HW_6_code

April 30, 2022

[1]:
```python
import numpy as np
from scipy.interpolate import lagrange
from numpy.polynomial.polynomial import Polynomial
import matplotlib.pyplot as plt
```

0.1 1(c)

[2]:
```python
# define the vander generator function
# note that to make the notation consistent
# np.flip is used
vander_gen = lambda n: np.flip(np.vander(np.linspace(-1, 1, n+1)))
for n in [5, 10, 20, 30]:
    V = vander_gen(n)
    kappa_2 = np.linalg.cond(V, 2)
    print(f"The condition number for n = {n:2d} is: kappa_2 = {kappa_2:.3f}")
```

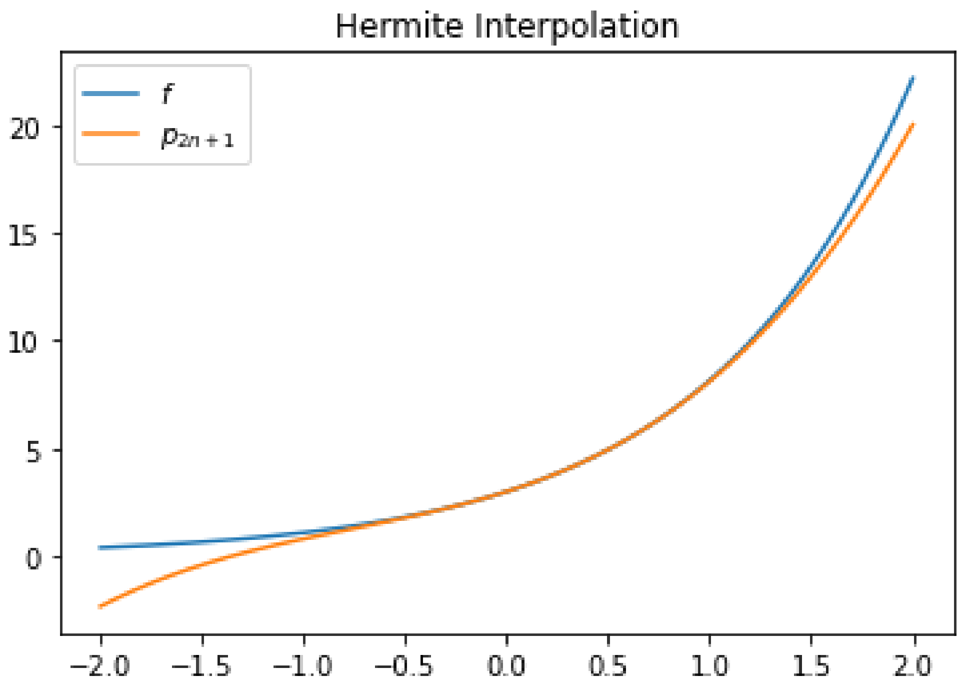
```
The condition number for n =  5 is: kappa_2 = 63.827
The condition number for n = 10 is: kappa_2 = 13951.627
The condition number for n = 20 is: kappa_2 = 831377053.878
The condition number for n = 30 is: kappa_2 = 56415165097885.938
```

- As we can see, when n grows, its condition number κ_2 grows quickly as well. It implies the basis transformation cannot be performed accurately.

0.2 2(d)

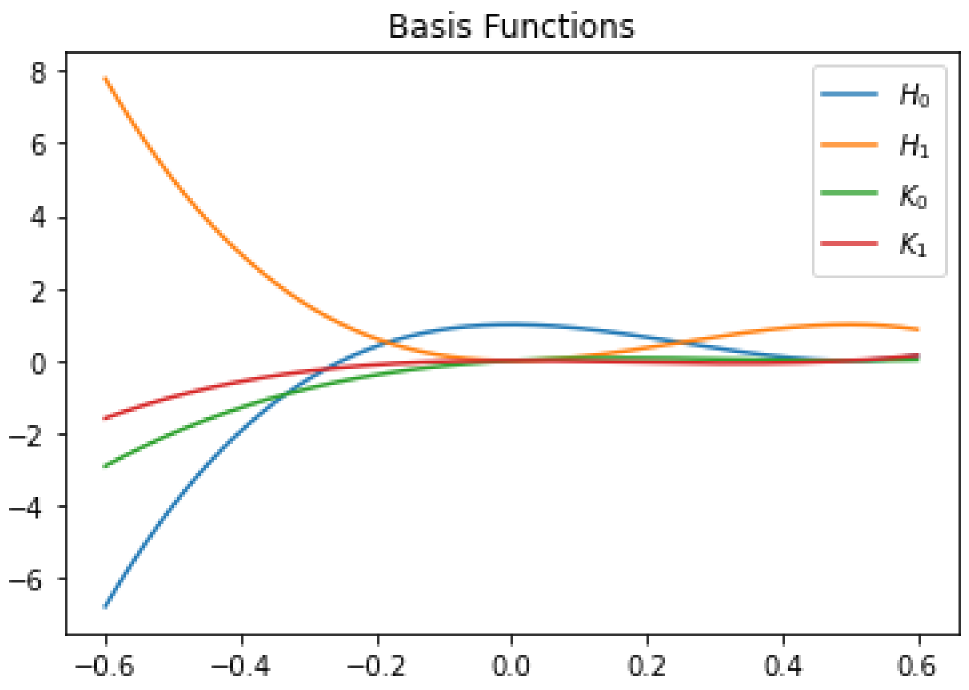
[3]:
```python
H_0 = lambda x: 4*(x - 1/2)**2 * (1 + 4*x)
H_1 = lambda x: 4*x**2 * (3 - 4*x)
K_0 = lambda x: 4*(x - 1/2)**2 * x
K_1 = lambda x: 4*x**2 * (x - 1/2)

y_0, y_1, z_0, z_1 = 3, 3 * np.exp(1/2), 3, 3 * np.exp(1/2)

f = lambda x: 3 * np.exp(x)
p = lambda x: y_0 * H_0(x) + z_0 * K_0(x) + y_1 * H_1(x) + z_1 * K_1(x)
```

[4]:
```python
x_space = np.linspace(-2, 2, 1000)
plt.plot(x_space, [f(x) for x in x_space], label=r"$f$")
plt.plot(x_space, [p(x) for x in x_space], label=r"$p_{2n+1}$")
plt.title("Hermite Interpolation")
plt.legend()
plt.show()
```



[5]:
```
x_space = np.linspace(-0.6, 0.6, 1000)
plt.plot(x_space, [H_0(x) for x in x_space], label=r"$H_0$")
plt.plot(x_space, [H_1(x) for x in x_space], label=r"$H_1$")
plt.plot(x_space, [K_0(x) for x in x_space], label=r"$K_0$")
plt.plot(x_space, [K_1(x) for x in x_space], label=r"$K_1$")
plt.title("Basis Functions")
plt.legend()
```

[5]:
```
<matplotlib.legend.Legend at 0x7fa1c6a92ac0>
```



0.3 4

[6]:
```
f = lambda x: 1 if x >= 0 else 0

def p(n, f):
    x_chebyshev = np.array([np.cos(((i + 1/2)*np.pi) / (n+1)) for i in␣
 ↪range(n+1)])
    y = np.array([f(x_i) for x_i in x_chebyshev])
    poly_coef = lagrange(x_chebyshev, y).coef[::-1]

    return Polynomial(poly_coef)

def l_infity(f, g, n):
    x_space = np.linspace(-1, 1, 10*n+1)
    candidates = [abs(f(x) - g(x)) for x in x_space]
    return max(candidates)

def l_2(f, g, n):
    x_space = np.linspace(-1, 1, 10*n)
    summation = sum([(f(x) - g(x))**2 for x in x_space])

    return ((2 * summation) / (10 * n)) ** (1/2)
```

[7]:
```python
err_2_history = []
err_m_history = []
n_space = [2, 4, 8, 16, 32, 64, 128, 256]

for n in n_space:
    p_n = p(n, f)
    err_2 = l_2(f, p_n, n)
    err_m = l_infity(f, p_n, n)
    err_2_history.append(err_2)
    err_m_history.append(err_m)
```

[8]:
```python
print("="*10 + " MAXIMAL ERROR " + "="*10)
for i, n in enumerate(n_space):
    print(f"n = {n:3d}: {err_m_history[i]}")

print()

print("="*12 + " L_2 ERROR "+"="*12)
for i, n in enumerate(n_space):
    print(f"n = {n:3d}: {err_2_history[i]}")
```

```
========== MAXIMAL ERROR ==========
n =   2: 0.9355983064143708
n =   4: 0.9425095430408871
n =   8: 0.9470005133845324
n =  16: 0.949636543244897
n =  32: 0.9510760906425554
n =  64: 524649557898087.44
n = 128: 1.2817169852842763e+47
n = 256: 5.814192150589475e+111

============ L_2 ERROR ============
n =   2: 0.602575825362048
n =   4: 0.4749343446137968
n =   8: 0.3582547960583235
n =  16: 0.2626019817271545
n =  32: 0.18945196425742922
n =  64: 61153226621720.484
n = 128: 1.0720035561706522e+46
n = 256: 3.464613188694958e+110
```

- The Maximal error doesn't converge in any senses.
- The l_2 error seems to converge, but the error blows up when $n \geq 32$. From my point of view, thoeretically, the l_2 loss should converge. The blowing-up is due to some numerical error of python.

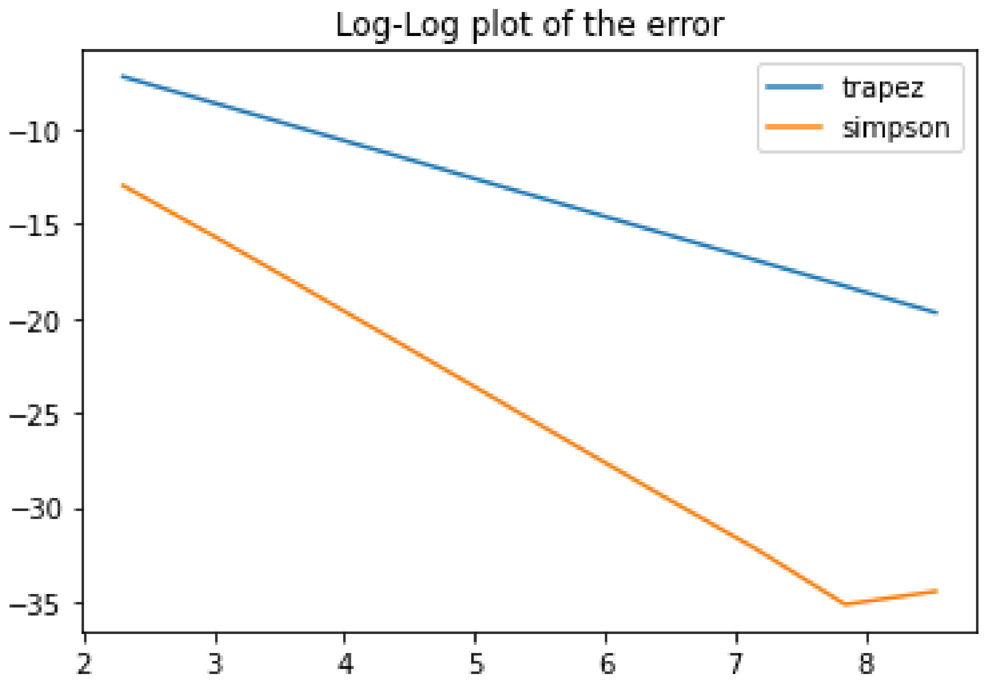
0.4 5

[9]:
```
def trapez(f, a, b, m):
    x_space = np.linspace(a, b, m+1)
    temp = 0

    for x in x_space[1:-1]:
        temp += f(x)
    temp += (f(x_space[0]) + f(x_space[-1]))/2

    return (temp * (b-a)) / m

def simpson(f, a, b, m):
    x_space = np.linspace(a, b, m+1)

    def sub_simpson(f, a, b):
        c = (a + b) / 2
        return ((b-a)/6) * (f(a) + 4*f(c) + f(b))

    summation = 0
    for i in range(m):
        a, b = x_space[i], x_space[i+1]
        summation += sub_simpson(f, a, b)

    return summation
```

[10]:
```
f = lambda x: x ** .5
a, b= 0.1, 1
I_gt = 2/3 - 1/(15*10**.5)
err_simpson = []
err_trap = []

m_space = [10*2**i for i in range(10)]
for m in m_space:
    err_trap.append(abs(I_gt - trapez(f, a, b, m)))
    err_simpson.append(abs(I_gt - simpson(f, a, b, m)))

log_e_trap = np.log(err_trap)
log_e_simpson = np.log(err_simpson)
log_m = np.log(m_space)
```

[11]:
```
# Visualization
plt.plot(log_m, log_e_trap, label="trapez")
plt.plot(log_m, log_e_simpson, label="simpson")
plt.title("Log-Log plot of the error")
plt.legend()
```

[11]:
```
<matplotlib.legend.Legend at 0x7fa1c6bcc0a0>
```



want to find, such that D, κ that minimizes

$$\log(\epsilon) = D + \kappa \log(m)$$

It is equivalent to do least square estimation or linear regression.

[12]:
```
# Optimization
n = len(m_space)
A = np.ones((n, 2))
A[:, 1] = log_m
D_1, kappa_1 = np.linalg.solve(A.T @ A, A.T @ log_e_trap)
D_2, kappa_2 = np.linalg.solve(A.T @ A, A.T @ log_e_simpson)

print("Composite Trapezoidal Rule")
print(f"D: {D_1:.4f}, kappa: {kappa_1:.4f}")
print("Composite Simpson Rule")
print(f"D: {D_2:.4f}, kappa: {kappa_2:.4f}")
```

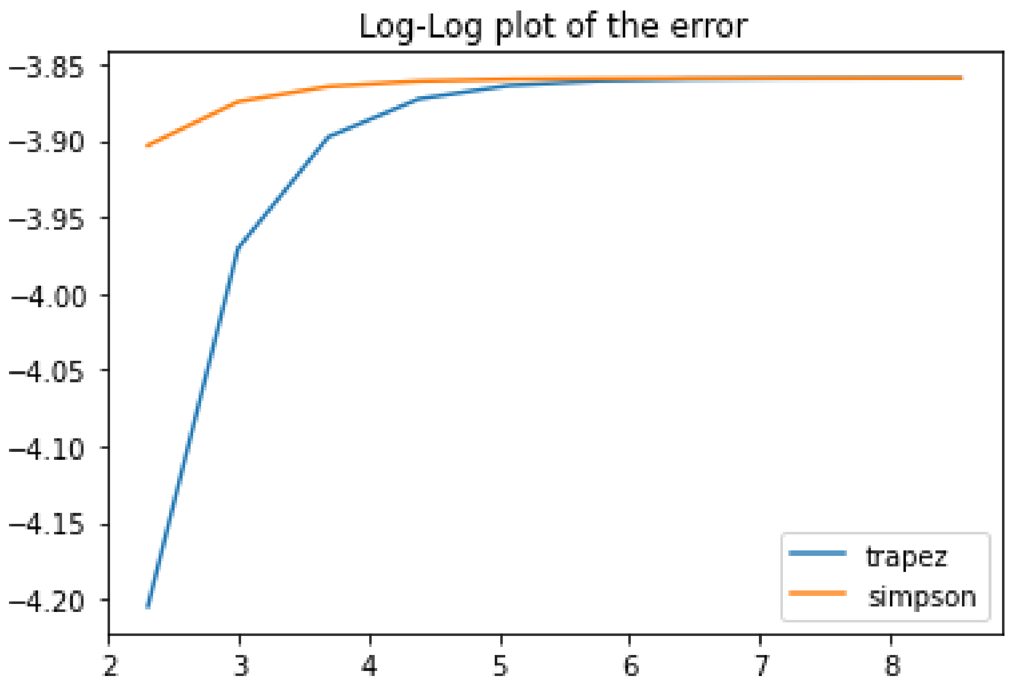
```
Composite Trapezoidal Rule
D: -2.6264, kappa: -1.9987
Composite Simpson Rule
D: -4.8010, kappa: -3.7244
```

0.4.1 Repeat that again for $a = 1$

[13]:
```
f = lambda x: x ** .5
a, b= 0, 1
I_gt = 2/3 - 1/(15*10**.5)
err_simpson = []
err_trap = []

m_space = [10*2**i for i in range(10)]
for m in m_space:
    err_trap.append(abs(I_gt - trapez(f, a, b, m)))
    err_simpson.append(abs(I_gt - simpson(f, a, b, m)))

log_e_trap = np.log(err_trap)
log_e_simpson = np.log(err_simpson)
log_m = np.log(m_space)
```

[14]:
```
# Visualization
plt.plot(log_m, log_e_trap, label="trapez")
plt.plot(log_m, log_e_simpson, label="simpson")
plt.title("Log-Log plot of the error")
plt.legend()
```

[14]:
```
<matplotlib.legend.Legend at 0x7fa1c6dac5e0>
```



```
[15]: # Optimization
n = len(m_space)
A = np.ones((n, 2))
A[:, 1] = log_m
D_1, kappa_1 = np.linalg.solve(A.T @ A, A.T @ log_e_trap)
D_2, kappa_2 = np.linalg.solve(A.T @ A, A.T @ log_e_simpson)

print("Composite Trapezoidal Rule")
print(f"D: {D_1:.4f}, kappa: {kappa_1:.4f}")
print("Composite Simpson Rule")
print(f"D: {D_2:.4f}, kappa: {kappa_2:.4f}")
```

```
Composite Trapezoidal Rule
D: -4.1063, kappa: 0.0360
Composite Simpson Rule
D: -3.8915, kappa: 0.0047
```

0.4.2 Theoretical estimates

Once we put the lowerbound of the integral to be 0, we encounter a problem. That is,

$$f'(x) = \frac{1}{2}x^{-\frac{1}{2}}$$

The slope blows up when x goes to 0. Therefore, the $\log(\epsilon)$ and $\log(m)$ is no longer in good linear relation, which fails the theoretical estimates