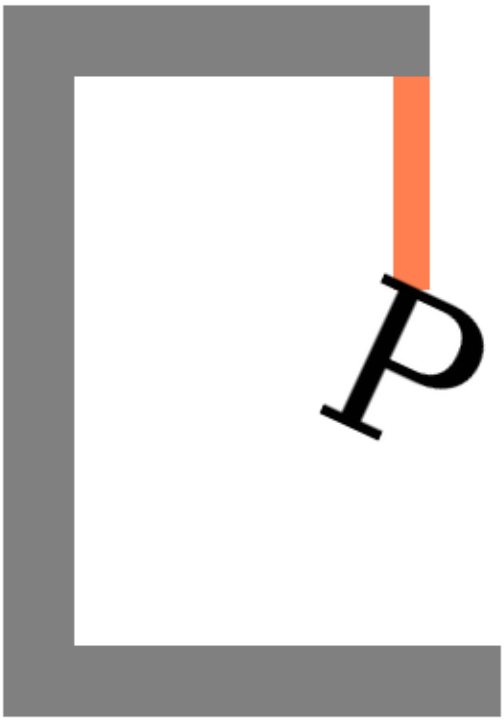
LIFE AFTER DEATH BY



also known as

How to solve it using Python

Kunal Gokhe

2024

Contents

The death-by-Python graphic **2**

Counting people from handshakes **4**

π by dartboard **6**

Scrambled eggs under gravity **9**

Hol(e)y polynomial doughnut **12**

Square root, the ancient way **15**

Graphics with granddaddy **19**

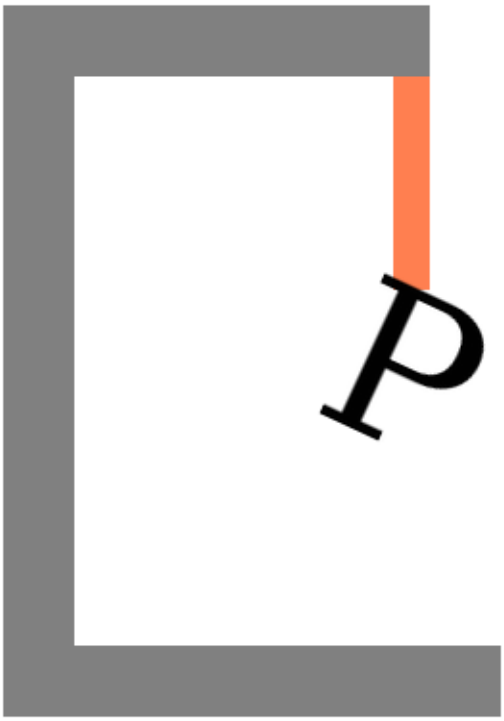
Kaprekar's 6174 **23**

Long live the Queens! **28**

The death-by-Python graphic

P R O B L E M

How might one re-create the death-by-Python title page graphic for this book as faithfully as possible?



A P P R O A C H

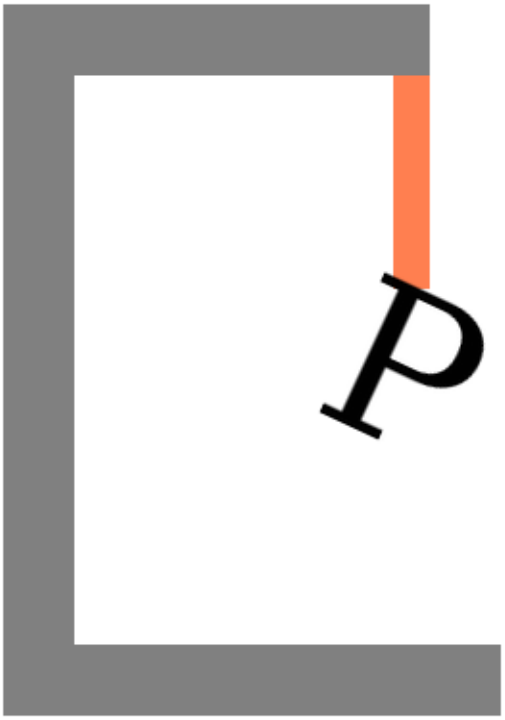
To create the graphic death by Python with letter "P" we first initialize the figure of square axis and add patches to for the grey beam by providing coordinates of corners and the vertical noose is also can be added with rectangle patch by assigning different color and letter P is added using text function in python with some angle of rotation.

S O L U T I O N

```
# Import libraries
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as pch
# Open figure
fig=plt.figure()
ax = fig.add_subplot(1,1,1)
plt.axis('square')
plt.axis([0,0.7,0,1])
# Add a grey polygon to the plot to form the background shape
ax.add_patch(pch.Polygon([[0,0],[0.7,0],[0.7,0.1],[0.1,0.1],[0.1,0.9]
```

```
                            ,[0.6,0.9],[0.6,1],[0,1],[0,0]],facecolor='grey'))
# Add a coral-colored smaller polygon on top of the grey background
ax.add_patch(pch.Polygon([[0.6,0.9],[0.6,0.6],[0.55,0.6],[0.55,0.9],[0.6,0.9]],
    facecolor='coral'))
ax.text(0.4,0.35,'P',fontsize=80,rotation=-25,fontfamily='serif')
ax.axis('off')
# Show the plot
plt.show()
```

OUTPUT



Counting people from handshakes

P R O B L E M

Image Courtesy: Wikipedia

A special interdisciplinary meeting is organized where a few computer scientists and some biolo- gists are invited. First, the two groups meet sepa- rately: Each person shakes hands with every other person within only her/his own group. In other words, computer scientists shake hands with com- puter scientists, while biologists shake hands with biologists. There are a total of 102 such hand- shakes. After that, all computer scientists shake hands with all biologists. There are a total of 108 such handshakes across the two groups. How many computer scientists attended the meeting?

A P P R O A C H

We will formulate the problem in such a way that it can be solve computationally, we have given that each total handshake within group is 102 and across groups is 108. let's n be the computer scientists and m be the biologists.

$$\binom{n}{2} + \binom{m}{2} = 102 \quad \text{within group handshakes, and} \tag{1}$$

$$nm = 108 \quad \text{across group handshakes} \tag{2}$$

whereas $\binom{n}{k} = \frac{n!}{k!(n-k)!}$
we will solve the equation by simplifying the expressions,

$$\frac{n(n-1)}{2} + \frac{m(m-1)}{2} = 102 \tag{3}$$

$$m = \frac{108}{n} \tag{4}$$

substitute the expression of m in the (3), we have

$$\frac{n(n-1)}{2} + \frac{\frac{108}{n}\left(\frac{108}{n} - 1\right)}{2} = 102$$

$$\frac{n^2 - n}{2} + \frac{\left(\frac{108^2}{n^2} - \frac{108}{n}\right)}{2} = 102$$

$$n^2 - n + \left(\frac{108^2}{n^2} - \frac{108}{n}\right) = 204$$

$$n^4 - n^3 + 108^2 - 108n = 204n^2$$

now, we have simplified expression

$$n^4 - n^3 - 204n^2 - 108n + 108^2 = 0$$

We will solve the above equation using root finding approach to get value of n, to find the root of the equation we first define the equation and then solve using fsolve function in scipy library.

SOLUTION

```
# Import Libraries
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import fsolve
# Define function
n = lambda m: 108/m
def comb(k):
    return k*(k-1)/2
f = lambda m: comb(n(m))+comb(m)-102
# Solve using fsolce
m=fsolve(f,100)
n=n(m)
# Display results
print(f'n={round(n[0])} and m={round(m[0])}')
```

OUTPUT

```
n=9 and m=12
```

By solving the problem using root finding approach we have $n = 9$ and $m = 12$ also the another solution is $m = 9$ and $n = 12$. So, possible values of $n = 9$ *or* 12. Therefore number of computer scientists attended the meeting is 9 and 12.

π by dartboard

P R O B L E M

The celebrated number π can be estimated through darts thrown at random at a simplistic dartboard (see figure). Simulate the experiment of throwing N random darts at this dartboard so that they fill up the dartboard with uniform density. Using this, estimate the value of π.

A P P R O A C H

To simulate the experiment for throwing the darts at the dartboard N time will gives us the approximate values of π as the number of experiment increase the values of π goes to its true values (Law of Large Number). To compute the approximate values of π we have to find the ratio of number of darts inside the circle to the total number of darts thrown times 4 will gives the value of π. The number of hits can be found out as

$$x^2 + y^2 \leq 1$$

The below python code show the simulation of throwing random darts.

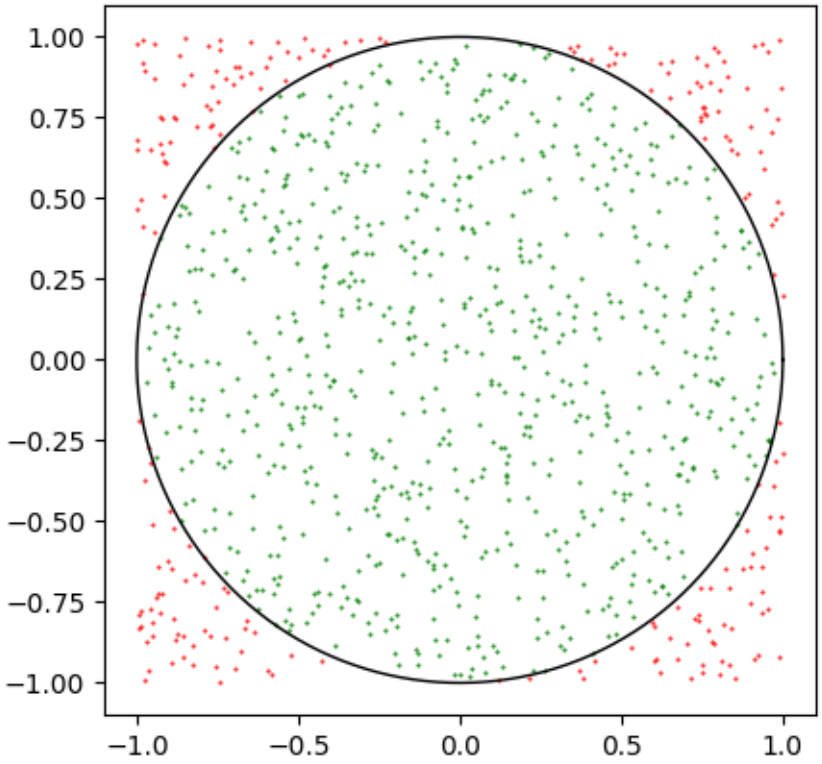
S O L U T I O N

```
# Import Libraries
import numpy as np
import random
import matplotlib.pyplot as plt
# number of darts
N=1000
# Generate position of darts
x=np.array(random.sample(range(-N,N),N))/N
y=np.array(random.sample(range(-N,N),N))/N
# plots circle
plt.plot(np.cos(2*np.pi*np.linspace(0,1,1000)),np.sin(2*np.pi*np.linspace
    (0,1,1000)),'k',linewidth=1)
# Compute distance from center
d=np.sqrt(x**2+y**2)
# Plot darts which is plotted inside circle as green
plt.scatter(x[d>1],y[d>1],marker='.',c='r',s=1)
plt.scatter(x[d<1],y[d<1],marker='.',c='g',s=1)
plt.axis([-1,1,-1,1])
plt.axis('square')
# Show the plot
plt.show()
print('Approximate values of pi is ',4*sum(d<1)/N)

# Simulate for multiple replications
val=[]
for N in range(100,10000,100):
    # Generate position of darts
    x=np.array(random.sample(range(-N,N),N))/N
    y=np.array(random.sample(range(-N,N),N))/N
    # Compute distance from center
    d=np.sqrt(x**2+y**2)
    val.append(4*sum(d<1)/N)
# Plot histogram
plt.hist(val,density=True)
```

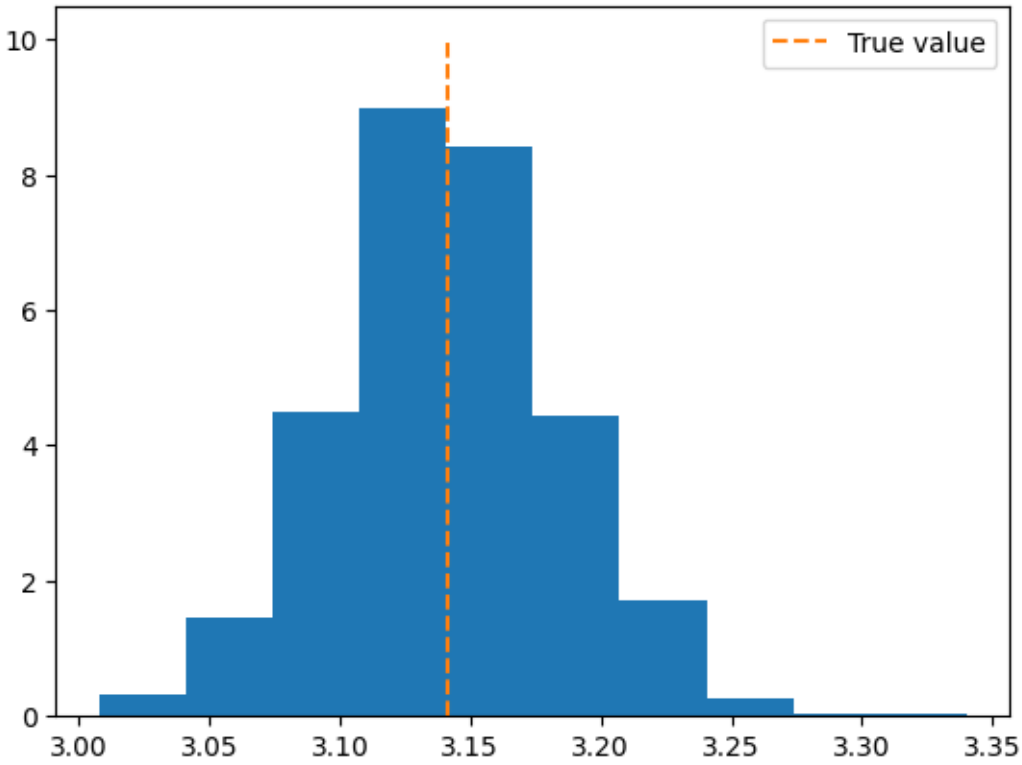
```
# Plot true values of pi
plt.plot([np.pi,np.pi],[0,20],'--')
plt.legend(['True'])
# Show the plot
plt.show()
```

OUTPUT

```
Approximate values of pi is  3.14
```



The below figures shows the distribution of the replication of the same experiment 1000 times, were we can see that the distribution is peak at around the true values of π. As the size of number of replication increase the mean of the distribution reaches toward the true values of π.


True value
10
8
6
4
2
0
3.00
3.05
3.10
3.15
3.20
3.25
3.30
3.35

Scrambled eggs under gravity

P R O B L E M

Image Courtesy: Wikipedia

A hypothetical comic-book planet with normal Newtonian gravity happens to have an atmosphere consisting of two entities, e and g, in the 1:2 proportion. The two entities have the same mass, and behave as ideal gases, and do not interact with each other in any way. Create a two-dimensional snapshot of this atmosphere, where one of the dimensions is the direction of gravity (i.e., the vertical direction), and the other one is any direction that is perpendicular to the vertical.

A P P R O A C H In this problem we are going to compute the vertical direction of the particle which is generated by exponential distribution of $\lambda = 1$. The density of an ideal gas under the gravity is given by

$$\rho(y) = \rho_0 \exp\left(\frac{-mg}{k_B T} y\right)$$

where as $\lambda = \dfrac{mg}{k_B T}$, which leaves us the expression for the PDF of the exponential distribution:

$$f(y) = \lambda \exp(-\lambda y), \quad y \geq 0, \ \lambda > 0$$

Now for the 2D plot we generate the position of the particle in x-direction using uniform distribution and generate the y-direction of particle using exponential distributed. we scatter the data on 2D plot and For the 3D plot we generate position of particle in x and y direction both using uniform distribution and generate the z-direction using exponential distribution. The below python code will show the two entities e and g represented by two color red and blue respectively.

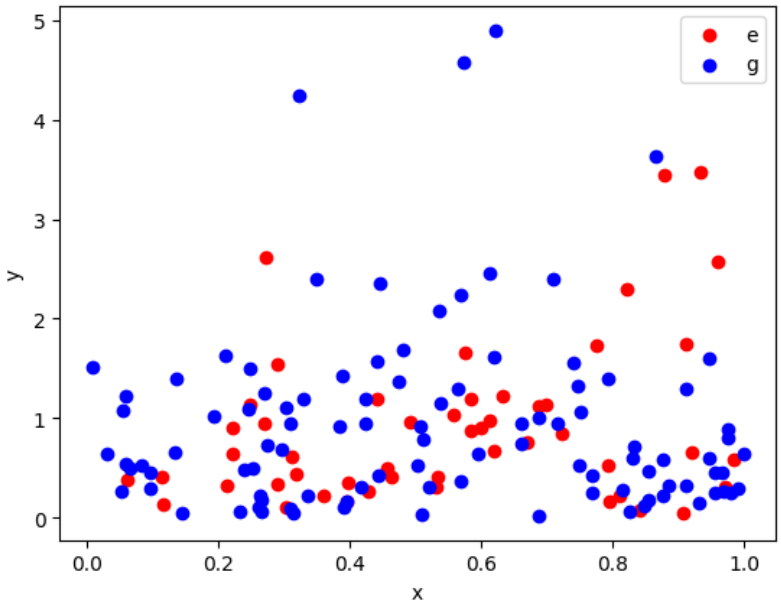
S O L U T I O N

```
# Import Libraries
import numpy as np
import matplotlib.pyplot as plt

# Define Parameters
lmda=1
Ne=50
Ng=2*Ne

# Plot 2D snapshot
# Generate y-direction data
ye=np.random.exponential(lmda,Ne)
yg=np.random.exponential(lmda,Ng)
# Generate x-direction data
x=np.random.uniform(0,1,3*Ne)
# Plot data
plt.scatter(x[0:Ne],ye,color='red')
```

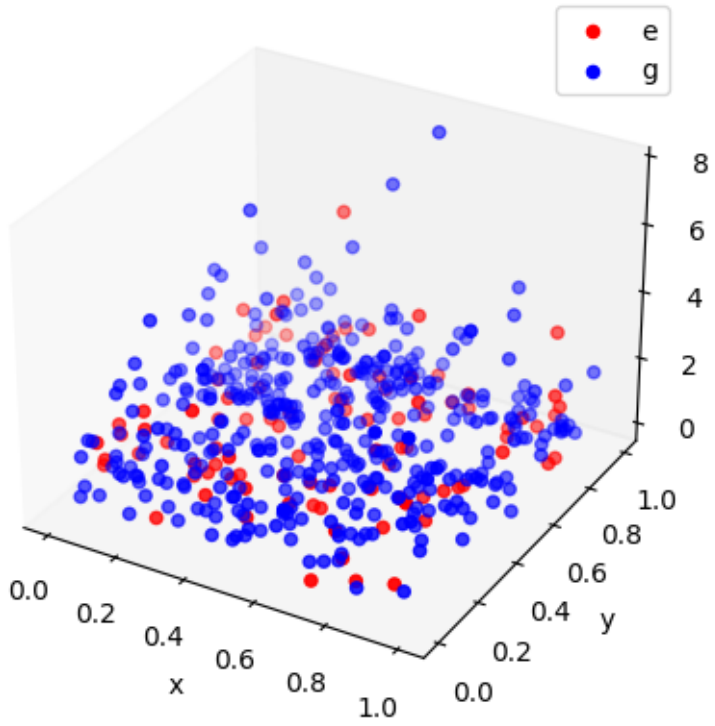
```
plt.scatter(x[Ne:],yg,color='blue')
plt.xlabel('x')
plt.ylabel('y')
plt.legend(['e','g'])
# Show plot
plt.show()

# Define Parameters
lmda=1
Ne=10
Ng=2*Ne
# Plot 3D snapshot
# Generate x and y-direction data
x=np.random.uniform(0,1,[3*Ne,3*Ne])
y=np.random.uniform(0,1,[3*Ne,3*Ne])
# Generate z-direction data
ze=np.random.exponential(lmda,[Ne,Ne])
zg=np.random.exponential(lmda,[Ng,Ng])
# Plot data
fig=plt.figure()
ax=plt.axes(projection='3d')
ax.scatter(x[0:Ne,0:Ne],y[0:Ne,0:Ne],ze,color='red')
ax.scatter(x[Ne:,Ne:],y[Ne:,Ne:],zg,color='blue')
ax.grid(False)
plt.xlabel('x')
plt.ylabel('y')
ax.set_zlabel('z')
plt.legend(['e','g'])
# Show plot
plt.show()
```

OUTPUT

The below figure shows the simulation of the generating the behaviours of two gases under the gravity which have different in proportion.

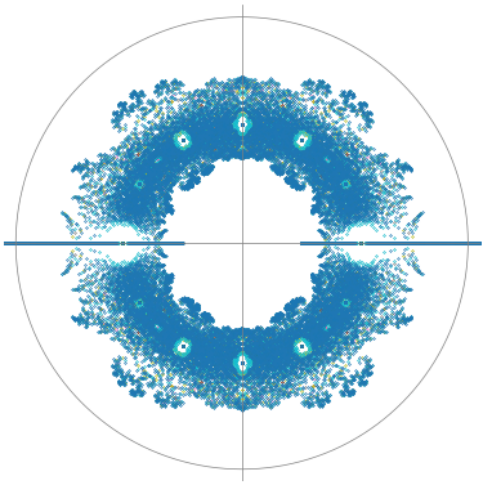


Similarly, for the 3-dimensional generated data shown below.



Hol(e)y polynomial doughnut

P R O B L E M



What is plotted in this figure are the complex roots of all polynomials with degree between 1 and 11 and coefficients = ±1. Polynomial roots have important roles to play in diverse domains such as filter design for signal processing (e.g., pole-zero plot), time series analysis (e.g., autoregressive models, unit root, etc.), theoretical computer science, statistical mechanics (e.g., Yang-Lee zeros), etc. A degree-m polynomial $p(x) = a_0 + a_1x + a_2x^2 + ... + a_mx^m$ has m zeros or roots, complex or real. Given any degree-m polynomial $p(x) = a_0 + a_1x + a_2x^2 + ... + a_mx^m$, find all its zeros/roots. Using this, produce a plot similar to the one on the left.

A P P R O A C H

Firstly, we have to find all the roots for the degree-m polynomial which has m roots, to find all the roots by solving $p(x) = 0$, happen to get the eigenvalues of $m \times m$ matrix. we have

$$H_p = \begin{bmatrix} -\frac{a_{m-1}}{a_m} & -\frac{a_{m-2}}{a_m} & ... & -\frac{a_1}{a_m} & -\frac{a_0}{a_m} \\ 1 & 0 & ... & 0 & 0 \\ : & : & ... & : & : \\ 0 & 0 & ... & 1 & 0 \end{bmatrix}$$

By computing the eigenvalues using numpy library of the above matrix we get the all the roots of the polynomial. we have takes the coefficients = ±1 for the polynomial and compute all the roots for all possible coefficients for degree between 1 and 11. after computing roots the imaginary values plotted on the y-axis and real values plotted on the x-axis which will give the Hol(e)y polynomial doughnut. The pseudo-code for the routine computeRoot is shown below.

Algorithm 1 Compute Roots for Given of Polynomial

Require: Integer m
Ensure: Arrays x and y containing real and imaginary parts of the roots.
  Initialize matrix $A \leftarrow \text{diag}(m-1, -1)$
  $n \leftarrow 2^{m+1}$ {Number of possible coefficient combinations}
  Initialize empty arrays $x \leftarrow [\,]$ and $y \leftarrow [\,]$
  for $i = 0$ to $n-1$ **do**
    Convert i to a binary string of length $m+1$
    Map the binary string to an array p where:

$$p_j \leftarrow 2 \times \text{binary value of the } j\text{-th bit} - 1$$

    Extract coefficients from p:

$$b \leftarrow \frac{p[1:]}{p[0]}$$

    Update the first row of A: $A[0,:] \leftarrow b$
    Compute the eigenvalues λ of matrix A
    Append $\text{Re}(\lambda)$ to x and $\text{Im}(\lambda)$ to y
  end for
  return x, y

Using the above algorithm we can create python code as given below.

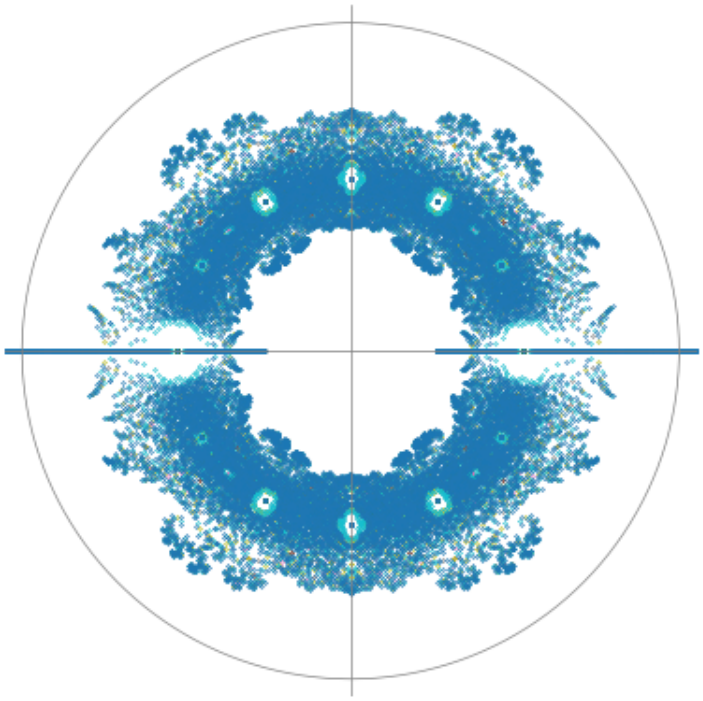
S O L U T I O N

```python
# Import Libraries
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm

# Compute root Fucntion
def computeRoot(m):
    # Create Hp Matrix
    A=np.diag([1]*(m-1),-1)
    # Number of possible combination of coefficients
    n=2**(m+1)
    x=[]
    y=[]
    # Compute root for each coefficients
    for i in range(n):
        # Create Hp matrix
        p=np.array([int(i)*2-1 for i in list('{:0{m}b}'.format(i,m=m+1))])
        b=p[:0:-1]/p[-1]
        A[0,:]=b
        # Compute eigenvalue (roots)
        eigval=np.linalg.eig(A).eigenvalues
        # Store real and imaginary roots
        x.append(np.real(eigval))
        y.append(np.imag(eigval))
    return np.array(x),np.array(y)

# Open figure
fig=plt.figure()
# Compute roots for degree 1 to 11
for m in range(1,12):
```

```
    x,y=computeRoot(m)
    # Plot roots
    plt.scatter([x,-x],[y,y],marker='.',s=0.1)
# Plot Axis and Circles
plt.plot([0,0],[-2,2],color='grey',linewidth=0.5)
plt.plot([-2,2],[0,0],color='grey',linewidth=0.5)
theta=np.linspace(0,2*np.pi,1001)
plt.plot(1.9*np.cos(theta),1.9*np.sin(theta),color='grey',linewidth=0.5)
plt.axis('square')
plt.axis([-2,2,-2,2])
plt.axis('off')
plt.show()
```

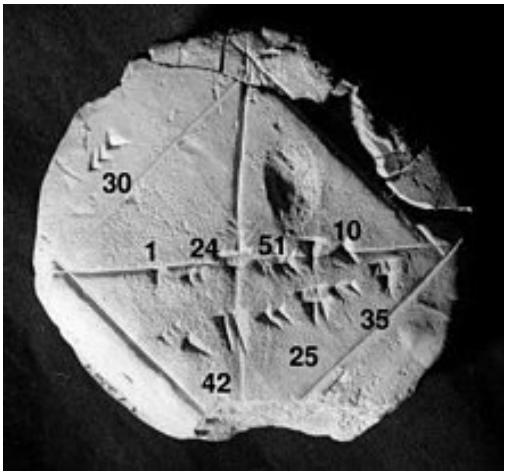
OUTPUT

The output below shows the scattered data of all the roots of degree between 1 and 11 assigning different color for all combinations of coefficient (± 1).



Square root, the ancient way

P R O B L E M



Babylonian tablet YBC 7289, circa 1800-1600 BC,showing approximate value of $\sqrt{2}$.

The Babylonian method for computing the square root of a positive real number S: Start with any arbitrary $x_0 > 0$, and iterate $(n = 0, 1, ...)$

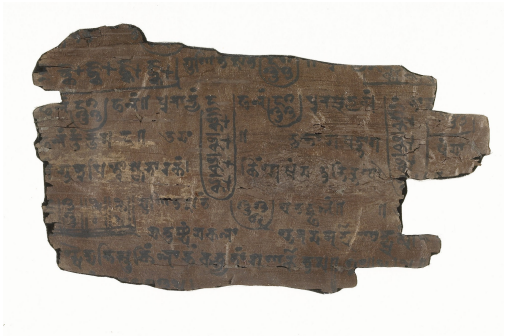
$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{S}{x_n}\right)$$

This method is also known as the *divide-and-average* method.

An Indian method for computing the square root of a positive real number S: Start with any arbitrary $x_0 > 0$, and iterate $(n = 0, 1, ...)$

$$a_n = \frac{S - x_n^2}{2x_n}$$

$$x_{n+1} = x_n + a_n - \frac{1}{2}\frac{a_n^2}{x_n + a_n}$$



A page from the Bakhshali manuscript. Date uncertain, but considered to be no later than 12th century. Figure 3 in this article shows another page that illustrates the calculation of square root. Image courtesy: Oxford University.

Let us remember that both methods are stated here in the modern mathematical language and notation, and not the way the ancients might have chosen to express them. For both methods, successive iterates x_n get closer and closer to $\sqrt{S}$ – eventually, but thankfully, reasonably quickly. The second method involves more computation at every iteration, but converges at a much faster rate than the first (here is a proof) – that is, in absence of finite-precision arithmetic artifacts.

Some perspective should help here: The ancients probably used these methods using integer arithmetic and for hand computation, and their intent was probably to obtain what we would call rational approximations to square roots. In any case, they neither had the benefit of R, nor had to cope up with the quirks of the modern-day finite-precision arithmetic.

A P P R O A C H

To find the square root using the two ancient way using numerically we have to convert the expression in such a way that it can be solved using fixed point iteration approach, which has a form of $x = f(x)$.
For the Babylonian method,

$$f(x) = \frac{1}{2}\left(x_n + \frac{S}{x_n}\right)$$

And for the Indian method,

$$f(x) = x_n + a_n - \frac{1}{2}\frac{a_n^2}{x_n + a_n}$$

The pseudo-code for the fixed point iteration method is described below. Now, we solve both

Algorithm 2 Fixed-Point Iteration

Require: Function f, initial guess x_{old}, tolerance tol, maximum iterations maxIter
Ensure: Approximate solution x_{new}, number of iterations itr, list of iterates x

Initialize iteration counter: $itr \leftarrow 0$
Initialize an empty list to store iterates: $x \leftarrow []$
while True **do**
    Append current guess to the list of iterates: $x \leftarrow x \cup \{x_{\text{old}}\}$
    Compute new approximation: $x_{\text{new}} \leftarrow f(x_{\text{old}})$
    if $|x_{\text{new}} - x_{\text{old}}| <$ tol **or** $itr >$ maxIter **then**
        Break the loop
    end if
    Update the old value: $x_{\text{old}} \leftarrow x_{\text{new}}$
    Increment iteration counter: $itr \leftarrow itr + 1$
end while
return x_{new}, itr, x

the ancient method and find the roots for 125348 which came to be 354.045.

S O L U T I O N

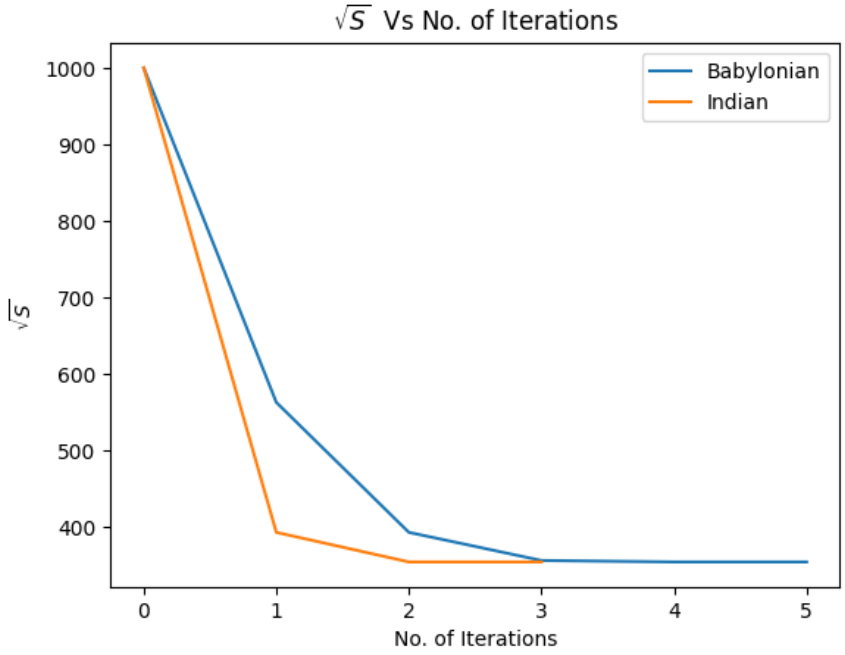
```python
# Import Libraries
import numpy as np
import matplotlib.pyplot as plt

# Fixed point iteration Function
def fixedPoint(f,xold,tol,maxIter):
    itr=0
    x=[]
    while True:
        x.append(xold)
        xnew=f(xold)
        if abs(xnew-xold)<tol or itr>maxIter:
            break
        xold=xnew
```

```
        itr+=1
    return xnew,itr,x

# Babylonian Method
# Define function
S=125348
f=lambda x: (x+S/x)/2
# Define parameters
xold=1000
maxIter=100
tol=1e-6
# Compute root
xnew,itr,x=fixedPoint(f,xold,tol,maxIter)
# Print and plot solution
print(f'Using babylonian Method sqrt{S} is {xnew}')
plt.plot(x)

# Indian Method
# Define function
S=125348
a=lambda x:(S-x**2)/(2*x)
f1=lambda x: x+a(x)-0.5*a(x)**2/(x+a(x))
# Define Parameters
xold=1000
maxIter=100
tol=1e-6
# Compute root
xnew,itr,x=fixedPoint(f1,xold,tol,maxIter)
# Print and plot solution
print(f'Using Indian Method sqrt{S} is {xnew}')
plt.plot(x)
plt.xlabel('No. of Iterations')
plt.ylabel(r'$\sqrt{S}$')
plt.legend(['Babylonian','Indian'])
plt.title(r'$\sqrt{S}$  Vs No. of Iterations')
# Show plot
plt.show()
```

OUTPUT

From the above computation we get the square root for $\sqrt{125348} = 354.04519$. To check which methods converge at faster rate we can plot the solution both method get at each step as shown in below figure. We can see that the convergence of the Indian method is faster than the Babylonian method. It takes 3 iteration to get the solution with in the tolerance of 10^{-6} but here the computation is more as compared to Babylonian method.

```
Using babylonian Method sqrt(125348) is 354.04519485512014
Using Indian Method sqrt(125348) is 354.04519485512014
```


$\sqrt{S}$ Vs No. of Iterations
Babylonian
Indian
1000
900
800
700
600
500
400
$\sqrt{S}$
0
1
2
3
4
5
No. of Iterations

Graphics with granddaddy

P R O B L E M

Image courtesy: Wikipedia

"We might call [the Euclidean algorithm] the granddaddy of all algorithms, because it is the oldest nontrivial algorithm that has survived to the present day." – Donald Knuth, The Art of Computer Programming, Vol. 2: Seminumerical Algorithms, 2nd edition (1981), p. 318.

This exercise is about two interesting graphics that can be produced with granddaddy: (A) coprime/relative prime pairs (a, b), and (B) the number of steps/iterations of the algorithm for each integer pair (a, b). In either case, a, b are both between 0 and some integer n. The resulting (n + 1) × (n + 1) matrices have interesting structure that can visualized in the form of color-coded images.

A P P R O A C H

In this we are going to compute coprime pairs and the number of steps of the algorithm for each integer pair (a, b) and plot the data on 2D plane to get graphics. Also, we will compute the coprime fraction which is getting closer to the $6/\pi^2$ as the number of (a, b) from 0 to n increases. The computation of the coprime is done with Greatest Common Divisor, the pseudo-code for the GCD is given below.

Algorithm 3 Greatest Common Divisor (GCD)

Require: Two integers a and b
Ensure: The greatest common divisor of a and b, and the number of iterations
1: Set $itr \leftarrow 0$ {Initialize iteration counter}
2: **while** $b \neq 0$ **do**
3: Compute $r \leftarrow a \mod b$ {Find remainder when a is divided by b}
4: Set $a \leftarrow b$ {Update a with the value of b}
5: Set $b \leftarrow r$ {Update b with the remainder r}
6: Increment $itr \leftarrow itr + 1$ {Increase iteration count}
7: **end while**
8: **return** a, itr {Return the GCD and number of iterations}

Using the above routine we can create graphics with granddaddy.

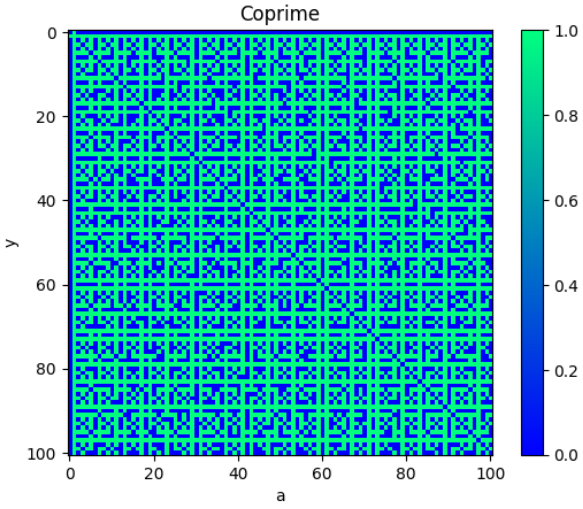
SOLUTION

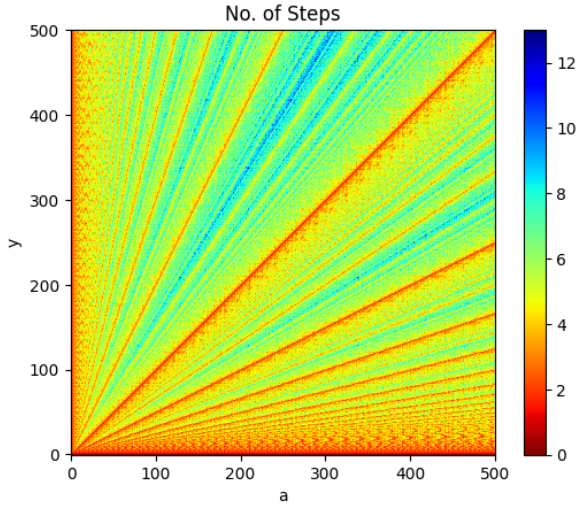
```
# Import Libraries
import numpy as np
import matplotlib.pyplot as plt

# Greatest Common Divisor Function
def GCD(a,b):
    itr=0
    while b!=0:
        r=np.mod(a,b)
        a,b=b,r
        itr+=1
    return a,itr
vf=np.vectorize(GCD)

## Coprime pair
N=100
a,b=np.meshgrid(np.linspace(0,N,N+1),np.linspace(0,N,N+1))
# Compute GCD
coprime,itr=vf(a,b)
# Plot coprime==1
plt.imshow(coprime==1,cmap='winter')
plt.colorbar()
plt.xlabel('a')
plt.ylabel('y')
plt.title('Coprime')
plt.show()

## No. of iteration for each pair(a,b)
N=500
a,b=np.meshgrid(np.linspace(0,N,N+1),np.linspace(0,N,N+1))
# Compute GCD
coprime,itr=vf(a,b)
# plot no. of iterations
map=plt.cm.get_cmap('jet')
plt.imshow(itr,cmap=map.reversed(),origin='lower')
plt.colorbar()
plt.xlabel('a')
plt.ylabel('y')
plt.title('No. of Steps')
plt.show()

# Convergence Analysis for coprime fraction
frac=[]
for N in range(1,501,10):
    a,b=np.meshgrid(np.linspace(0,N,N+1),np.linspace(0,N,N+1))
    coprime,itr=vf(a,b)
    frac.append(sum(sum(coprime==1))/(N*N))
# Plot coprime fraction
plt.plot(range(1,501,10),frac,linewidth=1)
plt.plot(range(1,501,10),[6/np.pi**2]*50,'--',linewidth=1)
plt.xlabel('n')
plt.ylabel('Coprime Fraction')
plt.title('Coprime fraction Vs n')
plt.legend(['Coprime',r'$6/\pi^2$'])
# Show plots
plt.show()
```

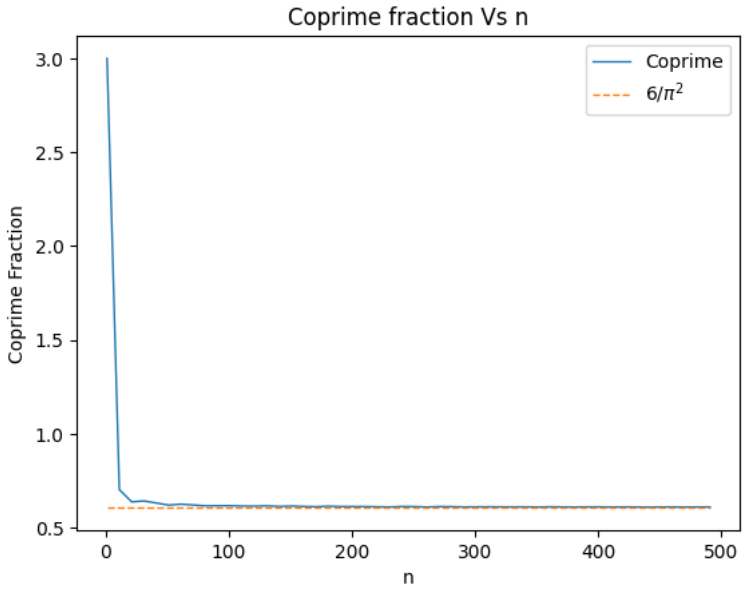
OUTPUT

Computing the GCD for all the values of a and b to n and comparing equal to 1 will give the

coprime/relative prime which is shown on the first image, the number of iteration take by GCD algorithm to return values is being plotted below. The interesting fact here is that if we compute the corpime fraction for larger values of n eventually it will reaches towards the value $6/\pi^2$ as shown.

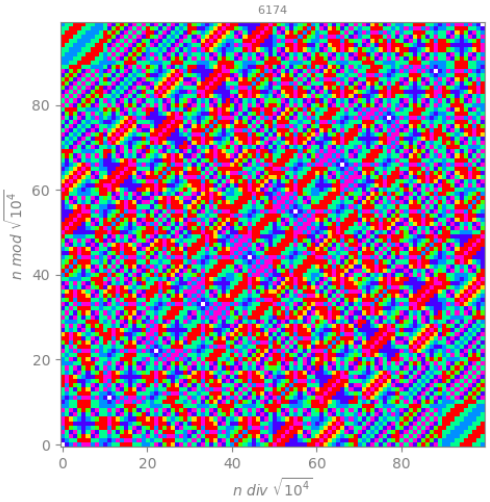





Coprime fraction Vs n
Coprime
$6/\pi^2$
Coprime Fraction
n
3.0
2.5
2.0
1.5
1.0
0.5
0
100
200
300
400
500

Kaprekar's 6174

P R O B L E M



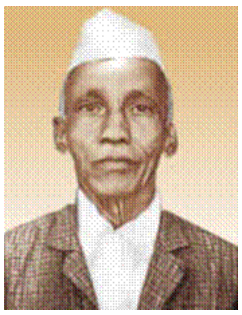
Here, color represents the number of steps required for the Kaprekar routine to reach a fixed point (0 or 6174) starting from integers $n = 0, ..., 9999$. The axes are related to n, the base(b = 10), and the number of digits ($k = 4$). "a *div* b" represents the integer division of a by b, "a *mod* b" stands for the integer remainder after integer division of a by b, and $[x]$ means the smallest integer $\geq x$.

In 1949, D. R. Kaprekar discovered a procedure, now known as the Kaprekar routine, which quickly takes a 4-decimal-digit number (which has at least two distinct digits) to the number 6174 in at most 7 steps of his procedure. Applying this procedure to 6174 produces 6174. Numbers of the form *dddd* all go to 0 in one step.

One step of the Kaprekar routine goes this way: Choose any positive k-digit integer n1 . If n1 has fewer than k digits, then take the leading digits to be 0 (i.e, consider 0s padded to n1 where they don't matter). Two integers n'_1 and $n"_1$ can be formed out of the k digits of n_1 ; namely, by sorting the digits in ascending and descending order. To get the next number n_2 in this sequence, take the difference of n'_1 and $n_1"$. That is, $n_2 = \max(n'_1, n_2") - \min(n'_1, n_2")$. Now apply the same routine to n_2 to get n_3, and so on. Any positive integer base other than 10 can also be used.

A variant of the routine omits the zero-padding step above. Depending on k and b, the two variants may show different behaviours.

One goal of this exercise is to produce a plot similar to the one above. An other goal is to computationally characterize the cyclic patterns produced by the Kaprekar routine. For example, 4-digits integers produce two distinct cyclic patterns: 0000 (integers of the form dddd) and 6174 (all other integers). Both these cycles have period = 1. Two-digit numbers produce two distinct cyclic patterns: period-1 cycle 00 (integers of the form *dd*), and the period-5 cycle 09, 81, 63, 27, 45 (all other integers). Fix the base b (say, to 10), and the number of digits k (say, to 4). Run the Kaprekar routine for each integer between 0 and $bk - 1$. Assume that this routine produces, starting from any integer, a sequence of numbers that eventually rolls into a cyclic pattern. Find out how many distinct cyclic patterns are produced for the given combination of b and k.



D. R. Kaprekar 1905-86

A P P R O A C H

Firstly, we will create the Kaprekar routine which takes the digit from 0 to 9999 and give us the number of iteration it takes to get 6174. after that the generated vector converted into the matrix so that the x-axis contain $n\ div\ \sqrt{10^4}$ and y-axis have $n\ mod\ \sqrt{10^4}$. The routine for finding the number of iterations takes for the given input is describe below. Now, we create

Algorithm 4 Kaprekar Function

Input: A four-digit integer a
Output: Number of iterations itr to reach Kaprekar's constant (6174)
Set $A \leftarrow a$
Set $itr \leftarrow 1$
while True **do**
  Convert a to a 4-digit number (pad with zeros if necessary)
  Sort the digits of a in descending order to get n_1
  Reverse the digits of a to get n_2
  Calculate $n \leftarrow |n_1 - n_2|$
  if $n = 6174$ **or** $itr > 8$ **then**
    if $itr > 7$ **then**
      Set $itr \leftarrow 0$
    end if
    break
  end if
  Set $a \leftarrow n$
  Increment itr
end while
return itr

python code based on the above pseudo-code to create the patters of kaprekar's number.

S O L U T I O N

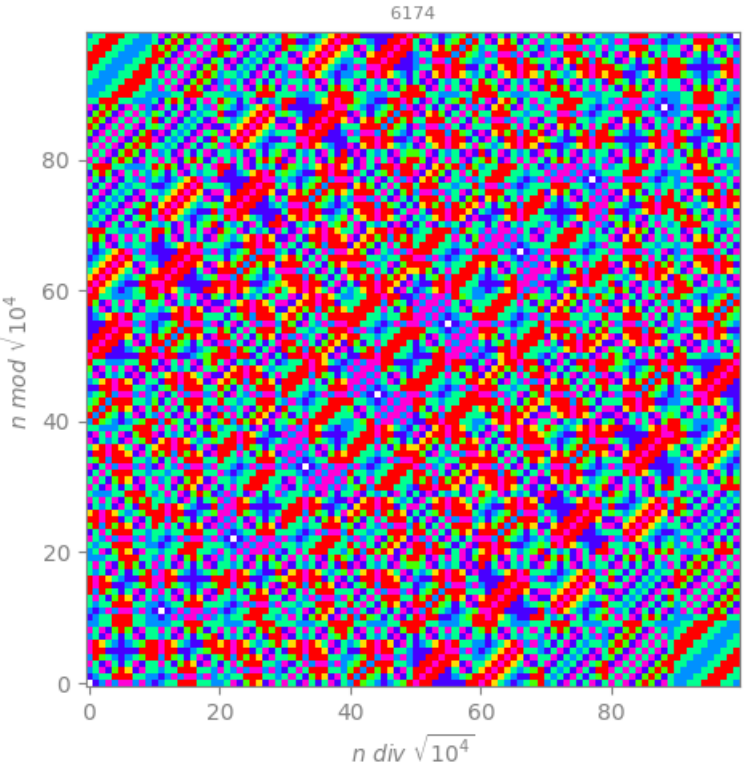
```python
# Import Libraries
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import LinearSegmentedColormap

# Kaprekar function
def kaprekar(a):
    A=a
    itr=1
    while True:
        a=''.join(sorted(list('{:04d}'.format(a)),reverse=True))
        n1=int(a)
        n2=int(a[::-1])
        n=max(n1,n2)-min(n1,n2)
        if int(n)==6174 or itr>8:
            if itr>7:
                itr=0
            break
        a=n
        itr+=1
    return itr

# Define n
```

```
n=list(range(0,10000))
# Compute kaprekar values
val=[kaprekar(i) for i in n]
# Compute x and y positions
y=np.mod(range(10000),100)
x=np.floor(np.array(list(range(10000))))
# Create matrix
m=np.zeros([100,100])
n=0
for i in range(100):
    for j in range(100):
        m[int(x[i]),int(y[j])]=val[n]
        n+=1
# Define color for iteration number
colors = {
    0: (1, 1, 1),
    1: (1, 6/7, 0),
    2: (2/7, 1, 0),
    3: (0, 1, 4/7),
    4: (0, 4/7, 1),
    5: (2/7, 0, 1),
    6: (1, 0, 6/7),
    7: (1, 0, 0)
}
# Create custom map
map = [(i/7, colors[i]) for i in range(8)]
cmap = LinearSegmentedColormap.from_list("custom_cmap", map)
# Show Image
plt.imshow(m,origin='lower',cmap = cmap)
plt.ylabel(r'$ n~mod~\sqrt{10^4} $',color='grey')
plt.xlabel(r'$ n~div~\sqrt{10^4} $',color='grey')
plt.title(r'$6174$',color='grey',size=8)
plt.tight_layout(pad=0.1)
ax=plt.gca()
# Grey all the axis
ax.spines['bottom'].set_color('grey')
ax.spines['left'].set_color('grey')
ax.spines['top'].set_color('grey')
ax.spines['right'].set_color('grey')
ax.tick_params(axis='x',colors='grey')
ax.tick_params(axis='y',colors='grey')
# Show Plot
plt.show()

## Extra for Odd and Even Number of Iterations
# Create matrix
m=np.zeros([100,100])
n=0
for i in range(100):
    for j in range(100):
        if np.mod(val[n],2)==0:
            m[int(x[i]),int(y[j])]=1
        n+=1
# Show Image
plt.imshow(m,origin='lower',cmap = 'winter')
plt.ylabel(r'$ n~mod~\sqrt{10^4} $',color='grey')
plt.xlabel(r'$ n~div~\sqrt{10^4} $',color='grey')
plt.title(r'$6174$',color='grey',size=8)
plt.tight_layout(pad=0.1)
ax=plt.gca()
# Grey all the axis
ax.spines['bottom'].set_color('grey')
```

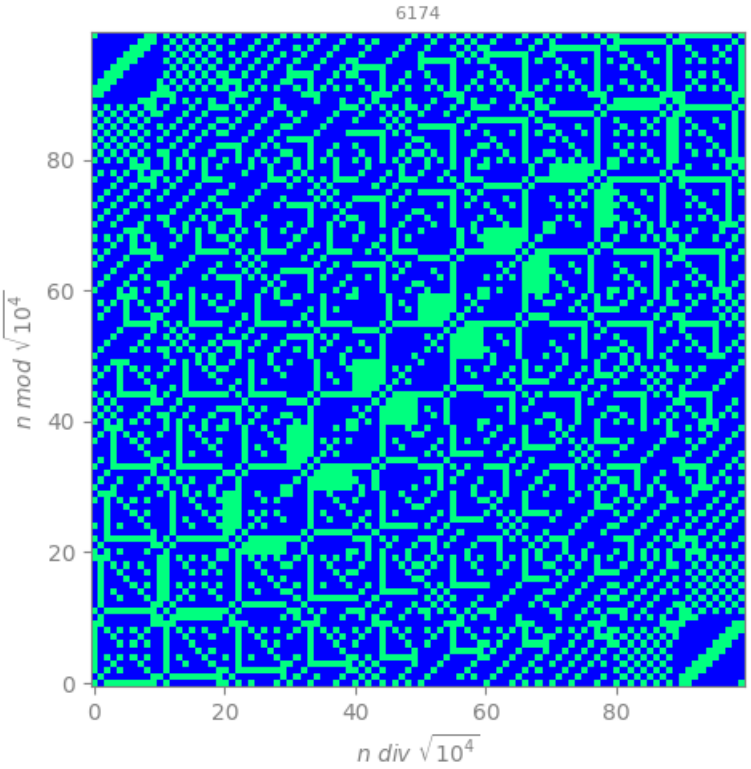
```
ax.spines['left'].set_color('grey')
ax.spines['top'].set_color('grey')
ax.spines['right'].set_color('grey')
ax.tick_params(axis='x',colors='grey')
ax.tick_params(axis='y',colors='grey')
# Show Plot
plt.show()
```

O U T P U T

Using the above code we have successfully generated the pattern using *mod* and *div* but we can try with other as well such as Odd and Even number of Iterations which we will see later in this section, but at first take a look at the below results we can see that there is pattern emerges is symmetric along the diagonal.

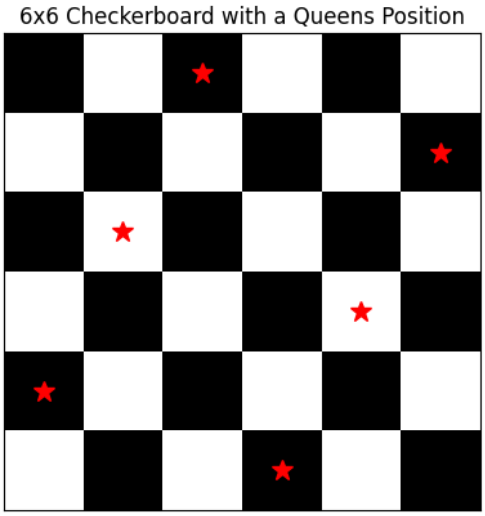


We have also created the pattern based on the number of iterations is even or odd, by computing the matrix we get the results below.


6174
0
20
40
60
80
n mod √10^4
0
20
40
60
80
n div √10^4

Long live the Queens!

PROBLEM



All conflict-free arrangements of queens

The problem of placing N queens on a N ×N chessboard so that they do not threaten each other is the celebrated N -queens problem. For this problem, color of the queen does not matter; they are all identical, highly territorial, and equally powerful. Peace prevails in this world only when the N queens do not see one another at all. Turing award winner Niklaus Wirth used this problem to illustrate a program design methodology called stepwise refinement in his 1995 article Program Development by Stepwise Refinement.
Try solving this problem in R using any approach, your own or from literature. There can be different flavours to this exercise: (A) find all the peaceful arrangements of N queens on a N × N chessboard, or (B) find one peaceful arrangement of queens as quickly as possible.

APPROACH

A chess queen exerts her influence along the vertical, the horizontal, and the two diagonal lines that cross at the her position on the chessboard. The vertical and horizontal constraints can be fulfilled by ensuring that there is exactly one queen in any row and in any column. Therefore, positions of the N queens can be specified through a vector of size N. Now, we have place the queen in such a way that here should not be any conflicts with other queens position, to check all the N position is safe we have created the routine for that as shown below.

Algorithm 5 isSafe Function for Queen Placement

Input: N, *position*
Output: Sum of safe queen placements
Initialize $A \leftarrow \mathbf{0}_{N \times N}$
for each p in *position* **do**
    Set $A[p[0], p[1]] \leftarrow 1$
end for
Initialize $flag \leftarrow []$
for each p in *position* **do**
    Set $i \leftarrow p[0]$, $j \leftarrow p[1]$
    Extract $h \leftarrow A[i, :]$, $v \leftarrow A[:, j]$
    Extract diagonal $d1 \leftarrow \text{diag}(A)$, or shifted versions based on i and j
    Extract anti-diagonal $d2 \leftarrow \text{diag}(\text{fliplr}(A))$ similarly
    Append $(\sum h + \sum v + \sum d1 + \sum d2) = 4$ to $flag$
end for
return $\sum flag$

As we get the number of position which is safe in the chessboard for the queen, by using this fact we develop the objective function whose objective is to maximize the number of position to be safe based on the available position of queens, to solve this problem we use stochastic optimization approach such as Genetic Algorithm which is a faster way to compute than other approach like Simulated Annealing method. we have to create different routines which will required for GA. some routines for GA are selection, mutation and crossover and their pseudo-code is shown below.

Algorithm 6 Selection Function

Input: Population *pop*, Fitness scores $fitness$
Output: Selected individuals for the next generation
Set $tournament\_size \leftarrow 5$
Initialize an empty list *selected*
for each individual in *pop* **do**
    Randomly select $tournament\_size$ individuals from *pop*
    Set $best\_individual \leftarrow \text{argmax}_{i \in \text{tournament}}(fitness[i])$
    Append $pop[best\_individual]$ to *selected*
end for
return *selected*

Algorithm 7 Crossover Function

Input: Parent individuals $parent1$, $parent2$
Output: Two offspring $child1$, $child2$
Randomly select a crossover point $point \leftarrow \text{randint}(1, N-1)$
Set $child1 \leftarrow parent1[:point] + parent2[point:]$
Set $child2 \leftarrow parent2[:point] + parent1[point:]$
return $child1$, $child2$

Algorithm 8 Mutation Function

Input: Individual *individual*
Output: Mutated individual
for each gene i in *individual* **do**
    if random number $<$ *mutationrate* **then**
        Set $individual[i] \leftarrow \text{randint}(0, N^2)$
    end if
end for
return *individual*

Now using all the developed routines for the Genetic algorithm we now have to define parameters to run the model such as Number of generation is 500, size of population is 2000, mutation rate is 2%, the complete process is iterative generation by generation the solution is getting improve so that we get higher fitness values. After all the computation the chessboard is plotted on figure with the safe location of queens, there are multiple ways the position of queen will be possible so at each run we can expect different location of queen by ensuring the higher fitness. this method is robust that we go for higher size of chessboard as well, we can demonstrate for 6 and 10 size of chessboard. The complete python code and the output is shown below section.

SOLUTION

```python
# Import Libraries
import numpy as np
import matplotlib.pyplot as plt

# Function to check placement of queen is SAFE
def isSafe(N,position):
    coord = np.array([[x, y] for x in range(N) for y in range(N)])
    pos=coord[position]
    unique_elements = list(set(map(tuple, pos)))
    pos = [list(item) for item in unique_elements]

    A=np.zeros([N,N])
    for i in pos:
        A[i[0],i[1]]=1
    flag=[]
    for p in pos:
        i=p[0]
        j=p[1]
        # Extract horizontal
        h=A[i,:]
        # EXtract vertical
        v=A[:,j]
        # Extract diagonal
        if i==j:
            d1=np.diag(A)
        elif i>j:
            d1=np.diag(A,-(max([i,j])-min([i,j])))
        elif i<j:
            d1=np.diag(A,(max([i,j])-min([i,j])))
        # Extract anti-diagonal
        A1=np.fliplr(A.copy())
        nj = N-1-j
        if i == nj:
            d2 = np.diag(A1)
        elif i > nj:
            d2 = np.diag(A1,-(max([i,nj])-min([i,nj])))
        else:
            d2 = np.diag(A1,(max([i,nj])-min([i,nj])))
        flag.append((sum(v)+sum(h)+sum(d1)+sum(d2))==4)
    return sum(flag)

# Function for Genetic Algorithm
# Selection Function
def selection(pop,fitness):
    tournament_size = 5
    selected = []
    for _ in range(len(pop)):
        tournament = np.random.choice(len(pop), tournament_size, replace=False)
        best_individual = max(tournament, key=lambda idx: fitness[idx])
        selected.append(pop[best_individual])
    return selected

# Crossover Function
def crossover(parent1, parent2):
    # Single point crossover
    point = np.random.randint(1, N-1)
    child1 = parent1[:point] + parent2[point:]
    child2 = parent2[:point] + parent1[point:]
    return child1, child2
```

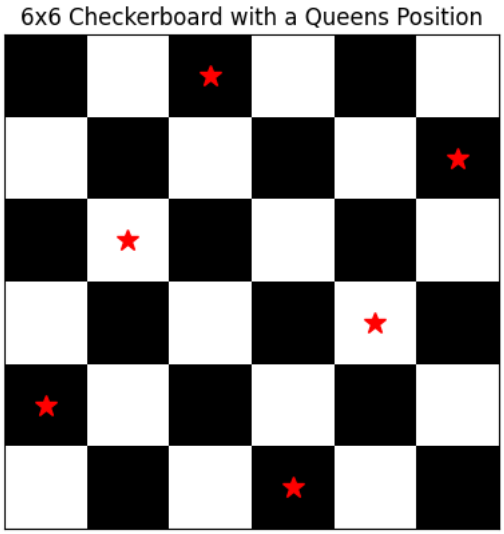
```
60
61 # Mutate Function
62 def mutate(individual):
63     # Mutate with a small probability
64     for i in range(N):
65         if np.random.rand() < mutationrate:
66             individual[i] = np.random.randint(N*N)
67     return individual
68
69 # Size of chessboard
70 N=6
71 # All possible coordinates of chessboard
72 coord = np.array([[x, y] for x in range(N) for y in range(N)])
73
74 # Define parameters
75 # Size of population
76 npop=2000
77 # No. of generation
78 ngeneration=500
79 # Mutation rate
80 mutationrate=0.02
81 # Generate Population
82 pop=[[np.random.randint(N*N) for i in range(N)] for i in range(npop)]
83 # Compute fitness
84 fitness=[isSafe(N,i) for i in pop]
85
86 # Main GA loop
87 for generation in range(ngeneration):
88     # Selection
89     selected_pop = selection(pop,fitness)
90     # Crossover and produce new population
91     new_pop = []
92     for i in range(0, len(selected_pop), 2):
93         parent1 = selected_pop[i]
94         parent2 = selected_pop[i+1] if i+1 < len(selected_pop) else selected_pop
    [0]
95         child1, child2 = crossover(parent1, parent2)
96         new_pop.append(child1)
97         new_pop.append(child2)
98     # Mutation
99     new_pop = [mutate(individual) for individual in new_pop]
100    # Update population
101    pop = new_pop[:npop]
102    # Calculate new fitness
103    fitness = [isSafe(N, i) for i in pop]
104    # Check if solution found
105    if np.max(fitness) == N:
106        print(f"Solution found in generation {generation}")
107        break
108
109 # Best solution
110 best_solution = pop[np.argmax(fitness)]
111 print(f"Best solution: \n{coord[best_solution]}\n Fitness: {np.max(fitness)}")
112
113 # Create Checkerboard
114 checkerboard = np.indices((N,N)).sum(axis=0)%2
115 plt.imshow(checkerboard,cmap='grey', interpolation='nearest')
116 plt.xticks([])
117 plt.yticks([])
118 # Plot location of queen
119 for row, col in coord[best_solution]:
120     plt.plot(col, row, marker='*', color='red', markersize=12)
```

```
plt.title(f'{N}x{N} Checkerboard with a Queens Position')
# Show plot
plt.show()
```

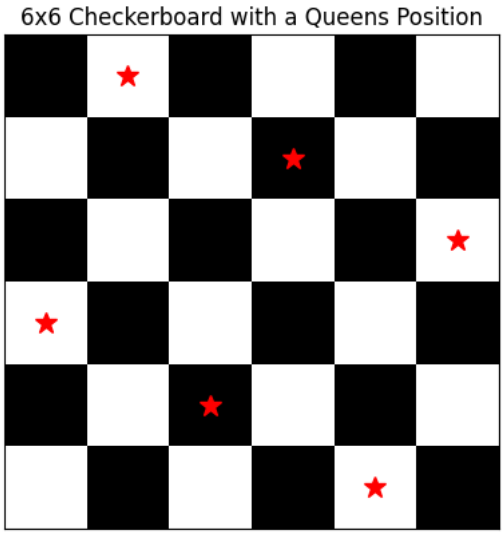
O U T P U T

We can see that for $N = 6$ size of chessboard we are able to get best solution in generation 7 and the location of queens is also displayed on the command window with plot as well.

```
Solution found in generation 7
Best solution:
[[4 0]
 [1 5]
 [3 4]
 [5 3]
 [0 2]
 [2 1]]
 Fitness: 6
```



Another possible solution,



Now, let's check for the higher size of chessboard say $N = 10$, the results are as follows.

```
Solution found in generation 7
Best solution:
[[0 3]
 [1 7]
 [2 2]
 [3 8]
 [4 5]
 [5 9]
 [6 0]
 [7 6]
 [8 4]
 [9 1]]
Fitness: 10
```