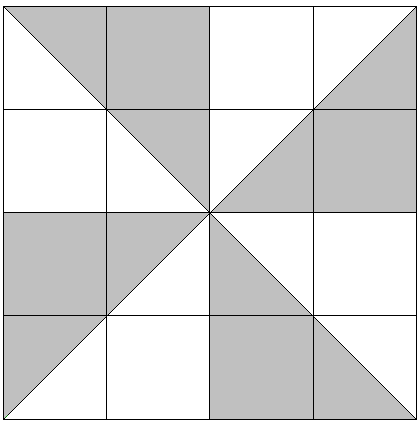
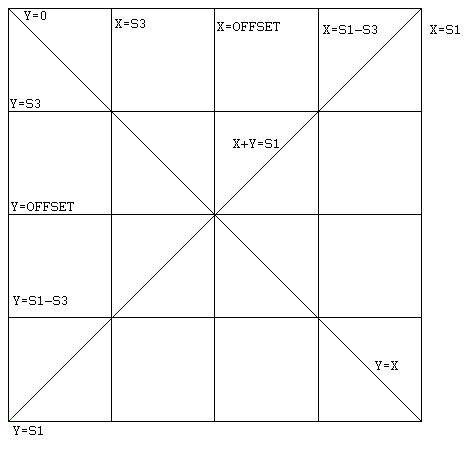
|  |  |
| --- | --- |
|  | **2013** |
|  | Ghost Code  Corey McCarty |

|  |
| --- |
| **[py-Sim-e fields]** Pronounced Pies-M-E Fields |
| -A lightweight app for rendering and calculating electric field |



With a great deal of programming languages to choose from, Python proves itself to be a major contender. Libraries like NumPy, SciPy, and MatPlotLib make calculations more like spoken language; while still keeping up with the power. The syntax of Python’s native for loops reads easier than most programming languages, and the available libraries and utilities for Python are leaps and bounds beyond Matlab. The process and memory intensive calculations of triple integrals and nested loops can be quite burdensome .



1. Layer
   1. Summary page 4
   2. Code page 5
2. Tools
   1. Summary page 8
   2. Code page 9
3. Simulation
   1. Summary page 11
   2. Code page 13
4. Layer
   1. Data:
      1. SPACE —
         1. square numpy array
      2. S—
         1. is the measure of height and width of the array
      3. S1—
         1. size+1 for symmetry the axes are added to size
      4. S2—
         1. half of the size
      5. S4—
         1. a quarter of the size
      6. SHAPE—
         1. (s,s)
   2. Functions:
      1. INSERT—
         1. basic setter for 2D array
      2. MIRROR2—
         1. inserts points at (x,y) and (-x,y)
      3. MIRROR4 —
         1. inserts mirror2 points and (x,-y) and (-x,-y)
      4. MIRROR8—
         1. inserts mirror4 as well as reflexions of each one about the lines x=y and x+y=k
      5. SHOW—
         1. outputs the array to the screen, solely for troubleshooting
   3. Algorithms
      1. Symmetry is used to reduce the amount of looping that is needed.
      2. 8 lines of symmetry were found due to the nature of the shapes involved

import numpy as np

from scipy.misc import toimage

# this creates a 2D layer to be used in the space class

class Layer:

def \_\_init\_\_(self, size):

self.s=size

self.s1=size+1

self.s2=(size/2)+1

self.s4=size/4

self.shape=np.array((self.s1,self.s1))

self.space=np.zeros((self.s1,self.s1))

def insert(self,x,y,value):

self.space[x,y]=value

def mirror8(self,x,y,value):

## this considers 4 lines of symmetry

## (x,y)&(y,x) mirrors across y=x

## (x,y)&(x2,y) mirrors left to right

## (x,y)&(x,y2) mirrors top to bottom

## (x,y)&(x2,y2) mirrors across x+y=max

if value==0:

return

x2=self.s-x

y2=self.s-y

self.space[x,y]=value

self.space[y,x]=value

self.space[x,y2]=value

self.space[y2,x]=value

self.space[x2,y]=value

self.space[y,x2]=value

self.space[x2,y2]=value

self.space[y2,x2]=value

def mirror4(self,x,y,value):

if value==0:

return

# this just considers mirror in vertical and horizontal

x2=self.s-x

y2=self.s-y

self.space[x,y]=value

self.space[x,y2]=value

self.space[x2,y]=value

self.space[x2,y2]=value

def mirror2(self,x,y,value):

# mirrors only in x

if value==0:

return

x2=self.s-x

self.space[x,y]=value

self.space[x2,y]=value

def show(self):

toimage(self.space).show()

1. Tools
   1. Data:
      1. None
   2. Functions:
      1. PI —
         1. input: none
         2. output: returns the math.pi value
      2. E0—
         1. Input: none
         2. Output: 8.85E-12
      3. READ—
         1. Input: image
         2. Output: numpy array
      4. WRITE—
         1. Input: array
         2. Output: an image
      5. RADV—
         1. Input: v1,v2
         2. Output: v3=v2-v1 vector
      6. RAD—
         1. Input: p1,p2
         2. Output: vector of form p2-p1
      7. MAGNITUDE—
         1. Input: from rad
         2. Output the magnitude from the rad function output
      8. UNITRAD—
         1. Input: from rad & magnitude
         2. Output: the direction from the rad vector
      9. BLANK—
         1. Input: none
         2. Output: the zero vector of # dimensions of float values
      10. FIELD—
          1. Input: 2 points and a charge value
          2. Output: (Runit\*q\*4\*pi\*E0)/Rmag^2

import numpy as np

import Image

import math

def pi():

return math.pi

def E0():

return 8.85\*(10\*\*-12)

def read(picture):

## reads image into np.array

## picture of given name is returned as numpy.array

## resulting array has values from 0 to 255

## black is 0 and white is 255

im = Image.open(picture)

im\_grey = im.convert('L') # convert the image to \*greyscale\*

im\_array = np.array(im\_grey)

return im\_array

def write(A, picture):

## this bit writes the array as an image

## takes in np.array A, outputs image of given name

im = Image.fromarray(A)

im.save(picture)

def radv(v1,v2):

## this returns a 3D vector for the radius between two points

## format of vector is [x,y,z]

v3=blank()

v3[0]=v2[0]-v1[0]

v3[1]=v2[1]-v1[1]

v3[2]=v2[2]-v1[2]

return v3

def rad(x1,y1,z1,x2,y2,z2):

## This returns the vector between to points in 3 space

## used to create the R vector in field formula

v1=np.array([x1,y1,z1])

v2=np.array([x2,y2,z2])

v3=radv(v1,v2)

return v3

def magnitude(vector):

## this returns the magnitude of 3D vector

## format of vector is [x,y,z]

## used for radius magnitude in field formula

## also used for creating unit vector

mag=math.sqrt(vector[0]\*\*2 + vector[1]\*\*2 + vector[2]\*\*2)

return mag

def unitrad(vector):

## this returns a unit vector for a given vector

## used for unit radius in field formula

x=vector[0]/magnitude(vector)

y=vector[1]/magnitude(vector)

z=vector[2]/magnitude(vector)

return np.array((x,y,z))

def blank():

## returns a zero vector of floats

## used for vector initialization

z=0.0

return np.array((z,z,z))

def field(i,j,k,x1,y1,z,value):

# field equation such that

# E=(Runit\*q\*c)/Rmag^2

c=float((4\*pi()\*E0())\*\*-1)

Rvect=rad(i,j,k,x1,y1,z) # Vector Radius

Rmag=magnitude(Rvect) # Magnitude Radius

Runit=unitrad(Rvect) # Unit vector Radius

return (Runit\*c\*value)/(Rmag\*\*2)

1. Class: SIMULATION
   1. Data:
      1. S—
         1. size not including axes
      2. S1—
         1. size including the axes
      3. S2—
         1. the distance in both directions from the axes
      4. OFFSET—
         1. location of the axes (the middle)
      5. S3—
         1. the floor value of the size/3
      6. MAGNITUDE—
         1. corresponds with the total charge of a finite object
         2. the linear charge density of infinite object
         3. or the voltage difference between plates
      7. INFINITE—
         1. boolean value for if the object is infinite
      8. IMAGE—
         1. the string that references the object type
      9. TOP, FRONT, & SIDE—
         1. are images representing magnitude of electric field due to the charged object
         2. top has positive x to the right and y down
         3. front has positive y right and z down
         4. side has positive x right and z down
   2. Functions:
      1. Interrogate—
         1. Input: p=(x,y,z) and bloat=(is it going to be difficult?)
         2. Output: electric force per unit charge at a given point
         3. Functionality:
            1. the algorythm checks to make sure that p is not inside the object

is true for coordinates inside of the object or outside of the parallel plates

* + - * 1. and then for the case of parallel plates(inside)

uniform field throughout the space between

* + - 1. next is the general infinite case
         1. (lambda)/(2\*pi\*E0\*R)
      2. then is the "bloated" portion (originally involved 3 nested loops)
         1. was able to reduce to linear time using symmetry and the total charge of the object
         2. used for finite cases that aren’t round
         3. see the figures and descriptions
      3. else statement models the object as a point at the origin
         1. can use this one for round objects
    1. DRAW—
       1. renders "images" of the objects being simulated
          1. Wire—single dot with length

put a dot in the middle

constant runtime

* + - * 1. Box—hollow cube

using the mirror 8 symmetry the loop draws a straight line

linear runtime

* + - * 1. Block—solid cube

drawn as concentric hollow boxes

(1/72)n\*\*2

* + - * 1. Tube—pipe—hollow cylinder

Mirror8 is used to cover skips in resolution

R\*\*2=(X-A)\*\*(Y-B)\*\*2

If A==B then Y=A+/-(R\*\*2-(X-A)\*\*2)\*\*(1/2)

When X=Y then X=Y=A+/-[(R\*\*2)/2]\*\*(1/2)

Drawn in n/6 linear time

* + - * 1. Rod—solid cylinder

Shaded as 1/8 of a circle

Given x, find y

Shaded over a piece of pie

* + - * 1. Parallel Plates—two infinitely large sheets of conductors

Constant throughout area between the

* + 1. render--
       1. loops over the "space" and interrogates each point that it comes to
       2. loops from 0<=X<=offset and 0<=Y<=X
       3. this allows for only 1/8 of the space to be interrogated
       4. meanwhile another there is another large area that doesn’t need calculation “inside” the object
    2. inside--
       1. this method interprets whether or not the point is inside the object,
       2. alternatively it returns true for the point being outside of the plates
       3. this stems from the intention that this method tell when not to calculate

import numpy as np

import tools

from Layer import \*

import math

class Simulation:

def \_\_init\_\_(self,infinite=1,image="wire",size=200,magnitude=1.6\*(10\*\*-19)):

# image chooses the type of object that is being simulated

# pictures is a boolean for rendering 3 views of the field

self.s=size

self.s1=size+1

self.s2=size/2

self.s3=math.floor(size/3)

self.offset=(size/2)+1

self.magnitude=magnitude #this is voltage difference for plates

self.infinite=infinite

self.image=image

self.layer = self.draw(image)

self.top=Layer(self.s)

self.front=Layer(self.s)

self.side=Layer(self.s)

def interrogate(self,x,y,z,bloat=0):

x1=x+self.offset

y1=y+self.offset

v=tools.blank() # this will collect the vector of the E-field

if self.inside(x,y,z):

return tools.blank()

elif self.image=="plates":

a=self.s3

b=self.s1-self.s3

if a<=x<=b:

return np.array(1,0,0)\*(self.magnitude/(b-a))

elif self.infinite:

# use (lambda)/(2piE0R)

c= self.magnitude/(2\*tools.pi()\*tools.E0())

if x1<self.s3:

r= math.sqrt((self.s3-x)\*\*2+(self.s3-y)\*\*2)

else:

r= (self.s3-y)

v=tools.unitrad(np.array((x,y,0)))

return v \* float(c/r)

elif bloat:

if x<self.s3:

delta=x1-y1

side=self.s1-2\*self.s3

short=side-delta

c=math.ceil(delta/2)

delta2=math.floor(delta/2)

A=side\*\*2

A1=short\*short

A2=delta\*side+(.5\*delta\*delta)

## position of cetroid in a trapazoid

h=((side+(2\*short))/(3\*(short+side)))\*delta

cornerx=self.s3+h

cornery=self.s1-cornerx

y2=0.5\*(self.s3+cornery)

x2=0.5\*(self.s1-self.s3+cornerx)

c2x=round(0.5\*(cornerx+x2))

c2y=round(0.5\*(cornery+y2))

C1=self.offset+c

v1=tools.field(C1,C1,0,x1,y1,z,(A1/A)\*self.magnitude)

v2=tools.field(c2x,c2y,0,x1,y1,z,(A2/A)\*self.magnitude)

return (v1+v2)

if self.s3<=x1<=self.offset and y<=self.s3:

delta=x1-self.s3

g=2\*x1-self.s3

height=self.s1-(2\*self.s3)

A=height\*\*2

A1=(2\*(x1-self.s3)\*height)

A2=A-A1

i=x1

j=self.offset

k=0

v1=tools.field(i,j,k,x1,y1,z,(A1/A)\*self.magnitude)

CentroidX2=round(.5\*(g+self.s1-self.s3))

v2=tools.field(CentroidX2,j,k,x1,y1,z,(A2/A)\*self.magnitude)

return (v1+v2)

def draw(self,image):

layer=Layer(self.s1)

self.count=0

self.square=0

if image=="wire":

layer.insert(self.offset,self.offset,self.magnitude)

elif image=="box":

# draws hollow square with side= size/3

# linear n/6

self.square=1

x=(self.s3)

while x<=self.offset:

layer.mirror8(x,self.s3,self.magnitude)

self.count+=8

x+=1

elif image=="block":

# draws solid block

# draws smaller box just inside last box

# repeats until fully shaded

# linear (1/72)(x^2+x)

self.square=1

x=(self.s/3)

while x<=self.offset:

y=x

while y<=self.offset:

layer.mirror8(x,y,self.magnitude)

self.count+=8

y+=1

x+=1

elif image=="tube":

# draws a circle with radius r about point a

# linear n/12 runtime

x=float(self.s3)

a=float(self.offset)

r=a-x

while x<=a:

g=(x-a)\*\*2

y=round(a+math.sqrt(r\*\*2-g))

layer.mirror8(x,y,self.magnitude)

layer.mirror8(x,y-1,self.magnitude)

self.count+=16

x+=1

elif image=="rod":

# draws a solid cylinder

## runs at (pi\*n^2)/72

x=float(self.s3)

a=float(self.offset)

r=a-x

while x<=a:

g=(x-a)\*\*2

y=round(a-math.sqrt(r\*\*2-g))

layer.mirror8(x,y,self.magnitude)

self.count+=8

while y<=a:

layer.mirror8(x,y,self.magnitude)

self.count+=8

y+=1

x+=1

elif image=="plates":

x=self.s3

y=0

while y<self.offset:

layer.mirror4(x,y,self.magnitude)

y+=1

#layer.show()

self.count-=7

return layer

def render(self):

i=-self.offset

k=i+self.s3

while i<=0:

x=i+self.offset

j=-self.offset

while j<=i:

y=j+self.offset

temp=self.interrogate(i,j,0,self.square)

temp=tools.magnitude(temp)

self.top.mirror8(x,y,temp)

if self.infinite==0:

temp=self.interrogate(0,i,j,1)

temp=tools.magnitude(temp)

self.side.mirror8(x,y,temp)

# temp=self.interrogate(i,0,j,1)

# temp=tools.magnitude(temp)

# self.front.mirror4(x,y,temp)

j+=1

i+=1

self.side.show()

v1=self.top.space[self.offset,:]

v2=self.top.space[:,self.offset]

if self.infinite:

i=0

while i<self.s1:

self.side.space[:,i]=v1

self.front.space[i,:]=v2

i+=1

self.top.show()

self.side.show()

if self.infinite:

self.front.show()

def inside(self,x,y,z):

x1=x+self.offset

y1=y+self.offset

true=1

false=0

a=self.offset-self.s3

if self.infinite==0:

if abs(z)>a:

return false

if self.image=="tube":

r=float(self.offset)-float(self.s3)

if math.sqrt(x\*\*2+y\*\*2)<=r:

return true

elif self.image=="box":

if x1>=self.s3:

if y1>=self.s3:

return true

elif self.image=="plates":

if x1>=self.s3:

temp=self.s1-self.s3

if x1<=temp:

return true

elif self.layer.space[x1,y1]:

return true

else:

return false