# 第四次上机作业

学号: 15220212202842　姓名: 陈子睿

October $18^{th}$ 2023

## 1 快速离散傅里叶变换 (FFT)

### 1.1 问题描述

离散傅里叶变换可以解释为三角函数多项式插值问题：给定 $f(x)$ 在 $[0, 2\pi]$ 中 $n$ 个等距节点 $(x_j = \frac{2\pi j}{n}, j = 0, 1, 2, ..., n-1)$ 上的函数值 $f_j = f(x_j)$, 求三角函数多项式 $\varphi(x) = \sum_{k=0}^{n-1} c_k e^{ikx}$ 中的系数 $c_k$，由此计算 $f_j = \varphi(x_j) = \sum_{k=0}^{n-1} c_k e^{ikx_j} = \sum_{k=0}^{n-1} c_k W^{-kj}$.

a) Matlab 中实现快速离散 Fourier 变换的函数主要有两个 fft，ifft，文档中关于它们的说明是这样的：$Y = fft(X)$ and $X = ifft(Y)$ implement the Fourier transform and inverse Fourier transform, respectively. For X and Y of length n, these transforms are defined as follows:

$$Y(k) = \sum_{j=1}^{n} X(j) W^{(j-1)(k-1)}, X(k) = \frac{1}{n} \sum_{j=1}^{n} Y(j) W^{(j-1)(k-1)}$$

其中 $W = e^{-i\frac{2\pi}{n}}$，我们考虑执行代码：

```
Y = fft(f)  % f(j)=f_j
```

并且分析返回值 $Y$ 与 $c_k = \sum_{l=0}^{N-1} a_l W^{kl}, k = 0, 1, ..., N-1$ 系数的关系，并且用实际算例验证. 另外验证 $ifft(fft(f)) = f$.

b) 离散傅里叶变换也可解释为另一个三角函数多项式插值问题：给定在 $[0, 2\pi]$ 中 n 个等距节点 $(x_j = \frac{2\pi j}{n}, j = 0, 1, 2, ..., n-1)$ 上的函数值 $f_j$，求三角函数多项式 $\tilde{\varphi}(x) = \sum_{k=-m_1}^{m_2} \tilde{c}_k e^{ikx}, m_1 = [\frac{n}{2}], m_2 = [\frac{n-1}{2}]$ 中的系数 $\tilde{c}_k$，s.t.$\tilde{\varphi}(x_j) = f_j$. 利用 $e^{ikx}$ 的离散正交性证明：

1）当 $0 \le k \le m_2$ 时，$\tilde{c}_k = c_k$

2）当 $-m_1 \le k < 0$ 时，$\tilde{c}_k = c_{k+n}$.

我们考虑执行代码：

```
Y = fft(f)  % f(j)=f_j
```

分析返回值 $Y$ 与系数 $\tilde{c}_k$ 的关系，例如取 $n=10$，$f(j)=\cos 2x_j=\frac{e^{-i2x}+e^{i2x}}{2}$ 对应的 $\tilde{c}_k(-m_1\le k\le m_2)$ 系数应该为 $[0,0,0,0.5,0,0,0,0.5,0,0]$，设例验证这个例子中的返回值 Y 是否与 a) 推测的关系符合.

c) 考虑如下的微分方程求解问题：已知函数 $f(x)$，求满足以下方程的函数 $u(x)$,s.t.

$$-\frac{d^2}{dx^2}u(x_j)+u(x_j)=f(x_j), j=0,1,...,n-1 \tag{1}$$

其中 $f(x)$,$u(x)$ 均为 $[0,2\pi]$ 上的周期的函数.Fourier 配点法（collocation method）是求解微分方程近似解的一种方法，可描述为求 $\tilde{u}(x)=\sum_{k=-m_1}^{m_2}\tilde{u}_k e^{ikx}$，s.t.

$$-\frac{d^2}{dx^2}\tilde{u}(x_j)+\tilde{u}(x_j)=f(x_j), j=0,1,...,n-1 \tag{2}$$

考虑求出三角多项式插值 $\tilde{f}(x)=\sum_{k=-m_1}^{m_2}\tilde{f}_k e^{ikx}$，则上面的问题等价于：

$$-\frac{d^2}{dx^2}u(x)+u(x)=f(x) \tag{3}$$

随后编写程序实现算法，并使用算例 $f(x)=\frac{2\sin x}{2+\cos x}-\frac{2\sin^3 x}{(2+\cos x)^3}-\frac{3\sin x\cos x}{(2+\cos x)^2}$，有精确解为 $u=\frac{\sin x}{2+\cos x}$，分别取 $n=5,6,7,...,21$ 计算一组点上的最大误差，绘制误差随 $n$ 的变化, 猜测误差与 n 的关系.

接下来我们将 Fourier 配点法稍作修改：求 $\tilde{u}(x)=\sum_{k=0}^{n-1}\hat{u}_k e^{ikx}$s.t.

$$-\tfrac{d^2}{dx^2}u(x_j)+u(x_j)=f(x_j)$$

取与上面相同的算例，记录点上最大误差 $\underset{0\le j<n}{max}|u(x_j)-\tilde{u}(x_j)|$ 随 n 的变化.

## 1.2 公式推导以及数值结果

**a)** 我们考虑使用 Matlab 代码计算一个函数 $f(x)$ 在一系列等距节点上的 DFT，然后由定义计算其对应的系数 $c_k$，并验证 $ifft(fft(f))=f$.

首先，我们定义一个函数 $f(x)$，例如 $f(x)=\sin(x)+e^x$. 我们在 $[0,2\pi]$ 区间内选择 $n$ 个等距点，并在这些点上计算 $f(x)$ 的值。然后，使用 fft 函数计算这些值的 DFT，并分析得到的结果 $Y$ 与 $c_k$ 系数的关系.

由 $c_k$ 定义以及 Matlab 对于 fft 函数的定义立刻可以得到：

$$c_k=\frac{Y(k+1)}{n} \tag{4}$$

汇总结果如下：

| $c_k$ 系数 | FFT 结果 (调整后) | 原始 $f$ | 重构的 $f$ |
|---|---|---|---|
| 55.9898 + 0.0000i | 55.9898 + 0.0000i | 1.0000 | 1.0000 |
| 13.5878 + 37.7532i | 13.5878 + 37.7532i | 2.9004 | 2.9004 |
| -11.4984 + 25.2193i | -11.4984 + 25.2193i | 5.8105 | 5.8105 |
| -19.1229 + 11.6263i | -19.1229 + 11.6263i | 11.2578 | 11.2578 |
| -20.9225 - 0.0000i | -20.9225 + 0.0000i | 23.1407 | 23.1407 |
| -19.1229 - 11.6263i | -19.1229 - 11.6263i | 50.0469 | 50.0469 |
| -11.4984 - 25.2193i | -11.4984 - 25.2193i | 110.3178 | 110.3178 |
| 13.5878 - 37.7532i | 13.5878 - 37.7532i | 243.4440 | 243.4440 |

表 1: DFT 计算结果

**b)** 为了分析离散傅里叶变换（DFT）中的 $\tilde{c}_k$ 系数与 fft 函数的返回值 $Y$ 之间的关系，我们需要先理解 $\tilde{c}_k$ 的计算方式以及如何从 $Y$ 得到 $\tilde{c}_k$. 在这个问题中，我们考虑的是在 $[0, 2\pi]$ 区间上 $n$ 个等距节点上的函数值 $f_j$，并寻找一个三角函数多项式 $\tilde{\varphi}(x)$ 来插值这些点. 在这种情况下，$\tilde{c}_k$ 的计算方式稍有不同于标准的 DFT.

**命题 1** 当 $0 \le k \le m_2$ 时，$\tilde{c}_k = c_k$

**证明** 首先，根据 $\tilde{c}_k$ 的定义，我们有：

$$\tilde{c}_k = \frac{1}{n} \sum_{j=0}^{n-1} f_j e^{-i\frac{2\pi kj}{n}} \tag{5}$$

由于 $f_j = \sum_{l=-m_1}^{m_2} \tilde{c}_l e^{i\frac{2\pi lj}{n}}$（这是由三角函数多项式插值定义的），我们将 $f_j$ 代入 $\tilde{c}_k$ 的公式中得到：

$$\tilde{c}_k = \frac{1}{n} \sum_{j=0}^{n-1} \Big( \sum_{l=-m_1}^{m_2} e^{i\frac{2\pi lj}{n}} e^{-i\frac{2\pi kj}{n}} \Big) \tag{6}$$

交换求和顺序：

$$\tilde{c}_k = \frac{1}{n} \sum_{l=-m_1}^{m_2} \tilde{c}_l \Big( \sum_{j=0}^{n-1} e^{i\frac{2\pi lj}{n}} e^{-i\frac{2\pi kj}{n}} \Big) \tag{7}$$

根据 $e^{ikx}$ 的离散正交性，内部求和将仅在 $l = k$ 时为 $n$，否则为 0。因此，对于 $0 \le k \le m_2$：

$$\tilde{c}_k = \frac{1}{n} \cdot n\tilde{c}_k = c_k \tag{8}$$

**命题 2** 当 $-m_1 \le k < 0$ 时，$\tilde{c}_k = c_{k+n}$

**证明** 类似地，利用 $e^{ikx}$ 的离散正交性，对于 $-m_1 \leq k < 0$:

$$\tilde{c}_k = \frac{1}{n}\sum_{j=0}^{n-1} f_j e^{-i\frac{2\pi kj}{n}} \tag{9}$$

将 $f_j$ 的表达式代入，并考虑到 $e^{i\frac{2\pi(l+n)j}{n}} = e^{i\frac{2\pi lj}{n}}$（由于复指数函数的周期性），我们可以得到:

$$\tilde{c}_k = c_{k+n} \tag{10}$$

因为当 $l = k + n$ 时，$e^{i\frac{2\pi(l+n)j}{n}}$ 与 $e^{-i\frac{2\pi kj}{n}}$ 在 $j$ 的求和中将是正交的，而对于其他 $l$ 值，求和将为 0. 这样，我们就利用 $e^{ikx}$ 的离散正交性证明了这两个关系.

随后我们通过执行代码得到以下结果:

| Index | Coefficient |
|---|---|
| 1 | $-0.0000 + 0.0000i$ |
| 2 | $-0.0000 - 0.0000i$ |
| 3 | $0.5000 - 0.0000i$ |
| 4 | $0.0000 + 0.0000i$ |
| 5 | $-0.0000 + 0.0000i$ |
| 6 | $0.0000 + 0.0000i$ |
| 7 | $-0.0000 - 0.0000i$ |
| 8 | $0.0000 - 0.0000i$ |
| 9 | $0.5000 + 0.0000i$ |
| 10 | $-0.0000 + 0.0000i$ |

表 2: $\tilde{c}_k$ coefficients

验证易得，这与前面推测的关系式一致.

**c)** 为了解决这个微分方程求解问题，我们首先考虑原始微分方程:

$$-\frac{d^2}{dx^2}u(x) + u(x) = f(x) \tag{11}$$

其中 $u(x)$ 和 $f(x)$ 是 $[o, 2\pi]$ 上的周期函数. 我们将使用傅里叶级数对 $u(x)$ 和 $f(x)$ 进行离散化表示, 其可以被表示为:

$$u(x) = \sum_{k=-\infty}^{\infty} u_k e^{ikx} \tag{12}$$

$$f(x) = \sum_{k=-\infty}^{\infty} f_k e^{ikx} \tag{13}$$

随后我们考虑使用傅里叶配点法，我们选取一组点 $x_j$，通常是等间距的点，例如 $x_j = \frac{2\pi j}{n}$（其中 $j = 0, 1, \ldots, n-1$）. 然后，我们求解微分方程在这些点上的近似值。在这些点上，微分方程变为：

$$-\frac{d^2}{dx^2}\tilde{u}(x_j) + \tilde{u}(x_j) = f(x_j) \tag{14}$$

为了求解这个方程，我们需要计算 $\tilde{u}(x)$ 和 $\tilde{f}(x)$ 的傅立叶系数。对于傅立叶系数的计算，我们使用离散傅立叶变换（DFT）：

$$f_k = \frac{1}{n}\sum_{j=0}^{n-1} f(x_j)e^{-ikx_j} \tag{15}$$

使用这些傅立叶系数，我们可以近似 $f(x)$。对于 $\tilde{u}(x)$ ，其系数可以通过解析地求解微分方程得到：

$$u_k = \frac{f_k}{k^2+1} \tag{16}$$

由于 $u(x)$ 和 $f(x)$ 都是周期函数，它们的傅立叶系数 $u_k$ 和 $f_k$ 在 $k$ 非常大时会迅速趋于零。因此，我们可以限制 $k$ 的范围在 $[-m_1, m_2]$ 内.

最后，我们使用特定的函数 $f(x) = \frac{2\sin x}{2+\cos x} - \frac{2\sin^3 x}{(2+\cos x)^3} - \frac{3\sin x\cos x}{(2+\cos x)^2}$ 来计算误差。通过比较计算得到的 $\tilde{u}(x)$ 和精确解 $u = \frac{\sin x}{2+\cos x}$ 在一组点上的差异，我们可以估计误差.

误差可以定义为：

$$\epsilon = \max_{x\in[0,2\pi]} |\tilde{u}(x) - u(x)| \tag{17}$$

我们可以通过改变 $n$ 的值来观察误差如何变化，并据此猜测误差与 $n$ 的关系。通常，随着 $n$ 的增加，近似解会越来越接近精确解，因此误差应该会减小.

通过运行编写的 python 程序，最终得到结果如下图所示：

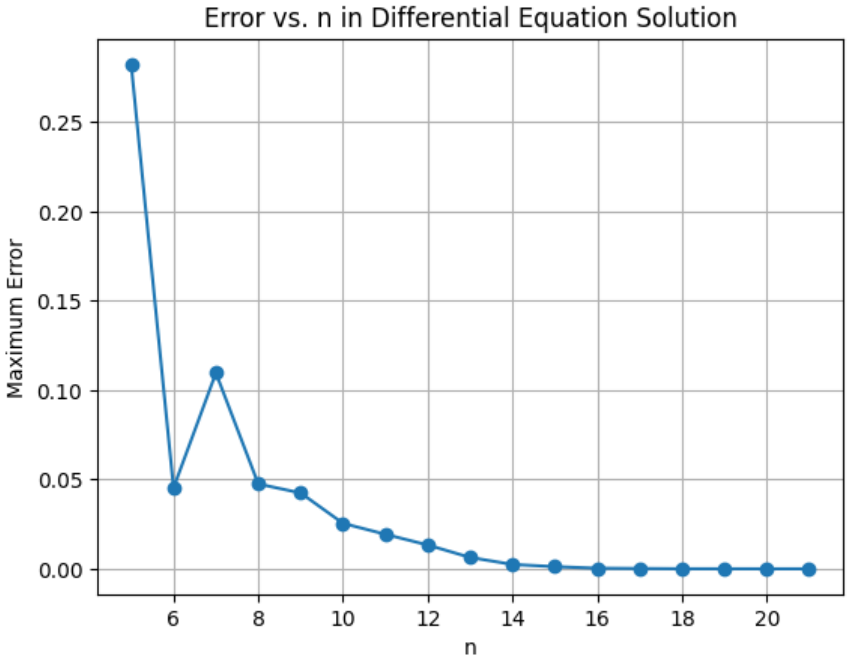


图 1: 微分方程近似解最大误差

倘若我们考虑修改后的傅里叶配点法，我们考虑的傅立叶级数仅包含非负的频率项：

$$\tilde{u}(x) = \sum_{k=0}^{n-1} \hat{u}_k e^{ikx} \tag{18}$$

这是一个重要的变化，因为它意味着我们在近似解中只考虑了一半的频率谱. 简化后的数值计算步骤如下：

1. **离散化**：在此方法中，我们使用的傅立叶级数变为：

$$\tilde{u}(x) = \sum_{k=0}^{n-1} \hat{u}_k e^{ikx} \tag{19}$$

2. **求解微分方程**：我们在给定的点 $x_j$ 上求解微分方程：

$$-\frac{d^2}{dx^2}u(x_j) + u(x_j) = f(x_j) \tag{20}$$

3. **计算误差**：误差计算为 $\max_{0\le j<n} |u(x_j) - \tilde{u}(x_j)|$.

通过运行编写的 python 程序，最终得到结果如下图所示：

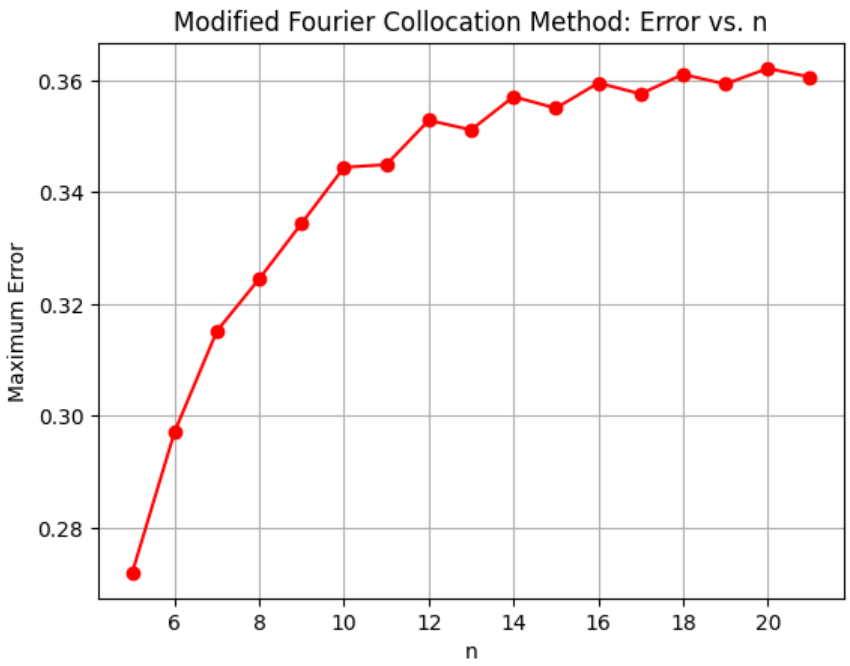


图 2: 微分方程近似解最大误差

我们观察到即使 $n$ 取值很大，误差也不会减小，考虑有以下原因：

1. **频率谱的不完整性**：由于我们只考虑了正频率和零频率，我们丢失了傅立叶级数中的负频率部分。这导致了近似解的频率谱不完整，从而影响了近似解的准确性.

2. **频率对称性**：在周期函数的傅立叶级数中，正频率和负频率通常是对称的，这有助于更准确地表示函数. 当省略负频率时，这种对称性被破坏，导致了近似解的不准确.

3. **高频振荡**：仅使用正频率可能导致近似解在高频部分出现不必要的振荡，这在数值解中是不希望出现的.

# 2 带权的最小二乘多项式拟合

## 2.1 问题描述

考虑编写一个函数用于计算带权的最小二乘多项式拟合，函数名、输入输出数据格式如下：

```
function [pp] = polyfit_weighted(x, y, m, w)
% 输入:
%   x, y: 1Xn 向量，欲拟合的数据点
%   m: 拟合多项式次数
%   w: 1Xn 向量，权值
% 输出:
%   pp: 1X (m+1) 向量，用于返回拟合多项式的系数，按逆序排列，即 pp(1) 表示 x^m 的系数。
```

a) 对下面的数据和权值调用程序计算四次拟合多项式，将结果与数据点画在一张图上，观察效果，解释权值的作用。函数编写完毕后需要先验证函数的正确性，当权值 $\omega = 1$ 为常数时，函数返回结果应该与 Matlab 自带函数 polyfit 的返回结果一致.

| $x_i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $y_i$ | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |

表 3: 数据点

| $\omega_i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 第一组 | 8 | 8 | 8 | 8 | 8 | 8 | 1 | 1 | 1 | 1 | 1 | 1 |
| 第二组 | 1 | 1 | 1 | 1 | 1 | 1 | 8 | 8 | 8 | 8 | 8 | 8 |

表 4: 权值

b) 使用公式

$$\tilde{\omega}_i = \frac{\omega_i}{\sum_{i-1}^{n} \omega_i} \tag{21}$$

对 a) 中的权进行规格化并且重新计算拟合多项式，并验证结果是否有区别.

c) 设计一组权值，使用四次多项式拟合，使得拟合曲线与下图基本一致：

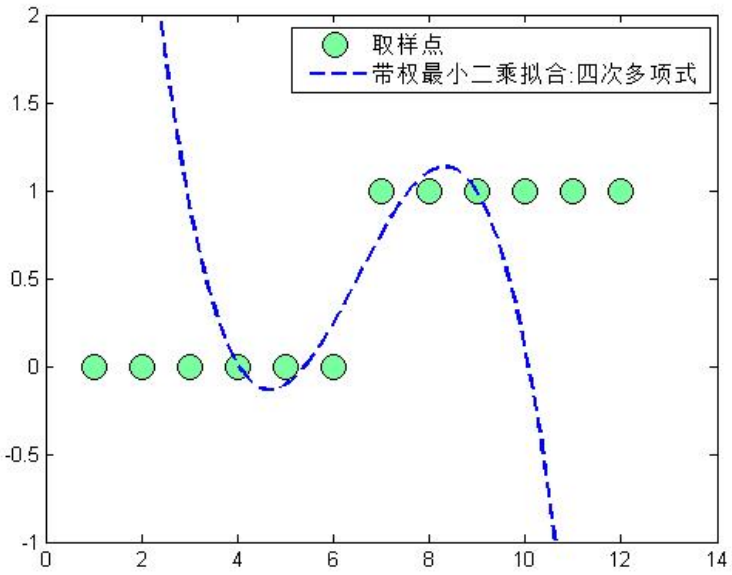


图 3: 加权最小二乘拟合

## 2.2 公式推导以及数值结果

带权的最小二乘多项式拟合的数学原理基于最小二乘法，但是在计算误差平方和时考虑了每个数据点的权重。具体来说，对于给定的数据点 $(x_i, y_i)$ 和权重 $\omega_i$，我们希望找到一个多项式 $p(x)$，使得加权的误差平方和最小，即

$$\min_{p(x)} \sum_{i=1}^{n} \omega_i [y_i - p(x_i)]^2 \tag{22}$$

其中 $p(x)$ 是一个 $m$ 次多项式，可以表示为:

$$p(x) = a_0 + a_1 x + a_2 x^2 + ... + a_m x^m \tag{23}$$

我们可以通过求解正规方程来找到最优的系数 $a_0, a_1, ..., a_m$. 正规方程是由加权的误差平方和对每个系数 $a_j$ 的偏导数等于零得到的一组线性方程. 对于每个 $j = 0, 1, ..., m$，我们有

$$\sum_{i=1}^{n} \omega_i x_i^j [y_i - p(x_i)] = 0 \tag{24}$$

也可以写成矩阵形式 $X^T W X a = X^T W y$，其中 $X$ 是设计矩阵，$W$ 是权重矩阵，$a$ 是系数向量，$y$ 是目标向量。然后，我们可以通过求解这个线性方程组来找到最优的系数向量 $a$.

**a)** 我们首先构造设计矩阵. 设计矩阵 $X$ 是一个每一行对应一个数据点，每一列对应一个多项式次数的矩阵。在这个矩阵中，每个元素 $X(i, j)$ 表示第 $i$ 个数据点的 $x$ 值的第 $j-1$ 次幂。这个矩阵是通过 bsxfun 函数计算 $x$ 的各个次数的幂来得到的.

随后我们构造权重矩阵. 权重矩阵 $W$ 是一个对角矩阵，其对角线上的元素就是各个数据点的权重 $\omega$.

通过向设计矩阵 $X$ 左乘权重矩阵 $W$，我们可以得到加权的设计矩阵和目标向量. 这样，每个数据点在最小二乘问题中的影响就与其权重成正比.

最后，我们通过求解线性方程组 $X/y$ 来得到拟合多项式的系数。这个线性方程组就是加权最小二乘问题的正规方程.

我们首先验证我们所编写的函数的正确性, 取权值 $\omega \equiv 1$ 运行我们编写的 polyfit_weighted 函数，并与 Matlab 自带的函数 polyfit 的返回结果进行对比，结果如下：

| | $a_4$ | $a_3$ | $a_2$ | $a_1$ | $a_0$ |
|---|---|---|---|---|---|
| polyfit _weighted | 0.0000 | -0.0054 | 0.1061 | -0.4479 | 0.4242 |
| polyfit | 0.0000 | -0.0054 | 0.1061 | -0.4479 | 0.4242 |

表 5: 对比结果

由此可见，程序是正确的，我们将结果与数据点呈现在一张图上，如下：

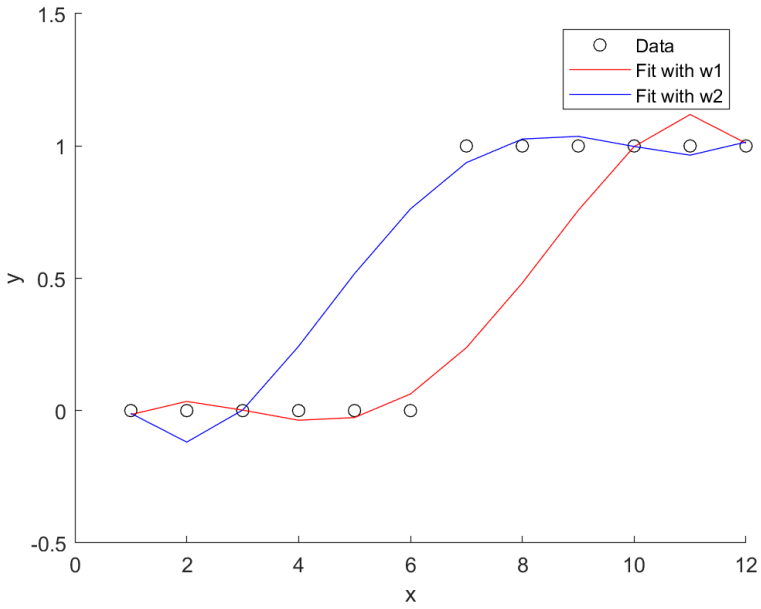


图 4: 加权最小二乘拟合结果 1

通过观察图像，我们可以得知权值具有以下作用:

权值在带权的最小二乘多项式拟合中起到了非常重要的作用. 每个数据点的权值决定了该数据点在拟合过程中的重要性. 权值越大，对应的数据点在拟合过程中的影响力越大. 这可以用于处理噪声数据，或者强调某些特定的数据点. 例如，如果某些数据点的测量精度更高，或者我们对某些数据点更感兴趣，我们可以通过增大这些数据点的权值来确保拟合结果更加符合这些数据点. 总的来说，权值提供了一种灵活的方式来控制拟合过程，使得我们可以根据实际需求来调整每个数据点的影响力.

**b)** 倘若我们要求使用规格化的权，即 $\sum_{i=1}^{n}\omega_i = 1$, 我们考虑使用 $\tilde{\omega}_i = \frac{\omega_i}{\sum_{i-1}^{n}\omega_i}$ 来对现有的权进行规格化，得到规格化后的权值如下:

| $\omega_i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 第一组 | $\frac{4}{27}$ | $\frac{4}{27}$ | $\frac{4}{27}$ | $\frac{4}{27}$ | $\frac{4}{27}$ | $\frac{4}{27}$ | $\frac{1}{54}$ | $\frac{1}{54}$ | $\frac{1}{54}$ | $\frac{1}{54}$ | $\frac{1}{54}$ | $\frac{1}{54}$ |
| 第二组 | $\frac{1}{54}$ | $\frac{1}{54}$ | $\frac{1}{54}$ | $\frac{1}{54}$ | $\frac{1}{54}$ | $\frac{1}{54}$ | $\frac{4}{27}$ | $\frac{4}{27}$ | $\frac{4}{27}$ | $\frac{4}{27}$ | $\frac{4}{27}$ | $\frac{4}{27}$ |

表 6: 规格化后的权值

我们将规格化后的权值输入程序，运行结果如下:

| | $a_4$ | $a_3$ | $a_2$ | $a_1$ | $a_0$ |
|---|---|---|---|---|---|
| 原权值 | -0.0010 | 0.0226 | -0.1527 | 0.3640 | -0.2479 |
| 规格化后的权值 | -0.0010 | 0.0226 | -0.1527 | 0.3640 | -0.2479 |

表 7: 对比结果 2

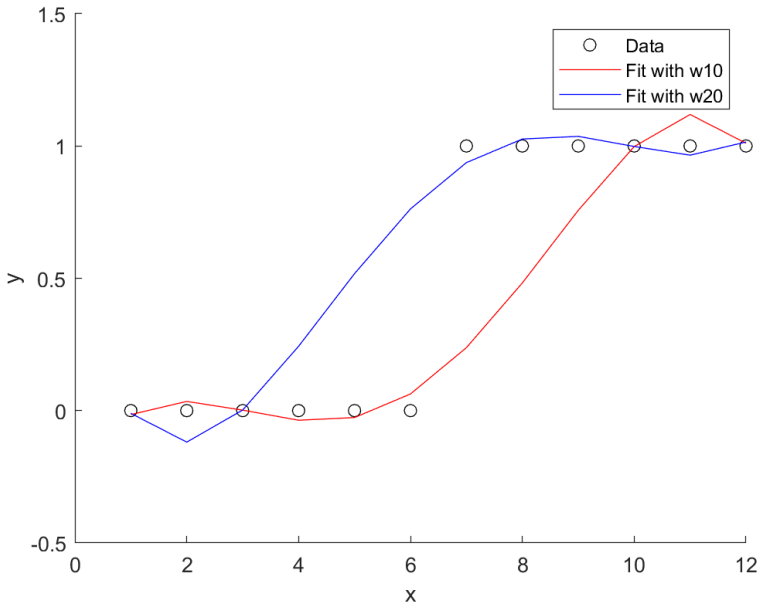


图 5: 加权最小二乘拟合结果 2

我们可以看出规格化前后得到的结果是相同的.

**c)** 通过观察所给图像，我们注意到拟合曲线在数据点 $x = 4, 5, 6, 7, 8, 9$ 六个点上的拟合精度较高，因此我们考虑赋予其更高的权值. 经过反复调试，我们考虑以下权值:

| $\omega_i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 权值 | 1 | 1 | 1 | 100 | 100 | 100 | 100 | 100 | 100 | 1 | 1 | 1 |

表 8: 绘制给定图像的权值

运行程序得到以下结果:

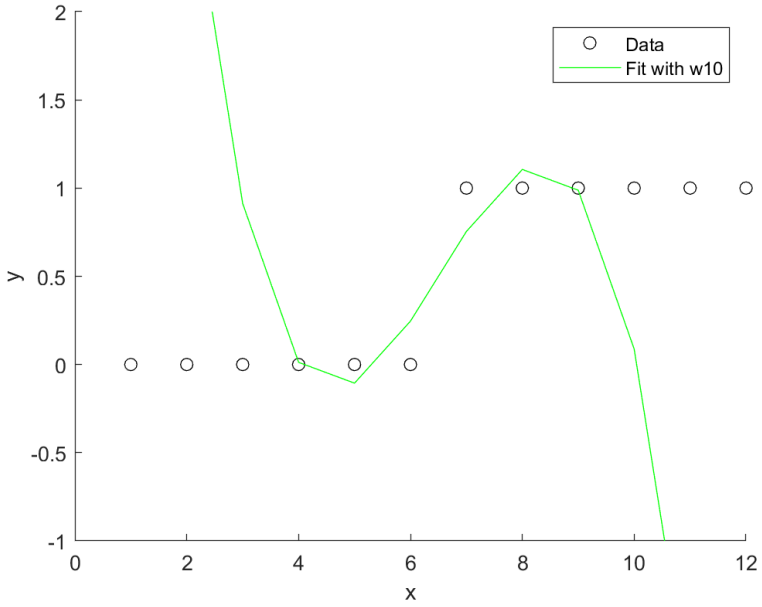


图 6: 绘制给定的图像

# 3 总结

这次作业我们研究了两个问题:

**快速离散傅里叶变换 (FFT)**: 介绍了一种使用傅里叶配点法求解微分方程的方法，以及如何使用离散傅里叶变换 (DFT) 计算傅里叶系数. 给出了一个具体的函数例子，并用 python 程序计算了误差，并绘制了误差图. 分析了使用正频率和负频率的区别，以及省略负频率的影响.

**带权的最小二乘多项式拟合**: 我们编写了一个函数用于计算带权的最小二乘多项式拟合，给出了函数的输入输出格式. 并使用了给定的数据和权值调用函数计算四次拟合多项式，并画图观察效果，解释权值的作用.

# 4 附录

## 4.1 FFT 相关代码

### 4.1.1 fft_a

```
    % 定义参数
n = 8; % 节点数
x = linspace(0, 2*pi, n+1); % 在 [0, 2*pi] 范围内等距取点
x = x(1:n); % 删除最后一个点以避免重复
f = sin(x) + exp(x); % 定义函数 f(x)

% 初始化 c_k 数组
c_k = zeros(1, n);

% 计算 c_k 系数
for k = 0:(n-1)
    sum = 0;
    for j = 1:n
        sum = sum + f(j) * exp(-1i * 2 * pi * k * (j-1) / n);
    end
    c_k(k+1) = sum / n;
end

% 输出 c_k 系数
disp('c_k coefficients:');
disp(c_k);

% 使用 fft 进行验证
Y = fft(f);

% 输出 fft 的结果进行对比
disp('FFT result (adjusted):');
disp(Y / n); % 调整 fft 的结果以匹配 c_k 的定义

% 验证 ifft(fft(f)) = f
f_reconstructed = ifft(Y);
```

```
% 输出重构的 f 和原始的 f 进行比较
disp('Original f:');
disp(f);
disp('Reconstructed f:');
disp(f_reconstructed);
```

#### 4.1.2 fft_b

```
    % 定义参数
n = 10; % 节点数
x = linspace(0, 2*pi, n+1); % 在 [0, 2*pi] 范围内等距取点
x = x(1:n); % 删除最后一个点以避免重复
f = cos(2*x); % 定义函数 f(x)

% 执行 FFT
Y = fft(f);

% 计算 m1 和 m2
m1 = floor(n / 2);
m2 = floor((n - 1) / 2);

% 初始化 Tilde c_k 数组
Tilde_c_k = zeros(1, n);

% 填充 Tilde c_k 数组
for k = -m1:m2
    if k >= 0
        Tilde_c_k(k + 1) = Y(k + 1) / n; % 当 0<=k<=m2 时
    else
        Tilde_c_k(k + n + 1) = Y(k + n + 1) / n; % 当-m1<=k<0 时
    end
end

% 输出 Tilde c_k
```

```
disp('Tilde c_k coefficients:');
disp(Tilde_c_k);
```

#### 4.1.3 fft_c

```
    import numpy as np
import matplotlib.pyplot as plt

def compute_fourier_coefficients(f, m1, m2, n):
    """ 计算傅立叶系数"""
    x = np.linspace(0, 2*np.pi, n, endpoint=False)
    f_values = f(x)
    f_coeffs = np.zeros(m2 - m1 + 1, dtype=complex)
    for k in range(m1, m2 + 1):
        f_coeffs[k - m1] = np.sum(f_values * np.exp(-1j * k * x)) / n
    return f_coeffs

def fourier_series(coeffs, m1, x):
    """ 傅立叶级数"""
    sum = np.zeros_like(x, dtype=complex)
    for k, coeff in enumerate(coeffs, start=m1):
        sum += coeff * np.exp(1j * k * x)
    return sum

def solve_differential_equation(f, n, m1, m2):
    """ 求解微分方程"""
    f_coeffs = compute_fourier_coefficients(f, m1, m2, n)
    u_coeffs = np.array([f_k / (k**2 + 1) if k != 0 else f_k for k, f_k in enumerate(f_coeffs
    return u_coeffs

def exact_solution(x):
    """ 精确解"""
    return np.sin(x) / (2 + np.cos(x))

def f_example(x):
    """ 算例的 f(x)"""
```

```
    return (2 * np.sin(x) / (2 + np.cos(x))
            - 2 * np.sin(x)**3 / (2 + np.cos(x))**3
            - 3 * np.sin(x) * np.cos(x) / (2 + np.cos(x))**2)

# 参数设置
m1, m2 = -10, 10

# 不同的 n 值
n_values = range(5, 22)
errors = []

for n in n_values:
    x = np.linspace(0, 2*np.pi, n, endpoint=False)
    u_coeffs = solve_differential_equation(f_example, n, m1, m2)
    u_approx = fourier_series(u_coeffs, m1, x)
    error = np.max(np.abs(u_approx - exact_solution(x)))
    errors.append(error)

# 绘制误差图
plt.plot(n_values, errors, marker='o')
plt.xlabel('n')
plt.ylabel('Maximum Error')
plt.title('Error vs. n in Differential Equation Solution')
plt.grid(True)
plt.show()

    import numpy as np
import matplotlib.pyplot as plt

def compute_positive_fourier_coefficients(f, n):
    """ 计算傅立叶系数 (仅正频率) """
    x = np.linspace(0, 2*np.pi, n, endpoint=False)
    f_values = f(x)
    f_coeffs = np.zeros(n, dtype=complex)
    for k in range(0, n):
        f_coeffs[k] = np.sum(f_values * np.exp(-1j * k * x)) / n
```

```
    return f_coeffs

def positive_fourier_series(coeffs, x):
    """ 傅立叶级数（仅正频率）"""
    sum = np.zeros_like(x, dtype=complex)
    for k, coeff in enumerate(coeffs):
        sum += coeff * np.exp(1j * k * x)
    return sum

def solve_modified_differential_equation(f, n):
    """ 求解修改后的微分方程"""
    f_coeffs = compute_positive_fourier_coefficients(f, n)
    u_coeffs = np.array([f_k / (k**2 + 1) if k != 0 else f_k for k, f_k in enumerate(f_coeffs)
    return u_coeffs

def exact_solution(x):
    """ 精确解"""
    return np.sin(x) / (2 + np.cos(x))

def f_example(x):
    """ 算例的 f(x)"""
    return (2 * np.sin(x) / (2 + np.cos(x))
            - 2 * np.sin(x)**3 / (2 + np.cos(x))**3
            - 3 * np.sin(x) * np.cos(x) / (2 + np.cos(x))**2)

# 不同的 n 值
n_values_modified = range(5, 22)
errors_modified = []

for n in n_values_modified:
    x = np.linspace(0, 2*np.pi, n, endpoint=False)
    u_coeffs = solve_modified_differential_equation(f_example, n)
    u_approx = positive_fourier_series(u_coeffs, x)
    error = np.max(np.abs(u_approx - exact_solution(x)))
    errors_modified.append(error)
```

```
# 绘制修改后的误差图
plt.plot(n_values_modified, errors_modified, marker='o', color='red')
plt.xlabel('n')
plt.ylabel('Maximum Error')
plt.title('Modified Fourier Collocation Method: Error vs. n')
plt.grid(True)
plt.show()
```

## 4.2 带权的最小二乘多项式相关代码

### 4.2.1 带权最小二乘多项式拟合函数

```
function [pp] = polyfit_weighted(x, y, m, w)

% 构造加权的设计矩阵
X = bsxfun(@power, x(:), m:-1:0);
W = diag(w);  % 构造权重矩阵
X = W * X;  % 加权设计矩阵
y = W * y(:);  % 加权目标向量

% 求解加权最小二乘问题
pp = X \ y;

end
```

### 4.2.2 主函数

```
function main()
% 数据点
x = 1:12;
y = [0 0 0 0 0 0 1 1 1 1 1 1];

% 权值
w1 = [8 8 8 8 8 8 1 1 1 1 1 1];
w2 = [1 1 1 1 1 1 8 8 8 8 8 8];
w3 = [1 1 1 1 1 1 1 1 1 1 1 1];
```

```
w10 = [4/27 4/27 4/27 4/27 4/27 4/27 1/54 1/54 1/54 1/54 1/54 1/54];
w20 = [1/54 1/54 1/54 1/54 1/54 1/54 4/27 4/27 4/27 4/27 4/27 4/27];
wx = [1 1 1 100 100 100 100 100 100 1 1 1];

% 拟合多项式次数
m = 4;

% 调用 polyfit_weighted 函数
pp1 = polyfit_weighted(x, y, m, w1);
pp2 = polyfit_weighted(x, y, m, w2);
pp3 = polyfit_weighted(x, y, m, w3);
pp10 = polyfit_weighted(x, y, m, w10);
pp20 = polyfit_weighted(x, y, m, w20);
ppx = polyfit_weighted(x, y, m, wx);

% 调用 Matlab 自带的 polyfit 函数
pp4 = polyfit(x,y,m);

% 验证函数的正确性
disp(pp3);
disp(pp4);
disp(pp1);
disp(pp10);

% 绘制结果
figure;
hold on;
plot(x, y, 'ko');  % 绘制数据点
plot(x, polyval(pp10, x), 'r-');  % 绘制拟合曲线
plot(x, polyval(pp20, x), 'b-');  % 绘制拟合曲线
ylim([-0.5,1.5]);  % 设置 y 轴的取值范围
hold off;
legend('Data', 'Fit with w10', 'Fit with w20');
xlabel('x');
ylabel('y');
```

```
    figure;
    hold on;
    plot(x, y, 'ko');  % 绘制数据点
    plot(x, polyval(ppx, x), 'g-');  % 绘制拟合曲线
    ylim([-1,2]);  % 设置 y 轴的取值范围
    hold off;
    legend('Data', 'Fit with w10', 'Fit with w20');
    xlabel('x');
    ylabel('y');

end
```