Homework 8

Abhigyan Chattopadhyay
ME19B001

October 31, 2019

Plotting Extrema and Inflection Points

October 29, 2019

1 Homework 8 - Q1

We're going to plot a function, $f(x) = x^3 - 20x$, along with it's minima, maxima and inflection points.

First, we set the variable to be `x`

[1]:
```
var('x')
```

[1]:
```
x
```

Now, we make our function, $f(x) = x^3 - 20x$:

[2]:
```
f(x)=x^3-20*x
```

Next, we find the minima by solving $f'(x) = 0$...

[3]:
```
t = solve(diff(f(x),x)==0,x)
```

...and then map the actual values to a list, so that we can access the number and not just the expression.

[4]:
```
exes = list(map(lambda a: a.rhs(),t))
```

Now, we solve the equation $f''(x) = 0$ to find the points of inflection, and map those values to a list.

[5]:
```
g=list(map(lambda t: t.rhs(),solve(diff(diff(f,x),x)==0,x)))
```

Next, we plot two dotted lines that end at the maxima and minima points

First, the maxima:

[6]:
```
p1=line([(exes[0],0),(exes[0],f(exes[0]))],color='black',linestyle='--',legend_label='Maxima')
p2=line([(0,f(exes[0])),(exes[0],f(exes[0]))],color='black',linestyle='--')
```

Then the minima:

[7]:
```
p3=line([(exes[1],0),(exes[1],f(exes[1]))],color='green',linestyle='--',legend_label='Minima')
p4=line([(0,f(exes[1])),(exes[1],f(exes[1]))],color='green',linestyle='--')
```

Now, the actual function:

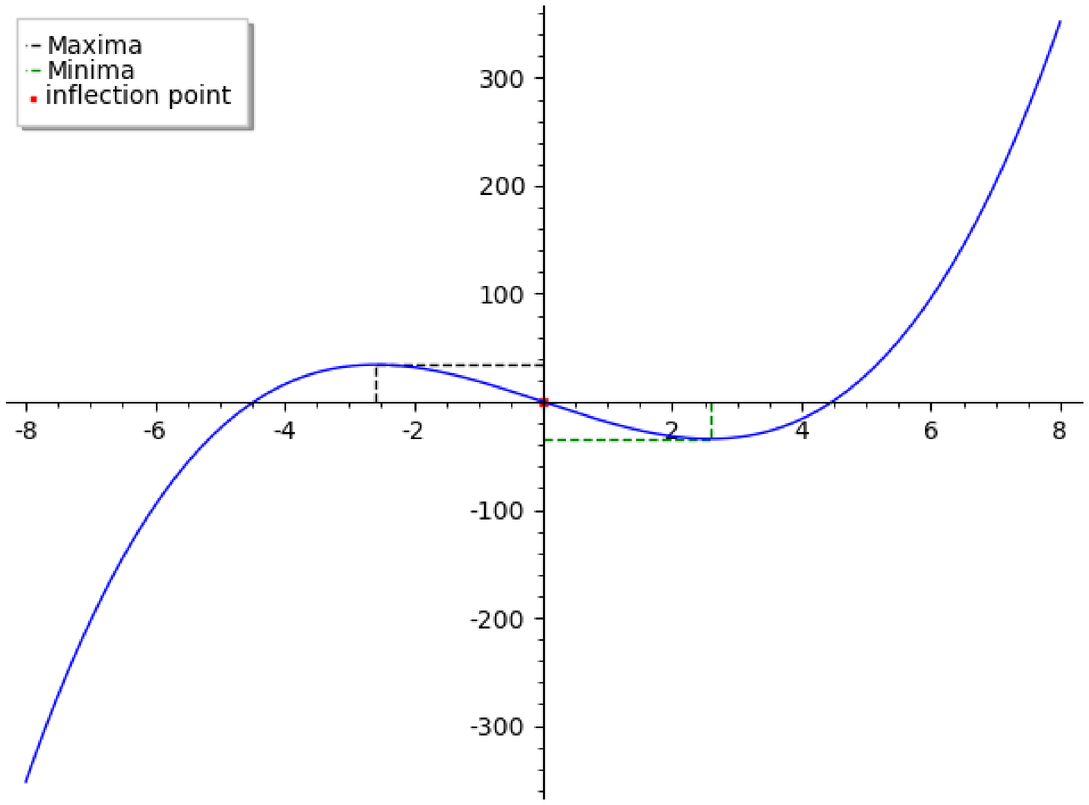
[8]:
```
p5=plot(f(x),(x,-8,8))
```

Now, the inflection point:

[9]:
```
p6=point((g[0],0),color='red',marker='x',legend_label='inflection point')
```

And, finally, we combine all the plots and plot them together in one nice graph!

[10]:
```
z = p1+p2+p3+p4+p5+p6
z.show()
```



Limits from other subjects

October 30, 2019

1 Limit of a function from Math:

This one is considerably short, and it's clear and concise unlike the other ones .

Either way, I'll explain what I've done like every time.

Basically, I'm going to take a multivariable limit using SageMath, so I need to use two variables: x, y

[1]:
```
var('x,y')
```

[1]:
```
(x, y)
```

Now, we shall input a function: $f(x, y) = \frac{e^y \sin(x)}{x}$...

[2]:
```
f(x,y) = e^y*sin(x)/x
```

...which SageMath will happily render in LaTeX for us

[3]:
```
f.show()
```

```
(x, y) |--> e^y*sin(x)/x
```

Now, we evaluate the limit as $(x, y) \to (0, 0)$

[4]:
```
f.limit(y=0).limit(x=0).show()
```

```
(x, y) |--> 1
```

Thus, the limit of $f(x, y)$ is 1.

Yay! Now we can practice math without needing any answer keys to validate our answers!

This question was taken from the Exercise Set 1 of the course MA1101, offered in Jul-Nov 2019, at IITM.

Solving Graphically

October 31, 2019

1 Solving for roots Graphically and Numerically:

Now, we will find the roots of an equation numerically as well as graphically, using SageMath 8.8.

Setting the variable as `x`:

```
[1]: var('x')
```

```
[1]: x
```

Now, creating the function $f(x) = -x + \cos(x)$

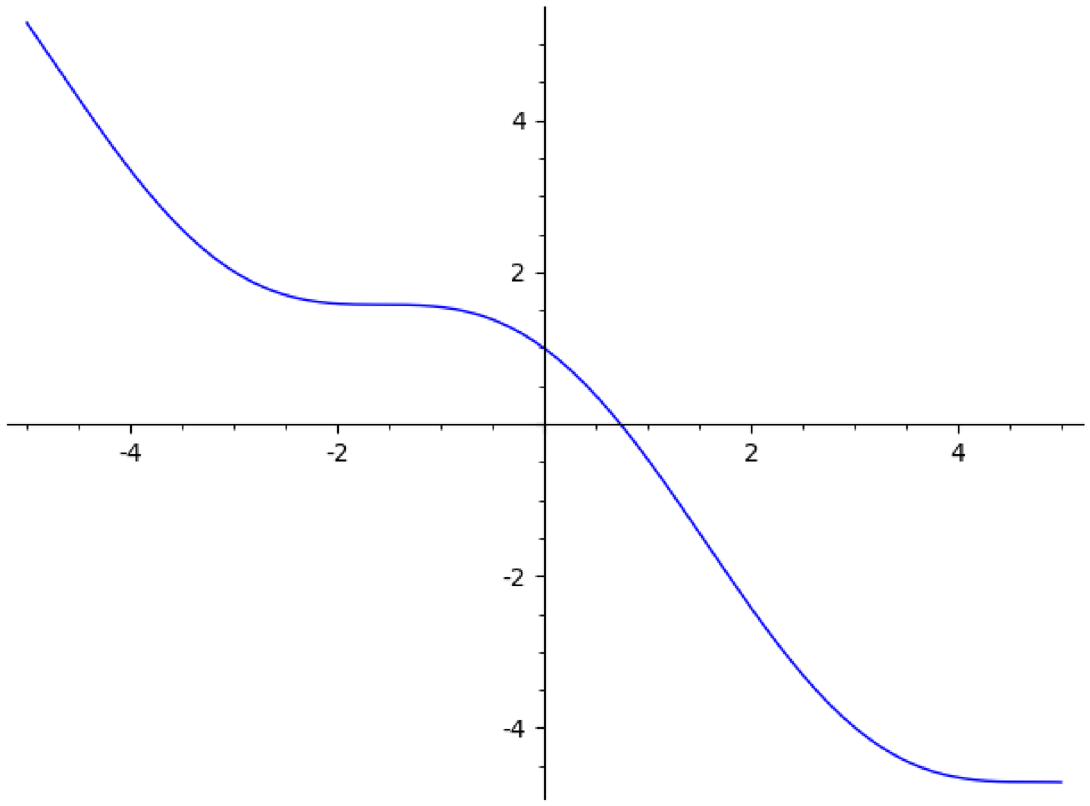
```
[2]: f(x)=-x+cos(x)
     f(x).show()
```

```
-x + cos(x)
```

Now, plotting the function in the range of $[-5, 5]$

```
[3]: plot(f(x),(x,-5,5))
```

```
[3]:
```



Finding the roots, using some numerical methods in SageMath (the find_root function):

(To be exact, we are finding roots in the range $(0, 1)$, since it looks like the root of the above equation is in that range from the graph above)

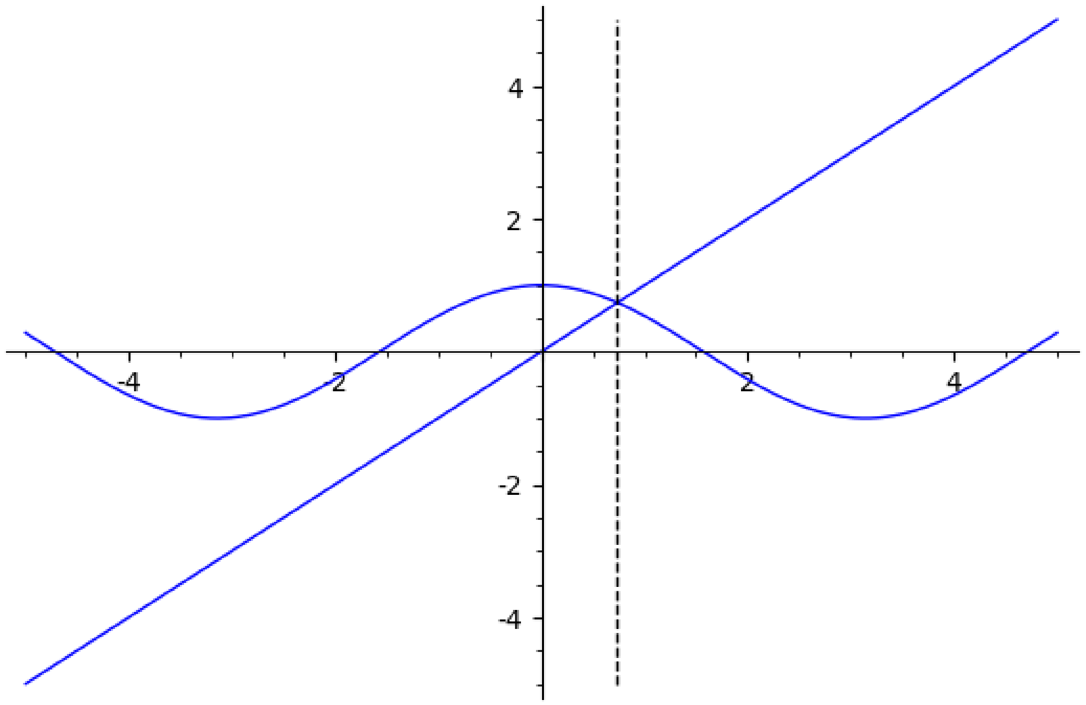
[4]:
```
root = f.find_root(0,1)
print(root)
```

```
0.7390851332151559
```

Now, we're plotting the functions $y = x$ and $y = \cos(x)$, so as to see where they intersect, and we also plot the line $x =$ root, which we evaluated in the previous part, so that we can check whether the two functions meet along the same line.

[5]:
```
p1 = plot(x,(x,-5,5))
p2 = plot(cos(x),(x,-5,5))
p3 = line([(root,-5),(root,5)],color='black',linestyle='--')
```

[6]:
```
z=p1+p2+p3
z.show()
```



Thus, we see that the two curves meet at the line where the original function was zero.

This proves that the solution of the equation $-x+\cos(x)=0$ is at the intersection point of the two functions: $y=x$ and $y=\cos(x)$.

Sierpinski Triangles

October 31, 2019

1 Homework 8 - Q4

To begin with, I've put a little playable number just below this cell, called depth, which will enable anyone using this Jupyter Notebook to change the depth to which the Sierpinski Triangles are rendered.

To meet the exact requirements of this question, I have used `depth = 4`. Feel free to try other variants of this code, which you can get from my GitHub.

[1]:
```
depth = 4
```

Our first step is to create a triangle boundary inside which we will plot all the other triangles.

We're doing that with 3 points, and since we want to later plot over the same plot, we're keeping it inside an list, p, to which we will keep adding all our other plots later on too.

Also, since we don't want the triangle to be filled, we're using the

[2]:
```
A = [0,0]
B = [10/2,10]
C = [10,0]
p=[]
p.append(polygon([A,B,C],fill=False))
```

Now, we need to find a lot of mid points, so might as well create a function `mid(A,B)` to find the mid point of the points `A` and `B`:

[3]:
```
def mid(A,B):
    return [(A[0]+B[0])/2,(A[1]+B[1])/2]
```

Now, we want to plot another triangle with the mid points of the previous one. So, we do the following:

[4]:
```
p.append(polygon([mid(A,B),mid(B,C),mid(C,A)],fill=False))
```

Now, the actual fun starts… (Or at least that's what I feel, cuz since I had to devise this one myself, it felt a lot harder to me than it probably looks to you)

We're basically going to create a list `todolist` in which we will store the points of the triangle inside which we want to draw triangles in the next couple of iterations.

Next, we're creating a function called `gothrough`, which does precisely that... It goes through and plots triangles inside the supplied triangle's coordinates.

The coordinates given as a 3×2 list of coordinates, like this:

$\begin{bmatrix} A_x & A_y \end{bmatrix}$

$\begin{bmatrix} B_x & B_y \end{bmatrix}$

$\begin{bmatrix} C_x & C_y \end{bmatrix}$

Now, we have to also add these to the same plot that we were plotting to earlier, for which we use the next parameter, `plots`, into which we append the new plots we make.

Now, the logic for the plotting is simple:

Find the mid-points of the triangle, then plot the sub-sub-triangles formed by joining the mid points of the lower-left sub-triangle. This is followed by the same for the upper sub-triangle and then the lower-right sub-triangle.

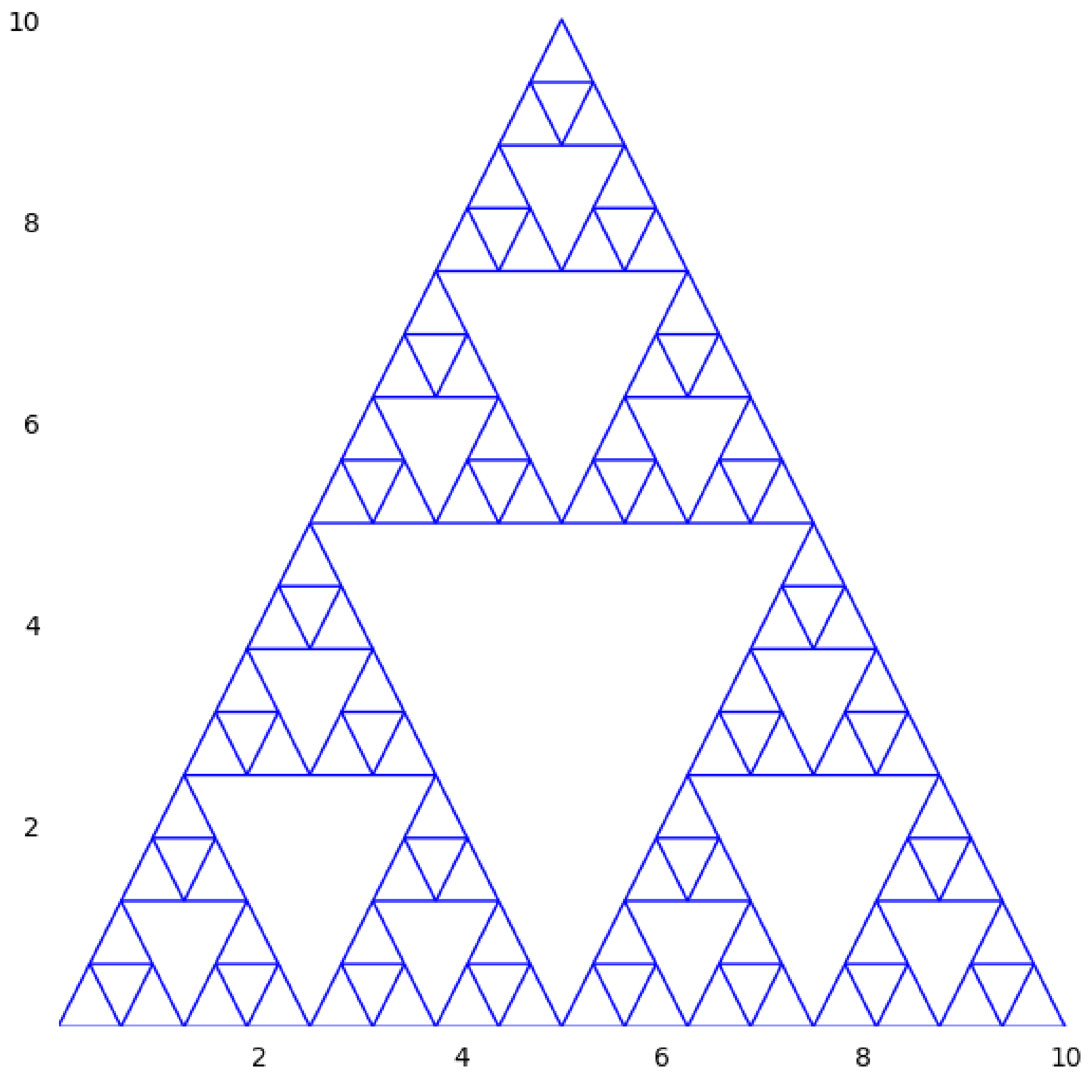
Basically, we're moving from one sub-triangle to another sub-triangle and plotting the sub-sub triangles in each of them, while at the same time, storing the locations of the sub-triangles in our `todolist` so that we can perform the same task in the sub-triangles later...

Go through the code and try to figure out what is happening in case this logic looks weird or is not explanatory enough.

[5]:
```
todolist=[]
def gothrough(P,plots):
    A = P[0]
    B = P[1]
    C = P[2]
    D = mid(A,B)
    E = mid(C,B)
    F = mid(A,C)
    plots.append(polygon([mid(A,D),mid(F,D),mid(A,F)],fill=False))
    todolist.append([A,D,F])
    plots.append(polygon([mid(B,D),mid(B,E),mid(D,E)],fill=False))
    todolist.append([D,B,E])
    plots.append(polygon([mid(F,E),mid(C,E),mid(C,F)],fill=False))
    todolist.append([F,E,C])
```

Now, we're trying to figure out what happens when we choose a depth less than or equal to 2, since this function is generalised only when the depth is more than 2. In the other cases, we have to hard code it, which is fairly alright, since, recursive functions generally have the lowest one or two cases defined from before (e.g. Fibbonacci Numbers and Factorial...)

[6]:
```
tot=0
for i in range(depth):
    tot+=3**i
if depth==1:
    t =sum(p)
    t.axes_color('white')
    t.show()
elif depth==2:
    gothrough([A,B,C],p)
    t=sum(p)
    t.axes_color('white')
    t.show()
elif depth>2:
    gothrough([A,B,C],p)
    for i in range((tot-4)/3):
        gothrough(todolist[i],p)
    t = sum(p)
    t.axes_color('white')
    show(t, figsize=8)
print("Behold the Sierpinski Triangle Fractal!\n(till depth",depth,")")
```



```
Behold the Sierpinski Triangle Fractal!
(till depth 4 )
```

Random Points

October 31, 2019

1 Homework 8 - Q7

We will now generate some random points, using a Points class that I wrote.

We will also use the `random` class that is made available in Python.

[1]:
```
from Points import Point
import random as rd
```

Now, we will choose how many points to plot: (Try out with different numbers)

[2]:
```
how_many = 5
```

Next, we will actually generate some random points, using the function `random.randint(0,10)`, which will basically enable us to create a random number in the range (0,9). (10 is excluded)

We will append those points to a list, and then print that list

[3]:
```
t = []
for i in range(how_many):
    if i==0:
        t.append(Point([rd.randint(0,10),rd.randint(0,10)]))
    else:
        t.append(Point([rd.randint(1,10),rd.randint(1,10)])+t[i-1])

for i in t:
    print(i)
```

```
[8,4]
[13,8]
[19,17]
[27,19]
[33,25]
```

For the next step, we need to create a list of variables. For this, I took some help from here.

In this process, we create `n` variables to fit an n^{th} degree equation, given n points.

[4]:
```
s = list(var('a_%d' % i) for i in range(how_many))
```

Next, we want to create Sage expressions that will equate the value of the function evaluated at a certain `x` with the corresponding value of `y`.

```
[5]: expr = []
     for i in range(how_many):
         ex = t[i][0]
         y = t[i][1]
         expr_str = ""
         for j in range(how_many):
             if j==(how_many-1):
                 expr_str+="s[%d]*ex**(%d)-y==0" % (j,j)
             else:
                 expr_str+="s[%d]*ex**(%d)+" % (j,j)
         expr.append(eval(expr_str))
```

We now have all our expressions in `expr`, and all our variables in `s`. Then all we need to do is solve the equation, and then take the `rhs()` of each of them, and send them to a new list.

```
[6]: Soln = solve(expr,s)
     soln_expr = []
     for i in Soln[0]:
         soln_expr.append(i.rhs())
```

Next, we need to create the plot, which also contains numerical parts, which is why we need to evaluate them using the `eval()` command. So, that's what we do:

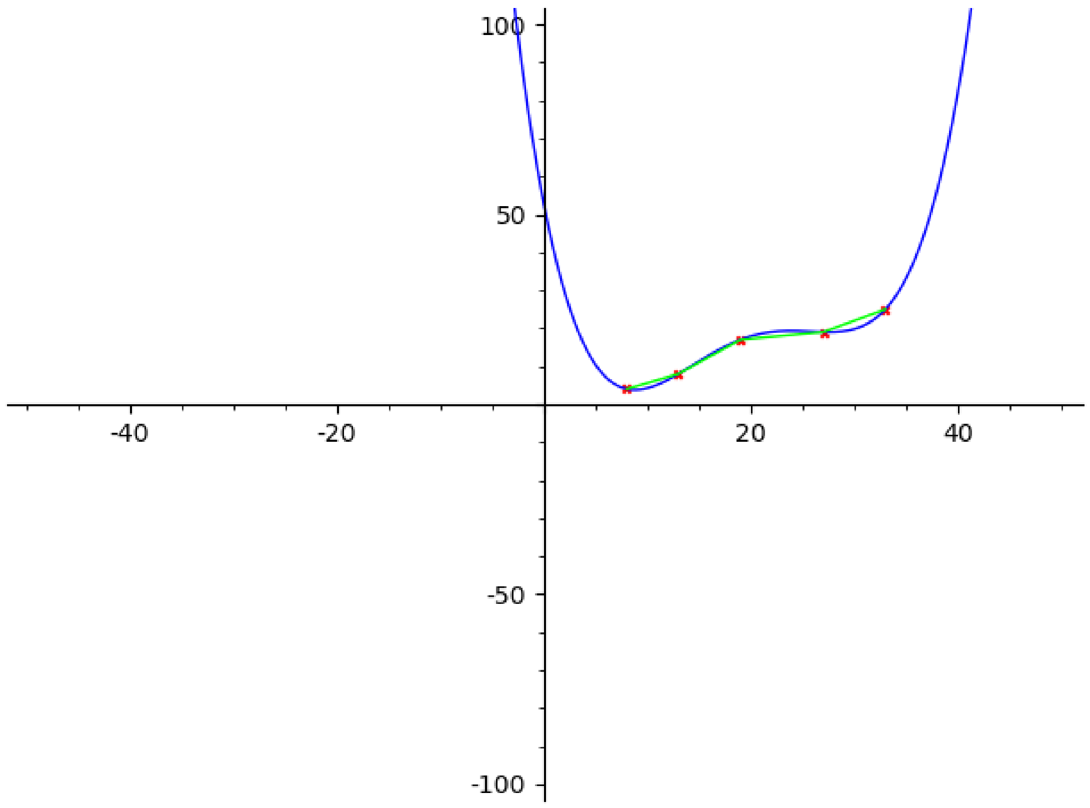
```
[7]: plot_str=""
     for i in range(len(soln_expr)):
         if i==len(soln_expr)-1:
             plot_str += str(float(soln_expr[i]))+"*x**"+str(i)
         else:
             plot_str += str(float(soln_expr[i]))+"*x**"+str(i)+"+0"

     plot_expr = eval(plot_str)
     plot_expr.show()
```

```
0.0006076555023923445*x^4 - 0.048761449077238554*x^3 + 1.3474962406015039*x^2 - 14.03807108680
```

Now, we have our final `plot_expr`, which we will plot using the plot command. We also plot the random points we had generated, and the line joining the points.

This is the final result:

```
[8]: p1 = plot(plot_expr,(x,-50,50),ymin=-100,ymax=100)
     p2 = line(t,rgbcolor=(0,1,0))
     p3 = point(t,rgbcolor=(1,0,0),marker='x')
     z = p3+p1+p2
     z.show()
```


100
50
-50
-100
-40
-20
20
40

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Oct 31 17:06:30 2019

@author: abhigyan
"""
class Point:
    def __init__(self, lst):
        self._lst=lst

    def __getitem__(self, item):
        return self._lst[item]

    def __add__(self, another):
        return Point([self[0]+another[0],self[1]+another[1]])

    def __truediv__(self, number):
        return Point([self[0]/number,self[1]/number])

    def __mul__(self, number):
        return Point([self[0]*number,self[1]*number])

    def __sub__(self, another):
        return Point([self[0]-another[0],self[1]-another[1]])

    def __str__(self):
        return "["+str(self[0])+","+str(self[1])+"]"
```

Pythagoras Tree

November 24, 2019

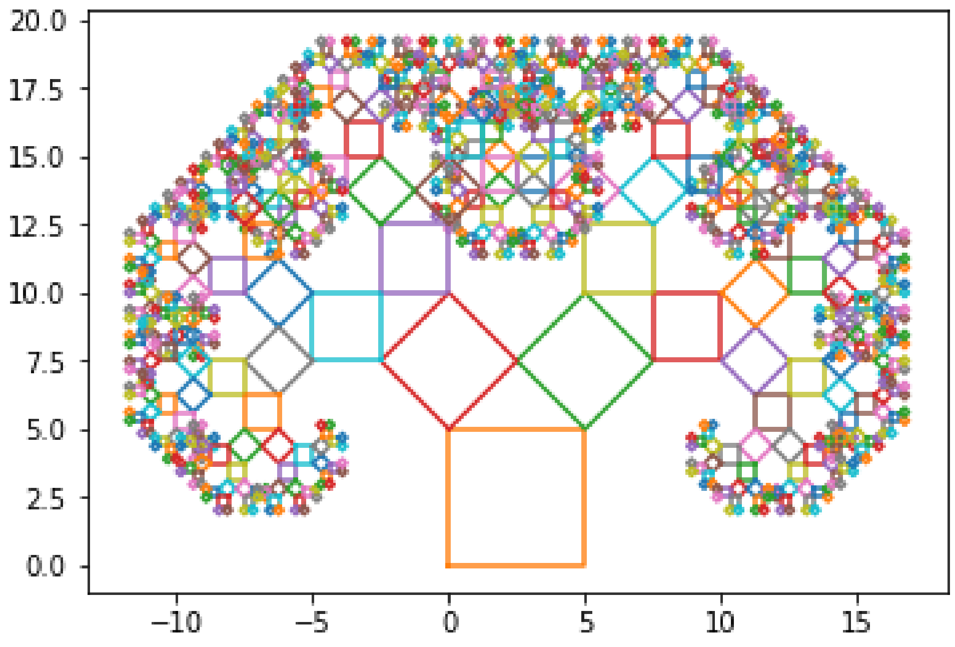
1 Pythagoras Tree

This code below plots Pythagoras Tree.

(It's been thrown in at the last minute, cuz I realized the method to draw it, and then added it in just before submission deadline cuz, why not? ;)

```
[4]: import math
     import matplotlib.pyplot as plt
     def pythagorasTree(x,y,side,theta,a,depth,plots):
         del1 = side*math.cos(theta)
         del2 = side*math.sin(theta)
         x1 = x + del1
         y1 = y + del2
         x2 = x + del1 - del2
         y2 = y + del1 + del2
         x3 = x - del2
         y3 = y + del1
         x4 = x3 + side*math.cos(theta+a)*math.cos(a)
         y4 = y3 + side*math.sin(theta+a)*math.cos(a)
         xs = [x,x1,x2,x3,x]
         ys = [y,y1,y2,y3,y]
         plots+=plt.plot(xs,ys)
         if depth>1:
             pythagorasTree(x4,y4,side*math.cos(a),theta+a-math.pi/2,a,depth-1,plots)
             pythagorasTree(x3,y3,side*math.cos(a),theta+a,a,depth-1,plots)
     plots = plt.plot(0,0)
     pythagorasTree(0,0,5,0,math.pi/4,10,plots)
     plt.axis("scaled")
```

```
[4]: (-13.3125, 18.3125, -0.96875, 20.34375)
```


20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
−10
−5
0
5
10
15