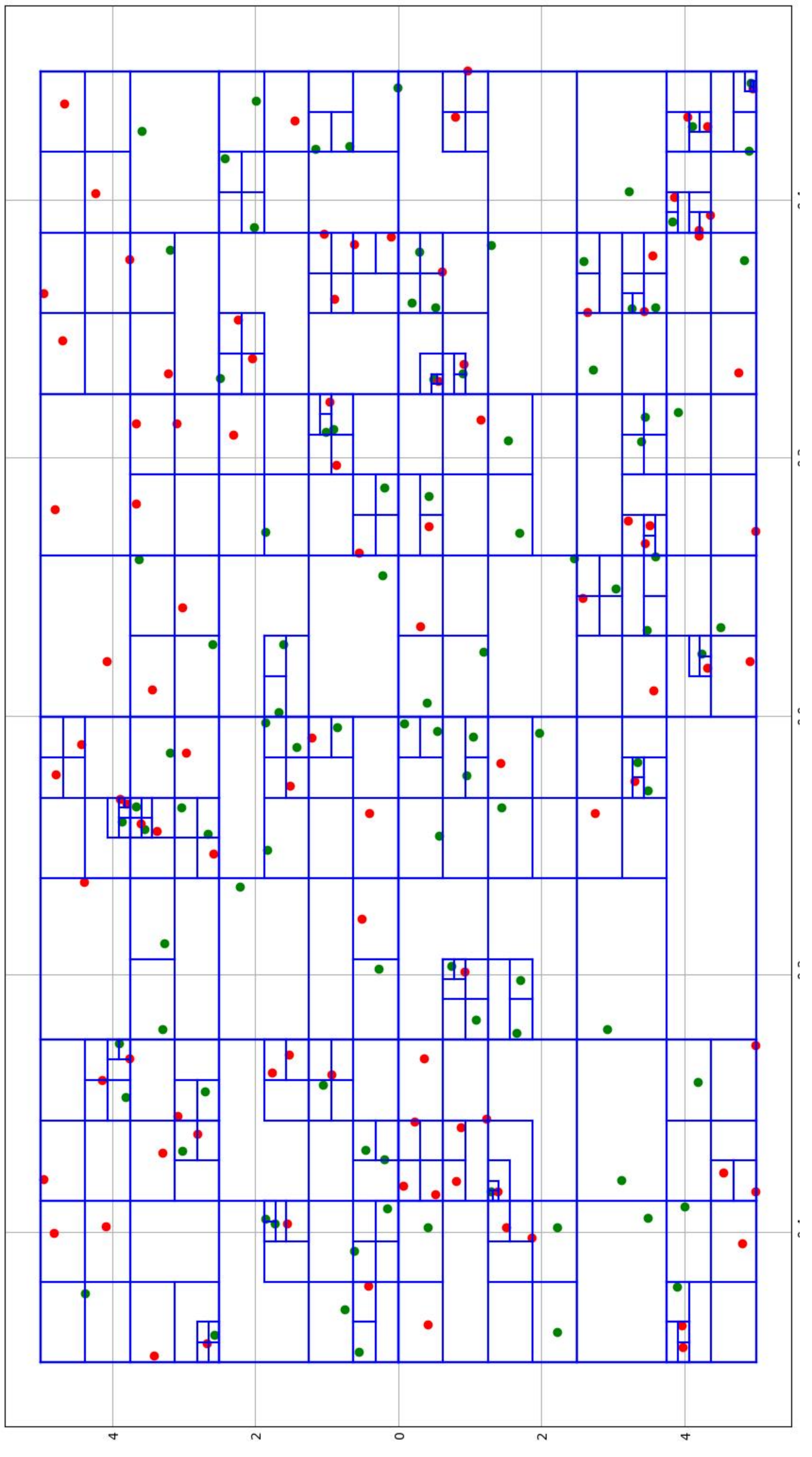

Figure 6: FMM: Partitioning

4 Python Code

4.1 electrostatics-py

```python
import math
from vector import *
import copy
import numpy as np
import fastmultipole as fmm

class Charge:
    def __init__(self,q,p,units=1.0):
        if not isinstance(q,(float,int)):
            raise Exception("Bad charge")
        if not isinstance(p,Point):
            raise Exception("Bad Point")
        self.q=q
        self.p=p
        self.units=units

    def ElectricField(self,point):
        c = self.units
        d = self.p.dist(point)
        d2 = d*d
        ef= c*self.q/d2
        unitVect = vector((-self.p.x + point.x)/d, (-self.p.y + point.y)/d, (-self.p.z + point.z)/d)
        return unitVect.scale(ef)

    def ElectricPotential(self,point):
        c = self.units
        d = self.p.dist(point)
        return c*self.q/d

    @staticmethod
    def getPotentialEnergy(charges,option):
        if option.get("method","exact")=="exact":
            n=len(charges)
            U = 0.0
            for i in range(n):
                for j in range(i+1,n):
                    dist = charges[i].p.dist(charges[j].p)
                    if abs(dist)>1e-16:
                        U = U + charges[i].q*charges[j].q/dist
            return U
        else:
            return fmm.QuadTree.getPotentialEnergy(charges)

    @staticmethod
    def getPotentialAtPoint(charges,p,option=None):
        if option is None:option = {}
        chs=copy.deepcopy(charges)
        chs.append(Charge(1.0,p))
        return Charge.getPotentialEnergy(chs,option) - Charge.getPotentialEnergy(charges,option)

if __name__ == "__main__":

    def singleChargeTest():
        ch = Charge(1.0, Point(0.0,0.0,0.0))
        fld1= ch.ElectricField(Point(1.0,1.0,0.0))
        print(fld1)

    def multipleChargeTest():
        l = [Charge(3.0,Point(4,0)), Charge(3.0,Point(0.0,0.0)), Charge(3.0,Point(0,4))]
        v = vector(0,0,0)
        p = Point(3.0,4.0)
        potential = 0.0
        for c in l:
            fld = c.ElectricField(p)
            v = v + fld
            potential = potential + c.ElectricPotential(p)

        print("Electric Field Vector Sum = {}".format(v))
        print("Electric Potential = {}".format(potential))
        def fldfunc(pnt):
            return Charge.getPotentialAtPoint(l,pnt)
        scalarfld = ScalarField(fldfunc)
        potential2 = scalarfld.getVal(p)
        print("ELectric Potential2 = {}".format(potential2))
        elecfld = scalarfld.gradient(p).scale(-1.0)
        print("Electric Field Gradient of Potential = {}".format(elecfld))

    def dipoleTest():
        l = [Charge(1.0,Point(-0.1,0)),Charge(-1.0,Point(0.1,0)) ]
        v = vector(0,0,0)
        p = Point(0.0,1.0)
        for c in l:
            fld = c.ElectricField(p)
            v = v + fld
        print(v)

    singleChargeTest()
    multipleChargeTest()
    dipoleTest()
```

4.2 vector-py

```python
# File: vector.py
# Purpose: Support for common vector operations
# Author: Abhikalp Shekhar
import math

def verify(cond,msg):
    if not cond:
        raise Exception(msg)

class BoundingBox:
    def __init__(self,xmin,xmax,ymin,ymax):
        self.xmax=xmax
        self.xmin=xmin
        self.ymax=ymax
        self.ymin=ymin

class Point:
    def __init__(self,x,y,z=0):
        if not isinstance(x,(float,int)):
            raise Exception("Bad x coordinate")
        if not isinstance(y,(float,int)):
            raise Exception("Bad y coordinate")
        self.x=x
        self.y=y
        self.z=z
    def dist(self,point):
        return math.sqrt((self.x - point.x)**2 + (self.y - point.y)**2 + (self.z - point.z)**2)

class vector:
    def __init__(self,xxx,yyy,zzz):
        if not isinstance(xxx,(float,int)):
            raise Exception("Bad x input")
        if not isinstance(yyy,(float,int)):
            raise Exception("Bad y input")
        if not isinstance(zzz,(float,int)):
            raise Exception("Bad z input")

        self.x = xxx
        self.y = yyy
        self.z = zzz

    def __str__(self):
        return "{}i + {}j + {}k".format(self.x,self.y,self.z)

    def length(self):
        l = math.sqrt(self.x**2 + self.y**2 + self.z**2)
        return l

    def scale(self,d):
        return vector(self.x*d, self.y*d, self.z*d)

    def unitvec(self):
        l = self.length()
        return self.scale(1.0/l)

    def __add__(self,b):
        return vector(self.x+b.x, self.y+b.y, self.z+b.z)

    def dot(self,b):
        return self.x*b.x + self.y*b.y + self.z*b.z

    def cross(self,b):
        return vector(self.y*b.z - self.z*b.y, self.z*b.x - self.x*b.z , self.x*b.y - self.y*b.x)

def polar(r,theta,z=0):
    return vector(r*math.cos(theta),r*math.sin(theta),z)

class ScalarField:
    def __init__(self,fieldFunc,perturb=1e-10):
        self.scalarFunc=fieldFunc
        self.perturb=perturb

    def gradient(self,p):
        pxdash=Point(p.x+self.perturb, p.y,p.z)
        pydash=Point(p.x,p.y+self.perturb,p.z)
        pzdash=Point(p.x,p.y,p.z+self.perturb)
        value=self.scalarFunc(p)
        vxdash=self.scalarFunc(pxdash)
        vydash=self.scalarFunc(pydash)
        vzdash=self.scalarFunc(pzdash)
        return vector((vxdash-value)/self.perturb,
            (vydash-value)/self.perturb,
            (vzdash-value)/self.perturb)

    def getVal(self,p):
        return self.scalarFunc(p)

class VectorField:
    def __init__(self,fieldFuncs,perturb=1e-10):
        verify(isinstance(fieldFuncs,(list,tuple)) and len(fieldFuncs)==3, "vectorfunc size should be 3")
        self.scalarFuncs=fieldFuncs
        self.perturb=perturb

    def getValx(self,p):
        return self.scalarFuncs[0](p)

    def getValy(self,p):
        return self.scalarFuncs[1](p)

    def getValz(self,p):
        return self.scalarFuncs[2](p)

    def getValxDash(self,p,bump=1e-10):
        px=Point(p.x+bump,p.y,p.z)
        return (self.getValx(px)-self.getValx(p))/bump

    def getValyDash(self,p,bump=1e-10):
        py=Point(p.x,p.y+bump,p.z)
        return (self.getValy(py)-self.getValy(p))/bump

    def getValzDash(self,p,bump=1e-10):
        pz=Point(p.x,p.y,p.z+bump)
        return (self.getValz(pz)-self.getValz(p))/bump

    def getVal(self,p):
        return vector(self.getValx(p),self.getValy(p),self.getValz(p))

    def divergence(self,p,bump=1e-10):
        return self.getValxDash(p,bump)+self.getValyDash(p,bump)+self.getValzDash(p,bump)

    def curl(self,p,bump=1e-10):
        vx,vy,vz=(self.getValxDash(p,bump),self.getValyDash(p,bump),self.getValzDash(p,bump))
        return vector(vz - vy, vx - vz , vy - vx)
```

4.3 plotfield-py

```python
#File: plotfield.py
#Purpose: helper function for plotting vector and scalar field
#Author: Abhikalp Shekhar

import numpy as np
from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
from matplotlib import cm
plt.style.use('_mpl-gallery')
from vector import *
from electrostatics import *
from Utils import *
import math

def getBoudingBox(charges):
    verify(isinstance(charges,(list,tuple)) and len(charges)>0, "charges should be a tuple")
    [verify(isinstance(x,(Charge,Point)),"bad input") for x in charges]
    if isinstance(charges[0], Charge):
        points = [x.p for x in charges]
    else:
        points = charges
    xmin = min([p.x for p in points])
    ymin = min([p.y for p in points])
    xmax = max([p.x for p in points])
    ymax = max([p.y for p in points])
    if xmax-xmin < 1:
        d = 1.0
        xmax=xmax+d/2.0
        xmin=xmin-d/2.0
    if ymax-ymin < 1:
        d=1.0
        ymax=ymax+d/2.0
        ymin=ymin-d/2.0
    return BoundingBox(xmin,xmax,ymin,ymax)

def showBox(node):
    def drawbox(box,x1=[],y1=[],x2=[],y2=[]):
        x1=[box.xmin,box.xmax]
        y1=[box.ymin,box.ymin]
        plt.plot(x1,y1,'b-')
        x1=[box.xmin,box.xmin]
        y1=[box.ymin,box.ymax]
        plt.plot(x1,y1,'b-')
        x1=[box.xmin,box.xmax]
        y1=[box.ymax,box.ymax]
        plt.plot(x1,y1,'b-')
        x1=[box.xmax,box.xmax]
        y1=[box.ymin,box.ymax]
        plt.plot(x1,y1,'b-')

    drawbox(node.box)
    if node.ne: showBox(node.ne)
    if node.nw: showBox(node.nw)
    if node.se: showBox(node.se)
    if node.sw: showBox(node.sw)

def showTree(node,wts,pts):
    showBox(node)
    sz=len(pts)
    for i in range(sz):
        if wts[i]<0:
            plt.scatter(pts[i].x,pts[i].y,c='red')
        else:
            plt.scatter(pts[i].x,pts[i].y,c='green')

def showCharges(charges,box,showvals,fig):
    xx=[c.p.x for c in charges]
    yy=[c.p.y for c in charges]
    mx=max([abs(x.q) for x in charges ])
    sz=[30+5*int(abs(x.q)/mx) for x in charges]
    vp=np.ma.masked_where(np.asarray([c.q for c in charges]) <0.0,sz)
    vn=np.ma.masked_where(np.asarray([c.q for c in charges]) >0.0,sz)
    axs = fig.add_subplot(221)
    axs.set_title('Charge Distribution')
    axs.scatter(xx,yy,s=vp,marker="+",c="red")
    axs.scatter(xx,yy,s=vn,marker="o",c="green")

def showPotentialsAndFields(charges,box,options,fig,plotfield=False):
    nxpoints = options.get('nxpoints',20)
    nypoints = options.get('nypoints',20)
    xx = np.linspace(box.xmin,box.xmax,nxpoints)
    yy = np.linspace(box.ymin,box.ymax,nypoints)
    XX,YY=np.meshgrid(xx,yy)
    potential = np.zeros((nxpoints,nypoints))
    def potentialfn(p):
        return Charge.getPotentialAtPoint(charges,p,options)

    if plotfield:
        ex=np.zeros((nxpoints,nypoints))
        ey=np.zeros((nxpoints,nypoints))
        scalarfld = ScalarField(potentialfn)

    for i in range(XX.shape[0]):
        for j in range(XX.shape[1]):
            p = Point(XX[i,j],YY[i,j])
            potential[i,j] = potentialfn(p)
            if plotfield:
                e = scalarfld.gradient(p).scale(-1.0)
                ex[i,j],ey[i,j]=(e.x,e.y)

    axs = fig.add_subplot(222, projection='3d')
    axs.plot_surface(XX, YY, potential, cmap='cool', alpha=0.8)
    #axs.set_zlim3d(-1, 1)
    axs.set_title('Potential', fontsize=14)
    axs.set_xlabel('x', fontsize=12)
    axs.set_ylabel('y', fontsize=12)
    axs.set_zlabel('z', fontsize=12)

    if plotfield:
        color = np.log(np.hypot(ex, ey))
        axs2 = fig.add_subplot(223)
        axs2.quiver(XX,YY,ex,ey,color='b', linewidth=0.5, cmap=plt.get_cmap('gist_earth'))
        axs2.set_aspect('equal')
        axs3 = fig.add_subplot(224)
        axs3.streamplot(XX,YY,ex,ey,color=color,linewidth=0.5, cmap=plt.cm.inferno, density = 2, arrowstyle='->', arrowsize=1)
        axs2.set_title('Electric Field', fontsize=14)
        axs3.set_title('Electric Field', fontsize=14)


def plot2DChargeAndFields(charges,box=None,options=None):
    if options is None:
        options = {}
    verify(isinstance(charges,(list,tuple)), "charges should be a tuple")
    [verify(isinstance(x,Charge),"bad input") for x in charges]
    if box is None:
        box = getBoudingBox(charges)
    fig = plt.figure()

    showCharges(charges,box,options.get("showvalues",True),fig)
    showPotentialsAndFields(charges,box,options,fig,True)
    plt.show()


if __name__ == "__main__":
    np.random.seed(97)
    charges = []
    for i in range(200):
        charges.append(Charge(np.random.uniform(-5.0,5.0),
        Point(np.random.uniform(-0.5,0.5),np.random.uniform(-0.5,0.5))))
    plot2DChargeAndFields(charges)
    #plot2DChargeAndFields(charges,options={"method":"fmm"})
    print("done")
```

4.4 fastmultipole-py

```python
import numpy as np
from vector import *
from electrostatics import *
from Utils import *
import math
import plotfield as pfl

class Node:
    def __init__(self,nw,ne,sw,se,wt,cg,l,box):
        self.nw=nw
        self.ne=ne
        self.sw=sw
        self.se=se
        self.l = l
        self.wt=wt
        self.cg=cg
        self.box=box

    def isleafnode(self):
        return self.ne== None and \
            self.nw == None and \
            self.se == None and \
            self.sw == None

    def energy(self):
        u = 0.0
        if(self.ne and not self.ne.isleafnode()):
            u = u+self.ne.energy()
        if(self.nw and not self.nw.isleafnode()):
            u = u+self.nw.energy()
        if(self.se and not self.se.isleafnode()):
            u = u+self.se.energy()
        if(self.sw and not self.sw.isleafnode()):
            u = u+self.sw.energy()

        if self.ne:
            if self.nw: u = u + 0.5*self.ne.wt*self.nw.wt/self.ne.cg.dist(self.nw.cg)
            if self.sw: u = u + 0.5*self.ne.wt*self.sw.wt/self.ne.cg.dist(self.sw.cg)
            if self.se: u = u + 0.5*self.ne.wt*self.se.wt/self.ne.cg.dist(self.se.cg)

        if self.nw:
            if self.ne: u = u + 0.5*self.nw.wt*self.ne.wt/self.nw.cg.dist(self.ne.cg)
            if self.sw: u = u + 0.5*self.nw.wt*self.sw.wt/self.nw.cg.dist(self.sw.cg)
            if self.se: u = u + 0.5*self.nw.wt*self.se.wt/self.nw.cg.dist(self.se.cg)

        if self.se:
            if self.nw: u = u + 0.5*self.se.wt*self.nw.wt/self.se.cg.dist(self.nw.cg)
            if self.sw: u = u + 0.5*self.se.wt*self.sw.wt/self.se.cg.dist(self.sw.cg)
            if self.ne: u = u + 0.5*self.se.wt*self.ne.wt/self.se.cg.dist(self.ne.cg)

        if self.sw:
            if self.ne: u = u + 0.5*self.sw.wt*self.ne.wt/self.sw.cg.dist(self.ne.cg)
            if self.nw: u = u + 0.5*self.sw.wt*self.nw.wt/self.sw.cg.dist(self.nw.cg)
            if self.se: u = u + 0.5*self.sw.wt*self.se.wt/self.sw.cg.dist(self.se.cg)

        return u

class QuadTree:
    def __init__(self,wts,pts):
        #assumes pts to be in (-1,1)*(1,1)
        self.wts=np.asanyarray(wts)
        self.pts=pts

    @staticmethod
    def getcg(wts,pts):
        wt = np.sum(wts)
        ptx = np.sum(np.asarray([p.x*q for p,q in zip(pts,wts)]))
        pty = np.sum(np.asarray([p.y*q for p,q in zip(pts,wts)]))
        return wt,Point(ptx/wt,pty/wt)

    @staticmethod
    def getPotentialEnergy(charges):
        topnode = createTree(np.asarray([c.q for c in charges]),
                             [c.p for c in charges])
        return topnode.energy()

def createTree(wts,pts,node=None,box=None, fig=None):
    wt,cg=QuadTree.getcg(wts, pts)
    if node==None:
        level=0
        node = Node(None,None,None,None,wt,cg,level,BoundingBox(-0.5,0.5,-0.5,0.5))
        box = node.box
    else:
        level=node.l
        node = Node(None,None,None,None,wt,cg,level+1,box)

    ptsne=[]
    wtsne=[]
    ptsnw=[]
    wtsnw=[]
    ptsse=[]
    wtsse=[]
    ptssw=[]
    wtssw=[]
    side = (box.xmax-box.xmin)/2.0
    for wt,pt in zip(wts,pts):
        if pt.x>box.xmin and pt.x>=box.xmin+side:
            if pt.y>box.ymin and pt.y>=box.ymin+side:
                ptsne.append(pt)
                wtsne.append(wt)
            else:
                ptsse.append(pt)
                wtsse.append(wt)
        else:
            if pt.y>box.ymin and pt.y>=box.ymin+side:
                ptsnw.append(pt)
                wtsnw.append(wt)
            else:
                ptssw.append(pt)
                wtssw.append(wt)

    if len(ptsne)>1:
        node.ne=createTree(wtsne,ptsne,node,BoundingBox(box.xmin+side,box.xmax,box.ymin+side,box.ymax))
    else:
        if len(ptsne)>0:
            wt,cg=QuadTree.getcg(wtsne, ptsne)
            node.ne=Node(None,None,None,None,wt,cg,level+1,BoundingBox(box.xmin+side,box.xmax,box.ymin+side,box.ymax))

    if len(ptsnw)>1:
        node.nw=createTree(wtsnw,ptsnw,node,BoundingBox(box.xmin,box.xmin+side,box.ymin+side,box.ymax))
    else:
        if len(ptsnw)>0:
            wt,cg=QuadTree.getcg(wtsnw, ptsnw)
            node.nw=Node(None,None,None,None,wt,cg,level+1,BoundingBox(box.xmin,box.xmin+side,box.ymin+side,box.ymax))

    if len(ptsse)>1:
        node.se=createTree(wtsse,ptsse,node,BoundingBox(box.xmin+side,box.xmax,box.ymin,box.ymin+side))
    else:
        if len(ptsse)>0:
            wt,cg=QuadTree.getcg(wtsse, ptsse)
            node.se=Node(None,None,None,None,wt,cg,level+1,BoundingBox(box.xmin+side,box.xmax,box.ymin,box.ymin+side))

    if len(ptssw)>1:
        node.sw=createTree(wtssw,ptssw,node,BoundingBox(box.xmin,box.xmin+side,box.ymin,box.ymin+side))
    else:
        if len(ptssw):
            wt,cg=QuadTree.getcg(wtssw, ptssw)
            node.sw=Node(None,None,None,None,wt,cg,level+1,BoundingBox(box.xmin,box.xmin+side,box.ymin,box.ymin+side))

    return node


if __name__=="__main__":
    #node = createTree([1.0,1.0,1.0,1.0],[Point(-0.25,0.25),Point(-0.25,-0.25),Point(0.25,-0.25),Point(0.25,0.25)])
    import matplotlib.pyplot as plt
    wts=[1.0,2.0,1.0,2.0,1.0,2.0,1.0,2.0]
    pts=[Point(-0.25,0.25),Point(-0.20,0.20),
        Point(-0.25,-0.25),Point(-0.20,-0.20),
        Point(0.25,-0.25),Point(0.20,-0.20),
        Point(0.25,0.25),Point(0.20,0.20)]
    node = createTree(wts,pts)
    energy = node.energy()
    charges = [Charge(x,p) for x,p in zip(wts,pts)]
    energy2 = Charge.getPotentialEnergy(charges,{})
    print("Energy = {}\tEnergy2 = {}".format(energy,energy2))
    pfl.showTree(node,wts,pts)
    plt.show()
    print("done")
    wts=[]
    pts=[]
    for i in range(200):
        wts.append(np.random.uniform(-5.0,5.0))
        pts.append(Point(np.random.uniform(-0.5,0.5),np.random.uniform(-0.5,0.5)))
    node = createTree(wts,pts)
    energy = node.energy()
    charges = [Charge(x,p) for x,p in zip(wts,pts)]
    energy2 = Charge.getPotentialEnergy(charges,{})
    print("Energy = {}\tEnergy2 = {}".format(energy,energy2))
    pfl.showTree(node,wts,pts)
    plt.show()
```