Lab 3:Nonlinear Solver

Ting Lin, 1700010644

May 8, 2020

Contents

In this report we introduce the FFT algorithm, and implements a filter based on FFT by C++.

1 FFT Algorithm

The FFT algorithm is developed as a fast solver of discrete fourier transform: We set $c = \hat{a}$ if

$$c_k = \sum_{j=0}^{N-1} a_j e^{-2\pi ijk/N}.$$

By inverse formula or direct calculation we can obtain the inverse DFT

$$a_k = \sum_{j=0}^{N-1} c_j e^{-2\pi ijk/N}.$$

For simplicity we only consider the case of $N = 2^m$.

The basic idea is from Danielson–Lanczos algorithm. Set

$$\begin{aligned} P(x) =& a_0 + a_1 x + \cdots a_N x^{N-1} \\ =& P_e(x^2) + xP_o(x^2) \end{aligned} \tag{1}$$

Define $\omega_k = \exp(-2\pi i/k)$, when $j = 0, \cdots, \frac{N}{2} - 1$, we have

$$c_j = P_(\omega_N^{2j}) + \omega_N^j P_o(\omega_N^{2j})$$

$$c_{N/2+j} = P_{(}\omega_N^{2j+N}) + \omega_N^{j+N/2} P_o(\omega_N^{2j+N})$$

Notice that

$$\omega_N^{2j} = \omega_N^{2j+N} = \omega_{N/2}^{j}, \omega_N^{N/2+j} = -\omega_N^{j},$$

we have

$$c_j = v_j + \omega_N^j u_j, c_{j+N/2} = v_j - \omega_N^j u_j$$

where $v_j = P_e(\omega_{N/2}^j), u_j = P_o(\omega_{N/2}^j)$

From this we define the D-L algorithm recursively

Algorithm 1 $c = \mathbf{FFT}(a))$

if $N := \text{size}(a) == 1$ **then**
    return a
end if
Compute $v_j = \mathbf{FFT}(a(0:2:N-1)), u_j = \mathbf{FFT}(a(1:2:N))$
$e = \exp(-2\pi i/N), w = 1$
for $j = 0 : N/2 - 1$ **do**
    $c_j = v_j + w u_j$
    $c_{j+N/2} = v_j - w u_j$
    $w = w \cdot e$
end for

We only need replace $w = e = \exp(-2\pi i/N)$ by $w = e = \exp(2\pi i/N)$ to obtain the IFFT algorithm.

For practical concerning, we adopt a 2-stage FFT: re-ordering and assembling.

The main trick of re-ordering is computing the bit-reverse of an index.

Algorithm 2 REORDER(a)

$j = 0$ (denote the bit-reverse result), $N = \text{size}(a)$
for $i = 0 : N$ **do**
    if $i < j$ **then**
        $\text{swap}(a_i, a_j)$
    end if
    $l = k >> 1, j = j \wedge l$
    while $j < l$ **do**
        $l = l >> 1$
        $j = j \wedge l$
    end while
end for

Here $\wedge, >>$ is bit operation from C. After re-ordering, assembling step is quite easy.

Algorithm 3 ASSEMBING(a)

Copy a to c.
$m = 2, n = \text{size}(a)$
while $m \leq n$ **do**
  $e = \exp(-2\pi i/N)$ {//$e = \exp(2\pi i/N)$ for ifft case}
  for $k = 0 : m : n - m$ **do**
    $w = 1$
    for $j = 0 : m/2 - 1$ **do**
      $t = w * a_{k+j+m/2}, u = a_{k+j}$
      $a_{k+j} = u + t, a_{k+j+m/2} = u - t$
    end for
  end for
  $m = 2m$
end while
{//$a_i = a_i/n$ for ifft case}

After this, we obtain the whole (inplace) FFT.

Algorithm 4 FFT*(a)

RE-ORDERING(a)
ASSEMBLING(a)

By considering comments in assembling, we obtain the IFFT* algorithm.

2 Numerical Result and Discussions

2.1 Filtering

We use FFT to implement the filtering algorithm. Suppose a is a N-array. For given threshold m, we set $c_i = (\hat{a})_i$ if $|i| \geq M$ otherwise $c_i = 0$. Then we denote $\mathcal{F}_m a = \check{c}$.

Mathematically, the filtering operator preserves all the low-frequency information and exclude the high-frequency one, this makes the function less oscillatory when function is smooth enough.

2.2 Numerical Result

We show an non-trivial example when the function is not continuous. Set $f(t) = \exp(-t^2/10)(sin(2t) + 2cos(4t) + 0.4sin(t)sin(50t))$, and we choose $f(2\pi k/256)(k = 0, 1, \cdots, 255)$ to discretize it . We plot for $m = 2, 3, 4, 5, 6, 10, 20, 30, 40, 50$. Notice that the gibbs phenomenon is easy to see. The gibbs phenomenon is caused by the jump in point 0. Furthermore, the approximation result has significant improvement when m increases from 20 to 30. Hence it will be better to a sparse frequency approximation then just filtering, and the behavior will be better.

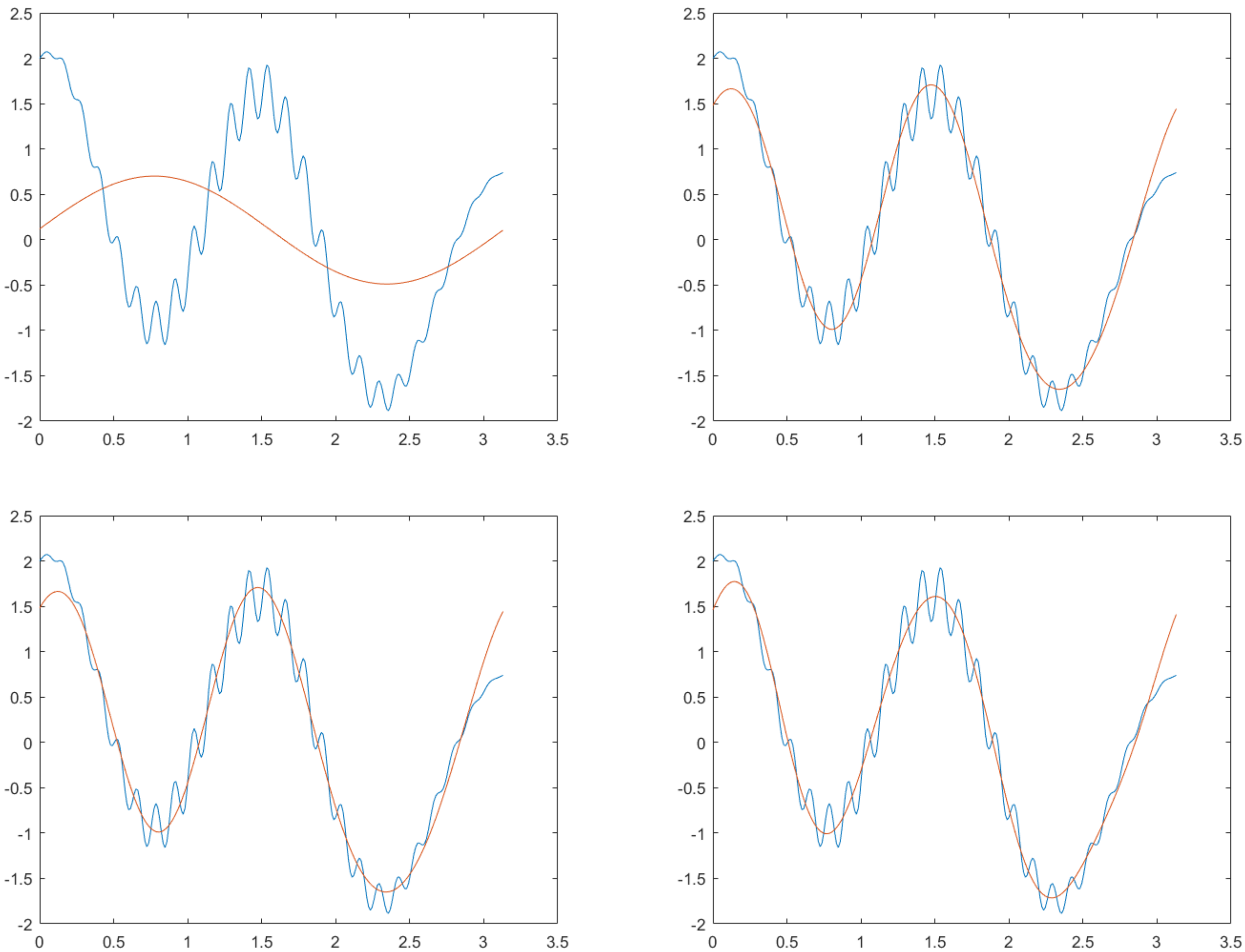

Figure 1: m = 2,3,4,5

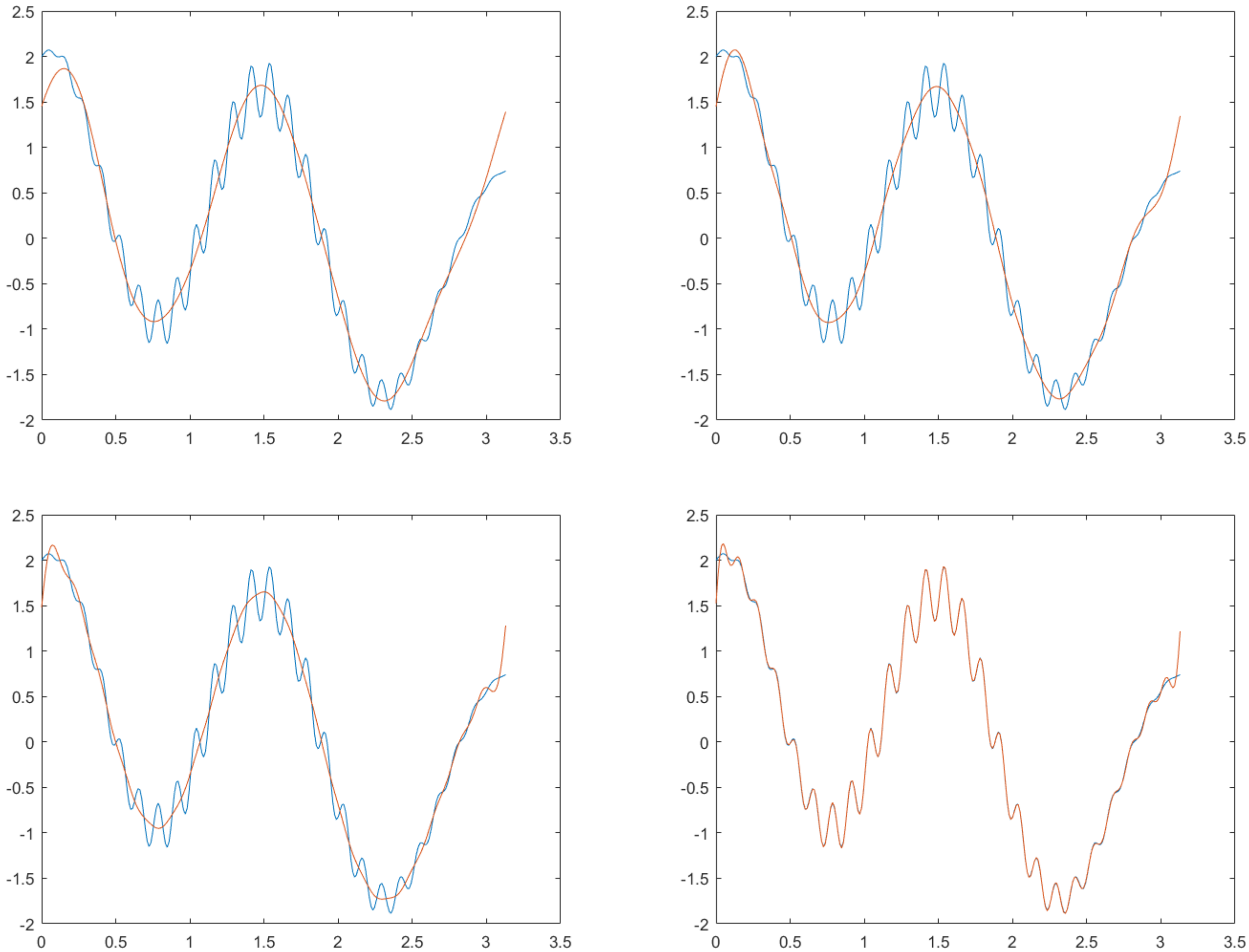

Figure 2: m = 6,10,20,30

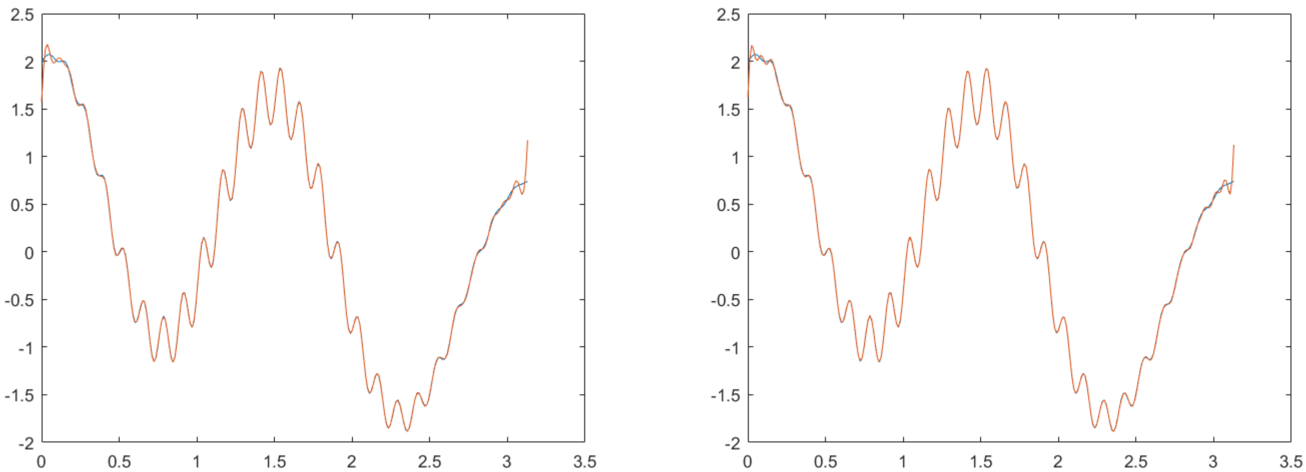

Figure 3: m = 40,50