Modelling an epidemic by S.I.R model

S.Janarthanan (EE21B060)

What is the S.I.R model?

The S.I.R model is a mathematical model that is used to study the spread of an epidemic, by theoretically computing the number of infected people with respect to time.

In this model, people are grouped into 3 categories :

- People who are *susceptible* to infection - those who are capable of getting infected (S)
- People who have got *infected* (I)
- People who got infected and *recovered* (R)

Assumptions of the S.I.R model

- A person can belong to only one category at any time. For instance, if a person if infected, they are no longer susceptible.
- The epidemic can affect all people. Everyone is susceptible initially.
- Once a person is recovered, they gain immunity against the infection. They are no longer susceptible. They get infected only once.
- These three categories encompass everyone. The total population is also a constant. That is death due to the epidemic or otherwise is low
- Infection spreads only by contact between susceptible and infected persons.

Mathematics involved and explanation of model

Let $S(t)$ be the number of susceptible people as a function of time, $I(t)$ be the number of infected people and $R(t)$ be the recovered people. From the assumptions, $S(t) + I(t) + R(t) = N$, where N is the total population - a constant.

A susceptible person becomes infected when he comes in contact with an infected person. The number of interactions between them will be proportional to both the number of infected as well as number of susceptible people. Higher the population of these 2 categories, more is the chance of contact, higher is the chance of infection.

Hence, people get infected at a rate $a \cdot S(t) \cdot I(t)$, where a is a positive constant of proportionality. Numerically,

$$\frac{dS}{dt} = -a \cdot S \cdot I$$

As people get infected, they leave the susceptible category and enter the infected category. Hence the minus sign.

People enter the infected category at a rate of $a \cdot S \cdot I$. But that is not the only factor affecting it. People get recovered too. We assume that more is the number of infected people, more would be the recovery rate, i.e recovery rate $\propto$ I(t).

Thus,

$$\frac{dI}{dt} = (a \cdot S \cdot I - b \cdot I) \text{ and } \frac{dR}{dt} = b \cdot I$$

where b is a positive constant of proportionality.

We have the equations :

$$\frac{dS}{dt} = -a \cdot S \cdot I \tag{1}$$

$$\frac{dI}{dt} = a \cdot S \cdot I - b \cdot I \tag{2}$$

$$\frac{dR}{dt} = b \cdot I \tag{3}$$

This set of three equations are an example of simultaneous system of differential equations. As there is a product term $S \cdot I$ involved in these equations, its solution is not trivial, but we can still do some analysis just with these derivative equations.

Some simple analysis

Near time = 0:

Let S_0 be the initial susceptible people, I_0 be the initial number of infected people. At the start of an epidemic, number of recovered people is 0.

At the start of an epidemic, the number of infected people is small (we detect the infection when number of infected people is low). Hence, I_0 is very low and correspondingly, S_0 is quite high.

An introduction to R_0 :

$\frac{dI}{dt}$ at $t = 0$ is $(a \cdot S_0 \cdot I_0 - b \cdot I_0)$. An aspect of interest would be to see if we will have an epidemic or not. This is determined by the sign of $\frac{dI}{dt}$ at t = 0. If it is less than 0, it means people recover faster than they get infected, thus the infected population becomes 0 even before it can increase. If it is greater than 0, there is a net spread of disease, hence there will be an epidemic.

So we ask "is $(a \cdot S_0 - b) \cdot I_0 < 0$?". This amounts to asking if

$$\frac{aS_0}{b} < 1.$$

This expression $\frac{a \cdot S_0}{b}$ is called R_0.

- If $R_0 < 1$, then the number of infected people reach 0 even before there is considerable spread, thus preventing an epidemic.
- Even if $R_0 > 1$, a low value of R_0 is preferred over a higher value. A high value means that the spread of infection is fast, which isn't desirable.

Factors affecting R_0 :

- R_0 is directly proportional to a, the proportionality constant pertaining to infection rate. When we do practices like preventing contact by social distancing, isolating the infected, washing hands and face regularly, we can reduce this a, hence reducing R_0. This is the idea of flattening the curve in a nutshell!
- We can also decrease S_0 by bringing vaccines into the picture. Vaccines bring about immunity from the disease, thus making them insusceptible.
- We cannot really affect b, unless we find a cure and improve healthcare to increase b.

I(t) near t = 0 :

S is quite high near this initial part of epidemic. We can roughly say that S is always S_0 near t = 0. Hence, we have the differential equation $\frac{dI}{dt} \approx I \cdot (a \cdot S_0 - b)$.

$I(t)$ has the solution $I(t) \approx e^{(a \cdot S_0 - b)t}$. *I(t) roughly follows an exponential solution near the start of an epidemic.*

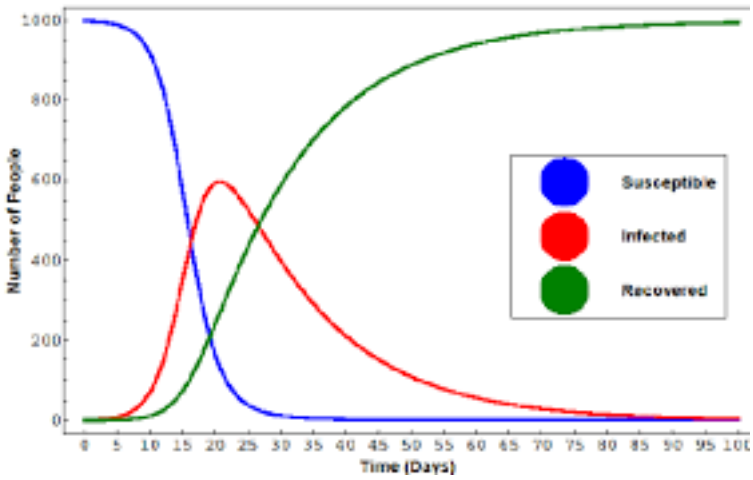


Figure 1: A model S.I.R graph. (Image taken from the net.)

Rough graphs of S(t), I(t) and R(t) :

- S(t) : Initially, everyone is susceptible and hence S_0 is close to N. As time progresses, S keeps decreasing as more and more get infected and after a point, everyone would have been infected, making S to be 0.
- I(t) : Initially, number of infected people is low, but it increases due to the $a \cdot S \cdot I$ term. Over time, I keeps increasing. However, after a point, the $-b \cdot I$ term increases and it begins to dominate. I(t) reaches a peak and then decreases and eventually becomes 0 as people recover over time.
- R(t) : The recovered people starts at 0 and keeps increasing. It keeps increasing at a rate proportional to infected people. After a point, everyone would have been infected and reached the recovered stage.

  A rough graph illustrating these points is shown in 1

Improvements and variants of the S.I.R model :

The S.I.R model is quite simple and not always representative of an actual epidemic. Several improvements can be added. Some are :

- Vital dynamics : We have to account for births and deaths as well. We can incorporate a birth and death rates. The total population too can fluctuate.
- S.E.I.R model : Sometimes, we don't get infected as soon as we contact an infected person. There is usually some latency period. We can introduce a new category called "exposed" people, who are in this latency period. They will spread the infection at a rate lower than people who are actually infected.
- S.I.S model : Sometimes, we don't actually recover even after getting infected. We can get infected again. We can use the S.I.S model, which is the susceptible, infected, cured-yet-susceptible model. Common cold is one such infection. It is also possible that some get immunity and some don't, depending on how good their immune system is.

A good model for an epidemic would be one which incorporates all these ideas. However, it is worth mentioning that the equations and solutions will get more complicated.

References

1. https://youtu.be/Qrp40ck3WpI
2. https://youtu.be/f1a8JYAixXU
3. https://en.wikipedia.org/wiki/Compartmental_models_in_epidemiology#The_SIRV_model

Logistic Map

The logistic equation is a recurrence relation given by $x_{n+1} = rx_n(1 - x_n)$. One of its main areas of application is in modelling population growths. Usually, we take $r > 0$ and $x \in [0, 1]$.

```
"""
logistic_map function is repeatedly applied starting with initial value
known as x_0.

xn is the value of logistic_map from last call.
r is the growth rate parameter.
"""
def logistic_map(xn, r):
    return r*xn*(1-xn)
```

Boundedness of the logistic map :

The logistic map always converges when $0 < r < 4$. We can analyse when the map gets unbounded (we assume $r \geq 0$ for simplicity). For $0 < r < 1$, it always reaches zero (as seen from the next problem) , hence we need not consider this case.

It is clear that once x_n becomes negative, it keeps becoming more and more negative, as both r and $(1 - x_n)$ are greater than 1. So, we need to find the values of r for which x_n may become negative.

Let's say in nth iteration x_n becomes negative for the first time. Then x_{n-1} has to be greater than 1. This is because, only the $(1 - x_{n-1})$ term can flip the sign of x_n. So we have the question :

When is $rx(1 - x) > 1$?

Solving for $rx(1 - x) = 1$, we get that :

$$x \in \left(\frac{1 - \sqrt{1 - \frac{4}{r}}}{2}, \frac{1 + \sqrt{1 - \frac{4}{r}}}{2} \right)$$

For x to be real, we require $r > 4$. Thus, for $r \in [0, 4]$, the relation is always bounded. It has been found that for $r > 4$, the recurrence is almost always unbounded irrespective of initial x_0.

We run a python program for $r = 3.4$ and $r = 4.1$, and $x_0 = 0.2$ to illustrate this.

```
from tabulate import tabulate

r = 3.4
x_0 = 0.2
output = []
for i in range(1000):
```

```
  x_0 = logistic_map(x_0, r)
output.append([3.4, 1000, x_0])

r = 4.1
x_0 = 0.2
for i in range(20):
  x_0 = logistic_map(x_0, r)

output.append([4.1, 20, x_0])
print((tabulate(output, headers = ['r', 'iterations', 'final value'])))
```

```
      r    iterations    final value
---  ---  ------------  -------------
3.4          1000       0.842154
4.1            20     -inf
```

Observation :

As expected, for $r = 3.4$, it is indeed bounded even after quite a few iterations, and for $r = 4.1$, it shoots to $-\infty$ even for very few iterations.

Logistic map for $0 < r < 1$:

The value of x eventually goes to 0 after a large number of iterations, when the $r \in (0, 1)$. This happens irrespective of the initial x value. The following is a python program for $x_0 = 0.2$ and $r = 0.5$. We display the value of x after 1, 10, 50, 100, 500 and 2000 iterations. It is clear that x keeps decreasing till it reaches 0.

```
x_0 = 0.2
output = []
r = 0.5
output.append([0, x_0])
for i in range (1, 2001): #we carry out the iteration 2000 times.
  x_0 = logistic_map(x_0, r)
  if i in [1, 10, 50, 100, 500, 2000]: # we are displaying the value of x after the
    output.append([i, x_0])

print(tabulate(output, headers = ['iterations', 'value']))
```

```
      iterations        value
------------  -----------
           0  0.2
           1  0.08
          10  0.000133701
          50  1.21568e-16
         100  1.07974e-31
         500  4.1814e-152
        2000  0
```

Logistic map for $1 < r < 3$:

Here, x reaches a certain value several iterations. After this, even if we iterate the map, x value remains the same. For $1 < r < 3$, this constant value is $\frac{r-1}{r}$. We run a python program for $r = 1.5$ and different initial values x_0. We can see that the value coverges to $\frac{1}{3}$, even for few iterations

```
r = 1.5
# The elements in this list are the various initial values taken
x_0_values = [0.00001, 0.001, 0.1, 0.2, 0.5 , 0.7]

# list of lists for tabular output
output = []
for x_0 in x_0_values:
  val = x_0
  for i in range(5):
    val = logistic_map(val, r)
  output.append([x_0, 5, val])
  for i in range(25):
    val = logistic_map(val, r)
  output.append([x_0, 30, val])
  for i in range(20): # we show what happens to x after 5, 30 and 50 iterations
    val = logistic_map(val,r)
  output.append([x_0, 50, val])
print(tabulate(output, headers = ['initial x', 'number of iterations', 'final value
```

initial x	number of iterations	final value
1e-05	5	7.59275e-05
1e-05	30	0.315156
1e-05	50	0.333333
0.001	5	0.00749458
0.001	30	0.333326
0.001	50	0.333333
0.1	5	0.28469
0.1	30	0.333333
0.1	50	0.333333
0.2	5	0.32303
0.2	30	0.333333
0.2	50	0.333333
0.5	5	0.335405
0.5	30	0.333333
0.5	50	0.333333
0.7	5	0.332061
0.7	30	0.333333
0.7	50	0.333333

Logistic map for $r > 3$:

The following is a python program for different initial x_0, with $r = 3.2$. We iterate the logistic_map 1000 times and display the last five iterations for each x_0. We can see that x does not reach a constant value. It alternates between two values. Interestingly, these two values

between which x alternates are the same, irrespective of the initial x_0 (about 0.513 and 0.799). Thus, we can say that these final **equillibrium values** depend only on r, and not on x_0.

However, iteration numbers corresponding to 0.513 and to 0.799 do depend on the initial x_0 (for example, every even iteration corresponds to 0.513 to $x_0 = 0.01$, but every even iteration corresponds to 0.799 for $x_0 = 0.2$).

It has been found that these two values between which x alternates are given by : $x_{final} = \frac{1}{2r} \cdot (r + 1 \pm \sqrt{(r-3)(r+1)})$. Putting r = 3.2, we do get the two values of x_{final} to be 0.799 and 0.513. This relation holds only when $r \in \left(3, 1 + \sqrt{6}\right)$.

```
x_0_values = [0.00001,0.001, 0.2, 0.7]
r = 3.2
output = []
for x_0 in x_0_values:
  a = x_0
  for j in range (1000):
    x_0 = logistic_map(x_0, r)
    if j >= 995:
      output.append([a, j+1, x_0])
print((tabulate(output, headers = ['initial x', 'iterations', 'final value'])))
```

```
  initial x    iterations    final value
-----------  ------------  -------------
      1e-05           996       0.513045
      1e-05           997       0.799455
      1e-05           998       0.513045
      1e-05           999       0.799455
      1e-05          1000       0.513045
      0.001           996       0.513045
      0.001           997       0.799455
      0.001           998       0.513045
      0.001           999       0.799455
      0.001          1000       0.513045
      0.2             996       0.799455
      0.2             997       0.513045
      0.2             998       0.799455
      0.2             999       0.513045
      0.2            1000       0.799455
      0.7             996       0.799455
      0.7             997       0.513045
      0.7             998       0.799455
      0.7             999       0.513045
      0.7            1000       0.799455
```

References

- Wikipedia Logistic Map
- Wolfram Logistic Map
- Visualizing Chaos and Randomness

Slope Fields.

Slope fields are used to visualize the shape of different solutions to a differential equation. This is particularly useful when a differential equation does not have an analytical solution. Here, we are given the system of differential equations:

$$\frac{dB}{dt} = r_b B(t) - kB(t)P(t)$$

and

$$\frac{dP}{dt} = -d_p P(t) + kB(t)P(t)$$

where r_b, d_p and k are constants.

We are asked to plot a slope field for B vs P. Let us take P as the $y-axis$ and B as the $x-axis$.

Using the chain rule, we have

$$\frac{dP}{dB} = \frac{\frac{dP}{dt}}{\frac{dB}{dt}} = \frac{-d_p P + kBP}{r_b B - kBP}$$

Now, let us run a code which will output our slope as a vector. When our slope is m, the corresponding vector is $\hat{i} + m \cdot \hat{j}$. We will consider $r_b = 0.8$, $d_p = 1.6$ and $k = 1.2$.

Let us find the slope fields at the grid $B \times P \in [-0.5, 0.5] \times [-0.5, 0.5]$, at every 0.2 interval. At each point, we will output the following:

1. slope m
2. direction $\hat{i} + m\hat{j}$
3. the $\arctan(m)$

This program can be cuztomized with different grid sizes, partitions on each axis , and different values of r_b, d_p, k.

```
import math

# Function to calculate the slope field at a given point (B, P)
def slope(B, P, r_b, d_p, k):
  numerator = (d_p * P) + (k * B * P)
  denominator = (r_b * B) - (k * B * P)
  try:
    return numerator / denominator
  except ZeroDivisionError:
    return math.nan
```

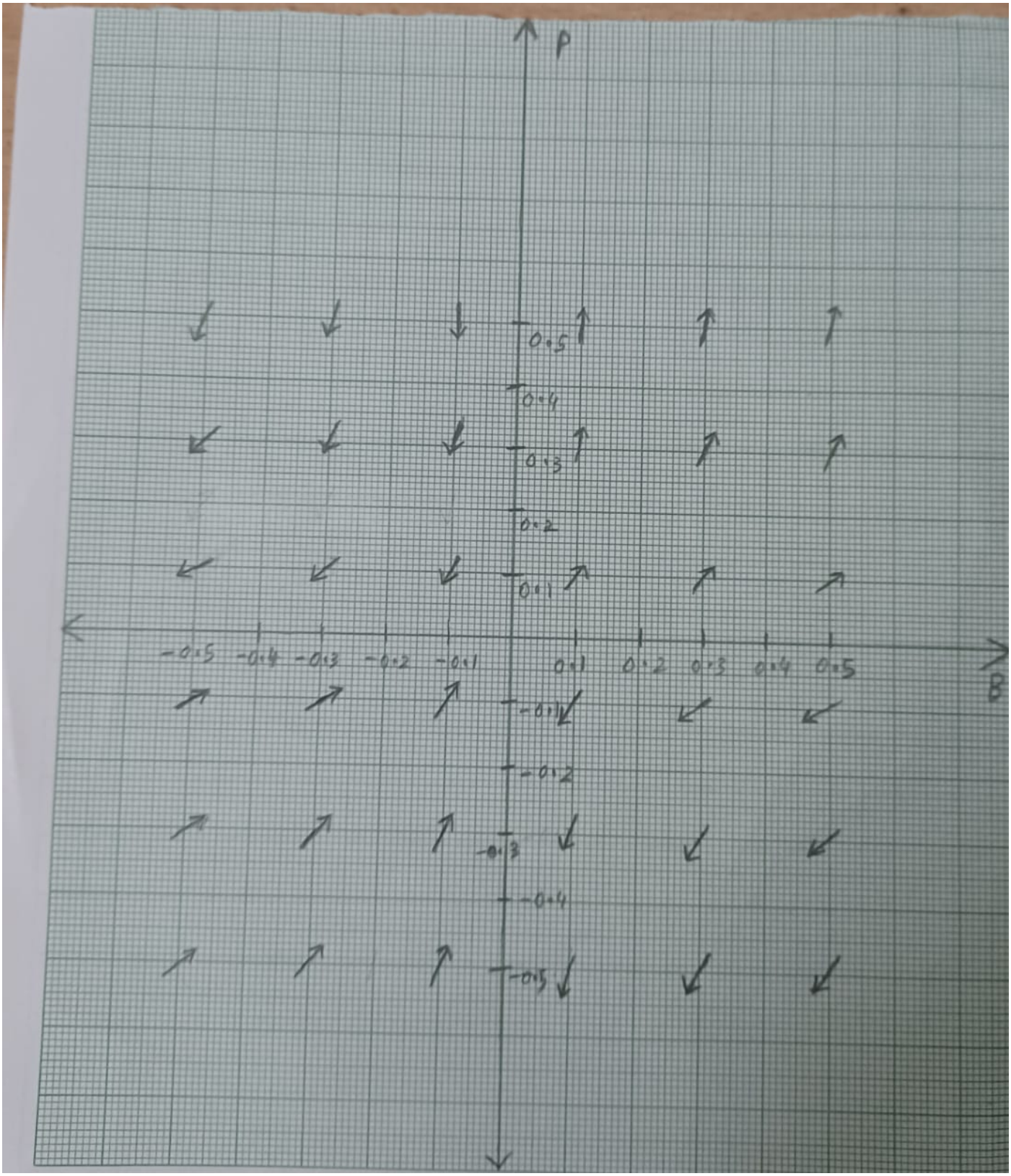
```
import numpy as np

def coordinates(lower_limit, upper_limit, partitions):
```

```
  gap = (upper_limit - lower_limit)/partitions
  output = []
  for i in range(0, partitions+1):
    output.append(lower_limit + i*gap)
  return output


from tabulate import tabulate


def print_slope_field(
    lower_limit_b = -0.5,
    upper_limit_b = 0.5,
    partitions_b = 5,
    lower_limit_p = -0.5,
    upper_limit_p = 0.5,
    partitions_p = 5,
    r_b = 0.8,
    d_p = 1.6,
    k = 1.2):

    output = []
    b_points = coordinates(lower_limit_b, upper_limit_b, partitions_b)
    p_points = coordinates(lower_limit_p, upper_limit_p, partitions_p)
    for b in b_points :
        for p in p_points:
            slp = slope(b, p, r_b, d_p, k)
            direction = (1, slp)
            # angle in degree
            angle = math.atan(slp)*180.0/math.pi
            output.append(["{0:.2f}".format(b), "{0:.2f}".format(p), direction, "{0
    print(tabulate(output, headers = ['B', 'P', 'vector', 'angle wrt B axis in degr


print_slope_field()
```

B	P	vector	angle wrt B axis in degree
-0.5	-0.5	(1, 0.7142857142857143)	35.54
-0.5	-0.3	(1, 0.5172413793103448)	27.35
-0.5	-0.1	(1, 0.21739130434782605)	12.26
-0.5	0.1	(1, -0.2941176470588238)	-16.39
-0.5	0.3	(1, -1.3636363636363638)	-53.75
-0.5	0.5	(1, -4.999999999999998)	-78.69
-0.3	-0.5	(1, 1.4761904761904765)	55.89
-0.3	-0.3	(1, 1.0689655172413794)	46.91
-0.3	-0.1	(1, 0.4492753623188406)	24.19
-0.3	0.1	(1, -0.6078431372549026)	-31.29
-0.3	0.3	(1, -2.8181818181818197)	-70.46
-0.3	0.5	(1, -10.333333333333336)	-84.47
-0.1	-0.5	(1, 5.285714285714288)	79.29
-0.1	-0.3	(1, 3.8275862068965525)	75.36
-0.1	-0.1	(1, 1.6086956521739133)	58.13
-0.1	0.1	(1, -2.1764705882352966)	-65.32
-0.1	0.3	(1, -10.090909090909095)	-84.34
-0.1	0.5	(1, -37.0)	-88.45

```
0.1  -0.5  (1, -6.142857142857138)          -80.75
0.1  -0.3  (1, -4.448275862068962)          -77.33
0.1  -0.1  (1, -1.8695652173913024)         -61.86
0.1   0.1  (1, 2.5294117647058827)           68.43
0.1   0.3  (1, 11.727272727272721)           85.13
0.1   0.5  (1, 42.999999999999964)           88.67
0.3  -0.5  (1, -2.3333333333333335)         -66.8
0.3  -0.3  (1, -1.6896551724137925)         -59.38
0.3  -0.1  (1, -0.7101449275362317)         -35.38
0.3   0.1  (1, 0.960784313725491)            43.85
0.3   0.3  (1, 4.454545454545455)           77.35
0.3   0.5  (1, 16.33333333333333)           86.5
0.5  -0.5  (1, -1.5714285714285716)         -57.53
0.5  -0.3  (1, -1.1379310344827585)         -48.69
0.5  -0.1  (1, -0.4782608695652173)         -25.56
0.5   0.1  (1, 0.6470588235294124)           32.91
0.5   0.3  (1, 3.0000000000000004)           71.57
0.5   0.5  (1, 10.999999999999996)           84.81
```

✓ 0s completed at 22:58


P
0.5
0.4
0.3
0.2
0.1
-0.5
-0.4
-0.3
-0.2
-0.1
0.1
0.2
0.3
0.4
0.5
B
-0.1
-0.2
-0.3
-0.4
-0.5

Integration Methods

Numerically computing $\int_0^1 x^2 dx$

```
"""
function that computes square of the given value x
"""
def square(x):
  return x*x
```

Riemann Sum Method

The sum is calculated by partitioning the region into shapes (usually rectangles or trapezoids), calculating area of each shape and adding all the areas. As the number of partitions get arbitrarily large, the Riemann sum gets arbitrarily closer to the actual definite integral.

[Wikipedia Riemann Sum Page]

Left Riemann Sum

The left Riemann Sum is found by approximating the function to its value at the left end point. The width of rectangle may or may not be uniform, but we take uniform partitioning. Once we add up all the areas, we get :

$$A_{left} = \Delta x \cdot [f(a) + f(a + \Delta x) + f(a + 2\Delta x) + \cdots + f(a + (n-1)\Delta x)]$$

```
"""
function to integrate given integrand from 'a' to 'b' using
the left Riemann sum method.

N is the number of rectangles to use
integrand is the function to integrate
a is the lower limit (default is 0)
b is the upper limit (default is 1)
"""
def left_riemann_sum (N, integrand, a = 0.0, b = 1.0) :
  delta_x = (b-a)/N # uniform partition of the interval [a, b]
  sum = 0
  for i in range (N):
    sum = sum + integrand(a + (i * delta_x))
  return sum/N
```

```
left_riemann_sum (10000, square)
```

```
0.3332833349999983
```

Right Riemann Sum

The Right Riemann Sum is found by approximating the funciton to its value at the right end point of the interval. Again, we consider uniform partition of the interval. Once we add up all the areas, we get :

$$A_{right} = \Delta x \cdot [f(a + \Delta x) + f(a + 2\Delta x) + \cdots + f(a + n\Delta x)]$$

```
"""
function to integrate given integrand from 'a' to 'b' using
the right Riemann sum method.

N is the number of rectangles to use
integrand is the function to integrate
a is the lower limit (default is 0)
b is the upper limit (default is 1)
"""
def right_riemann_sum (N, integrand, a = 0.0, b = 1.0) :
  delta_x = (b-a)/N # uniform partition of the interval [a, b]
  sum = 0
  for i in range (N):
    sum = sum + integrand(a + ((i+1) * delta_x))
  return sum/N
```

```
right_riemann_sum(10000, square)
```

```
0.3333833349999983
```

Trapezoidal Riemann Sum

Here, the trapezoidal area is taken, rather than rectangular area. Numerically, it is the average of left and right Riemann sums. When we add all the trapezoidal areas, we get :

$$A_{trpz} = \frac{\Delta x}{2} \cdot [f(a) + 2 \cdot f(a + \Delta x) + 2 \cdot f(a + 2\Delta x) + \cdots 2 \cdot f(a + (n-1)\Delta x) + f(b)]$$

```
"""
function to integrate given integrand from 'a' to 'b' using
the Riemann trapezoidal sum method.

N is the number of rectangles to use
integrand is the function to integrate
a is the lower limit (default is 0)
b is the upper limit (default is 1)
"""
def riemann_trapezoidal_sum(N, integrand, a = 0.0, b = 1.0):
  sum = 0
```

```
  delta_x = (b-a)/N #uniform partition of the interval [a, b]
  for i in range (N):
    left_plus_right = integrand(a + (i * delta_x)) + \
                      integrand(a + ((i+1) * delta_x))
    sum = sum + (left_plus_right/2)
  return sum/N
```

```
riemann_trapezoidal_sum(10000, square)
```

```
0.33333333500000173
```

Simpons method (aka parabolic approximation)

In this method, we will have to partition the interval into an even number of partitions. Unlike the previous methods, here we approximate the area not by straight lines, but by vertical parabolas. Once we calculate the area under each parabola, we will get :

$$\int_a^b f(x) \cdot dx \approx \frac{\Delta x}{3} \cdot (y_0 + 4y_1 + 2y_2 + 4y_3 + \cdots + 4y_{n-1} + y_n)$$

where $\Delta x = \frac{b-a}{n}$ and $y_i = f(a + i\Delta x)$

See also Wikipedia Simpson's_rule

```
"""
integrate using Simpson's method/ parabolic curve approximation,
where N is number of intervals
"""
def simpson_sum(N, integrand, a = 0.0, b = 1.0):
  if N % 2 != 0:
    # we have to make sure that N is even for Simpson's formula.
    # Hence we add 1 if N is odd
    N += 1

  delta_x = (b - a)/N # uniform partition of the interval [a, b]
  sum = 0
  for i in range(0, N-1, 2):
    sum += (delta_x/3) * ( integrand(a + i*delta_x) +
                4*integrand(a + (i+1)*delta_x) + integrand(a + (i+2)*delta_x) )
  return sum
```

```
simpson_sum(1, square)
```

```
0.3333333333333333
```

Monte Carlo Integration Method

The Monte Carlo method uses average value of function values evaluated at randomly chosen discrete points in the integration interval.

Explanation :

- Let the continuous random variable X have a probability density function p(x). Then, the expected value of a function f(X) is :

$$E(f(X)) = \int_{-\infty}^{\infty} f(x) \cdot p(x)dx$$

- If X is a uniform random variable in the interval [a, b] :

$$p(x) = \frac{1}{b-a},$$

and

$$E(f(X)) = \frac{1}{b-a} \cdot \int_{a}^{b} f(x)dx.$$

- Then, $\int_a^b f(x)dx = (b-a) \cdot E(f(X))$. This is the basis for Monte Carlo method.

$$E(f(x)) = \lim_{N\to\infty} \frac{\sum_{i=0}^{N} f(X_i)}{N}$$

$$E(f(x)) \approx \frac{\sum_{i=0}^{N} f(X_i)}{N} \text{ for some large} N$$

$$E(f(X)) \approx \frac{\sum_{i=0}^{N} f(X_i)}{N} \text{ for some large} N$$

$$\int_{a}^{b} f(x)dx \approx (b-a) \cdot \frac{\sum_{i=0}^{N} f(X_i)}{N} \text{ for some large} N$$

- See Also "Basics of Monte Carlo Integration Method"

```
import random

"""
Integrating the integrand by using Monte Carlo method,
where N is the number of points chosen for sampling.
"""
def Monte_Carlo_method(N, integrand, a = 0.0, b = 1.0):
  sum = 0
  for i in range(N):
    sum = sum + integrand(a + random.random()*(b-a))
  return sum/N
```

```
Monte_Carlo_method(10000, square)
```

```
0.3354817310601054
```

Numerically Integrating, given a set of points (x, y) of the function:

- The rectangular Riemann Sum methods now work by simply multiplying y coordinate with the difference of consecutive x coordinates, like : $\Sigma y_i \cdot (x_{i+1} - x_i)$, or $\Sigma y_{i+1} \cdot (x_{i+1} - x_i)$, and the trapezoidal sum works in a similar way too.
- We need an odd number of points (x, y) and even spacing between them (atleast between consecutive points) for Simpson's method to work. As we may not always get such a set of points, Simpson's method is NOT done here.
- For the Monte Carlo method, we average out all the y-coordinates and multiply by the total width, $(x_n - x_1)$.
- The set of points (x, y) is taken as a list of tuples using the python language. The first element of the tuple is the x-coordinate and the second element is the y-coordinate. Each element of a list is a tuple representing (x_i, y_i). We assume that the list is sorted by the first coordinate, i.e $x_1 < x_2 < x_3 < \cdots < x_n$.

```
def left_riemann_sum2(list_of_points):
  N = len(list_of_points)
  sum = 0
  for i in range(N - 1):
    sum = sum + (list_of_points[i][1] *
                 (list_of_points[i+1][0] - list_of_points[i][0]))
  return sum
```

```
left_riemann_sum2([ (i/10000, square(i/10000))  for i in range(10001)])
```

```
0.33328333499999957
```

```
x =  sorted([ random.random() for i in range(10000)])
left_riemann_sum2([(i, square(i)) for i in x])
```

```
0.333147647368984
```

```
def right_riemann_sum2(list_of_points):
  N = len(list_of_points)
  sum = 0
  for i in range(N-1):
    sum = sum + (list_of_points[i+1][1] *
                 (list_of_points[i+1][0] - list_of_points[i][0]))
  return sum
```

```
right_riemann_sum2([ (i/10000, square(i/10000)) for i in range(10001)])
```

```
0.3333833349999992
```

```
x =  sorted([ random.random() for i in range(10000)])
right_riemann_sum2([(i, square(i)) for i in x])
```

```
0.33339650309007884
```

```
def trapezoidal_sum2(list_of_points):
  N = len(list_of_points)
  sum = 0
  for i in range (N-1):
    left_plus_right = (list_of_points[i+1][1] + list_of_points[i][1]) * \
         (list_of_points[i+1][0] - list_of_points[i][0])
    sum = sum + (left_plus_right/2)
  return sum
```

```
trapezoidal_sum2([ ((i/10000), square(i/10000)) for i in range (10001)])
```

```
0.33333333499999973
```

```
def monte_carlo_sum2(list_of_points):
  N = len(list_of_points)
  sum = 0
  for i in range (N):
    sum = sum + list_of_points[i][1]
  integral = (sum/N) * (list_of_points[N-1][0] - list_of_points[0][0])
  return integral
```

```
x =  sorted([ random.random() for i in range(10000)])
monte_carlo_sum2([(i, square(i)) for i in x])
```

```
0.32807657703552123
```

Error analysis :

- The numerically computed integral is an approximation. It will differ from the actual value, and we can achieve arbitrary degree of precision by increasing our sample sizes.
- The absolute error is the absolute value of the difference between the actual value and the value computed by us. Percentage error is relative error, expressed as a percentage.
- Here, the exact value of $\int_0^1 x^2 dx$ is $\frac{1}{3}$ by Newton-Leibniz method.
- We run a python program illustrating the absolute and percentage errors for the different methods used to numerically integrate x^2. The data is expressed in a tabular format to emphasize the increasing accuracy of the computed values.
- For Monte Carlo method, N is the number of sample points, and for all other methods, N is the number of partitions.

```
from tabulate import tabulate # we use the tabulate module to print in a table form

def percent_error(a, b):
  return "{0:.3g}".format(100 * abs(a - b)/a) + "%"

N = [10, 100, 1000, 10000, 100000]
actual = 1/3
# d is a list of lists, and each element in d is a list containing the error inform
d = []
l = []
for i in N:
  computed = left_riemann_sum(i, square)
  l = [i, "left sum", computed, abs(actual - computed),
       percent_error(actual, computed)]
  d.append(l)
for i in N:
  computed = right_riemann_sum(i, square)
  l = [i, "right sum", computed, abs(actual - computed),
       percent_error(actual, computed)]
  d.append(l)
for i in N:
  computed = riemann_trapezoidal_sum(i, square)
  l = [i, "trapezoidal", computed, abs(actual - computed),
       percent_error(actual, computed)]
  d.append(l)
for i in N:
  computed = simpson_sum(i, square)
  l = [i, "Simpson", computed, abs(actual - computed),
       percent_error(actual, computed)]
  d.append(l)
for i in N:
  computed = Monte_Carlo_method(i, square)
  l = [i, "Monte Carlo", computed, abs(actual - computed),
       percent_error(actual, computed)]
  d.append(l)

print(tabulate(d, headers=["Number of samples", "Method",
                           "Computed value", "Error", "% error"]))
```

Number of samples	Method	Computed value	Error	% error
10	left sum	0.285	0.0483333	14.5%
100	left sum	0.32835	0.00498333	1.49%
1000	left sum	0.332833	0.000499833	0.15%
10000	left sum	0.333283	4.99983e-05	0.015%
100000	left sum	0.333328	4.99998e-06	0.0015%
10	right sum	0.385	0.0516667	15.5%
100	right sum	0.33835	0.00501667	1.51%
1000	right sum	0.333833	0.000500167	0.15%
10000	right sum	0.333383	5.00017e-05	0.015%
100000	right sum	0.333338	5.00002e-06	0.0015%
10	trapezoidal	0.335	0.00166667	0.5%
100	trapezoidal	0.33335	1.66667e-05	0.005%
1000	trapezoidal	0.333334	1.66667e-07	5e-05%

10000	trapezoidal	0.333333	1.66667e-09	5e-07%
100000	trapezoidal	0.333333	1.66619e-11	5e-09%
10	Simpson	0.333333	5.55112e-17	1.67e-14%
100	Simpson	0.333333	5.55112e-17	1.67e-14%
1000	Simpson	0.333333	1.11022e-16	3.33e-14%
10000	Simpson	0.333333	1.11022e-16	3.33e-14%
100000	Simpson	0.333333	1.44329e-15	4.33e-13%
10	Monte Carlo	0.173595	0.159738	47.9%
100	Monte Carlo	0.361122	0.0277886	8.34%
1000	Monte Carlo	0.350089	0.016756	5.03%
10000	Monte Carlo	0.334729	0.00139568	0.419%
100000	Monte Carlo	0.333301	3.22797e-05	0.00968%

Observations:

- As the number of samples/partitions increases, the error decreases in all the methods.
- The error in Simpson's rule is exceptionally small, for any amount of partitioning. This is because the Simpson's method is accurate for polynomials of degree 3 or less. Here our integrand is x^2
- The Monte Carlo method usually requires large number of samples as it depends on the "law of large numbers"
- As the number of intervals increases in the left Riemann sum, the computed value increases. This is because x^2 is an increasing function in [0, 1]. Similarly, the right Riemann sum decreases as we increase the number of partitions.

✓ 0s completed at 12:00

Introduction to Chaos Theory

Part 1 : Observations from the video.

Diverse natural phenomena like thermal convection in liquids, neural responses in our brain, the dripping of a faucet, the growth of population of rabbits in an area and the Mandelbrot set all have one aspect in common. This is the logistic equation!

$$x_{n+1} = rx_n(1 - x_n) \tag{1}$$

It is a recurrence relation. r is called growth rate. Upon analysis, it has been found that:

- For $0 < r < 1$, the x_n converges to zero, irrespective of the initial value of x.
- For $1 < r < 3$, x_n converges to some finite value which only depends on r. Irrespective of initial x, it finally converges to this value.
- Beyond $r = 3$, interesting things happen. There is no longer an equilibrium value. At first, x_n alternates between two values. As we increase r, it becomes four final values, and then eight, and then after a point, an infinite number of values are taken. It becomes aperiodic, and no value is taken more than once. In this chaotic region, there is sensitive dependence on both r and x_0. While these values after large number of iterations depend only on r, these values occur at different iterations based on x_0, and this causes the sensitive dependence in x_0 (we had already seen this in our code for logistic map with $r = 3.2$ and different values of x_0).
- At some places, there is a reprieve from chaos and stable cycles form again. For any arbitrary period, we can find a corresponding r for which the final cycle follows that period.

When these *equilibrium values* are plotted against r, we get the bifurcation diagram. The bifurcation pattern also occurs in the Mandelbrot set. When we see the final value(s) to which the Mandelbrot recurrence converges, we get the bifurcation diagram. One of the main reasons for this is that both the logistic map and the Mandelbrot map are equivalent. Under the linear transformation,

$$x_n = \frac{-z_n}{r} + \frac{1}{2} \tag{2}$$

the logistic map becomes the Mandelbrot map. When we start with $z_0 = 0$ in the Mandelbrot map, it is same as starting from $x_0 = \frac{1}{2}$ in our logistic map. But this period doubling bifurcations happen in a lot of other places.

- The nerve-response system in our brain also seems to follow this pattern when our eye is struck by periodic flickering light.
- The period doubling is seen in temperature in a fluid with convection due to temperature gradient.
- The heart rate of a rabbit follows this period doubling pattern during fibrillation. This helps us in knowing when to use electrical shocks to the heart to bring it back to periodicity.

Connection to chaos :

The logistic map is closely related to the chaos theory. A chaotic system is one whose solution is deterministic, but is sensitive to the initial conditions. As a result, even slightly tweaking these initial conditions may result in quite different solutions. *The future predicted by an approximate initial condition may be very different from the actual evolution of the system.*

What causes this sensitivity? A detour on phase space

A phase space is a space which represents all possible states which a system can take. Every state corresponds to a point in the phase space. Evolution of the system over time results in a *phase space trajectory.* The state can represent anything ranging from position-momentum pair to even x_n in the logistic map. One good aspect which comes from the idea of phase space is that we can now tell "how close" two states are, by numerically computing the Euclidean distance between them.

Now, lets say that our system is governed by some equation(s). It will have a solution set, where each solution in the set corresponds to one set of initial conditions. Let us think of the following scenarios now :

- Consider two "close by" initial states. Let these two states evolve according to the solutions governing them, and lets say we "freeze" the states after letting it evolve for some time.

- If the distance between the states has increased, we tell that our governing equations has *stretched out* the two points.
- If the distance between the states has decreased, we tell that our governing equations has *folded* the two points.

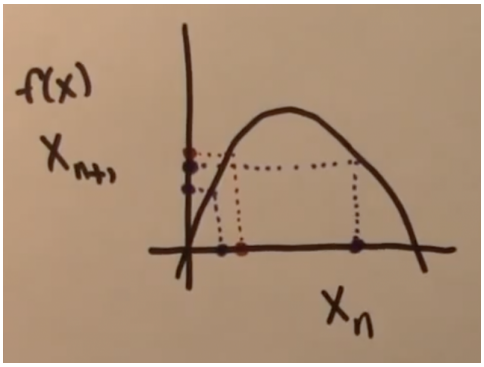
These stretching and folding operations done by our governing equation is the cause for chaos.

- Stretching results in nearby states going further apart. This results in prediction of future states difficult when we do not know the initial state accurately.
- Folding results in considerably far apart states to come close. This makes prediction of the past states inaccurate. Folding also ensures that phase space trajectories are bounded.

Stretching and folding mechanisms cloud both the past and the future.

But why is the logistic map chaotic?

Let us the consider the phase space of x_n. The phase space trajectory is an inverted parabola. And this is why it is chaotic! The map is non linear. There is an inherent stretching and folding due to the operations performed by our map. Below is an image illustrating these stretching and folding due to the map. We can see that two nearby x_n results in different x_{n+1},



but two values of x_n that are a bit far have nearby x_{n+1}.

Part 2 : Experiments to confirm chaotic behaviors

Any equation of motion governed by classical mechanics finally boils down to some differential equation(s). Usually, the solution becomes chaotic whenever the number of differential equations increase and/or when there is a non-linear term in the differential equations. While this statement is intuitive and not mathematically rigorous, it gives an idea of when we can expect chaos.

Note that in the case of logistic map, the system state is discrete and hence represented by a *difference equation(s)* rather than *differential equation(s)* as in continuous state systems.

The following are two simple experiments that can show chaotic behavior.

The double pendulum

In simple terms, a double pendulum is a pendulum attached to the end of another pendulum. These pendulums can be simple or compound (each is a rigid body). In order to avoid the pendulum colliding on itself, we can perform the experiment with compound pendulum. Upon solving by Lagrangian mechanics, we can see that its governing differential equations are highly non-linear, hence there is sensitive dependence on initial condition.

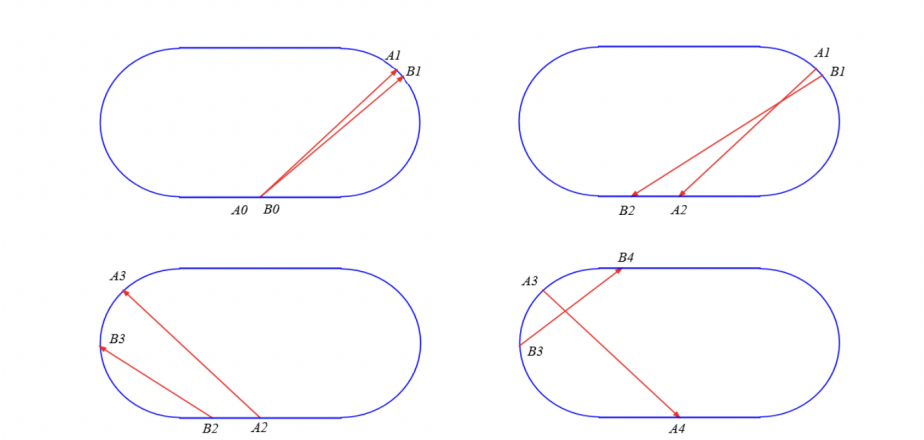
Resources needed to perform this experiment :

The resources needed are fairly common and simple too.

- Two uniformly dense metal arms with holes at their two ends.
- A stand for the double pendulum
- Any smooth ball bearing which can connect the two metal arms.
- We can also improvise on our experiment. We can take any photo luminescent material as a background, and attach some light source on the free end of our lower metal arm. This would illuminate the path traced by the lowermost point on the photo luminescent material, making the trajectory much clearer.

Dynamical billiards : Bunimovich Stadium

Another fairly simple experiment which is chaotic is the Bunimovich stadium. A stadium is a geometric shape constructed on a rectangle but with semicircles on a pair of opposite sides. Inside the stadium, we let a ball collide with the walls of this stadium. We assume that the collisions are perfectly elastic and that the ball doesn't experience any influence other than collisions i.e it travels in straight lines between successive collisions. The resultant motion is sensitive to initial conditions. Unlike the double pendulum, this doesn't have a differential equation to govern it - only the mechanics of elastic collisions apply. Yet, the system is chaotic. Below is an image illustrating this.



Resources needed to perform this experiment :

Again, the resources needed are fairly simple. However, we need to ensure that the materials we take have a high coefficient of restitution, so that the collisions are near-elastic. Stainless steel would be one such material.

- We would need to construct a stadium. We can try using 3D printing for this.
- We need a small ball bearing, which we will collide with the stadium.
- A projectile launcher, for launching our ball at a given angle and speed.
- We need to eliminate friction. This is not just for energy losses, but the friction can also cause unnecessary rotational motion on the ball. Lubricating the surface can help reduce friction.

There are several other interesting variants of this experiment such as putting some charge on our ball and introducing a magnetic field.

References

1. https://geoffboeing.com/2015/04/visualizing-chaos-and-randomness/
2. https://youtu.be/iDJ6ooKHw_c
3. https://en.wikipedia.org/wiki/Double_pendulum
4. https://youtu.be/mZ1hF_-cubA
5. https://en.wikipedia.org/wiki/Dynamical_billiards