量子ダイナミクス計算序論 with Python

佐藤駿丞 *

August 18, 2025

Contents

*東北大学理学研究科物理学専攻物性理論研究室

まえがき

このノートは、量子系の定常状態やダイナミクスを数値計算によって調べる手法を学ぶために作成したものである。数値計算は、紙と鉛筆だけでは解くことが難しい問題を、コンピュータを用いて調べるための強力な解析手段である。

ノートの前半では、Pythonを使ったプログラミングや算物理学の基礎的な内容に触れる。その後、一次元の量子系を題材として、波束の時間発展や基底状態の計算など、量子系を数値的に解析する方法を学んでいく。

また、本ノートの内容は随時更新されている。最近の内容は下記URLの最新のノートを確認してほしい。

`https://shunsuke-sato.github.io/page/etc/lecture_notes/LectureNoteForComputationalPhysics.pdf`

1 Pythonに慣れよう

この節では、Pythonを使ったプログラミングに慣れ、数値計算を行うための準備を行う。

1.1 Pythonの実行

まず最初に、Pythonを使ったプログラミングの基本的な手順について学ぼう。Pythonのプログラムは次のような手順で実行できる。

1. Python言語でコードを書き、拡張子`.py`を持つファイルとして保存する。
2. 前の手順で準備したファイルを`python`コマンドを用いて実行する。

この手順を学ぶために、次のPythonコードを書き、ファイル名を`hello.py`として保存してみよう。

ソースコード 1: Hello worldプログラム

```
1 print('Hello␣world!!')
```

ここで`print`文は、結果を出力するための関数である。上記のプログラムでは、文字列`Hello world!!`を出力するためにprint文が用いられている。また、Pythonでは文字をクォーテーション(')またはダブルクォーテーション(")で囲むことにより、その間の文字列を**文字列変数**として扱うことができる。

上記のコードが準備できたら、下記のコマンドをターミナル(端末)上で入力・実行することで、Pythonプログラムを実行してみよう。

```
$ python hello.py
Hello world!!
```

すると、文字列`Hello world!!`が出力されることが確認できるはずである。

ここまでで、Pythonプログラムを作成して実行する基本的な流れを理解した。次節では、Pythonにおける**変数**と**データ型**について学ぶ。

1.2 Pythonにおける変数の型

Pythonに限らず、プログラミングにおいて**変数**と**変数の型**は非常に重要な概念である。変数は、プログラム上の様々なデータの保持先である。また、その変数が保持しているデータの種類を**型**と呼ぶ。

- **変数**: データを格納する「名前付きの箱」
- **型**: 箱の中身（整数・実数・文字列など）の「データの種類」

Pythonにおける変数と型について理解するために、次のようなプログラムを書いて実行してみよう。

ソースコード 2: 変数の型を確認するコード

```
1  a = 2
2  b = 3.0
3  c = 'Tohoku'
4
5  print('a=',a)
6  print('b=',b)
7  print('c=',c)
8
9  print('type(a)=',type(a))
10 print('type(b)=',type(b))
11 print('type(c)=',type(c))
```

実行結果　（例）

```
b= 3.0
c= Tohoku
type(a)= <class 'int'>
type(b)= <class 'float'>
type(c)= <class 'str'>
```

Pythonでは、等号(=)は変数への代入操作を表す。この代入操作を用いて、ソースコード 2の1-3行目では、変数a,b,cに対して、それぞれ2, 3.0, 'Tohoku'を代入している。

ソースコード 2の5-7行目では、実際に代入されたデータをprint文を用いて出力している。この出力結果を見ることで、変数に正しいデータが代入されていることを確認することができる。

ソースコード 2の9-11行目では、上記の変数a,b,cの型をtype関数を用いて調べ、その結果をprint文を用いて出力している。type関数は、type(a)のように使うことで、変数aの型を確認することができる。

ソースコード 2を実行すると、変数a,b,cの型が、それぞれ**整数型(int)**,**浮動小数点型(float)**,**文字列型(str)**になっていることが確認できる。上記のプログラムでは、それぞれの変数にデータが代入される際に、データの型によって変数の型が定義されている。このように、Pythonではデータの代入などの際に、Pythonがデータの型を推定して処理を行ってくれる。それに対し、ほかの言語(例えば、C言語やFortran)では、変数を利用する前に、変数の方を明示的に指定することが必要となる場合がある。一見すると、自動的に型を推定して処理を行ってくれるのは大変便利な機能だが、型を明示的に指定することでソースコードのミス(バグ)を減らすことに繋がるため、型の指定には有用な一面もある。また、Pythonでソースコードを書く際にも、明示的に型を指定することも可能である。

1.3　Pythonにおける基礎的な数学演算

多くの物理シミュレーションは、四則演算やべき乗などの基礎的な数学演算を組み合わせて実行されている。ここでは、具体的にこのような数学演算を行うPythonコードを用いて、Pythonにおける数学演算の基礎について学ぶ。まず、次のPythonソースコード 3を見てみよう。

ソースコード 3: 四則演算コード

```
a = 3.0
b = 5.0

c = a + b
d = a - b
e = a * b
f = a / b
g = a ** b

print('a=',a)
print('b=',b)

print('a+b=',c)
print('a-b=',d)
print('a*b=',e)
print('a/b=',f)
print('a**b=',g)
```

このプログラムでは、まず第1行目と第2行目で、それぞれ変数aとbに実数値3.0と5.0を代入している。続いて、第4行目と第5行目では、上記で代入された実数値をprint文を用いて出力する命令が記述されている。

コードの8行目から12行目では、上記で用意した変数a,bを用いて四則演算とべき乗演算を実行し、その結果を新たな変数c,d,e,f,gへ代入している。Pythonでは加算演算を記号+、減算演算を記号-、乗算演算を記号*、除算演算を記号/、べき乗演算を記号**で表す。

上記のコードでは、最後に14行目から18行目で、上記の四則演算とべき乗演算の結果を、それぞれprint文を用いて出力する命令を行っている。

上記のコードを.py拡張子のファイルに保存し、ターミナル上でpythonコマンドを用いて実行してみよう。すると、上記のコードに書かれた命令が上から順番に実行されていく。その結果、得られた結果が想定された結果と一致しているか確認してみよう。

1.4 Pythonにおける整数演算

顕には述べなかったが、前節では浮動小数点数の演算について取り扱った。それに対し、ここでは整数型の変数に対する演算について見てみよう。ある整数の和・差・積は整数である。したがって、整数型変数の和・差・積もまた整数型変数であらわされる。しかし、整数同士の商は整数となるとは限らない。例えば、5/2は小数を用いれば2.5である。Pythonを用いて計算する場合も、記号/を用いた整数同士の商は浮動小数点数として扱われる。その一方で、整数同士の商として整数が与えられる方が便利な場合もある。例えば、5を2で割った商は2であり剰余は1である。このような整数の商を与える演算は記号//を用いて計算され、剰余を与える演算は記号%を用いて計算される。これらを踏まえて、以下のコード及びその実行結果を確認してみよう。

ソースコード 4: 四則演算コード

```
a = 11
b = 5

c = a + b
d = a - b
e = a * b
f = a / b
g = a ** b

h = a//b
i = a%b

print('a=',a)
print('b=',b)

print('a+b=',c)
print('a-b=',d)
print('a*b=',e)
print('a/b=',f)
print('a**b=',g)

print('a//b=',h)
print('a%b=',i)

print('type(a)=',type(a))
print('type(b)=',type(b))

print('type(c)=',type(c))
print('type(d)=',type(d))
print('type(e)=',type(e))
print('type(f)=',type(f))
print('type(g)=',type(g))

print('type(h)=',type(h))
print('type(i)=',type(i))
```

1.5 リスト

ここまであげたソースコードでは、一つの変数に対して一つの要素が格納されていた。このような一つの要素を格納するような変数をスカラー変数と呼ぶ。また、スカラー変数とは別に、複数の要素を格納するような変数も、実際のプログラミングを行う際には重要となる。ここでは、複数要素を一つの変数に格納するPythonの**リスト**について見てみよう。まずは、下記のプログラムを確認・実行してみよう。

ソースコード 5: リストの導入

```
la = ['Aoba', 'Kawauchi', 'Katahira']
lb = [1.23, 4.56, 7.98]
```

```
3
4  print('la=',la)
5
6  print('la[0]=',la[0])
7  print('la[1]=',la[1])
8  print('la[2]=',la[2])
9
10 print('lb=',lb)
11
12 print('lb[0]=',lb[0])
13 print('lb[1]=',lb[1])
14 print('lb[2]=',lb[2])
```

上記のソースコードの1行目では、'Aoba', 'Kawauchi', 'Katahira'という三つの文字列を要素として持つリストを[]により作成し、laという変数に代入している。このように、リストの作成は構成要素を[]で囲むことで実行できる。同様にして、2行目ではlbに3つの浮動小数点数を格納している。

ソースコードの4行目では、print文を用いてリスト全体を出力している。これに対して6から8行目では、リストの要素を一つ一つ出力している。このように、リストの一部の要素のみを読み出すことも可能であり、例えばla[2]はリストの2番目の要素を表す変数だ。ここで、リストの要素番号は0番目から数え始めることに注意しておこう。同様に、10行目から14行目では浮動小数点数を格納したリストlbの出力を行っている。

1.6 リストと反復処理

プログラムを書く際に、同一の手続きを繰り返し反復して実行する場合がある。そのような反復処理を行う際には、下記に述べるforループを用いるのが便利である。下記のソースコードについて見てみよう。

ソースコード 6: リストを用いた反復処理

```
1 la = ['Aoba', 'Kawauchi', 'Katahira']
2
3
4 for l in la:
5     print(l)
6
7
8 print('Bye!')
```

上記のソースコードの第1行目では'Apple', 'Banana', 'Chocolate'という3つの文字列を要素として持つリストを作り、変数laに格納している。

コードの3行目ではfor l in la:によってfor文による反復処理部分の領域の定義を開始している。ここで変数lは反復処理毎に変化する変数であり、ループの各反復毎にリスト変数laの各要素を値として持つ。上記のコードの場合、変数lに'Aoba', 'Kawauchi', 'Katahira'の3つの文字列が順番に代入され、それぞれの反復処理において反復領域の処理が実行される。

Pythonにおいては、for文の反復処理部分の領域は字下げ(インデント)を用いて定義される。例えば、上記のソースコードの場合は、print(l)の部分のみが反復処理される。

反復処理領域の記述が終わったら、インデントを削除し、次の処理内容を記載する。上のソースコードの例では、forループの反復処理の後に、print('Bye!')によって'Bye!'という文字列を出力し、プログラムを終了している。

まとめると、上記のプログラムではまず1行目で3つの要素を持つリスト変数laを作り、その後、forループを用いてリストの各要素をprint(l)によって出力し、最後に終了のログ('Bye!')を出力してプログラムを終了している。

上記のプログラムを実行し、リストとfor文を用いた反復処理の振る舞いについて確認してみよう。

1.7 range関数

前節 1.6では、リストを用いた反復処理について述べた。そこでは、リスト内の各要素に対し

て繰り返し処理を実行する方法について学んだ。for文を用いた反復処理には、リストを用いるほかに、次に述べるrange関数を用いることもできる。このrange関数は等間隔の数列を作る関数である。まずは、下記のソースコードを見てrange関数の振る舞いについて見てみよう。

ソースコード 7: range関数の振る舞い

```
1  r1 = range(3)
2  print(r)
3
4  l1 = list(r1)
5  print(l1)
6
7
8  r2 = range(7,11)
9  print(r2)
10
11 l2 = list(r2)
12 print(l2)
13
14
15 r3 = range(11,19,2)
16 print(r3)
17
18 l3 = list(r3)
19 print(l3)
```

上記のソースコードの第1行目では、range関数の最も単純な使い方を示している。このように、range(n)という形でrange関数を呼び出すと、整数0からn-1までの数列が生成される。上記のソースコードの場合は、range(3)によって0,1,2という数列が生成され、これが変数r1にrange型のオブジェクトとして格納されている。

ソースコードの第4行目では、range型の変数であるr1を関数list()を用いてリスト型の変数に変換している。5行目では、変換されたリストを出力している。

コードの第8行目では、range(m,n)という形でrange関数が用いられている。この場合、range関数はmからn-1までの数を含む数列を生成する。また、このようにして生成された数列もリスト関数(list())を用いてリスト型のオブジェクトに変換することができる。上記のコードの例では、range(7,11)によって7,8,9,10という数列が生成される。

最後に、コードの15行目ではrange(l,m,n)の形でrange関数が用いられている。この場合、range関数はlを始点としmを超えない範囲で公差がnの数列を生成する。すなわち、上記のコードの場合range(11,19,2)によって数列11,13,15,17が生成される。

実際に、上記のプログラムを実行することで、range関数の振る舞いを確かめてみよう。

1.8 range関数を用いた反復処理

次に、range関数を用いた反復処理についてみてみよう。listの場合と同様に、range関数によって生成した数列を要素として、forループによる反復処理を行うことができる。まずは、下記のソースコードを見てみよう。

ソースコード 8: range関数と反復処理

```
1  s = 0
2  for i in range(1,101):
3      s = s + i
4
5  print('s=',s)
```

上記のコードではまず、第1行目で変数sに0という値を代入し、変数を**初期化**している。第2行目では、変数iを用いたfor文により反復処理領域の定義を開始している。ここでは、range関数を用いて1から100までの整数の数列を生成し、変数iに数列の各整数値を代入して以降の反復処理領域の処理を行う。

コードの第3行目はインデント（字下げ）されており、これはその上のfor文によって定義された反復処理領域であることを表す。この行では、変数sに変数iの値を足し、その結果を変

数sに代入する処理を行っている。最後に、第5行目で、上記のforループの反復処理によって計算されたsの値をprint文を用いて出力している。

上記のコードでどのような処理が行われているのかを考えて、出力される結果を予想してみよう。また、実際にコードを実行することで、予想された結果が得られているかを確認してみよう。

1.9 Numpyを用いた数値計算の準備

Pythonで数値計算を行う際には、とても強力なライブラリであるNumpyを利用することができる。このノートで行う数値計算でも、Numpyを利用して様々な数値計算を行っていく。ここでは、Numpyの使い方の基本について学び、以降で学ぶ数値シミュレーションの準備を行う。

まずは、次のソースコード 9を見てみよう。

ソースコード 9: numpyの利用例

```
1  import numpy as np
2
3  a = np.array([0.0, 2.0, 4.0])
4  print(a)
5
6  x = np.linspace(0.0, 5.0, 6)
7  print(x)
8
9  y = np.exp(-x)
10 print(y)
```

ソースコード 9の1行目では、import numpy as npによってNumpyをインポートし、コードの中でNumpyを使う準備をしている。ここではas npによってNumpyを短縮名npを定義して短縮名でNumpyとして呼び出せるようにしている。

コードの3行目では、np.array関数によって、Pythonのリスト[0.0, 2.0, 4.0]をNumpyの配列(array)に変換している。Numpyの配列は、Pythonのリストに似ているが、Numpyによって効率的に計算できる便利なオブジェクトである。コードの４行目では、用意したNumpyの配列を画面に出力している。

コードの6行目では、Numpyの関数np.linspaceを用いて0.0から5.0までの数値を等間隔に分割した6点の値を要素に持つNumpyの配列を作成している。コードの7行目で、print文を用いて、作成した配列の要素を画面に出力している。

コードの9行目では、上記で用意したNumpyの配列xをNumpyの指数関数np.exp(-x)に入力し、配列xの各要素の指数関数の値を計算し、結果をNumpy配列yに結果を代入してる。さらに、コードの10行目でprint文を用いて結果を出力している。

1.10 matplotlibを利用した可視化

ここでは、Pythonを用いた数値計算の結果を可視化するための方法として、matplotlibを用いた可視化方法について学ぶ。matplotlibは可視化用の非常に便利なライブラリであり、Pythonで処理したデータをそのままプロットすることができる。

まずは、次のコードを見てみよう。

ソースコード 10: matplotlibを用いた作図例

```
1  from matplotlib import pyplot
2  import numpy as np
3
4
5  x = np.linspace(0.0, 10.0, 32)
6  y = np.cos(x)
7
8  pyplot.plot(x, y)
9  pyplot.savefig("plot.png")
10 pyplot.show()
```

ソースコードの第一行目では、`from matplotlib import pyplot`により`matplotlib`ライブラリの関数である`pyplot`をインポートしている。これにより、`pyplot`を用いたグラフ描画の準備が整う。第二行目では、三角関数の計算のために`numpy`をインポートしている。

第5行目では`x`を0から10まで32個の等間隔な点でサンプリングした`numpy`の配列を生成し、第6行目では生成したサンプル点の値xに対して三角関数$\sin(x)$を計算している。

第8行目では、`pyplot.plot(x,y)`により、上で用意した`x`及び`y`をそれぞれx軸、y軸の値として描画を行っている。さらに第9行目で、ここで作成された図を`plot.png`という名前の`png`ファイルに保存している。最後に、`pyplot.show()`により同様の図を画面上に出力してプログラムは終了する。

上記のソースコード10によって出力された図を以下に示す。

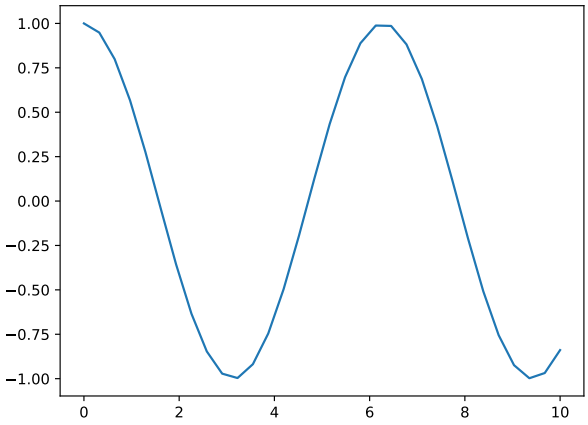

Figure 1: `matplotlib`を用いた可視化の例。

2　数値微分

この節では、数値計算を使って関数の微分を評価する**数値微分**について学ぶ。

2.1　差分近似と前進差分

数値的に微分を行う前に、数学的な微分の定義を振り返ってみよう。関数$f(x)$の微分は次のように定義される。

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}. \tag{1}$$

この定義に基づき、十分小さなhを用いることで関数$f(x)$の微分$f'(x)$を次のように近似することを考える。

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}. \tag{2}$$

このような有限の大きさの幅hを使って微分を近似する方法は**差分近似**と呼ばれる。特に、式(2)で表されるような近似公式は**前進差分公式**と呼ばれる。その意味は、次節で述べる**中心差分公式**と比較することで明らかになるだろう。

差分近似の有効性を検証するために、関数$f(x) = e^x$の$x = 0$における微分値を式(2)を用いて求め、厳密な値である$f'(0) = e^0 = 1$と比較してみよう。以下に、サンプルのソースコードを示す。

ソースコード 11: 前進差分公式による微分値の評価

```
1  import numpy as np
2
3
4  x = 0.0
5  h = 0.1
6
7  fx = np.exp(x)
8  fxph = np.exp(x+h)
9
10 num_dfdx = (fxph-fx)/h
11 ana_dfdx = np.exp(x)
12
13 error = np.abs(num_dfdx - ana_dfdx)
14
15 # Print results
16 print(f'For␣h␣=␣{h}:')
17 print(f'Numerical␣derivative␣of␣exp({x})␣=␣{num_dfdx}')
18 print(f'Analytical␣derivative␣of␣exp({x})␣=␣{np.exp(x)}')
19 print(f'Error␣=␣{error}')
```

ソースコードの第一行目では`import numpy as np`によって`numpy`という数値計算を行うための拡張モジュールをインポート(import)している。このインポートの際、`as np`を付けることによって、`numpy`を略称`np`を用いて呼び出すことが出来るようになる。今回のコードの例では、`numpy`モジュール内の指数関数を呼び出すためにモジュールをインポートしている。

上記のプログラムの例では$x = 1$における微分値を刻み幅$h = 0.1$で求めるために、4,5行目で`x = 0.0`と`h = 0.1`によって値を設定している。

第7,8行目ではx及び$x + h$における関数の値を`numpy`の指数関数`np.exp()`によって計算し、変数`fx`及`fxph`に代入している。さらに、第10行目では前進差分公式(2)を用いて微分値を評価している。第13行目では、上記の計算によって得られた微分の近似値、及び厳密な微分値との差が出力されている。

上記のプログラムを実行することで、差分近似によって微分値が良く近似できているか確かめてみよう。また、刻み幅hを変化させることで近似の精度がどのように変わるか調べてみよう。

2.2　中心差分公式と差分近似の精度

前節の差分近似の公式(2)は、幅hが小さくなるにつれて誤差も小さくなる。誤差の大きさは、次のようにTaylor展開を用いて見積もることが出来る。

$$f(x+h) = f(x) + f'(x)h + \mathcal{O}(h^2). \tag{3}$$

したがって、

$$f'(x) = \frac{f(x+h) - f(x)}{h} + \mathcal{O}(h). \tag{4}$$

このように、前進差分公式の誤差は$\mathcal{O}(h)$であり、hが小さい極限でhに比例して誤差が小さくなる。

ところで、次のような二つのTaylor展開を考えてみよう。

$$f(x+h) = f(x) + f'(x)h + \frac{1}{2}f"(x)h^2 + \mathcal{O}(h^3), \tag{5}$$

$$f(x-h) = f(x) - f'(x)h + \frac{1}{2}f"(x)h^2 + \mathcal{O}(h^3). \tag{6}$$

この二つの式の差を取ることで、次のような差分公式を得ることが出来る。

$$f'(x) = \frac{f(x+h) - f(x-h)}{h} + \mathcal{O}(h^2). \tag{7}$$

この差分公式は、**中心差分公式**と呼ばれ、正と負の領域に対称に差分を取っている。また、誤差項は$\mathcal{O}(h^2)$となっており、h^2のオーダーの誤差があることが分かる。

ここでは、前進差分公式(2)と中心差分公式(7)の精度が刻み幅hに対してどのように振舞うかを、実際にプログラムを書いて調べてみよう。上記の例と同様に、関数$f(x) = e^x$の$x = 0$における微分値を求めるプログラムを下記に示す。

ソースコード 12: 前進差分公式と中心差分公式による微分値の評価

```
1  from matplotlib import pyplot as plt
2  import numpy as np
3
4  # Parameters
5  x = 0.0  # Point at which derivative is evaluated
6  step_sizes = np.array([1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1.0])  # Differentiation step sizes
7
8  # Numerical derivatives
9  forward_diff = (np.exp(x + step_sizes) - np.exp(x)) / step_sizes
10 central_diff = (np.exp(x + step_sizes) - np.exp(x - step_sizes)) / (2.0 * step_sizes)
11
12 # Error evaluation
13 error_forward = np.abs(forward_diff - np.exp(x))
14 error_central = np.abs(central_diff - np.exp(x))
15
16 # Plotting the data
17 plt.plot(step_sizes, error_forward, label="Forward␣Difference", marker='o')
18 plt.plot(step_sizes, error_central, label="Central␣Difference", marker='x')
19 plt.xscale('log')
20 plt.yscale('log')
21 plt.xlabel('Step␣Size␣(h)')
22 plt.ylabel('Error')
23 plt.title('Error␣in␣Numerical␣Differentiation')
24 plt.legend()
25 plt.grid(True)
26
27 # Saving the plot
28 plt.savefig("error_derivative.png")
29 plt.show()
```

ここで、上記のPytonプログラムについて見てみよう。第4行目は#から始まるコメント行であり、プログラム上は何も行われず、人間がプログラムを読む際の参考にする行である。第6行

目では`x = 1.0`によって微分を計算する点を指定しており、6行目では、差分法によって微分を評価する際に用いてる様々な刻み幅の値をNumpyの配列として定義している。

次に 9 行目では、刻み幅の配列`step_sizes`の各値について、前進差分公式を用いて微分値を数値的に評価している。さらに10行目では、中心差分公式を用いて微分値を数値的に評価している。

次に、13行目、14行目では、前進差分・中心差分公式を用いて数値的に評価した微分値と厳密な微分値の差を計算し、差分公式の誤差の評価を行っている。

さらに第16行目以降では`matplotlib`を用いて、上記の計算で評価した差分公式の誤差をグラフとして出力している。Figure 2にこのコードで生成される図を示す。図の横軸と縦軸が対数軸となっていることに注意すると、前進差分近似の誤差(Forward difference; 青線)と中心差分の誤差(Central difference; オレンジ線)が幅hの異なるべき乗に比例している様子が分かる。この振る舞いは、まさに上で議論した誤差の大きさの議論と整合した結果となっている。

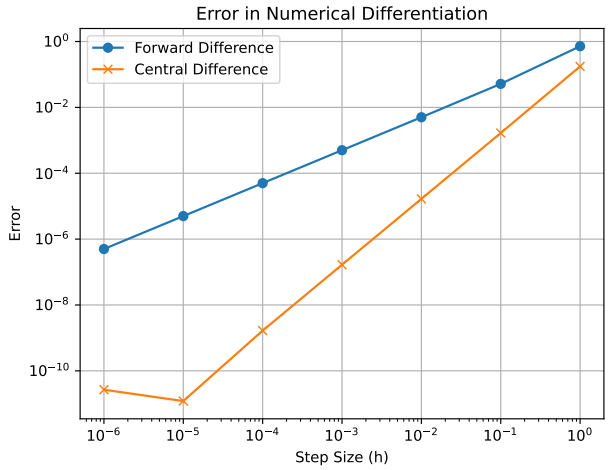

Figure 2: 前進差分近似と中心差分近似の誤差の振る舞い。

2.3 二階微分の数値微分

前節では、有限差分近似によって一階微分の数値計算を行った。ここでは、一階微分の際に学んだ知識を発展させ、二階微分を数値的に評価する方法を学ぶ。式(6)のTaylor展開を用いることで、次のような関係式を得ることが出来る。

$$\frac{f(x+h)-2f(x)+f(x-h)}{h^2} = \frac{d^2f(x)}{dx} + (O)(h^2). \tag{8}$$

この式を用いることで数値的に二階微分を評価することができる。また、誤差は(h^2)のオーダーである。

次のPythonコードは、関数$f(x) = \cos(x)$の二階微分$f''(x)$を数値的に計算し、結果をファイル`cos_derivative.dat`に出力するプログラムを表す。同様のコードを自作し、二階微分とPythonプログラムに慣れてみよう。

ソースコード 13: 差分近似を用いた2回微分の評価

```
1 import numpy as np
2 from matplotlib import pyplot as plt
3
4 # Constants
5 dx = 0.1
6 nx = 128
7 x_start = -6.3
8 x_end = 6.3
```

```
9
10 # Generate x values
11 x = np.linspace(x_start, x_end, nx)
12
13 # Compute function values
14 fx = np.cos(x)
15 fx_pdx = np.cos(x + dx)
16 fx_mdx = np.cos(x - dx)
17
18 # Second derivative using central difference
19 d2fdx2 = (fx_pdx - 2 * fx + fx_mdx) / dx**2
20
21 # Plotting the function and its second derivative
22 plt.plot(x, fx, label="f(x) = cos(x)")
23 plt.plot(x, d2fdx2, label="f''(x)")
24
25 plt.xlabel('x')
26 plt.ylabel('f(x)')
27 plt.title('Function and Second Derivative')
28 plt.legend()
29 plt.savefig("second_derivative_cos.png")
30 plt.show()
```

上記のプログラムを実行した結果得られる、$f(x) = \cos(x)$と$f''(x) = -\cos(x)$を比較した図をFigure 3に示す。

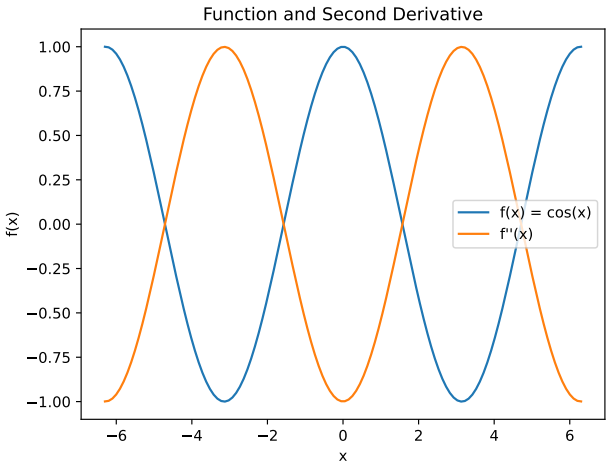

Figure 3: 前進差分近似と中心差分近似の誤差の振る舞い。

3 数値積分

3.1 台形公式

この節では、数値計算を使って関数の積分を評価する**数値積分**について学ぶ。まず、次のような一次元積分を考えてみよう。

$$S = \int_a^b dx f(x) \tag{9}$$

ここで、積分区間($a \le x \le b$)をN分割することを考える。分割の幅Δxは$\Delta x = (b-a)/N$によって与えられる。積分Sを次のように、分割された区間ごとの積分の和として書き直してみよう。

$$\begin{aligned} S &= \int_a^b dx f(x) = \int_a^{a+\Delta x} dx f(x) + \int_{a+\Delta x}^{a+2\Delta x} dx f(x) + \int_{a+2\Delta x}^{a+3\Delta x} dx f(x) + \cdots + \int_{a+(N-1)\Delta x}^{b} dx f(x) \\ &= \sum_{j=0}^{N-1} \int_{a+j\Delta x}^{a+(j+1)\Delta x} dx f(x) \end{aligned} \tag{10}$$

ここで、分割幅Δxが十分小さいとすると、被積分関数$f(x)$は各微小積分区間の端の二点を通る一次関数として近似できる。すなわち、微小区間($a + j\Delta x \le x \le a + (j+1)\Delta x$)において、$f(x)$を次のように近似することができる。

$$f(x) \approx f(a+j\Delta x) + \frac{f(a+(j+1)\Delta x) - f(a+j\Delta x)}{\Delta x} \left(x - a - j\Delta x\right). \tag{11}$$

また、微小区間の積分は次のように近似できる。

$$\int_{a+j\Delta x}^{a+(j+1)\Delta x} dx f(x) \approx \frac{f(a+(j+1)\Delta x) - f(a+j\Delta x)}{2} \Delta x. \tag{12}$$

すなわち、式(10)を次のように近似することが出来る。

$$\begin{aligned} S &= \sum_{j=0}^{N-1} \int_{a+j\Delta x}^{a+(j+1)\Delta x} f(x) \\ &\approx \sum_{j=0}^{N-1} \int_{a+j\Delta x}^{a+(j+1)\Delta x} \frac{f\left(a+(j+1)\Delta x\right) + f\left(a+j\Delta x\right)}{2} \\ &= \sum_{j=0}^{N-1} \frac{f\left(a+(j+1)\Delta x\right) + f\left(a+j\Delta x\right)}{2} \int_{a+j\Delta x}^{a+(j+1)\Delta x} \\ &= \sum_{j=0}^{N-1} \frac{f\left(a+(j+1)\Delta x\right) + f\left(a+j\Delta x\right)}{2} \Delta x \\ &= \frac{f(a)}{2}\Delta x + \left[\sum_{j=1}^{N-1} f(a+j\Delta x)\Delta x\right] + \frac{f(b)}{2}\Delta x. \end{aligned} \tag{13}$$

式(13)の近似式は**台形近似公式 (Trapezoidal rule)**と呼ばれる積分Sの近似公式である。分割数が大きい極限($N \to \infty$, $\Delta x \to 0$)で式(13)の台形公式が厳密な積分値Sと一致する。

次の積分を台形公式を用いて数値的に評価しよう。

$$S = \int_1^2 dx \frac{1}{x} = \log(2) - \log(1) = \log(2) \approx 0.6931471805599453 \tag{14}$$

以下に、上記の積分を数値的に評価するPythonコードを示す。数値積分とPythonプログラミングになれるために、サンプルコードを参考に自作のコードを作成して積分を評価してみよう。

ソースコード 14: 台形公式を用いた数値積分の例

```
import numpy as np

a = 1.0
b = 2.0
n = 64

h = (b-a)/n

s = (1.0/a+1.0/b)/2.0
for i in range(1,n):
    x = a + i*h
    s += 1/x

s = s*h

print('num. integral = ',s)
print('log(2)        = ',np.log(2))
```

4　1階常微分方程式の求解

微分方程式は、様々な現象を理解するうえで非常に重要であるが、限られた場合を除いて解析的に解を求めることが難しい。そのような場合でも、数値計算によって微分方程式を求解できる場合がある。この節では、1階常微分方程式を数値的に解く方法について学ぶ。まず、次のような1変数の1階常微分方程式について考えよう。

$$\frac{dy(x)}{dx} = f\left(x, y(x)\right). \tag{15}$$

4.1　Euler法による求解

まず初めに、最も基本的なEuler法について学ぶ。第2.1節で前進差分公式(2)について議論した場合と同様に、微分の定義について振り返ってみよう。ある関数$y(x)$の微分は次のように定義される。

$$y'(x) = \lim_{h \to 0} \frac{y(x+h) - y(x)}{h}. \tag{16}$$

この式においてhが十分小さいと仮定すると、次のように前進差分公式を用いて微分を近似することが出来る。

$$y'(x) \approx \frac{y(x+h) - y(x)}{h}. \tag{17}$$

この式を書き換えることで、さらに次式を得る。

$$y(x+h) \approx y(x) + y'(x)h. \tag{18}$$

式(18)を見ると、右辺はxにおける$y(x)$と$y'(x)$の情報のみで書かれており、これらの情報を使って$x+h$における関数の値$y(x+h)$を近似的に評価することができる。したがって、このような操作を逐次的に繰り返すことによって、任意の変数xにおける関数の値$y(x)$を評価することが出来る。このような微分方程式の数値解法をEuler(オイラー)法と呼ぶ。

以下で、もう少し具体的にEuler法の手続きについて述べる。

1. まず初めに、微分方程式(15)において初期条件$y(x_0) = y_0$を設定する。
2. 次に、$f(x_0, y(x_0))$を評価し、微分値$y'(x_0)$を評価する。
3. 評価した微分値$y'(x_0)$を用いて、式(18)により$x_0 + h$における関数の値$y(x_0 + h)$を評価する。
4. 評価した$y(x_0 + h)$をもとに、関数値$f(x_0 + h, y(x_0 + h))$を計算し、$x_0 + h$における微分値$y'(x_0 + h)$を評価する。
5. 評価した微分値$y'(x_0 + h)$を用いて、式(18)により$x_0 + 2h$における関数の値$y(x_0 + 2h)$を評価する。
6. 以下、同様の手順を繰り返す。

このような手続きを繰り返し実行することで、微分方程式を数値的に解くことが出来る。ここでは、Euler法を用いた数値的な微分方程式の求解の練習問題として、初期条件$y(x = 0) = 1$の下で次の微分方程式を数値的に解くことに挑戦してみよう:

$$y'(x) = -y(x). \tag{19}$$

また、この微分方程式の解は$y(x) = e^{-x}$という指数関数である。

各自、微分方程式を解くためのコードを自作し、刻み幅hを変えることで得られる解の精度がどのように変化するかを確認せよ。以下に、Pythonコードの例を示す。

ソースコード 15: Euler法による微分方程式の求解

```python
from matplotlib import pyplot as plt
import numpy as np

# compute dy/dx
def dydx(x,y):
    return -y

# compute y(x+h)
def euler_method(y,x,h):
    return y+dydx(x,y)*h


xi = 0.0
xf = 5.0
n = 10

h = (xf-xi)/n

# initial condition
x = xi
y = 1.0


x_j = np.zeros((n+1))
y_euler = np.zeros((n+1))
x_j[0] = x
y_euler[0] = y

for i in range(n):
    x = xi + i*h
    y = euler_method(y_euler[i],x,h)
    x_j[i+1] = x+h
    y_euler[i+1] = y


# plot
plt.plot(x_j, y_euler, label="Euler␣method")
plt.plot(x_j, np.exp(-x_j), label="Exact", linestyle='dashed')

plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.savefig("result_Euler.png")
plt.show()
```

上記のソースコード 15では、defユーザー関数の定義を行っている。このようにdef文によって新たな関数を定義することで、同じ手続きを繰り返す場合にプログラムを簡単に書くことができる。関数の定義は、次のような文によって行うことができる。
def 関数名(引数1, 引数2, 引数3, ···): さらに、関数によって行う手続きをインデント(字下げ)した領域に書き、最後にreturn文で、定義した関数を終了し、元のプログラムへ戻る。この時、return文の後に関数の**戻り値**として関数の出力を記述することができる。

上記のソースコード 15を実行した結果得られる図を以下に示す。

4.2 Heun法による求解

前節では最も簡単な求解としてEuler法について述べた。しかし、Euler法は精度の高い計算を実行することが難しく、実際の応用で用いられることは多くない。ここでは、Euler法より精度が高い、比較的簡単な求解法として**Heun法(ホイン法)**について述べる。

前節では、Euler法を前進差分近似に基づいて導出した。ここでは、前進差分公式よりも精度の高い中心差分公式(7)に基づいて数値解法を導くことを考える。まず、中心差分公式を用いることで、次のような近似公式を得ることが出来る。

$$\frac{dy\left(x+\frac{h}{2}\right)}{dx} \approx \frac{y(x+h)-y(x)}{h}. \tag{20}$$

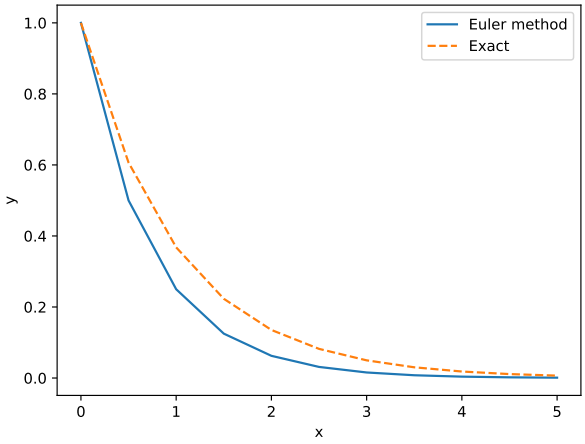

Figure 4: Euler法を用いた微分方程式の求解の例。

さらに、Euler法の場合と同様に、次のような書き換えを行う。

$$y(x+h) \approx y(x) + \frac{dy\left(x+\frac{h}{2}\right)}{dx}h. \tag{21}$$

ここで、微分値$\frac{dy\left(x+\frac{h}{2}\right)}{dx}$を$x+h$と$x$における微分値の平均で置き換え、さらに元の微分方程式(15)を用いると、次のような関係式を得ることが出来る。

$$y(x+h) \approx y(x) + h\frac{f\left(x+h, y(x+h)\right) + f\left(x, y(x)\right)}{2}. \tag{22}$$

この関係式は、Euler法よりも高精度な近似公式となっているが、実際の計算で用いるのは難しい。なぜなら、左辺の$y(x+h)$を評価するためには、右辺を計算する際に既に$x+h$における関数の値$y(x+h)$の情報が必要となるからである。そこで、この関係式を近似的に評価するため、次のような二段階の手続きを踏む。

まず、第一段階目の手続きでは$x+h$における$y(x)$の近似値をEuler法を使って次のように求める。

$$\bar{y}(x+h) = y(x) + hf\left(x, y(x)\right). \tag{23}$$

第二段階目の手続きで、前段で得られた近似値$\bar{y}(x+h)$を用いて、式(22)の右辺を評価し、$y(x+h)$を次のように求める。

$$y(x+h) = y(x) + h\frac{f\left(x+h, \bar{y}(x+h)\right) + f\left(x, y(x)\right)}{2}. \tag{24}$$

このような式(23)と式(24)を用いた2段階の手続きで微分方程式を球解する方法をHeun法と呼ぶ。Heun法の精度とEuler法の精度を比較するために、前節でEuler法を用いて解いた微分方程式(19)をHeun法によって解き、結果を比較してみよう。

以下には、Euler法のサンプルコード15を拡張し、Heun法の計算を追加したPythonコードを示す。

ソースコード 16: Heun法とEuler法による微分方程式の球解

```
1 from matplotlib import pyplot as plt
2 import numpy as np
3
```

```
# compute dy/dx
def dydx(x,y):
    return -y

# compute y(x+h)
def euler_method(y,x,h):
    return y+dydx(x,y)*h

# compute y(x+h)
def heun_method(y,x,h):
    y_bar = y+dydx(x,y)*h
    return y+0.5*h*(dydx(x,y)+dydx(x+h,y_bar))

xi = 0.0
xf = 5.0
n = 10

h = (xf-xi)/n

# initial condition
x = xi
y = 1.0


x_j = np.zeros((n+1))
y_euler = np.zeros((n+1))
y_heun = np.zeros((n+1))

x_j[0] = x
y_euler[0] = y
y_heun[0] = y

for i in range(n):
    x = xi + i*h
    y_euler[i+1] = euler_method(y_euler[i],x,h)
    y_heun[i+1] = heun_method(y_heun[i],x,h)
    x_j[i+1] = x+h



# plot
plt.plot(x_j, y_euler, label="Euler method")
plt.plot(x_j, y_heun, label="Heun method", linestyle='dashed')
plt.plot(x_j, np.exp(-x_j), label="Exact", linestyle='dotted')

plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.savefig("result_Heun.png")
plt.show()
```

上記のソースコード 16を実行した結果得られる図を以下に示す。

4.3 Runge-Kutta法による求解

Euler法やHeun法よりも高精度な球解法としてRunge-Kutta法がある。特に、4次のRunge-Kutta法は微分方程式の求解によく用いられる方法である。4次のRunge-Kutta法は次のような多段階の過程を経て、微分方程式

$$\frac{dy(x)}{dx} = f\left(x, y(x)\right) \tag{25}$$

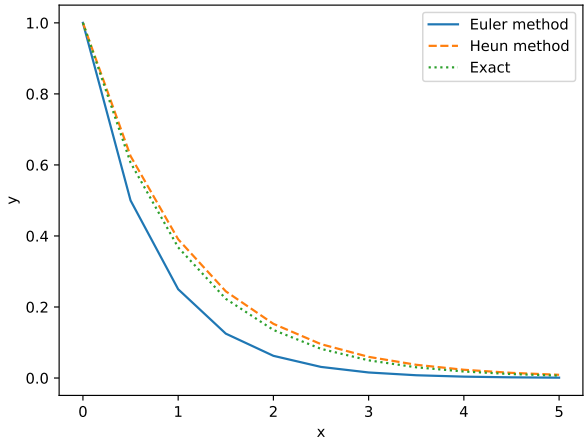

Figure 5: Heun法を用いた微分方程式の求解の例。

を解く:

$$k_1 = f\left(x, y(x)\right) \tag{26}$$

$$k_2 = f\left(x + \frac{h}{2}, y(x) + \frac{h}{2}k_1\right) \tag{27}$$

$$k_3 = f\left(x + \frac{h}{2}, y(x) + \frac{h}{2}k_2\right) \tag{28}$$

$$k_4 = f\left(x + h, y(x) + hk_3\right) \tag{29}$$

$$y(x+h) \approx y(x) + h\frac{k_1 + 2k_2 + 2k_3 + k_4}{6}. \tag{30}$$

Runge-Kutta法を用いて微分方程式(19)を解くプログラムを作成し、Euler法やHeun法の結果と比較して精度がどのように変化するか確認してみよう。

5　2階常微分方程式の求解

この節では、次のような形式で書くことが出来る1変数の2階常微分方程式を数値的に解く方法について学ぶ。

$$\frac{d^2y(x)}{dx^2} = f\left(x, y(x), \frac{dy(x)}{dx}\right). \tag{31}$$

このような2階常微分方程式の中には、Newtonの運動方程式($m\frac{d^2x(t)}{dt^2} = F(t)$)などが含まれる。

2階の常微分方程式を解く方法にはいくつかの種類があるが、ここでは補助変数$s(x) = \frac{dy(x)}{dx}$を導入することで、式(31)を次のような2変数の連立1階常微分方程式に書き換える。

$$\frac{ds(x)}{dx} = f\left(x, y(x), s(x)\right), \tag{32}$$

$$\frac{dy(x)}{dx} = s(x). \tag{33}$$

このような2変数の連立1階常微分方程式は、前節 4で述べた1階の常微分方程式の解法を用いることで数値的に解くことが出来る。

ここでは、例として次のような2階の常微分方程式を解くプログラムを書きながら、常微分方程式の数値解法について学んでいこう。

$$\frac{d^2x(t)}{dt} = -x(t). \tag{34}$$

ここで、初期条件は$x(0) = 0, \dot{x}(0) = 1$とする。

補助変数$v(t) = \frac{dx(t)}{dt}$を導入することで、式(34)を次のような連立微分方程式に書き換える。

$$\frac{dx(t)}{dt} = v(t), \tag{35}$$

$$\frac{dv(t)}{dt} = -x(t). \tag{36}$$

さらに、初期条件$x(0) = 0, \dot{x}(0) = 1$は、$x(0) = 0, v(0) = 1$と書き換えることが出来る。なお、この微分方程式の解は$x(t) = \sin(t), v(t) = \cos(t)$である。

以下のソースコードは、連立1次微分方程式,式(35)と式(36)をEuler法を用いて解くPythonコードである。このソースコードを参考に、連立微分方程式(35-36)を解くコードを自作してみよう。

ソースコード 17: 連立1次微分方程式をEuler法解くサンプルコード

```
1  from matplotlib import pyplot as plt
2  import numpy as np
3
4  def dxdt(v):
5      return v
6
7  def dvdt(x):
8      return -x
9
10 def euler_method(x,v,dt):
11     x_updated = x + dt*dxdt(v)
12     v_updated = v + dt*dvdt(x)
13     return x_updated, v_updated
14
15 ti = 0.0
16 tf = 15.0
17 n = 45
18 dt = (tf - ti)/n
19
20 # Initial conditions
21 x = 1.0
22 v = 0.0
```

```

tt = np.zeros(n+1)
xt = np.zeros(n+1)
vt = np.zeros(n+1)

tt[0] = ti
xt[0] = x
vt[0] = v

for j in range(n):
    x, v = euler_method(x,v,dt)
    xt[j+1] = x
    vt[j+1] = v
    tt[j+1] = ti + (j+1)*dt


# Plot the results
plt.plot(tt, xt, label='x(t)')
plt.plot(tt, vt, label='v(t)')
plt.plot(tt, np.cos(tt), label='x(t): Exact', linestyle='dashed')
plt.plot(tt, -np.sin(tt), label='v(t): Exact', linestyle='dashed')

plt.xlabel('t')
plt.ylabel('x,v')
plt.legend()
plt.savefig("result_Euler_2nd.png")
plt.show()
```

以下の図 6には、ソースコード 17を実行した結果得られる数値解と厳密解の比較が示されている。Euler法による数値解が、時間tの経過とともに厳密解から大きく離れていっている様子が確認できる。

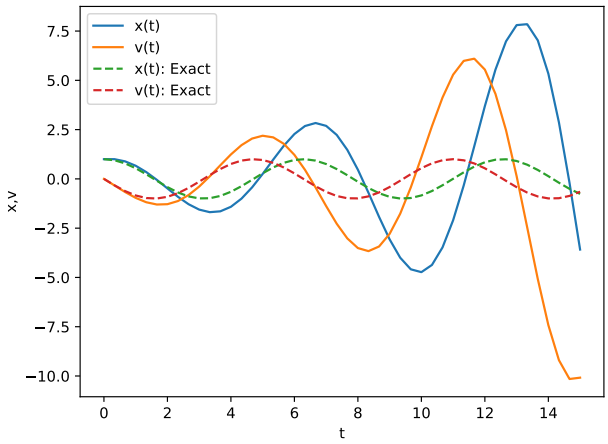

Figure 6: Euler法を用いて連立微分方程式(35-36)を解いた結果と厳密解の比較。

前節 4.1で学んだように、Heun法やRunge-Kutta法を用いて数値解の精度を高めることが出来る。以下のサンプルソースコード 18には、Heun法によって連立微分方程式(35-36)を解くコードの例を示した。また、Figure 7には、その結果得られる数値解と厳密解の比較を示した。Heun法を用いることで、Euler法に比べて精度を向上させることが出来る。

ソースコード 18: 連立1次微分方程式をHeun法で解くサンプルコード

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

def dxdt(v):
    return v

```

```
def dvdt(x):
    return -x

#def euler_method(x,v,dt):
#    x_updated = x + dt*dxdt(v)
#    v_updated = v + dt*dvdt(x)
#    return x_updated, v_updated

def heun_method(x,v,dt):
# first step
    x_bar = x + dt*dxdt(v)
    v_bar = v + dt*dvdt(x)

# second step
    x_updated = x + 0.5*dt*(dxdt(v)+dxdt(v_bar))
    v_updated = v + 0.5*dt*(dvdt(x)+dvdt(x_bar))
    return x_updated, v_updated

ti = 0.0
tf = 15.0
n = 45
dt = (tf - ti)/n

# Initial conditions
x = 1.0
v = 0.0


tt = np.zeros(n+1)
xt = np.zeros(n+1)
vt = np.zeros(n+1)

tt[0] = ti
xt[0] = x
vt[0] = v

for j in range(n):
    x, v = heun_method(x,v,dt)
    xt[j+1] = x
    vt[j+1] = v
    tt[j+1] = ti + (j+1)*dt


# Plot the results
plt.plot(tt, xt, label='x(t)')
plt.plot(tt, vt, label='v(t)')
plt.plot(tt, np.cos(tt), label='x(t): Exact', linestyle='dashed')
plt.plot(tt, -np.sin(tt), label='v(t): Exact', linestyle='dashed')

plt.xlabel('t')
plt.ylabel('x,v')
plt.legend()
plt.savefig("result_Heun_2nd.png")
plt.show()
```

各自、Heun法やRunge-Kutta法を用いて連立微分方程式(35-36)を解くプログラムを自作し、計算精度を確認してみよう。

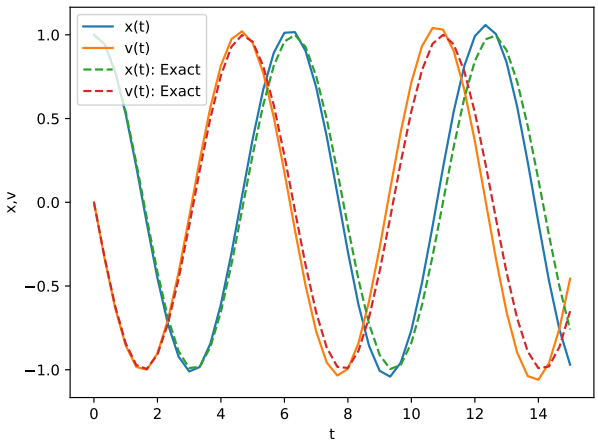

Figure 7: Heun法を用いて連立微分方程式(35-36)を解いた結果と厳密解の比較。

6　1次元系の量子ダイナミクスシミュレーション

この節では、時間依存Schrödinger方程式を数値的に解いて、1次元の波束の伝搬のシミュレーションに取り組む。時間に依存しないポテンシャル$V(x)$の下での粒子の運動は、次のようなSchrödinger方程式によって記述される。

$$i\hbar\frac{\partial}{\partial t}\psi(x,t) = \left[-\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2} + V(x)\right]\psi(x,t) = \hat{H}\psi(x,t). \tag{37}$$

ただし、ハミルトニアンHは次のように定義されている。

$$\hat{H} = -\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2} + V(x) \tag{38}$$

また、しばらくの間、境界条件$\psi(x=-L/2,t)=\psi(x=L/2,t)=0$を課すことにしよう。

このシュレディンガー方程式を、時刻$t=0$において初期条件

$$\psi(x,t=0) = e^{-\frac{(x-x_0)^2}{2\sigma_0^2}}e^{ik_0(x-x_0)} \tag{39}$$

の下で解くことを考える。ここでx_0は波束の中心位置、k_0は波束の中心波数、σ_0は波束の広がり幅である。

6.1　実空間法

コンピュータ上で波動関数を表すために、次のような空間座標の離散化を考えよう。

$$x \to x_j = -\frac{L}{2} + \Delta x \times (j+1) \qquad (-1 \le j \le N). \tag{40}$$

ただし、$\Delta x = L/(N+1)$である。このようにして用意した$(N+1)$個の点$\{x_{-1}, x_0, x_1, \cdots, x_N\}$における波動関数$\psi(x,t)$の値を、次のように表すこととしよう。

$$\psi_j(t) = \psi(x_j, t). \tag{41}$$

このように実空間において有限個の点の上で関数を取り扱う手法は**実空間法**と呼ばれる。実空間法による波動関数の記述に慣れるために、初期条件(式(39))で表される波動関数を図示してみよう。このとき、x_0, k_0, σ_0の値を自由に設定し、波動関数の実部と虚部がどのように変化するか確認してみよう。

以下に、可視化のためのサンプルソースコードを示す。各自、コードを自作して、波動関数の可視化に取り組んでみよう。

`https://github.com/shunsuke-sato/python_qe/blob/develop/note_comp_phys/src/qm_write_init_wf.py`

ソースコード 19: 初期波動関数の可視化

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

# Initialize the wavefunction
def initialize_wf(xj, x0, k0, sigma0):
    wf = np.exp(1j*k0*(xj-x0))*np.exp(-0.5*(xj-x0)**2/sigma0**2)
    return wf



# initial wavefunction parameters
x0 = -25.0
k0 = 0.85
sigma0 = 5.0

```

```
# set the coordinate
xmin = -100.0
xmax = 100.0
n = 2500

dx = (xmax-xmin)/(n+1)
xj = np.zeros(n)

for i in range(n):
    xj[i] = xmin + dx*(i+1)


# Initialize the wavefunction
wf = initialize_wf(xj, x0, k0, sigma0)

# Plot the results
plt.plot(xj, np.real(wf), label="Real part")
plt.plot(xj, np.imag(wf), label="Imaginary part")

plt.xlabel('x')
plt.ylabel('$\psi$(x)')
plt.legend()
plt.savefig("init_wf.png")
plt.show()
```

上記のソースコードでは、パラメータをそれぞれ$x_0 = -25$, $k_0 = 0.85$, $\sigma_0 = 25.0$と設定し波動関数の可視化を行っている。また、実空間法を用いて範囲$(-100 \leq x \leq 100)$を$N = 2500$点のgrid点に分割して可視化を行っている。

ソースコード 19を実行することで、Figure 8に示されるように波動関数を可視化することが出来る。物理パラメータx_0, k_0, σ_0や分割数Nを変えながら、可視化された波動関数の様子を観察してみよう。

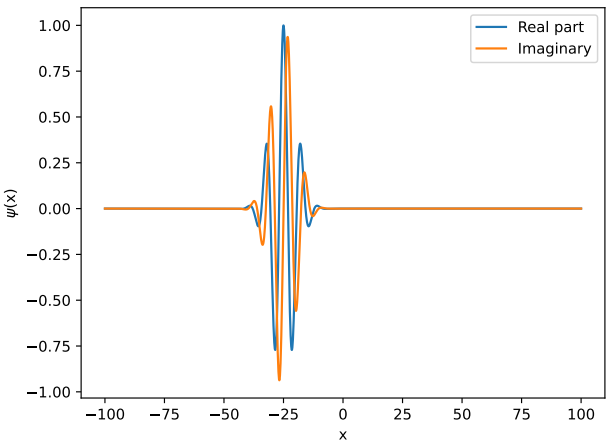

Figure 8: 初期波動関数の実部と虚部。

6.2 実時間法

次に、上記の実空間法で用意した有限個の離散的な点$\{x_j\}$の上で定義された波動関数の時間発展を計算してみよう。これを実行するために、まずはSchrödinger方程式(37)の形式的な解について考えてみよう。式(37)のSchrödinger方程式のハミルトニアンは時間に依存していないことに注意すると、次のような形でSchrödinger方程式の形式的な解を書き下すことが出来る。

$$\psi(x,t) = \exp\left[-\frac{i}{\hbar}\hat{H}t\right]\psi(x,0). \tag{42}$$

ただし、ここで一般の演算子Aの指数関数は次のようにTaylor展開により定義されている。

$$\exp\left[\hat{A}\right] = \mathcal{I} + \sum_{n=1}^{\infty} \frac{\hat{A}^n}{n!}. \tag{43}$$

ここで$\mathcal{I}$は恒等演算子を表す。

時間発展演算子$\hat{U}(t,0) = \exp\left[-\frac{i}{\hbar}\hat{H}t\right]$を数値的に評価することが出来れば、数値計算を用いて波動関数の時間発展を計算することが出来る。このような演算子の指数関数はいくつかの方法で評価できるが、ここでは時間発展演算子を微小時間発展演算子の積として記述する方法を採用する。すなわち、時間発展演算子$\hat{U}(t,0)$を次のような微小時間発展演算子の積として表す。

$$\hat{U}(t,0) = \exp\left[-i\frac{t}{\hbar}\hat{H}\right] = \exp\left[-i\frac{\Delta t}{\hbar}\hat{H}\right] \times \cdots \exp\left[-i\frac{\Delta t}{\hbar}\hat{H}\right] = \left[\exp\left[-i\frac{\Delta t}{\hbar}\hat{H}\right]\right]^{N_t}. \tag{44}$$

ただし、ここでは微小時間Δtは$\Delta t = t/N_t$によって定義されている。

式(44)を用いることで、波動関数の時間発展を微小時間Δtの時間発展の繰り返しによって表現することが出来る。次に、微小時間発展演算子,$\exp\left[-i\frac{\Delta t}{\hbar}\hat{H}\right]$,を数値計算によりどのように評価するかについて考えてみよう。演算子の指数関数が式(43)により定義されていることに注意すると、微小時間発展演算子は次のようなTalor展開の形で書き下すことが出来る。

$$\exp\left[-i\frac{\Delta t}{\hbar}\hat{H}\right] = \mathcal{I} + \sum_{n=1}^{\infty} \frac{(-i\Delta t/\hbar)^n}{n!}\hat{H}^n. \tag{45}$$

式(45)の微小時間発展演算子の展開は厳密な展開である。ここで、微小時間Δtが十分小さければ、式(45)の展開を有限の項で打ち切っても、微小時間発展演算子の十分正確な近似表現となると考えられる。したがって、$N_{\exp}$を有限な整数として、微小時間発展演算子を次のように近似する。

$$\exp\left[-i\frac{\Delta t}{\hbar}\hat{H}\right] \approx \mathcal{I} + \sum_{n=1}^{N_{\exp}} \frac{(-i\Delta t/\hbar)^n}{n!}\hat{H}^n. \tag{46}$$

式(46)を用いて、具体的に波動関数の時間発展方程式を書き下すと、以下の近似式を得ることが出来る。

$$\begin{aligned}\psi(x,t+\Delta t) &= \exp\left[-i\frac{\Delta t}{\hbar}\hat{H}\right]\psi(x,t) \\ &\approx \psi(x,t) + \sum_{n=1}^{N_{\exp}} \frac{(-i\Delta t/\hbar)^n}{n!}\hat{H}^n\psi(x,t).\end{aligned} \tag{47}$$

すなわち、時刻tの波動関数$\psi(x,t)$にハミルトニアンを複数回作用させて適当な定数を掛けて元の波動関数$\psi(x,t)$に足していくことで、次の時刻の波動関数を$\psi(x,t+\Delta t)$を近似的に得ることが出来る。この近似は、Δtの値が小さいほど正確となり、また打ち切りの上限値$N_{\exp}$の大きいほど正確となる。実用上、多くの計算では$N_{\exp} = 4$とし、十分小さなΔtを用いて時間発展計算を行うことが多い。

ここからは、上記で説明したアルゴリズムを用いて量子波束の時間発展を計算するためのコードを書いていこう。波動関数の時間発展は、ハミルトニアンを波動関数に作用させる操作を多数回繰り返すことで実現できるので、まずは波動関数にハミルトニアンを作用させるコードを書いてみよう。以下のソースコード 20に、コードの例を示す。

`https://github.com/shunsuke-sato/python_qe/blob/develop/note_comp_phys/src/qm_ham_write.py`

ソースコード 20: 波動関数にハミルトニアンを作用させるコードの例

```python
from matplotlib import pyplot as plt
import numpy as np

# Initialize the wavefunction
def initialize_wf(xj, x0, k0, sigma0):
    wf = np.exp(1j*k0*(xj-x0))*np.exp(-0.5*(xj-x0)**2/sigma0**2)
    return wf


# Operate the Hamiltonian to the wavefunction
def ham_wf(wf, vpot, dx):

    n = wf.size
    hwf = np.zeros(n, dtype=complex)

    for i in range(1,n-1):
        hwf[i] = -0.5*(wf[i+1]-2.0*wf[i]+wf[i-1])/(dx**2)

    i = 0
    hwf[i] = -0.5*(wf[i+1]-2.0*wf[i])/(dx**2)
    i = n-1
    hwf[i] = -0.5*(-2.0*wf[i]+wf[i-1])/(dx**2)

    hwf = hwf + vpot*wf

    return hwf


# initial wavefunction parameters
x0 = -25.0
k0 = 0.85
sigma0 = 5.0

# set the coordinate
xmin = -100.0
xmax = 100.0
n = 2500

dx = (xmax-xmin)/(n+1)
xj = np.zeros(n)

for i in range(n):
    xj[i] = xmin + dx*(i+1)


# Initialize the wavefunction
wf = initialize_wf(xj, x0, k0, sigma0)
vpot = np.zeros(n)

hwf = ham_wf(wf, vpot, dx)

# Plot the results
plt.plot(xj, np.real(wf), label="Real␣part␣(wf)")
plt.plot(xj, np.imag(wf), label="Imaginary␣part␣(wf)")
plt.plot(xj, np.real(hwf), label="Real␣part␣(ham␣wf)")
plt.plot(xj, np.imag(hwf), label="Imaginary␣part␣(ham␣wf)")

plt.xlabel('x')
plt.ylabel('$\psi$(x)')
plt.legend()
plt.savefig("ham_wf.png")
plt.show()
```

上記のソースコード 20を実行した結果、Figure 9のような図が得られる。ここでは、初期波動関数とそれにハミルトニアンを作用させて得られた波動関数の実部と虚部が示されている。

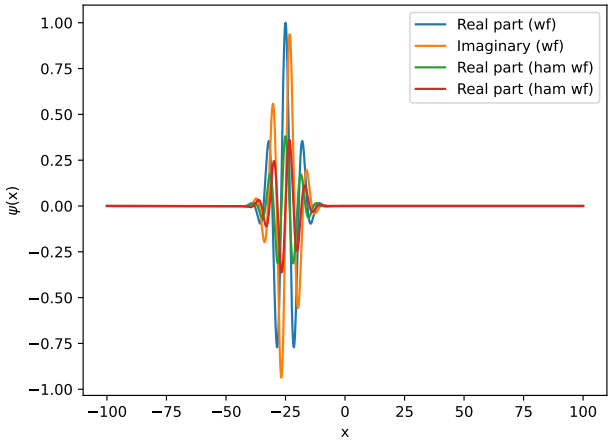

Figure 9: ハミルトニアンが作用された波動関数の可視化の様子。

次に、微小時間発展の方程式(47)を繰り返し用いて、波動関数の時間発展を計算するコードを書いてみよう。上記の例題(ソースコード 19)で作った、ハミルトニアンを波動関数をに作用させる関数を用いて、波動関数の微小時間発展を実行する。

ソースコード 21に、波動関数の時間発展を計算するコードの例を示す。

`https://github.com/shunsuke-sato/python_qe/blob/develop/note_comp_phys/src/qm_dynamics.py`

ソースコード 21: 1次元量子波束の時間発展計算

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

# Initialize the wavefunction
def initialize_wf(xj, x0, k0, sigma0):
    wf = np.exp(1j*k0*(xj-x0))*np.exp(-0.5*(xj-x0)**2/sigma0**2)
    return wf


# Operate the Hamiltonian to the wavefunction
def ham_wf(wf, vpot, dx):

    n = wf.size
    hwf = np.zeros(n, dtype=complex)

    for i in range(1,n-1):
        hwf[i] = -0.5*(wf[i+1]-2.0*wf[i]+wf[i-1])/(dx**2)

    i = 0
    hwf[i] = -0.5*(wf[i+1]-2.0*wf[i])/(dx**2)
    i = n-1
    hwf[i] = -0.5*(-2.0*wf[i]+wf[i-1])/(dx**2)

    hwf = hwf + vpot*wf

    return hwf


# Time propagation from t to t+dt
def time_propagation(wf, vpot, dx, dt):

    n = wf.size
    twf = np.zeros(n, dtype=complex)
    hwf = np.zeros(n, dtype=complex)

    twf = wf
    zfact = 1.0 + 0j
    for iexp in range(1,5):
```

```
        zfact = zfact*(-1j*dt)/iexp
        hwf = ham_wf(twf, vpot, dx)
        wf = wf + zfact*hwf
        twf = hwf

    return wf


# initial wavefunction parameters
x0 = -25.0
k0 = 0.85
sigma0 = 5.0

# time propagation parameters
#Tprop = 80.0
Tprop = 8.0
dt = 0.005
nt = int(Tprop/dt)+1

# set the coordinate
xmin = -100.0
xmax = 100.0
n = 2500

dx = (xmax-xmin)/(n+1)
xj = np.zeros(n)

for i in range(n):
    xj[i] = xmin + dx*(i+1)


# Initialize the wavefunction
wf = initialize_wf(xj, x0, k0, sigma0)
vpot = np.zeros(n)


# for loop for the time propagation
for it in range(nt+1):
    wf = time_propagation(wf, vpot, dx, dt)
    print(it, nt)

# Plot the results
plt.plot(xj, np.real(wf), label="Real␣part␣(wf)")
plt.plot(xj, np.imag(wf), label="Imaginary␣part␣(wf)")


plt.xlabel('x')
plt.ylabel('$\psi$(x)')
plt.legend()
plt.savefig("fin_wf.pdf")
plt.show()
```

上記のソースコード 21を実行することで、Figure 8に示されている初期波動関数から時間発展した結果として、Figure 10に示されるような波動関数が表示される。この波動関数のふるまいから、自由空間における量子波束の時間発展について考察してみよう。

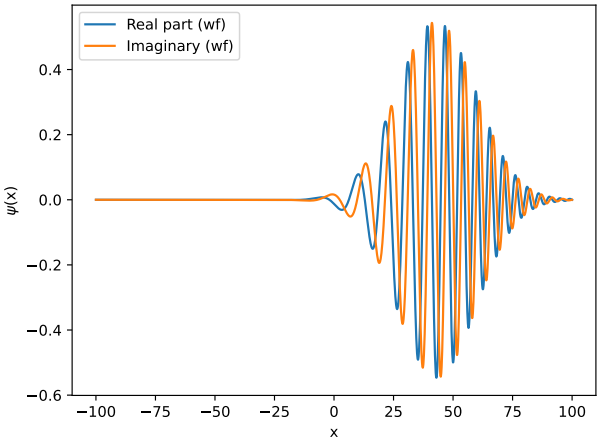

Figure 10: 時間発展した波動関数の様子。

6.2.1 Numbaを使ったPythonコードの高速化

上記のコードの実行には、しばらく時間がかかったかもしれない。一般的に、Pythonは実行速度がほかの言語と比較して遅い場合が多い。しかし、いくつかの処理を行うことで、Pythonコードの実行速度を改善することができる。ここでは、*Numba*のJIT(Just in Time)コンパイル機能を用いたコードの高速化を行う。

Numbaのjitの利用は、非常に簡単であり、プログラム文頭で`from numba import jit`により、NUmbaのjitを利用可能にするための準備を行う。次に、高速化したい関数の直上に`@jit(nopytohn=True)`と記述する。これにより、jitのコンパイラによって、指定された関数がコンパイルされ、Pythonコードが高速に実行される。

例として、量子波束のシミュレーションコード 21をNumbaのjitにより高速化してみよう。Numbaによって高速化されたコードの例を以下に示す。ただし、以下のコードでは関数`ham_wf`が高速化されており、また複素数配列の初期化の際の配列の型を`np.complex128`により、より厳密に定義してnumbaによる高速化を可能としている。

`https://github.com/shunsuke-sato/python_qe/blob/develop/note_comp_phys/src/qm_dynamics_numba.py`

ソースコード 22: 1次元量子波束の時間発展計算(Numbaによる高速化)

```
from numba import jit
from matplotlib import pyplot as plt
import numpy as np

# Initialize the wavefunction
def initialize_wf(xj, x0, k0, sigma0):
    wf = np.exp(1j*k0*(xj-x0))*np.exp(-0.5*(xj-x0)**2/sigma0**2)
    return wf


# Operate the Hamiltonian to the wavefunction
@jit(nopython=True)
def ham_wf(wf, vpot, dx):

    n = wf.size
    hwf = np.zeros(n, dtype=np.complex128)

    for i in range(1,n-1):
        hwf[i] = -0.5*(wf[i+1]-2.0*wf[i]+wf[i-1])/(dx**2)

    i = 0
    hwf[i] = -0.5*(wf[i+1]-2.0*wf[i])/(dx**2)
    i = n-1
```

```
    hwf[i] = -0.5*(-2.0*wf[i]+wf[i-1])/(dx**2)

    hwf = hwf + vpot*wf

    return hwf


# Time propagation from t to t+dt
def time_propagation(wf, vpot, dx, dt):

    n = wf.size
    twf = np.zeros(n, dtype=complex)
    hwf = np.zeros(n, dtype=complex)

    twf = wf
    zfact = 1.0 + 0j
    for iexp in range(1,5):
        zfact = zfact*(-1j*dt)/iexp
        hwf = ham_wf(twf, vpot, dx)
        wf = wf + zfact*hwf
        twf = hwf

    return wf


# initial wavefunction parameters
x0 = -25.0
k0 = 0.85
sigma0 = 5.0

# time propagation parameters
#Tprop = 80.0
Tprop = 8.0
dt = 0.005
nt = int(Tprop/dt)+1

# set the coordinate
xmin = -100.0
xmax = 100.0
n = 2500

dx = (xmax-xmin)/(n+1)
xj = np.zeros(n)

for i in range(n):
    xj[i] = xmin + dx*(i+1)


# Initialize the wavefunction
wf = initialize_wf(xj, x0, k0, sigma0)
vpot = np.zeros(n)


# for loop for the time propagation
for it in range(nt+1):
    wf = time_propagation(wf, vpot, dx, dt)
    print(it, nt)

# Plot the results
plt.plot(xj, np.real(wf), label="Real␣part␣(wf)")
plt.plot(xj, np.imag(wf), label="Imaginary␣part␣(wf)")


plt.xlabel('x')
plt.ylabel('$\psi$(x)')
plt.legend()
plt.savefig("fin_wf.pdf")
plt.show()
```

6.3 1次元量子波束のダイナミクスの動画作成

上記のサンプルソースコード 21では、時間発展計算の終時刻における波動関数のみが図として出力された。ここでは、初期時刻から終時刻まで時々刻々と時間発展する量子波束の様子を可視化するために、シミュレーション結果を動画にする方法について学ぶ。

シミュレーション結果を動画にするためには、パラパラ漫画の要領で多数の画像ファイル

を用意し、これらの画像ファイルを束ねることで動画にする方法がよく用いられる。以下のソースコード 23では、上記のソースコード 21を発展させ、時間発展計算の途中で、その時刻における波動関数のの情報を保存し、最後にこれらの波動関数の図を生成・結合して動画として出力している。

https://github.com/shunsuke-sato/python_qe/blob/develop/note_comp_phys/src/qm_dynamics_matplotlib.py

ソースコード 23: 量子波束ダイナミクスの動画作成用のコード

```
from numba import jit
from matplotlib import pyplot as plt
import numpy as np
import matplotlib.animation as animation
from matplotlib.animation import PillowWriter

# Initialize the wavefunction
def initialize_wf(xj, x0, k0, sigma0):
    wf = np.exp(1j*k0*(xj-x0))*np.exp(-0.5*(xj-x0)**2/sigma0**2)
    return wf


# Operate the Hamiltonian to the wavefunction
@jit(nopython=True)
def ham_wf(wf, vpot, dx):

    n = wf.size
    hwf = np.zeros(n, dtype=np.complex128)

    for i in range(1,n-1):
        hwf[i] = -0.5*(wf[i+1]-2.0*wf[i]+wf[i-1])/(dx**2)

    i = 0
    hwf[i] = -0.5*(wf[i+1]-2.0*wf[i])/(dx**2)
    i = n-1
    hwf[i] = -0.5*(-2.0*wf[i]+wf[i-1])/(dx**2)

    hwf = hwf + vpot*wf

    return hwf


# Time propagation from t to t+dt
def time_propagation(wf, vpot, dx, dt):

    n = wf.size
    twf = np.zeros(n, dtype=complex)
    hwf = np.zeros(n, dtype=complex)

    twf = wf
    zfact = 1.0 + 0j
    for iexp in range(1,5):
        zfact = zfact*(-1j*dt)/iexp
        hwf = ham_wf(twf, vpot, dx)
        wf = wf + zfact*hwf
        twf = hwf

    return wf


# initial wavefunction parameters
x0 = -25.0
k0 = 0.85
sigma0 = 5.0

# time propagation parameters
Tprop = 80.0
dt = 0.005
#dt = 0.00905
nt = int(Tprop/dt)+1

# set the coordinate
xmin = -100.0
xmax = 100.0
n = 2500

```

```
dx = (xmax-xmin)/(n+1)
xj = np.zeros(n)

for i in range(n):
    xj[i] = xmin + dx*(i+1)


# Initialize the wavefunction
wf = initialize_wf(xj, x0, k0, sigma0)
vpot = np.zeros(n)

# For loop for the time propagation
wavefunctions = []
for it in range(nt+1):
    if (it % (nt//100) == 0):
        wavefunctions.append(wf.copy())

    wf = time_propagation(wf, vpot, dx, dt)
    print(it, nt)

# Define function to update plot for each frame of the animation
def update_plot(frame):
    plt.cla()
    plt.xlim([-100, 100])
    plt.ylim([-1.2, 1.2])
    plt.plot(xj, np.real(wavefunctions[frame]), label="Real part of $\psi(x)$")
    plt.plot(xj, np.imag(wavefunctions[frame]), label="Imaginary part of $\psi(x)$")
    plt.xlabel('$x$')
    plt.ylabel('$\psi(x)$')
    plt.legend()

# Create the animation
fig = plt.figure()
ani = animation.FuncAnimation(fig, update_plot, frames=len(wavefunctions), interval=50)
#ani.save('wavefunction_animation.gif', writer='imagemagick')
ani.save('wavefunction_animation.gif', writer='pillow')
```

6.4 1次元量子波束の様々なダイナミクス

ここまでに作成した1次元量子波束ダイナミクスのシミュレーションコードを利用して、様々な問題における1次元波束の運動を調べてみよう。

6.4.1 トンネル現象

ここでは、トンネル現象に関する1次元波束のダイナミクスに関して、参考となる情報を記す。上記のコードではポテンシャルエネルギー$V(x)$をゼロとして計算を行っていたが、ここでは次のようなガウス型のポテンシャルの下での波束のダイナミクスを考えてみよう:

$$V(x) = V_0 e^{-\frac{x^2}{2\sigma_v}}. \tag{48}$$

例えば、$V_0 = 0.735$ a.u.、$\sigma_v = 0.5$ a.u.とし、さらに初期波動関数は式(39)の形のものを用いてみよう。この時、$k_0 = 0.85$ a.u., $x_0 = -25$ a.u., $\sigma_0 = 5$として計算を実行し、波束のダイナミクスの動画を作成してみよう。また、ポテンシャルの高さV_0やポテンシャルの幅σ_vを変えてシミュレーションを実行し、トンネル現象に慣れ親しんでみよう。また、ソースコード 24を参考にしてみよう。

6.4.2 調和ポテンシャルのコヒーレント状態

ここでは、調和ポテンシャル中での量子波束のダイナミクスに関して、参考となる情報を記す。調和ポテンシャルは下記のような二次関数型のポテンシャルである。

$$V(x) = \frac{K}{2}x^2. \tag{49}$$

ここでKはバネ定数であり、試しに$K = 1$ a.u.として計算を実行してみよう。また、初期波動関数としては、式(39)を用いてみよう。この時、$k_0 = 0$ a.u., $x_0 = -2$ a.u., $\sigma_0 = 1$として計算を

実行し、波束のダイナミクスの動画を作成してみよう。また、ソースコード 25を参考にしてみよう。

6.4.3 非調和ポテンシャル

上記の6.4.2 節において、ポテンシャルに非調和項を加え、量子波束のダイナミクスがどのように影響されるかを調べてみよう。例えば、次のようなポテンシャルの下での波束のダイナミクスを調べてみよう。

$$V(x) = \frac{K}{2}x^2 + 0.01x^4. \tag{50}$$

また、ソースコード 26を参考にしてみよう。
https://github.com/shunsuke-sato/python_qe/blob/develop/note_comp_phys/src/qm_dynamics_tunnel.py

ソースコード 24: 量子波束のトンネル現象に関するコードの例

```
from numba import jit
from matplotlib import pyplot as plt
import numpy as np
import matplotlib.animation as animation
from matplotlib.animation import PillowWriter

# Initialize the wavefunction
def initialize_wf(xj, x0, k0, sigma0):
    wf = np.exp(1j*k0*(xj-x0))*np.exp(-0.5*(xj-x0)**2/sigma0**2)
    return wf

# Initialize potential
def initialize_vpot(xj):
    v0 = 0.735
    sigma = 0.5
    return v0*np.exp(-0.5*(xj/sigma)**2)

# Operate the Hamiltonian to the wavefunction
@jit(nopython=True)
def ham_wf(wf, vpot, dx):

    n = wf.size
    hwf = np.zeros(n, dtype=np.complex128)

    for i in range(1,n-1):
        hwf[i] = -0.5*(wf[i+1]-2.0*wf[i]+wf[i-1])/(dx**2)

    i = 0
    hwf[i] = -0.5*(wf[i+1]-2.0*wf[i])/(dx**2)
    i = n-1
    hwf[i] = -0.5*(-2.0*wf[i]+wf[i-1])/(dx**2)

    hwf = hwf + vpot*wf

    return hwf


# Time propagation from t to t+dt
def time_propagation(wf, vpot, dx, dt):

    n = wf.size
    twf = np.zeros(n, dtype=complex)
    hwf = np.zeros(n, dtype=complex)

    twf = wf
    zfact = 1.0 + 0j
    for iexp in range(1,5):
        zfact = zfact*(-1j*dt)/iexp
        hwf = ham_wf(twf, vpot, dx)
        wf = wf + zfact*hwf
        twf = hwf

    return wf

```

```

# initial wavefunction parameters
x0 = -25.0
k0 = 0.85
sigma0 = 5.0

# time propagation parameters
Tprop = 80.0
dt = 0.005
#dt = 0.00905
nt = int(Tprop/dt)+1

# set the coordinate
xmin = -100.0
xmax = 100.0
n = 2500

dx = (xmax-xmin)/(n+1)
xj = np.zeros(n)

for i in range(n):
    xj[i] = xmin + dx*(i+1)


# Initialize the wavefunction
wf = initialize_wf(xj, x0, k0, sigma0)
#vpot = np.zeros(n)
vpot = initialize_vpot(xj)

# For loop for the time propagation
wavefunctions = []
for it in range(nt+1):
    if (it % (nt//100) == 0):
        wavefunctions.append(wf.copy())

    wf = time_propagation(wf, vpot, dx, dt)
    print(it, nt)

# Define function to update plot for each frame of the animation
def update_plot(frame):
    plt.cla()
    plt.xlim([-100, 100])
    plt.ylim([-1.2, 1.2])
    plt.plot(xj, np.real(wavefunctions[frame]), label="Real part of $\psi(x)$")
    plt.plot(xj, np.imag(wavefunctions[frame]), label="Imaginary part of $\psi(x)$")
    plt.plot(xj, vpot, label="$V(x)$")
    plt.xlabel('$x$')
    plt.ylabel('$\psi(x)$')
    plt.legend()

# Create the animation
fig = plt.figure()
ani = animation.FuncAnimation(fig, update_plot, frames=len(wavefunctions), interval=50)
#ani.save('wavefunction_animation.gif', writer='imagemagick')
ani.save('wavefunction_animation.gif', writer='pillow')
```

https://github.com/shunsuke-sato/python_qe/blob/develop/note_comp_phys/src/qm_dynamics_harmonic.py

ソースコード 25: 調和ポテンシャル中の量子波束ダイナミクスに関するコードの例

```
from numba import jit
from matplotlib import pyplot as plt
import numpy as np
import matplotlib.animation as animation
from matplotlib.animation import PillowWriter

# Initialize the wavefunction
def initialize_wf(xj, x0, k0, sigma0):
    wf = np.exp(1j*k0*(xj-x0))*np.exp(-0.5*(xj-x0)**2/sigma0**2)
    return wf

# Initialize potential
def initialize_vpot(xj):
    k0 = 1.0
    return 0.5*k0*xj**2
```

```
16
17 # Operate the Hamiltonian to the wavefunction
18 @jit(nopython=True)
19 def ham_wf(wf, vpot, dx):
20
21     n = wf.size
22     hwf = np.zeros(n, dtype=np.complex128)
23
24     for i in range(1,n-1):
25         hwf[i] = -0.5*(wf[i+1]-2.0*wf[i]+wf[i-1])/(dx**2)
26
27     i = 0
28     hwf[i] = -0.5*(wf[i+1]-2.0*wf[i])/(dx**2)
29     i = n-1
30     hwf[i] = -0.5*(-2.0*wf[i]+wf[i-1])/(dx**2)
31
32     hwf = hwf + vpot*wf
33
34     return hwf
35
36
37 # Time propagation from t to t+dt
38 def time_propagation(wf, vpot, dx, dt):
39
40     n = wf.size
41     twf = np.zeros(n, dtype=complex)
42     hwf = np.zeros(n, dtype=complex)
43
44     twf = wf
45     zfact = 1.0 + 0j
46     for iexp in range(1,5):
47         zfact = zfact*(-1j*dt)/iexp
48         hwf = ham_wf(twf, vpot, dx)
49         wf = wf + zfact*hwf
50         twf = hwf
51
52     return wf
53
54
55 # initial wavefunction parameters
56 x0 = -2.0
57 k0 = 0.00
58 sigma0 = 1.0
59
60 # time propagation parameters
61 Tprop = 40.0
62 dt = 0.005
63 #dt = 0.00905
64 nt = int(Tprop/dt)+1
65
66 # set the coordinate
67 xmin = -10.0
68 xmax = 10.0
69 n = 250
70
71 dx = (xmax-xmin)/(n+1)
72 xj = np.zeros(n)
73
74 for i in range(n):
75     xj[i] = xmin + dx*(i+1)
76
77
78 # Initialize the wavefunction
79 wf = initialize_wf(xj, x0, k0, sigma0)
80 #vpot = np.zeros(n)
81 vpot = initialize_vpot(xj)
82
83 # For loop for the time propagation
84 wavefunctions = []
85 for it in range(nt+1):
86     if (it % (nt//100) == 0):
87         wavefunctions.append(wf.copy())
88
89     wf = time_propagation(wf, vpot, dx, dt)
90     print(it, nt)
91
92 # Define function to update plot for each frame of the animation
93 def update_plot(frame):
94     plt.cla()
95     plt.xlim([-5, 5])
```

```
    plt.ylim([-1.2, 5.0])
    plt.plot(xj, np.real(wavefunctions[frame]), label="Real part of $\psi(x)$")
    plt.plot(xj, np.imag(wavefunctions[frame]), label="Imaginary part of $\psi(x)$")
    plt.plot(xj, np.abs(wavefunctions[frame])**2, label=" $|\psi(x)|^2$ ")
    plt.plot(xj, vpot, label="$V(x)$")
    plt.xlabel('$x$')
    plt.ylabel('$\psi(x)$')
    plt.legend()

# Create the animation
fig = plt.figure()
ani = animation.FuncAnimation(fig, update_plot, frames=len(wavefunctions), interval=150)
#ani.save('wavefunction_animation.gif', writer='imagemagick')
ani.save('wavefunction_animation.gif', writer='pillow')
```

https://github.com/shunsuke-sato/python_qe/blob/develop/note_comp_phys/src/qm_dynamics_anharmonic.py

ソースコード 26: 非調和ポテンシャル中の量子波束ダイナミクスに関するコードの例

```
from numba import jit
from matplotlib import pyplot as plt
import numpy as np
import matplotlib.animation as animation
from matplotlib.animation import PillowWriter

# Initialize the wavefunction
def initialize_wf(xj, x0, k0, sigma0):
    wf = np.exp(1j*k0*(xj-x0))*np.exp(-0.5*(xj-x0)**2/sigma0**2)
    return wf

# Initialize potential
def initialize_vpot(xj):
    k0 = 1.0
    return 0.5*k0*xj**2+0.01*xj**4

# Operate the Hamiltonian to the wavefunction
@jit(nopython=True)
def ham_wf(wf, vpot, dx):

    n = wf.size
    hwf = np.zeros(n, dtype=np.complex128)

    for i in range(1,n-1):
        hwf[i] = -0.5*(wf[i+1]-2.0*wf[i]+wf[i-1])/(dx**2)

    i = 0
    hwf[i] = -0.5*(wf[i+1]-2.0*wf[i])/(dx**2)
    i = n-1
    hwf[i] = -0.5*(-2.0*wf[i]+wf[i-1])/(dx**2)

    hwf = hwf + vpot*wf

    return hwf


# Time propagation from t to t+dt
def time_propagation(wf, vpot, dx, dt):

    n = wf.size
    twf = np.zeros(n, dtype=complex)
    hwf = np.zeros(n, dtype=complex)

    twf = wf
    zfact = 1.0 + 0j
    for iexp in range(1,5):
        zfact = zfact*(-1j*dt)/iexp
        hwf = ham_wf(twf, vpot, dx)
        wf = wf + zfact*hwf
        twf = hwf

    return wf


# initial wavefunction parameters
x0 = -2.0
```

```
k0 = 0.00
sigma0 = 1.0

# time propagation parameters
Tprop = 40.0
dt = 0.005
#dt = 0.00905
nt = int(Tprop/dt)+1

# set the coordinate
xmin = -10.0
xmax = 10.0
n = 250

dx = (xmax-xmin)/(n+1)
xj = np.zeros(n)

for i in range(n):
    xj[i] = xmin + dx*(i+1)


# Initialize the wavefunction
wf = initialize_wf(xj, x0, k0, sigma0)
#vpot = np.zeros(n)
vpot = initialize_vpot(xj)

# For loop for the time propagation
wavefunctions = []
for it in range(nt+1):
    if (it % (nt//100) == 0):
        wavefunctions.append(wf.copy())

    wf = time_propagation(wf, vpot, dx, dt)
    print(it, nt)

# Define function to update plot for each frame of the animation
def update_plot(frame):
    plt.cla()
    plt.xlim([-5, 5])
    plt.ylim([-1.2, 5.0])
    plt.plot(xj, np.real(wavefunctions[frame]), label="Real part of $\psi(x)$")
    plt.plot(xj, np.imag(wavefunctions[frame]), label="Imaginary part of $\psi(x)$")
    plt.plot(xj, np.abs(wavefunctions[frame])**2, label=" $|\psi(x)|^2$ ")
    plt.plot(xj, vpot, label="$V(x)$")
    plt.xlabel('$x$')
    plt.ylabel('$\psi(x)$')
    plt.legend()

# Create the animation
fig = plt.figure()
ani = animation.FuncAnimation(fig, update_plot, frames=len(wavefunctions), interval=150)
#ani.save('wavefunction_animation.gif', writer='imagemagick')
ani.save('wavefunction_animation.gif', writer='pillow')
```

6.4.4 調和ポテンシャル:位置と運動量の期待値、エーレンフェストの定理

ここでは、調和ポテンシャルの問題(6.4.2節)に戻って、一と運動量の期待値の時間発展を調べてみよう。また、力の期待値の時間発展を計算し、エーレンフェストの定理が数値計算でも成り立っていることを確かめてみよう。以下に、必要となる量子力学の基礎的な知識を列挙する。

位置の期待値の時間微分は下記のように運動量の期待値に比例する。

$$\frac{d}{dt}\langle x(t)\rangle = \frac{d}{dt}\int dx\psi^*(x,t)x\psi(x,t) = \int dx\psi^*(x,t)\frac{\left[x,\hat{H}\right]}{i\hbar}\psi(x,t) = \int dx\psi^*(x,t)\frac{\hat{p}_x}{m}\psi(x,t) = \frac{\langle p(t)\rangle}{m}. \tag{51}$$

一般に、演算子$\hat{A}$の期待値の時間微分は次のように書ける。

$$\frac{d}{dt}\langle A(t)\rangle = \langle [A,H]\rangle. \tag{52}$$

また、位置の期待値の時間に関する二階微分は次のように評価することが出来る。

$$\frac{d^2}{dt^2}\langle x(t)\rangle = \frac{1}{m}\frac{d}{dt}\langle p(t)\rangle = \frac{1}{m}\left\langle \frac{\left[p_x, \hat{H}\right]}{i\hbar} \right\rangle = \frac{1}{m}\left\langle -\frac{\partial V(x)}{\partial x} \right\rangle. \tag{53}$$

この式は、量子力学において、期待値に関してはNewtonの運動方程式が成り立っていることを示しており、Ehrenfestの定理として知られている。Enrenfestの定理が成り立っているか確認するために、ソースコード 27のようなコードを自作し、位置、運動量、力の期待値を計算してみよう。

https://github.com/shunsuke-sato/python_qe/blob/develop/note_comp_phys/src/qm_dynamics_harmonic_expectation.py

ソースコード 27: 調和ポテンシャル中の量子波束ダイナミクスの期待値の時間発展に関するコードの例

```
from numba import jit
from matplotlib import pyplot as plt
import numpy as np
import matplotlib.animation as animation
from matplotlib.animation import PillowWriter

# Initialize the wavefunction
def initialize_wf(xj, x0, k0, sigma0):
    wf = np.exp(1j*k0*(xj-x0))*np.exp(-0.5*(xj-x0)**2/sigma0**2)
    return wf

# Initialize potential
def initialize_vpot(xj):
    k0 = 1.0
    return 0.5*k0*xj**2

# Operate the Hamiltonian to the wavefunction
@jit(nopython=True)
def ham_wf(wf, vpot, dx):

    n = wf.size
    hwf = np.zeros(n, dtype=np.complex128)

    for i in range(1,n-1):
        hwf[i] = -0.5*(wf[i+1]-2.0*wf[i]+wf[i-1])/(dx**2)

    i = 0
    hwf[i] = -0.5*(wf[i+1]-2.0*wf[i])/(dx**2)
    i = n-1
    hwf[i] = -0.5*(-2.0*wf[i]+wf[i-1])/(dx**2)

    hwf = hwf + vpot*wf

    return hwf


# Time propagation from t to t+dt
def time_propagation(wf, vpot, dx, dt):

    n = wf.size
    twf = np.zeros(n, dtype=complex)
    hwf = np.zeros(n, dtype=complex)

    twf = wf
    zfact = 1.0 + 0j
    for iexp in range(1,5):
        zfact = zfact*(-1j*dt)/iexp
        hwf = ham_wf(twf, vpot, dx)
        wf = wf + zfact*hwf
        twf = hwf

    return wf

# Time propagation from t to t+dt
def calc_expectation_values(wf,xj,vpot):

    dx = xj[1]-xj[0]
```

```
    norm = np.sum(np.abs(wf)**2)*dx
    x_exp = np.sum(xj*np.abs(wf)**2)*dx
    x_exp = x_exp/norm

    n = wf.size
    pwf = np.zeros(n, dtype=complex)

    for i in range(1,n-1):
        pwf[i] = -1j*(wf[i+1]-wf[i-1])/(2.0*dx)

    p_exp = np.real(np.sum(np.conjugate(wf)*pwf)*dx)
    p_exp = p_exp/norm


    n = wf.size
    twf = np.zeros(n, dtype=complex)

    for i in range(1,n-1):
        twf[i] = -0.5*(wf[i+1]-2.0*wf[i]+wf[i-1])/(dx**2)


    Ekin = np.real(np.sum(np.conjugate(wf)*twf)*dx)
    Ekin = Ekin/norm

    Epot = np.real(np.sum(np.abs(wf)**2*vpot)*dx)
    Epot = Epot/norm

    return x_exp,p_exp,norm, Ekin, Epot


# initial wavefunction parameters
x0 = -2.0
k0 = 0.00
sigma0 = 1.0

# time propagation parameters
Tprop = 40.0
dt = 0.005
#dt = 0.00905
nt = int(Tprop/dt)+1

# set the coordinate
xmin = -10.0
xmax = 10.0
n = 250

dx = (xmax-xmin)/(n+1)
xj = np.zeros(n)

for i in range(n):
    xj[i] = xmin + dx*(i+1)


# Initialize the wavefunction
wf = initialize_wf(xj, x0, k0, sigma0)
#vpot = np.zeros(n)
vpot = initialize_vpot(xj)

# For expectation values
tt = np.zeros(nt+1)
xt = np.zeros(nt+1)
pt = np.zeros(nt+1)
norm_t = np.zeros(nt+1)
Ekin_t = np.zeros(nt+1)
Epot_t = np.zeros(nt+1)

# For loop for the time propagation
wavefunctions = []
for it in range(nt+1):
    if (it % (nt//100) == 0):
        wavefunctions.append(wf.copy())

    tt[it] = dt*it
    xt[it], pt[it], norm_t[it], Ekin_t[it], Epot_t[it]= calc_expectation_values(wf,xj,vpot)

    wf = time_propagation(wf, vpot, dx, dt)
    print(it, nt)

# Output the expectation value
plt.plot(tt,xt, label="x(t)")
```

```
plt.plot(tt,pt, label="p(t)")
plt.plot(tt,norm_t, label="norm(t)")
plt.xlabel('t')
plt.ylabel('Quantities')
plt.legend()

plt.savefig("expectation_value.pdf")
plt.cla()

plt.plot(tt,Ekin_t, label="Kinetic energy")
plt.plot(tt,Epot_t, label="Potential energy")
plt.plot(tt,Ekin_t+Epot_t, label="Total energy")
plt.xlabel('t')
plt.ylabel('Energy')
plt.legend()

plt.savefig("expectation_value_energy.pdf")

# Define function to update plot for each frame of the animation
def update_plot(frame):
    plt.cla()
    plt.xlim([-5, 5])
    plt.ylim([-1.2, 5.0])
    plt.plot(xj, np.real(wavefunctions[frame]), label="Real part of $\psi(x)$")
    plt.plot(xj, np.imag(wavefunctions[frame]), label="Imaginary part of $\psi(x)$")
    plt.plot(xj, np.abs(wavefunctions[frame])**2, label=" $|\psi(x)|^2$ ")
    plt.plot(xj, vpot, label="$V(x)$")
    plt.xlabel('$x$')
    plt.ylabel('$\psi(x)$')
    plt.legend()

# Create the animation
fig = plt.figure()
ani = animation.FuncAnimation(fig, update_plot, frames=len(wavefunctions), interval=150)
#ani.save('wavefunction_animation.gif', writer='imagemagick')
ani.save('wavefunction_animation.gif', writer='pillow')
```

7　1次元系量子系の基底状態・励起状態計算

この節では、時間に依存しないシュレディンガー方程式を数値的に解き、1次元量子系の基底状態、及び励起状態を調べる。

7.1　線形代数の復習

時間に依存しないシュレディンガー方程式の数値計算を行う前に、線形代数の基礎的な事項を復習しておこう。ここで、複素数を要素に持つ大きさ$n \times n$の正方行列Aを考える。行列Aのi行j°の要素をa_{ij}と書くことにしよう。

行列の転置はその行列の行と列を入れ替える操作を表し、行列Aの転置はA^Tと表す。したがって、$B = A^T$によって定義される行列Bのi行j列の要素b_{ij}と行列Aの要素には以下の関係がある。

$$b_{ij} = a_{ji}. \tag{54}$$

行列を転置しさらに複素共役を取る操作をエルミート共役と呼び、行列Aのエルミート共役は$A^\dagger$と表す。したがって、$C = A^\dagger$によって定義される行列Cのi行j列の要素c_{ij}と行列Aの要素には以下の関係がある。

$$c_{ij} = a^*_{ji}. \tag{55}$$

また、ある行列Aのエルミート共役$A^\dagger$と元の行列が一致する行列$(A^\dagger = A)$をエルミート行列と呼ぶ。行列Aがエルミート行列の場合、行列Aの固有値が実数となり、互いに直交したn個の固有ベクトルを見つけることが出来る。これを確かめてみよう。

7.1.1　エルミート行列の固有値が実数であることの証明

行列Aの固有値と固有ベクトル(列ベクトル)をλ_iと$\boldsymbol{u}_i$とすると次の式が成り立つ。

$$A\boldsymbol{u}_i = \lambda_i \boldsymbol{u}_i \tag{56}$$

式 (56)の左から両辺に$\boldsymbol{u}_i^\dagger$を掛けると、次式を得る。

$$\boldsymbol{u}_i^\dagger A\boldsymbol{u}_i = \lambda_i(\boldsymbol{u}_i^\dagger \boldsymbol{u}_i) \tag{57}$$

また、式(56)のエルミート共役を取ると次式を得る。

$$\boldsymbol{u}_i^\dagger A^\dagger = \boldsymbol{u}_i^\dagger A = \lambda_i^* \boldsymbol{u}_i^\dagger \tag{58}$$

さらに、式(58)の右から両辺に$\boldsymbol{u}_i$を掛けると、次式を得る。

$$\boldsymbol{u}_i^\dagger A\boldsymbol{u}_i = \lambda_i^*(\boldsymbol{u}_i^\dagger \boldsymbol{u}_i) \tag{59}$$

固有ベクトル$\boldsymbol{u}_i$のノルム$|\boldsymbol{u}_i|^2 = (\boldsymbol{u}_i^\dagger \boldsymbol{u}_i)$がゼロではないことに注意すると、式(57)と式(59)を比べることで、$\lambda_i = \lambda_i^*$であることが分かる。すなわち、エルミート行列の固有値は実数であることを確かめることができる。

7.1.2　エルミート行列の異なる固有値に属する固有ベクトルが直交することの証明

次に、エルミート行列について、異なる固有値に属する固有ベクトルは直交することを示す。エルミート行列の異なる二つの固有値をλ_iとλ_jとし、それぞれに対応する固有ベクトルを$\boldsymbol{u}_i$と$\boldsymbol{u}_j$とすると次の式が成り立つ。

$$A\boldsymbol{u}_i = \lambda_i \boldsymbol{u}_i, \tag{60}$$

$$A\boldsymbol{u}_j = \lambda_j \boldsymbol{u}_j. \tag{61}$$

また、式(61)のエルミート共役を取ることで次式を得る。

$$\boldsymbol{u}_j^\dagger A = \lambda_j \boldsymbol{u}_j^\dagger. \tag{62}$$

さらに、式(60)の両辺に左から$\boldsymbol{u}_j^\dagger$を掛け、また式(62)の両辺に右から$\boldsymbol{u}_i$を掛けることで次の関係式を得る。

$$\boldsymbol{u}_j^\dagger A \boldsymbol{u}_i = \lambda_i (\boldsymbol{u}_j^\dagger \boldsymbol{u}_i), \tag{63}$$

$$\boldsymbol{u}_j^\dagger A \boldsymbol{u}_i = \lambda_j (\boldsymbol{u}_j^\dagger \boldsymbol{u}_i). \tag{64}$$

ここで式(63)と式(64)の差を取ることにより、次式を得る。

$$(\lambda_i - \lambda_j)(\boldsymbol{u}_j^\dagger \boldsymbol{u}_i) = 0. \tag{65}$$

仮定よりλ_iとλ_jは異なる固有値であるため、$(\lambda_i - \lambda_j) \neq 0$であるので、式(65)より$(\boldsymbol{u}_j^\dagger \boldsymbol{u}_i) = 0$でで無ければならないことが分かる。したがって、エルミート行列の異なる固有値に属する固有ベクトルは直交することが分かる。

7.1.3 エルミート行列の等しい固有値に属する固有ベクトルの直交性について(固有値に縮退がある場合について)

次に、エルミート行列の固有値に縮退がある場合の固有ベクトルの直交性について述べる。ここではエルミート行列の固有値λがm重に縮退しており、既にm個の線形独立な固有ベクトル$\boldsymbol{u}_1, \boldsymbol{u}_2, \cdots, \boldsymbol{u}_m$が得られているものとする。節7.1.2では、エルミート行列の異なる固有値に属する固有ベクトルが直交することを示したが、ここで考えている固有ベクトル$(\boldsymbol{u}_1, \boldsymbol{u}_2, \cdots, \boldsymbol{u}_m)$の固有値は等しいため、一般にこれらの固有ベクトルは直交していない。しかし、これらの固有ベクトルの線形結合もまた同じ固有値λを持つ固有ベクトルになることを利用して、正規直交な新たな固有ベクトルの組を作ることが出来る。

ここでは、Gram-Schmidtの正規直交化法を用いて正規直交な固有ベクトルの組を作る方法について説明する。縮退した固有ベクトルの組$(\boldsymbol{u}_1, \boldsymbol{u}_2, \cdots, \boldsymbol{u}_m)$から正規直交化された固有ベクトルの組を作るために、まず最初の固有ベクトルを次のように規格化する。

$$\tilde{\boldsymbol{u}}_1 = \frac{1}{\sqrt{(\boldsymbol{u}_1^\dagger \boldsymbol{u}_1)}} \boldsymbol{u}_1. \tag{66}$$

次に、2番目の固有ベクトル$\boldsymbol{u}_2$とベクトル$\tilde{\boldsymbol{u}}_1$を直交化し、一次的なベクトル$\bar{\boldsymbol{u}}_2$を次のように導入する。

$$\bar{\boldsymbol{u}}_2 = \boldsymbol{u}_2 - \tilde{\boldsymbol{u}}_1 (\tilde{\boldsymbol{u}}_1^\dagger \boldsymbol{u}_2). \tag{67}$$

$\bar{\boldsymbol{u}}_2$と$\tilde{\boldsymbol{u}}_1$の内積を取ることで、二つのベクトルが直交していることを容易に確かめられる。この一次的なベクトル$\bar{\boldsymbol{u}}_2$を規格化することで、正規直交化された2番目の固有ベクトルを次のように導入する。

$$\tilde{\boldsymbol{u}}_2 = \frac{1}{\sqrt{(\bar{\boldsymbol{u}}_2^\dagger \bar{\boldsymbol{u}}_2)}} \bar{\boldsymbol{u}}_2. \tag{68}$$

同様に、3番目の固有ベクトル$\boldsymbol{u}_2$をベクトル$\tilde{\boldsymbol{u}}_1$とベクトル$\tilde{\boldsymbol{u}}_2$と直交化し、一次的なベクトル$\bar{\boldsymbol{u}}_3$を次のように導入する。

$$\bar{\boldsymbol{u}}_3 = \boldsymbol{u}_3 - \tilde{\boldsymbol{u}}_1 (\tilde{\boldsymbol{u}}_1^\dagger \boldsymbol{u}_3) - \tilde{\boldsymbol{u}}_2 (\tilde{\boldsymbol{u}}_2^\dagger \boldsymbol{u}_3). \tag{69}$$

この一次的なベクトル$\bar{\boldsymbol{u}}_3$を規格化することで、正規直交化された3番目の固有ベクトルを次のように導入する。

$$\tilde{\boldsymbol{u}}_3 = \frac{1}{\sqrt{(\bar{\boldsymbol{u}}_3^\dagger \bar{\boldsymbol{u}}_3)}} \bar{\boldsymbol{u}}_3. \tag{70}$$

同様の手続きを繰り返し、k番目の固有ベクトル$\boldsymbol{u}_k$を$\tilde{\boldsymbol{u}}_1, \tilde{\boldsymbol{u}}_2, \cdots, \tilde{\boldsymbol{u}}_{k-1}$と直交化し、一次的なベクトル$\bar{\boldsymbol{u}}_k$を次のように導入する。

$$\bar{\boldsymbol{u}}_k = \boldsymbol{u}_k - \sum_{i=1}^{k-1} \tilde{\boldsymbol{u}}_i (\tilde{\boldsymbol{u}}_i^\dagger \boldsymbol{u}_k). \tag{71}$$

この一次的なベクトル$\bar{\boldsymbol{u}}_k$を規格化することで、正規直交化されたk番目の固有ベクトルを次のように導入する。

$$\tilde{\boldsymbol{u}}_k = \frac{1}{\sqrt{(\bar{\boldsymbol{u}}_k^\dagger \bar{\boldsymbol{u}}_k)}} \bar{\boldsymbol{u}}_k. \tag{72}$$

このような手続きを$k = m$まで繰り返すことで、もともとのm個の固有ベクトルの組$(\boldsymbol{u}_1, \boldsymbol{u}_2, \cdots, \boldsymbol{u}_m)$から、正規直交化されたベクトルの組$(\tilde{\boldsymbol{u}}_1, \tilde{\boldsymbol{u}}_2, \cdots, \tilde{\boldsymbol{u}}_m)$を作ることが出来る。また、新しく導入されたベクトル$(\tilde{\boldsymbol{u}}_1, \tilde{\boldsymbol{u}}_2, \cdots, \tilde{\boldsymbol{u}}_m)$は、いずれも元の固有ベクトル$(\boldsymbol{u}_1, \boldsymbol{u}_2, \cdots, \boldsymbol{u}_m)$の線形結合として定義されており、正規直交化されたベクトルの組$(\tilde{\boldsymbol{u}}_1, \tilde{\boldsymbol{u}}_2, \cdots, \tilde{\boldsymbol{u}}_m)$も行列$A$の固有値$\lambda$に属する固有ベクトルとなっている。

7.1.4 エルミート行列の固有値・固有ベクトルの性質に関する簡単なまとめ

上記の節7.1.2で見たように、エルミート行列の固有値は実数となる。また、節7.1.3と節7.1.3で見たように、エルミート行列の固有ベクトルの組を正規直交化されたベクトルの組として選ぶことが出来る。

7.2 実対称行列の対角化の数値計算

ここでは、3×3の実対称行列を例に、行列の対角化を行うPythonコードについて学ぶ。次のような実対称行列を考えてみよう。

$$A = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \tag{73}$$

まずは、この実対称行列の固有値・固有ベクトルを求めてみよう。固有値λは、次の方程式の解として求めることが出来る。

$$|A - \lambda I| = 0. \tag{74}$$

次に、この固有値問題を数値的に求めるコードを書いてみよう。次のコードはNumpyの線形代数計算用関数(`numpy.linalg`)を用いて実対称行列の対角化を行い、固有値・固有ベクトルを求めている。ここでは、特に、実対称行列用の対角化関数である`numpy.linalg.eigh`を用いて対角化を行っている。Numpyの`numpy.linalg.eigh`に関するドキュメント(`https://numpy.org/doc/stable/reference/generated/numpy.linalg.eigh.html#numpy.linalg.eigh`)を読み、どのような関数なのか調べてみよう。

`https://github.com/shunsuke-sato/python_qe/blob/develop/note_comp_phys/src/eigen_3x3.py`

ソースコード 28: 3×3実対称行列の対角化コードの例

```
import numpy as np

matrix = np.array([[0.0, 1.0, 0.0],
                   [1.0, 0.0, 1.0],
                   [0.0, 1.0, 0.0]])

eigenvalues, eigenvectors = np.linalg.eigh(matrix)

```

```
print('First␣eigenvalue␣␣=',eigenvalues[0])
print('Second␣eigenvalue␣=',eigenvalues[1])
print('Third␣eigenvalue␣␣=',eigenvalues[2])
print()

print('First␣eigenvector')
print(eigenvectors[:,0])
print()

print('Second␣eigenvector')
print(eigenvectors[:,1])
print()

print('Third␣eigenvector')
print(eigenvectors[:,2])
print()
```

上記のコード 28を実行し、解析的に求めた固有値・固有ベクトルと数値計算により得られた固有値・固有ベクトルを比較してみよう。

7.3 実空間差分法を用いた時間に依存しないシュレディンガー方程式の求解

7.3.1 無限に深い井戸型ポテンシャル問題

ここでは、節 6.1で学んだ実空間法を用いて、時間に依存しないシュレディンガー方程式を解く方法について学ぶ。例題として、次のような無限に深い井戸型ポテンシャルの問題を考えよう。

$$\left[-\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2}+V(x)\right]u_n(x)=Eu(x), \tag{75}$$

$$V(x)=\begin{cases}0 & -\frac{L}{2}\leq x\leq\frac{L}{2}\\ \infty & \text{otherwise}\end{cases} \tag{76}$$

ここで、$|x|>L/2$の領域においてポテンシャル$V(x)$が発散しているので、この領域において波動関数$u(x)$がゼロとなる。すなわち、波動関数$u(x)$が有限の値を取るのは領域$(-L/2\leq x\leq L/2)$である。この領域をN個のグリッドに分割することを考える。この際、後の便宜のために(-1)番目の点の座標を$x_{-1}=-L/2$とし、N番目の点の座標を$x_N=L/2$とするとj番目の座標の位置を次のように表すことが出来る。

$$x_j=-\frac{L}{2}+\delta x\times(j+1). \tag{77}$$

ただし、刻み幅Δxは次のように定義されている。

$$\Delta x=\frac{L}{N+1}. \tag{78}$$

次に、式(75)を差分化することを考える。j番目のグリッド点x_jにおける波動関数の値を$u_j=u(x_j)$と表すことにし、二階微分を3点差分公式で近似することで、式(75)を次のような

連立方程式で近似することが出来る。

$$-\frac{\hbar^2}{2m}\frac{-2u_0+u_1}{\Delta x^2} = Eu_0 \tag{79}$$

$$-\frac{\hbar^2}{2m}\frac{u_0-2u_1+u_2}{\Delta x^2} = Eu_1 \tag{80}$$

$$\vdots$$

$$-\frac{\hbar^2}{2m}\frac{u_{j-1}-2u_j+u_{j+1}}{\Delta x^2} = Eu_j \tag{81}$$

$$\vdots$$

$$-\frac{\hbar^2}{2m}\frac{u_{N-3}-2u_{N-2}+u_{N-1}}{\Delta x^2} = Eu_{N-2} \tag{82}$$

$$-\frac{\hbar^2}{2m}\frac{u_{N-2}-2u_{N-1}}{\Delta x^2} = Eu_{N-1} \tag{83}$$

ここで、境界条件($u(x)=0$ for $|x|>L/2$)により$u_{-1}=u_N=0$であることに注意する。

上記の連立方程式を注意深く観察すると、次のような行列の固有問題として書き直せることが分かる。

$$-\frac{\hbar^2}{2m}\frac{1}{\Delta x^2}\begin{pmatrix} -2 & 1 & 0 & \cdots & 0 \\ 1 & -2 & 1 & \ddots & 0 \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & 1 & -2 & 1 \\ 0 & 0 & 0 & 1 & -2 \end{pmatrix}\begin{pmatrix} u_0 \\ u_1 \\ \vdots \\ u_{N-2} \\ u_{N-1} \end{pmatrix} = E\begin{pmatrix} u_0 \\ u_1 \\ \vdots \\ u_{N-2} \\ u_{N-1} \end{pmatrix} \tag{84}$$

したがって、元のシュレディンガー方程式(75)は、差分近似を用いることで式(84)のような行列の固有値問題に書き換えることが出来る。

節 7.2で学んだ行列の対角化を行うプログラムを参考にして、式(84)で表される無限に深い井戸型ポテンシャルの固有値問題を数値的に解き、固有値と固有関数を求めるプログラムを自作してみよう。また、数値計算によって得られた結果と解析的に得られる結果を比較してみよう。

参考のために、以下に、式(75)の厳密な固有値E_nと固有関数$u_n(x)$を記す。

$$E_n = n^2\frac{\pi^2\hbar^2}{2mL^2} \tag{85}$$

$$u_n(x) = \begin{cases} \sqrt{\frac{2}{L}}\cos\left(\frac{n\pi}{L}x\right) & (n=\text{odd}), (-L/2<x<L/2) \\ \sqrt{\frac{2}{L}}\sin\left(\frac{n\pi}{L}x\right) & (n=\text{even}), (-L/2<x<L/2) \\ 0 & (\text{otherwise}) \end{cases} \tag{86}$$

また、参考のためPythonのサンプルコードを以下に示す。

`https://github.com/shunsuke-sato/python_qe/blob/develop/note_comp_phys/src/qm_grid_infinite_well.py`

ソースコード 29: 無限に深い井戸型ポテンシャルの実空間差分法に依る求解を行うサンプルコード

```
1 import numpy as np
```

```
2  from matplotlib import pyplot as plt
3
4  # Constants
5  mass = 1.0
6  hbar = 1.0
7
8  # Define grid
9  num_grid = 64
10 length = 20.0
11 dx = length / (num_grid + 1)
12 xj = np.linspace(-length / 2 + dx, length / 2 - dx, num_grid)
13
14 # Hamiltonian Matrix
15 ham_mat = np.zeros((num_grid,num_grid))
16
17 for i in range(num_grid):
18     for j in range(num_grid):
19         if(i == j):
20             ham_mat[i,j]=-0.5*hbar**2/mass*(-2.0/dx**2)
21         elif(np.abs(i-j) == 1):
22             ham_mat[i,j]=-0.5*hbar**2/mass*(1.0/dx**2)
23
24 # Calculate eigenvectors and eigenvalues
25 eigenvalues, eigenvectors = np.linalg.eigh(ham_mat)
26
27 # Normalize and check the sign
28 wf = eigenvectors/np.sqrt(dx)
29 for i in range(num_grid):
30     sign = np.sign(wf[num_grid//2,i])
31     if(sign != 0.0):
32         wf[:,i] = wf[:,i]*sign
33
34
35 def exact_eigenvalue(n):
36     """Calculate exact eigenvalue for particle in a box."""
37     return n**2 * np.pi**2 * hbar**2 / (2.0 * mass * length**2)
38
39 # Print eigenvalues and errors
40 for i in range(3):
41     print(f"{i}-th␣eigenvalue␣=␣{eigenvalues[i]}")
42     print(f"{i}-th␣eigenvalue␣Error␣=␣{eigenvalues[i]␣-␣exact_eigenvalue(i␣+␣1)}")
43     print()
44
45 # Plotting
46 plt.figure(figsize=(8,6))
47 plt.plot(xj, wf[:, 0], label="Ground␣state␣(calc.)")
48 plt.plot(xj, np.sqrt(2.0/length)*np.cos(np.pi*xj/length), label="Ground␣state␣(exact.)",
     linestyle='dashed')
49 plt.plot(xj, wf[:, 1], label="1st␣excited␣state")
50 plt.plot(xj, np.sqrt(2.0/length)*np.sin(2.0*np.pi*xj/length), label="1st␣excited␣state␣(exact.)
     ", linestyle='dashed')
51 plt.plot(xj, wf[:, 2], label="2nd␣excited␣state")
52 plt.plot(xj, np.sqrt(2.0/length)*np.cos(3.0*np.pi*xj/length), label="2nd␣excited␣state␣(exact.)
     ", linestyle='dashed')
53
54 plt.xlim([-length/2.0,length/2.0])
55 plt.xlabel('x')
56 plt.ylabel('wave␣functions')
57 plt.legend()
58 plt.savefig('fig_quantum_well_wf.pdf')
59 plt.show()
```

上記のソースコード 29を実行することで、無限に深い井戸型ポテンシャルの固有値、及び波動関数を得ることが出来る。ここで、上記のコードについて何点か解説を記す。

ソースコード 29の28行目の`wf = eigenvectors/np.sqrt(dx)`では、対角化ルーチン`np.linalg.eigh`の出力である固有ベクトルについて規格化を行っている。対角化ルーチン`np.linalg.eigh`の出力する固有ベクトル$\boldsymbol{u}$は次のような定義で列ベクトルとして規格化されている。

$$|\boldsymbol{u}|^2 == \sum_{j=0}^{N-1} |u_j|^2 = 1. \tag{87}$$

しかし、通常の量子力学の問題では固有関数$u(x)$を次のように規格化するのが便利である。

$$\int_{-L/2}^{L/2} dx|u(x)|^2 = 1. \tag{88}$$

ここで、$u(x_j) = u_j$であることを思い出せば、規格化条件の式(88)は次のように書き直すことが出来る。

$$\int_{-L/2}^{L/2} dx|u(x)|^2 = \sum_{j=0}^{N-1} |u(x_j)|^2 \Delta x = \sum_{j=0}^{N-1} |u_j|^2 \Delta x = 1. \tag{89}$$

もともと、式(87)にて規格化されている固有ベクトルを、式(89)で規格化される新たな量に書き換えるには、次のような変換を行えばよいことが分かる。

$$u_j \to \frac{u_j}{\sqrt{\Delta x}}. \tag{90}$$

この置き換え操作(再規格化)を行っているのがソースコード 29の28行目である。

以下に、プログラムが出力する波動関数の図を示す。

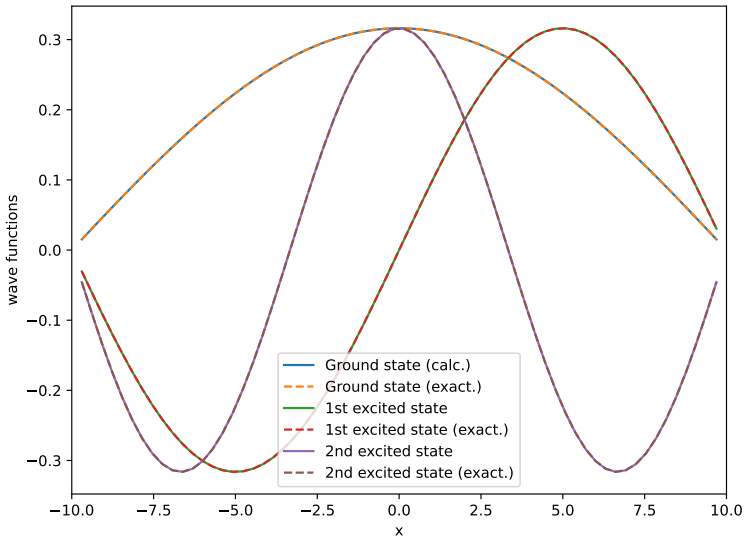

Figure 11: 無限に深い井戸型ポテンシャルの固有関数の様子。

7.3.2 1次元調和振動子の基底状態・励起状態計算

前節 7.3.1では無限に深い井戸型ポテンシャルの数値計算に取り組んだ。この節では、1次元の調和振動子の問題を数値計算に取り組んでみよう。1次元調和振動子のシュレディンガー方程式は次のように書くことが出来る。

$$\left[-\frac{\hbar^2}{2m}\frac{d^2}{dx^2} + \frac{k}{2}x^2\right] u(x) = Eu(x). \tag{91}$$

この微分方程式を$|x| \to \infty$において$u(x) \to 0$となる境界条件の下で解くことで、量子調和振動子のエネルギー固有値、及び固有状態を求めることが出来る。しかし、数値的にこの問題を扱う場合、無限遠での波動関数を扱うことは容易ではない。実際の量子系の固有状態計算では、

無限遠での境界条件を設定する代わりに、有限の、しかし十分遠くでの波動関数の値に境界条件を課すことが多い。具体的には、ある十分大きな長さLを導入し$u(\pm L) = 0$となる境界条件の下で式(91)のような微分方程式を解くことになる。この長さLが十分大きい場合、数値計算によって得られる結果は、無限遠で波動関数がゼロとなる境界条件のもとで得られる解に十分近づく。また、前節 7.3.1を参考に見方を変えると、境界条件$(u(\pm L) = 0)$課すことは、位置$x = \pm L$に無限に高いポテンシャルを設定するのと等価である。このような無限に高いポテンシャルを興味のある系よりも十分離れた位置に導入しても、興味のある量子系の性質はほとんど影響を受けないことは想像できるだろう。

前節の井戸型ポテンシャルの問題を参考に、時間に依存しないシュレディンガー方程式を差分化して、行列の対角化問題に書き換えることを考えてみよう。また、調和振動子の固有状態を求めるプログラムを自作し、基底状態、及び励起状態を調べてみよう。さらに、高エネルギーの励起状態の波動関数から計算できる位置の確率分布と古典的な位置の確率分布の関係について考察してみよう。

参考のため、調和振動子の基底状態、及び第一励起状態、第二励起状態の波動関数を以下に示す。

$$
\begin{aligned}
\psi_0(x) &= \left(\frac{m\omega}{\pi\hbar}\right)^{1/4} e^{-\frac{m\omega}{2\hbar}x^2}, \\
\psi_1(x) &= x\sqrt{\frac{2m\omega}{\hbar}}\left(\frac{m\omega}{\pi\hbar}\right)^{1/4} e^{-\frac{m\omega}{2\hbar}x^2}, \\
\psi_2(x) &= \sqrt{2}\left(1 - \frac{2m\omega x^2}{\hbar}\right)\left(\frac{m\omega}{\pi\hbar}\right)^{1/4} e^{-\frac{m\omega}{2\hbar}x^2}.
\end{aligned} \tag{92}
$$

https://github.com/shunsuke-sato/python_qe/blob/develop/note_comp_phys/src/qm_grid_ho.py

ソースコード 30: 1次元調和振動子の固有値・固有関数を求める計算コード例

```python
import numpy as np
from matplotlib import pyplot as plt

# Constants
mass = 1.0
hbar = 1.0
kconst = 1.0

# Define grid
num_grid = 128
length = 15.0
dx = length / (num_grid + 1)
xj = np.linspace(-length / 2 + dx, length / 2 - dx, num_grid)

# Potential
vpot = 0.5*kconst*xj**2

# Hamiltonian Matrix
ham_mat = np.zeros((num_grid,num_grid))

for i in range(num_grid):
    for j in range(num_grid):
        if(i == j):
            ham_mat[i,j]=-0.5*hbar**2/mass*(-2.0/dx**2) + vpot[i]
        elif(np.abs(i-j) == 1):
            ham_mat[i,j]=-0.5*hbar**2/mass*(1.0/dx**2)


# Calculate eigenvectors and eigenvalues
eigenvalues, eigenvectors = np.linalg.eigh(ham_mat)

# Normalize and check the sign
wf = eigenvectors/np.sqrt(dx)
for i in range(num_grid):
    sign = np.sign(wf[num_grid//2,i])
    if(sign != 0.0):
        wf[:,i] = wf[:,i]*sign

```

```
39
40 def exact_eigenvalue(n):
41     """Calculate exact eigenvalue for quantum harmonic oscillator."""
42     return hbar * np.sqrt(kconst / mass) * (n + 0.5)
43
44 # Print eigenvalues and errors
45 for i in range(3):
46     print(f"{i}-th␣eigenvalue␣=␣{eigenvalues[i]}")
47     print(f"{i}-th␣eigenvalue␣Error␣=␣{eigenvalues[i]␣-␣exact_eigenvalue(i)}")
48     print()
49
50
51 # Plotting
52 omega = np.sqrt(kconst/mass)
53
54 # Ground state plot
55 plt.figure(figsize=(8, 6))
56 plt.plot(xj, wf[:, 0], label="Ground␣state␣(calc.)")
57 plt.plot(xj, (mass * omega / (np.pi * hbar))**(1.0 / 4.0) * np.exp(-mass * omega * xj**2 / (2.0
    * hbar)),
58                  label="Ground␣state␣(exact.)", linestyle='dashed')
59 plt.xlim([-length / 2.0, length / 2.0])
60 plt.xlabel('x')
61 plt.ylabel('wave␣functions')
62 plt.legend()
63 plt.savefig('fig_harmonic_oscillator_ground_state.pdf')
64 plt.show()
65
66 # First excited state plot
67 plt.figure(figsize=(8, 6))
68 plt.plot(xj, wf[:, 1], label="1st␣excited␣state␣(calc.)")
69 plt.plot(xj, (mass * omega / (np.pi * hbar))**(1.0 / 4.0) * np.sqrt(2.0 * mass * omega / hbar)
   * xj * np.exp(-mass * omega * xj**2 / (2.0 * hbar)),
70                  label="1st␣excited␣state␣(exact.)", linestyle='dashed')
71 plt.xlim([-length / 2.0, length / 2.0])
72 plt.xlabel('x')
73 plt.ylabel('wave␣functions')
74 plt.legend()
75 plt.savefig('fig_harmonic_oscillator_1st_excited_state.pdf')
76 plt.show()
77
78 # Second excited state plot
79 plt.figure(figsize=(8, 6))
80 plt.plot(xj, wf[:, 2], label="2nd␣excited␣state␣(calc.)")
81 plt.plot(xj, (mass * omega / (np.pi * hbar))**(1.0 / 4.0) * np.sqrt(0.5) * (1.0 - 2.0 * mass *
   omega * xj**2 / hbar) * np.exp(-mass * omega * xj**2 / (2.0 * hbar)),
82                  label="2nd␣excited␣state␣(exact.)", linestyle='dashed')
83 plt.xlim([-length / 2.0, length / 2.0])
84 plt.xlabel('x')
85 plt.ylabel('wave␣functions')
86 plt.legend()
87 plt.savefig('fig_harmonic_oscillator_2nd_excited_state.pdf')
88 plt.show()
```

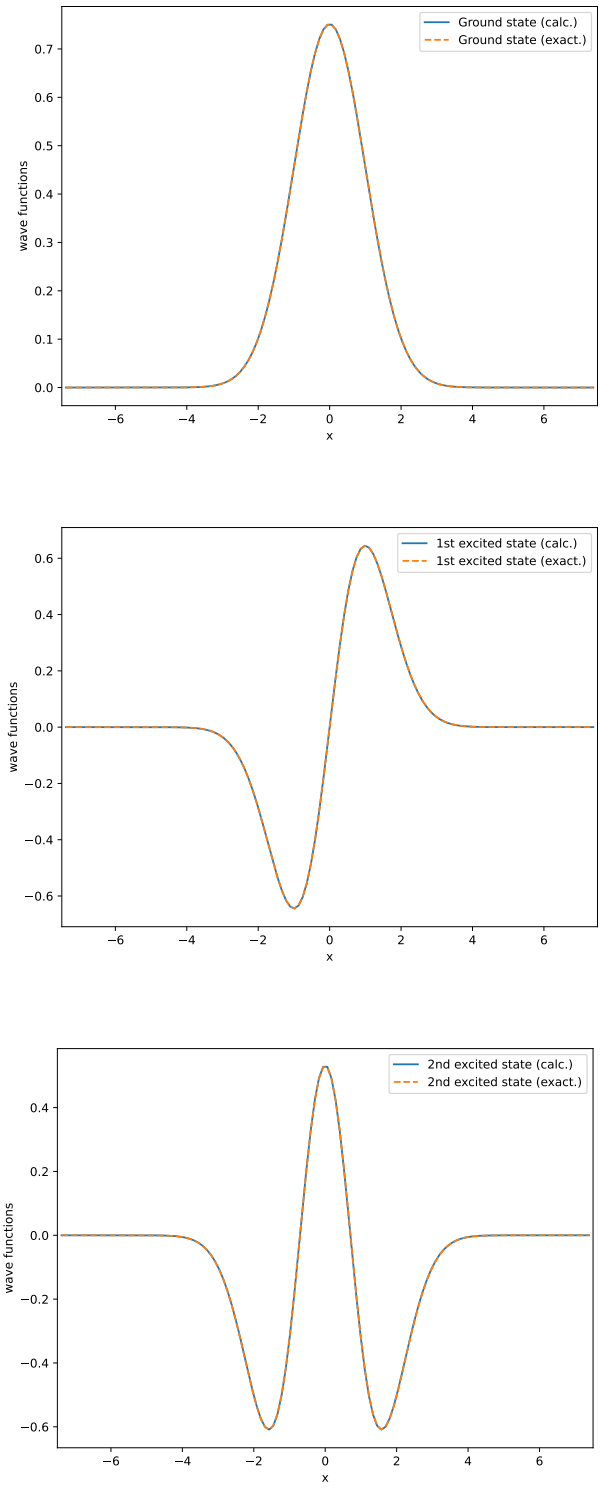

Figure 12: 調和振動子ポテンシャルの固有関数の様子。

ここで、古典的な粒子の確率分につついて考察してみよう。あるエネルギー状態Eの古典的な調和振動子に対し、ランダムな時刻に質点の位置を測定することを考える。このとき、

粒子の運動エネルギーと位置エネルギーの和がEであることを考慮すると、質点の位置は区間$(-\sqrt{2E/k} \leq x \leq \sqrt{2E/k})$の中でのみ見いだされることが分かる。また、この区間の中でランダムな時刻に質点の位置を測定すると、ある微小区間$[x, x+\delta x]$の間で質点を見出す確率$P(x)$は、質点の速度の絶対値の逆数に比例する。すなわち、

$$P(x) \sim \frac{1}{|v|}. \tag{93}$$

ところで、全エネルギーEが与えられているので、質点の各位置xにおける質点の速度の絶対値は以下のような式であらわされる。

$$|v| = \sqrt{\frac{2}{m}\left(E - \frac{1}{2}kx^2\right)} \tag{94}$$

したがって、エネルギーEの調和振動子において位置xで質点を見出す確率分布$P(x)$は次のように与えられる。

$$P(x) = \frac{1}{\pi\sqrt{\frac{2E}{k} - x^2}} \tag{95}$$

サンプルソースコード 31は、上記のソースコード 30に古典的な確率分布と量子的な確率分布を比較する機能を加えたものである。また、図 13には、このコードを実行することで得られる64番目の固有状態の量子的確率分布と、対応する古典的分布を比較した図を示した。

https://github.com/shunsuke-sato/python_qe/blob/develop/note_comp_phys/src/qm_grid_ho_high_energy.py

ソースコード 31: 1次元調和振動子の高エネルギー状態と古典確率分布の比較計算コード例

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

# Constants
mass = 1.0
hbar = 1.0
kconst = 1.0

# Define grid
num_grid = 512
length = 30.0
dx = length / (num_grid + 1)
xj = np.linspace(-length / 2 + dx, length / 2 - dx, num_grid)

# Potential
vpot = 0.5*kconst*xj**2

# Hamiltonian Matrix
ham_mat = np.zeros((num_grid,num_grid))

for i in range(num_grid):
    for j in range(num_grid):
        if(i == j):
            ham_mat[i,j]=-0.5*hbar**2/mass*(-2.0/dx**2) + vpot[i]
        elif(np.abs(i-j) == 1):
            ham_mat[i,j]=-0.5*hbar**2/mass*(1.0/dx**2)


# Calculate eigenvectors and eigenvalues
eigenvalues, eigenvectors = np.linalg.eigh(ham_mat)

# Normalize and check the sign
wf = eigenvectors/np.sqrt(dx)
for i in range(num_grid):
    sign = np.sign(wf[num_grid//2,i])
    if(sign != 0.0):
        wf[:,i] = wf[:,i]*sign

```

```python

def exact_eigenvalue(n):
    """Calculate exact eigenvalue for quantum harmonic oscillator."""
    return hbar * np.sqrt(kconst / mass) * (n + 0.5)

# Print eigenvalues and errors
for i in range(3):
    print(f"{i}-th eigenvalue = {eigenvalues[i]}")
    print(f"{i}-th eigenvalue Error = {eigenvalues[i] - exact_eigenvalue(i)}")
    print()


# Plotting
omega = np.sqrt(kconst/mass)


# Calculate the probability distribution fo a highly-excited state
n_eigen = 64
Ene = eigenvalues[n_eigen]

prob_x = np.zeros(num_grid)
for i in range(num_grid):
    if( 2.0*Ene/kconst-xj[i]**2 < 0):
        prob_x[i]=0.0
    else:
        prob_x[i]=1.0/(np.pi*np.sqrt(2.0*Ene/kconst-xj[i]**2))

plt.figure(figsize=(8,6))

plt.plot(xj, wf[:, n_eigen]**2, label="Quantum distribution")
plt.plot(xj, prob_x, label="Classical distribution")
plt.xlim([-length/2.0,length/2.0])
plt.xlabel('x')
plt.ylabel('$|\psi(x)|^2$')
plt.legend()
plt.savefig('harmonic_oscillator_high_energy_prob.pdf')
plt.show()
```

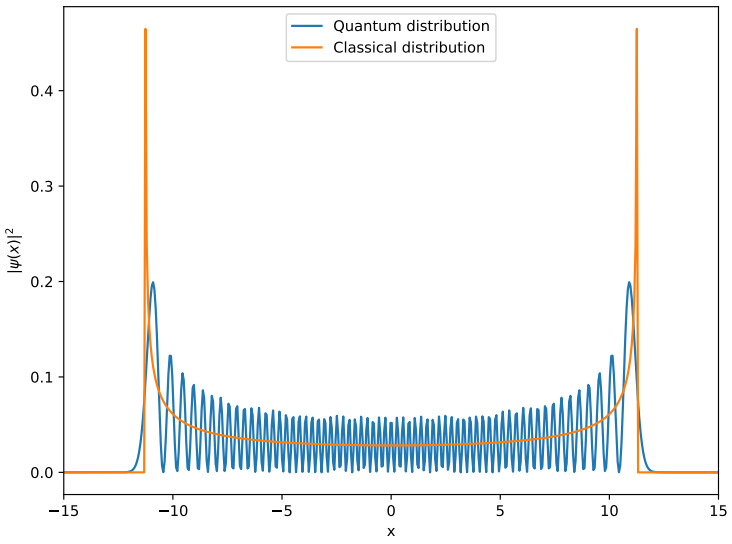

Figure 13: 調和振動子ポテンシャルの固有関数の様子。

7.4 基底関数展開法を用いた時間に依存しないシュレディンガー方程式の求解

前節までは、実空間差分法を用いてシュレディンガー方程式を解く方法について学んできた。

時間に依存しないシュレディンガー方程式に対しては、差分近似を用いることで、変微分方程式の固有値問題を行列の固有値問題へ書き換え、行列の対角化ルーチンにより解析を行った。この節では、実空間差分法とは異なる数値的方法である**基底関数展開法**について学ぶ。

8 時間に依存するハミルトニアンの下での量子ダイナミクス計算

8.1 時間に依存するハミルトニアンの下のでの時間発展

節 6では、時間に依存しないハミルトニアンの下での時間に依存するシュレディンガー方程式を数値的に解く方法について学んだ。この節では、その知見を発展させ、時間に依存するシュレディンガー方程式の下で時間に依存するシュレディンガー方程式を解く方法について学ぶ。このようなシミュレーションは、例えば、レーザー場の下での原子の中の電子ダイナミクスを調べるのに応用することが出来る。

時間に依存するシュレディンガー方程式を数値的に解く方法を考えるために、次のような時間に依存するハミルトニアンを含む時間依存シュレディンガー方程式を考えてみよう。

$$i\hbar\frac{d}{dt}\psi(x,t) = \hat{H}(t)\psi(x,t). \tag{96}$$

ハミルトニアンが時間に依存しない場合($H(t) = H_0$)は、シュレディンガー方程式の形式解を次のように書くことが出来た。

$$\psi(x,t) = \exp\left[-\frac{i}{\hbar}H_0 t\right]\psi(x,0). \tag{97}$$

しかしながら、ハミルトニアンが時間に依存する場合は、シュレディンガー方程式の形式解をこのような形で書くことはできない。そこで、ハミルトニアンが時間に依存する場合の形式解を求めるために、微小時間Δtだけ時間を進める微小時間発展を考えることにする。微小時間Δtを十分小さくとると、時刻tから$t+\Delta t$の微小時間の間のハミルトニアンの時間変化が十分小さくなり、ハミルトニアンの時間依存性を無視することが出来る。このような微小時間発展は、ハミルトニアンが時間によらない場合の形式解を用いることで、時刻t_0から$t_0+\Delta t$までの時間発展を次のように書くことが出来る。

$$\psi(x,t_0+\Delta t) = \exp\left[-\frac{i}{\hbar}H(t_0)\Delta t\right]\psi(x,t_0). \tag{98}$$

このような微小時間発展を繰り返し行うことで、シュレディンガー方程式の解を得ることが出来る。例えば、微小時間発展をN回行うことを考えると、時刻$t = t_N = \Delta t \times N$における波動関数は次のように表すことが出来る。

$$\begin{aligned}\psi(x,t_N) =& \exp\left[-\frac{i}{\hbar}H(t_{N-1})\Delta t\right]\times\exp\left[-\frac{i}{\hbar}H(t_{N-2})\Delta t\right]\times\cdots\\ &\times\exp\left[-\frac{i}{\hbar}H(t_1)\Delta t\right]\times\exp\left[-\frac{i}{\hbar}H(t_0)\Delta t\right]\psi(x,t_0).\end{aligned} \tag{99}$$

ただし、ここでは$t_j = t_0 + j\times\Delta t$と定義されている。

式(99)のように、各時刻でのハミルトニアンを用いた微小時間発展を繰り返し行うことによって、時間に依存するシュレディンガー方程式解を構築することができる。数値計算を行う際も、式(99)を用いて、微小時間発展を繰り返し実行することで逐次的に波動関数の時間発展を計算していく。

時間に依存するシュレディンガー方程式を解く前に、式(99)の形式解をより簡便な形に書き直すことを考えてみよう。もしハミルトニアンが演算子ではなくただの数であった場合、時刻t_0から時刻t_Nまでの時間発展演算子は次のように書くことが出来る。

$$U(t_N,t_0) = \exp\left[-\frac{i}{\hbar}H(t_{N-1})\Delta t\right]\times\cdots\times\exp\left[-\frac{i}{\hbar}H(t_0)\Delta t\right] \neq \exp\left[-\frac{i}{\hbar}\sum_{j=0}^{N-1}H(t_j)\Delta t\right]. \tag{100}$$

しかし、ハミルトニアンはただの数ではなく演算子であるため、式(100)のような単純な変形を行うことはできない。実際に、式(100)の両辺をテイラー展開してみると、左辺のテイラ

ー展開からは、異なる時刻のハミルトニアンの積(例えば、$H(t_2)H(t_1)H(t_0)$等)が時刻の順番に並ぶことが分かる(早い時刻のハミルトニアン$H(t_i)$)は、それより遅い時刻のハミルトニアン$H(t_j)$より必ず右側にある)。その一方で、右辺のテイラー展開においては、ハミルトニアンの積が様々な時間の順序で現れる(例えば、$H(t_1)H(t_2)H(t_0)$等)。したがって、異なる時刻のハミルトニアンが交換しない場合、式(100)のような書き換えができないことが分かる。

上記の議論で見方を変えると、式(100)の右辺のテイラー展開に現れるすべてのハミルトニアンの積について、それぞれ時間の早い順(例えば、$H(t_2)H(t_1)H(t_0)$等)に並び変える操作をすれば、左辺のテイラー展開と等しくなり、等号が成り立つことが分かる。このような時刻の順番に演算子の積を並べ替える操作を時間順序積(Time-ordered product, T-product, T積)と呼び、以下のように表現される。

$$\mathcal{T}\{H(t_0)H(t_2)H(t_1)\} = H(t_2)H(t_1)H(t_0). \tag{101}$$

このような時間順序積を用いると、式(99)の中の時間発展演算子は次のように表現することが出来る。

$$\begin{aligned} U(t_N,t_0) =& \exp\left[-\frac{i}{\hbar}H(t_{N-1})\Delta t\right]\times\exp\left[-\frac{i}{\hbar}H(t_{N-2})\Delta t\right]\times\cdots \\ & \times\exp\left[-\frac{i}{\hbar}H(t_1)\Delta t\right]\times\exp\left[-\frac{i}{\hbar}H(t_0)\right] \\ =& \mathcal{T}\left\{\exp\left[-\frac{i}{\hbar}\sum_0^{N-1}H(t_j)\Delta t\right]\right\} \\ =& \mathcal{T}\left\{\exp\left[-\frac{i}{\hbar}\int_{t_0}^{t_N}dtH(t)\right]\right\}. \end{aligned} \tag{102}$$

ここで、最後の行ではΔtが十分小さく和を積分に書き直せることを仮定した。このように、時間に依存するハミルトニアンの場合でも、時間発展演算子を時間順序積を用いて形式的に表現することが出来る。

8.2 振動する調和ポテンシャル内の量子波束のダイナミクス

時間に依存するハミルトニアンを用いた時間発展計算の例として、振動する1次元調和ポテンシャル内の量子波束のダイナミクスをシミュレーションしてみよう。解くべき方程式は、次のような時間に依存するシュレディンガー方程式である。

$$i\hbar\frac{\partial}{\partial t}\psi(x,t) = -\frac{\hbar^2}{2m}\frac{\partial^2\psi(x,t)}{\partial x^2} + \frac{m\omega_0^2}{2}\left(x-x_c(t)\right)^2\psi(x,t). \tag{103}$$

ここで、$x_c(t)$は時間とともに移動する調和ポテンシャルの中心である。また、この方程式の初期条件として、時刻$t=0$のハミルトニアンの基底状態を採用する。さらに簡単のため、ここからは$m=\hbar=\omega_0=1$として議論を進めよう。

この調和振動子の固有振動数($\omega_0=1$)に対し、(a)十分遅い時間スケールで$x_c(t)$が動く場合、(b)同じ時間スケールで$x_c(t)$が動く場合、(c)十分速い時間スケールで$x_c(t)$が動く場合について、計算を行ってみよう。具体的には、(a)十分遅い場合は、

$$x_c(t) = \cos(\Omega t) \tag{104}$$

とし、$\Omega = 0.2\omega_0$として計算を実行してみよう。また、(b)の共鳴条件では、

$$x_c(t) = \cos(\Omega t) \tag{105}$$

とし、$\Omega = \omega_0$として計算を実行してみよう。さらに、(c)の十分速い場合については、

$$x_c(t) = \sin(\Omega t) \tag{106}$$

とし、$\Omega = 8\omega_0$として計算を実行してみよう。

まずは、調和振動子のハミルトニアンの基底状態を求めるコードを書いてみよう。この基底状態が、時間発展の初期波動関数となる。以下のソースコード 32に、調和振動子基底状態を求めるPythonコードの例を示す。

https://github.com/shunsuke-sato/python_qe/blob/develop/note_comp_phys/src/qm_dynamics_td_ham_v1.py

ソースコード 32: 調和振動子基底状態を求めるコードの例

```python
from numba import jit
from matplotlib import pyplot as plt
import numpy as np
import matplotlib.animation as animation
from matplotlib.animation import PillowWriter


# Construct potential
def construct_potential(xj, xc):
    return 0.5*(xj-xc)**2

def calc_ground_state(xj, potential):

    num_grid = xj.size
    dx = xj[1]-xj[0]

    ham = np.zeros((num_grid, num_grid))

    for i in range(num_grid):
        for j in range(num_grid):
            if(i == j):
                ham[i,j] = -0.5*(-2.0/dx**2)+potential[i]
            elif(np.abs(i-j)==1):
                ham[i,j] = -0.5*(1.0/dx**2)

    eigenvalues, eigenvectors = np.linalg.eigh(ham)

    wf = np.zeros(num_grid, dtype=complex)

    wf.real = eigenvectors[:,0]/np.sqrt(dx)

    return wf

# time propagation parameters
Tprop = 40.0
dt = 0.005
#dt = 0.00905
nt = int(Tprop/dt)+1

# set the coordinate
xmin = -10.0
xmax = 10.0
num_grid = 250

xj = np.linspace(xmin, xmax, num_grid)

# set potential
xc = 1.0
potential = construct_potential(xj, xc)

# calculate the ground state
wf = calc_ground_state(xj, potential)



# plot the ground state density, |wf|^2
rho = np.abs(wf)**2

plt.figure(figsize=(8,6))
plt.plot(xj, rho, label="$|\psi(x)|^2$␣(calc.)")
plt.plot(xj, np.exp(-(xj-xc)**2)/np.sqrt(np.pi),
         label="$|\psi(x)|^2$␣(exact)", linestyle='dashed')
plt.plot(xj, 0.5*(xj-xc)**2,
         label="Harmonic␣potential", linestyle='dotted')

plt.xlim([-4.0, 4.0])
```

```
plt.ylim([0.0, 0.8])
plt.xlabel('x')
plt.ylabel('Density, Potential')
plt.legend()
plt.savefig('gs_density.pdf')
plt.show()
```

図 14には、上記のコードで得られた基底状態の密度が示されている。

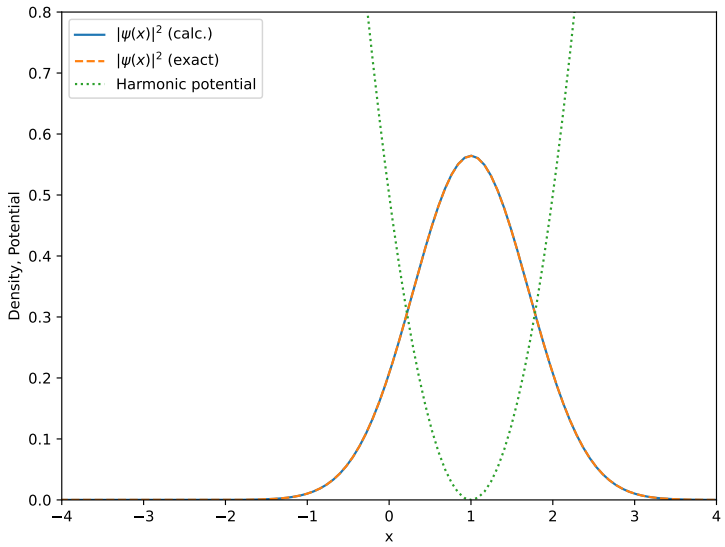

Figure 14: 調和振動子の変分法による解析

次に、上記のコードで得られた基底状態を初期条件として、波動関数の時間発展を調べるPythonコードを書いてみよう。時間に依存するハミルトニアンの下での時間発展は、式(99)で表されるように、各時刻のハミルトニアンを用いた微小時間発展演算子を複数回掛けることで記述することができる。この微小時間発展演算子を、4次のTaylor展開を用いて演算することで波動関数の時間発展を計算してみよう。

$$\begin{aligned}\psi(x,t_{j+1}) &\approx \exp\left[-\frac{i}{\hbar}\Delta t H(t_j)\right]\psi(x,t_j)\\ &\approx \sum_{n=0}^{4}\frac{1}{n!}\left(\frac{-i\Delta t}{\hbar}\right)^n H^n(t_j)\psi(x,t_j).\end{aligned} \tag{107}$$

以下のソースコード 33に、振動する調和振動子ポテンシャル内を運動する量子波束のダイナミクスを調べるPythonコードの例を示す。

https://github.com/shunsuke-sato/python_qe/blob/develop/note_comp_phys/src/qm_dynamics_td_ham_v2.py

ソースコード 33: 調和振動子基底状態を求めるコードの例

```
from numba import jit
from matplotlib import pyplot as plt
import numpy as np
import matplotlib.animation as animation
from matplotlib.animation import PillowWriter


# Construct potential
```

```
def construct_potential(xj, xc):
    return 0.5*(xj-xc)**2

def calc_ground_state(xj, potential):

    num_grid = xj.size
    dx = xj[1]-xj[0]

    ham = np.zeros((num_grid, num_grid))

    for i in range(num_grid):
        for j in range(num_grid):
            if(i == j):
                ham[i,j] = -0.5*(-2.0/dx**2)+potential[i]
            elif(np.abs(i-j)==1):
                ham[i,j] = -0.5*(1.0/dx**2)

    eigenvalues, eigenvectors = np.linalg.eigh(ham)

    wf = np.zeros(num_grid, dtype=complex)

    wf.real = eigenvectors[:,0]/np.sqrt(dx)

    return wf

# Operate the Hamiltonian to the wavefunction
@jit(nopython=True)
def ham_wf(wf, potential, dx):

    n = wf.size
    hwf = np.zeros(n, dtype=np.complex128)

    for i in range(1,n-1):
        hwf[i] = -0.5*(wf[i+1]-2.0*wf[i]+wf[i-1])/(dx**2)

    i = 0
    hwf[i] = -0.5*(wf[i+1]-2.0*wf[i])/(dx**2)
    i = n-1
    hwf[i] = -0.5*(-2.0*wf[i]+wf[i-1])/(dx**2)

    hwf = hwf + potential*wf

    return hwf


# Time propagation from t to t+dt
def time_propagation(wf, potential, dx, dt):

    n = wf.size
    twf = np.zeros(n, dtype=complex)
    hwf = np.zeros(n, dtype=complex)

    twf = wf
    zfact = 1.0 + 0j
    for iexp in range(1,5):
        zfact = zfact*(-1j*dt)/iexp
        hwf = ham_wf(twf, potential, dx)
        wf = wf + zfact*hwf
        twf = hwf

    return wf


# time propagation parameters
#omega = 8.0
omega = 0.2
Tprop = 80.0
dt = 0.005
nt = int(Tprop/dt)+1

# set the coordinate
xmin = -10.0
xmax = 10.0
num_grid = 250

xj = np.linspace(xmin, xmax, num_grid)
dx = xj[1] - xj[0]

# set potential
xc = 1.0
```

```
potential = construct_potential(xj, xc)

# calculate the ground state
wf = calc_ground_state(xj, potential)


# For loop for the time propagation
density_list = []
xc_list = []
for it in range(nt+1):
    tt = it*dt
    xc = np.cos(omega*tt)
    if(it % (nt//200) == 0):
        rho = np.abs(wf)**2
        density_list.append(rho.copy())
        xc_list.append(xc)


    potential = construct_potential(xj, xc)

    wf = time_propagation(wf, potential, dx, dt)
    print(it, nt)

# plot the density, |wf|^2

# Define function to update plot for each frame of the animation
def update_plot(frame):
    plt.cla()
    plt.xlim([-5.0, 5.0])
    plt.ylim([0.0, 0.8])
    xc = xc_list[frame]
    plt.plot(xj, density_list[frame], label="$|\psi(x)|^2$ (calc.)")
    plt.plot(xj, np.exp(-(xj-xc)**2)/np.sqrt(np.pi),
             label="$|\psi(x)|^2$ (ref.)", linestyle='dashed')
    plt.plot(xj, 0.5*(xj-xc)**2,
         label="Harmonic potential", linestyle='dotted')

    plt.xlabel('x')
    plt.ylabel('Density, Potential')
    plt.legend(loc = 'upper right')


# Create the animation
fig = plt.figure()
ani = animation.FuncAnimation(fig, update_plot, frames=len(density_list), interval=50)
#ani.save('wavefunction_animation.gif', writer='imagemagick')
ani.save('density_animation.gif', writer='pillow')
```

8.3 レーザー電場の下での1次元水素原子の電子ダイナミクス

時間に依存するハミルトニアンを用いた時間発展計算の応用例として、1次元水素原子にレーザー電場を印加し、駆動される電子のダイナミクスを調べるシミュレーションを行ってみよう。解くべき方程式として、次のような1次元の時間に依存するシュレディンガー方程式を考える。

$$
\begin{aligned}
i\hbar\frac{\partial}{\partial t}\psi(x,t) &= \left[\hat{H}_0 + \hat{V}(t)\right]\psi(x,t) \\
&= \left[-\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2} - \frac{1}{\sqrt{x^2+1}} - eE(t)x\right]\psi(x,t).
\end{aligned} \tag{108}
$$

ここで、ポテンシャル,$-1/\sqrt{x^2+1}$,はクーロンポテンシャルを模した1次元ポテンシャルであり、計算の困難を回避するために原点での発散を取り除いている。また、ポテンシャル,$-eE(t)x$,は一様電場$E(t)$に対応するスカラーポテンシャルである。また、便宜上、ハミルトニアン全体を次のように非摂動項$\hat{H}_0$と摂動項$\hat{V}(t)$に分けて定義しておく。

$$
\hat{H}_0 = -\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2} - \frac{1}{\sqrt{x^2+1}}, \tag{109}
$$

$$
\hat{V}(t) = -eE(t)x. \tag{110}
$$

今回の計算では、時刻$t=0$において波動関数$\psi(x,t=0)$を電場の無い$(E(t)=0)$場合のハミルトニアンの基底状態として設定し、時刻$t>0$において次のような振動電場を印加して電子のダイナミクスを計算する。

$$E(t)=\begin{cases} E_0\cos^2\left[\frac{\pi}{T_0}\left(t-\frac{T_0}{2}\right)\right]\sin\left[\omega_0\left(t-\frac{T_0}{2}\right)\right] & 0<t<T_0 \\ 0 & \text{otherwise} \end{cases} \tag{111}$$

実際の数値計算では、式(99)のように、Δtの微小時間発展を繰り返し行うことで波動関数の時間発展を計算していく。数値計算を実行しやすいように、この微小時間発展演算子を次のように近似することを考える。

$$\begin{aligned} \exp\left[-\frac{i}{\hbar}H(t_0)\Delta t\right] &= \exp\left[-\frac{i}{\hbar}\left(\hat{H}_0+\hat{V}(t_0)\right)\Delta t\right] \\ &= \exp\left[-\frac{i}{\hbar}\hat{V}(t_0)\frac{\Delta t}{2}\right] \\ &\quad \exp\left[-\frac{i}{\hbar}\hat{H}_0\Delta t\right]\exp\left[-\frac{i}{\hbar}\hat{V}(t_0)\frac{\Delta t}{2}\right]+\mathcal{O}(\Delta t^3). \end{aligned} \tag{112}$$

この近似式は、指数関数の各項をテイラー展開することで確認することができる。この近似式を用いて、式(111)で表される電場の下での電子ダイナミクスを計算してみよう。

https://github.com/shunsuke-sato/python_qe/blob/develop/note_comp_phys/src/qm_dynamics_hydrogen.py

ソースコード 34: レーザー電場の下での電子ダイナミクス計算コード例

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from matplotlib.animation import PillowWriter

# define the potential
def calc_potential(xj):
    vpot = -1.0/np.sqrt(xj**2+1.0)
    return vpot


def calc_static_hamiltonian(num_grid, xj, vpot):

    ham = np.zeros((num_grid, num_grid))
    dx = xj[1]-xj[0]

    for i in range(num_grid):
        for j in range(num_grid):
            if(i == j):
                ham[i,j] = -0.5*(-2.0/dx**2)+vpot[i]
            elif(np.abs(i-j)==1):
                ham[i,j] = -0.5*(1.0/dx**2)


    return ham


def calc_gs_wf(num_grid, ham):

    eigenvalues, eigenvectors = np.linalg.eigh(ham)
    print("gs energy =", eigenvalues[0])

    wf = np.zeros(num_grid, dtype=complex)
    wf.real = eigenvectors[:,0]
    wf[0] = 0.0
    wf[-1] = 0.0

    return wf


def calc_laser_field(tt):

```

```
    omega0 = 0.05
    E0 = 0.1
    tpulse = 10*2.0*np.pi/omega0
    xx = tt-0.5*tpulse

    if(np.abs(xx)/tpulse < 0.5):
        Et = E0*np.cos(np.pi*xx/tpulse)**2*np.sin(omega0*xx)
    else:
        Et = 0.0

    return Et



def ham_wf(wf, vpot, dx):

    num_grid = wf.size
    hwf = np.zeros(num_grid, dtype=complex)

    for i in range(1, num_grid-1):
        hwf[i] = -0.5*(wf[i+1]-2.0*wf[i]+wf[i-1])/(dx**2)

    hwf = hwf + vpot*wf
    return hwf


def time_propagation(xj, wf, vpot, dx, tt, dt):

    Et = calc_laser_field(tt)
    v_Et = -Et*xj

    # apply exp(-0.5*0j*v_Et*dt)
    wf = wf*np.exp(-0.5*1j*v_Et*dt)

    # propagate
    twf = wf


    zfact = 1.0 + 0j
    for iexp in range(1,5):
        zfact = zfact*(-1j*dt)/iexp
        hwf = ham_wf(twf, vpot, dx)
        wf = wf + zfact*hwf
        twf = hwf


    # apply exp(-0.5*0j*v_Et*dt)
    wf = wf*np.exp(-0.5*1j*v_Et*dt)

    return wf

def calc_dipole(xj, wf):

    dx = xj[1]-xj[0]
    dipole = np.sum(np.abs(wf)**2*xj)*dx

    return dipole


# Set the coordinate
xmin = -50.0
xmax = 50.0
num_grid = 500

xj = np.linspace(xmin, xmax, num_grid)
dx = xj[1]-xj[0]

# Time propagation parameters
Tprop = 1300.0 #80.0
dt = 0.05
nt = int(Tprop/dt)+1



# set the potential
vpot = calc_potential(xj)

# set the static Hamiltonian
ham = calc_static_hamiltonian(num_grid, xj, vpot)
```

```
123
124 # set the initial wavefunction (ground state)
125 wf = calc_gs_wf(num_grid, ham)
126
127 # set output quantities
128 tt_out = np.zeros(nt)
129 Et_out = np.zeros(nt)
130 dipole_out = np.zeros(nt)
131 norm_out = np.zeros(nt)
132
133 # wavefunction array to make a movie
134 wavefunctions = []
135
136 # For loop for the time propagation
137 for it in range(nt):
138     if(it%(nt//100) == 0):
139         print("it=",it,nt)
140         wavefunctions.append(wf.copy())
141
142     tt = dt*it
143
144     # compute outputs
145     tt_out[it] = dt*it
146     Et_out[it] = calc_laser_field(tt)
147     dipole_out[it] = calc_dipole(xj, wf)
148     norm_out[it] = np.sum(np.abs(wf)**2)*dx
149
150     wf = time_propagation(xj, wf, vpot, dx, tt, dt)
151
152
153 plt.figure()
154 plt.plot(tt_out, Et_out, label="E(t)")
155 plt.plot(tt_out, dipole_out, label="d(t)")
156
157 plt.xlabel("t")
158 plt.ylabel("E(t),␣d(t)")
159 plt.legend()
160
161 plt.savefig("dipole_t.png")
162
163
164 # Define function to update plot for each frame of the animation
165 def update_plot(frame):
166     plt.cla()
167     plt.xlim([-20, 20])
168     plt.ylim([0.0, 0.12])
169     plt.plot(xj, np.abs(wavefunctions[frame])**2, label="Density")
170     plt.xlabel('$x$')
171     plt.ylabel('$Density$')
172     plt.legend()
173
174 # Create the animation
175 fig = plt.figure()
176 ani = animation.FuncAnimation(fig, update_plot, frames=len(wavefunctions), interval=50)
177 #ani.save('wavefunction_animation.gif', writer='imagemagick')
178 ani.save('density_animation.gif', writer='pillow')
```

8.4 吸収ポテンシャル

前節の計算では、原子からイオン化した電子がシミュレーション・ボックスの境界で反射される非物理的な現象が起こってしまう。そこで、レーザー電場によってイオン化した電子をを吸収するために純虚数のポテンシャルを導入することを考える。次のような複素ポテンシャルを考え、イオン化された電子のノルムが時間とともに減っていく吸収境界条件を課すことを考えよう。

$$\hat{V}(t) = \begin{cases} -eE(t)x & |x| < 40 \\ -eE(t)x - i\left(|x| - 40\right) & |x| < 40 \end{cases} \tag{113}$$

この吸収ポテンシャルを摂動に加えることによって、イオン化された電子をポテンシャルによって吸収し、シミュレーション・ボックス境界における非物理的な緩和の影響を緩和することができる。実際に、吸収ポテンシャルを導入した計算を実行し、その影響を確認してみよう。

https://github.com/shunsuke-sato/python_qe/blob/develop/note_comp_phys/src/qm_dynamics_hydrogen_abs.py

ソースコード 35: レーザー電場の下での電子ダイナミクス計算コード例

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from matplotlib.animation import PillowWriter

# define the potential
def calc_potential(xj):
    vpot = -1.0/np.sqrt(xj**2+1.0)
    return vpot


def calc_static_hamiltonian(num_grid, xj, vpot):

    ham = np.zeros((num_grid, num_grid))
    dx = xj[1]-xj[0]

    for i in range(num_grid):
        for j in range(num_grid):
            if(i == j):
                ham[i,j] = -0.5*(-2.0/dx**2)+vpot[i]
            elif(np.abs(i-j)==1):
                ham[i,j] = -0.5*(1.0/dx**2)


    return ham


def calc_gs_wf(num_grid, ham):

    eigenvalues, eigenvectors = np.linalg.eigh(ham)
    print("gs energy =", eigenvalues[0])

    wf = np.zeros(num_grid, dtype=complex)
    wf.real = eigenvectors[:,0]
    wf[0] = 0.0
    wf[-1] = 0.0

    return wf


def calc_laser_field(tt):

    omega0 = 0.05
    E0 = 0.1
    tpulse = 10*2.0*np.pi/omega0
    xx = tt-0.5*tpulse

    if(np.abs(xx)/tpulse < 0.5):
        Et = E0*np.cos(np.pi*xx/tpulse)**2*np.sin(omega0*xx)
    else:
        Et = 0.0

    return Et



def ham_wf(wf, vpot, dx):

    num_grid = wf.size
    hwf = np.zeros(num_grid, dtype=complex)

    for i in range(1, num_grid-1):
        hwf[i] = -0.5*(wf[i+1]-2.0*wf[i]+wf[i-1])/(dx**2)

    hwf = hwf + vpot*wf
    return hwf


def time_propagation(xj, wf, vpot, dx, tt, dt):

    num_grid = xj.size

```

```
    Et = calc_laser_field(tt)
    v_Et = -Et*xj

    # set absorbing boundary
    v_abs = np.zeros(num_grid, dtype=complex)
    for i in range(num_grid):
        if(np.abs(xj[i]) > 40.0):
            v_abs[i] = -0.2*1j*(np.abs(xj[i]) - 40.0)



    # apply exp(-0.5*0j*v_Et*dt)
    wf = wf*np.exp(-0.5*1j*(v_Et+v_abs)*dt)

    # propagate
    twf = wf


    zfact = 1.0 + 0j
    for iexp in range(1,5):
        zfact = zfact*(-1j*dt)/iexp
        hwf = ham_wf(twf, vpot, dx)
        wf = wf + zfact*hwf
        twf = hwf


    # apply exp(-0.5*0j*v_Et*dt)
    wf = wf*np.exp(-0.5*1j*(v_Et+v_abs)*dt)

    return wf

def calc_dipole(xj, wf):

    dx = xj[1]-xj[0]
    dipole = np.sum(np.abs(wf)**2*xj)*dx

    return dipole


# Set the coordinate
xmin = -50.0
xmax = 50.0
num_grid = 500

xj = np.linspace(xmin, xmax, num_grid)
dx = xj[1]-xj[0]

# Time propagation parameters
Tprop = 1300.0 #80.0
dt = 0.05
nt = int(Tprop/dt)+1




# set the potential
vpot = calc_potential(xj)

# set the static Hamiltonian
ham = calc_static_hamiltonian(num_grid, xj, vpot)

# set the initial wavefunction (ground state)
wf = calc_gs_wf(num_grid, ham)

# set output quantities
tt_out = np.zeros(nt)
Et_out = np.zeros(nt)
dipole_out = np.zeros(nt)
norm_out = np.zeros(nt)

# wavefunction array to make a movie
wavefunctions = []

# For loop for the time propagation
for it in range(nt):
    if(it%(nt//100) == 0):
        print("it=",it,nt)
        wavefunctions.append(wf.copy())

    tt = dt*it
```

```
153
154     # compute outputs
155     tt_out[it] = dt*it
156     Et_out[it] = calc_laser_field(tt)
157     dipole_out[it] = calc_dipole(xj, wf)
158     norm_out[it] = np.sum(np.abs(wf)**2)*dx
159
160     wf = time_propagation(xj, wf, vpot, dx, tt, dt)
161
162
163 plt.figure()
164 plt.plot(tt_out, Et_out, label="E(t)")
165 plt.plot(tt_out, dipole_out, label="d(t)")
166
167 plt.xlabel("t")
168 plt.ylabel("E(t),␣d(t)")
169 plt.legend()
170
171 plt.savefig("dipole_t_abs.png")
172
173
174 # Define function to update plot for each frame of the animation
175 def update_plot(frame):
176     plt.cla()
177     plt.xlim([-20, 20])
178     plt.ylim([0.0, 0.12])
179     plt.plot(xj, np.abs(wavefunctions[frame])**2, label="Density")
180     plt.xlabel('$x$')
181     plt.ylabel('$Density$')
182     plt.legend()
183
184 # Create the animation
185 fig = plt.figure()
186 ani = animation.FuncAnimation(fig, update_plot, frames=len(wavefunctions), interval=50)
187 #ani.save('wavefunction_animation.gif', writer='imagemagick')
188 ani.save('density_animation_abs.gif', writer='pillow')
```

8.5 高次高調波発生の解析

上のシミュレーションでは、原子に高強度レーザーパルスが照射された際に駆動される電子のダイナミクスを調べた。その解析の結果、光電場が誘起する電気双極子モーメント$d(t)$(この場合は、位置の期待値と等価)を時間の関数として計算することが出来る。また、荷電粒子が加速度運動に伴って電磁波が放射されるので、荷電粒子の加速度の振動数成分を調べることで、光駆動された電子系から放出される光のスペクトルを計算することが出来る。具体的には、次のように双極子モーメントのフーリエ変換を調べると便利である。

$$\tilde{d}(\omega) = \int_{-\infty}^{\infty} dt e^{i\omega t} d(t). \tag{114}$$

ここで、電子の加速度$a(t)$が双極子モーメントの2回時間微分に比例することを考えると、光駆動された電子系から放出される光のスペクトルは次のように表される。

$$I(\omega) \sim \omega^2 \left|\tilde{d}(\omega)\right|^2. \tag{115}$$

また、実際のシミュレーションでは無限に長い時間を取り扱うことはできないので、式(114)の代わりに有限の長さの時間でのフーリエ変換を行うことにしよう。

$$\tilde{d}(\omega) = \int_{0}^{T_{\mathrm{sim}}} dt e^{i\omega t} d(t) W(t). \tag{116}$$

ここで、T_{sim}はシミュレーション時間であり、関数$W(t)$は$t = 0 = T_{\sim}$で滑らかにゼロになる窓関数である。このような窓関数を導入することで、有限の時間でのフーリエ変換に現れるノイズを低減できる。

前回までに作った電子ダイナミクス計算の末尾に、上記のフーリエ変換を用いた解析コードを追加して、実際に光励起された原子から発生する光のスペクトルを計算してみよう。

https://github.com/shunsuke-sato/python_qe/blob/develop/note_comp_phys/src/qm_dynamics_hydrogen_abs_fft.py

ソースコード 36: レーザー電場の下での電子ダイナミクス計算と高次高調波スペクトル解析コード例

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from matplotlib.animation import PillowWriter

# define the potential
def calc_potential(xj):
    vpot = -1.0/np.sqrt(xj**2+1.0)
    return vpot


def calc_static_hamiltonian(num_grid, xj, vpot):

    ham = np.zeros((num_grid, num_grid))
    dx = xj[1]-xj[0]

    for i in range(num_grid):
        for j in range(num_grid):
            if(i == j):
                ham[i,j] = -0.5*(-2.0/dx**2)+vpot[i]
            elif(np.abs(i-j)==1):
                ham[i,j] = -0.5*(1.0/dx**2)


    return ham


def calc_gs_wf(num_grid, ham):

    eigenvalues, eigenvectors = np.linalg.eigh(ham)
    print("gs␣energy␣=", eigenvalues[0])

    wf = np.zeros(num_grid, dtype=complex)
    wf.real = eigenvectors[:,0]
    wf[0] = 0.0
    wf[-1] = 0.0

    return wf


def calc_laser_field(tt):

    omega0 = 0.05
    E0 = 0.1
    tpulse = 10*2.0*np.pi/omega0
    xx = tt-0.5*tpulse

    if(np.abs(xx)/tpulse < 0.5):
        Et = E0*np.cos(np.pi*xx/tpulse)**2*np.sin(omega0*xx)
    else:
        Et = 0.0

    return Et



def ham_wf(wf, vpot, dx):

    num_grid = wf.size
    hwf = np.zeros(num_grid, dtype=complex)

    for i in range(1, num_grid-1):
        hwf[i] = -0.5*(wf[i+1]-2.0*wf[i]+wf[i-1])/(dx**2)

    hwf = hwf + vpot*wf
    return hwf


def time_propagation(xj, wf, vpot, dx, tt, dt):

    num_grid = xj.size
```

```
72
73      Et = calc_laser_field(tt)
74      v_Et = -Et*xj
75
76      # set absorbing boundary
77      v_abs = np.zeros(num_grid, dtype=complex)
78      for i in range(num_grid):
79          if(np.abs(xj[i]) > 40.0):
80              v_abs[i] = -0.2*1j*(np.abs(xj[i]) - 40.0)
81
82
83
84      # apply exp(-0.5*0j*v_Et*dt)
85      wf = wf*np.exp(-0.5*1j*(v_Et+v_abs)*dt)
86
87      # propagate
88      twf = wf
89
90
91      zfact = 1.0 + 0j
92      for iexp in range(1,5):
93          zfact = zfact*(-1j*dt)/iexp
94          hwf = ham_wf(twf, vpot, dx)
95          wf = wf + zfact*hwf
96          twf = hwf
97
98
99      # apply exp(-0.5*0j*v_Et*dt)
100     wf = wf*np.exp(-0.5*1j*(v_Et+v_abs)*dt)
101
102     return wf
103
104 def calc_dipole(xj, wf):
105
106     dx = xj[1]-xj[0]
107     dipole = np.sum(np.abs(wf)**2*xj)*dx
108
109     return dipole
110
111
112 # Set the coordinate
113 xmin = -50.0
114 xmax = 50.0
115 num_grid = 500
116
117 xj = np.linspace(xmin, xmax, num_grid)
118 dx = xj[1]-xj[0]
119
120 # Time propagation parameters
121 Tprop = 1300.0 #80.0
122 dt = 0.05
123 nt = int(Tprop/dt)+1
124
125
126
127
128 # set the potential
129 vpot = calc_potential(xj)
130
131 # set the static Hamiltonian
132 ham = calc_static_hamiltonian(num_grid, xj, vpot)
133
134 # set the initial wavefunction (ground state)
135 wf = calc_gs_wf(num_grid, ham)
136
137 # set output quantities
138 tt_out = np.zeros(nt)
139 Et_out = np.zeros(nt)
140 dipole_out = np.zeros(nt)
141 norm_out = np.zeros(nt)
142
143 # wavefunction array to make a movie
144 wavefunctions = []
145
146 # For loop for the time propagation
147 for it in range(nt):
148     if(it%(nt//100) == 0):
149         print("it=",it,nt)
150         wavefunctions.append(wf.copy())
151
```

```
    tt = dt*it

    # compute outputs
    tt_out[it] = dt*it
    Et_out[it] = calc_laser_field(tt)
    dipole_out[it] = calc_dipole(xj, wf)
    norm_out[it] = np.sum(np.abs(wf)**2)*dx

    wf = time_propagation(xj, wf, vpot, dx, tt, dt)


plt.figure()
plt.plot(tt_out, Et_out, label="E(t)")
plt.plot(tt_out, dipole_out, label="d(t)")

plt.xlabel("t")
plt.ylabel("E(t), d(t)")
plt.legend()

plt.savefig("dipole_t_abs.png")


# Define function to update plot for each frame of the animation
def update_plot(frame):
    plt.cla()
    plt.xlim([-20, 20])
    plt.ylim([0.0, 0.12])
    plt.plot(xj, np.abs(wavefunctions[frame])**2, label="Density")
    plt.xlabel('$x$')
    plt.ylabel('$Density$')
    plt.legend()

# Create the animation
fig = plt.figure()
ani = animation.FuncAnimation(fig, update_plot, frames=len(wavefunctions), interval=50)
#ani.save('wavefunction_animation.gif', writer='imagemagick')
ani.save('density_animation_abs.gif', writer='pillow')


###  Analysis of HHG ###
def apply_envelope(nt, dt, ft):

    omega0 = 0.05
    tpulse = 10*2.0*np.pi/omega0
    ft_env = np.zeros(nt)

    for it in range(nt):
        tt = it*dt
        xx = tt-0.5*tpulse
        if(np.abs(xx)/tpulse < 0.5):
          ft_env[it] = ft[it]*np.cos(np.pi*xx/tpulse)**2
    return ft_env

# Apply envelope function
Et_env = apply_envelope(nt, dt, Et_out)
dipole_env = apply_envelope(nt, dt, dipole_out)

# Apply Fourier transform
Ew_out = np.fft.fft(Et_env)
spec_Ew = np.abs(Ew_out)**2

dipole_w_out = np.fft.fft(dipole_env)
spec_dipole = np.abs(dipole_w_out)**2

# Compute the frequency
omega = np.fft.fftfreq(nt, d=dt)*(2.0*np.pi)


# Figure 1
plt.figure()
plt.plot(omega, omega**2*spec_Ew, label="$E(\omega)$")
plt.plot(omega, omega**2*spec_dipole, label="$d(\omega)$")

plt.xlabel("$\omega$")
plt.ylabel("$|E(\omega)|^2$, $|d(\omega)|^2$")

plt.xlim(0,6)
plt.ylim(1e-10,1e4)
plt.yscale("log")
plt.legend()
```

```
plt.savefig("hhg_spec_1.png")

# Figure 2
omega0 = 0.05
plt.figure()
plt.plot(omega/omega0, omega**2*spec_Ew, label="$E(\omega)$")
plt.plot(omega/omega0, omega**2*spec_dipole, label="$d(\omega)$")

plt.xlabel("$\omega␣/␣\omega_0$")
plt.ylabel("$|E(\omega)|^2$,␣$|d(\omega)|^2$")

plt.xticks(np.arange(1,22,2))
plt.xlim(0,22)
plt.ylim(1e-10,1e4)
plt.yscale("log")
plt.grid(axis='x')
plt.legend()

plt.savefig("hhg_spec_2.png")
```

上で見たような高強度レーザーを原子に照射した際に生じる高次高調波発生では、放出される光のエネルギーがある閾値よりも高くなると、急激に光の強度が落ちることが知られている。このエネルギーをカットオフエネルギーと呼び、高次高調波発生が観測された当時は、このカットオフエネルギーを決めるメカニズムが大きな謎であった。その後、Paul Corkumによって半古典的な3ステップ模型が提唱され、カットオフエネルギーが次の式であらわされることを明らかにした。

$$U_{\mathrm{cutoff}} = I_p + 3.17U_p. \tag{117}$$

ここで、I_pはイオン化ポテンシャルであり、U_pはポンデルモーティブ・エネルギーと呼ばれる量で、振動電場中の荷電粒子の持つ平均運動エネルギーを表す。U_pは次の式によって定義されている。

$$U_p = \frac{e^2E_0^2}{4m_e\omega_0^2}. \tag{118}$$

この式を理解するために、振動電場中での電子のNewton方程式を考えてみよう。

$$m_e\frac{d}{dt}v(t) = -eE_0\cos(\omega_0 t). \tag{119}$$

したがって、振動電場の下での電子の速度は次式で与えられる。

$$v(t) = -\frac{e}{m_e}E_0\sin(\omega_0 t). \tag{120}$$

また、運動エネルギーの時間平均(電場振動の1周期Tの平均)は次式によって与えられる。

$$< E_k >= \frac{m_e}{2}\frac{1}{T}\int_0^T dtv^2(t) = \frac{m_e}{2}\frac{1}{T}\int_0^T dt\frac{e^2}{m_e^2}E_0^2\sin^2(\omega_0 t) = \frac{E_0^2}{4m_e\omega_0^2} = U_p. \tag{121}$$

上のシミュレーション中では、$E_0 = 0.1$, $\omega_0 = 0.05$, $e = m_e = 1$と設定しており、模型原子のイオン化ポテンシャルは$I_p = 0.67$である。したがって、3ステップ模型から予想されるカットオフエネルギーの値は$U_{\mathrm{cutoff}} = I_p + 3.17U_p \approx 3.84$である。この値と、上記の計算で得られたスペクトルを比較してみよう。