AMS 595 Project 4: Mandelbrot Fractal, Markov Chains, and Taylor Approximation

Jane Condon

October 28, 2025

1 Introduction

There are 3 sections to this project. First, we compute the Mandelbrot set over the range [-2,1] x [-1.5,1.5] and construct an image of the Mandelbrot fractal. In the second part, we work with Markov chains and create a script that works with 5 states. In the third part, we write a script that can approximate any function using Taylor series approximation and measure the elapsed time as well as the absolute error for different polynomial degrees.

2 Computing the Mandelbrot Fractal

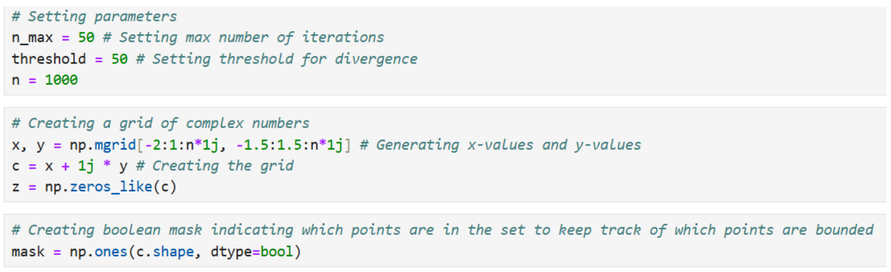
In this section, we write a Python script which computes the Mandelbrot set over the range [-2,1] x [-1.5,1.5], using a threshold of 50. Given the following iteration

$$z_{n+1} = z_n^2 + c, \quad z_i,\, c \in \mathbb{C}$$

consider that $z_0 = 0$. This iteration will be bounded or diverge to infinity, based on the value that is chosen for c. A complex number c is part of the Mandelbrot set M if, using c in the above iteration:

$$|z_i| < \infty \quad \text{for all } i > 0$$

To compute the Mandelbrot set with the specified parameters above, we start by creating a grid of complex numbers using the numpy library. Next, we create a boolean 'mask' variable indicating which points are in the set, to keep track of which points are bounded. This can be achieved with the following code:



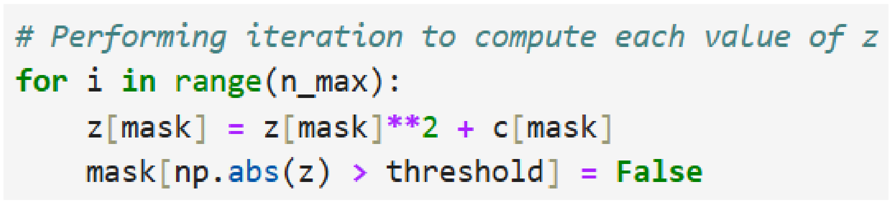
```python
# Setting parameters
n_max = 50 # Setting max number of iterations
threshold = 50 # Setting threshold for divergence
n = 1000
```

```python
# Creating a grid of complex numbers
x, y = np.mgrid[-2:1:n*1j, -1.5:1.5:n*1j] # Generating x-values and y-values
c = x + 1j * y # Creating the grid
z = np.zeros_like(c)
```

```python
# Creating boolean mask indicating which points are in the set to keep track of which points are bounded
mask = np.ones(c.shape, dtype=bool)
```

Figure 1: Code used to create a complex grid and boolean 'mask' variable.

To compute the values in the Mandelbrot set, we perform the following iteration:



```
# Performing iteration to compute each value of z
for i in range(n_max):
    z[mask] = z[mask]**2 + c[mask]
    mask[np.abs(z) > threshold] = False
```

Figure 2: Iteration performed to compute each value of z.

As shown above, we modify the iteration to update the value of z_{ij} only if:

$$|z_{ij}| < \text{threshold}$$

This way, we can avoid overflow. After computing the values inside the Mandelbrot set, we can plot an image of the Mandelbrot fractal using the matplotlib library.

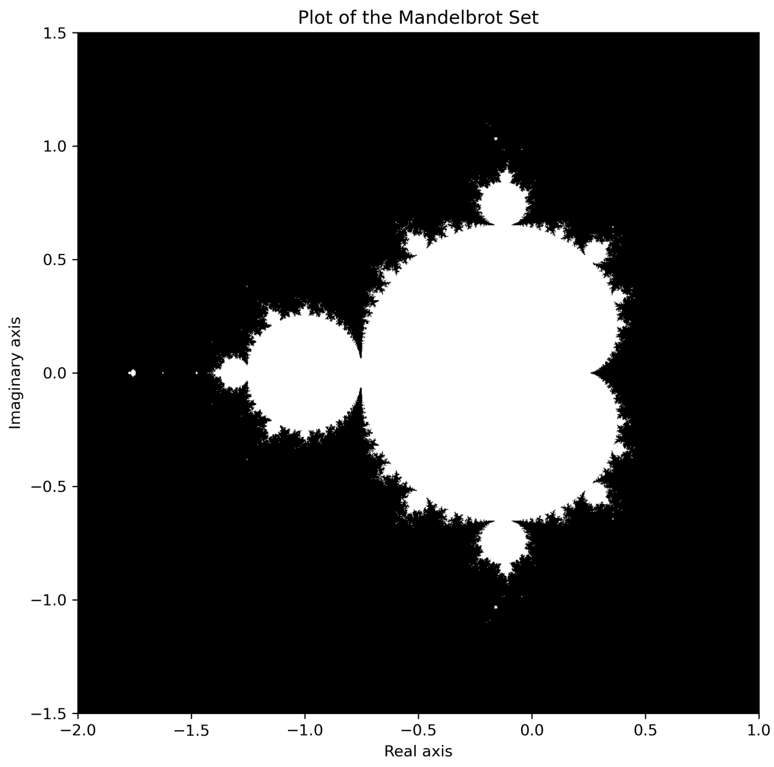


Figure 3: Plot of the Mandelbrot fractal using the points computed above.

3 Markov Chains

A Markov transition chain transition matrix P has a set of n states, where p_{ij} is the probability, where $0 \leq P_{ij} \leq 1$, of going from state i to state j. Each row is normalized so that:

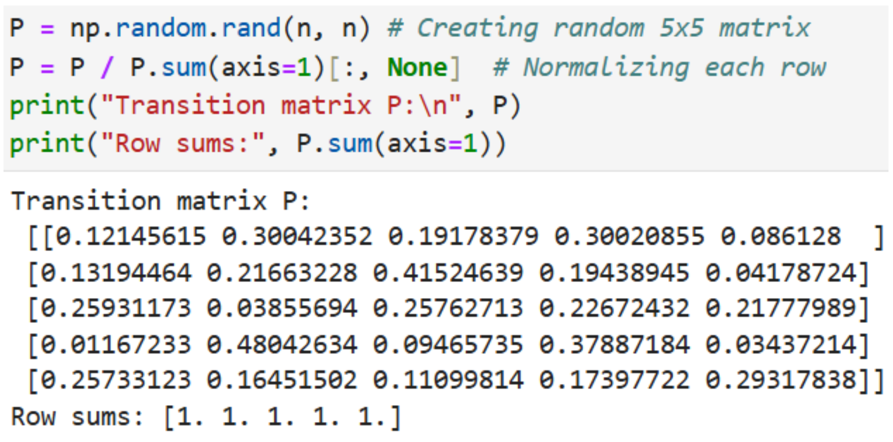
$$\sum_{j=1}^{n} P_{ij} = 1$$

In this section, we create a python script that works with 5 states.

3.1 Task 1: Constructing Random 5x5 Matrix with Normalized Rows

In this task, we use numpy to generate a random 5x5 matrix. Then, we normalize each row so that

$$\sum_{j=1}^{5} p_{ij} = 1$$



```
P = np.random.rand(n, n) # Creating random 5x5 matrix
P = P / P.sum(axis=1)[:, None]  # Normalizing each row
print("Transition matrix P:\n", P)
print("Row sums:", P.sum(axis=1))
```

```
Transition matrix P:
 [[0.12145615 0.30042352 0.19178379 0.30020855 0.086128  ]
 [0.13194464 0.21663228 0.41524639 0.19438945 0.04178724]
 [0.25931173 0.03855694 0.25762713 0.22672432 0.21777989]
 [0.01167233 0.48042634 0.09465735 0.37887184 0.03437214]
 [0.25733123 0.16451502 0.11099814 0.17397722 0.29317838]]
Row sums: [1. 1. 1. 1. 1.]
```

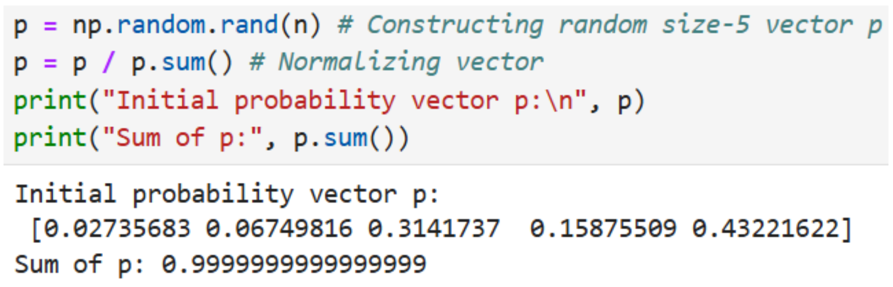
Figure 4: Code used to construct transition matrix P.

Note that each row sum is equal to 1, indicating that the rows have been normalized.

3.2 Task 2: Constructing a Random Normalized Size-5 Vector P

In this task, we construct a random size-5 vector p and normalize it so that:

$$\sum_{j=1}^{5} p_j = 1$$

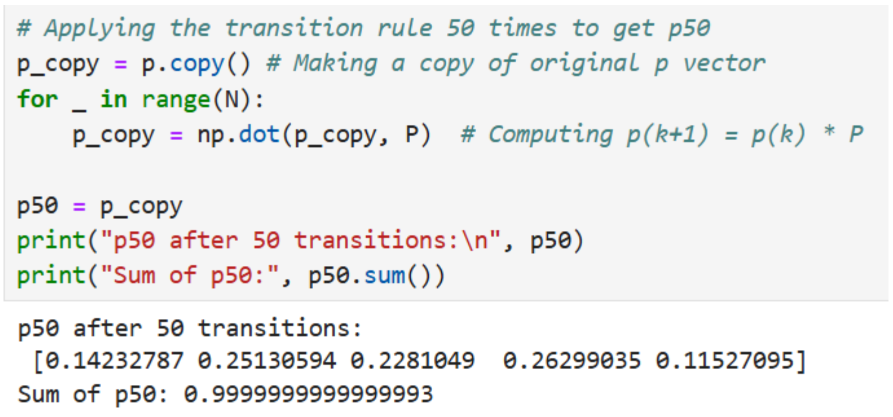


```
p = np.random.rand(n) # Constructing random size-5 vector p
p = p / p.sum() # Normalizing vector
print("Initial probability vector p:\n", p)
print("Sum of p:", p.sum())
```

```
Initial probability vector p:
 [0.02735683 0.06749816 0.3141737  0.15875509 0.43221622]
Sum of p: 0.9999999999999999
```

Figure 5: Code used to construct the initial probablity vector p.

Note that the sum of p is approximately equal to 1, indicating that it has been normalized. After creating this initial probability vector p, we apply the transition rule 50 times to obtain p_{50}.



```python
# Applying the transition rule 50 times to get p50
p_copy = p.copy() # Making a copy of original p vector
for _ in range(N):
    p_copy = np.dot(p_copy, P)  # Computing p(k+1) = p(k) * P

p50 = p_copy
print("p50 after 50 transitions:\n", p50)
print("Sum of p50:", p50.sum())
```

```
p50 after 50 transitions:
 [0.14232787 0.25130594 0.2281049  0.26299035 0.11527095]
Sum of p50: 0.9999999999999993
```

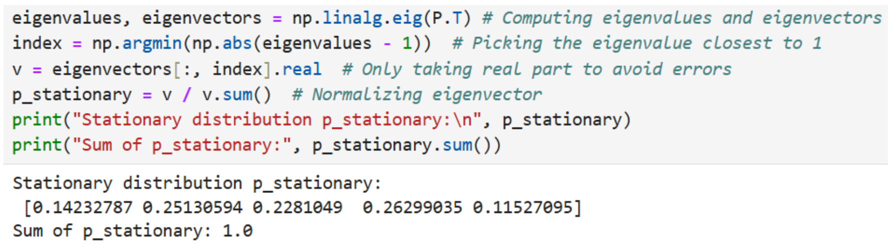
Figure 6: Code used to obtain probability vector p50 after applying transition rule 50 times.

3.3 Task 3: Obtaining the Stationary Distribution

To obtain the stationary distribution, we compute the eigenvector v of P^T corresponding to the eigenvalue 1 and then scale it so that

$$\sum_{j=1}^{5} v_j = 1$$

We can use the numpy library to solve for the eigenvalues and eigenvectors, pick the eigenvalue closest to 1 and its corresponding eigenvector, and normalize:



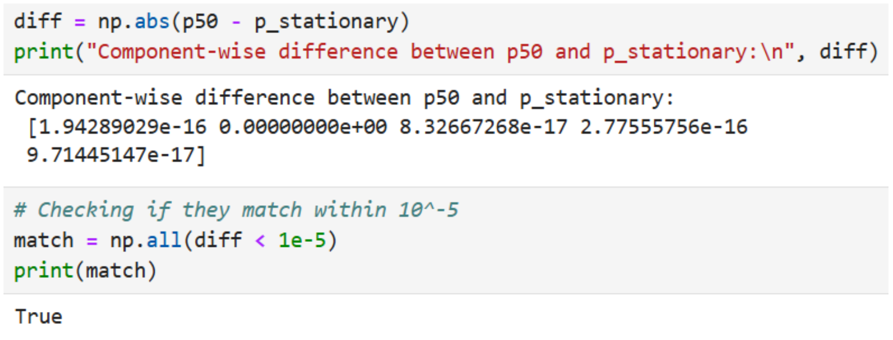
```python
eigenvalues, eigenvectors = np.linalg.eig(P.T) # Computing eigenvalues and eigenvectors
index = np.argmin(np.abs(eigenvalues - 1))  # Picking the eigenvalue closest to 1
v = eigenvectors[:, index].real  # Only taking real part to avoid errors
p_stationary = v / v.sum()  # Normalizing eigenvector
print("Stationary distribution p_stationary:\n", p_stationary)
print("Sum of p_stationary:", p_stationary.sum())
```

```
Stationary distribution p_stationary:
 [0.14232787 0.25130594 0.2281049  0.26299035 0.11527095]
Sum of p_stationary: 1.0
```

Figure 7: Code used to obtain and normalize eigenvector v.

3.4 Task 4: Computing Component Wise Difference Between p_{50} and the Stationary Distribution

Next, we check to see if the computed vector p_{50} matches the stationary distribution that we just calculated. In theory, these two vectors should be very similar. If 50 iterations is enough, they should be very close numerically. Otherwise, we would need to use more iterations.



```python
diff = np.abs(p50 - p_stationary)
print("Component-wise difference between p50 and p_stationary:\n", diff)
```

```
Component-wise difference between p50 and p_stationary:
 [1.94289029e-16 0.00000000e+00 8.32667268e-17 2.77555756e-16
 9.71445147e-17]
```

```python
# Checking if they match within 10^-5
match = np.all(diff < 1e-5)
print(match)
```

```
True
```

Figure 8: Code used to check component wise difference.

As shown above, p_{50} and the stationary distribution match within tolerance 10^{-5}.

4 Taylor Series Approximation

For a function f that is infinitely differentiable, its Taylor Series around a point c is the series:

$$\sum_{n=0}^{\infty} \frac{f^{(n)}(c)}{n!}(x-c)^n = f(c) + f'(c)(x-c) + \frac{f''(c)}{2}(x-c)^2 + \frac{f'''(c)}{3!}(x-c)^3 + \cdots$$

We can use the following equation to approximate a function using Taylor Series:

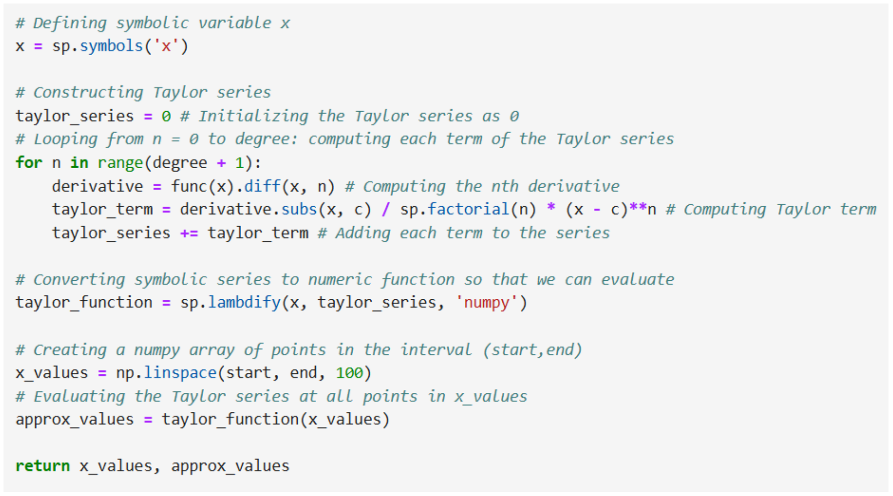
$$f(x) \approx \sum_{n=0}^{m} \frac{f^{(n)}(c)}{n!}(x-c)^n$$

4.1 Task 1: Writing a Function That Can Approximate a Function Using Taylor Series Approximation

In this section we write a function that takes the following inputs:

- func: function to approximate
- start: beginning of the interval
- end: end of the interval
- degree: last integer where you will be truncating your series.
- `fixed_c`: the point c which you will be expanding around

and outputs a numpy array of points which represent the approximation of function f within the specified interval. The function takes a user specified function and turns it into a symbolic function, then loops from 0 to the specified degree to compute each term of the Taylor Series. Within the loop, we compute the nth derivative and use this to compute the Taylor term. Each Taylor term is added to the Taylor series as the loop goes on.



```python
# Defining symbolic variable x
x = sp.symbols('x')

# Constructing Taylor series
taylor_series = 0 # Initializing the Taylor series as 0
# Looping from n = 0 to degree: computing each term of the Taylor series
for n in range(degree + 1):
    derivative = func(x).diff(x, n) # Computing the nth derivative
    taylor_term = derivative.subs(x, c) / sp.factorial(n) * (x - c)**n # Computing Taylor term
    taylor_series += taylor_term # Adding each term to the series

# Converting symbolic series to numeric function so that we can evaluate
taylor_function = sp.lambdify(x, taylor_series, 'numpy')

# Creating a numpy array of points in the interval (start,end)
x_values = np.linspace(start, end, 100)
# Evaluating the Taylor series at all points in x_values
approx_values = taylor_function(x_values)

return x_values, approx_values
```

Figure 9: Code inside the function.

4.2 Task 2: Testing the Function with Specified Parameters

In this section, we test the function with the following parameters:

- $f(x) = x\sin^2(x) + \cos(x)$
- Numerical domain: $[-10, 10]$ with 100 points
- Total of 100 terms in the series
- Expand around the point $c = 0$

and plot the actual function vs. the approximation computed by the function.

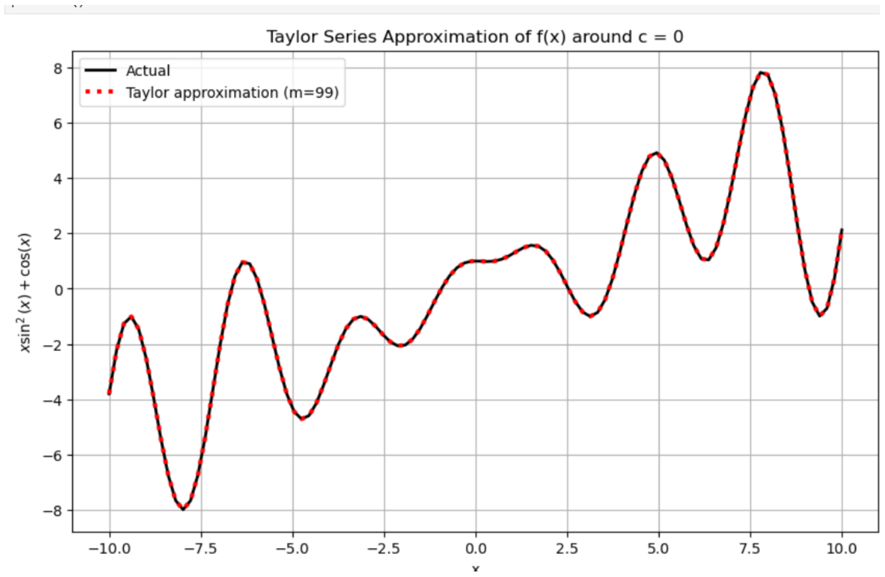


Figure 10: Plot of actual function vs. approximation for the specified parameters.

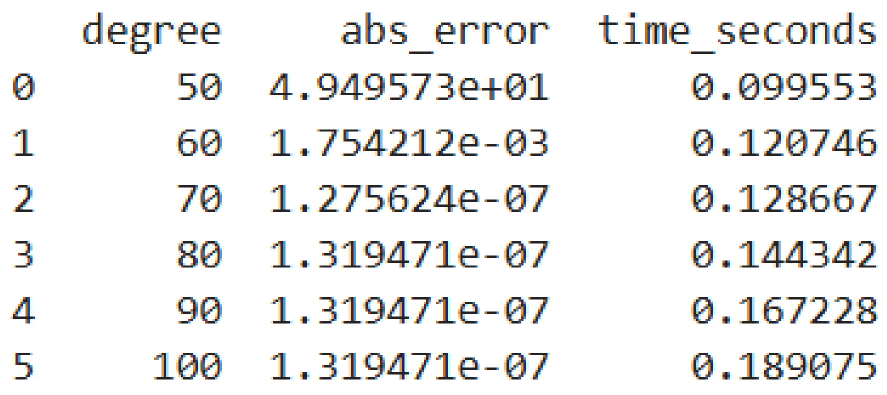
We use a solid line for the actual function, and a red dotted line for the approximated values.

4.3 Task 3: Modifying the Function to Measure Error and Elapsed Time for Different Polynomial Degrees

In this section, we add the following inputs to the previous function:

- initial degree
- final degree
- degree step

For example, we can use 50 as the initial degree, 100 as the final degree, and 10 as the degree step. The function will measure absolute error and elapsed time for every 10 polynomial degrees between 50-100 (e.g., 50, 60, 70, etc.).



```
   degree     abs_error  time_seconds
0      50  4.949573e+01      0.099553
1      60  1.754212e-03      0.120746
2      70  1.275624e-07      0.128667
3      80  1.319471e-07      0.144342
4      90  1.319471e-07      0.167228
5     100  1.319471e-07      0.189075
```

Figure 11: Results of the function for initial degree = 50, final degree = 100, degree step = 10.

As shown in the table above, using a higher polynomial degree is associated with higher accuracy, but it is computationally expensive compared to using a lower polynomial degree. Using 100 as the polynomial degree takes twice as long as using a polynomial degree of 50, but the absolute error is significantly lower, making the 100 degree polynomial a better choice in this scenario. Ultimately, there is a trade-off between accuracy and efficiency.