Optional Assignment 2 (Yan Chen Zhou 100496757)

A general expression for the local truncation error for any r-step linear multistep method is:

$$\begin{aligned}\tau(t_{n+r}) &= \frac{1}{k}\left[\sum_{j=0}^{r}\alpha_j u(t_{n+j}) + k\sum_{j=0}^{r}\beta_j u'(t_{n+j})\right] \\ &= \frac{1}{k}\left[\sum_{j=0}^{r}\alpha_j\right] + \frac{1}{k}\left[\sum_{j=0}^{r}(j\alpha_j - \beta_j)\right] + \cdots + k^{q-1}\left[\sum_{j=0}^{r}\left(\frac{1}{q!}j^q\alpha_j - \frac{1}{(q-1)!}j^{q-1}\beta_j\right)\right]u^{(q)}(t_n) + \cdot\end{aligned}$$

To achieve the maximum efficiency possible, we want to banish the coefficients up to order r. We can translate this problem into this system of linear equations

$$\begin{bmatrix} 1 & 0 & 1 & 0 & \cdots & 1 & 0 \\ 0 & -1 & 1 & -1 & \cdots & r & -1 \\ 0 & 0 & \frac{1}{2} & -1 & \cdots & \frac{1}{2}r^2 & -r \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \frac{1}{r!} & -\frac{1}{(r-1)!} & \cdots & \frac{1}{r!}r^r & -\frac{1}{(r-1)!}r^{r-1} \end{bmatrix} \begin{bmatrix} \alpha_0 \\ \beta_0 \\ \alpha_1 \\ \beta_1 \\ \vdots \\ \alpha_r \\ \beta_r \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}$$

which has infinite solutions at the moment.

Adams-Bashforth

For Adams-Bashforth method, we choose $\alpha_r = 1, \alpha_{r-1} = -1, \alpha_j = 0$ for $j < r-1$ and $\beta_r = 0$, then we define a system of linear equations such that

$$\begin{bmatrix} 0 & 0 & \cdots & 0 & 0 \\ -1 & -1 & \cdots & -1 & -1 \\ 0 & -1 & \cdots & -(r-2) & -(r-1) \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & -\frac{1}{(r-1)!} & \cdots & -\frac{1}{(r-1)!}(r-2)^{r-1} & -\frac{1}{(r-1)!}(r-1)^{r-1} \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \vdots \\ \beta_{r-1} \end{bmatrix} = \begin{bmatrix} 0 \\ -1 \\ \frac{1}{2}(r-1)^2 - \frac{1}{2}r^2 \\ \vdots \\ \frac{1}{r!}(r-1)^r - \frac{1}{r!}r^r \end{bmatrix}$$

Now we can obtain the coefficients in Python by solving this system of linear equations. First we setting up the coefficients

```
# Import Libraries
import numpy as np
from scipy.special import factorial

def c_alpha(q: int, j: int) -> float:
    return j**q/factorial(q) if q!=0 else 1

def c_beta(q: int, j: int) -> float:
    return -(j)**(q-1)/factorial(q-1) if q!=0 else 0
```

Now we can solve the SLE and obtain the coefficients of the r-step as follows

```
def ab_eq(order: int, q: int) -> list[float]:
    if order < 1:
        raise ValueError("r must be positive")
```

```
    Ai = [0] * order
    for j in range(order):
        Ai[j] = c_beta(q, j)

    bi = c_alpha(q, order-1) - c_alpha(q, order)
    return Ai, bi

def adams_bashforth(order: int):
    if order < 1:
        raise ValueError("r must be positive")

    A, b = [], []
    for i in range(1, order+1):
        Ai, bi = ab_eq(order, i)
        A.append(Ai)
        b.append(bi)

    A = np.array(A)
    b = np.array(b)
    x = np.linalg.solve(A, b)

    return x
```

The above code returns

$$[\beta_0, \beta_1, \cdots, \beta_r]$$

The coefficients of Adams-Bashforth up to order 4 are:

```
print(f'ADAMS-BASHFORTH COEFFICIENTS')
for i in range(1, 5):
    print(f'r={i}: {adams_bashforth(i)}')
```

```
ADAMS-BASHFORTH COEFFICIENTS
r=1: [1.]
r=2: [-0.5  1.5]
r=3: [ 0.41666667 -1.33333333  1.91666667]
r=4: [-0.375       1.54166667 -2.45833333  2.29166667]
```

We can verify that the coefficients are indeed correct.

Adams Moulton

For Adams-Moulton method, we choose $\alpha_r = 1, \alpha_{r-1} = -1$ and $\alpha_j = 0$ for $j < r - 1$. We need to integrate one more coefficient into the SLE to form

$$\begin{bmatrix} 0 & 0 & \cdots & 0 & 0 \\ -1 & -1 & \cdots & -1 & -1 \\ 0 & -1 & \cdots & -(r-1) & -r \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & -\frac{1}{(r-1)!} & \cdots & -\frac{1}{(r-1)!}(r-1)^{r-1} & -\frac{1}{(r-1)!}r^{r-1} \\ 0 & -\frac{1}{r!} & \cdots & -\frac{1}{r!}(r-1)^r & -\frac{1}{r!}r^r \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_{r-1} \\ \beta_r \end{bmatrix} = \begin{bmatrix} 0 \\ -1 \\ \vdots \\ \frac{1}{r!}(r-1)^r - \frac{1}{r!}r^r \\ \frac{1}{(r+1)!}(r-1)^{r+1} - \frac{1}{(r+1)!}r^{r+1} \end{bmatrix}$$

for each r-step. Now we solve it with

```
def am_eq(order: int, q: int) -> list[float]:
    if order < 1:
        raise ValueError("r must be positive")
```

```
    Ai = [0] * (order+1)
    for j in range(order+1):
        Ai[j] = c_beta(q, j)

    bi = c_alpha(q, order-1) - c_alpha(q, order)
    return Ai, bi

def adams_moulton(order: int):
    if order < 1:
        raise ValueError("r must be positive")

    A, b = [], []
    for i in range(1, order+2):
        Ai, bi = am_eq(order, i)
        A.append(Ai)
        b.append(bi)

    A = np.array(A)
    b = np.array(b)
    x = np.linalg.solve(A, b)

    return x
```

That returns

$$[\beta_0, \beta_1, \cdots, \beta_r]$$

The coefficients of Adams-Moulton up to order 4 are:

```
print(f'ADAMS-MOULTON COEFFICIENTS')
for i in range(1, 5):
    print(f'r={i}: {adams_moulton(i)}')
```

```
ADAMS-MOULTON COEFFICIENTS
r=1: [0.5 0.5]
r=2: [-0.08333333  0.66666667  0.41666667]
r=3: [ 0.04166667 -0.20833333  0.79166667  0.375     ]
r=4: [-0.02638889  0.14722222 -0.36666667  0.89722222  0.34861111]
```

Backward Differentation

For Backward Differentation method, we choose $\beta_j = 0$ for $j < r$ so that

$$\sum_{j=0}^{r} \alpha_j U_{n+j} = k\beta_r f(U_{n+r}, t_{n+r})$$

we can set $\beta_r = 1$ to solve

$$\begin{bmatrix} 1 & 1 & \cdots & 1 & 1 \\ 0 & 1 & \cdots & r-1 & r \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & -\frac{1}{(r-1)!} & \cdots & -\frac{1}{(r-1)!}(r-1)^{r-1} & -\frac{1}{(r-1)!}r^{r-1} \\ 0 & -\frac{1}{r!} & \cdots & -\frac{1}{r!}(r-1)^r & -\frac{1}{r!}r^r \end{bmatrix} \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ \vdots \\ \alpha_{r-1} \\ \alpha_r \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ \frac{1}{(r-2)!}r^{r-2} \\ \frac{1}{(r-1)!}r^{r-1} \end{bmatrix}$$

We use the code below:

```
def bd_eq(order: int, q: int) -> list[float]:
    if order < 1:
        raise ValueError("Order must be positive")

    Ai = [0] * (order+1)
```

```python
    for j in range(order+1):
        Ai[j] = c_alpha(q, j)

    return Ai, -c_beta(q, order)

def backward_differentation(order: int):
    if order < 1:
        raise ValueError("Order must be positive")

    A, b = [], []
    for i in range(order+1):
        Ai, bi = bd_eq(order, i)
        A.append(Ai)
        b.append(bi)

    A = np.array(A)
    b = np.array(b)
    x = np.linalg.solve(A, b)

    return x
```

The output is

$$[\alpha_0, \ldots, \alpha_r]$$

The coefficients of Backward Differentiation up to order 4 (with $\beta_r = 1$) are:

```python
print(f'BACKWARD DIFFERENTIATION COEFFICIENTS')
for i in range(1, 5):
    print(f'r={i}: {backward_differentation(i)}')
```

```
BACKWARD DIFFERENTIATION COEFFICIENTS
r=1: [-1.  1.]
r=2: [ 0.5 -2.   1.5]
r=3: [-0.33333333  1.5        -3.          1.83333333]
r=4: [ 0.25       -1.33333333  3.         -4.          2.08333333]
```