Numerical Recipes in Physics Report

G.C Liu

August 1, 2021

"Study hard what interests you the most in the most undisciplined, irreverent and original manner possible."

— Richard Feynmann

Contents

§1 Lecture 1: Introduction

[1]

§1.1 Objectives

- The student understands the numerical methods in the solution of problems in physics and astronomy
- The student becomes familiar of the strength and limitation of different numerical methods
- The student implements the numerical methods in real problems via computer
- programing

§1.2 Roundoff error

It is most problematic when numbers of very different size are added or subtracted:

Example 1.1

in single-precision

$1.0 + 0.001 = 1.0010000$

$1.0 + 0.000000001 = 1.0000000$

Solutions to underflow and overflow errors:

- Use double precision
- Rescale or use a logarithmic scale
- Choose proper order of operations

Example 1.2

$10^{-30} \times 10^{-30}/10^{-30} = 0,\ 10^{-30}/10^{-30} \times 10^{-30} = 10^{-30}$

§2 Lecture 2:Root Finding

§2.1 Bisection Method

INPUT:$[a, b], f(x)$ OUTPUT: Approximate Root : p

Example 2.1

Show that $f(x) = x^3 + 4x^2 - 10$ has a root in $[1, 2]$, and use the Bisection method to determine an approximation to the root that is accurate to at least within 10^{-8}.

```
#DEFINE Function
def f(x):
    f = x**3+4*x**2-10
    return f
#Bisection Method
def Bisection(a, b ,accurate):
    p = a+(b-a)/2
    if  abs(f(p)) < accurate:
        print(p)
    else:
        if f(a)*f(p) > 0:
            a = p
            return Bisection(a, b)
        else:
            b = p
            return Bisection(a, b)

Bisection(1,2,1e-8)
```

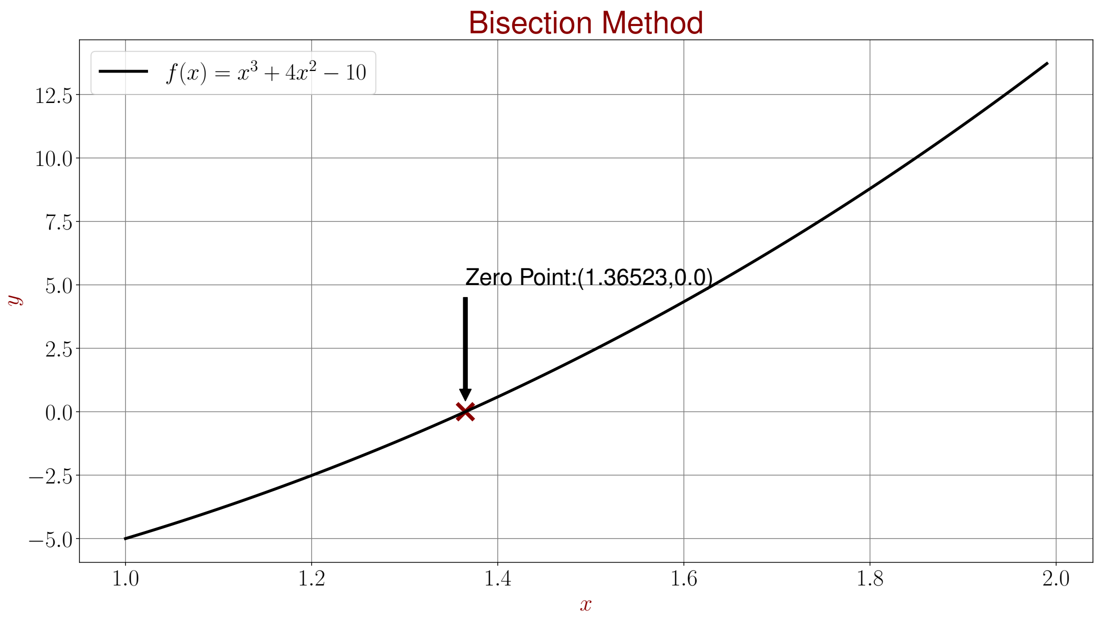


Figure 1: Bisection Method

```
OUTPUT : 1.3652300136116383
```

§2.2 Fixed-point Iteration Method

- fixed point for a function is a number at which the value of the function does not change:
- The number p is a fixed point for a given function $g(x)$ if $g(x) = x$

Given a root-finding problem$f(p) = 0$, we can define functions $g(x)$ with a fixed point at p in a number of ways

- (i) if $g \in C[a, b]$ and $g(x) \in [a, b]$ for all x in $[a, b]$,then g has at least one fixed point at $[a, b]$
- (ii) if in addition $g'(x)$ exists on (a, b) and a positive constant $k < 1$ exists with: $|g'(x)| \leq k$ for all $x \in (a, b)$

Remark 2.2. Step :

Choose a initial approximation p_0 and generate the list p_n
Recurrence relationship is $p_n = g(p_{n-1})$[function iteration]

Example 2.3

Show that $f(x) = x^3 + 4x^2 - 10$has a root in [1, 2], and use the Iteration Method to determine an approximation to the root that is accurate to at least within 10^{-8}.

```python
#DEFINE Function
def g(x):
    g = 0.5*(10-x**3)**0.5
    return g
def f(x):
    f = x**3 + 4*x**2-10
    return f
def Iteration(x,accurate):
    N= 0
    while True:
        x = g(x)
        N = N+1
        if abs(f(x))< accurate:
            break
        else:
            pass
    return x,N#Number of iterations

Iteration(1.5,1e-8)
```

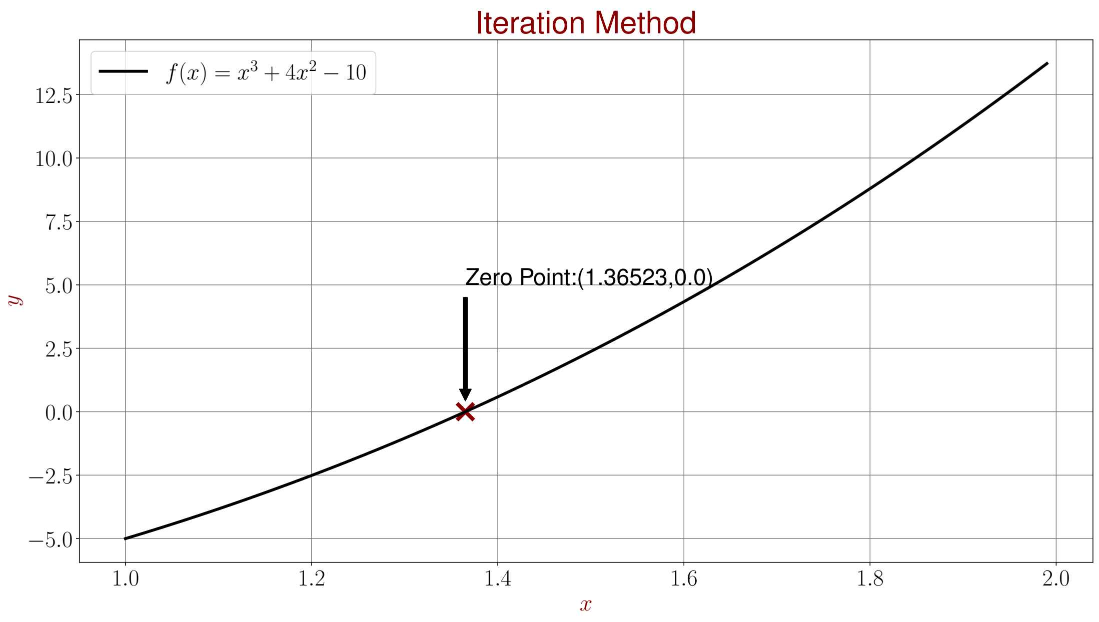


Figure 2: Iteration Method

```
OUTPUT : 1.3652300128759014 and the number of iterations is 29
```

Remark 2.4. *

1.The initial value of x is $(a + b)/2(a$ and b are the value ranges)

2.When we construct the iteration function we always use the function $g(x) = x - f(x)$ because that will contribute to the result is divergence so we use $f(x) = 0$ to get the relationship

§2.3 Newton's Method

Assuming that since $|p - p_0|$ is small and the term involving $(p - p_0)^2$ is much smaller so

$$f(p_0) + (p - p_0)f'(p_0) \approx 0 \tag{1}$$

We get the iterative relationship

$$p_n = p_{n-1} - \frac{f(p_{n-1})}{f'(p_{n-1})} \tag{2}$$

Example 2.5

Show that $f(x) = x^3 + 4x^2 - 10$has a root in [1, 2], and use the Newton's Method to determine an approximation to the root that is accurate to at least within 10^{-8}.

```
#DEFINE Function
def f(x):
    f = x**3 + 4*x**2-10
    return f
#Newton's Method
def Newton(x,accurate):
    N= 0
    while True:
        x = x - f(x)/(3*x**2+8*x)
        N = N+1
        if abs(f(x)) < accurate:
            break
        else:
            pass
    return x,N #Number of iterations

Newton(1.5,1e-8)
```

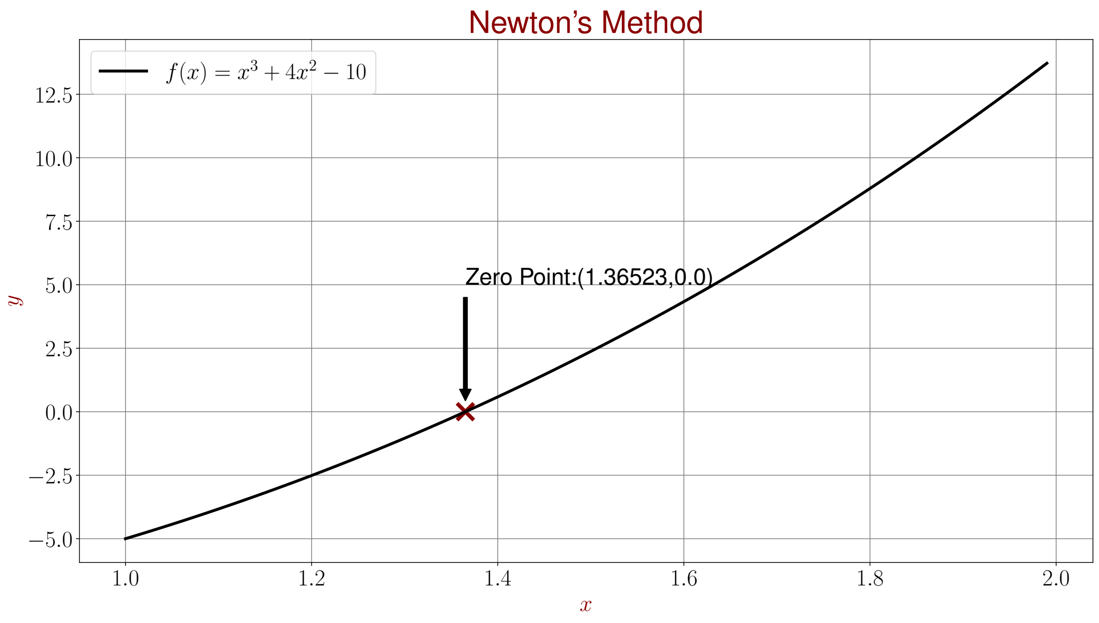


Figure 3: Newton's Method

```
OUTPUT : 1.3652300139161466 and the number of iterations is 3
```

Remark 2.6. The Taylor series derivation of Newton' s method at the beginning of the section points out the importance of an accurate initial approximation. so let $(p - p_0)^2$ almost to zero

§2.4 Secant Method

$$f'(p_{n-1}) = \lim_{x \to p_{n-1}} \frac{f(x) - f(p_{n-1})}{x - p_{n-1}} \tag{3}$$

If P_{n-2} is close to p_{n-1},then

$$p_n = p_{n-1} - \frac{f(p_{n-1})(p_{n-1} - p_{n-2})}{f(p_{n-1}) - f(p_{n-2})} \tag{4}$$

Put in the first formula to get

$$p_n = p_{n-1} - \frac{f(p_{n-1})(p_{n-1} - p_{n-2})}{f(p_{n-1}) - f(p_{n-2})} \tag{5}$$

Example 2.7

Show that $f(x) = x^3 + 4x^2 - 10$has a root in $[1, 2]$, and use the Secant Method to determine an approximation to the root that is accurate to at least within 10^{-8}.

```python
#DEFINE Function
def f(x):
    f = x**3 + 4*x**2-10
    return f
#Secant Method
def Secant(x0,x1,accurate):
    N= 0
    while True:
        x2 = x1 - f(x1)*(x1-x0)/(f(x1)-f(x0))
        x0 = x1
        x1 = x2
        N= N+1
        if abs(f(x2)) < accurate:
            break
        else:
            pass
    return x2,N #Number of iterations

Secant(1,2,1e-8)
```

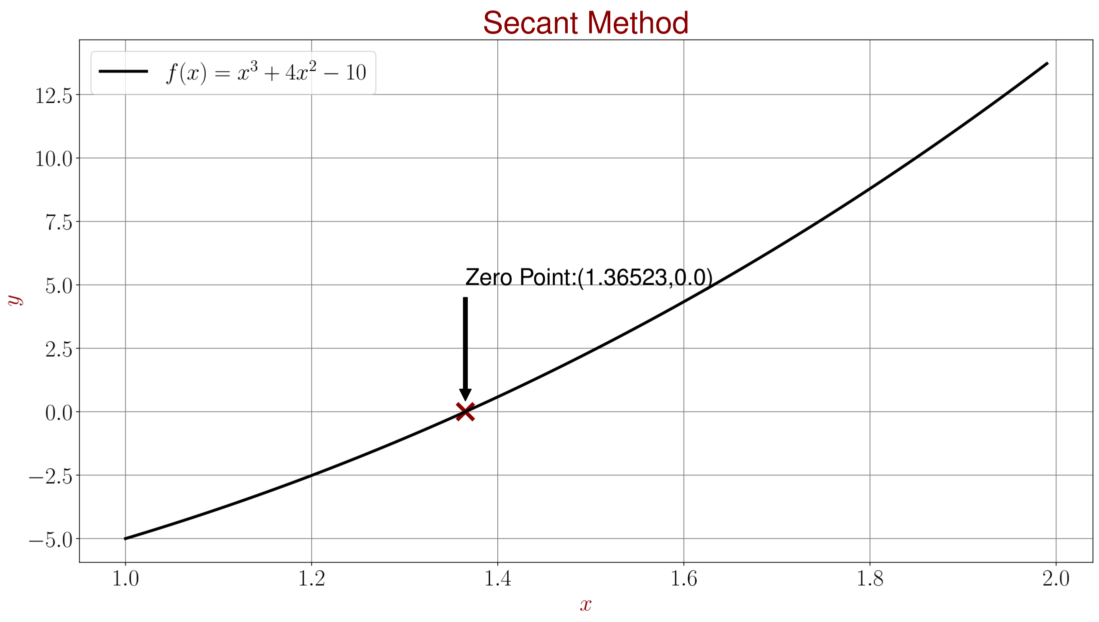


Figure 4: Secant Method

```
OUTPUT : 1.3652300134142061 and the number of iterations is 6
```

§2.5 Summary

we can calculate the running time with this four method(with the same accurate):

Example 2.8

Show that $f(x) = cos(x) - x$has a root in [0, 1], and use the Four Method to get an approximation to the root that is accurate to at least within 10^{-8} OUTPUT:Four Method Running Time

```python
from math import *
#cos,sin
#DEFINE Function
def f(x):
    f = cos(x) - x
    return f
def g(x):
    g = cos(x)
    return g
#Bisection Method
```

```
def Bisection(a,b,accurate):
    p = a+(b-a)/2
    if  abs(f(p)) < accurate:
        return p
    else:
        if f(a)*f(p) > 0:
            a = p
            return Bisection(a, b,accurate)
        else:
            b = p
            return Bisection(a, b,accurate)

def Iteration(x,accurate):
    N= 0
    while True:
        x = g(x)
        N = N+1
        if abs(f(x))< accurate:
            break
        else:
            pass
    return x,N#Number of iterations
#Newton's Method
def Newton(x,accurate):
    N= 0
    while True:
        x = x - f(x)/(3*x**2+8*x)
        N = N+1
        if abs(f(x)) < accurate:
            break
        else:
            pass
    return x,N#Number of iterations

#Secant Method
def Secant(x0,x1,accurate):
    N= 0
    while True:
        x2 = x1 - f(x1)*(x1-x0)/(f(x1)-f(x0))
        x0 = x1
        x1 = x2
        N= N+1
        if abs(f(x2)) < accurate:
            break
        else:
            pass
    return x2,N #Number of iterations

%timeit Bisection(0,1,1e-8)
%timeit Iteration(1.5,1e-8)
%timeit Newton(1.5,1e-8)
%timeit Secant(0,1,1e-8)
```

```
+
    Bisection:20.2 µs ± 1.83 µs per loop (10000 loops each)
    Iteration:21.7 µs ± 792 ns per loop ( 10000 loops each)
    Newton:2.8 µs ± 205 ns per loop (100000 loops each)
    Secant:5.22 µs ± 156 ns per loop (100000 loops each)
```

§3 Lecture 3:Numerical Integration

§3.1 Lagrange Interpolating Polynomials

Simple example: Two Points

Define the functions:

$$L_0(x) = \frac{x - x_1}{x_0 - x_1} \qquad L_1(x) = \frac{x - x_0}{x_1 - x_0} \tag{6}$$

so

$$P(x) = L_0(x)f(x_0) + L_1(x)f(x_1) \tag{7}$$

so How to generate the interpolation to n+1 points?

Construct a polynomial with the highest power of n to pass $n + 1$ points

Consider:

$$L_{n,k}(x) = \frac{(x - x_0)(x - x_1)\cdots(x - x_{k-1})(x - x_{k+1})\cdots(x - x_n)}{(x_k - x_0)(x_k - x_1)\cdots(x_k - x_{k-1})(x_k - x_{k+1})\cdots(x_k - x_n)} = \prod_{i=0(i\neq k)}^{n} \frac{x - x_i}{x_k - x_i} \tag{8}$$

so

$$P(x) = L_{n,0}f(x_0) + L_{n,1}f(x_1) + \cdots L_{n,n}f(x_n) \tag{9}$$

§3.2 Rectangular Approximation

$$\int_a^b f(x)dx = \lim_{max\Delta x_i \to 0} \sum_{i=1}^{n} f(x_i)x_i = \lim_{n\to\infty} \frac{b-a}{n} \sum_{i=0}^{n} f(x_i) \tag{10}$$

```
#DEFINE Function
def f(x):
    f = 3*x**2
    return f

#Rectangular Approximation
def Rectangular(a,b,n):
    h = abs(b-a)/n
    s = 0
    for i in range(int(n)):
        s+= f(i*h)*h
    return s

Rectangular(0,1,1e5)
```

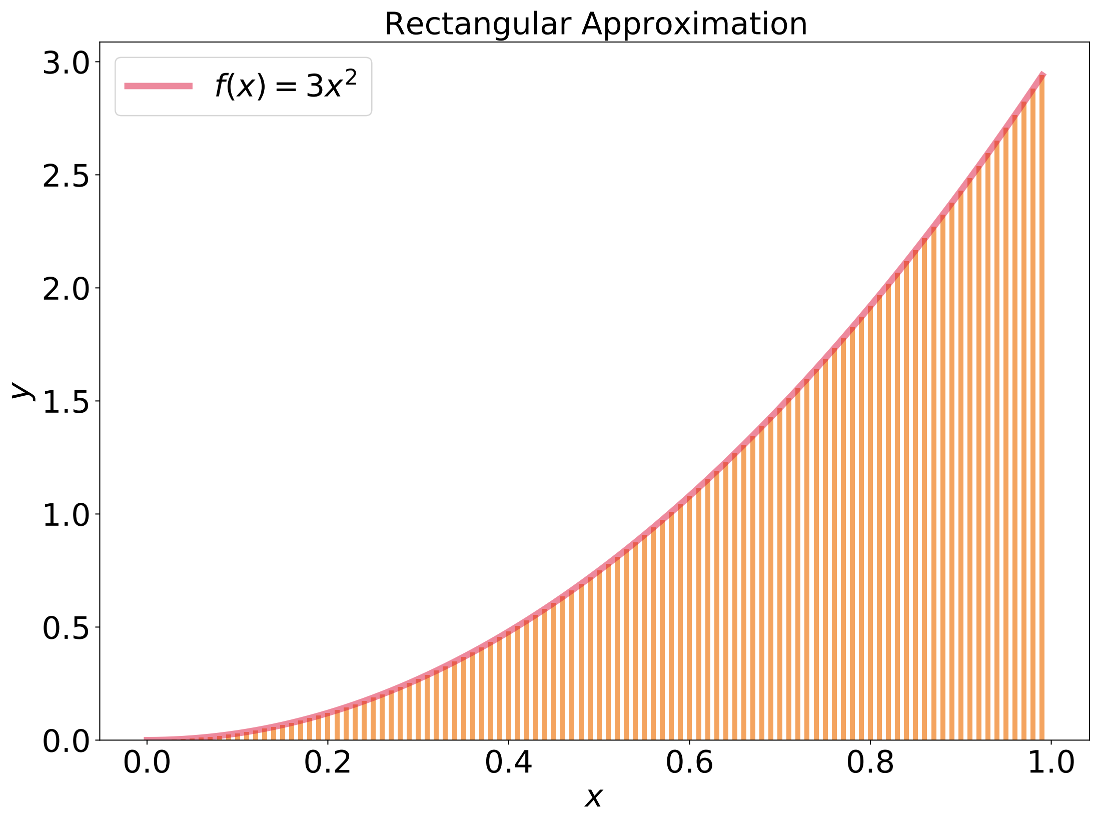


Figure 5: Rectangular Approximation

```
OUTPUT :0.9999850000499966
```

§3.3 Trapezoidal Rule

$$\int_a^b f(x)dx \approx \frac{h}{2}\sum_{k=1}^{n}(f(x_{k+1})+f(x_k)) = \frac{b-a}{2N}(f(x_1)+2f(x_2)+2f(x_3)\cdots 2f(x_{N-1})+f(x_N)) \tag{11}$$

§3.4 Simpson’s Rule

$$\begin{aligned}\int_a^b f(x)dx &\approx \frac{h}{3}\sum_{j=1}^{n/2}[f(x_{2j-2})+4f(x_{2j-1})+f(x_{2j})] \\ &= \frac{h}{3}[f(x_0)+4f(x_1)+2f(x_2)+4f(x_3)+2f(x_4)+\cdots 4f(x_{n-1})+f(x_n)]\end{aligned} \tag{12}$$

How to derive them? For Polynomial

$$P_n(x) = \sum_{i=0}^{n} f(x_i)L_i(x) \tag{13}$$

so

$$\int_a^b f(x)dx = \sum_{i=0}^{n} \int_a^b f(x_i)L_i(x)dx = \sum_{i=0}^{n} a_i f(x_i) \tag{14}$$

where $a_i = \int_a^b L_i(x)dx$ So for Trapezoidal Rule : $n = 1$

$$P_1(x) = \frac{x - x_1}{x_0 - x_1} f(x_0) + \frac{x - x_0}{x_1 - x_0} f(x_1) \tag{15}$$

so

$$\int_a^b P_1(x)dx = \int_{x_0}^{x_1} P_1(x)dx = \frac{x_1 - x_0}{2}[f(x_0) + f(x_1)] \tag{16}$$

so for Simpson' s Rule : $n = 2$

$$P_2(x) = \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} f(x_0) + \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)} f(x_1) + \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)} f(x_2) \tag{17}$$

so:

$$\int_a^b P_2(x)dx = \int_{x_0}^{x_2} P_2(x)dx = \frac{h}{3}[f(x_0) + 4f(x_1) + f(x_2)] \tag{18}$$

They are collectively referred to as Newton-Cotes formula

(n+1)-point closed Newton-Cotes formula

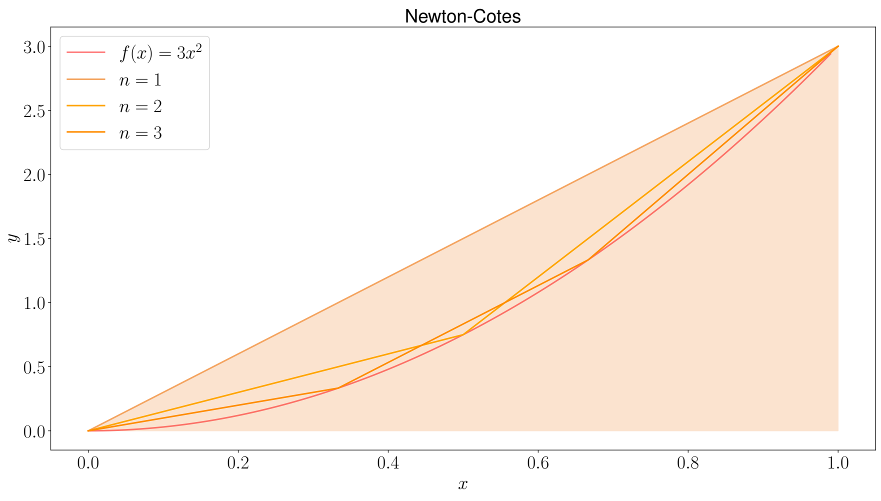
$$\int_a^b f(x)dx \approx \sum_{i=0}^{n} a_i f(x_i) \tag{19}$$

where

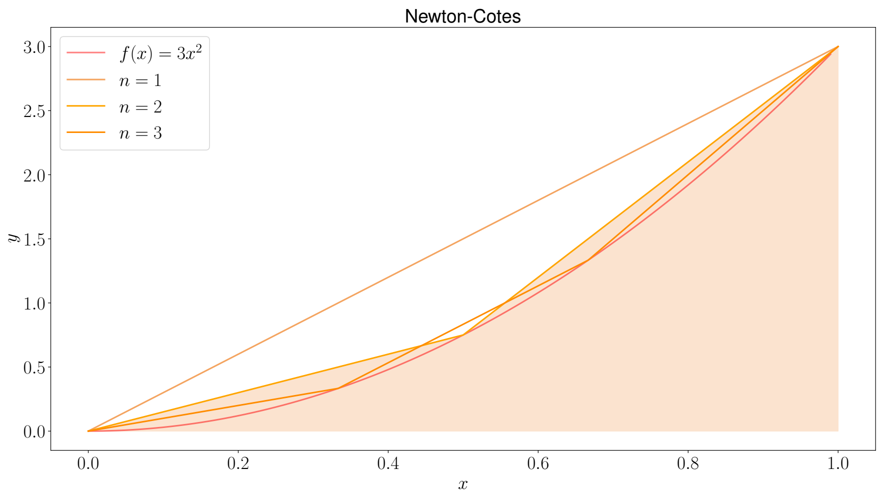
$$a_i = \int_{x_0}^{x_n} L_i(x) = \int_{x_0}^{x_n} \prod_{j=0, j\neq i}^{n} \frac{x - x_j}{x_i - x_j} dx \tag{20}$$

Integral converted to polynomial

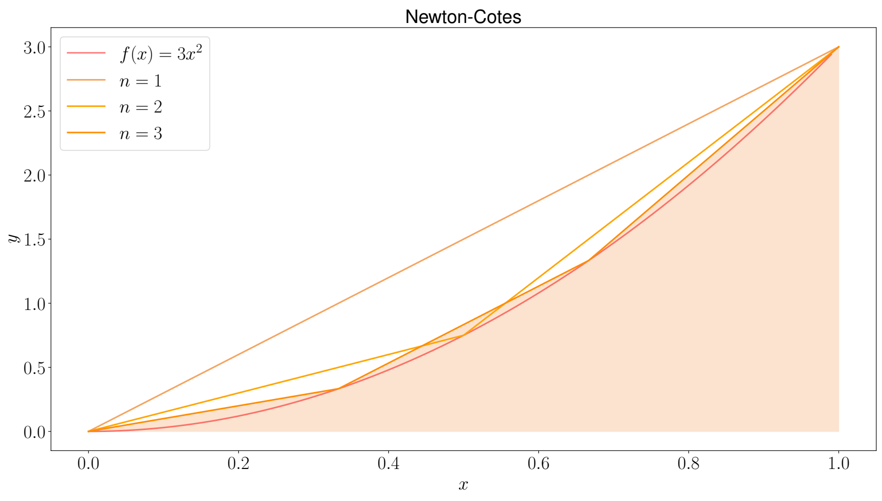
```python
from sympy import *
#Import a package to calculate integral
#or you can use your own code to calculate the integral
Pi = 3.14
#DEFINE Lagrange Interpolating Polynomials
def L(n, k, a, b):
    x = symbols('x')
    h = (b-a)/n
    prob = 1
    for i in range(n+1):
        if i == k:
            pass
        else:
            prob = prob*(x - (a+i*h))/(k*h-i*h)
    return integrate(prob,(x, a, b))
#DEFINE Function
def f(x):
    g = 3*x**2
    return g
#DEFINE Polynomial
def P(n,a,b):
    h = (b-a)/n
    p = 0
    for i in range(n+1):
        p = p + f(a+i*h)*L(n,i,a,b)
    return p

P(1, 0, 1)#Trapezoidal Rule
P(2, 0, 1)#Simpson' s Rule
```



(a) Trapezoidal Rule



(b) Simpson' s Rule



(c) Simpson' s 3/8 Rule

Figure 6: Newton-Cotes

§3.5 Composite Numerical Integration

The kernel idea is to use trapezoids instead of rectangles

Composite Trapezodial Rule

$$\int_a^b f(x)dx = \frac{h}{2}[f(a) + 2\sum_{j=1}^{n-1} f(x_j) + f(b)] - \frac{b-a}{12}h^2 f''(\mu) \tag{21}$$

Composite Simpson' s Rule

$$\int_a^b f(x)dx = \frac{h}{3}[f(a) + 2\sum_{j=1}^{n/2-1} f(x_{2j}) + 4\sum_{j=1}^{n/2} f(x_{2j-1}) + f(b)] - \frac{b-a}{180}h^4 f^{(4)}(\mu) \tag{22}$$

Example 3.1

Determine the value of h that will ensure an approximation error of less than 0.00002 when approxmating $\int_0^\pi sin(x)dx$ and employing

(a)Composite Trapezodial rule (b) Composite Simpon's rule

```python
from sympy import *
#Import a package to calculate integral
#or you can use your own code to calculate the integral
#DEFINE Lagrange Interpolating Polynomials
def L(n, k, a, b):
    x = symbols('x')
    h = (b-a)/n
    prob = 1
    for i in range(n+1):
        if i == k:
            pass
        else:
            prob = prob*(x - (a+i*h))/(k*h-i*h)
    return integrate(prob,(x, a, b))
#DEFINE Function
def f(x):
    g = sin(x)
    return g
#DEFINE Polynomial
def P(n,a,b):
    h = (b-a)/n
    p = 0
    for i in range(n+1):
        p = p + f(a+i*h)*L(n, i, a, b)
    return p
#n:fragment,m:number of iteration,a:Lower bound,b:Upper bound
def Composite(n,m,a,b):
    h = (b-a)/n
    c = 0
    for i in range(m):
        c = c + P(n, a+i*s, a+(i+1)*s)
    return c
Composite(1, 360, 0, pi)#Composite Trapezoidal Rule
Composite(2, 18, 0, Pi)#Composite Simpson' s Rule
```

```
OUTPUT:
    Composite Trapezoidal Rule:1.99998730759189
    Running Time :15.5 s ± 388 ms per loop ( 1 loop each)
    Composite Simpson' s Rule:2.00000064497207
    Running Time :9.47 s ± 1.79 s per loop ( 1 loop each)
```

§3.6 Monte Carlo Integration

This method is particularly useful for higher-dimensional integrals.

such as 2-dim

$$\iint_R f(x,y)dA = \int_a^b (\int_c^d f(x,y)dy)dx \tag{23}$$

Apply Trapezoidal rule

$$\int_c^d f(x,y)dy \approx \frac{k}{2}[f(x,c) + f(x,d) + 2f(x, \frac{c+d}{2})] \tag{24}$$

apply Trapezoidal rule again

we can get:

$$\iint_R f(x,y)dA \approx \frac{(b-a)(d-c)}{16}[f(a,c) + f(a,d) + f(b,c) + f(b,d) + 2[f(\frac{a+b}{2}, c) + f(\frac{a+b}{2}, d) + f(a, \frac{c+d}{2}) + f(b, \frac{c+d}{2})] + 4f(\frac{a+b}{2}, \frac{c+d}{2})] \tag{25}$$

Apply Simpson' s rule we can also get the almost same result

Example 3.2

Approxmating $\int_0^1 \int_0^1 xydxdy$ and employing trapezoidal rule

```python
from sympy import *
#Import a package to calculate integral
#or you can use your own code to calculate the integral
#DEFINE Lagrange Interpolating Polynomials
def L(n, k, a, b):
    x = symbols('x')
    h = (b-a)/n
    prob = 1
    for i in range(n+1):
        if i == k:
            pass
        else:
            prob = prob*(x - (a+i*h))/(k*h-i*h)
    return integrate(prob,(x, a, b))
#DEFINE Function
def f(x,y):
    g = x*y
    return g
#DEFINE Polynomial
y = symbols('y')
def Px(n,a,b):
    hx = abs(b-a)/n
    p = 0
    for i in range(n+1):
        p = p + f(a+i*hx,y)*L(n,i,a,b)
    return p
def Py(n,a,b,c,d):
    hy = abs(d-c)/n
    p = 0
    for i in range(n+1):
        p = p + Px(n,a,b).subs(y,c+i*hy)*L(n,i,c,d)
    return p
Py(1,0,1,0,1)
#we first calculate the integral of x and then calculate integral of y
```

§3.7 Numerical Differentiation

Three Universal Method:

$$\begin{aligned}
f'(x) &= \lim_{h\to 0}\frac{f(x+h)-f(x)}{h} \\
f'(x) &= \lim_{h\to 0}\frac{f(x+h)-f(x-h)}{2h} \\
f'(x) &= \lim_{h\to 0}\frac{-f(x+2h)+8f(x+h)-8f(x-h)+f(x-2h)}{12h}+\frac{h^4}{30}f^{(5)}(c)
\end{aligned} \tag{26}$$

from the above discusstion we can get:

$$f(x)=\sum_{k=0}^{n} f(x_k)L_k(x)+\frac{(x-x_0)\cdots(x-x_n)}{(n+1)!}f^{(n+1)}(\xi(x)) \tag{27}$$

so

$$f'(x)=\sum_{k=0}^{n} f(x_k)L_k'(x)+(\frac{(x-x_0)\cdots(x-x_n)}{(n+1)!}f^{(n+1)}(\xi(x)))' \tag{28}$$

and so we can get the approxmate (n+1)-point formula

$$f'(x_j)=\sum_{k=0}^{n} f(x_k)L_k'(x_j)+\frac{f^{(n+1)(\xi(x_j))}}{(n+1)!}\prod_{k=0(k\neq j)}^{n}(x_j-x_k) \tag{29}$$

Example 3.3

we use three-point formulas and consider their errors

$$L_0(x)=\frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} \tag{30}$$

so we get

$$L_0'(x)=\frac{2x-x_1-x_2}{(x_0-x_1)(x_0-x_1)} \tag{31}$$

so

$$f'(x_j)=f(x_0)[\frac{2x_j-x_1-x_2}{(x_0-x_1)(x_0-x_2)}]+f(x_1)[\frac{2x_j-x_0-x_2}{(x_1-x_0)(x_1-x_2)}]+f(x_2)[\frac{2x_j-x_0-x_1}{(x_2-x_0)(x_2-x_1)}]+\frac{1}{6}f^{(3)}(\xi_j)\prod_{k=0,k\neq j}^{2}(x_j-x_k) \tag{32}$$

so

$$f'(x_0)=\frac{1}{2h}[-3f(x_0)+4f(x_0+h)-f(x_0+2h)]+\frac{h^2}{3}f^{(3)}(\xi_0) \tag{33}$$

§4 Lecture 4:Monte Carlo Integration Further

$$F=\int_a^b f(x)dx \tag{34}$$

sample mean method

$$F_n = (b-a)\langle f\rangle = (b-a)\frac{1}{n}\sum f(x_i) \tag{35}$$

Run another n trial and get the the average of error
define

$$\sigma_m^2 = \langle M^2\rangle - \langle M\rangle^2 \tag{36}$$

and

$$\langle M\rangle = \frac{1}{m}\sum_{\alpha=1}^{m} M_\alpha \tag{37}$$

$$\langle M^2\rangle = \frac{1}{m}\sum_{\alpha=1}^{m} M_\alpha^2 \tag{38}$$

Consider m set of measurement and n trial
and index α to denote a particular value of measurement and i is the ith trial
So we get

$$M_\alpha = \frac{1}{n}\sum_{i=1}^{n} x_{\alpha,i} \tag{39}$$

and

$$\bar{M} = \frac{1}{m}\sum_{\alpha=1}^{m} M_\alpha = \frac{1}{mn}\sum_{\alpha=1}^{m}\sum_{i=1}^{n} x_{\alpha,i} \tag{40}$$

the difference between M_α and $\bar{M}$ define $e_\alpha = M_\alpha - \bar{M}$
so

$$\sigma_m^2 = \langle M^2\rangle - \langle M\rangle^2 \tag{41}$$

and

$$\frac{1}{m}\sum_{\alpha=1}^{m} e_\alpha^2 = \frac{1}{m}\sum(M_\alpha - \bar{M})^2 = \frac{1}{m}\sum(M_\alpha^2 - 2M_\alpha\bar{M} + \bar{M}^2) \tag{42}$$

and $\sum \bar{M} = \sum M_\alpha$
so

$$\sigma_m^2 =< M^2 > -\frac{1}{m}\bar{M}(\sum 2M_\alpha - \sum \bar{M}) =< M^2 > - < M >^2 = \frac{1}{m}\sum_{\alpha=1}^{m} e_\alpha^2 \tag{43}$$

The discrepancy of $d_{\alpha,i} = x_{\alpha,i} - \bar{M}$

$$\sigma^2 = \frac{1}{mn}\sum_{\alpha=1}^{m}\sum_{i=1}^{n} d_{\alpha,i}^2 \tag{44}$$

from the calculation
we can get $\sigma_m^2 = \frac{\sigma^2}{n}$
Error Analysis

Example 4.1

Use the hit or miss Monte Carlo method to estimate F_n, the integral of $f(x) = 4\sqrt{1-x^2}$in the interval $0 \leq x \geq 1$ as a function of n.Choose $a = 0, b = 1, h = 1$, and compute the mean value of the function $\sqrt{1-x^2}$ Multiply the estimate by 4 to determine F_n Calculate the difference between F_n, and the exact result of π This difference is a measure of the error associated with the Monte Carlo estimate. Make a log-log plot of the error as a function of n. What is the approximate functional dependence of the error on n for large n, for example, $n > 10^4$

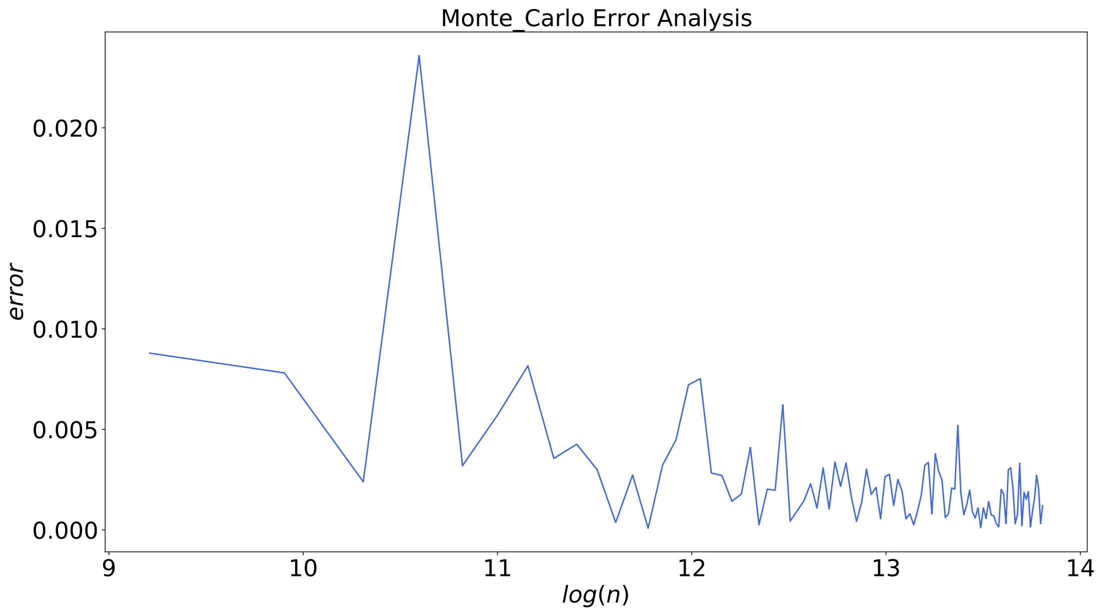


Figure 7: Monte Carlo Error Analysis

§5 Lecture 5:Fourier Transform

Most Importance Formula:

frequency to time and time to frequency

$$\begin{aligned} H(w) &= \int_{-\infty}^{\infty} h(t)e^{-iwt}dt \\ h(t) &= \frac{1}{2\pi}\int_{-\infty}^{\infty} H(w)e^{iwt}dw \end{aligned} \tag{45}$$

where $w \equiv 2\pi f$

so it can also rewritten as:

$$\begin{aligned} H(f) &= \int_{-\infty}^{\infty} h(t)e^{-2\pi ift}dt \\ h(t) &= \int_{-\infty}^{\infty} H(f)e^{2\pi ift}df \end{aligned} \tag{46}$$

and fourier transform has its own property:

$$\begin{aligned} &h(at) \Leftrightarrow \frac{1}{|a|}H(\frac{f}{a}) \\ &h(t-t_0) = H(f)e^{2\pi ift_0} \\ &h(t)e^{-2\pi if_0t} = H(f-f_0) \end{aligned} \tag{47}$$

Convolution and Correlation

Theorem 5.1

Convolution

$$g * h = \int_{-\infty}^{\infty} g(\tau)h(t-\tau)d\tau \tag{48}$$

Theorem 5.2

Correlation

$$\mathrm{Corr}(g,h) = g(t) \otimes f(t) = \int_{-\infty}^{\infty} g(\tau+t)h(\tau)d\tau \tag{49}$$

Remark 5.3.

$$F[f(t) * g(t)] = F_{f(t)}[w]F_{g(t)}[w] \tag{50}$$

Remark 5.4.

$$F[f(t) \otimes g(t)] = F^*_{f(t)}[w]F_{g(t)}[w] \tag{51}$$

Remark 5.5.

$$F^*_{f(t)}[w] = F^*_{f(t)}[-w] \tag{52}$$

Proof. **Remark 5.3.**

$$\begin{aligned}F(f(t)*g(t)) &= \int_{-\infty}^{\infty}(\int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau)e^{-iwt}dt\\ &= \int_{-\infty}^{\infty} f(\tau)(\int_{-\infty}^{\infty} g(t-\tau)e^{-iwt}dt)d\tau\\ &= F[f(\tau)]F[g(t-\tau)]\\ &= h_1(w)h_2(w))\end{aligned} \tag{53}$$

□

§5.1 Discrete Fourier Transform

$h(t)$ is recorded spaced intervals in time

$h_n = h(n\Delta)[n = \cdots -3, -2, -1, 0, 1, 2, 3\cdots]$

and $h_k \equiv h(t_k)[t_k \equiv k\Delta]$

so

$$H(f_n) = \int_{-\infty}^{\infty} h(t)e^{2\pi i f_n t}dt \approx \sum_{k=0}^{N-1} h_k e^{2\pi i f_n t_k}\Delta \tag{54}$$

where $h_k =\equiv h(t_k), t_k \equiv k\Delta, k = 0, 1, 2, 3, 4\cdots N-1$

per cycle has two sampled value

where $f_c \equiv \frac{1}{2\Delta}$

Theorem 5.6

Sampling theorem

- $h(t)$ is continuous
- sampled limited to frequencies smaller in magnitude than f_c

Example 5.7

Use the Discrete Fourier Transform to get the frequency of $sin(t)$

```
import numpy as np


#Define Function
def f(x):
    return np.sin(2*np.pi*x)



#Get the Sampled Frequency:
# sampling rate:sr
# sampling time:st
def gsf(sr,st):
    ts = 1.0/sr
    t = np.arange(0,st,ts)
    y = f(t)
    return y

# Method 1
def DFT1(x):
    sr = len(x)
    w = 0
    wl = []
    result = []
    # DFT:
    for j in range(sr):
        for i in range(sr):
            w = w + x[i]*(cos(2*pi*(j)*i/sr) - I*sin(2*pi*(j)*i/sr))
        wl.append(w)
        w = 0
    for i in wl:
        result.append(abs(i**2))
    return result
# Method 2[Matrix Method]
def DFT2(x):
    x = np.array(x)
    N = x.shape[0]
    n = np.arange(N)
    k = n.reshape((N, 1))
    M = np.exp(-2j * np.pi * k * n / N)
    result = []
    for i in np.dot(M, x).reshape(N, 1):
        result.append(abs(i**2))
    return result
```

Remark 5.8. Matrix method is a good method we can compare it with the normal method:

Normal method:5.46 ms ± 195 µs per loop (100 loops each)

Matrix method:33.9 µs ± 1.03 µs per loop (10000 loops each)

That is because the normal method use the loop

Matrix Method:

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-i2\pi kn/N} \tag{55}$$

It is equal to :

$$\underbrace{X_k}_{ColumnVector} = \underbrace{e^{-i2\pi kn/N}}_{Matrix} \underbrace{x_n}_{ColumnVector} \tag{56}$$

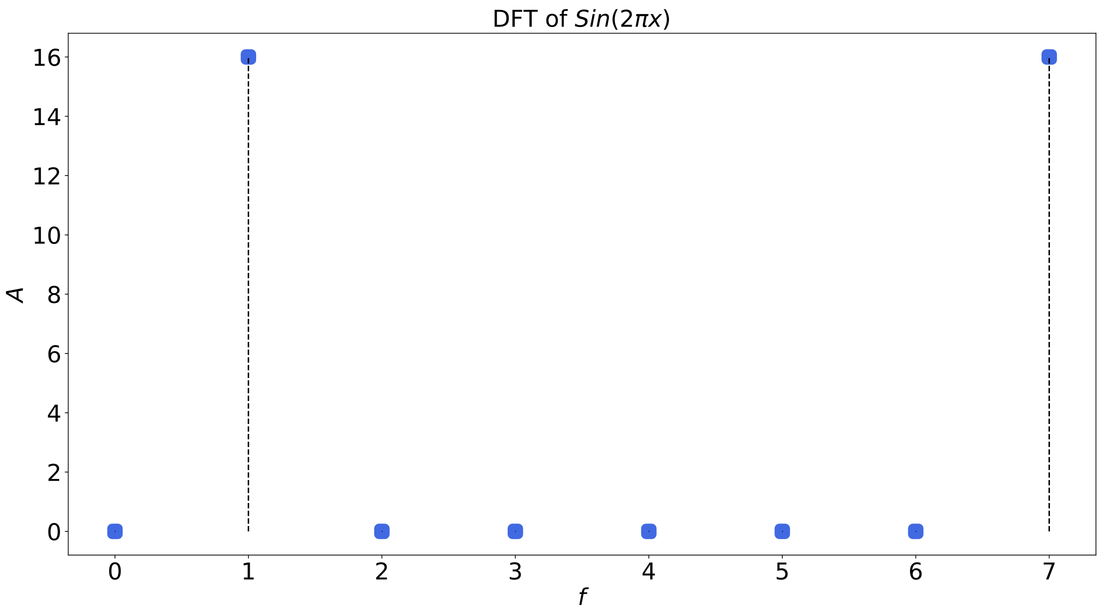


Figure 8: DFT-sin($2\pi x$)

Remark 5.9. The main issue with the above DFT implementation is that it is not efficient if we have a signal with many data points. It may take a long time to compute the DFT if the signal is large.— Python Numerical methods.berkeley.edu

§5.2 Fast Fourier Transform

Definition 5.10. The discrete Fourier transform can, in fact, be computed in $O(Nlog_2N)$operations with an algorithm called the fast Fourier transform, or FFT

Remark 5.11. Since we know there are symmetries in the DFT[See in Figure 8], we can consider to use it reduce the computation, because if we need to calculate both Xk and Xk+N, we only need to do this once. This is exactly the idea behind the FFT.

so from the DFT we can transform it become:

$$\begin{aligned} X_k &= \sum_{m=0}^{N/2-1} x_{2m} \cdot e^{-i2\pi k(2m)/N} + \sum_{m=0}^{N/2-1} x_{2m+1} \cdot e^{-i2\pi k(2m+1)/N} \\ &= \sum_{m=0}^{N/2-1} x_{2m} \cdot e^{-i2\pi k(m)/N/2} + e^{-i2\pi k/N} \sum_{m=0}^{N/2-1} x_{2m+1} \cdot e^{-i2\pi k(2m)/N/2} \end{aligned} \tag{57}$$

so we can use the DFT to calculate the two apartly.

$$\begin{aligned} &\sum_{m=0}^{N/2-1} x_{2m} \cdot e^{-i2\pi k(m)/N/2} = \mathrm{DFT}(x[even]) \\ &e^{-i2\pi k/N} \sum_{m=0}^{N/2-1} x_{2m+1} \cdot e^{-i2\pi k(2m)/N/2} = e^{-i2\pi k/N}\mathrm{DFT}(x[odd]) \end{aligned} \tag{58}$$

This is how FFT works using this recursive approach.

```
import numpy as np
#Define DFT
def DFT(x):
    x = np.asarray(x, dtype=complex)
    N = x.shape[0]
    n = np.arange(N)
    k = n.reshape((N, 1))
    M = np.exp(-2j * np.pi * k * n / N)
    return np.dot(M, x)
#Define FFT
def FFT(x):
    x = np.array(x, dtype=complex)
    N = x.shape[0]
    if N % 2 > 0:
        raise ValueError("size of x must be a power of 2")
    elif N <= 8:
        return DFT(x)
    else:
        X_even = FFT(x[::2])
        X_odd = FFT(x[1::2])
        factor = np.exp(-2j*np.pi*np.arange(N) / N)
        X = np.concatenate(
            [X_even+factor[:int(N/2)]*X_odd,
            X_even+factor[int(N/2):]*X_odd])
    return X
```

Compare the time with DFT and FFT we can find that:

```
sr = 128
timeit DFT(gsf(sr,1))
timeit FFT(gsf(sr,1))
```

```
OUTPUT:
    1.45 ms ± 286 µs per loop (1 loop each)
    410 µs ± 5.72 µs per loop (1000 loops each)
```

§6 Lecture 6:Partial Differential Equation

§6.1 Partial Differential Equation Solving

Example 6.1

$$\frac{\partial u(x,t)}{\partial t} = a^2\frac{\partial^2 u(x,t)}{\partial x^2} + f(x,t) \tag{59}$$

with the boundary condition:$u(x,0) = f(x), u(0,t) = g_1(t), u(l,t) = g_2(t)$

Discretization:

$$\frac{\partial^2 u(x,t)}{\partial x^2}_{|i,k} = \frac{u_{i-1,k} - 2u_{i,k} + u_{i,k+1}}{h^2} \tag{60}$$

$$\frac{\partial u(x,t)}{\partial t}_{|i,k} = \frac{u_{i,k+1} - u_{i,k}}{\tau} \tag{61}$$

so we can get the Recurrence Relationship

$$u_{i,k+1} = a^2\Delta t(\frac{u_{i-1,k} - 2u_{i,k} + u_{i,k+1}}{\Delta x^2}) + \Delta t f(x,t) + u_{i,k} \tag{62}$$

and so

$$\vec{u}_{k+1} = \vec{u}_k + (\frac{a^2\Delta t}{\Delta x^2})A\vec{u_k} + \Delta t\vec{f_k} \tag{63}$$

where $u_{i,0} = f(ih), u_{0,k} = g_1(k\tau), u_{N,k} = g_2(k\tau)$

and with boundary condition:

$u_{i,0} = f(ih) i = 1, 2, 3 \cdots N-1, N = \frac{l}{h}$

$u_{0,k} = g_1(k\tau), u_{N,k} = g_2(k\tau) k = 0, 1, 3 \cdots M, M = \frac{T}{\tau}$

Example 6.2

$$\frac{\partial u(x,t)}{\partial t} = \lambda\frac{\partial^2 u(x,t)}{\partial x^2} \tag{64}$$

where $\lambda \equiv \frac{\kappa}{c\rho}$

Define $\lambda = 1, l = 3, T = 1$

and boundary condition:$u(x,0) = 4x(3-x), u(0,t) = 0, u(3,t) = 0$

```
import numpy as np
h = 0.1#Define Grid step
N = 30#Define Number
dt = 0.0001#Define Time Grid step
M = 10000#Define Number
```

```
U = np.zeros([N+1, M+1])
#Define the Matrix A
S = -2*np.eye(N+1, dtype=int)+
np.eye(N+1, dtype=int,k=1)+np.eye(N+1, dtype=int,k=-1)

#Boundary Condition
for k in np.arange(M+1):
    U[0, k] = 0
    U[N, k] = 0
for i in np.arange(N+1):
    U[i, 0] = 4*h*i*(3-i*h)
#Matrix Operator
U = np.array(U)
for t in np.arange(M):
    U[:,t+1] = U[:,t]+(1*dt/h**2)* np.dot(S,U[:,t])
```

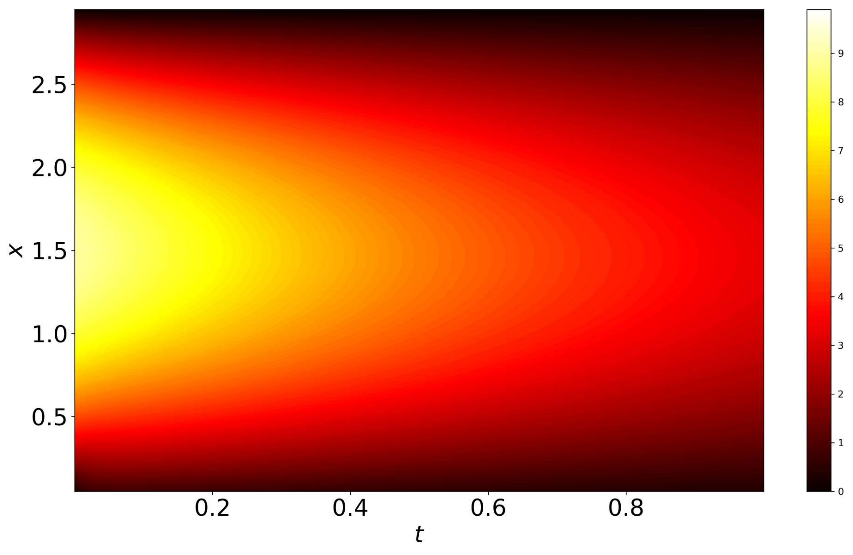


Figure 9: Heat conduction equation

§6.2 Finite Difference Method

Maxwell Equation:

$$\begin{aligned}\nabla\cdot\mathbf{D}&=\rho_f\\ \nabla\cdot\mathbf{B}&=0\\ \nabla\times\mathbf{E}&=-\frac{\partial\mathbf{B}}{\partial t}\\ \nabla\times\mathbf{H}&=\mathbf{J}_f+\frac{\partial\mathbf{D}}{\partial t}\end{aligned}\tag{65}$$

1-D

As the same method of solving the Heat conduction equation:

Because the Maxwell Equation can be rewritten the as:

$$\frac{\partial}{\partial t}\begin{pmatrix}X(t)\\ Y(t)\end{pmatrix}=\begin{pmatrix}0 & -\frac{1}{\sqrt{\mu}}\bigtriangledown\times\frac{1}{\sqrt{\varepsilon}}\\ \frac{1}{\sqrt{\varepsilon}}\bigtriangledown\times\frac{1}{\sqrt{\mu}} & 0\end{pmatrix}\begin{pmatrix}X(t)\\ Y(t)\end{pmatrix}\equiv H\begin{pmatrix}X(t)\\ Y(t)\end{pmatrix}\tag{66}$$

where $X(t)=\sqrt{\mu}H(t), Y(t)=\sqrt{\varepsilon}E(t)$

and H is skew-symmetric i.e. $H^T=-H$

simplify the equation $\frac{\partial}{\partial t}\varphi(t)=H\varphi(t)$

so for the 1-dim:for T-M Waves:

$$\begin{aligned}\frac{\partial}{\partial t}H_y(t)&=\frac{1}{\mu}\frac{\partial}{\partial x}E_z(t)\\ \frac{\partial}{\partial t}E_z(t)&=\frac{1}{\varepsilon}\frac{\partial}{\partial x}H_y(t)\end{aligned}\tag{67}$$

so $X(t)=\sqrt{\mu}H(t), Y(t)=\sqrt{\varepsilon}E(t)$ we can get

$$\begin{aligned}\frac{\partial}{\partial t}X_y(x,t)&=\frac{1}{\sqrt{\mu(x)}}\frac{\partial}{\partial x}(\frac{Y_z(x,t)}{\sqrt{\varepsilon(x)}})\\ \frac{\partial}{\partial t}Y_z(x,t)&=\frac{1}{\sqrt{\mu(x)}}\frac{\partial}{\partial x}(\frac{X_y(x,t)}{\sqrt{\varepsilon(x)}})\end{aligned}\tag{68}$$

discrete it:

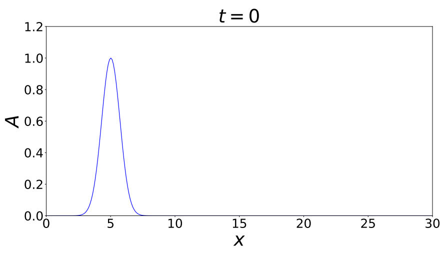
we can finally get the same form as the heat conduction equation

$$\begin{aligned}(X_y(i,t_{n+1})-X_y(i,t_n))/dt&=\frac{1}{\delta\sqrt{\mu_i}}(\frac{Y_z(i+1,t_n)}{\sqrt{\varepsilon_{i+1}}}-\frac{Y_z(i-1,t_n)}{\sqrt{\varepsilon_{i-1}}})\\ (Y_z(i,t_{n+1})-Y_z(i,t_n))/dt&=\frac{1}{\delta\sqrt{\varepsilon_j}}(\frac{X_y(j+1,t_n)}{\sqrt{\mu_{i+1}}}-\frac{X_y(j-1,t_n)}{\sqrt{\mu_{i-1}}})\end{aligned}\tag{69}$$

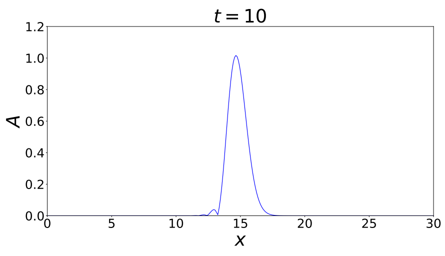
Example 6.3

Example of a wavepacket in one dimension.The cavity measures L = 30, with a mesh δ = 0.1.initial wavepacket, intensity of a Gaussian packet with parameters σ = 1 and x_0 = 5 Get the image of the magnetic field intensity distribution at t = 5

```python
import numpy as np
L=30#The Length of the Box
dx= 0.1#Mesh
c = 3e8#The Velocity of Light
#Define Original Wavepocket
def f(x,t):
    sigma = 1
    x0 = 5
    c = 3e8
    f = np.exp(-(x-c*t-x0)**2/sigma**2)
    return f
def TDME(x,dx,t,dt):
    N = int(x/dx)
    M = int(t/dt)
    U = np.zeros([N, M])
    U = np.array(U)
    S = (1/dx)*(np.eye(N, dtype=int,k=1)- np.eye(N, dtype=int,k=-1))
    for i in np.arange(M):
        U[0,i]= 0
        U[N-1,i]= 0
    for i in np.arange(N):
        U[i,0] = f(i*dx,0)
    for i in np.arange(M-1):
        U[:,i+1] = U[:,i]+dt*np.dot(S,U[:,i])
    return U
```



(a) t=0

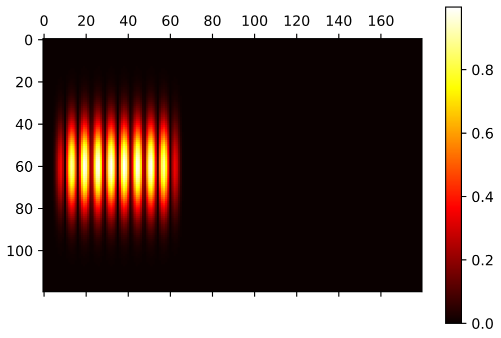


(b) t=5

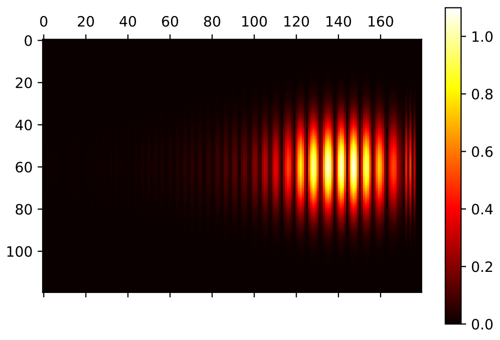
Figure 10: One Dimension Maxwell Equation

2-D

As the same way of 1D



(a) t=0



(b) t=10

Figure 11: Two Dimension Maxwell Equation

Remark 6.4. Because some reason I don't know why the wave do not diverge in the y direction

For 1-Dimension with $V = 0$

$$i\hbar\frac{\partial}{\partial t}\psi_l(t) = (-\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2} + V(x))\psi_l(t) \tag{70}$$

Consider $\hbar = 1$ $m = 1$

So we get:

$$\frac{\partial}{\partial t}\psi_l(t) = -i(-\delta^2[\psi_{l+1}(t) - 2\psi_l(t) + \psi_{l-1}(t)] + V(x))\psi_l(t) \tag{71}$$

and we can use the same method to solve this.

Example 6.5

Example of a wavepacket in one dimension.The cavity measures $L = 10$, with ameshδ= 0.1.initial wavepacket:intensity of a Gaussian packet with parameters σ= 1 and $x_0 = 2$ Get the image of the intensity distribution More Detail in github or in the Final-Project Part

All we need to solve is to solve function like:

$$\frac{\partial}{\partial t}\psi = H\psi \rightarrow \psi(t) = e^{tH}\psi(0) \tag{72}$$

What we need is to decomposite the e^{tH}

§6.3 Lie-Trotter-Suzuki Time Integration

Trotter-Suzuki Formula[2]

$$e^{x(A+B)} = \lim_{m\to\infty}(e^{\frac{xA}{m}}e^{\frac{xB}{m}})^m \tag{73}$$

and

$$e^{\frac{x(A+B)}{m}} = I + \frac{x}{m}(A+B) + \frac{1}{2}\frac{x^2}{m^2}(A^2 + AB + BA + B^2) + O((\frac{x}{m})^3) \tag{74}$$

$$e^{\frac{xA}{m}}e^{\frac{xB}{m}} = I + \frac{x}{m}(A+B) + \frac{1}{2}\frac{x^2}{m^2}(A^2 + 2AB + B^2) + O((\frac{x}{m})^3) \tag{75}$$

So

$$||e^{x(A+B)/m} - e^{xA/m}e^{xB/m}|| \leq \frac{x^2}{2m^2}||[A,B]|| \tag{76}$$

where $[A, B] = AB - BA$

Define:

$$U_1(\tau) = e^{-i\tau A_1}e^{-i\tau A_2}e^{-i\tau A_3}\cdots e^{-i\tau A_p} \tag{77}$$

$$U_2(\tau) = U_1^T(\frac{\tau}{2})U_1(\frac{\tau}{2}) \tag{78}$$

$$U_4(\tau) = U_2(p\tau)U_2(p\tau)U_2((1-4p)\tau)U_2(p\tau)U_2(p\tau) \tag{79}$$

where $p = 1/(4-4^{\frac{1}{3}})$

and from this decomposition we can decomposite the e^{tH}

because we have the relationship between the

$$exp[\alpha\begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix}] = \begin{pmatrix}cos\alpha & sin\alpha\\ sin\alpha & cos\alpha\end{pmatrix} \tag{80}$$

§6.4 Chebyshev Time Integration

Remember what we need:

Decomposite the e^{tH}

The expansion of a scalar function in Chebyshev polynomials:

$$f(x) = \frac{1}{2}a_0T_0(x) + \sum_{n=1}^{\infty}a_nT_n(x) \tag{81}$$

where $a_n = \frac{2}{\pi}\int_0^{\pi}cos(n\theta)f(cos\theta)d\theta$

and chebyshev polynomials T_n are given by $T_n(x) = cos(narccos(x))$

But its recursion relation is:

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x) \tag{82}$$

with $T_0(x) = 1, T_1(x) = x$

Because H is a real anti-symmetric matrix

So the eigenvalues of skew-symmetric matrix H are purely imaginary

If we want to get the real we can construct a matrix $A = -iH$

And A is is Hermitian and all its eignvalues are real and lie in the range$[-\rho(A), \rho(A)]$

where $\rho(A)$ is the spectral radius of A,$\rho(A) = max_{1\leq i\leq n}|\lambda_i|$

So:

$$\psi(t) = exp(tH)\psi(0) = exp(izB)\psi(0) = [\frac{1}{2}a_0(z)I + \sum_{n=1}^{\infty}a_n(z)T_n(B)]\psi(0) \tag{83}$$

where $a_n(z) = \frac{2}{\pi}\int_0^{\pi}cos(n\theta)exp(izcos\theta)d\theta = 2J_n(z)i^n$

so finally we get:

$$\exp(tH)\psi(0) = \left[J_0(z)I + 2\sum_{n=1}^{\infty} J_n(z) i^n T_n(B) \right] \psi(0) \tag{84}$$

Example 6.6

Example of a wavepacket in one dimension.The cavity measures $L = 30$, with a mesh $\delta = 0.1$ Initial wavepacket, magnetic field intensity of a Gaussian packet with parameters $\sigma = 1$ and $x_0 = 5$.8.1 Time evolution is solved with the Chebyshev algorithm ($\kappa = 10^{-13}$)

Remark 6.7. There we prepare two program to solve the problem one is to prepare the Chebyshev-algorithm and the other one is the main program,More detail in github

§7 Homework

§7.1 Homework 1

Python Version

```python
import numpy as np
#Define the recurrence relationship
def Bessel(x, n, m):
    if n >= m:
        return 0
    elif n == m-1:
        return 1
    else:
        return -Bessel(x, n+2, m) + 2*(n+1)*Bessel(x, n+1, m)/x

#Define normalization coefficient
coefficient = 0
#Calculate the normalization coefficient
def normailzed(x, m):
    sum = Bessel(x, 0, m)**2
    for i in range(1, m+1):
        sum += 2*Bessel(x, i, m)**2

    coefficient = (1/sum)**0.5

    return coefficient
#get the bessel function
#m is a larger number
Bessel(x, n, m)*normailzed(x, m)
```

Fortran Version

```fortran
real function bessel(n,m,x)
implicit none
real :: x
integer n,m,i,k
REAL(kind = 8), Allocatable :: J(:)
REAL(kind = 8) :: SUM
REAL(kind = 8) :: coff
```

```
Allocate(J(0:m-1))
i=m-3
J(m-1) = 0.0
J(m-2) = 1.0
do while(i>=0)
    J(i)=2*(i+1)*J(i+1)/x-J(i+2)
    i=i-1
end do
SUM=J(0)**2
k = M-1
DO while(k>=1)
SUM=SUM+2*J(k)**2
k =k-1
END DO
coff=(1.0/SUM)**0.5
bessel = coff*J(n)
end function
```

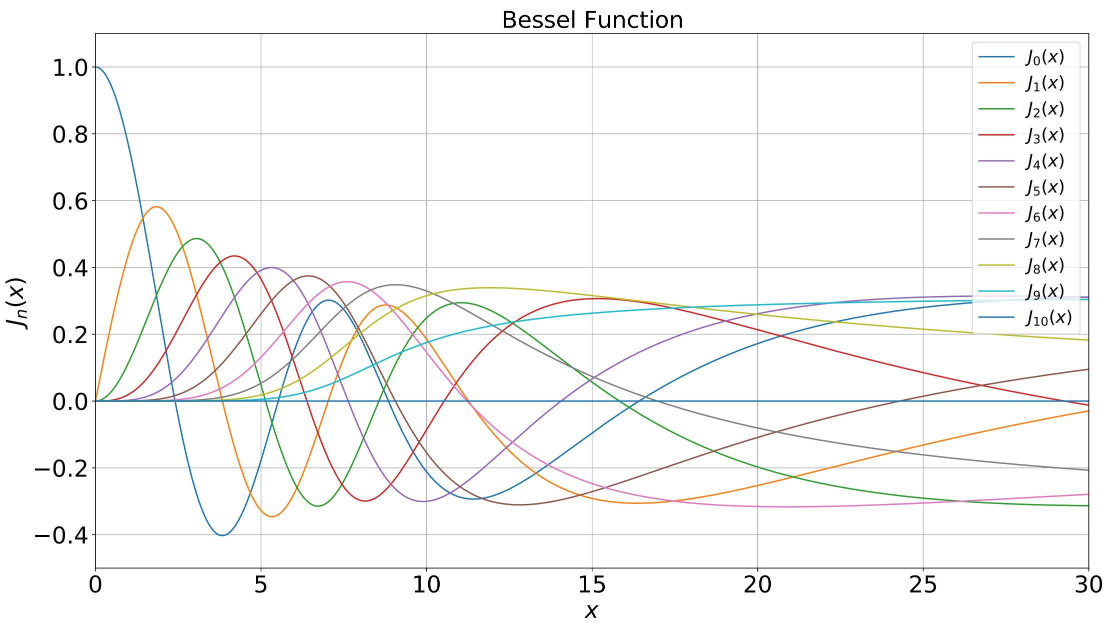


Figure 12: Bessel Function and the number of iterations is 10

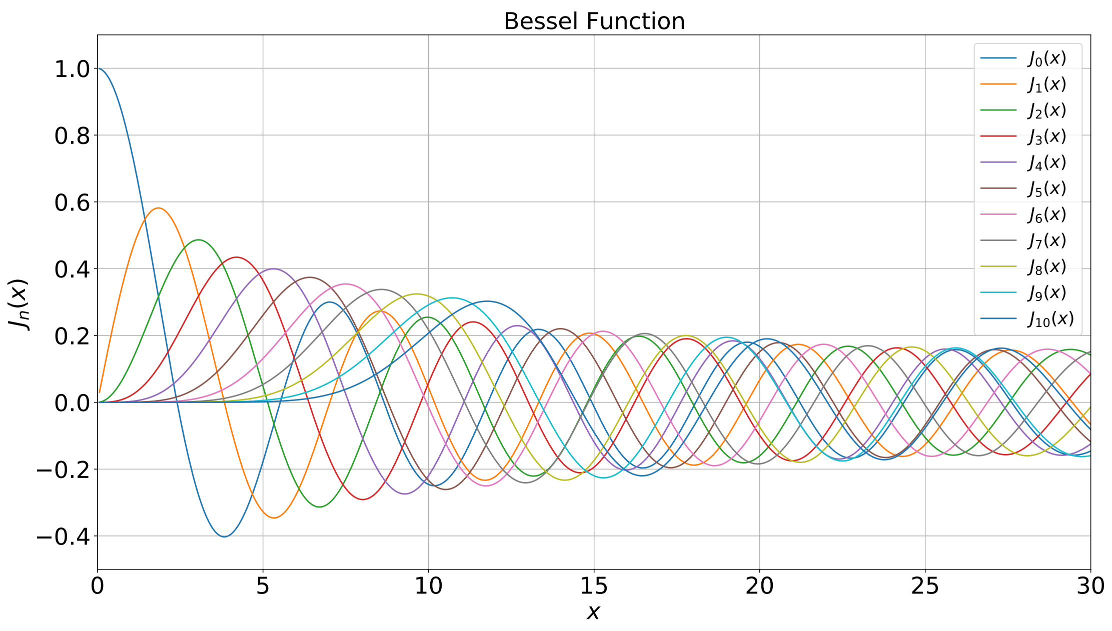


Figure 13: Bessel Function and the number of iterations is 30

Remark 7.1. The more your number of iterations you increase the more accurate data you will get,but cost more time

§7.2 Homework 2

§7.2.1 Homework 2-1

Python Version

```python
from sympy import *
#Import a package to calculate integral
#or you can use your own code to calculate the integral
Pi = 3.14
#DEFINE Lagrange Interpolating Polynomials
def L(n, k, a, b):
    x = symbols('x')
    h = (b-a)/n
    prob = 1
    for i in range(n+1):
        if i == k:
            pass
        else:
            prob = prob*(x - (a+i*h))/(k*h-i*h)
    return integrate(prob,(x, a, b))
#DEFINE Function
def f(x):
    g = 3*x**2
    return g
#DEFINE Polynomial
def P(n,a,b):
    h = (b-a)/n
    p = 0
    for i in range(n+1):
        p = p + f(a+i*h)*L(n,i,a,b)
    return p

P(1, 0, 1)#Trapezoidal Rule
P(2, 0, 1)#Simpson’ s Rule
abs(P(1, 0, 1)-Pi)/Pi
abs(P(2, 0, 1)-Pi)/Pi
```

fortran Version

```fortran
real function rect_inte(fx,a,b)
```

```fortran
real:: a,b,h
integer n
real:: fx(:)
n=size(fx)
h=(b-a)/n
rect_inte = sum(fx(:))*h
end

real function trap_inte(fx,a,b)
real:: a,b,h
integer n
real:: fx(:)
n=size(fx)
h=(b-a)/n
trap_inte = (sum(fx(1:n-1))+sum(fx(2:n)))*h/2;
end

real function simp_inte(fx,a,b)
real:: a,b,h
integer n
real:: fx(:)
n=size(fx)
h=(b-a)/n
simp_inte = (fx(1)+fx(n)+2*sum(fx(2:n-1:2))+4*sum(fx(2:n:2)))*h/3;
end
```

Remark 7.2. As soon as we code the function with fortran we can import the function into the python with a package "f2py".More detail in[3]

OUTPUT

```
Trapezoidal Rule method result:3.00000000000000
Relative error:0.0450703414486279
Simpson's Rule method result:3.13333333333333
Relative error:0.00262902329078934
```

§7.2.2 Homework 2-2

```
from sympy import *
#Import a package to calculate integral
#or you can use your own code to calculate the integral
Pi = 3.141592653589793
def L(n, k, a, b):
    x = symbols('x')
    h = (b-a)/n
    prob = 1
    for i in range(n+1):
        if i == k:
            pass
        else:
            prob = prob*(x - (a+i*h))/(k*h-i*h)
    return integrate(prob,(x, a, b))
def f(x):
    f = 4/(1+x**2)
    return f
def P(n,a,b):
    h = (b-a)/n
    p = 0
    for i in range(n+1):
        p = p + f(a+i*h)*L(n,i,a,b)
    return p
P(1, 0, 1)#Trapezoidal Rule
P(2, 0, 1)#Simpson' s Rule
abs(P(1, 0, 1)-Pi)/Pi
abs(P(2, 0, 1)-Pi)/Pi
```

```
OUTPUT
    Trapezoidal Rule method result:3.00000000000000
    Relative error:0.0450703414486279
    Simpson's Rule method result:3.13333333333333
    Relative error:0.00262902329078934
```

§7.2.3 Homework 2-3

```
import random
Pi = 3.14159265358979
m = 10#Number of repetitions
N = 1000000#Number of points
a = 0
for j in range(m):
    for i in range(N):
        x = random.uniform(-1, 1)
        y = random.uniform(-1, 1)
        if x**2 + y**2 <= 1:
            a = a + 1
        else:
            pass
Result = 4*a/(m*N)
Relative error = abs(4*a/(m*N)-Pi)/Pi
```

```
OUTPUT
    Result = 3.1418848
    Relative error 9.299309058299175e-05
```

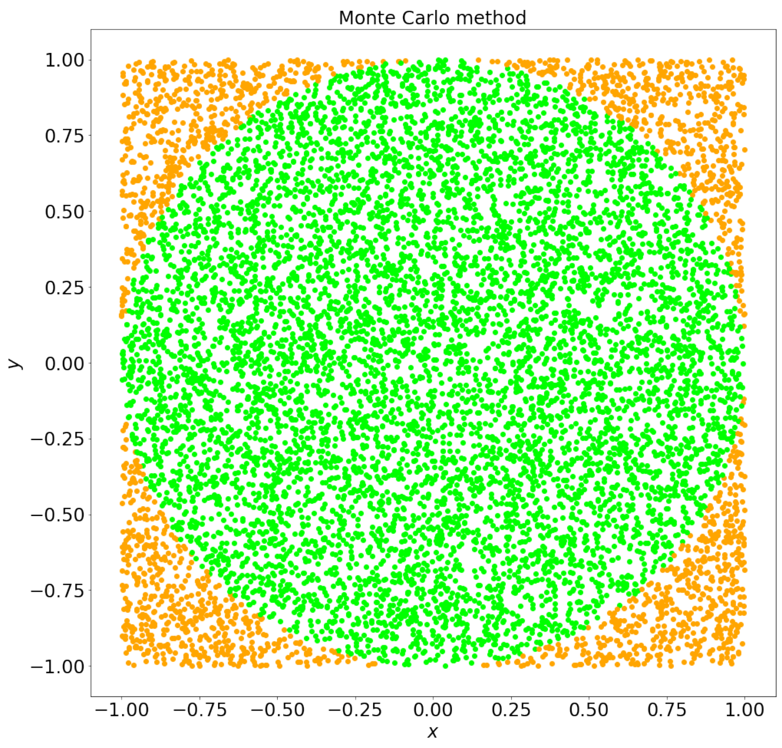


Figure 14: Monte Carlo method to calculate π

§7.3 Homework 3

§7.3.1 Homework 3-1

```
import numpy as np
#Define DFT we use the matrix form
def DFT(x):
    x = np.asarray(x, dtype=complex)
    N = x.shape[0]
    n = np.arange(N)
    k = n.reshape((N, 1))
    M = np.exp(-2j * np.pi * k * n / N)
    return np.dot(M, x)

def FFT(x):
    x = np.asarray(x, dtype=complex)
    N = x.shape[0]
    if N % 2 > 0:
        raise ValueError
        ("size of x must be a power of 2")
    elif N <= 8:
        return DFT(x)
    else:
        X_even = FFT(x[::2])
        X_odd = FFT(x[1::2])
        factor = np.exp(-2j*np.pi*np.arange(N) / N)
        X = np.concatenate(
            [X_even+factor[:int(N/2)]*X_odd,
            X_even+factor[int(N/2):]*X_odd])
    return X

#Find the inverse of a matrix
def iDFT(x):
    x = np.array(x, dtype=complex)
    N = x.shape[0]
    n = np.arange(N)
    k = n.reshape((N, 1))
    M = np.exp(2j * np.pi * k * n / N)
    return np.dot(M, x).reshape(N, 1)
```

```
def iFFT(x):
    x = np.asarray(x, dtype=complex)
    N = x.shape[0]
    if N % 2 > 0:
        raise ValueError
        ("size of x must be a power of 2")
    elif N <= 8:
        return iDFT(x)
    else:
        X_even = iFFT(x[::2])
        X_odd = iFFT(x[1::2])
        factor = np.exp(2j*np.pi*np.arange(N)/N)/N
        X = np.concatenate(
            [X_even+factor[:int(N/2)]*X_odd,
            X_even+factor[int(N/2):]*X_odd])
    return X
def CFFT(x, ISIGN):
    if ISIGN == 1:
        result = FFT(x)
    elif ISIGN == -1:
        result = iFFT(x)
    else:
        print("Please Input the Correct Parameter")
    return result
```

Fortran Version

```
module DFT
    contains
function DFT_fourier(x) result(w)
implicit none
integer n,i,j
complex :: x(:)
real :: pi
integer, dimension(size(x),1) :: k
complex :: Comp
complex, Allocatable :: result(:,:),x1(:,:)
complex, dimension(size(x)) :: w
pi = 3.1415926
```

```
Comp = (0,2)
n = size(x)
Allocate(x1(n,1))
Allocate(result(n,1))
x1 = reshape(x,(/n,1/))
DO j=0,n-1
k(j+1,1) = j
END DO
result = MATMUL(k,TRANSPOSE(k))
do i = 1,n
do j = 1,n
result(i,j) = exp(-Comp*pi*result(i,j)/n)
end do
end do
w = reshape(MATMUL(result,x1),(/n/))
end function
end module

function DFT_fourier(x) result(w)
implicit none
integer n,i,j
complex :: x(:)
real :: pi
integer, dimension(size(x),1) :: k
complex :: Comp
complex, Allocatable :: result(:,:),x1(:,:)
complex, dimension(size(x)) :: w
pi = 3.1415926
Comp = (0,2)
n = size(x)
Allocate(x1(n,1))
Allocate(result(n,1))
x1 = reshape(x,(/n,1/))
DO j=0,n-1
k(j+1,1) = j
END DO
result = MATMUL(k,TRANSPOSE(k))
do i = 1,n
do j = 1,n
result(i,j) = exp(-Comp*pi*result(i,j)/n)
end do
end do
w = reshape(MATMUL(result,x1),(/n/))
end function

recursive function FFT_fourier(x) result(w)
use DFT
implicit none
integer n,k
real :: pi
complex :: x(0:)
complex :: CompN,i,l
complex, Allocatable :: x_odd(:),x_even(:),w(:)
n = size(x)
Allocate(x_odd(0:n/2-1))
Allocate(x_even(0:n/2-1))
Allocate(w(0:n-1))
pi = 3.1415926535
i = (0,1)
CompN = exp(-2*i*pi/n)
DO k=0,(n/2)-1
    x_odd(k)=x(2*k+1)
    x_even(k)=x(2*k)
END DO
l = (1,0)
if (n<=4) then
w = DFT_fourier(x)
else
x_even = FFT_fourier(x_even)
x_odd = FFT_fourier(x_odd)
DO k=0,(n/2)-1
w(k)=x_even(k)+l*x_odd(k)
w(k+n/2)=x_even(k)-l*x_odd(k)
l = l*CompN
END DO
end if
end function
```

§7.3.2 Homework 3-2

```
import numpy as np
#Define Function
def f(z,i,N):
    return np.exp(1j*z*np.cos(2*np.pi*i/N))
#get the sequence of the DFT
def gs(z, N):
    s = []
    for m in range(N):
        s.append(f(z, m, N))#N must be the power of 2
    return s
def DFT(x,n):
    w = 0
    x = np.asarray(x, dtype=complex)
    N = x.shape[0]
    result = []
    # DFT:
    for k in range(N):
        w = w + x[k]*(1j**(-n)/N)*np.exp(2*np.pi*1j*n*k/N)
    result.append(w)
    return result

{J}_n(z) = DFT(gs(z, N),n)
```

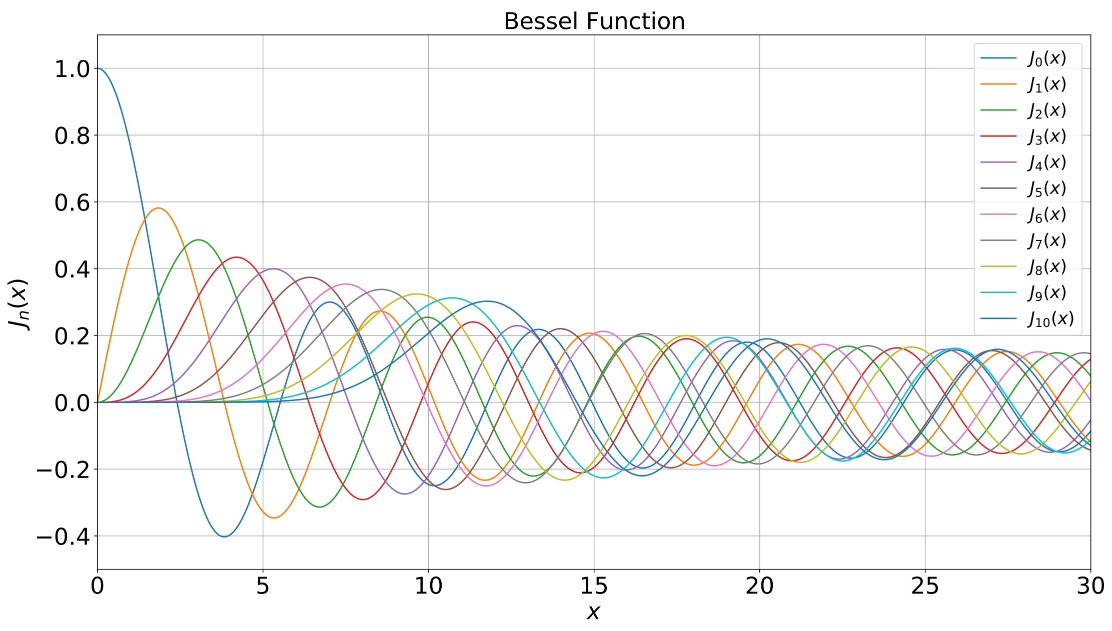


Figure 15: Bessel Function with Fourier Transform

§8 Final Project

Remark 8.1. More Details about the method in Lecture 6 6

§8.1 1D Maxwell Equation

Yee Method(FDTD Method)

```python
import numpy as np
#Define Parameter
L=30
dx= 0.1
c = 3e8
#Define Initial Wave
def f(x,t):
    sigma = 1
    x0 = 5
    c = 3e8
    f = np.exp(-(x-c*t-x0)**2/sigma**2)
    return f
def TDME(x,dx,t,dt):
    N = int(x/dx)
    M = int(t/dt)
    U = np.zeros([N, M])
    #Define the Hamiltonian
    S = (1/dx)*(np.eye(N, dtype=int,k=1)- np.eye(N, dtype=int,k=-1))
    #Time Evolution
    for i in np.arange(M):
        U[0,i]= 0
        U[N-1,i]= 0
        U[:,0] = f(np.arange(0,L,dx),0)
    for i in np.arange(M-1):
        U[:,i+1] = U[:,i]+dt*np.dot(S,U[:,i])
    return U
```

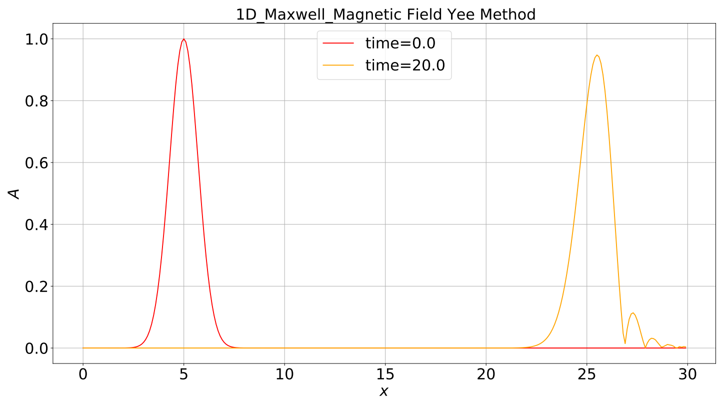


Figure 16: Yee Method(FDTD Method)

Trotter-Suzuki Method

```
import numpy as np
#Define Parameter
L=30
dx= 0.1
c = 3e8
#Define Initial Wave
def f(x,t):
    sigma = 1
    x0 = 5
    c = 3e8
    f = np.exp(-(x-c*t-x0)**2/sigma**2)
    return f
#Use Trotter-Suzuki Method
def H1(i):
    U1 = np.zeros(N)
    U2 = np.zeros(N)
    U1[i] = 1
    U2[i+1] =1
    return np.outer(U1,U2)-np.outer(U2,U1)
def H2(i):
    U1 = np.zeros(N)
    U2 = np.zeros(N)
    U1[i+1] = 1
    U2[i+2] = 1
    return np.outer(U1,U2)-np.outer(U2,U1)
```

```
for i in np.arange(1,N-1,2):
    Ux = Ux +H1(i)
for i in np.arange(-1,N-2,2):
    Uy = Uy +H2(i)
def Rotation(A,t,d):
    a,b = np.shape(A)
    coff = t/d
    for i in range(a):
        for j in range(b):
            if A[i,j] ==1:
                A[i,j] = np.sin(coff)
                A[i+1,j] = np.cos(coff)
                A[i,j-1] = np.cos(coff)
            elif A[i,j] == -1:
                A[i,j] = -np.sin(coff)
    return A
F = np.dot(Rotation(Ux,dt,dx),Rotation(Uy,dt,dx))
for i in range(int(t/dt)):
    U = np.dot(F,U)
#F acts as the Hamiltonian
```

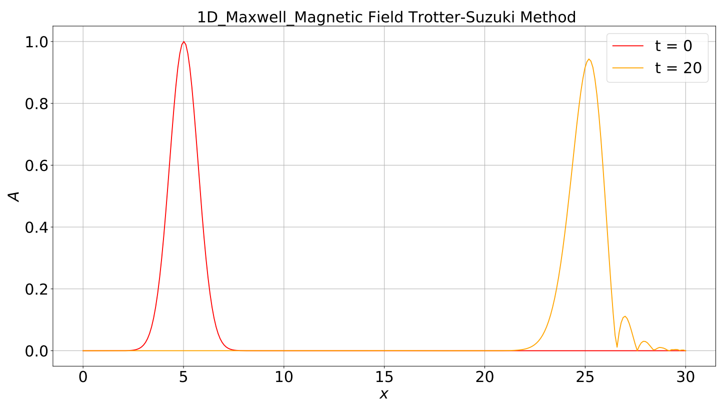


Figure 17: Trotter-Suzuki Method

Remark 8.2. The Main idea of the Trotter-Suzuki Method is to transform the exp operator into the rotation operator

Chebyshev Method

Remark 8.3. This part I will only post the Chebyshev-algorithm.py because the other part is as the same as the code above

```
import numpy as np
import scipy.special as spl
from spectral_radius import spectral_radius
c = 3*10**8
k = 1e-13
def cheb(x,dx,t):
    N = int(x/dx)
    Id = np.identity(N)
    H  = (1/dx)*(np.eye(N,k=1) - np.eye(N,k=-1))
    A = -1j*H
    z = t*spectral_radius(A)
    B = A/spectral_radius(A)
    T0 = Id
    T1 = 1j*B
    T2 = 2j*np.dot(B,T1)+T0
    F = spl.jv(0,z)*T0+2*spl.jv(1,z)*T1+2*spl.jv(2,z)*T2
    i = 3
    while True:
        if abs(spl.jv(i, z)) <k :
            break
        T0 = T1
        T1 = T2
        T2 = 2j*np.dot(B,T1)+T0
        F = F + 2*spl.jv(i,z)*T2
        i = i+1
    return F
```

Remark 8.4. When we use the Chebyshev Method we can first integrate a function like a module and use it to speed our code and make the main program look more concise

so my code constructure is:

|-Main-TDME.py

|-Chebyshev-algorithm.py

|-spectral-radius.py

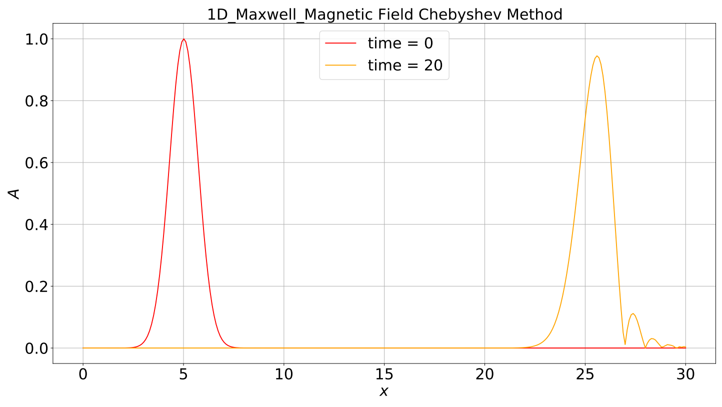


Figure 18: Chebyshev Method

§8.2 1D Schrödinger Equation

Remark 8.5. We can use the same method to solve the 1D Schrödinger Equation.The only difference between them is that the Hamiltonian is different.So what we need to change is the Hamiltonian.More details in github [4]

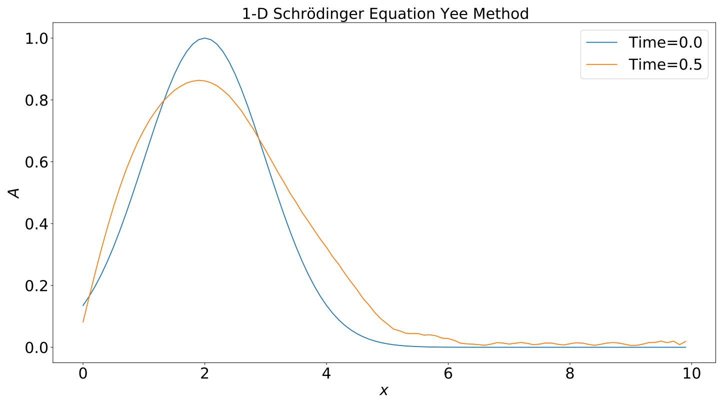


Figure 19: Yee Method(FDTD Method)

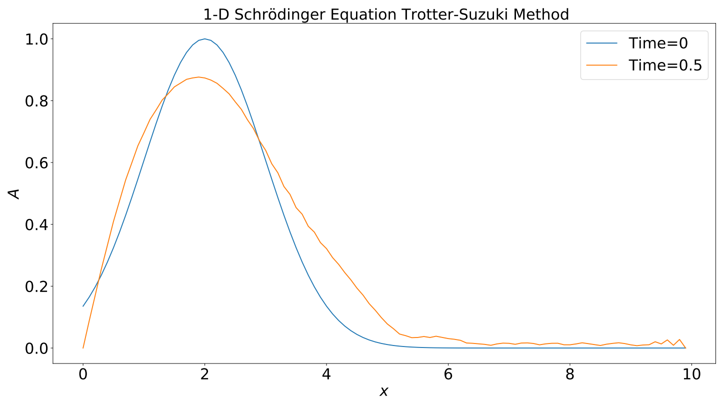


Figure 20: Trotter-Suzuki Method

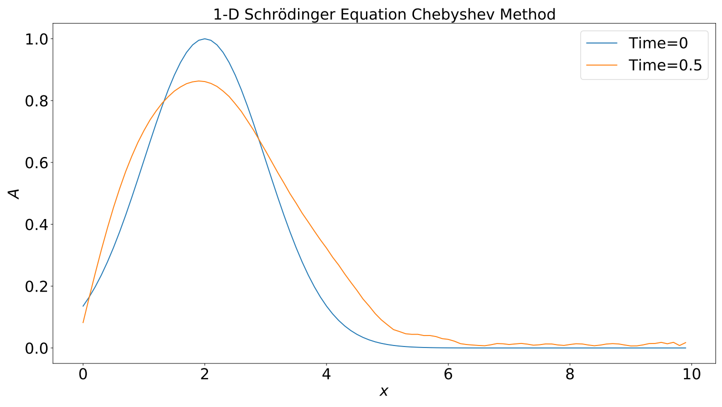
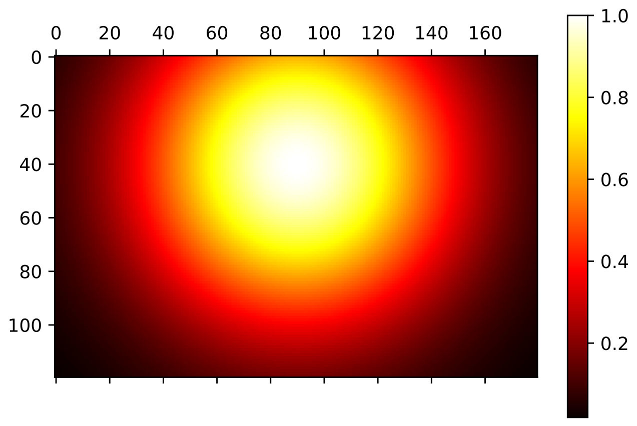


Figure 21: Chebyshev Method

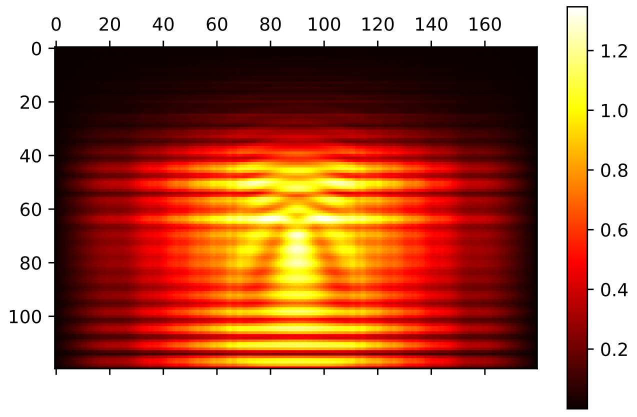
§8.3 2D Schrödinger Equation

[5]

> **Remark 8.6.** First we use the python package numpy to calculate and find it takes a long time to run so we replace the method with sparse matrix by using ”Scipy-Sparse” package to restore the matrix and perform matrix operations also we can use the Runge–Kutta methods to reduce the running time further



(a) t=0



(b) t=0.6

Figure 22: Two Dimension Schrödinger Equation

```
import numpy as np
from scipy import sparse
#Define Some Parameter
k=5
dx = 0.1
dy = 0.1
Lx=12
Ly=18
x0 = 4.0
y0 = 9.0
```

```
sigmax = 6
sigmay = 6
m = 1
hb = 1
N1 = int(Lx/dx)
N2 = int(Ly/dy)
x = np.arange(0,Lx,dx)
y = np.arange(0,Ly,dy)
y,x = np.meshgrid(y,x)
dt = 0.0001
#Define Initial Wave
def g(x,y):
    s = np.exp(1j*k*x)*np.exp(-((x-x0)/sigmax)**2-((y-y0)/sigmay)**2)
    return s
U0 = g(x,y)
#Define the Initial Potential
A=np.zeros([N1,N2])
for i in range(N1):
    for j in range(N2):
        a = j<85 or j>95
        b = 63 < i and i<68
        if a and b:
            A[i,j] = 5
        else:
            pass
get_potential = sparse.diags(A.flatten())
initial_wave = U0.flatten()
#Define Hamiltonian
diag1 = np.ones(N1)
diag2 = np.ones(N2)
diags1 = np.array([diag1,-2*diag1,diag1])
diags2 = np.array([diag2,-2*diag2,diag2])
D1 = sparse.spdiags(diags1,np.array([-1,0,1]),N1,N1)/dx**2
D2 = sparse.spdiags(diags2,np.array([-1,0,1]),N2,N2)/dy**2
H = sparse.csr_matrix(1j*(sparse.kronsum(D2,D1)/2-get_potential))
#Time Evolution
U = initial_wave
t = 0.7
```

```
for i in range(int(t/dt)):
    U = U + H@U*dt
#U is the result
```

References

[1] "numerical.recipes," http://numerical.recipes/, (Accessed on 07/06/2021).

[2] "Introduction —trottersuzuki 1.6.2 documentation," https://trotter-suzuki-mpi.readthedocs.io/en/stable/, (Accessed on 07/06/2021).

[3] "F2py users guide and reference manual —numpy v1.21 manual," https://numpy.org/doc/stable/f2py/, (Accessed on 07/14/2021).

[4] "Elondormancy/physics-note: Physics notes learn by myself," https://github.com/ElonDormancy/Physics-Note, (Accessed on 07/17/2021).

[5] H. De Raedt, "Computer simulation of quantum phenomena in nanoscale devices," *Annual Reviews of Computational Physics IV*, pp. 107–146, 1996.