Numerical Analysis Project
MATH 5600
Homework 2

Authors:
Dane Gollero · Ike Griss Salas · Magon Bowling

Due: March 3, 2021

Contents

1 Homework Problems

-1- (Taylor Series.) Let

$$f(x) = e^x \text{ and } g(x) = \ln(x+1) \tag{1}$$

and let p_n and q_n be the Taylor polynomials of degree n for f and g, respectively, about

$$x_0 = 0. \tag{2}$$

Plot the graphs of f, g, p_n and q_n, for some small values of n, and comment on your results. Discuss in particular how well f and g are approximated by their Taylor polynomials. Explain your observations in terms of a suitable expression for the error in the approximation.

Recall the Taylor Series

$$f(x) = \sum_{i=0}^{\infty} f^{(i)}(\alpha) \cdot \frac{(x-\alpha)^i}{i!}.$$

We know that for all $i \in \mathbb{N}$,

$$f(x) = e^x \Rightarrow f^{(i)}(0) = 1.$$

Thus,

$$f(x) = e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!}.$$

We also have

$$\begin{aligned}
g(x) &= \ln(x+1) = 0 \quad \text{for } x = 0 \\
g'(x) &= \frac{1}{x+1} = 1 = 1! \qquad \text{at } x = 0 \\
g''(x) &= \frac{-1}{(x+1)^2} = -1 = -1! \qquad \text{at } x = 0 \\
g'''(x) &= \frac{2}{(x+1)^3} = 2 = 2! \qquad \text{at } x = 0 \\
g^{(4)}(x) &= \frac{-6}{(x+1)^4} = -6 = -3! \qquad \text{at } x = 0 \\
&\vdots \\
g^{(n)}(x) &= \frac{(-1)^{n-1}(n-1)!}{(x+1)^n} = (-1)^{n-1}(n-1)! \quad \text{for } x = 0, n > 0
\end{aligned}$$

$$\implies \sum_{n=1}^{\infty} (-1)^{n-1}(n-1)! \cdot \frac{x^n}{n!}.$$

Below, we will illustrate some Taylor polynomial approximations for f and g, respectively.[1]

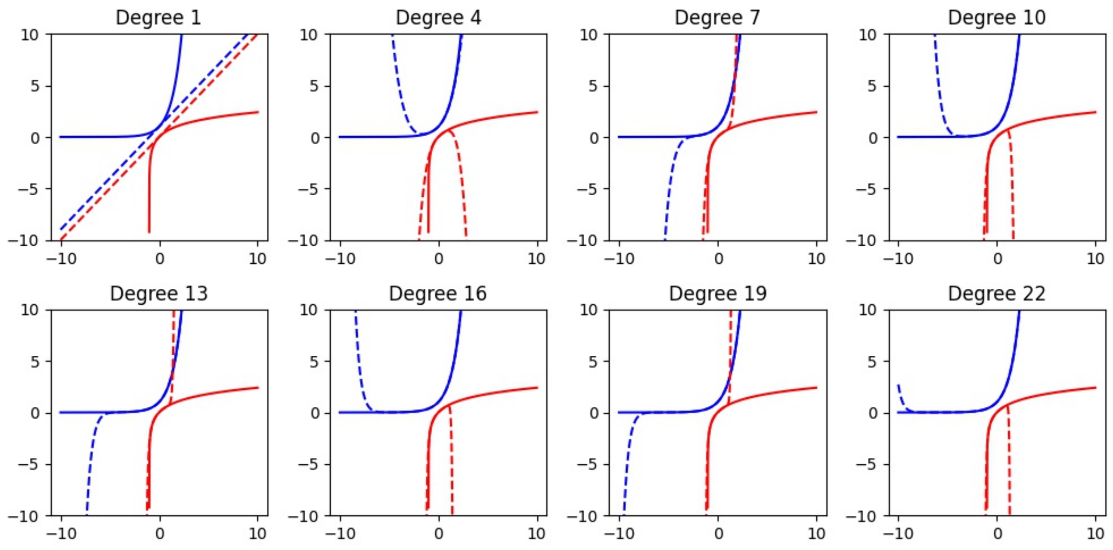


Figure 1: Solid Blue indicates e^x, Solid Red indicates $\ln(x+1)$, and the Dashed lines represent their Taylor approximations respectively.

We show a GIF of the Taylor polynomial approximations here: Taylor Polynomial GIF

Qualitatively we can see the Taylor polynomial, p_n, begins to better approximate e^x as n gets larger. However, the Taylor polynomial, q_n, tends to be problematic for larger n. It seems that q_n doesn't approximate $g(x)$ very well for $x > 1$, despite increasing values of n. We define the suitable

[1]Code in Appendix -A-.

error functions below.

$$E_n^f(x) = \left| e^x - \sum_{k=0}^{n} \frac{x^k}{k!} \right|$$

$$\begin{aligned} E_n^g(x) &= \left| \ln(x+1) - \sum_{k=1}^{n} (-1)^{k-1}(k-1)! \cdot \frac{x^k}{k!} \right| \\ &= \left| \ln(x+1) - \sum_{k=1}^{n} (-1)^{k-1} \frac{x^k}{k} \right| \end{aligned}$$

We show the graphs for our suitable errors as a function n for fixed $x \geq 1$.[2]

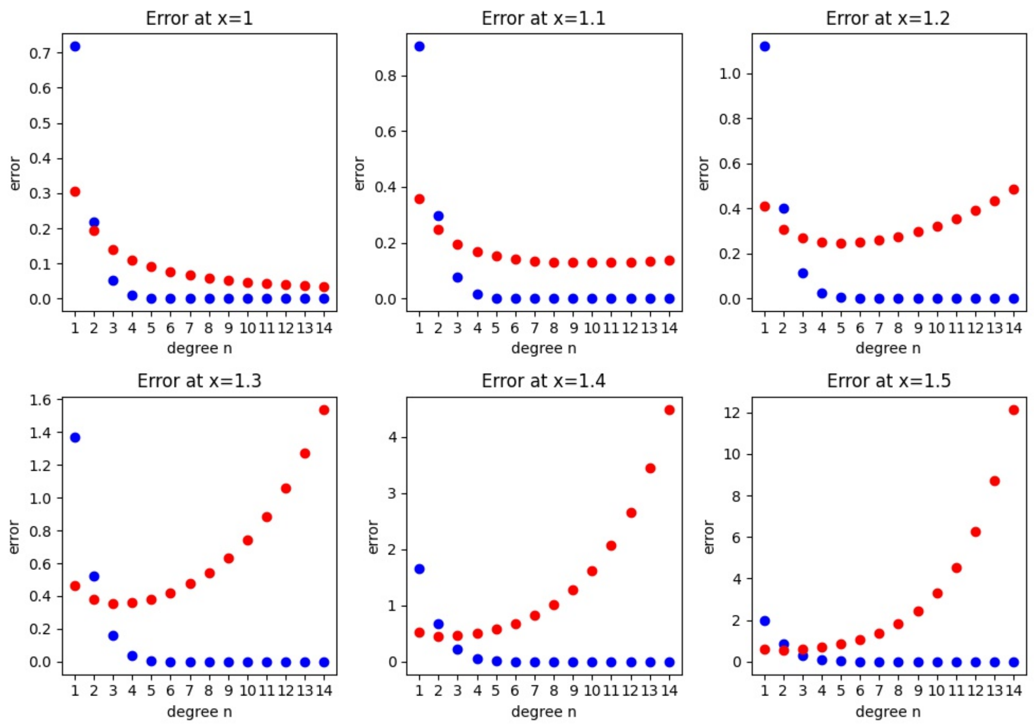


Figure 2: Blue dots indicate the error E_n^f and the Red dots indicate the error E_n^g.

As we can see increasing our polynomial degree for q_n does not imply a better approximation everywhere.

-2- (A "simple" program.) Write a program that reads n and the entries x_1, x_2, ..., x_n of a vector $x \in \mathbb{R}^2$ from standard input and prints

$$||x||_2 = \sqrt{\sum_{i=1}^{n} x_i^2}$$

to standard output. Mail me your code before the lecture on March 3.

[2]Code in Appendix -B-.

```
from math import sqrt

def norm(vec):
    square = [i**2 for i in vec]
    square_sum = sum(square)
    size = sqrt(square_sum)
    return size
```

-3- (Some Iteration.) Consider the iteration

$$x_{n+1} = F(x_n) = \sin x_n, \quad x_0 = 1 \tag{3}$$

(where of course the angle is measured in radians). What does our theory tell us about convergence? Show that the iteration does converge! What is the limit? How fast does the iteration converge? Carefully explain the effects of rounding errors.

We have the fixed point iteration $x_{n+1} = g(x_n)$ converges if we have $|g'(\alpha)| < 1$ where $g(\alpha) = \alpha$, and x_0 is sufficiently close to α. In our case, we have $g'(\alpha) = \cos(0) = 1$. On its own this tells us nothing, thus our convergence theory is not particularly helpful. However, we can say that it indeed converges. For x on the interval $[0, \frac{\pi}{2}]$ we know that

$$\sin x \leq x.$$

Thus if we assume x_0 is sufficiently close to 0 (on the interval) we have that

$$\begin{aligned} x_1 &= \sin(x_0) \leq x_0 \\ x_2 &= \sin(x_1) \leq \sin(x_0) = x_1 \\ &\vdots \end{aligned}$$

Inductively, we can see $\sin(x)$ is monotonically decreasing on $[0, \frac{\pi}{2}]$. And we know $\sin(x)$ defined on $[0, \frac{\pi}{2}]$ is bounded below by 0. By the Monotone Convergence Theorem, the sequence defined by $x_{n+1} = \sin(x_n)$ converges for x_0 sufficiently close to zero. Furthermore, since $\sin(x)$ is a continuous function on $[0, \frac{\pi}{2}]$ we can say

$$\lim_{x \to 0^+} \sin(x) = \sin(0) = 0$$

is equivalent to

$$\lim_{n \to \infty} \sin(x_n).$$

Therefore, our sequence converges to 0. This converges much slower than linearly since $g'(\alpha) = 1$. Below we show a table of runtimes for achieving certain error tolerances for this iteration. [3]

Error	Steps	Runtime (seconds)
0.1	295	5.39×10^{-5}
0.01	29992	0.005967
0.001	2999989	0.677399
0.0001	299999986	67.413355
0.00001	29999999984	6726.195585

[3] Runtime will differ for different computers. Code included in Appendix -C-.

We can see that $x_{n+1} = \sin(x_n)$ will take a substantial amount of time to converge. The issues of rounding errors can have catastrophic effects on the ability for our sequence to converge. If we have a machine that only computes to say the first 3 decimal points (i.e., it rounds), then the sequence will never converge. As the iteration progresses, the difference in outputs is less than 3 decimal places after a certain point, thus, the change in output becomes so minute that the algorithm will not converge to the right value. In fact, this becomes a constant when running it in Python. Letting $x_0 = 1$ we have

n	$\sin(x_n) = x_{n+1}$ (rounded)
0	0.841
1	0.745
2	0.678
$\vdots$	$\vdots$
125	0.145
126	0.144
127	0.144
128	0.144
$\vdots$	$\vdots$

The iteration settles on 0.144 after 126 steps. Now of course, we know that $\sin(x_n) = x_{n+1}$ for $x_0 = 1$ does in fact converges to zero from our deeper analysis. If we simply went to iteration via the computer, we may get misleading results.

-4- Newton's Method Suppose f has a root of multiplicity $p > 1$ at $x = \alpha$, i.e.,

$$f(\alpha) = f'(\alpha) = ... = f^{(p-1)}(\alpha) = 0, \quad f^{(p)}(\alpha) \neq 0. \tag{4}$$

a. Show that Newton's method applied to $f(x) = 0$ converges linearly to α.

b. Show that this modification of Newton's Method:

$$x_{k+1} = x_k - p\frac{f(x_k)}{f'(x_k)} \tag{5}$$

converges quadratically to α. **Hint:** You probably are thinking of using the Rule of L'Hopital, but the problem is much easier if you think of f as being defined by $f(x) = (x-\alpha)^p F(x)$ where $F(\alpha) \neq 0$.

(a)

$$\begin{aligned} g = x - \frac{f}{f'} &= x - \frac{(x-\alpha)^p F(x)}{p(x-\alpha)^{p-1}F(x) + (x-\alpha)^p F'(x)} \\ &= x - \frac{(x-\alpha)F(x)}{pF(x) + (x-\alpha)F'(x)} \end{aligned}$$

$$g' = 1 - \frac{\big(pF(x) + (x-\alpha)F'(x)\big)\big(F(x) + (x-\alpha)F'(x)\big) - (x-\alpha)F(x)\Big(\big(pF(x) + (x-\alpha)F'(x)\big)^2\Big)'}{\big(pF(x) + (x-\alpha)F'(x)\big)^2}$$

$$g'(\alpha) = 1 - \frac{p \cdot F(\alpha) \cdot F(\alpha)}{p^2 \cdot F(\alpha)^2} = 1 - \frac{1}{p} = \left|\frac{p-1}{p}\right| < 1, \quad \text{for all } p > 1$$

We have the absolute value of $g'(\alpha)$ such that $0 < g'(\alpha) < 1$ for all $p > 1$. Thus f converges linearly.
(b)

$$\begin{aligned} g(x) &= x - p\frac{f}{f'} \\ g'(x) &= 1 - p\left(\frac{f}{f'}\right)' \implies g'(\alpha) = 1 - p \cdot \frac{1}{p} = 0 \end{aligned}$$

This converges at least quadratically.

-5- (Division without division.) Suppose you have a computer or calculator that has no built-in division. Come up with a fixed point iteration that converges to $1/r$ for any given non-zero number r, and that only uses addition, subtraction, and multiplication. **Hint:** Write down an equation satisfied by $1/r$, apply Newton's method to that equation, and then modify Newton's method so that it doesn't use division. Your resulting method should converge of order 2.

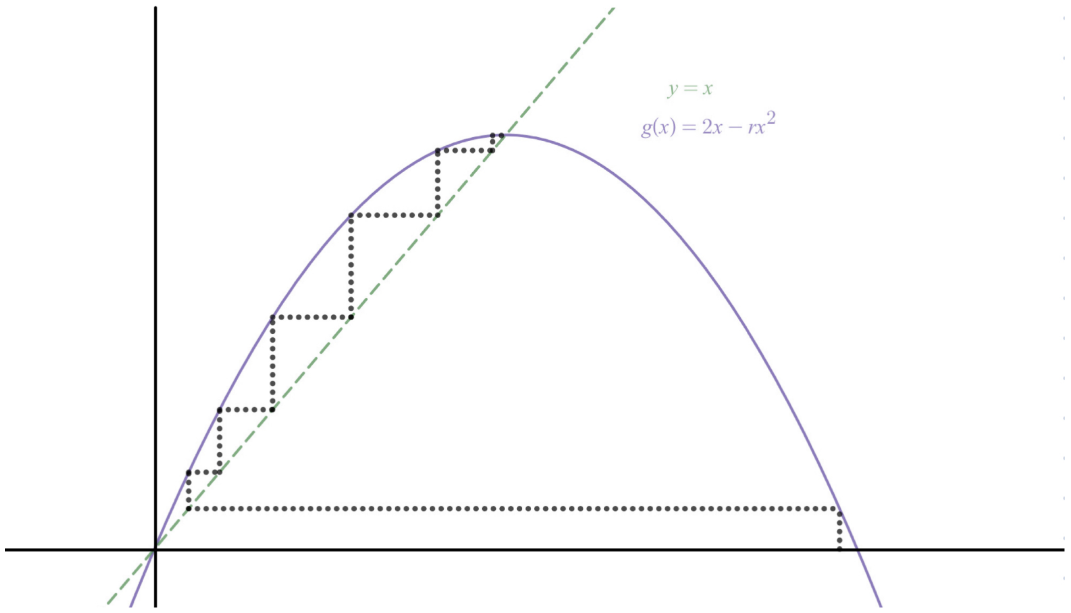
Let $f(x) = \frac{1}{x} - r$. Now f is satisfied by $x = \frac{1}{r}$ for $r \neq 0$. We have

$$\begin{aligned} f(x) &= \frac{1}{x} - r \\ f'(x) &= \frac{-1}{x^2} \\ f''(x) &= \frac{2}{x^3} \end{aligned}$$

Both f' and f'' are non-zero at $x = \frac{1}{r}$ meaning the Newton's method converges quadratically. We now look at Newton's method:

$$\begin{aligned} g(x) &= x - \frac{f}{f'} = x - \frac{\frac{1}{x} - r}{\frac{-1}{x^2}} \\ &= x - (-x + rx^2) \\ &= 2x - rx^2. \end{aligned}$$

Again, $g(x)$ converges quadratically since $g'(x) = 2 - 2rx$ is zero at $x = \frac{1}{r}$, and $g''(x) = -2r$ is non-zero for all x. Thus we have a fixed point iteration that does not use division. Below we show a hypothetical convergence for some r.



-6- (A cubically convergent method.) Consider the iteration

$$x_{k+1} = g(x_k) \text{ where } g(x) = x - \frac{f(x)}{f'(x)} - \frac{1}{2}\frac{f^2(x)f''(x)}{(f'(x))^3}.$$

(We assume f is sufficiently differentiable, and $f'(x) \neq 0$.) Suppose that $g(\alpha) = \alpha$. Show that

$$g'(\alpha) = g''(\alpha) = 0.$$

(Thus the fixed point method will converge of order at least 3 if we start sufficiently close to α.)

$$\begin{aligned}
g &= x - \frac{f}{f'} - \frac{1}{2}\frac{f^2 f''}{f'^3} \\
g' &= 1 - \frac{f'^2 - ff''}{f'^2} - \frac{1}{2}\frac{f'^3(2ff'f'' + f^2 f''') - 3f'^2 f'' f^2 f''}{f'^6} \\
&= \frac{ff''}{f'^2} - \frac{1}{2}\frac{2ff'^4 f''}{f'^6} - \frac{1}{2}\frac{f^2 f'^3 f'''}{f'^6} + \frac{1}{2}\frac{3f^2 f'^2 f''^2}{f'^6} \\
&= \frac{ff''}{f'^2} - \frac{ff''}{f'^2} - \frac{1}{2}\frac{f^2 f'''}{f'^3} + \frac{1}{2}\frac{3f^2 f''^2}{f'^4} \\
&= \frac{1}{2}f^2\left(\frac{f''^2}{f'^4} - \frac{f'''}{f'^3}\right) = 0 \quad \text{at } x = \alpha \text{ since } f(\alpha) = 0 \text{ and } f'(\alpha) \neq 0 \\
g'' &= ff'\left(\frac{f''^2}{f'^4} - \frac{f'''}{f'^3}\right) + \frac{1}{2}f^2\left(\frac{f''^2}{f'^4} - \frac{f'''}{f'^3}\right)' \\
&= 0 \quad \text{at } x = \alpha \text{ since each term has a multiple of } f \text{ which is 0 at } \alpha.
\end{aligned}$$

Thus our fixed point iteration converges of order at least 3.

-7- (Polynomial Interpolation.) Suppose you want to interpolate to the data $(x_i, y_i), i = 0, ..., n$ by a polynomial of degree n. Recall that the interpolating polynomial p can be written in its Lagrange form as

$$p(x) = \sum_{i=0}^{n} y_i L_i(x) \text{ where } L_i(x) = \frac{\prod_{i \neq j}(x - x_j)}{\prod_{i \neq j}(x_i - x_j)}. \tag{6}$$

Show that

$$\sum_{i=0}^{n} x_i^j L_i(x) = x^j \text{ for } j = 0, ..., n. \tag{7}$$

We first note that we have $n+1$ data points, thus any polynomial interpolation up to degree n will be unique, assuming each x_i is distinct. We have the points $\left(x_i, x_i^j\right)$ for $i = 0, 1, 2, ..., n$ and where j is a fixed power that can take on the values $j = 0, 1, 2, ..., n$, which lets us interpolate uniquely. Also, we assume all the points are distinct. By the construction of these points, the polynomial x^j interpolates our data. Since the interpolant is unique, we have

$$P(x) = \sum_{i=0}^{n} x_i^j L_i(x) = x^j \quad \text{for } j = 0, 1, ..., n.$$

-8- (Uniqueness of the interpolating polynomial.) Assume you are given the data

$$\begin{array}{rcccc} x_i: & 1 & 2 & 4 & 8 \\ y_i: & 1 & 2 & 3 & 4 \end{array} \tag{8}$$

Construct the interpolating polynomial using

a. the power form obtained by solving the Vandermonde system,

b. the Lagrange form,

c. the Newton form,
and show that they all yield the same polynomial.

(a) Vandermonde System: We have 4 nodes, thus we interpolate with polynomial of degree 3:

$$a + bx + cx^2 + dx^3 = y$$

$$\begin{aligned} a + b + c + d &= 1 \\ a + 2b + 4c + 8d &= 2 \\ a + 4b + 16c + 64d &= 3 \\ a + 8b + 64c + 512d &= 4 \end{aligned} \iff \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 \\ 1 & 4 & 16 & 64 \\ 1 & 8 & 64 & 512 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

Solving the corresponding augmented matrix reveals

$$\begin{bmatrix} a & b & c & d \end{bmatrix}^T = \begin{bmatrix} -\frac{10}{21} & \frac{7}{4} & -\frac{7}{24} & \frac{1}{56} \end{bmatrix}^T$$

$$\implies P(x) = -\frac{10}{21} + \frac{7}{4}x - \frac{7}{24}x^2 + \frac{1}{56}x^3$$

(b) Lagrange Form:

$$L_0(x) = \frac{(x-2)(x-4)(x-8)}{(-1)(-3)(-7)} = \frac{x^3-14x^2+56x-64}{-21}$$
$$L_1(x) = \frac{(x-1)(x-4)(x-8)}{(1)(-2)(-6)} = \frac{x^3-13x^2+44x-32}{12}$$
$$L_2(x) = \frac{(x-1)(x-2)(x-8)}{(3)(2)(-4)} = \frac{x^3-11x^2+26x-16}{-24}$$
$$L_3(x) = \frac{(x-1)(x-2)(x-4)}{(7)(6)(4)} = \frac{x^3-7x^2+14x-8}{168}$$

$$\begin{aligned}
\Longrightarrow P(x) = \sum_{i=0}^{3} y_i L_i(x) &= \frac{1}{-21}(x^3-14x^2+56x-64) + \frac{2}{12}(x^3-13x^2+44x-32) \\
&\quad - \frac{3}{24}(x^3-11x^2+26x-16) + \frac{4}{168}(x^3-7x^2+14x-8) \\
&= \frac{1}{169}(-8x^3+112x^2-448x+512+28x^3-364x^2+1232x-896 \\
&\quad -21x^3+231x^2-546x+336+4x^3-28x^2 56x-32) \\
&= \frac{1}{168}(3x^3-49x^2+294x-80) \\
&= \frac{1}{56}x^3 - \frac{7}{24}x^2 + \frac{7}{4}x - \frac{10}{21}
\end{aligned}$$

(c) Newton Form:

x	f	$F(x_i, x_j)$	$F(x_i, x_j, x_k)$	$F(x_i, x_j, x_k, x_\ell)$
1	1	$F(x_0, x_1) = \frac{f_1-f_0}{x_1-x_0} = 1$	$F(x_0, x_1, x_2) = \frac{F(x_1,x_2)-F(x_0,x_1)}{x_2-x_0} = -\frac{1}{6}$	$F(x_0, x_1, x_2, x_3) = \frac{F(x_1,x_2,x_3)-F(x_0,x_1,x_2)}{x_3-x_0} = \frac{1}{56}$
2	2	$F(x_1, x_2) = \frac{f_2-f_1}{x_2-x_1} = \frac{1}{2}$	$F(x_1, x_2, x_3) = \frac{F(x_2,x_3)-F(x_1,x_2)}{x_3-x_1} = -\frac{1}{24}$	
4	3	$F(x_2, x_3) = \frac{f_3-f_2}{x_3-x_2} = \frac{1}{4}$		
8	4			

$$\begin{aligned}
P(x) &= f_0 + (x-x_0)F(x_0,x_1) + (x-x_0)(x-x_1)F(x_0,x_1,x_2) + (x-x_0)(x-x_1)(x-x_2)F(x_0,x_1,x_2,x_3) \\
&= 1 + (x-1)\cdot 1 + (x-1)(x-2)\left(-\frac{1}{6}\right) + (x-1)(x-2)(x-4)\left(\frac{1}{56}\right) \\
&= 1 + x - 1 - \frac{1}{6}(x^2-3x+2) + \frac{1}{56}(x^3-7x^2+14x-8) \\
&= x - \frac{1}{6}x^2 + \frac{1}{2}x - \frac{1}{3} + \frac{1}{56}x^3 - \frac{1}{8}x^2 + \frac{1}{4}x - \frac{1}{7} \\
&= \frac{1}{56}x^3 - \frac{7}{24}x^2 + \frac{7}{4}x - \frac{10}{21}
\end{aligned}$$

All of these interpolating methods yield the same polynomial.

-9- (The infamous Runge-Phenomenon.) It is not generally true that higher degree interpolation polynomials yield more accurate approximations. This is illustrated in this problem.

Let

$$f(x) = \frac{1}{1+x^2} \quad \text{and} \quad x_j = -5 + jh, \quad j = 0, 1, ..., n, \quad h = \frac{10}{n}.$$

For

$$n = 1, 2, 3, ..., 20$$

plot the graph (in the interval $[-5, 5]$) of the interpolant.

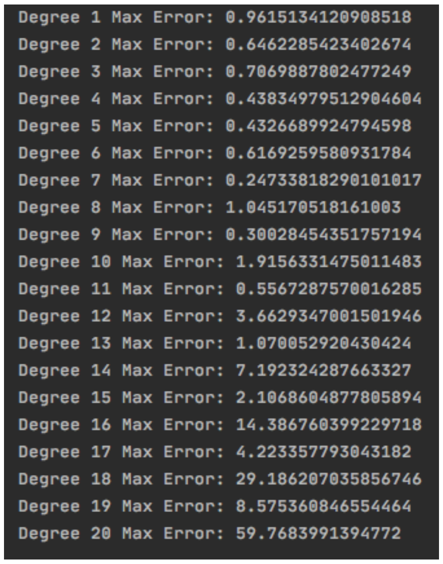
$$p(x) = \sum_{i=0}^{n} \alpha_i x^i$$

defined by

$$p(x_i) = f(x_i), \quad i = 0, 1, ..., n.$$

Also list the approximate maximum error in the interval $[-5, 5]$ for each polynomial degree. To approximate the maximum error sample the error at 200 evenly spaced points (at least!) in the interval.

We have the interpolants in the form of a GIF here: Runge-Phenomenon Interpolant.
We can see that the maximum error increases in size as the degree of the interpolating polynomial increases.[4]



```
Degree 1 Max Error: 0.9615134120908518
Degree 2 Max Error: 0.6462285423402674
Degree 3 Max Error: 0.7069887802477249
Degree 4 Max Error: 0.43834979512904604
Degree 5 Max Error: 0.4326689924794598
Degree 6 Max Error: 0.6169259580931784
Degree 7 Max Error: 0.24733818290101017
Degree 8 Max Error: 1.045170518161003
Degree 9 Max Error: 0.30028454351757194
Degree 10 Max Error: 1.9156331475011483
Degree 11 Max Error: 0.5567287570016285
Degree 12 Max Error: 3.6629347001501946
Degree 13 Max Error: 1.070052920430424
Degree 14 Max Error: 7.192324287663327
Degree 15 Max Error: 2.1068604877805894
Degree 16 Max Error: 14.386760399229718
Degree 17 Max Error: 4.223357793043182
Degree 18 Max Error: 29.186207035856746
Degree 19 Max Error: 8.575360846554464
Degree 20 Max Error: 59.7683991394772
```

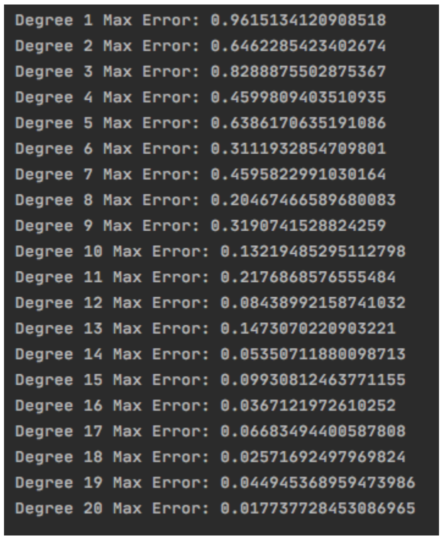
-10- (Judicious interpolation.) Repeat the above except that you interpolate at the roots of the Chebycheff polynomials, i.e.,

$$x_i = 5\cos\frac{i\pi}{n}, \quad i = 0, 1, ..., n. \tag{9}$$

We have the interpolants in the form of a GIF here: Judicious Interpolant.
It is true for most of the maximum errors below that increasing the degree of the Interpolant

[4]Code in Appendix -D-.

decreases the maximum error with respect to the previous error. That being said, there were a few outliers such as the 9th error compared to the 8th error.[5]



```
Degree 1 Max Error: 0.9615134120908518
Degree 2 Max Error: 0.6462285423402674
Degree 3 Max Error: 0.8288875502875367
Degree 4 Max Error: 0.4599809403510935
Degree 5 Max Error: 0.6386170635191086
Degree 6 Max Error: 0.3111932854709801
Degree 7 Max Error: 0.4595822991030164
Degree 8 Max Error: 0.20467466589680083
Degree 9 Max Error: 0.3190741528824259
Degree 10 Max Error: 0.13219485295112798
Degree 11 Max Error: 0.2176868576555484
Degree 12 Max Error: 0.08438992158741032
Degree 13 Max Error: 0.1473070220903221
Degree 14 Max Error: 0.05350711880098713
Degree 15 Max Error: 0.09930812463771155
Degree 16 Max Error: 0.0367121972610252
Degree 17 Max Error: 0.06683494400587808
Degree 18 Max Error: 0.02571692497969824
Degree 19 Max Error: 0.044945368959473986
Degree 20 Max Error: 0.017737728453086965
```

-11- (Least Squares approximation of functions.) Find a linear function $\ell(x)$ such that

$$\int_0^1 (e^x - \ell(x))^2 dx = \min . \tag{10}$$

If we let $\ell(x) = mx + b$ for $m, b \in \mathbb{R}$, then we can think of equation (10) as a function of m and b

$$F(m,b) = \int_0^1 (e^x - mx - b)^2 dx.$$

We want to minimize $F(m,b)$, so we take partial derivatives

$$\begin{aligned}
\frac{\partial F}{\partial m} &= \frac{\partial}{\partial m}\int_0^1 (e^x - mx - b)^2 dx \\
&= \int_0^1 \frac{\partial}{\partial m}(e^x - mx - b)^2 dx \\
&= \int_0^1 2(e^x - mx - b)(-x) dx \\
&= 2\int_1^0 x(e^x - mx - b) dx \\
&= 2\left[\left(xe^x - \frac{m}{2}x^3 - bx^2\right)\Big|_1^0 - \left(e^x - \frac{m}{6}x^3 - \frac{b}{2}x^2\right)\Big|_1^0\right] \\
&= 2\left[-\left(e - \frac{m}{2} - b\right) - \left(1 - \left(e - \frac{m}{6} - \frac{b}{2}\right)\right)\right] \\
&= 2\left(-e + \frac{m}{2} + b - 1 + e - \frac{m}{6} - \frac{b}{2}\right) \\
&= 2\left(-1 + \frac{m}{3} + \frac{b}{2}\right) \\
&= b + \frac{2m}{3} - 2
\end{aligned}$$

[5]Code in Appendix -E-.

$$\begin{aligned}\frac{\partial F}{\partial b} &= \frac{\partial}{\partial b}\int_0^1 (e^x - mx - b)^2 dx\\ &= \int_0^1 \frac{\partial}{\partial b}(e^x - mx - b)^2 dx\\ &= \int_0^1 2(e^x - mx - b)(-1)dx\\ &= 2\int_1^0 (e^x - mx - b)dx\\ &= 2\left[\left(e^x - \frac{m}{2}x^2 - bx\right)\Big|_1^0\right]\\ &= 2\left[1 - \left(e - \frac{m}{2} - b\right)\right]\\ &= 2\left(\frac{m}{2} + b + 1 - e\right)\\ &= m + 2b + 2 - 2e\end{aligned}$$

To minimize, we set partial derivatives equal to zero and solve for m and b.

$$m + 2b + 2 - 2e = 0$$

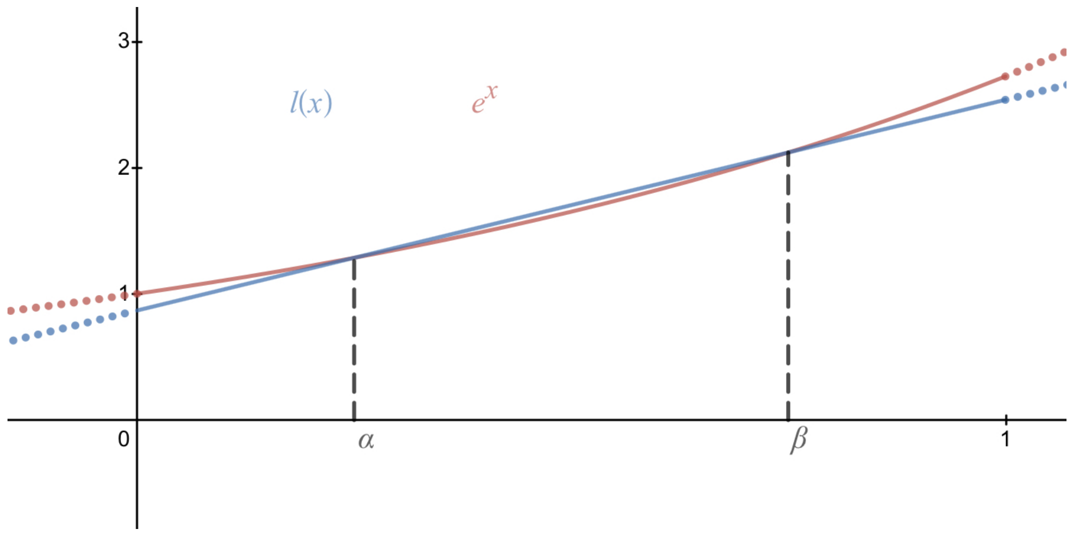
$$b + \frac{2m}{3} - 2 = 0$$

Solving the linear system yields $b = 4e - 10$ and $m = 18 - 6e$. Thus, $\ell(x) = (18 - 6e)x + (4e - 10)$ minimizes equation (10).

-12- (An alternative approximation problem.) Find a linear function $\ell(x)$ such that

$$\int_0^1 |e^x - \ell(x)|dx = \min. \tag{11}$$

We know that our linear function $\ell(x)$ must actually intersect e^x twice on the interval $[0, 1]$. The graph illustrates the points of intersection α and β, some arbitrary points.



We want to minimize the following

$$I = \int_0^1 |e^x - \ell(x)|dx = \int_0^\alpha (e^x - \ell(x))dx + \int_\alpha^\beta (\ell(x) - e^x)dx + \int_\beta^1 (e^x - \ell(x))dx.$$

If we let $\ell(x) = mx + b$, the following evaluates to

$$\begin{aligned}
I &= \left[e^x - \frac{m}{2}x^2 - bx\right]\Bigg|_0^\alpha + \left[\frac{m}{2}x^2 + bx - e^x\right]\Bigg|_\alpha^\beta + \left[e^x - \frac{m}{2}x^2 - bx\right]\Bigg|_\beta^1 \\
&= \left(e^\alpha - \frac{\alpha^2}{2}m - \alpha b - 1\right) + \left(\frac{\beta^2}{2}m + \beta b - e^\beta\right) - \left(\frac{\alpha^2}{2}m + \alpha b - e^\alpha\right) \\
&\qquad + \left(e - \frac{1}{2}m - b\right) - \left(e^\beta - \frac{\beta^2}{2}m - \beta b\right) \\
&= e^\alpha - \frac{\alpha^2}{2}m - \alpha b - 1 + \frac{\beta^2}{2}m + \beta b - e^\beta - \frac{\alpha^2}{2}m - \alpha b + e^\alpha \\
&\qquad + e - \frac{1}{2}m - b - e^\beta + \frac{\beta^2}{2}m + \beta b
\end{aligned}$$

$$F(m,b) = I = \left(\beta^2 - \alpha^2 - \frac{1}{2}\right)m + (2\beta - 2\alpha - 1)\,b + \left(2e^\alpha - 2e^\beta + e - 1\right)$$

$$\frac{\partial F}{\partial m} = \beta^2 - \alpha^2 - \frac{1}{2} = 0 \Longrightarrow (\beta - \alpha)(\beta + \alpha) = \frac{1}{2}$$

$$\frac{\partial F}{\partial b} = 2\beta - 2\alpha - 1 = 0 \Longrightarrow \beta - \alpha = \frac{1}{2}$$

Therefore we have,

$$\alpha = \frac{1}{4} \text{ and } \beta = \frac{3}{4}.$$

Now

$$m = \frac{e^\beta - e^\alpha}{\beta - \alpha} = 2\left(e^{3/4} - e^{1/4}\right)$$

which means that

$$y = 2\left(e^{3/4} - e^{1/4}\right)x + b \Longrightarrow e^{1/4} = \frac{1}{2}e^{3/4} - \frac{1}{2}e^{1/4} + b, \text{ and}$$

$$b = \frac{1}{2}\left(3e^{1/4} - e^{3/4}\right).$$

Therefore,

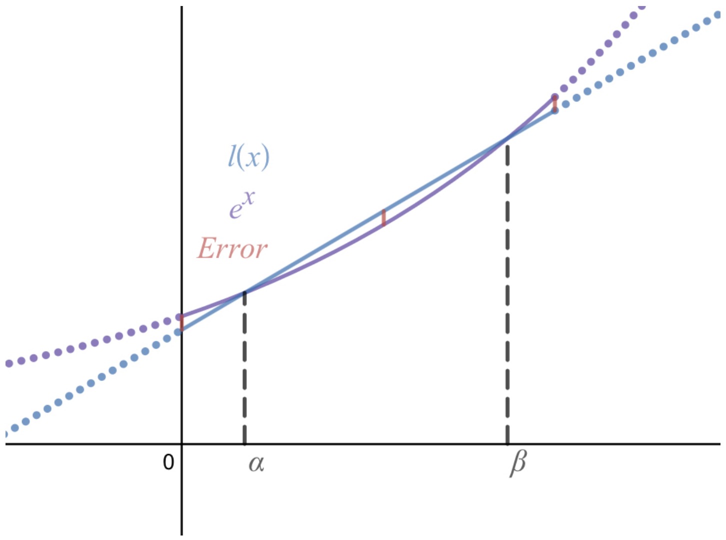
$$\ell(x) = 2\left(e^{3/4} - e^{1/4}\right)x + \frac{1}{2}\left(3e^{1/4} - e^{3/4}\right) \quad \text{minimizes (11).}$$

-13- (Another alternative approximation problem.) Find a linear function $\ell(x)$ such that

$$\max_{0 \le x \le 1} |e^x - \ell(x)| = \min\,. \qquad (12)$$

We know that our function $\ell(x)$ will intersect e^x twice, say at α and β. However, we won't focus on these values all that much. The idea is that the maximum error will occur at one of these intervals. We wish to minimize the maximum error, to do this the maximum error on each interval

$[0,\alpha],[\alpha,\beta],\text{and}[\beta,1]$ must be identical. We can always adjust the line to reduce the error. We can see what this looks like, qualitatively, below.



We know that based on the shape of e^x on the interval, the minimized maximum error will occur at $x=0, x=1$ and some $\hat{x}$ on the interval $[\alpha,\beta]$. If we let $\ell(x)=mx+b$ and let E denote the minimized maximum error, then we have the following system,

$$e^0-(m(0)+b)=E$$

$$e^{\hat{x}}-(m\hat{x}+b)=-E$$

$$e^1-(m(1)+b)=E$$

We also note that $E'(\hat{x})=e^{\hat{x}}-m=0 \Leftrightarrow e^{\hat{x}}=m \Leftrightarrow \hat{x}=\ln(m)$. Solving the system we have,

$$\begin{aligned} e-m-b&=E \\ (-)\quad 1-b&=E \end{aligned}$$

$$\Longrightarrow \quad e-1-m=0 \quad \Longleftrightarrow \quad m=e-1$$

$$\begin{aligned} e-m-b&=E \\ (+)\quad e^{\hat{x}}-m\hat{x}-b&=-E \end{aligned}$$

$$\Longrightarrow \quad e+e^{\hat{x}}-m-m\hat{x}-2b=0$$

$$\begin{aligned} \Longrightarrow \quad 2b&=e+e^{\hat{x}}-m-m\hat{x} \\ b&=\frac{1}{2}(e-m\hat{x}) \\ &=\frac{1}{2}(e-(e-1)\ln(e-1)) \\ &=\frac{1}{2}(e-e\ln(e-1)+\ln(e-1)) \end{aligned}$$

Thus, $\ell(x)=(e-1)x+\frac{1}{2}(e-e\ln(e-1)+\ln(e-1))$ minimizes our function.

2 Appendix

-A- Python Code for Graphs in problem 1.

```
import matplotlib.pyplot as plt
import numpy as np
from math import factorial

t = np.linspace(-10, 10, 1000)

# I need this one for the ln function since I don't know how to
# deal with singularity of -1
t_ln = np.linspace(-0.9999, 10, 1000)

# Define e^x
def f(x):
    return np.exp(x)

# Define ln(x+1)
def g(x):
    return np.log(x+1)

# Define taylor poly for e^x
def p(x,n):
    if n == 0:
        return 1
    else:
        return ((x**n) / factorial(n)) + p(x, n-1)

# Define taylor poly for ln(x+1)
def q(x,n):
    if n == 1:
        return x
    else:
        return (-1)**(n-1) * factorial(n-1)*x**n / factorial(n)
            + q(x, n-1)

# For subplots
fig, ax = plt.subplots(2, 4, figsize=(10,5))
i = 0
for n in range(1, 23, 3):
    # print(n)
    test = ax[i // 4, i % 4]
    test.plot(t, f(t), color='blue', label='e^x')
    test.plot(t, p(t, n), color='blue', linestyle='--', label='p')
    test.plot(t_ln, g(t_ln), color='red', label='ln(1+x)')
    test.plot(t, q(t, n), color='red', linestyle='--', label='q')
    test.set_ylim(-10, 10)
```

```
        test.set_title(f'Degree {n}')
        i+=1

    fig.show()
```

-B- Python Code for Errors in problem 1.

```
import numpy as np
from math import factorial
import matplotlib.pyplot as plt

def f(x):
    return np.exp(x)

# Define ln(x+1)
def g(x):
    return np.log(x+1)

# Define taylor poly for e^x
def p(x,n):
    if n == 0:
        return 1
    else:
        return ((x**n) / factorial(n)) + p(x, n-1)

# Define taylor poly for ln(x+1)
def q(x,n):
    if n == 1:
        return x
    else:
        return (-1)**(n-1) * factorial(n-1)*x**n / factorial(n)
            + q(x, n-1)

def error_f(x,n):
     return abs(f(x) - p(x, n))

def error_g(x,n):
    if x <= -1:
        return 'undefined'
    else:
        return abs(g(x) - q(x, n))

# Parameters for graph
x = 1
n = 14

domain = [i for i in range(1, n+1)]
```

```
codomain_f = [error_f(x, i) for i in range(1, n+1)]
codomain_g = [error_g(x, i) for i in range(1, n+1)]

print("e^x errors:")
print('----------')
for i in range(0,n):
    print(f'degree {i+1}: {codomain_f[i]}')
print()

print("ln errors:")
print('----------')
for i in range(0,n):
    print(f'degree {i+1}: {codomain_g[i]}')

x_list = [1, 1.1, 1.2, 1.3, 1.4, 1.5]
fig, ax = plt.subplots(2, 3, figsize=(10, 7))

for j in range(6):
    codomain_f = [error_f(x_list[j], i) for i in range(1, n + 1)]
    codomain_g = [error_g(x_list[j], i) for i in range(1, n + 1)]
    test = ax[j // 3, j % 3]
    x_ticks = np.arange(1, n+1, 1)
    test.set_xticks(x_ticks)
    test.set_title(f'Error at x={x_list[j]}')
    test.scatter(x=domain, y=codomain_f, color='blue',
        label='e^x approximation error')
    test.scatter(x=domain, y=codomain_g, color='red',
        label='ln approximation error')
    test.set_xlabel('degree n')
    test.set_ylabel('error')


fig.show()
```

-C- Python Code for Runtimes in problem 3.

```
from math import sin
import time

# My first program to test how many steps it takes to get epsilon
# distance from 0
i = 0
x = 1
tolerance = 0.001
flag1 = time.process_time()
while True:
    i += 1
```

```
        x = sin(x)
        if x < tolerance:
            flag2 = time.process_time()
            break
    time_length = flag2-flag1
    print(f'error: {x}\n', f'steps: {i}\n', f'time: {time_length}')
    print()
    # Code to look at how at effects of rounding errors.
    i = 0
    x = 1
    n = 300
    rounding = 3

    for step in range(n):
        i += 1
        x = round(sin(x), rounding)
        print(f'{step}: {x}')
```

-D- Python Code for Interpolants in problem 9.

```
import matplotlib.pyplot as plt
import numpy as np
# from scipy import interpolate, linalg

# Define a linespace (or domain)
domain = np.linspace(-5, 5, 1000)

# Define the function
def f(x):
    return 1/(1+x**2)

# Interpolate with different degree polynomials
for n in range(1,21):
    h = 10 / n

    # creating our nodes
    x = [-5+i*h for i in range(n+1)]
    y = [f(i) for i in x]

   # Creating the Vondermond Matrix
    rows = []
    for i in x:
        row = [i**j for j in range(n+1)]
        rows.append(row)

    # Create my system that needs to be solved
    x_vondermond = np.array(rows)
    y_vondermond = np.array(y)

    # Solve the system of equations!
    sol = np.linalg.solve(x_vondermond, y_vondermond)

    # I built a function recursively.
    def poly(x, n):
        if n == 0:
            return sol[0]
        else:
            return sol[n]*x**n + poly(x, n-1)

    # Find the max error
    error = [abs(f(i)-poly(i, n)) for i in domain]
    max_error = max(error)
    max_error_index = error.index(max_error)
    print(f"Degree␣{n}␣Max␣Error:␣{max_error}")
    plot_value = -5 + max_error_index/100
    # You divide by 1000 and then multiply by 10--think about it...

    # Plot the results
    plt.plot(domain, f(domain), label='function', color='blue')
    plt.plot(domain, poly(domain,n), label='Interpolant')
    plt.ylim([-0.5, 1.5]) #To fix plotting frame.
    plt.text(-2, 1.25, f'Max␣Error:␣{max_error}')
    plt.vlines(x=plot_value, ymin=min(f(plot_value),
        poly(plot_value, n)), ymax=max(f(plot_value),
        poly(plot_value, n)), linestyle='--', color='red',
        label="Max␣Error")
    plt.title(f"Degree␣{n}␣Interpolation")
    plt.legend(loc=8)
    # plt.savefig(f'degree_{n}.jpg', bbox_inches='tight')
    plt.show()
```

-E- Python Code for Interpolants in problem 10.

```
# Chebycheff Polynomial
import matplotlib.pyplot as plt
import numpy as np
# from scipy import interpolate, linalg

# Define a linespace (or domain)
domain = np.linspace(-5, 5, 1000)

# Define the function
def f(x):
    return 1/(1+x**2)
```

```
# Interpolate with different degree polynomials
for n in range(1, 21):

    # creating our nodes
    x = [5*np.cos(i*np.pi/n) for i in range(n+1)]
    y = [f(i) for i in x]

    # Creating the Vondermond Matrix
    rows = []
    for i in x:
        row = [i**j for j in range(n+1)]
        rows.append(row)

    # Create my system that needs to be solved
    x_vondermond = np.array(rows)
    y_vondermond = np.array(y)

    # Solve the system of equations!
    sol = np.linalg.solve(x_vondermond, y_vondermond)

    # I built the function recursively.
    def poly(x, n):
        if n == 0:
            return sol[0]
        else:
            return sol[n]*x**n + poly(x, n-1)

    # Find the max error
    error = [abs(f(i)-poly(i, n)) for i in domain]
    max_error = max(error)
    max_error_index = error.index(max_error)
    print(f"Degree {n} Max Error: {max_error}")
    plot_value = -5 + max_error_index/100
    # You divide by 1000 and then multiply by 10--think about it...

    # Plot the results
    plt.plot(domain, f(domain), label='function', color='blue')
    plt.plot(domain, poly(domain,n),
        label='Chebycheff Interpolant')
    plt.text(0, 0.1, f'Max Error: {max_error}',
        horizontalalignment='center', verticalalignment='center')
    plt.vlines(x=plot_value, ymin=min(f(plot_value),
        poly(plot_value, n)), ymax=max(f(plot_value),
        poly(plot_value, n)), linestyle='--', color='red',
        label="Max Error")
    plt.title(f"Degree {n} Chebycheff Interpolation")
    plt.legend(loc=1)
```

```
# plt.savefig(f'chebycheff_{n}.jpg', bbox_inches='tight')
plt.show()
```