National Central University

Department of Mathematics

Numerical Method for Differential Equations

Numerical Method for Differential Equations

Author: 111201528 王元亨

June 17, 2024

Contents

1 Introduction

The document is an in-depth exploration of various numerical techniques, covering topics from polynomial interpolation to numerical differentiation and integration, as well as specific methods for solving ordinary differential equations (ODEs). Describe in detail the derivation process, results, and errors of each algorithm, and use MATLAB to conduct actual numerical experiments to explore the problems, advantages and disadvantages of the algorithms.

1. **Polynomial interpolation:** Polynomial interpolation discusses different forms such as the standard, Newton, and Lagrange forms. It delves into the time complexity, error estimation, and convergence properties of these methods, providing examples and exploring other interpolation techniques.

2. **Numerical Differentiation:** This part explains methods for numerical finite differentiation, including the centered differencing formula, interpolation approach, and Richardson extrapolation.

3. **Numerical Integration:** Introduce that numerical integration methods such as the trapezoidal rule and Simpson's rule.

4. **Numerical Differential Equations:** This section discusses several numerical methods for solving initial-value problem(IVP), and explains stability issues in solving stiff differential equations. The last part introduce boundary-value problems(BVP) and numerical methods for solving linear boundary-value problems

2 Polynomial Interpolation

Polynomial interpolation is the interpolation of a given bivariate data set by the polynomial of lowest possible degree that passes through the points of the dataset.

Given a set of $n+1$ data points (x_0, y_0), (x_1, y_1), ..., (x_n, y_n), with no two x_j the same. A polynomial function

$$p(x) = c_0 + c_1 x + c_2 x^2 + \cdots + c_n x^n$$

is said to interpolate the data if $p(x_j) = y_j, \forall j \in \{0, 1, \cdots, n\}$.

2.1 Standard form

The standard form of the interpolating polynomials is

$$p(x) = c_0 + c_1 x + c_2 x^2 + \cdots + c_n x^n$$

And we have $p(x_i) = y_i$, for $0 \leq i \leq n$, lead to a system of $n+1$ linear equations for determining $c_0, c_1, \cdots, c_n$:

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ 1 & x_2 & x_2^2 & \cdots & x_2^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

An interpolant $p(x)$ corresponds to a solution $\mathbf{c} = \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{bmatrix}$ of the above system $\mathbf{Xc} = \mathbf{Y}$

The matrix $\mathbf{X}$ on the left is a *Vandermonde matrix*, whose determinant is known to be $det(\mathbf{X}) = \prod_{0 \le i < j \le n}(x_j - x_i)$, which is non-zero since the nodes x_j are all distinct. This ensures that the matrix is invertible and the equation has the unique solution $\mathbf{c} = \mathbf{X^{-1}Y}$; that is, $p(x)$ exists and is unique[2].

However, as n increases, the condition number of the Vandermonde matrix will become large, and this linear system will become difficult to solve. Therefore, this approach is not recommended.

2.2 Newton form

The Newton form of the interpolating polynomials is

$$p(x) = \sum_{j=0}^{n} c_j \left(\prod_{i=0}^{j} (x - x_i) \right)$$

with coefficients $c_j \in \mathbb{R}$, $\forall j = 0, 1, \cdots, n$.

Thus, we have the iteration relation

$$p_k(x) = p_{k-1}(x) + c_k(x - x_0)(x - x_1) \cdots (x - x_{k-1})$$

and to solve for the coefficients:

$$c_k = \frac{y_k - p_{k-1}(x_k)}{(x_k - x_0)(x_k - x_1) \cdots (x_k - x_{k-1})}$$

Example: Given the data point $x = [1/3; 1/4; 1]$ and $y = [2; -1; 7]$, constructing the Newton form of interpolating polynomial of degree 2.

$$\begin{aligned} p_0(x) &= c_0 \\ p_1(x) &= c_0 + c_1(x - x_0) \\ p_2(x) &= c_0 + c_1(x - x_0) + c_2(x - x_0)(x - x_1) \end{aligned}$$

Calculating the coefficients:

$$\begin{aligned} &p_0(\frac{1}{3}) = 2 \\ &p_1(\frac{1}{4}) = 2 + c_1\left(\frac{1}{4} - \frac{1}{3}\right) = -1 \implies c_1 = 36 \\ &p_2(1) = 2 + 36(1 - \frac{1}{3}) + c_2(1 - \frac{1}{3})(1 - \frac{1}{4}) = 7 \implies c_2 = -38 \end{aligned}$$

We then obtain the Newton form of interpolating polynomial

$$\begin{aligned} p(x) &= 2 + 36(x - \frac{1}{3}) - 38(x - \frac{1}{3})(x - \frac{1}{4}) \\ &= 26x^2 + \frac{19}{6}x + \frac{49}{6} \end{aligned}$$

2.3 Lagrange form

The Lagrange form of the interpolating polynomials is

$$p(x) = \sum_{j=0}^{n} y_j l_j(x)$$

where

$$l_j(x) = \prod_{i \neq j}^{n} \frac{x - x_i}{x_j - x_i}$$

is the Lagrange basis polynomial.

Example: Given the data points $x = [-1; 0; 1]$ and $y = [-3; 1; 9]$, constructing the Lagrange form of interpolating polynomial of degree 2.

Let calculate the Lagrange basis polynomial:

$$\begin{aligned} l_0(x) &= \frac{x-0}{-1-0} \times \frac{x-1}{-1-1} = \frac{1}{2}x^2 - \frac{1}{2}x \\ l_1(x) &= \frac{x+1}{0+1} \times \frac{x-1}{0-1} = -x^2 + 1 \\ l_2(x) &= \frac{x+1}{1+1} \times \frac{x-0}{1-0} = \frac{1}{2}x^2 + \frac{1}{2}x \end{aligned}$$

Substitute these into the polynomial:

$$\begin{aligned} p_2(x) &= y_0 l_0(x) + y_1 l_1(x) + y_2 l_2(x) \\ &= -3\left(\frac{1}{2}x^2 - \frac{1}{2}x\right) + -x^2 + 1 + 9\left(\frac{1}{2}x^2 + \frac{1}{2}x\right) \\ &= 2x^2 + 6x + 1 \end{aligned}$$

2.4 Time complexity

Methods	Standard form	Newton form	Lagrange form
Time complexity	$\mathbf{O}(n^3)$	$\mathbf{O}(n^2)$	$\mathbf{O}(n^2)$

Table 1: Time complexity of polynomial interpolation

2.5 Error estimation

Theorem: Lagrange remainder formula[2][1]: Let f be a function in $C^{n+1}[a, b]$, and let p be the polynomial of degree at most n that interpolates the function f at $n+1$ distinct points x_0, x_1,..., x_n in the interval $[a, b]$. To each $x \in [a, b]$ there corresponds a point $\xi_x \in (a, b)$ such that

$$f(x) - p_n(x) = \frac{f^{(n+1)}(\xi_x)}{(n+1)!} \prod_{i=0}^{n} (x - x_i)$$

2.6 Convergence properties

From Lagrange remainder formula, the error bound[9] of polynomial interpolation is

$$\max_{x\in[a,b]} \|f(x) - p_n(x)\| \leq \frac{M_{n+1}}{(n+1)!} \max_{x\in[a,b]} \|\omega(x)\|$$

where

$$M_{n+1} = \max_{\xi\in[a,b]} \|f^{(n+1)}(\xi)\|, \omega(x) = \prod_{j=1}^{n}(x - x_j)$$

We can observe that

$$M_{n+1} \max_{x\in[a,b]} \|\omega(x)\| = \mathbf{O}((n+1)!)$$

which means

$$\frac{M_{n+1}}{(n+1)!} \max_{x\in[a,b]} \|\omega(x)\| \text{ does not converge to 0 as } n \to \infty$$

Thus, the interpolation does not converge to f on $[a, b]$.

2.7 Example

1. Using 11 equally spaced nodes on the interval $[-5, 5]$, find the standard form interpolating polynomial p_{10} of degree 10 the function $f(x) = \frac{1}{x^2+1}$. Compare f and p_{10} and label the data points.

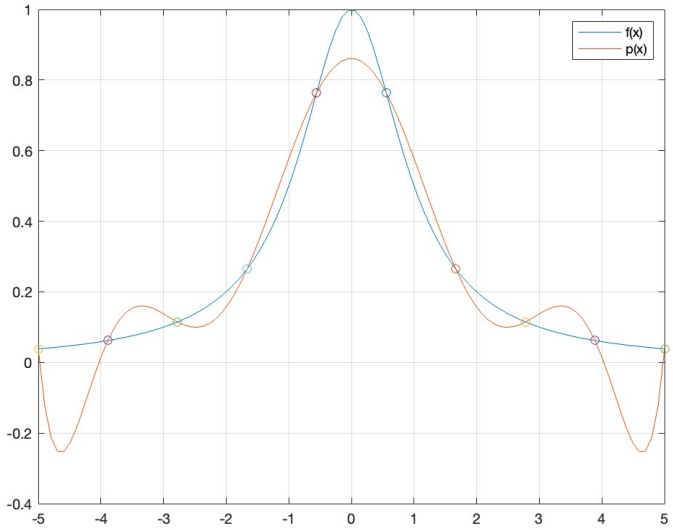


Figure 1: the standard form of f

2. Given the data points $x = [-1; 0; 1]$ and $y = [y_0; y_1; y_2]$ is determined by $f(x) = x^3 + 2x^2 + 5x + 1$. Evaluate the values of $p_2(x)$ by Lagrange interpolation at some points other than the grid points.

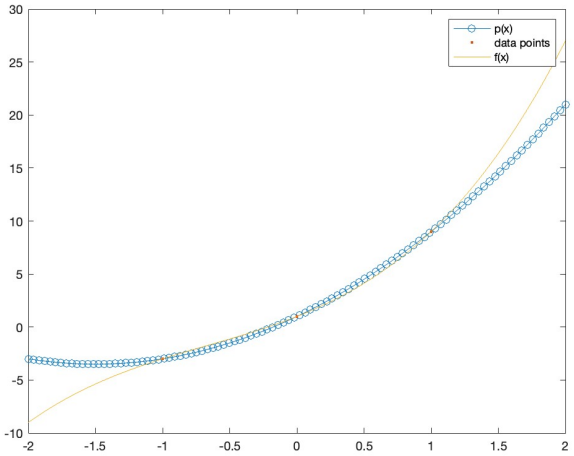


Figure 2: Lagrange interpolation of f

3. Compare different numbers of equally spaced nodes to construct the Lagrange form of interpolating polynomials for $g(x) = e^{-x}$ on $[-5, 5]$.

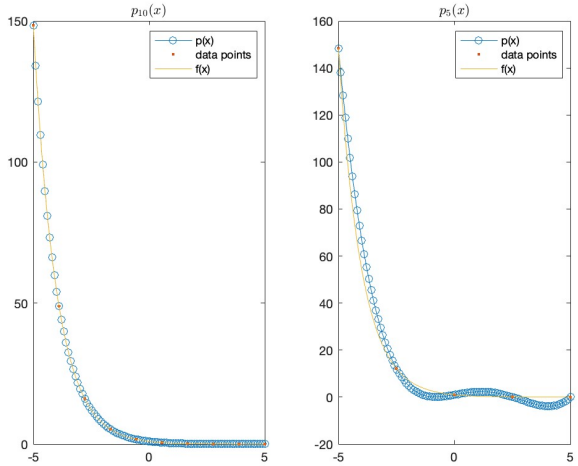


Figure 3: compare 10 and 5 of equally spaced nodes

	error of $p_{10}(x)$	error of $p_5(x)$
$\|\cdot\|_2$	0.1945	48.0830
$\|\cdot\|_\infty$	0.0810	13.2247

Table 2: compare error of given different numbers of data points

3 Numerical Differetiation

3.1 The centered differencing formula

Given a continuous finction $f : \mathbb{R} \to \mathbb{R}$. By Taylor's series, we get the first derivative of f simply.

$$f'(x) = \frac{f(x+h) - f(x)}{h} + \frac{h}{2}f''(x) = \frac{f(x+h) - f(x)}{h} + \mathbf{O}(h)$$

$$f(x+h) \approx f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{3!}f'''(x) \tag{1}$$

$$f(x-h) \approx f(x) - hf'(x) + \frac{h^2}{2}f''(x) - \frac{h^3}{3!}f'''(x) \tag{2}$$

Adding (2) to (1), we obtain

$$f'(x) = \frac{1}{2h}(f(x+h) - f(x-h)) + \mathbf{O}(h^2)[3]$$

And we do the same thing to get second derivative of f

$$f''(x) = \frac{1}{h^2}(f(x+h) - 2f(x) + f(x-h)) + \mathbf{O}(h^2)[3]$$

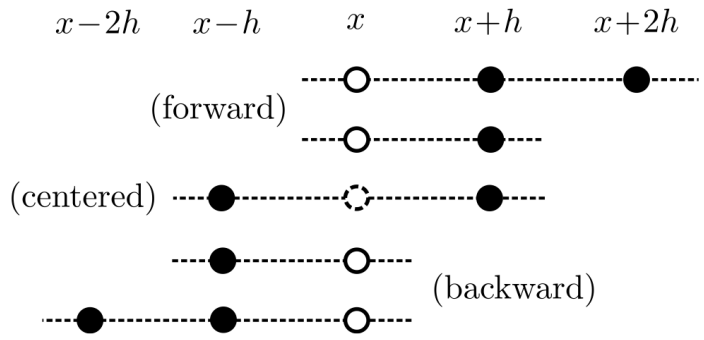


Figure 4: forward / backward finite difference formula[12]

3.2 Richardson extrapolation

Richardson extrapolation is a general procedure to improve accuracy. For Taylor series, we have

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{3!}f'''(x) + \mathbf{O}(h^4)$$

$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2}f''(x) - \frac{h^3}{3!}f'''(x) + \mathbf{O}(h^4)$$

Subtract and rearrange to obtain

$$f'(x) = \frac{1}{2h}(f(x+h) - f(x-h)) - \left[\frac{h^2}{3!}f^{(3)}(x) + \frac{h^4}{5!}f^{(5)}(x) + \frac{h^6}{7!}f^{(7)}(x) + \cdots\right]$$

We rewrite the formula

$$L = \phi(h) + [a_2 h^2 + a_4 h^4 + a_6 h^6 + \cdots] \tag{3}$$

If $a_2 \neq 0$, the truncation error is $\mathbf{O}(h^2)$. We then rewrite the abstract form for $\frac{h}{2}$

$$L = \phi(\frac{h}{2}) + \left[a_2(\frac{h}{2})^2 + a_4(\frac{h}{2})^4 + a_6(\frac{h}{2})^6 + \cdots\right] \tag{4}$$

Multiply (4) by 4 and subtracting from (3) to obtain

$$L = \frac{4}{3}\phi(\frac{h}{2}) - \frac{1}{3}\phi(h) + \left[\frac{a_4}{4}h^4 + \frac{5a_6}{16}h^6 + \cdots\right] \tag{5}$$

This formula is the first step in Richardson extrapolation. It shows that a simple combination of $\phi(h)$ and $\phi(\frac{h}{2})$ furnishes an estimate of L with accuracy $\mathbf{O}(h^4)$.

Using induction, L can be approximated by

$$L = \phi(\frac{h}{2^m}) + \mathbf{O}(h^{2m}) := N_m(h) + \mathbf{O}(h^{2m})$$

where

$$N_m(h) = N_{m-1}(h) + \frac{1}{4^m}\left(N_{m-1}(\frac{h}{2}) - N_{m-1}(h)\right)$$

with

$$N_0(h) = \frac{1}{2h}(f(x+h) - f(x-h))$$

$\mathbf{O}(h^2)$	$\mathbf{O}(h^4)$	$\mathbf{O}(h^6)$	$\mathbf{O}(h^8)$	$\cdots$	$\mathbf{O}(h^{2m})$
$N_0(h)$					
$N_0\left(\frac{h}{2}\right)$	$N_1(h)$				
$N_0\left(\frac{h}{4}\right)$	$N_1\left(\frac{h}{2}\right)$	$N_2(h)$			
$N_0\left(\frac{h}{8}\right)$	$N_1\left(\frac{h}{4}\right)$	$N_2\left(\frac{h}{2}\right)$	$N_3(h)$		
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	
$N_0\left(\frac{h}{2^m}\right)$	$N_1\left(\frac{h}{2^{m-1}}\right)$	$N_2\left(\frac{h}{2^{m-2}}\right)$	$N_3\left(\frac{h}{2^{m-3}}\right)$	$\cdots$	$N_m(h)$

Table 3: Table of Richardson extrapolation[10]

4 Numerical Integration

Let f be a continuous function over $[a, b]$, having a second derivative f'' over $[a, b]$. We have known that the definite integration of f on $[a, b]$ is

$$\int_a^b f(x)dx = F(b) - F(a)$$

where F is the antiderivative of f. Otherwise, we can find an approximation of f, say g whose integral is easy to compute, then

$$\int_a^b f(x)dx \approx \int_a^b g(x)dx$$

So, we choose polynomial interpolation to approximate f.

Let g be the Lagrange form interpolation of f. We select some interpolation points x_0, x_1, ..., x_n in $[a, b]$, and define

$$g(x) = p_n(x) = \sum_{j=0}^{n} f(x_j)l_j(x)$$

where

$$l_j(x) = \prod_{i \neq j}^{n} \frac{x - x_i}{x_j - x_i}$$

Then, we have

$$\int_a^b f(x)dx \approx \int_a^b p_n(x)dx = \sum_{j=0}^{n} f(x_j) \int_a^b l_j(x)dx$$

Let us denote $A_i \approx \int_a^b l_j(x)dx$, which is independent of f. Then we have a numerical integration formula:

$$\int_a^b f(x)dx \approx \sum_{j=0}^{n} A_j f(x_j)$$

If the interpolation points are equally spaced, then this is called a **Newton-Cotes formula**.

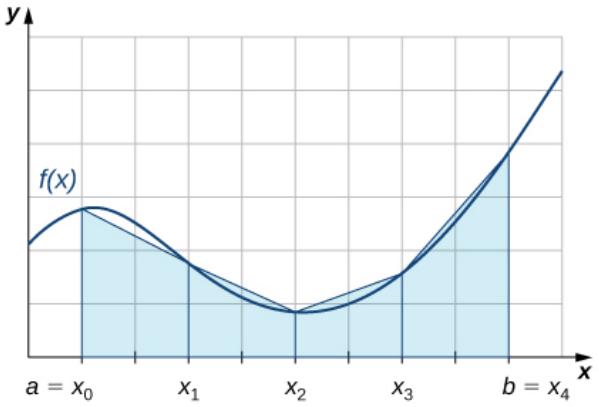
4.1 Trapezoidal rule

Trapezoidal rule is a kind of Newton-Cotes formula for $n = 1$ and the interpolation points are $x_0 = a$,and $x_1 = b$. Then, we obtain

$$A_0 = \frac{1}{2}(b - a), A_1 = \frac{1}{2}(b - a)$$

Thus, the trapezoidal rule is

$$\int_a^b f(x)dx \approx \frac{1}{2}(b - a)(f(a) + f(b))$$



Composite trapezoidal rule: Given a uniform partition $a = x_0 < x_1 < \cdots < x_n = b$, and let $h = \frac{b-a}{n}$, the composite trapezoidal rule for specific definite intergal

$$\begin{aligned}\int_a^b f(x)dx &= \sum_{i=0}^{n-1} \int_{x_i}^{x_{i+1}} f(x)dx \\ &\approx \sum_{i=0}^{n-1} \frac{1}{2}(x_{i+1} - x_i)(f(x_i) + f(x_{i+1})) \\ &= \frac{h}{2}\left(f(a) + 2\sum_{i=1}^{n-1} f(x_i) + f(b)\right) \text{ [3]}\end{aligned}$$

Algorithm 1 Composite Trapezoidal Integration

procedure COMPOSITE TRAPEZOIDAL INTEGRATION(f, a, b, err)
  $I \leftarrow \frac{(b-a)\cdot(f(a)+f(b))}{2}$ ▷ 初始積分值
  $I_{\text{old}} \leftarrow 0$
  $k \leftarrow 1$ ▷ 迭代次數
  while $|I - I_{\text{old}}| \geq err$ **do**
    $I_{\text{old}} \leftarrow I$
    $k \leftarrow k + 1$
    $n \leftarrow 2^k$ ▷ 區間分割大小
    $x \leftarrow x_1, x_2, \cdots, x_{n+1}$ ▷ 將區間分爲$n+1$ 等分
    $I \leftarrow \frac{b-a}{n}\left(\sum_{i=1}^{n} f(x_i) - \frac{f(a)}{2} - \frac{f(b)}{2}\right)$ ▷ composite trapezoidal rule
  end while
  return I
end procedure

Error bound[5]: The upper bounds for the error in using composite trapezoidal rule to estimate the definite integration of f on $[a, b]$ is

$$\mathbf{T}_n \leq \frac{M(b-a)^3}{12n^2} = \frac{Mh^2(b-a)}{12} = \mathbf{O}(h^2)$$

where $M = \max_{x\in[a,b]} |f''(x)|$.

4.2 Simpson's rule

Simpson's rule is also a kind of Newton-Cotes formula for $n = 2$ and the interpolation points are $x_0 = a$, $x_1 = \frac{a+b}{2}$, and $x_2 = b$. In order to reduce the amount of calculation, we change the interval on $[0, 1]$. Then,

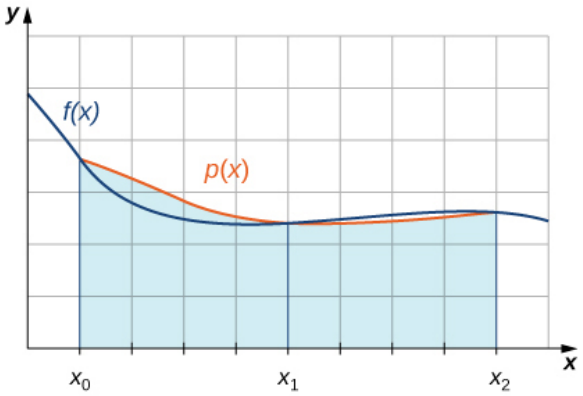
$$A_0 = \int_0^1 l_0(x)dx = \int_0^1 \frac{x-0.5}{0-0.5}\frac{x-1}{0-1}dx = \frac{1}{6}$$
$$A_1 = \int_0^1 l_1(x)dx = \int_0^1 \frac{x-0}{0.5-0}\frac{x-1}{0.5-1}dx = \frac{2}{3}$$
$$A_2 = \int_0^1 l_2(x)dx = \int_0^1 \frac{x-0}{1-0}\frac{x-0.5}{1-0.5}dx = \frac{1}{6}$$

Then, let $y = (b-a)x + a$ to change the interval on $[a, b]$, we obtain

$$A_0 = \frac{1}{6}(b-a), A_1 = \frac{2}{3}(b-a), A_2 = \frac{1}{6}(b-a)$$

Thus, the Simpson's rule is

$$\int_a^b f(x)dx \approx \frac{1}{6}(b-a)(f(a) + 4f(\frac{a+b}{2}) + f(b))$$



Composite Simpson's rule: Given a uniform partition $a = x_0 < x_1 < \cdots < x_n = b$, and let $h = \frac{b-a}{n}$, the composite Simpson's rule for specific definite intergal

$$\begin{aligned}
\int_a^b f(x)dx &= \sum_{i=1}^{n-1} \int_{x_{i-1}}^{x_{i+1}} f(x)dx \\
&\approx \sum_{i=1}^{n-1} \frac{1}{6}(x_{i+1} - x_{i-1})(f(x_{i-1}) + 4f(x_i) + f(x_{i+1})) \\
&= \frac{h}{3}\left(f(a) + 4f(x_1) + 2f(x_2) + 4f(x_3) + 2f(x_4) + \cdots + 4f(x_{n-2}) + 2f(x_{n-1}) + f(b)\right) \\
&= \frac{h}{3}\left(f(a) + 2\sum_{i=2}^{\frac{n}{2}} f(x_{2i-2}) + 4\sum_{i=1}^{\frac{n}{2}} f(x_{2i-1}) + f(b)\right) \text{ [3]}
\end{aligned}$$

Error bound[5]: The upper bounds for the error in using composite Simpson's rule to estimate the definite integration of f on $[a, b]$ is

$$\mathbf{S}_n \leq \frac{M(b-a)^5}{180n^4} = \frac{Mh^4(b-a)}{180} = \mathbf{O}(h^4)$$

where $M = \max_{x\in[a,b]} |f''(x)|$.

Algorithm 2 Composite Simpson's Rule

Input: function f, interval $[a, b]$, error tolerance err
Output: Integral approximation I
$I \leftarrow \frac{(b-a)}{6}\left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right)$ ▷ Initial integral value
$I_{\text{old}} \leftarrow 0$
$k \leftarrow 1$ ▷ Iteration count
while $|I - I_{\text{old}}| \geq err$ **do**
  $I_{\text{old}} \leftarrow I$
  $k \leftarrow k + 1$
  $n \leftarrow 4^k$
  $x \leftarrow x_1, x_2, \cdots, x_{n+1}$ ▷ Divide $[a, b]$ into $n + 1$ equally spaced points
  $I \leftarrow 0$
  for $i \leftarrow 2$ to n step 2 **do**
    $I \leftarrow I + f(x[i-1]) + 4f(x[i]) + f(x[i+1])$
  end for
  $I \leftarrow I \cdot \frac{x[2]-x[1]}{3}$
end while
return I

4.3 Example

Let $f(x) = x^2, x \in [2, 4]$, $g(x) = \sqrt{1+x^3}, x \in [-1, 1]$, and $h(x) = e^{-x^2}, x \in [0, 1]$. We count the definite integration of each function on the interval. Because Simpson's rule requires 2 times more grid points than the trapezoidal rule, we set 2 and 4 as the base of the grid points for each iteration respectively.

	n	residue $\lvert I_{old} - I\rvert$	I
trapezoidal rule	$2^{12} = 4096$	3.8146×10^{-6}	18.666667938232422
Simpson's rule	$4^2 = 16$	0	18.666666666666668

Table 4: the definite integration of f on $[2, 4]$

	n	residue $\lvert I_{old} - I\rvert$	I
trapezoidal rule	$2^{12} = 4096$	7.0397×10^{-6}	1.952753369914074
Simpson's rule	$4^7 = 16384$	1.3276×10^{-6}	1.952757044077307

Table 5: the definite integration of g on $[-1, 1]$

	n	residue $\lvert I_{old} - I \rvert$	I
trapezoidal rule	$2^8 = 256$	2.8067×10^{-6}	0.746823197246152
Simpson's rule	$4^3 = 64$	1.2414×10^{-6}	0.746824133299673

Table 6: the definite integration of h on $[0, 1]$

From this example we can see that Simpson's rule converges to a given fault tolerance faster than the trapezoidal rule.

5 Numerical Differential Equation

The general form of the first order initial value problem differential equation (IVP) is

$$\begin{cases} x'(t) = f(t,x) \\ x(t_0) = x_0 \end{cases}$$

- **Single step and multi step:** In a *single-step method*, one only needs a condition to start, however in a *multistep method* one might need the solution at several values before it can be implemented.
- **Explicit methods and implicit method:** *Explicit methods* calculate the state of a system at a later time from the state of the system at the current time, while *implicit methods* find a solution by solving an equation involving both the current state of the system and the later one.

Method Type	Single Step Methods	Multistep Methods
Explicit Methods	Euler's method Runge-Kutta methods Heun's method	Adams-Bashforth formulas
Implicit Methods	Modified Euler's method Implicit Runge-Kutta methods	Adams-Moulton formulas

Table 7: Classification of Single Step and Multistep Methods, Explicit and Implicit Methods

5.1 Single step method

5.1.1 Taylor-series method

We consider the IVP

$$\begin{cases} x'(t) = f(t,x) \\ x(t_0) = x_0 \end{cases}$$

By Taylor series,

$$\begin{aligned} x(t+h) &= x(t) + hx'(t) + \frac{h^2}{2}x''(t) + \frac{h^3}{3!}x'''(t) + \mathbf{O}(h^4) \\ &= x(t) + hf(t,x) + \frac{h^2}{2}f'(t,x) + \frac{h^3}{3!}f''(t,x) + \mathbf{O}(h^4) \end{aligned}$$

If we truncate at h^3 then the truncation for obtaining $x(t+h)$ is $O(h^4)$. We say the method is of order 3. The estimate of the local truncation error can be done by looking at

$$E_n = \frac{h^{n+1}}{(n+1)!}x^{(n+1)}(t+\theta h), \theta \in [0,1]$$

Advantages and disadvantages of Taylor-series method[4]

- **Advantages**
  1. Conceptual simplicity

2. Potential for high precision
   If we get, e.g., 20 derivatives of $x(t)$, then the method is of order 20 (i.e., terms up to and including the one involving h^{20}).

- **Disadvantages**

1. The method depends on repeated differentiation of differential equations. (unless we intend to use only the method of order 1). That is, $f(t, x)$ must have partial derivatives of sufficiently high order in the region where we are solve the problem. Such an assumption is not necessary for the existence of a solution.
2. The various derivatives formula need to be programmed.

5.1.2 Euler's method

Euler's method is the Taylor-series method for order $n = 1$[4]. The key step of Euler's method is

$$x(t+h) = x(t) + hf(t, x(t)) + \mathbf{O}(h^2)$$

or, equivalently

$$x_{k+1} = x_k + hf(t_k, x_k) + \mathbf{O}(h^2)$$

Algorithm 3 Euler's method

```
procedure EULER'S METHOD(f, t_0, x_0, t_M, M)
    h ← (t_M − t_0)/M                ▷ step size
    x = [x_0, x_1, ··· , x_M]        ▷ values of x(t)
    t = [t_0, t_1, ··· , t_M]        ▷ values of t
    for k = 1 to M do
        x_{k+1} ← x_k + h f(t_k, x_k)
        t_{k+1} ← t_k + h
    end for
    return t, x
end procedure
```

5.1.3 Runge-Kutta methods

Let the IVP be

$$\begin{cases} x'(t) = f(t, x) \\ x(t_0) = x_0 \end{cases}$$

By chain rule, we have

$$x'' = \frac{\partial}{\partial t} f + x' \frac{\partial}{\partial x} f = \frac{\partial}{\partial t} f + f \frac{\partial}{\partial x} f$$

For the Taylor expansion of x, we have

$$\begin{aligned} x(t+h) &= x(t) + hx'(t) + \frac{h^2}{2}x''(t) + \mathbf{O}(h^3) \\ &= x(t) + hf + \frac{h^2}{2}(f_t + f_x f) + \mathbf{O}(h^3) \\ &= x(t) + \frac{h}{2}f + \frac{h}{2}(f + hf_t + hf_x f) + \mathbf{O}(h^3) \\ &= x(t) + \frac{h}{2}f + \frac{h}{2}f(t+h, x+hf) + \mathbf{O}(h^3) \end{aligned}$$

Than, the second-order Runge-Kutta method (RK2) is

$$x(t+h) = x(t) + \frac{h}{2}f(t,x) + \frac{h}{2}f(t+h, x+hf) + \mathbf{O}(h^3)$$

or, equivalently

$$x(t+h) = x(t) + \frac{h}{2}(F_1 + F_2)$$

which $F_1 = f(t,x)$, $F_2 = f(t+h, x+hf)$. It is also known as **Heun's method**[4].

The fourth-order Runge-Kutta method (RK4)[4] is

$$x(t+h) = x(t) + \frac{h}{6}(F_1 + 2F_2 + 2F_3 + F_4) + \mathbf{O}(h^5)$$

where

$$\begin{cases} F_1 = f(t,x) \\ F_2 = f(t+\frac{h}{2}, x+\frac{1}{2}F_1) \\ F_3 = f(t+\frac{h}{2}, x+\frac{1}{2}F_2) \\ F_4 = f(t+h, x+F_3) \end{cases}$$

5.1.4 Heun's method

Algorithm 4 Heun's method

```
procedure HEUN'S METHOD(f, t_0, x_0, t_M, M)
    h ← (t_M − t_0)/M                          ▷ step size
    x = [x_0, x_1, ··· , x_M]                  ▷ values of x(t)
    t = [t_0, t_1, ··· , t_M]                  ▷ values of t
    for k = 1 to M do
        x_{k+1} ← x_k + (h/2)(f(t_k, x_k) + f(t_k + h, x_k + hf(t_k, x_k)))
        t_{k+1} ← t_k + h
    end for
    return t, x
end procedure
```

5.1.5 Modified Euler's method

In general, 2nd-order RK method needs

$$\begin{aligned} x(t+h) &= x(t) + \omega_1 hf + \omega_2 hf(t+\alpha h, x+\beta hf) + \mathbf{O}(h^3) \\ &= x(t) + (\omega_1+\omega_2)hf + \alpha\omega_1 h^2 f_t + \beta\omega_2 h^2 f f_x + \mathbf{O}(h^3) \\ \text{(compare with)} &= x(t) + hf + \frac{h^2}{2}(f_t + f_x f) + \mathbf{O}(h^3) \end{aligned}$$

We have

$$\omega_1 + \omega_2 = 1, \alpha\omega_1 = \frac{1}{2}, \beta\omega_2 = \frac{1}{2}$$

The Heun's method is obtained by setting

$$\omega_1 = \omega_2 = \frac{1}{2}, \alpha = \beta = 1$$

Alternatively, setting

$$\omega_1 = 0, \omega_2 = 1, \alpha = \beta = \frac{1}{2}$$

Then we obtain the modified Euler method[4]:

$$x(t+h) = x(t) + hf(t + \frac{h}{2}, f(t+h, x(t) + \frac{h}{2}f(t,x))$$

or, equivalently

$$x_{k+1} = x_k + hf(t_k + \frac{h}{2}, f(t_k + h, x_k + \frac{h}{2}f(t_k, x_k))$$

Algorithm 5 Modified Euler's method

```
procedure MODIFIED EULER'S METHOD(f, t_0, x_0, t_M, M)
    h ← (t_M − t_0)/M                          ▷ step size
    x = [x_1, x_2, ··· , x_{M+1}]              ▷ values of x(t)
    t = [t_1, t_2, ··· , t_{M+1}]              ▷ values of t
    for k = 1 to M do
        x_{k+1} ← x_k + hf(t_k + h/2, f(t_k + h, x_k + (h/2) f(t_k, x_k)))
        t_{k+1} ← t_k + h
    end for
    return t, x
end procedure
```

5.2 Multistep methods

5.2.1 Adams-Bashforth formula

[8]The kth-order Adams-Bashforth formula[4] (explicit) is

$$x_{n+1} = x_n + a_n f_n + a_{n-1} f_{n-1} + \cdots + a_{n-k+1} f_{n-k+1}$$

where $f_i = f(t_i, x_i)$

For the 5 order Adams-Bashforth formula

$$x_{n+1} = x_n + \frac{h}{720}\left[1901 f_n - 2774 f_{n-1} + 2616 f_{n-2}\grave{}\, 1274 f_{n-3} + 251 f_{n-4}\right]$$

5.2.2 Adams-Moulton formula

The kth-order Adams-Moulton formula[4] (implicit) is

$$x_{n+1} = x_n + a_{n+1}f_{n+1} + a_{n-1}f_{n-1} + \cdots + a_{n-k+2}f_{n-k+2}$$

where $f_i = f(t_i, x_i)$

For the 5 order Adams-Moulton formula

$$x_{n+1} = x_n + \frac{h}{720}\left[251f_{n+1} + 646f_n - 264f_{n-1} + 106f_{n-2} - 19f_{n-3}\right]$$

5.3 Stiff equations: the stability of the numerical method

Stiffness[13], in a system of ordinary differential equations, refers to a wide disparity in the time scales of the components in the vector solution. Some numerical procedures that are quite satisfactory in general will perform poorly on stiff equations. This happens when stability in the numerical solution can be achieved with a tiny step size.

A stiff differential equation is numerically unstable unless the step size is extremely small.

Stiff differential equations are characterized as those whose exact solution has a term of the form e^{-ct}, where c is a large positive constant.

5.3.1 Example

Consider the IVP

$$\begin{cases} x'(t) = \lambda x, \lambda < 0 \\ x(t_0) = x_0 \end{cases}$$

The exact solution of the IVP is $x(t) = x_0 e^{\lambda t}$. Using forward Euler's method to solve the IVP, we have

$$\begin{aligned} x_{n+1} &= x_n + hf(t_n, x_n) = (1 + \lambda h)x_n \\ x_n &= (1 + \lambda h)x_{n-1} \\ &\vdots \\ x_1 &= (1 + \lambda h)x_0 \end{aligned}$$

Then, we get a geometric sequence $x_n = x_0(1+\lambda h)^n$. The sequence converges if $|1+\lambda h| < 1$. So, the method is stable when the step size $h < \frac{2}{|\lambda|}$.

On the other hand, use backward Euler's method, we have

$$\begin{aligned} x_{n+1} &= x_n + hf(t_{n+1}, x_{n+1}) = x_n + \lambda h x_{n+1} \\ \Leftrightarrow x_{n+1} &= \frac{1}{1 - \lambda h}x_n \\ x_n &= \frac{1}{1 - \lambda h}x_{n-1} \\ &\vdots \\ x_1 &= \frac{1}{1 - \lambda h}x_0 \end{aligned}$$

Then, we get a geometric sequence $x_n = x_0(\frac{1}{1-\lambda h})^n$. The sequence converges if $|\frac{1}{1-\lambda h}| < 1$. So, the method is stable for all $h \in \mathbb{R}$.

In conclusion, explicit method is cheap but conditionally stable. Implicit method is expensive but unconditionally stable[4].

5.4 System and higher order ordinary differential equations

The general form of an n-th order initial value problem (IVP) for an ordinary differential equation (ODE) is expressed as follows:

$$y^{(n)}(t) = f\left(t, y(t), y^{(1)}(t), y^{(2)}(t), \ldots, y^{(n-1)}(t)\right)$$

corresponding the $n-1$ initial condition

$$y(t_0) = y_0, y'(t_0) = y_0^{(1)}, y''(t_0) = y_0^{(2)}, \cdots, y^{(n-1)} = y_0^{(n-1)}$$

where y_0, $y_0^{(1)}$, $y_0^{(2)}$, ..., $y_0^{(n-1)}$ are given constants.

5.4.1 Converting to a System of First-Order ODEs

The original n-th ODE problem is

$$y^{(n)}(t) = f\left(t, y(t), y^{(1)}(t), y^{(2)}(t), \ldots, y^{(n-1)}(t)\right)$$

with the $n-1$ initial condition

$$y(t_0) = y_0, y'(t_0) = y_0^{(1)}, y''(t_0) = y_0^{(2)}, \cdots, y^{(n-1)}(t_0) = y_0^{(n-1)}$$

Then, the system of first-order ODEs is defined by:

$$\begin{cases} y_0(t) & = y(t) \\ y_1(t) & = y_0'(t) = y'(t) \\ y_2(t) & = y_1'(t) = y''(t) \\ & \vdots \\ y_{n-1}(t) & = y_{n-2}'(t) = y^{(n-1)}(t) \\ y_n(t) & = f\left(t, y_1, y_2, \cdots, y_{n-1}\right) \end{cases} \Leftrightarrow \begin{bmatrix} y_0'(t) \\ y_1'(t) \\ \vdots \\ y_{n-1}'(t) \end{bmatrix} = \begin{bmatrix} y_1(t) \\ y_2(t) \\ \vdots \\ f(t, y_1, y_2, \cdots, y_{n-1}) \end{bmatrix}$$

It is equivalent to a vector value ODE $\mathbf{Y}' = \mathbf{F}(t, \mathbf{Y})$, with the initial condition

$$\begin{cases} y_0(t_0) & = y(t_0) \\ y_1(t_0) & = y_0'(t_0) \\ y_2(t_0) & = y_1'(t_0) \\ & \vdots \\ y_{n-1}(t_0) & = y_{n-2}'(t_0) \end{cases} \Leftrightarrow \mathbf{Y}(t_0) = \mathbf{Y}_0 = \begin{bmatrix} y_0 \\ y_0^{(1)} \\ y_0^{(2)} \\ \vdots \\ y_0^{(n-1)} \end{bmatrix}$$

To solve the IVP $\mathbf{Y}' = \mathbf{F}(t, \mathbf{Y}), \mathbf{Y}(t_0) = \mathbf{Y}_0$, we can still use the previous single-step methods or multistep methods.

5.4.2 Example

Consider the following mass-spring-damper systems,

$$\begin{cases} m\ddot{x} + c\dot{x} + kx = 0 \\ x(0) = x_0 \\ \dot{x}(0) = v_0 \end{cases}$$

We convert this second order IVP into a system of the first-order IVPs.

$$\begin{cases} x_0(t) = x(t) \\ x_1(t) = x_0'(t) \\ x_2(t) = x_1'(t) = -\frac{c}{m}x_1(t) - \frac{k}{m}x_0(t) \end{cases}$$

with the initial condition $x_0(0) = x_0, x_1(0) = v_0$

Then, we change the system of the first-order IVPs into a linear system

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\frac{k}{m} & -\frac{c}{m} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \end{bmatrix}$$

with initial condition $\vec{x} = \begin{bmatrix} x_0(0) \\ x_1(0) \end{bmatrix} = \begin{bmatrix} x_0 \\ v_0 \end{bmatrix}$

Given $m = 1$, $c = 0.5$, and $k = 2$, and initial conditions $x(0) = 1$ and $\dot{x}(0) = 0$. we solve this system using MATLAB `ode45` over the time span $t = [0, 10]$.

Introduction of MATLAB `ode45`

`ode45` Solve non-stiff differential equations, medium order method.

`[TOUT,YOUT] = ode45(ODEFUN,TSPAN,Y0)` integrates the system of differential equations `y' = f(t,y)` from time `TSPAN(1)` to `TSPAN(end)` with initial conditions `Y0`. Each row in the solution array `YOUT` corresponds to a time in the column vector `TOUT`.

- `ODEFUN` is a function handle. For a scalar T and a vector Y, `ODEFUN(T,Y)` must return a column vector corresponding to f(t,y).
- `TSPAN` is a two-element vector `[T0 TFINAL]` or a vector with several time points `[T0 T1 ... TFINAL]`. If you specify more than two time points, ode45 returns interpolated solutions at the requested times.
- `Y0` is a column vector of initial conditions, one for each equation.

In solving the first order systems with n equations, `x0` and `YOUT` have n rows.

Result:

MATLAB Code:

```
% setting problem
m = 1;
c = 0.5;
k = 2;
x0 = [1; 0];     % Initial conditions
t = [0 10];  % Time vector
% system of first order ODE
f = @(t, x) [0, 1; -(k/m), -(c/m)]*x;
```

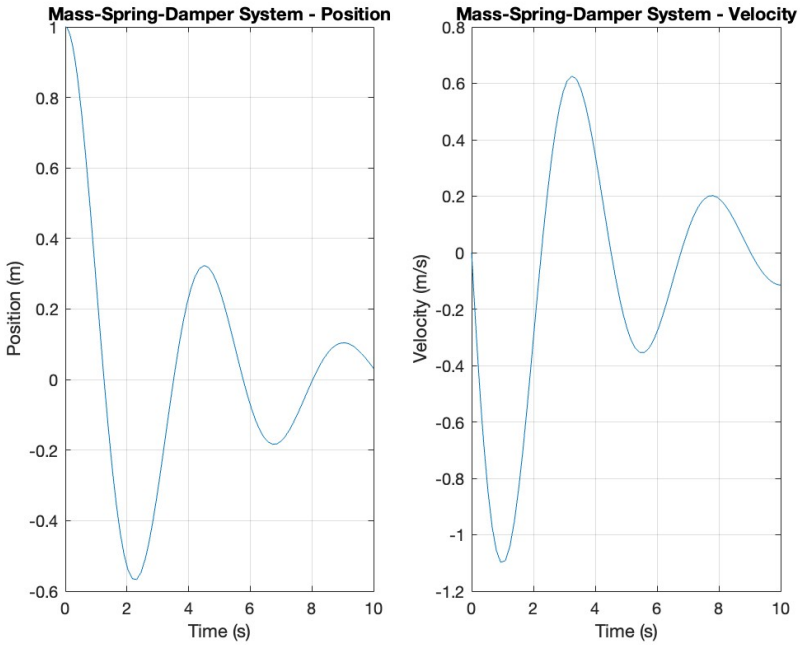


Figure 5: result of mass-spring-damper systems

```
% solving by ode45
[tout, yout] = ode45(f, t, x0);

% plot solution
subplot(1, 2, 1);
plot(tout, yout(:, 1));
xlabel('Time (s)');
ylabel('Position (m)');
title('Mass-Spring-Damper System - Position');

subplot(1, 2, 2);
plot(tout, yout(:, 2));
xlabel('Time (s)');
ylabel('Velocity (m/s)');
title('Mass-Spring-Damper System - Velocity');
```

5.5 Boundary-value problems

An *ODE boundary value problem*[6] consists of an ODE in some interval $[a, b]$ and a set of 'boundary conditions' involving the data at both endpoints.

After converting to a first order system, any BVP can be written as a system of m-equations for a solution $\mathbf{y}(x) : \mathbb{R} \to \mathbb{R}^m$ satisfying

$$\begin{cases} \mathbf{y}' = F(t, \mathbf{y}), x \in [a, b] & \text{differential equations} \\ \mathbf{y}(a) = \mathbf{y}_a, \mathbf{y}(b) = \mathbf{y}_b, & \text{boundary conditions} \end{cases}$$

5.5.1 The numerical method of solving linear BVP

Let the linear two-points boundary value problem be

$$\begin{cases} x'' = u(t) + v(t)x + w(t)x', t \in [a, b] \\ x(a) = \alpha \\ x(b) = \beta \end{cases}$$

Given a uniform partition $a = t_0 < t_1 < \cdots < t_{n+1} = b$ on $[a, b]$, and let $h = \dfrac{b-a}{n+1}$. From the centered differencing formula[11] of x'', x',

$$\begin{cases} x'(t_i) = \frac{1}{2h}(x(t_{i+1}) - x(t_{i-1})) + \mathbf{O}(h^2) \\ x''(t_i) = \frac{1}{h^2}(x(t_{i+1}) - 2x(t_i) + x(t_{i-1})) + \mathbf{O}(h^2) \end{cases}$$

We set $u_i = u(t_i)$, $v_i = v(t_i)$, $w_i = w(t_i)$ and use $y_i \approx x(t_i)$. Then, we have

$$\frac{1}{h^2}(y_{i+1} - 2y_i + y_{i-1}) = u_i + v_i y_i + w_i \frac{y_{i+1} - y_{i-1}}{2h}$$

Multiply by h^2 and rearrange to obtain

$$\begin{cases} (-1 - \frac{h}{2}w_i)y_{i-1} + (2 + h^2 v_i)y_i + (-1 + \frac{h}{2}w_i)y_{i+1} = -h^2 u_i, \quad i = 1, 2, \cdots, n \\ y_0 = \alpha \\ y_{n+1} = \beta \end{cases}$$

Let $a_{i-1} = -1 - \frac{h}{2}w_i$, $d_i = 2 + h^2 v_i$, $c_i = -1 + \frac{h}{2}w_i$, $b_i = -h^2 u_i$. Then, we obtain a tridiagonal system

$$\begin{bmatrix} d_1 & c_1 & & & & \\ a_1 & d_2 & c_2 & & & \\ & a_2 & d_3 & c_3 & & \\ & & \ddots & \ddots & \ddots & \\ & & & a_{n-2} & d_{n-1} & c_{n-1} \\ & & & & a_{n-1} & d_n \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \\ y_n \end{bmatrix} = \begin{bmatrix} b_1 - a_0\alpha \\ b_2 \\ b_3 \\ \vdots \\ b_{n-1} \\ b_n - c_n\beta \end{bmatrix}$$

The system can be solved by a special Gaussian elimination (GE) algorithm. Also the matrix is diagonally dominant if $v_i > 0$ and h is small enough so that $|\frac{1}{2}hw_i| < 1$, which implies that the GE algorithm does not require pivoting[4].

6 Appendix

6.1 MATLAB Code of standard form for polynomial Interpolation

```
function v = polyinterp_standard(x, y, u)
    A = vander(x);      % the vandermonde matrix of x
    c = A\y;            % coefficient of the interplating polynomial
    v = polyval(c, u);  % calculate the polynomial v = p(u)
end
```

6.2 MATLAB Code of Newton form for polynomial Interpolation

```
function [v] = interp_newton(x, y, u)
      c = interp_newton_constr(x, y);
      v = interp_newton_eval(x, c, u);
end

function [c] = interp_newton_constr(x, y)
  n = length(x) - 1;
  c = zeros(1, n + 1);
  c(1) = y(1);
  for k = 1:n
      for i = 1:k
          term = 1;
          for j = 1:k
              if j ~= i
                  term = term * (x(k+1) - x(j));
              end
          end
          c(k+1) = c(k+1) + (y(k+1) -
                   interp_newton_eval(x(1:k), c(1:k), x(k+1))) / term;
      end
  end
end

function [v] = interp_newton_eval(x, c, u)
    n = length(x) - 1;
    v = c(n + 1);
    for i = n:-1:1
        v = v .* (u - x(i)) + c(i);
    end
end
```

6.3 MATLAB Code of Lagrange form for polynomial Interpolation, [7]

```
function v = polyinterp(x,y,u)
    %POLYINTERP  Polynomial interpolation.
    %   v = POLYINTERP(x,y,u) computes v(j) = P(u(j)) where P is the
    %   polynomial of degree d = length(x)-1 with P(x(i)) = y(i).
```

```
    %   Copyright 2014 Cleve Moler
    %   Copyright 2014 The MathWorks, Inc.

    % Use Lagrangian representation.
    % Evaluate at all elements of u simultaneously.

    n = length(x);
    v = zeros(size(u));
    for k = 1:n
       w = ones(size(u));
       for j = [1:k-1 k+1:n]
          w = (u-x(j))./(x(k)-x(j)).*w;
       end
       v = v + w*y(k);
    end
```

6.4 MATLAB Code of Composite trapezoidal rule

```
function [I] = composite_trapezoidal_integration(f, a, b, err)
    % f: the function to be integrated
    % a, b: the intervals to be calculated integral of f
    % err: tolerance

    I = (b - a) * (f(a) + f(b)) / 2; % 初始積分值
    Iold = 0;
    k = 1;      % 迭代次數

    while (abs(I - Iold) >= err)
        % 如果積分值的誤差大於err則重算積分
        Iold = I;
        k = k+1;
        n = 2^k;
        x = linspace(a, b, n+1);    % 將區間分爲n+1等分
        I = (b-a)/n*(sum(f(x))-f(a)/2-f(b)/2);  % trapezoidal rule
    end
end
```

6.5 MATLAB Code of Composite Simpson's rule

```
function [I] = composite_simpson_rule(f, a, b, err)
    I = (b - a) * (f(a) + 4*f((a+b)/2) + f(b)) / 6; % 初始積分值
    Iold = 0;
    k = 1;      % 迭代次數

    while (abs(I - Iold) >= err)
        Iold = I;
        k=k+1;
```

```
        n = 4^k;
        x = linspace(a, b, n+1);
        I = 0;
        for i = 2:2:n
            I = I + f(x(i-1)) + 4*f(x(i)) + f(x(i+1));
        end
        I = I*(x(2)-x(1))/3;
    end
    disp(k);
end
```

6.6 MATLAB Code of Euler's method

```
function [t_values, x_values] = ode_ivp_euler(f, t0, x0, tm, M)
    % EULER_METHOD Solves an initial value problem (IVP) using Euler's method.
    %   [x_values, y_values] = euler_method(f, x0, y0, xn, h) solves the IVP
    %   dy/dx = f(x, y) with initial condition y(x0) = y0 using Euler's method.
    %   f is the function handle for the ODE, x0 is the initial value of x,
    %   y0 is the initial value of y, xn is the final value of x, and h is
    %   the step size.
    %
    %   Returns the values of x and y calculated by Euler's method.

    % Number of steps
    h = (tm - t0) / M;

    % Arrays to store values of t and x
    t_values = zeros(M+1, 1);
    x_values = zeros(M+1, 1);

    % Initial values
    t_values(1) = t0;
    x_values(1) = x0;

    % Euler's method
    for i = 1:M
        t = t_values(i);
        x = x_values(i);
        x_new = x + h * f(t, x); % Euler's method formula
        t_values(i+1) = t + h;
        x_values(i+1) = x_new;
    end

end
```

6.7 MATLAB Code of Heun's method

```
function [t_values, x_values] = ode_ivp_heun(f, t0, x0, tm, M)
```

```
    % HEUNS_METHOD Solves an initial value problem (IVP) using Heun's method.
    %   [x_values, y_values] = heuns_method(f, x0, y0, xn, h) solves the IVP
    %   dy/dx = f(x, y) with initial condition y(x0) = y0 using Heun's method.
    %   f is the function handle for the ODE, x0 is the initial value of x,
    %   y0 is the initial value of y, xn is the final value of x, and h is
    %   the step size.
    %
    %   Returns the values of x and y calculated by Heun's method.

    % Number of steps
    h = (tm - t0) / M;

    % Arrays to store values of t and x
    t_values = zeros(M+1, 1);
    x_values = zeros(M+1, 1);

    % Initial values
    t_values(1) = t0;
    x_values(1) = x0;

    % Heun's method
    for i = 1:M
        t = t_values(i);
        x = x_values(i);

        x_new = x + (h/2) * (f(t, x) + f(t + h, x + h * f(t, x)));

        t_values(i+1) = t + h;
        x_values(i+1) = x_new;
    end

end
```

6.8 MATLAB Code of modified Euler's method

```
function [t_values, x_values] = ode_ivp_modified_euler(f, t0, x0, tm, M)
    % MODIFIED_EULER_METHOD Solves an initial value problem (IVP)
        using Modified Euler's method.
    %   [x_values, y_values] = modified_euler_method(f, x0, y0, xn, h) solves the IVP
    %   dy/dx = f(x, y) with initial condition y(x0) = y0 using Modified Euler's method.
    %   f is the function handle for the ODE, x0 is the initial value of x,
    %   y0 is the initial value of y, xn is the final value of x, and h is
    %   the step size.
    %
    %   Returns the values of x and y calculated by Modified Euler's method.

    % Number of steps
    h = (tm - t0) / M;
```

```
    % Arrays to store values of t and x
    t_values = zeros(M+1, 1);
    x_values = zeros(M+1, 1);
    % Initial values
    t_values(1) = t0;
    x_values(1) = x0;

    % Modified Euler's method
    for i = 1:M
        t = t_values(i);
        x = x_values(i);

        x_corrector = x + h * f(t + h/2, x + h * f(t, x));

        t_values(i+1) = t + h;
        x_values(i+1) = x_corrector;
    end

end
```

References

[1] Elena Celledoni. Interpolation summary. Technical report, Department of Mathematical Sciences, NTNU, February 2020. https://wiki.math.ntnu.no/_media/ma2501/2020v/interpolationsummary.pdf.

[2] Feng-Nan Hwang. Lecture notes on interpolation. Technical report, Department of Mathematics, National Central University, 2024.

[3] Feng-Nan Hwang. Lecture notes on numerical differentiation and integration. Technical report, Department of Mathematics, National Central University, 2024.

[4] Feng-Nan Hwang. Lecture notes on ordinary differential equations. Technical report, Department of Mathematics, National Central University, 2024.

[5] LibreTexts. Numerical integration - midpoint, trapezoid, simpson's rule. https://math.libretexts.org/Courses/Mount_Royal_University/MATH_2200%3A_Calculus_for_Scientists_II/2%3A_Techniques_of_Integration/2.5%3A_Numerical_Integration_-_Midpoint%2C_Trapezoid%2C_Simpson's_rule.

[6] HELM (Helping Engineers Learn Mathematics). Numerical boundary value problems. https://www.lboro.ac.uk/media/media/schoolanddepartments/mlsc/downloads/HELM%20Workbook%2033%20Numerical%20Boundary%20Value%20Problems.pdf.

[7] MathWorks. Polyinterp: Polynomial interpolation. https://viewer.mathworks.com/?viewer=plain_code&url=https%3A%2F%2Fde.mathworks.com%2Fmatlabcentral%2Fmlc-downloads%2Fdownloads%2Fsubmissions%2F37976%2Fversions%2F7%2Fcontents%2Fpolyinterp.m&embed=web.

[8] Unknown. Appendix i: Numerical methods for ordinary differential equations, multistep methods. https://vmm.math.uci.edu/ODEandCM/PDF_Files/Appendices/AppendixI.pdf.

[9] Unknown. Lecture 1: Polynomial interpolation. Technical report, Department of Mathematical Sciences, Duke University. https://services.math.duke.edu/~jtwong/math563-2020/lectures/Lec1-polyinterp.pdf.

[10] Wang Weichung. Richardson extrapolation. https://www.math.nthu.edu.tw/~wangwc/teaching/NA_10f/slides/Richardson_extrapolation.pdf.

[11] Jeffrey Wong. Numerical methods for boundary value problems. https://services.math.duke.edu/~jtwong/math563-2020/lectures/Lec9-BVPs.pdf, April 2020.

[12] Toby J. Wong. Lecture notes on derivatives. https://services.math.duke.edu/~jtwong/math563-2020/lectures/Lec2-derivatives.pdf. Accessed: 2024-06-15.

[13] Zhiliang Xu. Lecture notes on numerical methods for partial differential equations. https://www3.nd.edu/~zxu2/acms40390F15/Lec-5.11.pdf. Accessed: 2024-06-15.