Physics, Numerical Methods and Python (DRAFT)

Peter Nightingale
Department of Physics
University of Rhode island
Kingston, RI 02881
United States
email: nigh@pobox.com

Tuesday 14th December, 2021

Contents

1 Introduction to the World According to Python

1.1 Jupyter Notebook

We'll share work in this course with Jupyter Notebooks. You can launch a notebook by clicking on the Jupyter Notebook icon installed by Anaconda—if that is what you used— or by typing `jupyter notebook` in a terminal window or by opening the the command prompt; `python3 -m notebook` may also work, if you

Table 1: A typical exchange with Python in a terminal window; single argument case

```
>>> def greetings1(text):
...    print('you entered: ', text)
...
>>> greetings1('ladida!')
you entered:  ladida!
>>>
```

Table 2: A typical exchange with Python in a terminal window; variable number of arguments

```
>>> def greetings2(*args):
...    for arg in args:
...        print(arg)
...
>>> greetings2('abandon', 'all', 'hope')
abandon
all
hope
>>>
```

system can find the `python3` command. The details depend on the operating system you use. There is more information on the Web.[1]

There are a lot of examples in these notes. If you copy and paste from these notes keep in mind that the single and double quotation marks can cause problems. They may produce characters that look right but may be misinterpreted nonetheless and produce strange error messages. The safest way to deal with this problem is to erase and retype these characters.

Make sure that you use Python 3.x, which should be an option when you open a new notebook or run it from the command line. It should also be listed at the top right in open a Jupyter Notebook.

To create a notebook click on `New` in the upper right corner of the window that opens; make sure to select `Python 3`. Alternatively, you can click on a notebook you created previously in the list that shows up. If you create a new notebook, type `1+1` in the `In[ ]` box and then `<<shift><return>>` both keys at the same time but `<shift>` first. In general, `<name>` refers to a key on your keyboard; `<<key 1><key 2>>` indicates pressing two keys at the same time in the order shown.

When you upload a Jupyter Notebook the code is displayed but it is **not run.** To run the code click on the double triangle pointing right in the tool bar and then on `Restart and Run All Cells`.

Some of the illustrations in these notes look like the ones in Tables 1-3. This what you get from running Python from the command line in an interactive shell rather than from inside a Jupyter Notebook. `>>>` means that Python is asking for input, which in this example is `def greetings(**hello):` followed by `<return>`. Python responds with `...` and the colon at the end of the first line requires indentation, as does the next line. The definition `def greetings` ends with a blank line and a `<return>`.

Tables 1-3 also illustrate different ways of passing arguments:

1. Precisely one argument;

2. A variable number of arguments;

3. A variable number of keyed arguments, identifying them.

In a Jupyter Notebook the rules for indentation are the same as in an interactive shell, but there are no `>>>` and `...`. The instructions are executed by `<<shift><return>>`. In either environment, comments are preceded by `#`.

---

[1] *Jupyter/iPython Notebook Quick Start Guide.* URL: https://jupyter-notebook-beginner-guide.readthedocs.io/en/latest/what_is_jupyter.html.

Table 3: A typical exchange with Python in a terminal window; variable number of keyed arguments

```
>>> 1+1
2
>>> def greetings(**hello): # search the Web for '*args and **kwargs'
...    for key, value in hello.items():
...        print('%s = %s' % (key, value))
...
>>> greetings(a = 'Hello!', b = 'Not bad', c = 'See you later')
a = Hello!
b = Not bad
c = See you later
>>>
```

Markdown is a mark-up—ha ha—language that can be used to interpolate comments and equations in Jupyter Notebooks. These comments can use regular text, and formats derived from HTML, and LaTeX. Here is a brief overview: Learn How to Write Markdown & LaTeX in The Jupyter Notebook. Look in particular for the section "complex maths and physics equations."

A general reference to Python as used for computational physics is the following: *Scipy Lecture Notes— One document to learn numerics, science, and data with Python.*

1.1.1 Before you begin

1. Take the *User Interface Tour,* the first item on the *Help* menu that appears once have opened a notebook from the initial page that displays a list of files (aka documents). You get out of the tour by using the <esc> key or some key like it. These kind of details are probably different on different systems and for different implementations on the same system.

2. Look at the second menu item *Keyboard Shortcuts;* those are different for different operating systems and keyboards. You may be able to have these shortcuts pop up by clicking in the notebook window somewhere outside an In [ ] box and then hitting an <h>. Try other keys, such as <a>, <b>, <p>, <q>, and <x> to see what happens.

3. Remember that there is help and there are reference links for:

   (a) Notebook
   (b) Markdown
   (c) Python
   (d) SymPy

4. If you click on the background and subsequently on, for instance' a `p`, you get a list of commands; experiment with this or be confused when it happens "spontaneously" :-)

Also see Table 4.

Table 4: Useful keyboard shortcuts and help in Jupyter Notebooks

<esc>a	insert cell above
<esc>b	insert cell below
<esc>dd	delete cell below
<esc>h	show keyboard shortcuts
<esc>x	delete cell
<<shift><return>>	run cell
<tab>	list possible ways to complete command
help('some built-in function')	provide help on 'some builtin function'

1.2 W3Schools

For basic Python a good place to start is the W3Schools web site. It has an elementary introduction to Python with try-it-yourself examples.[2]

The first element of a Python list and a NumPy array has index zero. Arrays and lists are similar but different in important, and at times, confusing ways. NumPy arrays are important because they can speed up and simplify code.

1.3 Accessing arrays and lists

Tables 5 and 6 show some examples that show how to access individual elements of lists and arrays and how to reverse arrays.

Table 5: Array access examples: the `>>>` is the Python prompt; what follows is the user command. Lines without promts are Python output.

```
>>> import numpy as np
>>> a = np.array([1,2,3,4,5])
>>> a[::-1] # reverse array
array([5, 4, 3, 2, 1])
>>> a[0] # first element
1
>>> a[-1] # last element
5
>>> a[3:] # drop first 3 elements
array([4, 5])
>>> a[:-2] # drop last 2 elements
array([1, 2, 3])
```

Table 6: NumPy enumerate supplies an automatic running index associated with the array elements

```
>>> a = [5,4,3,2,1]
>>> for ind, elem in enumerate(a):
...    print('index = ', ind, 'element = ', elem)
...
index =  0 element =  5
index =  1 element =  4
index =  2 element =  3
index =  3 element =  2
index =  4 element =  1
>>> for ind, elem in enumerate(a,1): # 1 overrides default starting value 0
...    print('index = ', ind, 'element = ', elem)
...
index =  1 element =  5
index =  2 element =  4
index =  3 element =  3
index =  4 element =  2
index =  5 element =  1
```

Table 7 shows several ways of obtaining the rows and columns of a matrix. The examples in Table 8 demonstrate generalized Hadamard products, aka an element-wise matrix or Schur product Schur product— see Hadamard product .

---

[2]w3schools. *Python Tutorial*. URL: https://www.w3schools.com/python/.

The standard definition of the Hadamard product is as follows. If

$$\boldsymbol{A} = \begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1q} \\ a_{21} & a_{22} & \ldots & a_{2q} \\ \vdots & \vdots & \vdots & \vdots \\ a_{p1} & a_{p2} & \ldots & a_{pq} \end{pmatrix} \tag{1.1}$$

and

$$\boldsymbol{B} = \begin{pmatrix} b_{11} & b_{12} & \ldots & b_{1q} \\ b_{21} & b_{22} & \ldots & b_{2q} \\ \vdots & \vdots & \vdots & \vdots \\ b_{p1} & b_{p2} & \ldots & b_{pq} \end{pmatrix} \tag{1.2}$$

then

$$\boldsymbol{A} \circ \boldsymbol{B} = \boldsymbol{B} \circ \boldsymbol{A} = \begin{pmatrix} a_{11}b_{11} & a_{12}b_{12} & \ldots & a_{1q}b_{1q} \\ a_{21}b_{21} & a_{22}b_{22} & \ldots & a_{2q}b_{2q} \\ \vdots & \vdots & \vdots & \vdots \\ a_{p1}b_{p1} & a_{p2}b_{p2} & \ldots & a_{pq}b_{pq} \end{pmatrix} \tag{1.3}$$

Notice that in contrast to the standard matrix product, the Hadamard product is commutative. In addition, as shown in Table 8, Python has the generalization in which one of these matrices can be $1 \times q$ or $p \times 1$. The Hadamard product in that case is obtained by first replicating that one row or column p respectively q times, or alternatively by considering the full column and rows of the other matrix to be matrix elements.

Table 7 has a simple example showing how to define and transpose matrices.[3] The example also shows how to recover their rows and columns and how to multiply rows and columns by arrays of numbers. These products are **not** matrix product, but row- and column-wise products, akin to Hadamard products.[4]

---

[3] J. VanderPlas. *The Basics of NumPy Arrays*. URL: https://jakevdp.github.io/PythonDataScienceHandbook/02.02-the-basics-of-numpy-arrays.html.

[4] Wikipedia. *Hadamard product (matrices)*. URL: https://en.wikipedia.org/wiki/Hadamard_product_(matrices).

Table 7: Matrix operations: extracting rows and columns, transposition

```
>>> import numpy as np
>>> A = np.array([[1,2,3],[4,5,6],[7,8,9],[10,11,12]])
>>> print(A)
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
>>> print(A.T) # transpose A
[[ 1  4  7 10]
 [ 2  5  8 11]
 [ 3  6  9 12]]
>>> for i, row in enumerate(A, 1):
...    print('row ',i, ': ',row)
...
row  1 :  [1 2 3]
row  2 :  [4 5 6]
row  3 :  [7 8 9]
row  4 :  [10 11 12]
>>> for i, column in enumerate(A.T, 1):
...     print('column ',i, ': ', column)
...
column  1 :  [ 1  4  7 10]
column  2 :  [ 2  5  8 11]
column  3 :  [ 3  6  9 12]
>>> shp = np.shape(A)
>>> type(shp)
<class 'tuple'>
>>> for i in range(0,shp[0]):
...     print('row ',i+1 ,A[i,:])
...
row  1 [1 2 3]
row  2 [4 5 6]
row  3 [7 8 9]
row  4 [10 11 12]
>>> for j in range(0,shp[1]):
...     print('column ', j+1, A[:,j])
...
column  1 [ 1  4  7 10]
column  2 [ 2  5  8 11]
column  3 [ 3  6  9 12]
```

Table 8: Matrix operations: generalized Hadamard products

```
>>> c =[10,100,1000]
>>> print(np.shape(c))
(3,)
>>> print('c =', c)
c = [10, 100, 1000]
>>> print('c*A =\n', c*A) # Multiplying the columns by 10, 100 and 1000 respectively:
c*A =
[[   10   200  3000]
 [   40   500  6000]
 [   70   800  9000]
 [  100  1100 12000]]
>>> c =[[10],[100],[1000],[10000]]
>>> print(np.shape(c))
(4, 1)
>>> print('c =', c)
c = [[10], [100], [1000], [10000]]
>>> print('c*A =\n', c*A) # Multiplying the rows by 10, 100, 1000 and 1000 respectively:
c*A =
[[    10     20     30]
 [   400    500    600]
 [  7000   8000   9000]
 [100000 110000 120000]]
```

1.3.1   Beware of types

Be careful to make sure that lists and NumPy arrays have the correct data types. Failing to distinguish between integers and floating point numbers can produce wrong results. Probably the most obvious way to make the distinction is to use a decimal point to make the distinction: use `1` for the integer and `1.0` for the floating point number; `1.` will work too but adding the trailing `0` is clearer. The examples in Table 9 illustrate what can go wrong when floating points are automatically and unexpectedly truncated to integers.

Table 9: Illustrating integer and floating point types, truncations and truncation errors

```
>>> import numpy as np
>>> a = np.array([1,2,3])
>>> print('a:',a)
a: [1 2 3]
>>> a[0] = 0.5 # a[0] will be truncated
>>> print('a:',a)
a: [0 2 3]
>>> a = np.array([1.0,2.0,3.0])
>>> print('a:',a)
a: [1. 2. 3.]
>>> a[0] = 0.5 # a[0] will not be truncated
>>> print('a:',a)
a: [0.5 2.  3. ]
>>> a = np.array([1,2,3], dtype = float)
>>> print('a:',a)
a: [1. 2. 3.]
>>> a[0] = 0.5
>>> print('a:',a)
a: [0.5 2.  3. ]
```

1.4 Python objects: naming and copying

Python represents all data—such as numbers, strings and lists—as *objects.*[5] Every object has an *identity,* a *type* and a *value.* The *identity* never changes once an object has been created. You can think of the *identity* as an address in computer memory where the description of the type and value starts. The function `id()` returns the address that defines the *identity.*

A single object can have several *names,* which can be thought of as what in other languages are called aliases or pointers. As illustrated in Table 11, the assignment `b = a` creates a new name for the object `a`. If an object has two names, the same change will take place independent of which name you use, because there is in fact only one single object. You can verify this by looking at the address of `a` and `b`. Indeed, `id(a)` and `id(b)` are the same. Also the `is` comparison operator allows you to verify whether or not two names point to the same object. If so, `a is b` returns the value `True`.

Objects can be mutable or immutable. Simple objects, such as numbers and strings, are immutable. For instance, as Table 10 shows, adding one to a number creates a *new* object without changing the name.

Table 10: Performing an arithmetic operation on an integer, an immutable object, creates a new object with the old name and a new value. Parts of the value of a mutable object can be changed without creating a new object, as the addresses returned by `id()` show.

```
>>> x = 1
>>> id(x)
4433230128
>>> x += 1
>>> x
2
>>> id(x)
4433230160
>>> a = [7, 8, 9]
>>> id(a)
4436491392
>>> a[2] += 1
>>> a
[7, 8, 10]
>>> id(a) # returns the same value as before the change to a
4436491392
```

The value of mutable objects, as the name indicates, can be changed without changing their identity. Obviously, if changing a single element of a large array would mean that the whole array had to be copied, this would be very time consuming. For simple objects, such as numbers, this is not a problem. There might be a problem if you create a single string that captures the contents of the World Wide Web or an integer with 10 million digits. Table 10, and in more detail Table 11, illustrate this difference in behavior of mutable and immutable objects.

The assignment `a = b` creates a new name for the same object. To create a new object rather than a just a new name pointing to the same old object, use `c = a.copy()`. Now, `id(a)` and `id(c)` are *not* the same, `a is c` returns the value `False`. Once again, see Table 11.

If `x` is the name of an integer, floating point number, or a string object `y = x` creates a new name for the same. Suppose that `a` is a NumPy array. You want to make change to `a` but also want to keep a copy of `a` in its original form. `b = a` will not do that; all it does is is that it creates a new name for the same object. As a consequence, any change made to `a` will automatically be made also to `b`. To create an independent copy of a use `b = a.copy()`. This is illustrated in the example in Table 11, but it works only for objects that have a `copy` attribute, that is for mutable objects. Check that `a.fill(0)` changes the elements of `a` but not the location of the object. On the other hand, `a = np.zeros([2,3])` creates a new object in a different memory location.

---

[5]python.org. *3. Data model: 1. Objects, values, types.* URL: https://docs.python.org/3.9/reference/datamodel.html.

Table 11: Illustrating the difference between `a = b` and `a = b.copy()`

```
>>> import numpy as np
>>> a = np.array([[1,2,3], [4,5,6]])
>>> print('a:\n',a)
a:
[[1 2 3], [4 5 6]]—
>>> b = a # changing a will also affect b and v.v.
>>> a is b # because a and b are identical
True
>>> print('id(a) = ', id(a),  'id(b) = ', id(b))
id(a) =  4530030608 id(b) =  4530030608 # and located at the same place in memory
>>> a.fill(0)
>>> print('a:\n',a)
a:
 [[0 0 0], [0 0 0]]
>>> print('b:\n',b)
b:
 [[0 0 0],[0 0 0]]
>>> a = [[1,2,3], [4,5,6]]
>>> print('a:\n',a)
a:
 [[1 2 3], [4 5 6]]
>>> b = a.copy() # changing a will not affect b and v.v.
>>> a is b # because a and b are not identical
False
>>> a.fill(0)
>>> print('a:\n',a)
a:
 [[0 0 0], [0 0 0]]
>>> print('b:\n',b)
b:
 [[1 2 3], [4 5 6]]
```

1.4.1   Scoping rules: local, nonlocal and global variables

The simplest way to illustrate the meaning of *scope* is by some examples.[6]

Example 1:

```
>>> def f(n):
...    n += 1
...
>>> n   = 10
>>> f(n)
>>> print('n = ', n)
n =  10
```

In example 1 the function `f` is defined, a value of 10 is assigned to `n`, and then `f` is called with argument `n`. Function `f` increases `n` by 1. The new value of `n` is stored in a new object, but its existence is local to `f` and remains unknown to the global environment (aka namespace) that calls `f`. As the print statement after the call to `f` shows, the global value of `n` remains unchanged. The object created in the environment of `f`—its local namespace—is abandoned when control returns from `f`. A `return n` at the end of `f` will change the value of `n` in the global namespace, but only if the return value is explicitly assigned to the global object `n`.

---

[6]python.org. *9.2 Python Scopes and Namespaces*. URL: https://docs.python.org/3.9/tutorial/classes.html#a-word-about-names-and-objects.

Example 2:
Program:

```
#!/usr/bin/env python3
# coding: utf-8
def f():
    # here n is treated as a local variable which cannot be
    # printed because no assignment has been made at this point
    print('f: n =',n)
    n += 1
    return n
def f1():
    # here n is treated as a global variable it can be printed
    print('f1: n =',n)
n = 10
f1()
print('survived f1 call and printed n')
f()
```

Output:

```
f1: n = 10
survived f1 call and printed n
Traceback (most recent call last):
  File "/Users/nigh/Desktop/ex2.py", line 15, in <module>
    f()
  File "/Users/nigh/Desktop/ex2.py", line 6, in f
    print('f: n =',n)
UnboundLocalError: local variable 'n' referenced before assignment
```

In example 2, a script,[7] an assignment is made in the `f` namespace, to `n`, where that name is treated as local. As a consequence, its value cannot be printed before the assignment is made. Without that `print(n)` will generate an error message. However, if no assignment is made, as is the case for `f1`, the instruction `print(n)` will work as expected, because `n` is treated as global and defined. This quirk can easily turn a functioning routine into one that unexpectedly fails with the error message shown in the example.

Example 3:

```
>>> def g():
...    global n
...    n += 1
...
>>> n   = 10
>>> g()
>>> print('n = ', n)
n =  11
```

Example 3 shows how the statement "`global n`" in the namespace of the function `g` makes sure that the assignment of a new value to `n` in that local environment makes `n` part of the global namespace so that assignments are visible both locally and globally.

---

[7]See footnote 8.

Example 4:

```python
#!/usr/bin/env python3
# coding: utf-8

def f1():
    print('f1: treats n as a global value: n', n)
    m = 99
    print('f1: initial value of m:', m)

    def f2():
        nonlocal m
        m += 1
    f2()
    print('f1: knows that f2 has assigned a new value to m:', m)
def f0():
    print('f0: before calling f1')
    f1()

n = 10
f0()
print(m)
```

Example 4 displays a script, as the "`#!/usr/bin/env python3`" statement shows.[8] The script can be executed from the command line. The new feature in this example is the use of the "`nonlocal m`" declaration. It does the same as the `global` declaration but is more limited in scope. It makes `m` in function `f2` visible by the function `f1`, but not globally, as the error message at the end of the output shows:

```
f0: before calling f1
f1: treats n as a global value: n 10
f1: initial value of m: 99
f1: knows that f2 has assigned a new value to m: 100
Traceback (most recent call last):
  File "/tmp/scope.py", line 20, in <module>
    print(m)
    NameError: name 'm' is not defined
Traceback (most recent call last):
  File "/tmp/scope.py", line 19, in <module>
    print(m)
NameError: name 'm' is not defined
```

---

[8]A script is a text file that is made executable. In Unix-like operating systems the commands in a script are executed by typing the name of the script on the command line followed by a `<return>`. The line starting with "#!"—aka shebang—tells the operating system which program should run the commands in the text file, python3 in the case of example 4.

Example 5:

```python
#!/usr/bin/env python3
# coding: utf-8

import numpy as np
u = np.array([[1,2],[3,4],[5,6]])
def f(a):
    t = a[0]
    a[0] = a[1] # this overwrites t
    a[1] = t
print('original u:\n', u)
f(u)
print('elements of u overwritten not swapped:\n',u)

v = np.array([[1,2],[3,4],[5,6]])
def g(a):
    t = a[0].copy() # t is a new object independent of a
    a[0] = a[1] # this doesn't overwrite t
    a[1] = t
g(v)
print('elements of v swapped:\n',v)

w = [1,2,3] # w is not a numpy array;
            # : no copy() necessary or even possible in this case
def h(a):
    t = a[0] # t is an immutable integer and lacking attribute copy()
    a[0] = a[1]
    a[1] = t
h(w)
print('elements of w swapped:\n', w)
```

Example 5 shows that objects, NumPy arrays in particular, can have sub-objects that can be mutable or immutable. The latter have no attribute 'copy()'; the former do. Unfortunately, in cases like this, the `id()` function and the `is` comparison operator do not seem to produce useful information. **Keep in mind that the difference in behavior of these two types of objects can be surprising and the difference between correct and incorrect code.**

This is what the output of example 5 looks like:

```
original u:
 [[1 2]
 [3 4]
 [5 6]]
elements of u overwritten not swapped:
 [[3 4]
 [3 4]
 [5 6]]
elements of v swapped:
 [[3 4]
 [1 2]
 [5 6]]
elements of w swapped:
 [2, 1, 3]
```

1.5   Producing readable output

Python and NumPy, in particular, provide many ways to produce readable output, an important but often neglected skill. The technical details are powerful but obscure. Table 12 shows some examples. The example in Table 12 uses `numpy.printoptions` for a local change to override the default. The default printoptions can

be retrieved with `np.get_printoptions`. A global change can be made with `np.set_printoptions`. Use `help(np.printoptions)` . These options apply to NumPy arrays. For more general formatting examples see Tables 3 and 13

Table 12: Formatting *array* printing

```
>>>import numpy as np
>>> b = np.array([np.pi**-2, np.pi**-1, np.pi, np.pi**2, np.pi**3])
>>> with np.printoptions(formatter={'float': '{: 0.3f}'.format}):
...    print(b)
...
[ 0.101  0.318  3.142  9.870  31.006]
>>> with np.printoptions(formatter={'float': '{: 0.3e}'.format}):
...    print(b)
...
[ 1.013e-01  3.183e-01  3.142e+00  9.870e+00  3.101e+01]
>>> b = np.array([np.pi**-2, np.pi**-1, np.pi, np.pi**2, np.pi**3, 7/3-4/3-1])
>>> with np.printoptions(precision=5, suppress=True):
...    print(b)
...
[ 0.10132  0.31831  3.14159  9.8696  31.00628  0.     ]
>>> with np.printoptions(precision=5, suppress=False):
...    print(b)
...
[1.01321e-01 3.18310e-01 3.14159e+00 9.86960e+00 3.10063e+01 2.22045e-16]
>>> np.set_printoptions(precision = 2)
>>> r = np.random.rand(3,3) # initialize 3 x 3 matrix with random numbers
>>> print(r) # use the globally defined print options
[[0.56 0.51 0.55]
 [0.38 0.31 0.41]
 [0.53 0.16 0.19]]
```

1.5.1 Pandas data analysis

Pandas is a data analysis module that has nice features for formatting and other data analysis utilities useful in particular for non-numerical data as encountered in finance, economics, statistics etc.[9]

1.6 Lists and arrays: examples and timing

There are different ways to initialize lists and NumPy arrays. Table 13 is an example of timing various ways of creating and working with NumPy arrays and with standard lists.

Take for example the simple task of creating two lists containing the same large number of integers or real numbers. Suppose we have a routine `add_list` that does this and adds the numbers. The routine `add_array` does the same with NumPy arrays. In the latter case, if `a` and `b` are two NumPy arrays `c = a + b` will create a numpy array containing the elementwise sum of `a` and `b`. For lists the addition requires a user supplied loop, which tends to be much slower.

There are also different ways of performing the kind of timing required to measure the difference in speed. There is standard Python `timeit.timeit` demonstrated in Table 13. The other method, shown in Table 14, uses IPython built in magic commands, as identified by the leading %-sign, that may not be available in all implementations.[10]

---

[9]*Pandas: getting started.* pandas. URL: https://pandas.pydata.org/pandas-docs/stable/getting_started/tutorials.html.

[10]The IPython Development Team. *Built-in magic commands.* URL: https://ipython.readthedocs.io/en/stable/interactive/magics.html.

Table 13: Using the standard Python timing routine; `add_list` is the name of some previously defined routine that adds the elements of two lists containing numbers. `add_array` is the same using NumPy arrays.

```
import timeit
loops = 10**4
t = timeit.timeit(add_list, number = loops)/loops
print('add_list  %.2e sec.' % t)
t = timeit.timeit(add_array, number = loops)/loops
print('add_array %.2e sec.' % t)
```

Table 14: Using %timeit *magic* for timing—this works in Jupyter Notebook but not in standard command line Python script

```
import numpy as np
n = 10000
%timeit a=np.empty(n); a.fill(5)
%timeit a=np.empty(n); a[:]=5
%timeit a=np.ones(n)*5
%timeit a=np.repeat(5,(n))
%timeit a=np.tile(5,[n])
```

1.6.1 Assignments

The following is a do-it-yourself assignment for in class or homework.

Assignment 1 list vs. array: timing

N vectors $\mathbf{x} = (x_1, x_2, \ldots, x_N)$ and $\mathbf{y} = (y_1, y_2, \ldots, y_N)$ can represented by NumPy arrays as follows:

```
import numpy as np
N = <some big number>
x = np.linspace(0, 10, N, endpoint=True)
y = np.linspace(10, 0, N, endpoint=True)
```

1. Use `np.inner(x,y)` to compute the inner product $\sum_i x_i y_i$ and time how long the operation takes.
2. Use `u = x.tolist()` and `v = y.tolist()` to convert the NumPy arrays to lists. Compute the inner product by writing your own loop to sum the product of the components. Time this and compare with item 1.

The NumPy on-line documentation has a detailed description of `linspace`.[11] You can also execute the command `help(np.linespace)`, at least if you have already executed `import numpy as np`.

The Eratosthenes sieve is a classic programming exercise. The sieve selects all primes up to a certain number from among the natural numbers, as described in Algorithm 1.

The following exercise constructs an Eratosthenes sieve. NumPy filters are a convenient for the selecting a subset of array elements of a given array. For an example that shows the use of the filter follow this link.. Table 15 shows another example of the use of a filter.

To initialize a NumPy array so that it can be used as a filter, you have to initialize it specifying the data type boolean so that the elements are interpreted as `True` or `False`.

Rather than using `np.append`, you can also use `np.ones`, as in Table 15, which relies on the fact that boolean 1 stands for `True`.

---

[11] NumPy.org. URL: https://numpy.org/doc/stable/reference/generated/numpy.linspace.html.

Algorithm 1 Eratosthenes sieve

Start with natural numbers up to some given maximum N

1. Keep 1;

2. Keep 2, but remove all other integral multiples of 2;

3. Keep 3, but remove all other integral multiples of 3;

4. 4 has already been removed; keep 5 and remove all other integral multiples of 5;

5. Keep doing the same for the next integers $n, n+1, \ldots$ until $n > \sqrt{N}$.

Table 15: Illustrating the use of a filter to remove odd numbers in a range

---

—

```
>>> import numpy as np
>>> n= 10
>>> numbers = np.array(range(1,n+1))
>>> even = np.ones(10, dtype = bool) # 1 stands for True
>>> for i, n in enumerate(numbers): # remove odd number slots from filter
...     if n%2 != 0: # % is the modulo operator
...         even[i] = False
...
>>> even_numbers = numbers[even]
>>> for i in even_numbers:
...     print(i)
...
2
4
6
8
10
```

Assignment 2 Eratosthenes sieve

Write code using a NumPy filter to implement the Eratosthenes sieve. You do this by creating an array or a list in which all elements are Boolean variables assuming the value `True`. Then you turn the values corresponding to natural numbers that are not prime into `False`. Finally, you apply the filter to the array containing all integers to select the primes and print them.

1.7 Plotting in Python

Matplotlib contains most if not all of what of what we'll ever be looking for.[12] Be a parasitic user; modify existing examples and never read any documentation unless all else fails.

In that spirit, here are more examples:

1. Graph Plotting in Python — Set 1

2. Graph Plotting in Python — Set 2

3. Graph Plotting in Python — Set 3

4. Plotting with SymPy

These notes contain the code used for some of the figures; check section 12.

---

[12] *Matplotlib: Visualization with Python—Gallery*. URL: https://matplotlib.org/stable/gallery/index.html and J. D. Hunter. "Matplotlib: A 2D graphics environment". In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55

1.8 Complex numbers

Most complex number functions are present in Python's NumPy. Almost all you need to know is that $i = \sqrt{-1}$ is represented by `1j`. Some simple examples are in Table 16. Note that if `numpy.zeros` declares an NumPy array, `dtype` declares its type, `complex` in this case.

Table 16: Complex numbers: examples of elementary use

```
>>> import numpy as np
>>> i = 1j
>>> print(i**2)
(-1+0j)
>>> print(np.cos(i), np.cosh(1.0))
(1.5430806348152437-0j) 1.5430806348152437
>>> print(np.sin(i)/i, np.sinh(1.0))
(1.1752011936438014+0j) 1.1752011936438014
>>> z = 1+1j
>>> print(type(z))
<class 'complex'>
>>> print(np.real(z))
1.0
>>> print(np.imag(z))
1.0
>>> print(np.abs(z))
1.4142135623730951
>>> import cmath as cm
>>> print(cm.phase(z))
0.7853981633974483
>>> print(cm.polar(z))
(1.4142135623730951, 0.7853981633974483)
>>> c = np.zeros([3,3], dtype = complex)
>>> print('c\n', c)
c
 [[0.+0.j 0.+0.j 0.+0.j]
  [0.+0.j 0.+0.j 0.+0.j]
  [0.+0.j 0.+0.j 0.+0.j]]
```

1.9 Formula manipulation: SageMath and SymPy

1. SageMath is a free, open-source symbolic computer mathematics software system.[13] It is licensed under the GNU General Public License (GNU GPL or GPL);

2. SymPy is a Python package for symbolic computer mathematics written in Python.[14]

Both SageMath and SymPy are written in Python. They use the same syntax. SymPy can mixed with other Python packages.

1.9.1 Simple SymPy examples

Table 17 shows simple examples of integration, differentiation, and series expansion.

---

[13]SageMath.org. URL: https://www.sagemath.org.
[14]SymPy.org. URL: https://www.sympy.org/en/index.html.

Table 17: Simple illustration of SymPy

```
>>> from sympy import *
>>> x = Symbol('x')
>>> f = Function('f')
>>> f = integrate(x**2 + x +1,x)
>>> f
x**3/3 + x**2/2 + x
>>> diff(f,x)
x**2 + x + 1
>>> series(sin(x),x,0,10)
x - x**3/6 + x**5/120 - x**7/5040 + x**9/362880 + O(x**10)
```

2 Basic numerical facts and methods

2.1 Integers and floating point numbers

A floating point number r is represented by a sign bit s, a mantissa m, and an exponent x in the form

$$r = sm2^x, \tag{2.1}$$

which is the digital analog of the scientific notation as in 1.0100×10^3. Of course, 0.1010×10^4 represents the same number but given the constraint of five digits it's a less accurate representation because the leading zero takes up space while it contains no useful information. The form in which m contains no leading zeros is called normalized.[15] For more details see the IEEE 754 standard or follow the link in the footnote.[16]

Because floating point numbers lack infinite precision, computations suffer from truncation errors. These can be substantial, but can often be avoided. The standard, high school solution of the quadratic equation $ax^2 + bx + c = 0$ must be re-written to avoid this problem.[17]

Assignment 3 Roots of a quadratic equation
Make a table to compare the results obtained from *Numerical Recipes* Eqs. (5.6.2), (5.6.3), and (5.6.5)[18] for values of a and c that decrease by successive factors of $\sqrt{10}$. Arrange the table so that it clearly demonstrates that Eqs. (5.6.2) and (5.6.3) both get one root "right" and the other "wrong."

Python has lots of data types,[19] but most data types are determined automatically. That can be convenient, but it can also produce wrong results, if you don't know what you are doing. Try some of the examples in Table 18. Also recall Table 9.

---

[15]W. H. Press et al. *Numerical Recipes: 1.2 Error, Accuracy, and Stability*. URL: http://phys.uri.edu/~nigh/NumRec/bookfpdf/f1-2.pdf#page=2. There is a new, third edition of *Numerical Recipes Numerical Recipes: The Art of Scientific Computing. Third Edition*. URL: http://www.numerical.recipes/

[16]Wikipedia. *Exponent bias*. URL: https://en.wikipedia.org/wiki/Exponent_bias.

[17]W. H. Press et al. *Numerical Recipes: 5.6 Quadratic and Cubic Equations*. URL: http://phys.uri.edu/~nigh/NumRec/bookfpdf/f5-6.pdf.

[18]Ibid.

[19]NumPy.org. *data types*. URL: https://numpy.org/doc/stable/user/basics.types.html.

Table 18: Real and integer types

```
>>> a = 2
>>> b = 3
>>> c = 4.0
>>> print(type(a))
<class 'int'>
>>> print(type(b))
<class 'int'>
>>> print(type(c))
<class 'float'>
>>> print(a/b)
0.6666666666666666
>>> print(np.int64(a/b))
0
>>> print(round(a/b,0))
1.0
>>> print(round(a/b))
1
```

The n-digit base-b representation of an integer N is a string $d_{n-1}d_{n-2}\ldots d_0$ in which the d_i integers with $0 \le d_i \le b-1$ such that

$$N = \sum_{k=0}^{n-1} d_k b^k \tag{2.2}$$

NumPy integers are faster than Python integers, which can automatically expand the number of binary digits they use, but speed comes at a price: overflow, as illustrated in Table 19. Keep the following in mind:[20]

> The behavior of NumPy and Python integer types differs significantly for integer overflows and may confuse users expecting NumPy integers to behave similar to Python's int. Unlike NumPy, the size of Python's int is flexible. This means Python integers may expand to accommodate any integer and will not overflow.

Table 19: Overflowing NumPy integers

```
>>> import numpy as np
>>> np.power(100, 8, dtype=np.int64)
10000000000000000
>>> np.power(100, 8, dtype=np.int32)
1874919424
```

2.1.1 Assignments

Assignment 4 Integer overflow

1. Write a loop that uses Python integers to generate 2^k and $2^k - 1$ for $k = 1, \ldots, 100$. Print the results in the usual base-10 notation, in binary (base-2), octal (base-8) and in hexadecimal (base-16) form. Use the `bin(i)`, `oct(i)`, `hex(i)` for this purpose. The latter uses A, B, ..., F to represent the "digits" 10 through 15.

2. Do the same using NumPy.int64 arithmetic; `np.int64(2)**k` for $k = 1, \ldots, 100$ will do the trick. Beware of unexpected signs.

3. What happens when you add 1 to the largest positive `np.int64` number?

4. What happens when you subtract 1 from the largest (in absolute value) *negative* `np.int64` number?

---

[20] NumPy.org, *data types*.

2.2 Arbitrary precision computation with mpmath

The mpmath package provides arbitrary precision floating point numbers.[21,22] The following is a link to the basics of mpmath. Some additional examples are in Table 20.

Table 20: Decimal precision (mp.pds) always is roughly $\frac{1}{3}$ of binary working precision (mp.prec)

```
>>> import mpmath as mp
>>> print('decimal working precision: ', mp.mp.dps)
decimal working precision:  15
>>> print('binary working precision:  ', mp.mp.prec)
binary working precision:   53
>>> print('1/9 as a 15 dgt fl pnt numb: ', mp.mpf(1)/mp.mpf(9))
1/9 as a 15 dgt fl pnt nmb:  0.111111111111111
>>> mp.mp.dps = 50
>>> print('1/9 as a 50 dgt fl pnt numb: ', mp.mpf(1)/mp.mpf(9))
1/9 as a 50 dgt fl pnt nmb:   0.11111111111111111111111111111111111111111111111111
>>> print('pi as a 50 dgt fl pnt nmb:  ', mp.pi)
pi as a 50 dgt fl pnt nmb:   3.1415926535897932384626433832795028841971693993751
```

2.2.1 Assignments

Assignment 5 floating point precision

1. Suppose that $x \approx 1.000$ and $y \approx 0.0001$, then $x + y \approx 1.000$ within the precision implied by the number of decimals provided. Similarly, find the smallest negative power of two that can be added to the number 1 using several choices of mpmath arbitrary floating point precision so that the result is different than 1.

2. Compare the result with $\varepsilon_{\text{prec}} = 7/3 - 4/3 - 1$ evaluated to the chosen precision. Explain why $7/3 - 4/3 - 1$ works, but $7/4 - 3/4 - 1$ does not.

3. Show that addition of floating point numbers is not associative

$$(1 + \frac{1}{2}\varepsilon_{\text{prec}}) + \frac{1}{2}\varepsilon_{\text{prec}} \neq 1 + (\frac{1}{2}\varepsilon_{\text{prec}} + \frac{1}{2}\varepsilon_{\text{prec}}) \tag{2.3}$$

2.3 Recursion: blessings and curses

A recursion relation of the form

$$x_{n+1} = g(x_n), \tag{2.4}$$

where g is a well-behaved function, may have a fixed point

$$x^* = g(x^*), \tag{2.5}$$

as is the case for Eq. (2.10), but there is no guarantee that the fixed point is stable, that is that small perturbations will not cause run-away behavior. In the vicinity of x^* we can expand g and write

$$g(x^* + \Delta x) = x^* + \alpha\Delta x + o(\Delta x), \tag{2.6}$$

where $o(\Delta x)$ stands for terms that go to zero faster than Δx as $\Delta x \to 0$. [23] If $|\alpha| < 1$, the fixed point is stable. If $|\alpha| > 1$, it is unstable. If $|\alpha| = 1$ it can be either one of those.

The equation

$$x^2 = a, \text{ with } a > 0 \tag{2.7}$$

---

[21] F. Johansson et al. *mpmath: a Python library for arbitrary-precision floating-point arithmetic (version 1.1.0)*. Dec. 2018. URL: http://mpmath.org/.

[22] F. Johansson et al. *mpmath: a Python library for arbitrary-precision floating-point arithmetic (version 1.1.0)*. Dec. 2018. URL: http://mpmath.org/doc/current.

[23] This an example of the Backmann-Landau "big O" ($\mathcal{O}$) and "small o" iasymptotic notation.

can be written as

$$f(x, a) = x, \tag{2.8}$$

where

$$f(x, a) = \frac{1}{2}\left(x + \frac{a}{x}\right). \tag{2.9}$$

Let

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{c}{x_n}\right) \tag{2.10}$$

As $n \to \infty$ the x_n approach a fixed point $x_\infty = \sqrt{a}$. Close to this point the function $f(x)$ has the following Taylor series expansion

$$f(x, a) = \sqrt{a} + \frac{1}{2\sqrt{a}}(x - \sqrt{a})^2 + \mathcal{O}\left((x - \sqrt{a})^3\right). \tag{2.11}$$

Because the term linear in $x - \sqrt{a}$ vanishes, the approach to the fixed-point of the sequence defined by Eq. (2.10) is quadratic in the sense that the number of correct digits asymptotically doubles every iteration. In the typical case, in the presence of a linear term the convergence is linear, that is exponential in the number of iterations. There only is convergence for amplitudes less than unity in absolute value, otherwise the recursion method produces a divergent sequence.

2.3.1   Assignments

Assignment 6 Recursive stability and instability

1. Consider the (attempted) iterative solution of the equation $x = g(x, a)$ for several choices of a and definitions of g as specified in item # 2 below. To visualize progress, or lack thereof, plot the pairs (x_n, x_{n+1}) with $x_{n+1} = g(x_n, a)$—as for example in Eq. (2.10)— for $n = 0, 1, \dots, N - 1$ for suitable choices of x_0 and N.

2. Use three different definitions of g:

   (a) $g(x, a) = 1 + a(x - 1)$ with $|a| > 1$;
   (b) $g(x, a) = 1 + a(x - 1)$ with $|a| < 1$; and
   (c) $g(x, a) = (x + a/x)/2$ with $a > 0$.

   Hints:

   1. Table 22 shows how Python can be used to append successive points to a NumPy array called `points`;
   2. Table 23 contains a simple plot example; and
   3. So as to be able to use one and the same method of generating the list of points, it helps to define `sequence` with the declaration `def sequence(g, a, x0, n)`, where
      (a) `g` is an argument that determines which function is to be used;
      (b) `a` is a parameter passed to `g`;
      (c) `x0` is the initial value;
      (d) `n` is the number of points to be generated; and
      (e) `g` in its various forms can be defined as

      ```
      def g(x, a):
          ...
          return <value of g(x, a)>
      ```
      or alternatively as lambda functions:
      i. `line = lambda x, a: 1 + a * (x-1)`
      ii. `sq = lambda x, a: 0.5 * (x+a/x)`

The lambda function used above has the following structure:

```
<name> = lambda <arguments separated by commas>: <function of the arguments>
```

The angular brackets and what is inside them should be replaced by what is described inside the brackets. The rest, the word "lambda" and the colon, are required by the syntax.

There also is a variant called an anonymous lambda function. The function in this case has no name and is defined for single use. The example in Table 21 uses the Python `map()` function which has two arguments; the first one is a function and the second one a list. The example demonstrates the use of an anonymous lambda function, which creates a new list obtained by squaring all elements of the input list.

Table 21: Example of an anonymous lambda function

```
>>> list_1 = [1,2,3,4,5]
>>> list_2 = list(map(lambda x: x**2, list_1))
>>> print(list_2)
[1, 4, 9, 16, 25]
```

Table 22: Create an empty NumPy array and append an element

```
>>> import numpy as np
>>> a = np.array([])
>>> a = np.append(a, 1)
>>> print(a)
[1.] # note that floating point numbers are used as the default
```

Figure 1: Simple plot; the code is in Table 23.

$$y = ax^3$$
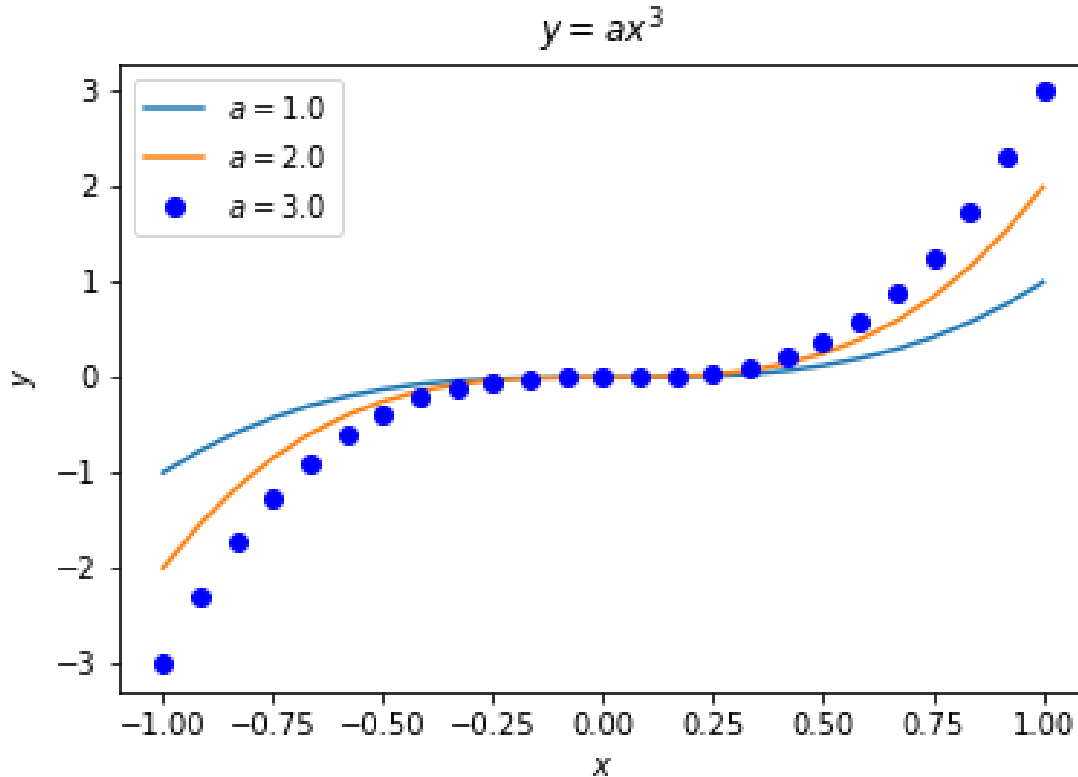
Table 23: Simple plot instructions produce Fig. 1. Note that the matplotlib understands LaTeX, the text between dollar signs.

```
import numpy as np
import matplotlib.pyplot as plt
f = lambda x, a: a * x**3
x = np.linspace(-1, 1, 25)
plt.plot(x, f(x, 1.0), label = '$a=$' + str(1.0))
plt.plot(x, f(x, 2.0), label = '$a=$' + str(2.0))
plt.plot(x, f(x,3.0), 'bo',label = '$a=$' + str(3.0) )
plt.title('$y = a x^3$')
plt.xlabel('$x$')
plt.ylabel('$y$')
plt.legend(loc='upper left')
plt.savefig('simple_plot.png')
# avoid losing edges with plt.savefig('simple_plot.png',bbox_inches='tight')
```

2.3.2 Numerical instability

Define

$$y_n = \int_0^1 \frac{x^n}{x+a}\, dx. \tag{2.12}$$

y_n satisfies the recursion relation

$$y_n + ay_{n-1} = \frac{1}{n}, \tag{2.13}$$

which immediately follows from

$$y_n + ay_{n-1} = \int_0^1 \frac{x^{n-1}(x+a)}{x+a}\, dx = \frac{1}{n}. \tag{2.14}$$

If we replace the right-hand side of this equation by zero, we get the homogeneous equation

$$y_n + ay_{n-1} = 0, \tag{2.15}$$

the solution of which is

$$y_n \propto \alpha(-a)^n. \tag{2.16}$$

The recursion relation in Eq. (2.14) allows us to compute y_n in terms of $y_0 = \log(1+1/a)$. Numbers generated by the recursion relation are susceptible to minor perturbations of the y_n. Rounding floating point numbers is such a perturbation. The effect grows exponentially whenever $|a| > 1$. As a consequence, computing y_n by means Eq. (2.14) is numerically unstable for $|a| > 1$.

It turns out that the integral in Eq. (2.12) can be expressed in term of the hypergeometric function $_2\mathrm{F}_1$[24]

$$y_n = \frac{_2F_1(1, 1+n; 2+n; -a^{-1})}{a(1+n)} \tag{2.17}$$

In the following exercise use the following:[25]

```
from mpmath import mp
from scipy.special import hyp2f1
...
mp.dps = 10
...
mp.dps = 40
```

---

[24]Wikipedia. *Hypergeometric function*. URL: https://en.wikipedia.org/wiki/Hypergeometric_function.
[25]SciPy.org. *Special Functions*. URL: https://docs.scipy.org/doc/scipy/reference/special.html.

2.3.3 Assignments

Assignment 7 Arbitrary floating point precision

1. Use the recursion relation given in Eq. (2.14) to compute y_n for $n = 0, \ldots, 25$. Do so using 10, 15, 20, and 25 decimal precision and create a table presenting the results in 5 decimal accuracy. Use something like this:

   ```
   print('i = %4i ' % i, '; x = %7.3f' % x, '; y = %7.2e' % y)
   ```

   where `i` is an integer; `x` and `y` are reals. For arrays you can use for instance

   ```
   with np.printoptions(precision=5, suppress=True):
       print(<whatever>)
   ```

   for this purpose. Also see Table 12.[26],[27],[28]

2. Run the recursion relation of Eq. (2.14) backward, expressing y_{n-1} in terms of y_n starting at some sufficiently large value of n_0. You can pick any reasonable value for y_{n_0}. The backward recursion will quickly converge to the correct value of y_n for $n_0 \gg n$.

2.4 Random number generation

Most of the time people use the term *random numbers*, but pseudo-random numbers is a more accurate description, because on computers "random" numbers are generated deterministically. They have a lot in common with real random numbers, or so one hopes. The "pseudo" is usually understood and omitted. *Numerical Recipes* provides the basic ideas, but the details are seriously outdated.[29] Python uses the Mersenne Twister which generates 53-bit precision floating point numbers sampled uniformly from the semi-open interval $[0, 1)$.[30]

Sometimes, for reproducibility it necessary to initialize a sequence of random numbers at the same place. `random.seed(<seed>)` (with `<seed>` any integer of your choice) serves this purpose. The basic random number generator produces a $U(0, 1)$ random number, selected from the interval $(0, 1)$ with uniform probability density. If no seed is supplied, the system time is used to initialize the random number generator.

2.5 The Fibonacci sequence

The Fibonacci sequence satisfies:

$$x_n = \begin{cases} 1 & \text{if } n = 1 \text{ or } n = 2 \\ x_{n-1} + x_{n-2} & \text{if } n > 2 \end{cases} \tag{2.18}$$

The recursion relation, Eq. (2.18), can be written as

$$(\mathbf{F}^2 - \mathbf{F} - \mathbb{1})x_n = 0, \tag{2.19}$$

where $\mathbf{F}$ is the forward-shift operator $\mathbf{F}x_k = x_{k+1}$; $\mathbb{1}$ is the identity operator. Substitute $x_n = \alpha f^n$ into Eq. (2.19) and you see that f satisfies the characteristic equation

$$f^2 - f - 1 = 0. \tag{2.20}$$

The solution of Eq. (2.18) is

$$x_n = \alpha_1 f_1^n + \alpha_2 f_2^n, \tag{2.21}$$

where the f_i are the roots of Eq. (2.20) and the α_i are determined by the initial conditions $x_1 = x_2 = 1$.

Notice that the characteristic equation of a homogeneous ordinary, second-order differential equation plays the same role. In both cases, generalization to higher orders in the derivatives or the number of terms

---

[26] w3schools. *Python String Formatting: Multiple Values*. URL: https://www.w3schools.com/python/python_string_formatting.asp.

[27] For a more elaborate formatting example follow this link.

[28] Here is a general reference: PyFormat.info. *Python formatting with practical examples*. URL: https://pyformat.info/#simple

[29] W. H. Press et al. *Numerical Recipes: Chapter 7. Random Numbers*. URL: http://phys.uri.edu/~nigh/NumRec/bookfpdf/f7-0.pdf.

[30] P. Documentation. *Generate pseudo-random numbers*. URL: https://docs.python.org/3/library/random.html.

in the recursion relation is straightforward. Characteristic equations involving higher-order polynomials result in such cases.

In the $n \to \infty$ limit, the ratio of subsequent elements of the Fibonacci sequence satisfy the equation

$$r^{-1} - 1 - r = 0. \tag{2.22}$$

2.5.1 Assignments

Assignment 8 Fibonacci sequence

Write a Jupyter Notebook program that:

1. Asks the user how many elements of the sequence the program should print;

2. Prints the number of the element of the sequence, the element itself, and the ratio of the current and the previous element of the Fibonacci sequence. Use integer format for the first two and for the ratio use scientific notation, as, for example 1.001e3 which stands for $1.001 \times 10^3 = 1001$.[31],[32]

3. Use this Pyplot Tutorial to plot the subsequent ratios and the asymptote.

4. Find the α_i and f_i in Eq. (2.21) and write code that shows that the result agrees with the sequence constructed directly from the recursion relation.

2.6 List of packages

Table 24 show a list of package invocations. The list is based on the examples used in these notes. Be a minimalist: import only what you need. If you import lots of packages, you might be importing totally different routines with the same names. In that case, you program may not do what you expect and you'll end up wasting a lot of time debugging.

Table 24: Python packages encountered in this course

from matplotlib import pyplot as plt	plot functions and arrays
from math import erf	error function
from mpmath import mp	floating-point arithmetic with arbitrary precision
from numpy import random	import only NumPy's random number generator
from scipy.linalg import circulant, expm	specific linear algebra routines
from scipy.special import hyp2f1	hypergeometric function $_2F_1$
from scipy.stats import norm	normal distribution
from mpl_toolkits.mplot3d import Axes3D	for 3D plots
import sympy as smp	abbreviate sympy as sp
from sympy import *	import all SymPy routines[33]
from sympy import Symbol, Function, series, simplify, solve	SymPy routines imported individually
from sympy import Matrix, exp, cos, Rational, log	import named routines from SymPy
from sympy.abc import x, t, d	declare x, t and d as SymPy symbols
import cmath as cm	expanded methods for complex numbers
import matplotlib.animation as animation	create animated plot
import matplotlib.pyplot as plt	standard Python plotting utilities
import numpy as np	NumPy routines
import random as rnd	abbreviate random as rnd
import scipy.linalg as la	abbreviate linalg as la
import scipy.special as sc	abbreviate scipy.special as sc
import sys	system utilities to interact with the operating system
import timeit	measure execution time of code snippets

---

[31] Here is a fairly simple approach: w3schools, *Python String Formatting: Multiple Values*.
[32] For a more elaborate formatting example look at Table 12.
[33] This may be dangerous; it could cause clashes with other packages.

2.7 Beyond the basics

There's a lot out there, including a bag of neat little tricks for once you have mastered the basics like the `for ... else` and `break` features in Table 25.[34]

Table 25: For ... else construction

```
>>> for n in range(1, 11):
...     for x in range(2, n):
...         if n % x == 0: # % is the modulo operator
...             print( n, 'equals', x, '*', int(n/x))
...             break
...     else:
...         # loop ended without finding a factor
...         print(n, 'is a prime number')
...
1 is a prime number
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
10 equals 2 * 5
```

3 Elementary numerical methods

3.1 Finding roots: Newton-Raphson

The Newton-Raphson method is explained in *Numerical Recipes*: 9.4 Newton-Raphson Method Using Derivative. For the related secant and regula falsi methods follow this link.

As an example consider finding roots of a polynomial

$$p(x) = a_0 x^n + a_1 x^{(n-1)} + \cdots + a_n \tag{3.1}$$

Polynomials can be calculated efficiently using Horner's rule, that is, by rewriting

$$p(x) = (\cdots ((a_0 x + a_1)x + a_2)x + \cdots)x + a_n, \tag{3.2}$$

and successively computing b_i and finally $p(x)$ as follows

$$\left.\begin{array}{l} b_0 = a_0 \\ b_1 = a_1 + x b_0 \\ b_2 = a_2 + x b_1 \\ \vdots \\ b_n = a_n + x b_{n-1} \\ p(x) = b_n \end{array}\right\} \tag{3.3}$$

3.1.1 Assignments

Assignment 9 Polynomial root finding

Consider polynomial of the form of Eq. (3.1) with $(a_0, a_1, a_2, a_3, a_4, a_5) = (1, 2, 3, 4, 5, 6)$. The purpose of the exercise is to find the roots of this polynomial.

1. Write a routine to evaluate the polynomial and its derivative with Horner's rule.

---

[34]M. Y. U. Khalid. *Python Tips*. URL: https://book.pythontips.com/en/latest.

2. Write a routine that will do a Newton-Raphson step for an arbitrary function. The definition of the routine will look something like `def newt_raph(x, func, func1, a):`, where x is the current estimate of the root; `func` and `func1` evaluate the function and its derivative and a are parameters that define the function, in this case the coefficients of the polynomial.

3. Write a routine that keeps making Newton-Raphson steps until the relative change in the estimate of the root no longer changes within the accuracy of the computation. (Note: if zero were a root, the relative change would not be the right measure. What would you do in that case?)

   Hint: Take into account that a polynomial can have roots with non-vanishing imaginary parts. You'll never find those if you start the search on the real axis. (Why not?)

4. Use what is called *numerical deflation* to find the other roots of the polynomial. In general, if you know that $f(x) \equiv f_0(x)$ has a zero x_0, you can define $f_1(x) \equiv f(x)(x - x_0)^{-1}$ and look a zero of $f_1(x)$. This procedure can be repeated: $f_2(x) \equiv f(x)(x - x_0)^{-1}(x - x_1)^{-1}$ iterating the procedure. Ignore the fact that f_2 is once again a polynomial; pretend that f_2 is some arbitrary function instead. So as not to have to calculate the derivatives, use the secant method to find $x_1, x_2, \dots$.

Note: Instead of using the secant method, one could use the Newton-Raphson method approximating the derivative with a finite-difference expression—see section 3.5 for more.

3.2 Minimization: golden section search

A simple and fast way to find a minimum of a function is to look for a zero of its derivative. The Newton-Raphson method can be used for that. If it works, it is very fast, but it typically is stable only in the vicinity of the minimum. The method can used in dimensions greater than one.

In one dimension this is what Newton-Raphson looks like. Suppose that we have a near-quadratic function and a point x from which we want to find the location of the minimum $x + \Delta x$. We have

$$f(x + \Delta x) = f(x) + f'(x)\Delta x + \frac{1}{2}f''(x)\Delta x^2 + \mathcal{O}(\Delta x^3). \tag{3.4}$$

The minimum occurs at a value of Δ such that

$$\frac{df(x + \Delta x)}{d\Delta x} = 0. \tag{3.5}$$

Solving for Δx gives

$$\Delta x \approx -\frac{f'(x)}{f''(x)}. \tag{3.6}$$

The same reasoning applied in d dimensions gives

$$\Delta \boldsymbol{x} \approx -\boldsymbol{H}^{-1}\nabla f(\boldsymbol{x}), \tag{3.7}$$

where $\boldsymbol{x} = (x_1, x_2, \dots, x_d \nabla f(\boldsymbol{x})$ is the d-component gradient of f and $\boldsymbol{H}$ the $d \times d$ matrix of second-order derivatives, aka the Hessian matrix. Eqs. (3.6) and (3.7) ignore higher-order corrections, a problem that is dealt with as usual, namely iteratively.

More often than not the second-order derivative of f is not strictly positive everywhere nor is the matrix of second-order derivatives positive definite as is required for finding a minimum. This makes the method unstable if used in this simple form. In addition, in higher dimensions, the computation of the Hessian matrix is time consuming.[35]

In one dimension a stable alternative is the golden mean section search to find the minimum of a continuous function possibly lacking derivatives. The basic idea is simple. To bracket a zero of a continuous function one needs *two* points, one where the function is negative and one where it is positive. On the other hand, to bracket a minimum, one needs *three* points: with two high function values on the sides and a low one in the middle.

Let's assume that at some point the minimum is bracketed by the triplet $[A, B, C$ as illustrated in Fig. 2. In this figure the triplet $[A, D, C]$ might be possible alternative triplet of departure, but it can be obtained by inversion so that we can ignore this particular geometric arrangment without loss of generality and pretend

---

[35] For a detailed discussion of way of dealing with these problems see. W. H. Press et al. *Numerical Recipes: 10 Minimization or Maximization of Functions*. URL: http://phys.uri.edu/~nigh/NumRec/bookfpdf/f10-0.pdf

that point D is not there to begin with. Also without loss of generality, we can scale and translate the triplet bracket, which is why we have A at 0 and C at 1 in Fig. 2.

Now choose point D as the mirror image of B with respect to the middle of $[A, B]$.and determine the new bracketing interval: A, D, B or D, B, C so as to obtain the lowest function value. Depending on that we choose a new point E or F. Here is the crux: the points are chosen so that B and D assume the same *relative* positions in the interval $[A, D]$ as D and E in the interval $[A, B]$ or, as the case may be, points B and F in the interval $[D, C]$.

This implies that

$$1 - x = \frac{x}{1 - x}. \tag{3.8}$$

Note that the the $1 - x$ in the denominator is a rescaling factor required because the bracketing interval has shrunk by that factor. This equation has only one solution with $|x| < 1$:

$$x = \frac{1}{2}\left(3 - \sqrt{5}\right) = 0.38196\ldots \tag{3.9}$$

so that $x = 2 - \varphi$ with the golden ratio $\varphi = \frac{1}{2}(1 + \sqrt{5}) = 1.6180\ldots$ If follows that the next bracketing interval, consist of the points A, D, B or of the points D, B, C, will be narrower by a factor $\varphi - 1$.

Figure 2: Golden mean section: start from $[A, B, C]$. Construct D as the mirror image of B with respect to the middle of $[A, B]$. Choose $[A, D, B]$ or $[D, B, C]$ depending on which gives the lowest function value. In the next step contine in the same way with either E or F playing the role of D.
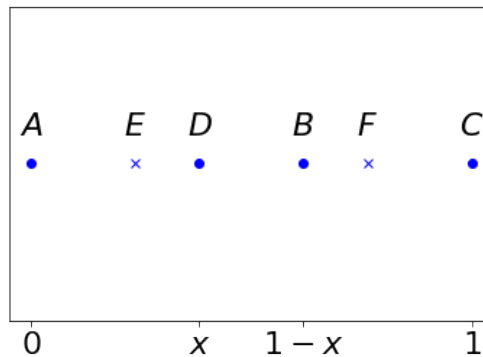
A question we did not address is how to get the process started See *Numerical Recipes* section 10.1 Golden Section Search in One Dimension for more about this. In the following simple exercise this won't be a problem.

Iterative methods require stopping criteria. Typically, in the vicinity of a minimum the function behaves quadratically. That is

$$f(x) \approx c + \frac{1}{2}a(x - b)^2. \tag{3.10}$$

If relative precision of the floating point arithmetic is ε it's impossible the determine x more accurately than

$$|x - b| \approx \sqrt{\frac{2\varepsilon|c|}{a}}. \tag{3.11}$$

It's not too difficult to come up with an order-of-magnitude estimate of the constants a and c on the right-hand side of this expression either before or during the iteration.

For the case $a = 0$ in the following example, Eq. (3.10) is not valid, but this is fairly atypical.

3.2.1 Assignments

Assignment 10 Golden section search

Define $f(x) = x^4 + \alpha x - 1$ use the golden section search to find the minima for $\alpha = 1$, $\alpha = 0$ and $\alpha = -1$. Choose random initial points consistent with the bracketing condition.

3.3 Steepest decent

The steepest decent method to find a minimum is based on the fact that the gradient of a function defines the direction of most rapid change of a differentiable, many-variable function $f(x_1, x_2, \ldots, x_d) \equiv f(\boldsymbol{x})$. Suppose that at some stage of the process, we have point $\boldsymbol{x}_k$. The next point is found from;

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \gamma \nabla f(\boldsymbol{x}_k). \tag{3.12}$$

Here $\gamma > 0$ is chosen so as to minimize $f(\boldsymbol{x}_{k+1})$ with respect to γ.

3.3.1 Assignments

Assignment 11 Steepest descent search
Consider the function defined by

$$y(t; t_1, t_2) = e^{-t/t_1} + e^{-t/t_2}, \tag{3.13}$$

$$f(t_1, t_2) = \sum_{t=0}^{4} [y(t; t_1, t_2) - y(t; 2, 3)]^2. \tag{3.14}$$

1. Make a contour plot of $f(t_1, t_2)$; for helpful sample code see Table 26. Make sure your picture shows the banana-shaped contours surrounding the minimum.

2. Use the steepest decent method to find the minimum of the $f(t_1, t_2)$ and plot the successive points generated by this algorithm. Choose two methods as described in the hints for this problem to solve the one-dimensional minimization along the gradient direction.

Hints:

1. The function to be minimized in this case is simple enough to calculate and program its gradient analytically;

2. The value of γ that minimizes the function along the direction of the gradient can be found using the golden section search discussed in section 3.2;

3. An alternative way to find the value of γ that minimizes the function along the direction of the gradient can be found by using a generalization of the Newton-Raphson method of section 3.1 on page 26. The generalization consists in applying the method to find the place where the derivative with respect to γ vanishes. The required first-order derivative in this case is the *directional derivative.* That is, suppose that g is a function of $\boldsymbol{x}$ and $\boldsymbol{n}$ is a unit vector. The directional derivative of g is

$$g'_{\boldsymbol{n}}(\boldsymbol{x}) = \lim_{h \to 0} \frac{g(\boldsymbol{x} + h\boldsymbol{n}) - g(\boldsymbol{x})}{h} = \boldsymbol{n} \cdot \boldsymbol{\nabla} g = \sum_i n_i \frac{\partial g}{\partial x_i}. \tag{3.15}$$

The second-order derivative is also required in this case. This becomes quite messy, but for two variables it is still doable. In this case the second-order derivative takes the form

$$
\begin{aligned}
g''_{\boldsymbol{n}}(x_1, x_2) &= n_2 \left(n_1 g^{(1,1)}(x_1, x_2) + n_2 g^{(0,2)}(x_1, x_2) \right) + \\
&+ n_1 \left(n_1 g^{(2,0)}(x_1, x_2) + n_2 g^{(1,1)}(x_1, x_2) \right)
\end{aligned}
\tag{3.16}
$$

There are also numerical ways of approximately obtaining the required derivatives;

4. A simple, but slow alternative for finding a good value of γ is to try a random number. All you have to do is make sure that you go downhill as a function of γ; and

5. Also `scipy.optimize.brent`[36] provides an excellent choice.

Variants of the steepest decents method are used for the optimization of neural networks. In such cases, most of the methods described in *Numerical Recipes* chapter 10 are impractical, because the number of variables is too large.[37]

---

[36] *Given a function of one variable, return a local minimum.* Scipy. URL: https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.brent.html.

[37] For more on the variants of the steepest decent method used for the optimization of neural networks see for example P. Mehta et al. "A high-bias, low-variance introduction to Machine Learning for physicists". In: *Physics Reports* 810 (2019). A high-bias, low-variance introduction to Machine Learning for physicists, pp. 1 –124. ISSN: 0370-1573. DOI: https://doi.org/10.1016/j.physrep.2019.03.001. URL: http://www.sciencedirect.com/science/article/pii/S0370157319300766.

Table 26: Code to generate a simple contour plot, as shown in Fig. 3 of the Newtonian gravitational potential of two equal masses; see Tables 43 and 44 in appendix 11.1 on p. 76 ff. for more about np.meshgrid

```
>>> import matplotlib
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>>
>>> def newton(x,y):
...     y12 = 0
...     x1 = -1
...     x2 = 1
...     return -1/np.sqrt(((x-x1)**2+y**2)) -1/np.sqrt(((x-x2)**2+y**2))
...
>>>
>>> delta = 0.025
>>> x = np.arange(-4.0, 4.01, delta)
>>> y = np.arange(-4.0, 4.01, delta)
>>> X, Y = np.meshgrid(x, y)
>>> Z = newton(X,Y)
>>>
>>> fig, ax = plt.subplots(figsize=(6,6))
>>> pl = ax.contour(X, Y, Z,np.array([-5,-4,-3,-2,-1,-0.75,-0.5]), colors='b')
>>> ax.clabel(pl, inline=True, fontsize=10)
<a list of 5 text.Text objects>
>>> plt.plot([-1,1],[0,0], 'rx')
[<matplotlib.lines.Line2D object at 0x10e3f9fd0>]
>>> ax.set_title('Equipotential surfaces of the Newton potential')
Text(0.5, 1.0, 'Equipotential surfaces of the Newton potential')
>>> plt.savefig('NewtonEquipotential.png')
# avoid losing edges with plt.savefig('NewtonEquipotential.png', bbox_inches='tight')
```

Figure 3: Simple contour plot, as generated by the code shown in Table 26 of the Newtonian gravitational potential of two equal masses
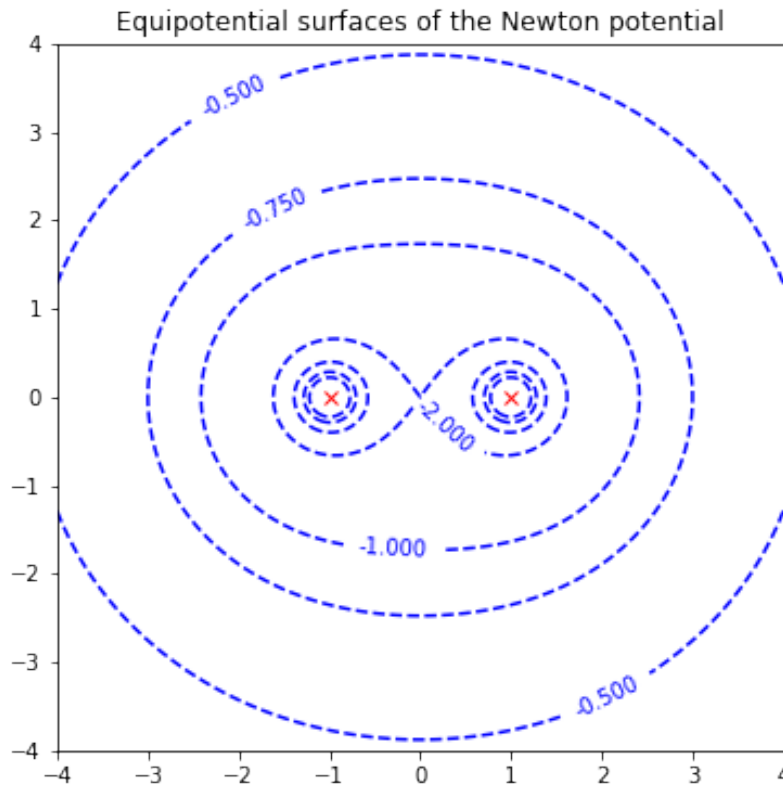
3.4 Finite differences

Guess the next number in this sequence $(c_i^{(0)})_{i=1}^{10} = (-1, 2, 23, 86, 215, 434, 767, 1238, 1871, 2690)$. There is a simple trick that may help. Make an array of differences, and differences of differences on so on

$$c_i^{(j)} = c_{i+1}^{(j-1)} - c_i^{(j-1)}, \text{ for } j = 1, 2, \ldots, 9 \text{ and } i = 1, 2, \ldots, 10 - j. \tag{3.17}$$

Once that's done, the answer to this "intelligence test" will become obvious.

This trick relies on the mathematical feature that a $N+1$ times repeated finite-difference of a polynomial of order N vanishes, the same as is true for the $N + 1$ order derivative. The proof follows from Newton's binomial theorem. Consider the monomial $p(x) = \alpha x^N$. Then

$$(x + \Delta)^N = \sum_{k=0}^{N} \binom{N}{k} x^{N-k} \Delta^k \tag{3.18}$$

This shows that $(x + \Delta)^N - x^N$ is a polynomial of order $N - 1$. Take finite differences repeatedly and use the linearity of the operation to obtain the result.

3.4.1 Assignments

Assignment 12 Finite differences

1. Use SymPy to create the table defined in Eq. (3.17). For the construction of the first column of the matrix see Table 27. The construct used in the last line of code is called *list comprehension*, which is an archaic use of the word "comprehension."[38]

---

[38]SymPy. *List Comprehensions*. URL: https://docs.python.org/2/tutorial/datastructures.html#list-comprehensions.

2. Find a polynomial $p(i)$ in i so that $p(i)$ for $i = 1, \ldots, 10$ reproduces the $c_i^{(0)}$. For this purpose you can use `linsolve([eq1, eq2, eq3],[x1, x2, x3])`, which solves for `x1, x2, x3` so that the expressions `eq1, eq2, eq3` evaluated at `[x1,x2,x3]` vanish.

Note that once you have found the polynomial p in part 2 of this assignments, you can intelligently, but not necessarily correctly, predict the value of c_n^0 for $n > 10$.

As mentioned in the table caption of Table 27, Sympy has a special real, nan, which stands for not a number. Also Numpy has one, as in this example: `import numpy as np; r = np.nan; print(ra.)`

Table 27: Table of iterated differences. Note that Sympy has a symbol `nan`—elsewhere aka NaN or NAN— that represents a IEEE 754 standard floating point quantity that is not a number nor infinity, "undefined" in other words.

```
from sympy import *
sequence = [-1,2,23,86,215,434,767,1238,1871,2690]
size = 10
table = Matrix([[nan for i in range(0,size)] for j in range(0,size)])
table[0] = Matrix([sequence[i] for i in range(0,size)])
```

3.5 Numerical differentiation

3.5.1 First-order derivatives

The derivative, if it exists, of a function f at point x is defined as

$$f^{(1)}(x) = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}. \tag{3.19}$$

If a derivative can only be obtained numerically, one replaces the limit by a finite-difference. One possible choice is the *central-difference*

$$\frac{f(x + \Delta x) - f(x - \Delta x)}{2\Delta x} = f^{(1)}(x) + \frac{1}{6} f^{(3)}(x) \Delta x^2 + \mathcal{O}\left(\Delta x^4\right), \tag{3.20}$$

where Taylor series expansion gives the right-hand side of this expression. A value of $\Delta x \neq 0$ produces an error, but if f is known only to within a factor $\varepsilon > 0$ the numerator on the left-hand side may be off by 2ε. Therefore, the total error η_1 may be as big as

$$\eta_1 = \frac{1}{6} |f^{(3)}(x)| \Delta x^2 + \frac{\varepsilon |f(x)|}{|\Delta x|}. \tag{3.21}$$

By minimization of η_1 with respect to Δx we find the optimal value of Δx

$$\Delta x = \left(\frac{3\varepsilon |f(x)|}{|f^{(3)}(x)|} \right)^{\frac{1}{3}}. \tag{3.22}$$

Note that, as should be the case, the right-hand side of this expression has the dimension of Δx. As written, this expression is of no practical use, because $f/f^{(3)}$ is not known. The standard way out is to assume that the scale of x is chosen so that this ratio is expected to be of order unity; of course, there are no guaraatees [Compare with Eq. (3.11)].

In practical cases often $f(x)$ is known, which means that two additional computations are required to obtain $f(x \pm \Delta x)$ for the use of the central difference of Eq. (3.20). If computation of f is time consuming, one can sacrifice accuracy and gain speed by using the *forward difference*:

$$f^{(1)}(x) \approx \frac{f(x + \Delta x) - f(x)}{\Delta x} = f^{(1)}(x) + \frac{1}{2} f^{(2)}(x) \Delta x + \mathcal{O}\left(\Delta x^2\right). \tag{3.23}$$

The optimal value of Δx for the forward difference is of order $\varepsilon^{\frac{1}{2}}$. The resulting error—that is the appropriately modified η_1 as in Eq. (3.21)—is of order $\varepsilon^{\frac{1}{2}}$. This is to be compared to the order of magnitude of the error for the central difference expression of Eq. (3.20), which is $\varepsilon^{\frac{2}{3}} \ll \varepsilon^{\frac{1}{2}}$.

3.5.2 Second-order derivatives

$$\frac{f(x + \Delta x) - 2f(x) + f(x - \Delta x)}{\Delta x^2} = f^{(2)}(x) + \frac{1}{12}f^{(4)}(x)\Delta x^2 + \mathcal{O}\left(\Delta x^4\right). \tag{3.24}$$

This expression produces an error bounded from above by

$$\eta_2 = \frac{1}{12}|f^{(4)}(x)|\Delta x^2 + \frac{4\varepsilon|f(x)|}{\Delta x^2} \tag{3.25}$$

The optimal Δx in this case is on the order of $\varepsilon^{\frac{1}{4}}$ as is the minimal error. The factor of 4 in the numerator of the second term on the right-hand side of this expression is the sum of the absolute values of the coefficients in the numerator of the left-hand side of Eq. (3.24).

3.5.3 Lagrange interpolation formula

Suppose that for $m + 1$ values x_i, $i = 0, 1, \ldots, m$ function values $f_i \equiv f(x_i)$ are given. The Lagrange interpolation formula constructs a polynomial of order $m + 1$, $Q(x)$, such that $Q(x_i) = f_i$. The polynomial is of the following form

$$Q(x) = \sum_{i=0}^{m} f_i \delta_i(x). \tag{3.26}$$

The polynomials $\delta_i(x)$ are constructed to satisfy the conditions given in the following two separately numbered equations:

$$\delta_i(x_j) = \begin{cases} 0, & \text{if } i \neq j \\ 1, & \text{if } i = j \end{cases} \tag{3.27}$$

for $i, j = 0, 1, \ldots, m$. Functions δ_i that satisfy these conditions are

$$\delta_i(x) = \frac{\displaystyle\prod_{\substack{k=0 \\ k \neq i}}^{m}(x - x_k)}{\displaystyle\prod_{\substack{k=0 \\ k \neq i}}^{m}(x_i - x_k)}. \tag{3.28}$$

Define Φ by

$$\Phi(x) = \prod_{k=0}^{m}(x - x_k). \tag{3.29}$$

Verify that the following relationship holds:[39]

$$\delta_i(x) = \frac{\Phi(x)}{(x - x_i)\Phi'(x_i)}. \tag{3.30}$$

Eqs. (3.26) through (3.30) can be used to construct finite-difference approximations that agree with the derivatives to higher order in Δx than the expressions given in Eqs. (3.20) and (3.24). The same can be done for higher-order derivatives. To suppress higher-order corrections more function evaluations are required, which can be computationally expensive. Another problem is that as the expressions grow in complexity, they will contain larger and larger coefficients that may cause serious cancellations and loss of accuracy.

3.5.4 Assignments

Assignment 13 Numerical derivatives: general finite-differences

1. For $m \in \mathbb{N}$ consider abscissas $x = -mh/2, -mh/2 + h, \ldots mh/2$ and the corresponding function values $f(x)$. Note that for odd values of m the function value $f(0)$ is missing. Compute $Q(x)$ as defined in Eq. (3.26) symbolically with SymPy.

2. By symbolically taking the first and second-order derivatives at $x = 0$ show that for $m = 8$ the lowest order correction to the resulting finite-difference approximants to the derivatives of f at $x = 0$ are proportional to h^8.

---

[39]G. Dahlquist and N. M. Åke Björk. *Numerical Methods.* Prentice-Hall, 1974.

3. Find the optimal value of h assuming that the function values of f are of order unity for all relevant values of x.

Hints:

- The function f is any arbitrary function with sufficiently many derivatives. The step size h plays the role of Δx in Eqs. (3.20) and (3.24).
- See Table 28 for sample code that illustrates a possible way of using SymPy. Also see Fig. 4.
- The case $m = 3$ gives

$$f^{(1)}(0) \approx \frac{f\left(-\frac{3h}{2}\right) - 27f\left(-\frac{h}{2}\right) + 27f\left(\frac{h}{2}\right) - f\left(\frac{3h}{2}\right)}{24h} = f^{(1)}(0) - \frac{3}{640}f^{(5)} + \mathcal{O}(h^6). \quad (3.31)$$

- Sympy has routines called `simplify` and `cancel` that reduce the complexity of symbolic expressions. They can be used to cancel common factors in numerator and denominator.

Table 28: Sample code for deriving finite-difference approximants for numerical derivatives; see the assignment on page 33. Note that _x is a Python dummy variable used in the derivative. In Fig. 4 this dummy variable is denoted by ξ.

```
>>> from sympy import *
>>> from sympy.abc import c, g, x
>>> g = Function('g')
>>> def taylor(x):
...    return series(g(x), x, n = 2)
...
>>> taylor(x)
g(0) + x*Subs(Derivative(g(_x), _x), _x, 0) + O(x**2)
>>> simplify((taylor(x)-taylor(-x))/(2*x))
Subs(Derivative(g(_x), _x), _x, 0) + O(x)
>>> simplify((taylor(x)-2 * g(0) + taylor(-x))/(x**2))
O(1)
>>> tayl = simplify((taylor(x)-g(0))/x)
>>> print('tayl ', tayl)
tayl  Subs(Derivative(g(_x), _x), _x, 0) + O(x)
>>> Fi = Function('Fi')
>>> m = Symbol('m')
>>> h = Symbol('h')
>>> f = Function('f')
>>> def point(i,m,h):
...    return (i-Rational(1,2)*m)*h
...
>>> def Fi(x, m):
...    p = 1
...    for i in range(0, m+1):
...       p *= x-point(i,m,h)
...    return p
...
>>> m = 2
>>> print('m = ', m)
m =  2
>>> for i in range(0, m+1): print('point ', i, ': ', point(i,m,h))
...
point  0 :  -h
point  1 :  0
point  2 :  h
>>> print('Fi(x, 2) =', Fi(x, 2))
Fi(x, 2) = x*(-h + x)*(h + x)
>>> print('Fi(x, 3) =', Fi(x, 3))
Fi(x, 3) = (-3*h/2 + x)*(-h/2 + x)*(h/2 + x)*(3*h/2 + x)
>>> print('Fi(x, 4) =', Fi(x, 4))
Fi(x, 4) = x*(-2*h + x)*(-h + x)*(h + x)*(2*h + x)
>>> eps = Symbol('eps')
>>> s = solve((h**2 + eps/h).diff(h),h)[0]
>>> print('minimum is at h = ', s)
minimum is at h =  2**(2/3)*eps**(1/3)/2
>>> eps_mach = 7/3-4/3-1
>>> v = (h**2 + eps/h).subs(h,s).subs(eps,eps_mach)
>>> print('value at minimum: ', N(v))
value at minimum:  6.92991766249849e-11
```

Figure 4: Taylor expansions of some standard finite-difference approximants for first- and second-order derivatives; see the assignment on page .

```
[1]: from sympy import *
     from sympy.abc import c, g, x
     g = Function('g')
     def taylor(x):
       return series(g(x), x, n = 6)

     tail0 = taylor(x)
     tayl1_central = simplify((taylor(x)-taylor(-x))/(2*x))
     tayl2 = simplify((taylor(x)-2 * g(0) + taylor(-x))/(x**2))
     tayl1_forward = simplify((taylor(x)-g(0))/x)
```

Taylor expansion of g at $x = 0$:

```
[2]: tail0
```

$[2]$:

$$g(0) + x \left.\frac{d}{d\xi}g(\xi)\right|_{\xi=0} + \frac{x^2 \left.\frac{d^2}{d\xi^2}g(\xi)\right|_{\xi=0}}{2} + \frac{x^3 \left.\frac{d^3}{d\xi^3}g(\xi)\right|_{\xi=0}}{6} + \frac{x^4 \left.\frac{d^4}{d\xi^4}g(\xi)\right|_{\xi=0}}{24} + \frac{x^5 \left.\frac{d^5}{d\xi^5}g(\xi)\right|_{\xi=0}}{120} + O\left(x^6\right)$$

Taylor expansion of central difference approximation of $g'(0)$. The lowest-order correction is $O(x^2)$.

```
[3]: tayl1_central
```

$[3]$:

$$\left.\frac{d}{d\xi}g(\xi)\right|_{\xi=0} + \frac{x^2 \left.\frac{d^3}{d\xi^3}g(\xi)\right|_{\xi=0}}{6} + \frac{x^4 \left.\frac{d^5}{d\xi^5}g(\xi)\right|_{\xi=0}}{120} + O\left(x^5\right)$$

Taylor expansion of forward difference approximation of $g'(0)$. The lowest-order correction is $O(x)$.

```
[4]: tayl1_forward
```

$[4]$:

$$\left.\frac{d}{d\xi}g(\xi)\right|_{\xi=0} + \frac{x \left.\frac{d^2}{d\xi^2}g(\xi)\right|_{\xi=0}}{2} + \frac{x^2 \left.\frac{d^3}{d\xi^3}g(\xi)\right|_{\xi=0}}{6} + \frac{x^3 \left.\frac{d^4}{d\xi^4}g(\xi)\right|_{\xi=0}}{24} + \frac{x^4 \left.\frac{d^5}{d\xi^5}g(\xi)\right|_{\xi=0}}{120} + O\left(x^5\right)$$

Taylor expansion of central difference approximation of $g''(0)$. The lowest-order correction is $O(x^2)$.

```
[5]: tayl2
```

$[5]$:

$$\left.\frac{d^2}{d\xi^2}g(\xi)\right|_{\xi=0} + \frac{x^2 \left.\frac{d^4}{d\xi^4}g(\xi)\right|_{\xi=0}}{12} + O\left(x^4\right)$$

```
[6]: print(tayl2)
```

```
Subs(Derivative(g(xi), (xi, 2)), xi, 0) + x**2*Subs(Derivative(g(xi), (xi, 4)),
xi, 0)/12 + O(x**4)
```

4 Extrapolation methods

Often iteration methods produce a sequence of increasingly accurate results. In many cases these sequences can be extrapolated once and often repeatedly to obtain dramatically more rapid convergence than is possible by continued iteration.

4.1 Richardson extrapolation

Suppose that

$$f(h) = f(0) + a_1 h^p + \mathcal{O}(h^q). \tag{4.1}$$

Suppose we know the value of $f(h)$ and p and that $0 < p < q$, but we don't know a_1, while $f(0) = \lim_{h \to 0} f(h)$ is the quantity of interest. Suppose we compute $f(h')$ with $h' \neq h$. From the expansion in Eq. (4.1) it follows that

$$f(h') = f(0) + a_1 (h')^p + \mathcal{O}(h'^q). \tag{4.2}$$

By combining Eqs. (4.1) and (4.2) we can eliminate the unknown constant a_1. In so doing we obtain an expression that converges more rapidly to $f(0)$, namely

$$f(0) = f(h) + \frac{f(h) - (h'/h)^{p_1} f(h')}{(h'/h)^{p_1} - 1} + \mathcal{O}(h^{p_2}). \tag{4.3}$$

In other words, we have sped up convergence because we have eliminated the dominant, $\mathcal{O}(h^p)$ correction term.

The process can be repeated, if we know, or can guess, the value of p_2 in the remaining correction term. The resulting process is called repeated Richardson extrapolation.

The algorithm is given in Algorithm 2 and illustrated in Table 29.

Algorithm 2 The repeated Richardson algorithm; $a \leftarrow b$ means replace the value of a by the value of b.

Start from $h_1 > h_2 > \cdots > h_n > 0$; k=0.

1. For $i = 1, \ldots, n$ compute $f(h_i) \equiv f_1(h_i)$

2. $k \leftarrow k + 1$; For $i = 1, \ldots, n - k$ compute $f_k(h_i)$ as defined or suggested in Table 29 with appropriate values of p_k.

3. Repeat step 2 until accuracy or data are exhausted.

Table 29: Repeated Richardson extrapolation. The third column is obtained from the second by the substitutions $f_2 \to f_3$ and $p_1 \to p_2$. The process can be repeated to generate a sequence of columns of decreasing length.

$f_1(h_1)$	$f_2(h_1) = [h_1^{p_1} f_1(h_2) - h_2^{p_1} f_1(h_1)]/(h_1^{p_1} - h_2^{p_1})$	$f_3(h_1) = \ldots$
$f_1(h_2)$	$f_2(h_2) = [h_2^{p_1} f_1(h_3) - h_3^{p_1} f_1(h_2)]/(h_2^{p_1} - h_3^{p_1})$	$f_3(h_2) = \ldots$
$f_1(h_3)$	$f_2(h_3) = [h_3^{p_1} f_1(h_4) - h_4^{p_1} f_1(h_3)]/(h_3^{p_1} - h_4^{p_1})$	—
$f_1(h_4)$	—	

4.2 Aitken extrapolation

Sometimes one knows that a series converges as

$$f_{n,0} = a + bc^n + o(c^n) \tag{4.4}$$

with $|c| < 1$ and where the last term on the right indicates a correction approaching zero more rapidly than c^n as $n \to \infty$. From three successive values of f_n one finds

$$f_{n,1} = \frac{f_{n,0} f_{0,n+2} - f_{0,n+1}^2}{f_{0,n} - 2 f_{0,n+1} + f_{0,n+2}} \tag{4.5}$$

The same procedure may be applied to the new series $f_{n,1}$ and so on, if there is reason to believe that the corrections are more rapidly decaying exponentials. Of course, numerical cancellations will ultimately limit the procedure as is the case for repeated Richardson extrapolation. The repeated process is also called the Shanks transformation.

5 Numerical evaluation of definite integrals

5.1 Trapezoidal rule

We can compute integrals numerically with the trapezoidal rule. As long as the integration interval is finite we lose no generality by assuming that the integral interval runs from 0 to 1.

$$I = \int_0^1 f(x)\,dx \approx \Delta x \left(\frac{1}{2} f(0) + \sum_{i=1}^{N-1} f(i\Delta x) + \frac{1}{2} f(1) \right) \equiv T_N, \tag{5.1}$$

where $\Delta x = N^{-1}$.

For increasing N the approximation improves and becomes exact in the limit $N \to \infty$ for "reasonable" functions f.[40]

Figure 5: Trapezoidal rule
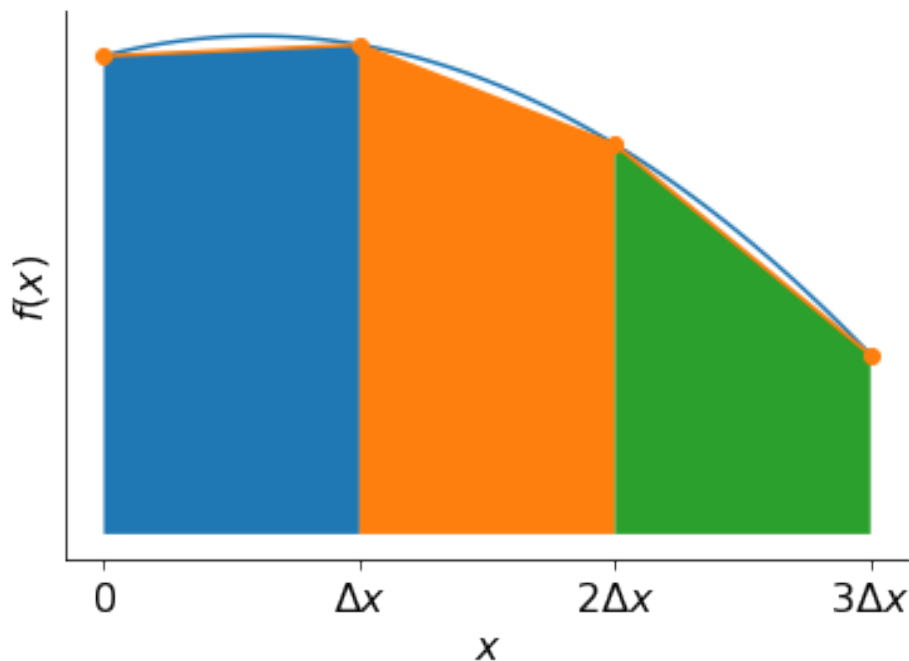
Convergence as $N \to \infty$ of the trapezoidal rule typically is power-law convergence, *i.e.*,

$$T_N = I + a_2 N^{-2} + a_4 N^{-4} + \cdots + a_{2k} N^{-2K} + \mathcal{O}\left(N^{2k+2}\right). \tag{5.2}$$

Here the a_{2k} ae constants, the values of which play no significan role.

As illustrated in Fig. 5, the approximate contribution to the integral from trapezoid i with basis $\big[(i-1)\Delta x, i\Delta x\big]$ is $\Delta x \big[f(x_i) + f(x_{i+1})\big]/2$, the absolute value of which is the area of the trapezoid. This differs from the contribution to the integral by an amount proportional to the area between the curve $y = f(x)$ and the trapezoid in that interval. The curve and the top of the trapezoid differ by an amount on the order of Δx^2, which means that the difference in area is proportional to Δx^3.

The number of trapezoids in the interval $[0, 1]$ is Δx^{-1}, which explains the second term on the right-hand side of Eq. (5.2) Of the higher-order corrections the odd term in the Taylor series expansion about the

---

[40]W. H. Press et al. *Numerical Recipes: 4.2 Integration of Functions—Elementary Algorithms*. URL: http://phys.uri.edu/~nigh/NumRec/bookfpdf/f4-2.pdf.

midpoint of the interval $\left[(i-1)\Delta x, i\Delta x\right]$ vanish, because of the anti-symmetry of odd terms about those points.

Assuming that the required derivatives exist in the integration interval, one can derive the Euler-McLaurin expansion:[41]

$$T(N) = \int_0^1 f(x)\,dx \quad + \quad \frac{1}{12}N^{-2}[f^{(1)}(1) - f^{(1)}(0)] - \frac{1}{720}N^{-4}[f^{(3)}(1) - f^{(3)}(0)] \cdots +$$
$$+ \quad c_{2k}[f^{(2k-1)}(1) - f^{(2k-1)}(0)] + \mathcal{O}\left(N^{-(2k+2)}\right). \tag{5.3}$$

The coefficients c_{2k} can easily be found from the assumption that indeed such an expansion exists and that the same coefficients are valid for any function f.[42] For the special case $f(x) = e^x$ we can calculate the integral, the trapezoidal approximation, and the derivatives at the endpoints. This gives

$$T(N) = \Delta x \left(\frac{1}{2} + \sum_{k=1}^{N-1} e^{k\Delta x} + \frac{1}{2}e\right) = \Delta x \left(\frac{e^{N\Delta x} - 1}{e^{\Delta x} - 1} + \frac{1}{2}(e - 1)\right). \tag{5.4}$$

Using $N\Delta x = 1$ we find

$$T(N) = \frac{\Delta x}{2}\frac{(e-1)(e^{\Delta x}+1)}{e^{\Delta x} - 1}. \tag{5.5}$$

Substitute this into Eq. (5.3) to find

$$\frac{1}{2}\Delta x\frac{e^{\Delta x}+1}{e^{\Delta x}-1} = \frac{1}{2}\Delta x\frac{e^{\frac{1}{2}\Delta x} + e^{-\frac{1}{2}\Delta x}}{e^{\frac{1}{2}\Delta x} - e^{-\frac{1}{2}\Delta x}} = \sum_{k=0}^{\infty} c_{2k}\Delta x^{2k}. \tag{5.6}$$

The expression in the middle shows that the expressions are symmetric under $\Delta x \to -\Delta x$. The conclusion is that the function on the left is the generating function of the coefficients c_{2k}, that is a function of which the c_{2k} are the Taylor series expansion coefficients.

To speed up the convergence of the trapezoidal integration one can use Richardson extrapolation. The practical way to do this is to decrease Δx by a factor of two. The factor of two is convenient because $T(N)$ and T_{2N} have about half of the terms in T_{2N} sum has already been computed.

From Table 29 with $f_1 \to T \equiv T_1$, $h_1 \to N$, $h_2 \to 2N$, and $p \to 2$ we find

$$T_2(N) = \frac{4}{3}T_1(2N) - \frac{1}{3}T_1(N)). \tag{5.7}$$

This process can be repeated. The next step involves a correction term with $p_2 = 4$, as implied by Eq. (5.2) gives

$$T_3(N) = \frac{16}{15}T_2(2N) - \frac{1}{15}T_2(N). \tag{5.8}$$

For sufficiently well-behaved periodic functions integrated over a period, the corrections are exponential rather than of power-law form i.e.,

$$T(N) = I + be^{-cN} + \text{ higher-order correction}, \tag{5.9}$$

with $c > 0$. The constant c is the distance to the nearest singularity of the integrand in the imaginary plane. For more on this see this paper by Weideman.[43]

5.2 Assignments

Assignment 14 Numerical integration: Richardson extrapolation

1. Use the trapezoidal rule to write a simple program to integrate $\cos x$ from 0 to 1;
2. Calculate $T(N)$ as given in Eq. (5.2) for $N = 2, 2^2, 2^3 \ldots,$.

[41]T. M. Apostol. "An Elementary View of Euler's Summation Formula". In: *The American Mathematical Monthly* 106.5 (1999), pp. 409–418. DOI: 10.1080/00029890.1999.12005063 uses Bernoulli numbers B_k: $c_{2k} = (-1)^{k+1}B_k/(2k)!$.

[42]§7.4.4 of Dahlquist and Björk, *Numerical Methods*

[43]J. A. C. Weideman. "Numerical Integration of Periodic Functions: A Few Examples". In: *The American Mathematical Monthly* 109.1 (2002), pp. 21–36. DOI: 10.2307/2695765. URL: http://www.jstor.org/stable/2695765.

3. Generalize Eqs. (5.7) and (5.8) to an arbitrary number of m repeated Richardson extrapolation steps.

4. Subtract the exact value of the integral to reduce cancellations and to study the convergence behavior. Make a table of the results and figure out after how many Richardson iterations the process becomes meaningless because of the finite precision of floating point arithmetic.

Assignment 15 Energy of a 3D crystal: Richardson extrapolation

Consider a 3-dimensional cubic crystal with particles located at positions (n_1, n_2, n_3) with the $n_i \in \mathbb{Z}$. The particles interact with a Lennard-Jones potential

$$V(r) = \frac{1}{r^6} \left(\frac{1}{r^6} - 2 \right), \tag{5.10}$$

where r is the inter-particle distance.

The name of the game is to compute E, the energy per lattice site of the infinite crystal.

1. This can be done by computing for increasing values of n the sum

$$E_n = \sum_{n_1=-n}^{n}{}' \sum_{n_2=-n}^{n}{}' \sum_{n_3=-n}^{n}{}' V\left(\sqrt{n_1^2 + n_2^2 + n_3^2}\right), \tag{5.11}$$

where the primes indicate that the term $(n_1, n_2, n_3) = (0, 0, 0)$ should be excluded from the sum.

2. Replacing the sum in Eq. (5.11) by an integral over a sphere of radius n for order-of-magnitude purposes suggests that

$$E_n = E_\infty + \sum_{i=0}^{\infty} c_i r^{-p-i}. \tag{5.12}$$

What is the value of p?

3. Show what repeated Richardson extrapolation does to the convergence of E_n.

Assignment 16 1D crystal: integrated rest term

Consider a one-dimensional crystal with

$$V(r) = \frac{1}{r^2} \left(\frac{1}{r^2} - 2 \right). \tag{5.13}$$

The energy per lattice site in this case can be calculated exactly in terms of the Riemann zeta function[44]:

$$E_\infty(\alpha) = 2 \sum_{n=1}^{\infty} n^{-\alpha}(n^{-\alpha} - 2) = 2\zeta(2\alpha) - 4\zeta(\alpha). \tag{5.14}$$

For $\alpha = 2$ this becomes

$$E_\infty = \frac{1}{45}\pi^2 \left(\pi^2 - 30 \right). \tag{5.15}$$

The part of the sum left off in E_n can be related to the trapezoidal rule for integration

$$\frac{1}{2}(E_\infty - E_n) = \frac{1}{2}V(n+1) + \frac{1}{2}V(n+1) + V(n+2) + V(n+3) + \cdots \approx \frac{1}{2}V(n+1) + \int_{n+1}^{\infty} V(\nu)\, d\nu. \tag{5.16}$$

- Compute E_n and compare with what you get by adding the approximate rest term given by Eq. (5.16). Collect the results in a table or plot them to show how the rest term accelerates convergence.

---

[44]This is a zeta function classic: H. M. Edwards. *Riemanns' Zeta Function.* URL: https://archive.org/details/riemannszetafunc00edwa_0

6 Probability theory

Let $\tilde{\xi}$ be a stochastic variable, that is a variable combined with a probability that it assumes a particular value, as defined more precisely below. The possible outcomes of a physical experiment is an example. Consider the *univariate* case in which one number describes the outcome. A position measurement in three dimensions is an example of a multivariate stochastic variable because there are three coordinates. The tilde indicates that a symbol refers to a stochastic variable. A particular realization of $\tilde{\xi}$ will be denoted by ξ without the tilde.

Let $\mathbf{P}\{A\}$ denote the probability of event A. The cumulative distribution function F of $\tilde{\xi}$ is defined by

$$F(\xi) = \mathbf{P}\{\tilde{\xi} \leq \xi\}. \tag{6.1}$$

$F(\xi)$ goes to zero as $x \to -\infty$, to unity as $\xi \to \infty$ and it is monotonic non-decreasing. If F has a discontinuity at ξ_0 the event $\tilde{\xi} = \xi_0$ has a non-zero probability. Eq. (6.1) implies that

$$\mathbf{P}\{\tilde{\xi} = \xi_0\} = \lim_{x \downarrow \xi_0} F(x) - \lim_{x \uparrow \xi_0} F(x), \tag{6.2}$$

an expression that does not depend on the conventional choice of right-continuity implied by the $\leq$ sign in Eq. (6.1).

In the special case that $\tilde{\xi}$ assumes only a finite number of n possible values $\xi_1 < \xi_2, \cdots < \xi_n$ with probabilities $p_1 < p_2, \cdots < p_n$ we have

$$F(\xi) = \begin{cases} 0 & \text{for } \xi < \xi_1, \\ \sum_{k=1}^{i-1} p_k & \text{for } \xi_{i-1} \leq \xi < x_i \text{ and } 2 \leq i \leq n, \\ 1 & \text{for } \xi_n \leq \xi. \end{cases} \tag{6.3}$$

The probability density function $\rho(x)$ of $\tilde{\xi}$ is defined as

$$\rho_{\tilde{\xi}}(x) = \lim_{\Delta x \downarrow 0} \frac{\mathbf{P}\{x < \tilde{\xi} < x + \Delta x\}}{\Delta x} \tag{6.4}$$

If the cumulative distribution function F has a discontinuity of size p_1 at $x = x_1$, the probability density function will contain a Dirac δ-function term $p_1 \delta(x - x_1)$. Also see section 8.1 on page 52.

In terms of an infinitesimal $d\xi > 0$ Eq. (6.4) takes the form

$$\rho_{\tilde{\xi}}(\xi)\, d\xi = \mathbf{P}\{\tilde{\xi} \in (\xi, \xi + d\xi)\}. \tag{6.5}$$

That is $\rho_{\tilde{\xi}}(\xi)\, d\xi$ is the probability that $\tilde{\xi}$ assumes a value in the interval $(\xi, \xi + d\xi)$.

All of the above can be generalized to the bivariate case defined by two stochastic variables $\tilde{\xi}_1, \tilde{\xi}_2$. In that case, Eq. (6.5) becomes

$$\rho_{\xi_1, \xi_2}(\xi_1, \xi_2)\, d\xi_1\, d\xi_2 = \mathbf{P}\{(\tilde{\xi}_1, \tilde{\xi}_2) \in (\xi_1, \xi_1 + d\xi_1) \times (\xi_2, \xi_2 + d\xi_2)\}, \tag{6.6}$$

where the $\times$ symbol indicates the direct product, which when applied to two one-dimensional intervals produces a two-dimensional rectangle. Eq. (6.6) can be generalized to more than two stochastic variables.

The mean (aka the average) or the expectation value $\langle \tilde{\xi} \rangle$ is defined as

$$\langle \tilde{\xi} \rangle = \int_{-\infty}^{\infty} \xi \rho_{\tilde{\xi}}(\xi)\, d\xi \equiv \mu \tag{6.7}$$

The variance is defined by

$$\mathrm{var}\, \tilde{\xi} = \langle (\tilde{\xi} - \mu)^2 \rangle \equiv \sigma^2. \tag{6.8}$$

σ is called the standard deviation. It can be thought of as setting the scale of the probability density function.

More generally, the n-th ($n \in \mathbb{N}_0$) moment is defined as

$$\langle \tilde{\xi}^n \rangle = \int_{-\infty}^{\infty} x^n \rho_{\tilde{\xi}}(x)\, dx. \tag{6.9}$$

Similarly, the n-th moment about the mean (aka central moment) is defined as

$$\langle (\tilde{\xi} - \mu)^n \rangle = \int_{-\infty}^{\infty} (\xi - \mu)^n \rho_{\tilde{\xi}}(\xi)\, d\xi. \tag{6.10}$$

The zeroth moment is 1 and it always exists. Higher moments may not exist.

Example 1

For the Cauchy (aka Lorentz or Breit-Wigner) distribution $\pi^{-1}/(1 + x^2)$ neither the mean nor the variance are defined; both integrals diverge at infinity.

Example 2

The Gaussian, aka normal distribution, with mean μ and standard deviation σ is defined as

$$\mathcal{N}(x, \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x - \mu)^2}{2\sigma^2}\right) \tag{6.11}$$

Beware that the in the mathematical statistical literature the Gaussian probability density function is usually parameterized by the variance σ^2 and denoted by $\mathcal{N}(x, \mu, \sigma^2)$. The computational crowd, as does SciPy, often use the standard deviation σ as the parameter, writing $\mathcal{N}(x, \mu, \sigma)$. As long as symbols are used the distinction is clear, but once the symbols are replaced by numbers, you have to know what the convention is.

6.1 The central limit theorem

Events A and B are independent if and only if

$$\mathbf{P}\{A \cap B\} = \mathbf{P}\{A\}\mathbf{P}\{B\}. \tag{6.12}$$

For the probability density function independence means

$$\rho_{\tilde{\xi}_1, \tilde{\xi}_2}(\xi_1, \xi_2) = \rho_{\tilde{\xi}_1}(\xi_1)\rho_{\tilde{\xi}_2}(\xi_2), \tag{6.13}$$

and similarly for more than two stochastic variables.

Note that for n independent stochastic variables $\tilde{x}_i$ one has

$$\operatorname{var} \sum_{i=1}^{n} \tilde{x}_i = \sum_{i=1}^{n} \operatorname{var} \tilde{x}_i. \tag{6.14}$$

The equality results from the fact that upon calculation of the expectation value the cross terms vanish because of the independence of the stochastic variables.

Suppose that $\tilde{\xi}_i$ with $i = 1, \ldots, n$ are independent, identically distributed stochastic variables with mean μ and variance σ^2. Define

$$\tilde{\nu}_i = \frac{\tilde{\xi}_i - \mu}{\sigma}, \tag{6.15}$$

$$\tilde{S}_n = \frac{1}{\sqrt{n}} \sum_{i=1}^{n} \tilde{\nu}_i. \tag{6.16}$$

The central limit theorem states that the probability density function of $\tilde{S}_n$ converges to the standard normal distribution $\mathcal{N}(0, 1)$.

The proof of this uses the *characteristic function* of a probability density function defined by

$$\varphi_{\tilde{\xi}}(t) \equiv \langle e^{it\tilde{\xi}} \rangle = \int_{-\infty}^{\infty} e^{i\xi t} \rho_{\tilde{\xi}}(\xi) \, d\xi. \tag{6.17}$$

A probability density function defines a characteristic function and *vice versa.* The relationship apart from conventional factors is the same as that between a function and its Fourier transform. Therefore, let's calculate the asymptotic, $n \to \infty$, characteristic function of $\tilde{S}_n$. From the fact that the $\tilde{\xi}_i$ are independent and identically distributed it follows that

$$\varphi_{\tilde{S}_n}(t) = \left[\varphi_{\tilde{\xi}_1}\left(\frac{t}{\sqrt{n}}\right)\right]^n = \left[1 - \frac{1}{2n}t^2 + \mathcal{O}\left(\left(\frac{t}{\sqrt{n}}\right)^3\right)\right]^n \tag{6.18}$$

The right-hand side of this equation follows by Taylor series expansion. Use

$$\lim_{n \to \infty} \left(1 + \frac{x}{n}\right)^n = e^x. \tag{6.19}$$

Table 30: How to plot a histogram that approximates a probability density function: this is accomplished by 'density = True'

```
import numpy as np
import matplotlib.pyplot as plt
n = 100000
z = 2*np.random.random(n)
plt.hist(z, density = True, bins = int(np.sqrt(n)))
```

This implies that

$$\varphi_{\tilde{S}_n}(t) \to e^{-\frac{1}{2}t^2}, \text{ as } n \to \infty, \tag{6.20}$$

the characteristic function of the $\mathcal{N}(0,1)$, the standard normal probability density function. The latter can be seen from

$$\varphi_{\mathcal{N}(0,1)}(t) \quad = \quad \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{-\frac{1}{2}(x^2+ixt)} \, dx \tag{6.21}$$

$$= \quad \frac{e^{-\frac{1}{2}t^2}}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{-\frac{1}{2}(x-it)^2} \, dx \tag{6.22}$$

$$= \quad e^{-\frac{1}{2}t^2} \tag{6.23}$$

The step leading from Eq. (6.21) to Eq. (6.22) results from completing the square in the exponent. The shift off the real axis of the integration path in the next step does not change the integral, which is a consequence of the analyticity of the integrand.

A more useful, equivalent statement can be obtained by rescaling. This is about the estimator of the keywordmean —or if you like average or expectation value— $\tilde{\mu}$, the stochastic variable defined by

$$\tilde{\mu} = \frac{1}{n} \sum_{i=1}^{n} \tilde{\xi}_i. \tag{6.24}$$

The statement is that this estimator has a probability density function that approaches the standard normal distribution $\mathcal{N}(\mu, \frac{\sigma^2}{n})$ in the large n limit. A sufficient assumption for this is that the $\tilde{\xi}_i$ are independent identically distributed stochastic variables with mean μ and variance σ^2.[45]

6.1.1 Assignments

Assignment 17 Add independent $U(0,1)$ stochastic variables

1. Generate two arrays, z_1 and z_2, of n $U(0,1)$ random numbers. Make a histogram of the $z_1 + z_2$.

2. Plot the probability density function of $z_1 + z_2$.

3. Make a single plot that shows that this probability density function is approximated by the histogram.

4. Do the same for m arrays z_i, $i = 1, \ldots, m$ and $m^{-1} \sum_{i=1}^{m} z_i$

5. Compare this with the Gaussian (normal) probability density function.

Hints:

1. How to plot a histogram: see Table 30.

2. To find the Gaussian distribution that resembles the sum of the independent variables calculate the variance of a a $U(0,1)$ stochastic variable and use the fact that for independent stochastic variables the variance of the sum is the sum of the variances, Eq. 6.14.

---

[45]For generalizations see Wikipedia. *Central limit theorem: generalized theorem.* URL: https://en.wikipedia.org/wiki/Central_limit_theorem#Generalized_theorem

6.2 The χ^2 distribution

Suppose we have stochastic variables $\tilde{\xi}$ independently, standard normal $\mathcal{N}(0,1)$ distributed. The question is what is the probability density function of

$$\tilde{\chi^2}_n = \sum_{i=1}^{n} \tilde{\xi_i}^2. \tag{6.25}$$

Before answering that question let's consider some geometry in two and higher dimensions that can help us to evaluate for instance the normalization constant of the $\mathcal{N}(0,1)$ distribution, that is the integral

$$I = \int_{-\infty}^{\infty} e^{-x^2/2} \, dx. \tag{6.26}$$

By evaluating I^2 using polar coordinates in two dimensions we find:

$$
\begin{aligned}
I^2 &= \int_{-\infty}^{\infty} e^{-x^2/2} \, dx \int_{-\infty}^{\infty} e^{-x^2/2} \, dy && \text{(6.27)} \\
&= \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} e^{-(x^2+y^2)/2} \, dx \, dy && \text{(6.28)} \\
&= \int_{0}^{\infty} r e^{-r^2/2} \, dr \int_{0}^{2\pi} d\phi = 2\pi. && \text{(6.29)}
\end{aligned}
$$

A similar trick can be used in higher dimensions. Now that we know the values of I, we have

$$\int_{-\infty}^{\infty} \cdots \int_{-\infty}^{\infty} e^{-\sum_i x_i^2} \, dx_1 \ldots dx_p = \pi^{\frac{p}{2}}, \tag{6.30}$$

once again exploiting the factorization property of the exponential function. Alternatively, because of the rotational symmetry of the integrand, we can integrate over an infinitesimal spherical shell of radius R $d(c_p R^p) = p c_p R^{p-1} \, dR$ with c_p the area of the unit sphere in p dimensions. This gives

$$\int_{-\infty}^{\infty} \cdots \int_{-\infty}^{\infty} e^{-\sum_i x_i^2} \, dx_1 \ldots dx_p = \int_{0}^{\infty} e^{-R^2} p c_p R^{p-1} \, dR. \tag{6.31}$$

We conclude that

$$c_p = \frac{\pi^{\frac{p}{2}}}{p \int_{0}^{\infty} e^{-R^2} R^{p-1} \, dR}. \tag{6.32}$$

Integrating the denominator by parts, we can write

$$c_p = \frac{\pi^{\frac{p}{2}}}{\Pi\left(\frac{p}{2}\right)}, \tag{6.33}$$

where

$$\Pi(x) \equiv \int_{0}^{\infty} e^{-t} t^x \, dx. \tag{6.34}$$

The function Π is related to the Γ–function by $\Pi(x) = \Gamma(x+1)$.

Now consider the probability density function of the stochastic variable $\tilde{\chi^2}$ defined in Eq. (6.25). For the probability density function Q_n of χ_n we have

$$Q_n(q_1, q_2, \ldots, q_n) = (2\pi)^{n/2} e^{-\sum_{i=1}^{n} q_i/2}, \tag{6.35}$$

Let

$$
\begin{aligned}
F(\chi^2; n) d\chi^2 &= P(\chi^2 \leq \tilde{\chi^2} \leq \chi^2 + d\chi^2) && \text{(6.36)} \\
&= \int_{\boldsymbol{q} \in S} Q(\boldsymbol{q}) \, dq_1 dq_2 \ldots dq_n && \text{(6.37)} \\
&= 2 d_n e^{-\chi^2/2} \chi^{n-1} \, d\chi, && \text{(6.38)}
\end{aligned}
$$

where S is a spherical shell in n dimensions of radius χ and thickness $d\chi$. Note that $d\chi^2 = 2\chi d\chi$. The constant d_n is related to c_d but we can also obtain it directly from the requirement that the the probability density function F must integrate unity. Putting all of this together gives

$$F(\chi^2; n) = \frac{e^{-\chi^2/2}\chi^{n-2}}{2^{n/2}\Gamma(n/2)}. \tag{6.39}$$

This expression defines the χ^2 distribution with n degrees of freedom. Keep in mind that there are different conventions depending on whether the first argument is χ^2 or, as in the definition used by SciPy, χ.

In the large n limit, as implied by the central limit theorem, the χ^2 distribution approaches the normal distribution with average $\mu = n$ and variance $\sigma^2 = 2n$.

The chi^2 distribution plays an important role in parameter fitting and the determination of confidence intervals of the parameters obtained this way. The validity of the latter stands and falls with validity of the underlying assumption that the conditions of the central limit theorem are satisfied.

In practice there is no guarantee that this is the case. These days less mathematically elegant methods but more reliable Monte Carlo approaches are often used instead.

7   Maximum-likelihood estimators

Suppose we have a data set consisting realizations x_i with $i = 1, 2, \ldots, N$ of independently distributed $\mathcal{N}(\mu, \sigma^2)$ Gaussian stochastic variables $\tilde{x}_i$ with μ and σ^2 unknown. Given these realizations what are the most likely values of these unknown parameters?

The probability density of the given data is the likelihood $\mathcal{L}$ with

$$\mathcal{L} = \prod_{i=1}^{N} \mathcal{N}(x_i; \mu, \sigma^2) \tag{7.1}$$

The assumptions of independence and normality imply that

$$\log \mathcal{L} = \sum_{i=1}^{N} \frac{(x_i - \mu)^2}{2\sigma^2} + \text{ a constant independent of } \mu \tag{7.2}$$

To maximize the likelihood we minimize

$$\chi^2 = \sum_{i=1}^{N} \frac{(x_i - \mu)^2}{2\sigma^2} \tag{7.3}$$

by requiring that $\partial\chi^2/\partial\mu = 0$. In terms of stochastic variables rather than their realizations this gives

$$\tilde{\mu} = \frac{1}{N}\sum \tilde{x}_i. \tag{7.4}$$

A similar argument, generalized so as to account for the dependence on variance, produces the maximum likelihood estimator for the variance

$$\tilde{\sigma}^2 = \frac{1}{N}\sum_{i=1}^{N}(\tilde{x}_i - \tilde{\mu})^2 \tag{7.5}$$

It is simple to show that $\tilde{\mu}$ as defined in Eq. (7.4) satisfies

$$\langle \tilde{\mu} \rangle = \mu. \tag{7.6}$$

An estimator with this property is called *unbiased*.

The estimator for the variance in Eq. (7.5 lacks this property. It's not difficult to show by using the independence of the $\tilde{x}_i$ that the following two estimators are unbiased

$$\tilde{\sigma}_0^2 = \frac{1}{N}\sum_{i=1}^{N}(\tilde{x}_i - \mu)^2 \tag{7.7}$$

$$\tilde{\sigma}_1^2 = \frac{1}{N-1}\sum_{i=1}^{N}(\tilde{x}_i - \tilde{\mu})^2 \tag{7.8}$$

Obviously, Eq. (7.7) is only of practical use if μ is known as would be the case if the system under investigation has, for instance, $\tilde{x}_i \leftrightarrow -\tilde{x}_i$ symmetry.

Although the estimator for the average as given in Eq. (7.4) seems obvious, it does not necessarily produce the most likely value of the average as the following example shows.

Suppose that the $\tilde{x}_i$ have a probability density function proportional to $e^{-|x-\mu|/\sigma}$. The maximum-likelihood estimator is obtained by minimizing

$$\chi = \sum_{i=1}^{N} \frac{|x_i - \mu|}{\sigma}. \tag{7.9}$$

This produces the median as the most likely estimate of μ. That is, if the x_i are sorted so that $x_1 \leq x_2, \cdots \leq x_N$ then

$$\mu = \begin{cases} x_{\frac{1}{2}(N+1)} & \text{if } N \text{ is odd,} \\ \frac{1}{2}(x_{\frac{1}{2}N} + x_{\frac{1}{2}N+1}) & \text{if } N \text{ is even.} \end{cases} \tag{7.10}$$

The derivative of χ with respect to μ is defined between the μ is the number of x_i to the left of μ minus the difference of the number of x_i on right.

Any choice between the middle two values will work; the halfway point is a matter of convention.

7.1 Parameter fitting

Study sections 15.0 through 15.4 of of Chapter 15 of *Numerical Recipes.*

7.1.1 Assignments

Assignment 18 Parameter fit: Gaussian noise

1. Define

$$f(x; a_0, \ldots, a_{m-1}) = \sum_{k=0}^{m-1} a_k x^k, \tag{7.11}$$

with $m = 4$ and $(a_0, a_1, a_2, a_3) = (1, 2, 3, 4)$

2. For $x_i = i/n$ with $i = 0, \ldots, n-1$ let

$$f_i = f(x_i; a_0, \ldots, a_{m-1}) + \xi_i, \tag{7.12}$$

where the ξ_i are random numbers sampled from $\mathcal{N}(\xi; \mu, \sigma^2)$, with $\mu = 0$ and $\sigma = 0.1$—see Eq. (6.11).

3. Reconstruct the parameters a_i using the method described in *Numerical Recipes* section 15.4 General Linear Least Squares.

4. Use *Numerical Recipes* Eq. (15.4.15) to obtain the standard errors, aka the uncertainties, of the parameter estimates.

5. Obtain the standard error of the estimate of $a_0 + a_1 + a_2 + a_3$. (See *Numerical Recipes* Eq. (15.6.6).)

Note that the "linear" used here refers to linearity of Eq. (7.11) in the parameters a_i and *not* in the function argument x.

Hints:

1. Table 31 illustrates some of the linear algebra needed for this problem: @ is the matrix multiplication and `numpy.linalg.inv` performs matrix inversion.

2. When using the least-squares method, one has more equations than unknowns. That is one has a set of equations of the form

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}, \tag{7.13}$$

where $\boldsymbol{A}$ is an $n \times m$ matrix with $n > m$; compare this with *Numerical Recipes* Fig. 15.4.1[46]. In that case, $\boldsymbol{x}$ is obtained from

$$\boldsymbol{A}^T \boldsymbol{A} \boldsymbol{x} = \boldsymbol{A}^T \boldsymbol{b}, \tag{7.14}$$

---

[46]W. H. Press et al. *15.4 General Linear Least Squares.* URL: http://phys.uri.edu/~nigh/NumRec/bookfpdf/f15-4.pdf.

by inverting the square $M \times M$ matrix $\boldsymbol{A}^T \boldsymbol{A}$, unless a singular value decomposition is used, which in difficult cases is more robust.

3. To check your progam you can omit the noise term in Eq. (7.12).

4. To create a NumPy matrix use `import numpy as np; A = np.zero([n,m])` and subsequently fill in the values of the matrix elements, as shown in more detail in Table 32.

5. In Numpy—imported as `np`—`print(np.random.normal(0,1,10))` prints an array of 10 standard normal $\mathcal{N}(0, 1)$ random numbers.

6. For item # 5 look at *Numerical Recipes* Eqs. (5.4.14) and (5.4.15) and in particular at the paragraph just below them. It refers to section 15.6.
The variance of the sum of the estimated coefficients in this case is not the sum of the variances, because the estimates are not independent.

Table 31: Solution of $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ for a 3×3 matrix random $\boldsymbol{A}$ and a random vector 3-component vector $\boldsymbol{b}$. The symbol `@` represents a matrix multiplication.

```
>>> import numpy as np
>>> A = np.random.random([3,3])        # initialize A
>>> b = np.random.random([3])          # initialize b
>>> x = np.linalg.inv(A) @ b           # evaluate A⁻¹b
>>> r = A @ x - b                      # compute residue vector r
>>> print(np.sqrt(np.sum(r * r)))      # print norm of r
3.1499827100292216e-15
```

Table 32: Alternative ways of creating matrices

```
>>> import numpy as np
>>> A = np.zeros([3,4])
>>> for i in range(0,3):
...    for j in range(0,4):
...       A[i,j] = i+j
...
>>> print(A)
[[0. 1. 2. 3.]
 [1. 2. 3. 4.]
 [2. 3. 4. 5.]]
>>> B = np.array([[i + j for j in range(0,4)] for i in range(0,3)])
>>> print(B)
[[0. 1. 2. 3.]
 [1. 2. 3. 4.]
 [2. 3. 4. 5.]]
```

In case of normally distributed stochastic variables, the standard error Δa of an estimate a is defined such that the actual value has 68% probability of falling in the range $a - \Delta a, a + \Delta a$. This is a consequence of the fact that

$$\frac{1}{\sqrt{2\pi\sigma^2}} \int_{-\sigma}^{\sigma} e^{-x^2/(2\sigma^2)} \, dx = \text{erf}\left(\frac{\sqrt{2}}{2}\right) \approx 0.68 \tag{7.15}$$

The error function, erf, can be calculated in Python by
    `from math import erf; from numpy import *; erf(sqrt(2)/2)`
The fact that $\text{erf}(\sigma) = 68\%$, $\text{erf}(2\sigma) = 95\%$, and $\text{erf}(3\sigma) = 99.7\%$ with $\sigma = \sqrt{2}/2$ is known as the 68-95-99.7 rule, which tells you how outliers are distributed in the "normal" case. This is illustrated in Table 33.

```
>>> import numpy as np
>>> from math import erf
>>> p1 = erf(np.sqrt(2)/2)
>>> print(p1)
0.6826894921370861
>>> p2 = erf(2*np.sqrt(2)/2)
>>> print(p2)
0.9544997361036416
>>> p3 = erf(3*np.sqrt(2)/2)
>>> print(p3)
0.9973002039367398
>>> n = 100000
>>> xi = np.random.normal(0, 1, n)
>>> xi = sorted(np.abs(xi))# note the absolute value
>>> xi[p1 * n]
1.0043476888701346
>>> xi[p2 * n]
2.0060203911142858
>>> xi[p3 * n]
3.0164245527439997 # the deviations from 1, 2 and 3 increase; why?
```

Assignment 19 Parameter fit: verify parameter error estimate

This is a continuation of assignment 18 on page 46.

1. Repeat the procedure of items 2 through 3 of assignment 18 a couple of hundred times. Save and sort the absolute values of the differences of the fitted parameters and their exact values. For each parameter and the sum of all parameters find a value of the deviation so that 68% are smaller or equal to this value. Compare this result with the standard error obtained in item 4 and 5 of assignment 18.

7.2   Random numbers beyond $U(0,1)$

In practice, often other distributions are required besides the the $U(0,1)$ distribution. Suppose we want to generate a probability density function $\rho_{\tilde{x}}(x)$, that is, we want to construct a stochastic variable $\tilde{x}$ so that $P(x < \tilde{x} \le x + dx) = \rho_{\tilde{x}}(x)\,dx$, where by $P(A)$ is the probability that event A occurs. There are several methods to accomplish this. Of the methods to be discussed, the transformation and rejection methods can be used to generate independent numbers sampled from the desired distribution. We won't discuss the Metropolis-Hastings algorithm, which is more powerful, but generates correlated random numbers. That means that more data are required to obtain a given statistical accuracy.

7.2.1   Transformation method

Consider transformation of a n-dimensional coordinate system that maps the coordinates $x_1, \ldots, x_n$ to new coordinates $y_j = y_j(x_1, \ldots, x_n)$ with $j = 1, \ldots, n$. The transformation maps the infinitesimal hypercube $d\Gamma$ determined by the points $x_1, \ldots, x_n$ and $x_1 + dx_1, \ldots, x_n + dx_n$ with content (n-dimensional volume) $dx_1 \ldots dx_n$ onto a parallelotope $d\Gamma'$ with content $J\,dy_1 \ldots dy_n$, where $J = |\partial(x_1, \ldots, x_n)/\partial(y_1, \ldots, y_n)|$, the absolute value of the determinant of the $n \times n$ Jacobian matrix with elements $\partial x_i/\partial y_j$.

The transformation maps the n-dimensional stochastic variable $\tilde{\boldsymbol{X}} = (\tilde{x}_1, \ldots, \tilde{x}_n)$ onto $\tilde{\boldsymbol{Y}}$ with components $\tilde{y}_j = \tilde{y}_j(\tilde{x}_1, \ldots, \tilde{y}_n)$. Conservation of probability implies that

$$P(\tilde{\boldsymbol{X}} \in d\Gamma) = P(\tilde{\boldsymbol{Y}} \in d\Gamma'), \tag{7.16}$$

so that accordingly $\rho_{\tilde{\boldsymbol{X}}}(x_1, \ldots, x_n)$ is transformed into a new probability density function τ with

$$\tau(y_1, \ldots, y_n) = J\rho(x_1, \ldots, x_n) \tag{7.17}$$

An example of this is the Box-Muller transformation which transforms a pair of independent $U(0,1)$ variates (X_1, X_2) into a pair of independent standard normal variates (Y_1, Y_2) by means of the transformation

$$y_1 = \sqrt{-2\log x_1}\cos(2\pi x_2), \tag{7.18}$$

$$y_2 = \sqrt{-2\log x_1}\sin(2\pi x_2), \tag{7.19}$$

as can be verified as follows

$$dy_1 dy_2 = \left|\frac{\partial(y_1, y_2)}{\partial(x_1, x_2)}\right| dx_1 dx_2 = \frac{2\pi}{x_1} dx_1 dx_2 = 2\pi e^{\frac{1}{2}(y_1^2 + y_2^2)} dx_1 dx_2. \tag{7.20}$$

See also Marsaglia-Tsang's Monty Python method for the generation of normal and other variates.[47]

Another example of the tranformation method is often used to construct one-dimensional probability density functions. Suppose that the desired probability density function is $f(x)$ with cumulative distribution function $F(x) = \int_{-\infty}^x f(y)\,dy$. As illustrated in Fig. 6, if $\tilde{\xi}$ is a $U(0,1)$ variate, $F^{-1}(\xi)$ has the desired probability density function f, because the derivative of F is f, as required by Eq. (7.17) with $\rho(x) = 1$ for $x \in (0,1)$.

Figure 6: The cumulative distribution function $F(x)$ vs. x (solid curve); the interval along the vertical axis that contains ξ is mapped onto the interval along the horizontal axis that contains $F^{-1}(\xi)$. Note that the ratio of the intervals satisfies Eq. (7.17).
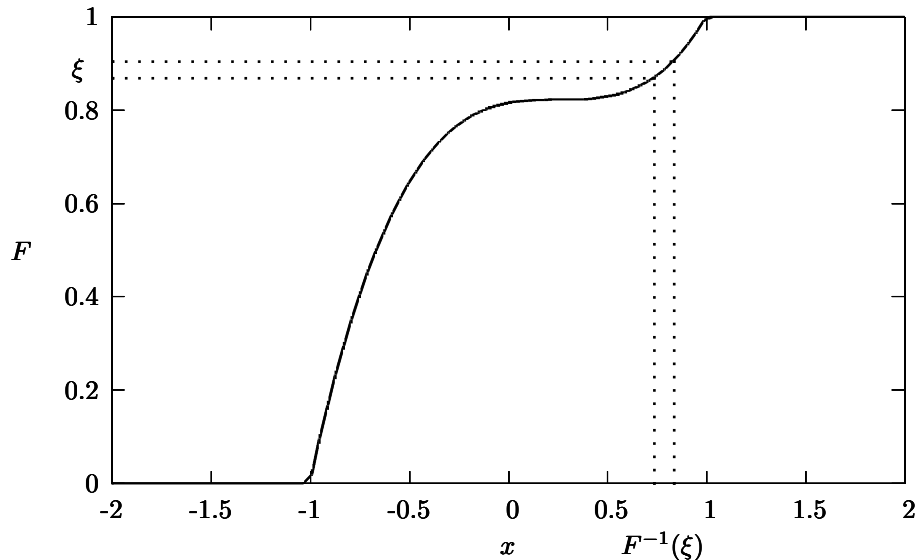
The transformation method, illustrated in Fig. 6, can also be used to sample from a discrete probability density function $P(k) = p_k$ with $p_i > 0$ and $\sum_{k=1}^n p_k = 1$. In this case, the cumulative distribution function F is the step function given by $F(x) = P(k \le x)$.

7.2.2 Rejection method

Suppose you want to sample from the probability density function ρ. Find a probability density function σ with a domain (the set of arguments for which σ is defined) containing the domain of ρ. Suppose that σ can be sampled directly. If necessary extend the domain of ρ to that of σ and choose $\rho(x) = 0$ in the extended part of the domain. Find a constant C such that $C\sigma(x) \ge \rho(x)$ for all x. Algorithm 3 generates variates ξ with probability density function ρ. Step 2 is implemented by drawing a $U(0,1)$ random number. If that number is smaller than $p(\xi)$ output ξ. Typically, in step 1 one uses the $U(0,1)$ distribution, but it could be any convenient choice. For the constant C in step 2 the most efficient value is the smallest value of C so that $p(\xi)$ in Eq. (7.21) is guaranteed to satisfy $p(\xi) \le 1$ for all ξ.

---

[47]G. Marsaglia and W. W. Tsang. "The Monty Python method for generating random variables". In: *ACM Trans. Math. Softw.* 24 (3 Sept. 1998), pp. 341–350. ISSN: 0098-3500. DOI: http://doi.acm.org/10.1145/292395.292453. URL: http://doi.acm.org/10.1145/292395.292453.

Algorithm 3 The rejection method

Given C s.t. $C\sigma(x) \geq \rho(x)$ for all x

1. Directly sample ξ from probability density function σ;

2. Output ξ with probability

$$p(\xi) = \frac{\rho(\xi)}{C\sigma(\xi)}; \tag{7.21}$$

3. Repeat from step 1 until step 2 produces output.

This method can also be used in higher dimensions, but very quickly it becomes very difficult to find an efficient constant C. In other words, if $p(\xi) \ll 1$ most of the time, the method becomes very inefficient because of the excessive number of repetitions of steps 1 and 2 in algorithm 3.

7.2.3 Assignments

Assignment 20 Sampling $2x$ on $(0,1)$ by rejection and transformation

Consider the probability density function $2x$ with $x \in (0,1)$. Verify the validity of your program in each of the following cases by plotting a histogram. (See Table 30 for how to plot a histogram.)

1. Use algorithm 3 to generate random numbers that sample this probability density function.
2. The sum of two independent $U(0,1)$ stochastic variables has a triangular distribution. Use a variation of the transformation method to obtain the desired probability density function from this sum.
3. Use the transformation method, illustrated in Fig. 6, to sample the probability density function.

Assignment 21 Linear probability density function: transformation and rejection
Consider an arbitrary linear probability density function defined on a finite interval. As in assignment 20 use a histogram for program verification.

1. Write a routine to sample this distribution using the transformation method;
2. Write a routine to do the same using the rejection method.

Hints:

1. As a first step assume that the finite interval in which the probability density function is concentrated is the interval $(0,1)$. Then transform the result to an arbitrary finite interval.
2. The algebra can be simplified by assuming that the probability density function has the form

$$\rho(x) \propto a + bx, \tag{7.22}$$

This function must be non-negative on the $[0,1]$ interval. Then impose the condition that

$$\int_0^1 \rho(x)\,dx = 1. \tag{7.23}$$

This gives something of the form

$$\rho(x) = a' + b'x, \tag{7.24}$$

where a' and b' are no longer independent; only the ratio of a and b matters.
3. The cumulative distribution function will be a quadratic function. Inverting it, as is necessary for the transformation method will produce two solutions. Only one works: the one corresponding to a monotonically increasing cumulative distribution function.

Assignment 22 Rejection in D dimensions
Consider sampling points uniformly from a d dimensional unit hyper-sphere by the rejection method. For this purpose choose the probability density function σ to be the uniform distribution defined on the circumscribing hyper-cube.

1. How would you define the efficiency of this method?

2. Write a program to compute the efficiency of this method as a function of dimension D.

Assignment 23 Discrete probability density function

An experiment can result in outcome k with probability $p_k > 0$ with $k = 1, 2, ..., n$.

1. Write a routine that starting from $U(0,1)$ random numbers samples the discrete distribution defined by the probabilities p_k concentrated on these points $k = 1, 2, ..., n$.

2. Test your routine by choosing n random points in the interval $(0,1)$. Scale the numbers so that they add up to one and use them to define the discrete probability distribution.

3. Make a histogram for $N \gg n$ random numbers generated by your routine and check visually that the histogram agrees with this probability density function.

4. Use the χ^2 test, as described in hint 3 below for the statistical verification of your routine.

Hints: 1. Use the bisection method to find in $\mathcal{O}(\log_2 n)$ steps the value of $k = 1, \ldots, n$ that corresponds to the random number $\xi \in (0,1)$. This is illustrated in Fig. 7.

Figure 7: A discrete analog of the cumulative distribution function of Fig. 6. There are four possible outcomes: 1, 2, 3 and 4. If ξ, sampled from $U(0,1)$, lands in the interval $(0.0, 0.4)$ the result is 1. The interval $(0.4, 0.5)$ results in 2 and so on.
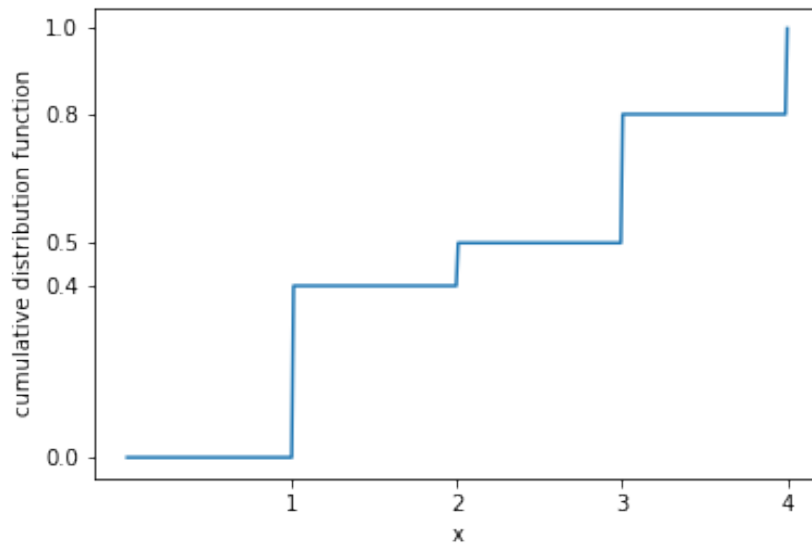
2. For N trials, n_k, the number of times that your routine produces the integer k, is given by the binomial distribution

$$B(n_k; N, p_k) = \binom{N}{n_k} p_k^{n_k} (1 - p_k)^{N - n_k}. \tag{7.25}$$

Average and variance are given by

$$\langle n_k \rangle = p_k N, \tag{7.26}$$
$$\text{var } n_k = p_k(1 - p_k)N. \tag{7.27}$$

3. Use the statistic

$$\tilde{\chi}^2 = \sum_{k=1}^{n} \frac{(\tilde{n}_k - p_k N)^2}{N p_k (1 - p_k)} \tag{7.28}$$

to assess the validity of your routine. For large N the χ^2 will have a χ^2-distribution with $n-1$ degrees of freedom: you loose one degree of freedom because of the constraint $\sum_k n_k = N$.

4. The χ^2 approximation is only valid when each term in Eq. (7.28) behaves as the square of a Gaussian variable, that is, if $B(n_k : N, p_k) \approx \mathcal{N}[Np_k, Np_k(1 - p_k)]$, for all $k = 1, \dots, n$. In practice, this condition is almost always satisfied if $Np_k - 3\sqrt{Np_k(1 - p_K)} > 0$ and $Np_k + 3\sqrt{Np_k(1 - p_K)} < N$, that is, if $N > 9\max[(1 - p)/p, p/(1 - p)]$. As before, $\mathcal{N}(\mu, \sigma^2)$ denotes the normal distribution with mean μ and variance σ^2. (The factor 3 comes from the 68-95-99.7 rule on page 47.)

8 Linear algebra

Classical texts on this subject are Wilkinson,[48] Gantmacher[49] and Golub and Van Loan.[50]

8.1 Kronecker and Dirac δ-functions

The Kronecker δ_{kl}, with k and l integers, is defined by

$$\delta_{kl} = \begin{cases} 1 \text{ if } k = l, \\ 0 \text{ if } k \neq l. \end{cases} \tag{8.1}$$

The Dirac δ-function plays an important role in what follows. It satisfies the following relations:

$$\int_{-\infty}^{\infty} \delta(x - a) f(x) \, dx = f(a), \tag{8.2}$$

$$\int_{-\infty}^{\infty} \delta^{(1)}(x - a) f(x) \, dx = -f^{(1)}(a), \tag{8.3}$$

$$\int_{-\infty}^{\infty} \delta^{(n)}(x - a) f(x) \, dx = (-1)^n f^{(n)}(a). \tag{8.4}$$

The superscripts in parentheses indicate derivatives. The last two equations follow from the first by using integration by parts. Alternatively, define

$$g_n(x) = \frac{1}{\sqrt{2\pi/n^2}} e^{-n^2 2x^2/2}. \tag{8.5}$$

The $g_n(x)$ converge to a Dirac δ-function as $n \to \infty$. This is illustrated in Fig. 8, which also shows how the derivatives $g_n'(x)$ behave. This shows, at least qualitatively, that the derivative of a δ-function can be considered as two closely spaced δ-functions of opposite signs, which produces the negative derivative of f in Eq. (8.3).

Figure 8: Elements of the δ-sequence $g_n(x)$, as defined in Eq. (8.5) and their derivatives.
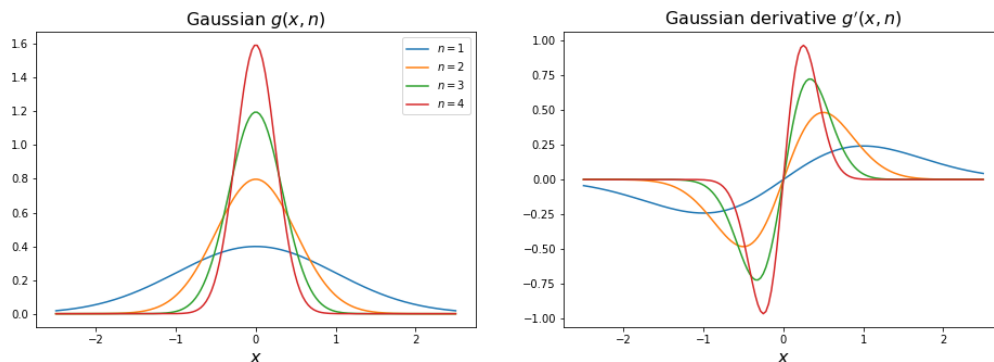
Also recall that

$$\delta(ax) = \frac{1}{|a|} \delta(x), \tag{8.6}$$

[48] J. H. Wilkinson. *The Algebraic Eigenvalue Problem.* Oxford University Press, 1988.
[49] F. R. Gantmacher. *The Theory of Matrices, Vol. I and II.* American Mathematical Society, 1998.
[50] G. H. Golub and C. F. Van Loan. *Matrix Computations, 4th edition.* Johns Hopkins University Press, 2013.

which follows from introducing a new variable $x' = ax$. The absolute value comes from the fact that an $a < 0$ reverses the integration limits, which upon undoing gives a minus sign:

$$\int_\alpha^\beta f(x)\,dx = -\int_\beta^\alpha f(x)\,dx. \tag{8.7}$$

8.2 Orthonormality and completeness

Consider a column vector

$$\boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}, \tag{8.8}$$

where $n \in \mathbb{N}$ and $x_i \in \mathbb{C}$. Suppose that

$$\boldsymbol{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} \tag{8.9}$$

is another column vector. The inner product $(\boldsymbol{x}, \boldsymbol{y})$ is defined by

$$(\boldsymbol{x}, \boldsymbol{y}) = \sum_{i=1}^n x_i^* y_i. \tag{8.10}$$

The inner product has the following properties for any $\boldsymbol{x}$, $\boldsymbol{y}$, $\boldsymbol{z} \in \mathbb{C}^n$ and $\alpha, \beta \in \mathbb{C}$

1.
$$(\boldsymbol{x}, \boldsymbol{y}) = (\boldsymbol{y}, \boldsymbol{x})^*; \tag{8.11}$$

2.
$$(\boldsymbol{x}, \alpha\boldsymbol{y} + \beta\boldsymbol{z}) = \alpha(\boldsymbol{x}, \boldsymbol{y}) + \beta(\boldsymbol{x}, \boldsymbol{z}); \text{ and} \tag{8.12}$$

3.
$$(\alpha\boldsymbol{x} + \beta\boldsymbol{y}, \boldsymbol{z}) = \alpha^*(\boldsymbol{x}, \boldsymbol{y}) + \beta^*(\boldsymbol{x}, \boldsymbol{z}). \tag{8.13}$$

Property (8.12) is called linearity; property (8.13) is called antilinearity or sesquilinearity.

It is convenient to write the inner product as a matrix product

$$(\boldsymbol{x}, \boldsymbol{y}) = (x_1^*, x_2^*, \ldots, x_n^*) \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} = \boldsymbol{x}^\dagger \boldsymbol{y}. \tag{8.14}$$

The dagger indicates the Hermitian adjoint—the complex conjugate transpose, i.e.,

$$\boldsymbol{x}^\dagger = (x_1^*, x_2^*, \ldots, x_n^*). \tag{8.15}$$

Let $\boldsymbol{A}$ be a $p \times q$ matrix

$$\boldsymbol{A} = \begin{pmatrix} A_{11} & A_{12} & \ldots & A_{1q} \\ A_{21} & A_{22} & \ldots & A_{2q} \\ \vdots & \vdots & \vdots & \vdots \\ A_{p1} & A_{p2} & \ldots & A_{pq} \end{pmatrix} \tag{8.16}$$

In general, we denote by $\boldsymbol{A}^T$ and $\boldsymbol{A}^\dagger$ respectively the transpose and Hermitian adjoint of $\boldsymbol{A}$:

$$\boldsymbol{A} = (A_{ij})_{i=1,j=1}^{p,q} \tag{8.17}$$

$$\boldsymbol{A}^T = (A_{ji})_{j=1,i=1}^{q,p} \tag{8.18}$$

$$\boldsymbol{A}^\dagger = \left(A_{ji}^*\right)_{j=1,i=1}^{q,p} \tag{8.19}$$

The matrix product of a $p \times q$ matrix $\boldsymbol{A}$ and a $q \times r$ matrix $\boldsymbol{B}$ is a $p \times r$ matrix $\boldsymbol{C}$ with elements

$$\boldsymbol{C} = \boldsymbol{AB}, \tag{8.20}$$

$$c_{ij} = \sum_{k=1}^{q} a_{ik} b_{kj}, \tag{8.21}$$

with $i = 1, \ldots, p$ and $j = 1, \ldots, r$. The matrix product in Eq. (8.20) is to be distinguished from the element-wise Hadamard product $\boldsymbol{A} \circ \boldsymbol{B}$ defined in Eq. (1.3). The latter is commutative, that is $\boldsymbol{A} \circ \boldsymbol{B} = \boldsymbol{B} \circ \boldsymbol{A}$, while the former only under special circumstances. Note that

$$(\boldsymbol{AB})^{\dagger} = \boldsymbol{B}^{\dagger} \boldsymbol{A}^{\dagger}, \tag{8.22}$$

$$(\boldsymbol{AB})^{T} = \boldsymbol{B}^{T} \boldsymbol{A}^{T}. \tag{8.23}$$

A matrix with complex elements is called Hermitian if $\boldsymbol{A}^{\dagger} = \boldsymbol{A}$; if the latter relation holds and if the matrix has only real elements, it is called symmetric so that $\boldsymbol{A}^{T} = \boldsymbol{A}$.

Clearly, the inner product $\boldsymbol{x}^{\dagger} \boldsymbol{y}$ is a matrix product of a $1 \times n$ and an $n \times 1$ matrix which we can interpret as 1×1 matrix, i.e., an object with the same algebraic properties a complex number.

A matrix $\boldsymbol{A}$ has a Hermitian adjoint $\boldsymbol{A}^{\dagger}$ that satisfies

$$(\boldsymbol{x}, \boldsymbol{Ay}) = (\boldsymbol{A}^{\dagger} \boldsymbol{x}, \boldsymbol{y}), \tag{8.24}$$

or equivalently

$$\boldsymbol{x}^{\dagger} \boldsymbol{Ay} = \left(\boldsymbol{A}^{\dagger} \boldsymbol{x} \right)^{\dagger} \boldsymbol{y}. \tag{8.25}$$

For the special case of a Hermitian matrix this implies that

$$(\boldsymbol{x}, \boldsymbol{Ay})^{*} = (\boldsymbol{Ay}, \boldsymbol{x}) = (\boldsymbol{y}, \boldsymbol{Ax}). \tag{8.26}$$

in other words:

$$(\boldsymbol{x}^{\dagger} \boldsymbol{Ay})^{*} = (\boldsymbol{Ay})^{\dagger} \boldsymbol{x} = \boldsymbol{y}^{\dagger} \boldsymbol{Ax}. \tag{8.27}$$

The matrix product of the $p \times 1$ matrix $\boldsymbol{x}$, that is a column p-vector, and the $1 \times q$ matrix $\boldsymbol{y}^{\dagger}$, a row q-vector, produces a $p \times q$ matrix, aka a dyadic, $\boldsymbol{D}$ with elements $D_{ij} = x_i y_j$:

$$\boldsymbol{D} = (x_i y_j)_{i=1, j=1}^{p, q}, \tag{8.28}$$

or more explicitly

$$\boldsymbol{D} = \boldsymbol{x} \boldsymbol{y}^{T} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_p \end{pmatrix} (y_1, y_2, \ldots, y_q) = \begin{pmatrix} x_1 y_1 & x_1 y_2 & \ldots & x_1 y_q \\ x_2 y_1 & x_2 y_2 & \ldots & x_2 y_q \\ \vdots & \vdots & \vdots & \vdots \\ x_p y_1 & x_p y_2 & \ldots & x_p y_q \end{pmatrix}. \tag{8.29}$$

Dyadic matrices can be constructed in NumPy as illustrated in Table 34. As you see, the result of the matrix product `a @ b` is not a number and therefore it does not represent an inner product. The result is an array with one element which in turn is an array. The latter has one element, which is a number, the inner product.

Notice that the shape of `z` is different from the shapes of both `a` and `b`. Transposition (`.T`) works on `a` and `b` as expected, but it does nothing on `z`.

Let $\boldsymbol{U}$ be an arbitrary, unitary $n \times n$ matrix, that is a matrix that satisfies

$$\boldsymbol{U} \boldsymbol{U}^{\dagger} = \boldsymbol{U}^{\dagger} \boldsymbol{U} = \mathbb{1}, \tag{8.30}$$

where $\mathbb{1}$ is the $n \times n$ identity matrix $\mathbb{1} \equiv \mathrm{diag}(1, 1, \ldots, 1)$.

The inner product is invariant under unitary transformation:

$$(\boldsymbol{Ux}, \boldsymbol{Uy}) = (\boldsymbol{Ux})^{\dagger} (\boldsymbol{Uy}) = \boldsymbol{x}^{\dagger} (\boldsymbol{U}^{\dagger} \boldsymbol{U}) \boldsymbol{y} = (\boldsymbol{x}, \boldsymbol{y}). \tag{8.31}$$

Conversely, as is easy to verify, a matrix that leaves the inner products of all pairs of vectors invariant must satisfy Eq. (8.31).

Table 34: Constructing dyadics in NumPy

```
>>> z = np.array([1,2,3])
>>> a = np.reshape(a,[1,3])
>>> a = np.reshape(z,[1,3])
>>> b = np.reshape(z,[3,1])
>>> a @ b # @ denotes the matrix product, short for np.matmul(a,b)
array([[14]])
>>> b @ a
array([[1, 2, 3],
       [2, 4, 6],
       [3, 6, 9]])
>>> print(a)
[[1 2 3]]
>>> print(b)
[[1]
 [2]
 [3]]
>>> print(a.T)
[[1]
 [2]
 [3]]
>>> print(b.T)
[[1 2 3]]
>>> z.shape
(3,)
>>> a.shape
(1, 3)
>>> b.shape
(3, 1)
```

Let $\boldsymbol{v}$ be a normalized column n-vector, i.e., a matrix $\boldsymbol{P}$ that satisfies

$$\boldsymbol{PP} = \boldsymbol{P}^2 = \boldsymbol{P} \tag{8.32}$$

is called a projection matrix.

Let $\boldsymbol{v}$ be a normalized column n-vector, i.e.,

$$\|\boldsymbol{v}\| \equiv \sqrt{(\boldsymbol{v}, \boldsymbol{v})} = \sqrt{\boldsymbol{v}^\dagger \boldsymbol{v}} = \sqrt{\sum_{i=1}^{n} v_i^* v_i} = 1. \tag{8.33}$$

It follows immediately from the associatively of matrix multiplication and Eq. (8.33) that the dyadic $\boldsymbol{P_v} = \boldsymbol{vv}^\dagger$ is a projection matrix

$$(\boldsymbol{vv}^\dagger)(\boldsymbol{vv}^\dagger) = \boldsymbol{v}(\boldsymbol{v}^\dagger\boldsymbol{v})\boldsymbol{v}^\dagger = \boldsymbol{vv}^\dagger. \tag{8.34}$$

With the projection operator $\boldsymbol{P_v}$ one can decompose any vector $\boldsymbol{v}$ into components $\boldsymbol{v}_\parallel$ and $\boldsymbol{v}_\perp$ respectively parallel and perpendicular to $\boldsymbol{v}$

$$\boldsymbol{x} = \boldsymbol{x}_\parallel + \boldsymbol{x}_\perp, \tag{8.35}$$

with

$$\boldsymbol{x}_\parallel = \boldsymbol{P_v}\boldsymbol{x} = \boldsymbol{vv}^\dagger\boldsymbol{x} = (\boldsymbol{v}, \boldsymbol{x})\boldsymbol{v}, \tag{8.36}$$

$$\boldsymbol{x}_\perp = (\mathbb{1} - \boldsymbol{vv}^\dagger)\boldsymbol{x} = \boldsymbol{x} - \boldsymbol{x}_\parallel. \tag{8.37}$$

It follows from Eqs. (8.11) through (8.13) and the fact that $\|\boldsymbol{v}\| = 1$ that $\boldsymbol{v}_\perp$ and $\boldsymbol{v}_\parallel$ are indeed orthogonal:

$$(\boldsymbol{x}_\perp, \boldsymbol{x}_\parallel) = \big(\boldsymbol{x} - (\boldsymbol{v}, \boldsymbol{x})\boldsymbol{v}, (\boldsymbol{v}, \boldsymbol{x})\boldsymbol{v}\big) = 0. \tag{8.38}$$

8.3 Eigensystems of Hermitian matrices

The eigenvalues of a Hermitian $\boldsymbol{H}$ matrix are real. This can be seen as follows. Let $\boldsymbol{v}$ have eigenvalue λ, i.e.,

$$\boldsymbol{H}\boldsymbol{v} = \lambda\boldsymbol{v}. \tag{8.39}$$

Multiply this equation by $\boldsymbol{v}^\dagger$ from the left and divide through by $\boldsymbol{v}^\dagger\boldsymbol{v}$. This give

$$\lambda = \frac{\boldsymbol{v}^\dagger\boldsymbol{H}\boldsymbol{v}}{\boldsymbol{v}^\dagger\boldsymbol{v}}. \tag{8.40}$$

By virtue of Eq. (8.11) and the Hermiticity of $\boldsymbol{H}$, Eq. (8.27) the right-hand side of this equation is real.

Eigenvectors of $\boldsymbol{H}$ with *different* eigenvalues are orthogonal. Let

$$\boldsymbol{H}\boldsymbol{w} = \mu\boldsymbol{w}. \tag{8.41}$$

Take the Hermitian adjoint of Eq. (8.39) to obtain

$$\boldsymbol{v}^\dagger\boldsymbol{H} = \lambda\boldsymbol{v}^\dagger. \tag{8.42}$$

Multiply this equation by $\boldsymbol{w}^\dagger$ from the left and Eq. (8.41) by $\boldsymbol{v}$ from the right and subtract the two results. This gives

$$(\lambda - \mu)\boldsymbol{w}^\dagger\boldsymbol{v} = 0. \tag{8.43}$$

Because $\lambda \neq \mu$, the second factor in this equation vanishes, which means that the inner product of $\boldsymbol{v}$ and $\boldsymbol{w}$ vanishes. In other words, these two eigenvectors are orthogonal.

The eigenvalues of an $n \times n$ matrix $\boldsymbol{A}$ are the roots of the secular equation (aka the characteristic equation)

$$\det(z\mathbb{1} - \boldsymbol{A}) = 0. \tag{8.44}$$

If all n eigenvalues λ_i with $i = 1, \ldots, n$ are distinct, the corresponding normalized eigenvectors form an orthonormal basis of the linear space of all n-component column vectors with components in $\mathbb{C}$. That is any n-component vector can be written uniquely as a sum of these basis vectors. In this case of distinct eigenvalues, the normalized eigenvectors are uniquely determined up to phase factors, complex numbers that lie on the unit circle in the complex plane.

For degenerate eigenvalues—eigenvalues common to more than one eigenvector—the eigenvectors can be constructed so as to be orthonormal, but because an arbitrary linear combination of eigenvectors with the same eigenvalue is an eigenvector of the same eigenvalue, the choice of the basis vectors is non-unique beyond the usual arbitrary phase factors.

The construction of orthonormal eigenvectors from a set of eigenvectors of unit length $\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_l$ with the same eigenvalue can be constructed by means of the Gram-Schmidt orthonormalization as given in Algorithm 4. The algorithm will fail if the $\boldsymbol{u}_i$ are linearly dependent. Numerically, cancellations tend to produce increasing loss of orthoginality of the resulting eigenvectors.[51]

Algorithm 4 Gram-Schmidt orthonormalization based on Eqs. (8.36) and (8.37)

Input: $\boldsymbol{u}_i$ n-component vectors of unit length. Output: $\boldsymbol{u}_i'$ a set of orthonormal vectors; $i = 1, \ldots, m$.

Step 1: $\boldsymbol{u}_1' = \boldsymbol{u}_1$

Step 2: $\boldsymbol{u}_2'$ is the normalized component of $\boldsymbol{u}_2$ perpendicular to $\boldsymbol{u}_1$;

Step 3: $\boldsymbol{u}_3'$ is the normalized component of $\boldsymbol{u}_3$ perpendicular to $\boldsymbol{u}_1'$ and $\boldsymbol{u}_2'$;

$\ldots$

Step m: $\boldsymbol{u}_m'$ is the normalized component of $\boldsymbol{u}_m$ perpendicular to $\boldsymbol{u}_1', \ldots, \boldsymbol{u}_{m-1}'$.

Let $\boldsymbol{x}_i$ with $i = 1, \ldots, p \leq n$ a set of n-component vectors. By definition $\mathrm{span}(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_p)$ is the set of all linear combinations of the $\boldsymbol{x}_i$:

$$\mathrm{span}(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_p) = \left\{ \sum_{i=1}^{p} \alpha_i \boldsymbol{x}_i | \alpha_1, \ldots, \alpha_p \in \mathbb{C} \right\}. \tag{8.45}$$

Let $\boldsymbol{H}$ be a Hermitian matrix with eigenvalues and normalized eigenvectors

$$\left. \begin{array}{l} \boldsymbol{H}\boldsymbol{u}_i = \lambda_i \boldsymbol{u}_i \\ (\boldsymbol{u}_i, \boldsymbol{u}_j) = \boldsymbol{u}_i^\dagger \boldsymbol{u}_j = \delta_{ij} \end{array} \right\} \text{ with } i, j = 1, 2, \ldots, n. \tag{8.46}$$

The fact that the $\boldsymbol{u}_i$ form a basis implies that every column vector $\boldsymbol{x}$ can be expanded as

$$\boldsymbol{x} = \sum_{j=1}^{n} \alpha_j \boldsymbol{u}_j. \tag{8.47}$$

Multiply this equation from the left by $\boldsymbol{u}_i$ and use the othonormality of the $\boldsymbol{u}_i$ to find that for $i = 1, 2, \ldots, n$

$$\alpha_i = \boldsymbol{u}_i^\dagger \boldsymbol{x} \tag{8.48}$$

Now substitute this equation into Eq. (8.47) to find that

$$\boldsymbol{x} = \sum_{i=1}^{n} \boldsymbol{u}_i (\boldsymbol{u}_i^\dagger \boldsymbol{x}) \tag{8.49}$$

$$= \sum_{i=1}^{n} (\boldsymbol{u}_i \boldsymbol{u}_i^\dagger) \boldsymbol{x} \tag{8.50}$$

Recalling the discussion after Eq. (8.32), we define the projection matrices $\boldsymbol{P}_i$ that project onto the eigenvectors $\boldsymbol{u}_i$ as

$$\boldsymbol{P}_i = \boldsymbol{u}_i \boldsymbol{u}_i^\dagger. \tag{8.51}$$

Because Eq. (8.50) holds for any $\boldsymbol{x}$ we find

$$\sum_{i=1}^{n} \boldsymbol{u}_i \boldsymbol{u}_i^\dagger = \sum_{i=1}^{n} \boldsymbol{P}_i = \mathbb{1}. \tag{8.52}$$

---

[51] For a modified Gram-Schmidt algorithm that addresses this problem see Golub and Van Loan, *Matrix Computations, 4th edition*.

This relation is known as the completeness or the closure relation; the set matrices $\boldsymbol{P}_i$ form a what is called a *resolution of the identity.*

Consider the action of the Hermitian matrix $\boldsymbol{H}$ with eigenvalues λ_i and corresponding eigenvectors $\boldsymbol{u}_i$ on an arbitrary column vector $\boldsymbol{x}$

$$\boldsymbol{H}\boldsymbol{x} \;=\; \boldsymbol{H}\sum_{i=1}^{n}\boldsymbol{u}_i\boldsymbol{u}_i^{\dagger}\boldsymbol{x} \tag{8.53}$$

$$=\; \sum_{i=1}^{n}\boldsymbol{H}\boldsymbol{u}_i\boldsymbol{u}_i^{\dagger}\boldsymbol{x} \tag{8.54}$$

$$=\; \sum_{i=1}^{n}\lambda_i\boldsymbol{u}_i\boldsymbol{u}_i^{\dagger}\boldsymbol{x}. \tag{8.55}$$

Again, this is true for all vectors $\boldsymbol{x}$ and as a consequence we find the so-called spectral decomposition

$$\boldsymbol{H}=\sum_{i=1}^{n}\boldsymbol{u}_i\lambda_i\boldsymbol{u}_i^{\dagger} \tag{8.56}$$

In the standard basis consisting of unit vectors $\boldsymbol{e}_i$ with

$$\boldsymbol{e}_i = (e_{ji})_{j=1}^{n} = (\delta_{ij})_{j=1}^{n} \tag{8.57}$$

Eq. (8.56) takes the form or the following expression for the matrix element H_{kl} of $\boldsymbol{H}$

$$H_{kl}=\sum_{i=1}^{n}u_{ki}\lambda_i u_{il}^{\dagger}=\sum_{i=1}^{n}u_{ki}\lambda_i u_{li}^{*}. \tag{8.58}$$

In matrix form this looks like

$$\boldsymbol{H}=\boldsymbol{U}\boldsymbol{D}\boldsymbol{U}^{\dagger}, \tag{8.59}$$

where $\boldsymbol{U}$ is a unitary matrix—$\boldsymbol{U}\boldsymbol{U}^{\dagger}=\mathbb{1}$— the columns of which are the normalized eigenvectors of $\boldsymbol{H}$. The matrix $\boldsymbol{D}$ is the diagonal matrix with the corresponding eigenvalues along the diagonal

$$D_{ij}=\lambda_i\delta_{ij}. \tag{8.60}$$

Note that it is conceptually simple to program Eq. (8.58) as written, that is with two matrix multiplications, but it is not computationally efficient. A computationally more efficient way of doing this results from the realization that multiplication of a matrix by a diagonal matrix multiplies each *column* by the corresponding diagonal element if the diagonal matrix is on the right. It does the same with the *rows* if the diagonal matrix is on the left. This is illustrated in Table 35; also see Tables 8 and 39.

Eq. (8.59) means that the matrix $\boldsymbol{U}$ diagonalizes the matrix $\boldsymbol{H}$:

$$\boldsymbol{D}=\mathrm{diag}(\lambda_1,\lambda_2,\ldots,\lambda_n)=\boldsymbol{U}^{\dagger}\boldsymbol{H}\boldsymbol{U}, \tag{8.61}$$

Another interesting feature of Eq. (8.58) is that it allows one to define a function f of a diagonalizable matrix.[52] If $\boldsymbol{D}$ is a diagonal matrix with elements λ_i that is $\boldsymbol{D}=\mathrm{diag}(\lambda_1,\lambda_2,\ldots,\lambda_n)$ the following definition makes sense

$$f(\boldsymbol{D})=\mathrm{diag}\left(f(\lambda_1),f(\lambda_2),\ldots,f(\lambda_n)\right). \tag{8.62}$$

Analogously, if $\boldsymbol{H}$ can be writen in the form of Eq. (8.59), one can define

$$f(\boldsymbol{H})=\boldsymbol{U}f(\boldsymbol{D})\boldsymbol{U}^{\dagger}, \tag{8.63}$$

or more explicitly with Eq. (8.58)

$$f(\boldsymbol{H})_{kl}=\sum_{i=1}^{n}u_{ki}f(\lambda_i)u_{il}^{\dagger}=\sum_{i=1}^{n}u_{ki}f(\lambda_i)u_{li}^{*}. \tag{8.64}$$

Note that in general $f(H_{kl})\neq f(\boldsymbol{H})_{kl}$, because the left-hand side means that the f is applied to the elements of the matrix rather than to the matrix itself. In Numpy a function can be applied in this element-wise fashion by writing `f(A)` where `A` is a NumPy array. This works if f is a standard function that comes with the package such as `sin`, `sqrt`, etc. If that's not the case, one has to write `vectorize(f)(A)` using the NumPy's `vectorize` to make sure that `f` expects an array of parameters as input rather than a single parameter.

---

[52]There are many other ways of defining a functions of matrices see for instance N. J. Higham. *Functions of Matrices: Theory and Computation.* DOI: https://doi.org/10.1137/1.9780898717778. URL: https://epdf.pub/functions-of-matrices-theory-and-computation339cc971f225e5cd9d3692ea08c28c6091364.html.

Table 35: A matrix multiplication of three matrices of which one is diagonal can be implemented by a regular matrix product (@) and a Hadamard multiplication (*). Using the Hadamard product is about twice as fast. Why do some of the computations produce exactly the same results, as witnessed by vanishing norms of the differences?

```
[1]: import numpy as np
     n = 1000
     A = np.random.normal(0,1,[n,n])
     B = np.random.normal(0,1,[n,n])
     d = np.random.normal(0,1,n)
     D = np.diag(d)
     E1 = A @ (D @ B)
     E2 = (A @ D) @ B
     F = A @ (d*B.T).T
     G = (A*d) @ B
     print('||E1-F|| = ',np.linalg.norm(E1-F))
     print('||E2-F|| = ',np.linalg.norm(E2-F))
     print('||E1-G|| = ',np.linalg.norm(E1-G))
     print('||E2-G|| = ',np.linalg.norm(E2-G))
```

```
||E1-F|| =  0.0
||E2-F|| =  7.786675526205083e-12
||E1-G|| =  7.786675526205083e-12
||E2-G|| =  0.0
```

```
[2]: %timeit E1 = A @ (D @ B)
     %timeit E2 = (A @ D) @ B
     %timeit F = A @ (d*B.T).T
     %timeit G = (A*d) @ B
```

```
49.1 ms ± 996 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
48.1 ms ± 188 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
25.9 ms ± 243 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
26.7 ms ± 359 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Table 36: Accessing columns of a random symmetric matrix

```
import numpy as np
m =5
n = 10
x = np.random.random([n,m])
print(x) # all m columns
...
print(x[:,2]  # only column 2, the third one
...
```

Table 37: Creating a random Hermitian matrix with complex elements

```
import numpy as np
n = 5
M = np.random.random([n,n]) +  1j * np.random.random([n,n])
M = M + np.conjugate(M.T)
eval, evec = np.linalg.eigh(M)
```

8.3.1   Assignments

Assignment 24 Gram-Schmidt orthonormalization

1. Pick some non-trivial integers m and n with $m < n$. Generate n-component random vectors $\boldsymbol{x}_i$, $i = 1, \ldots, m$ and use the Gram-Schmidt procedure of algorithm 4 to construct a set of orthonormal vectors $\boldsymbol{y}_i$, $i = 1, \ldots, m$ such that

$$\text{span}(\boldsymbol{x}_i | i = 1, \ldots, m) = \text{span}(\boldsymbol{y}_i | i = 1, \ldots, m) \tag{8.65}$$

2. To demonstrate that Eq. (8.65) is satisfied, choose a random linear combination of the vectors $\boldsymbol{x}_i$. Find its components in the $\boldsymbol{y}_i$ basis and show that the corresponding linear combination of $\boldsymbol{y}_i$ reproduces the same vector.

Hint: Table 36 once again shows how to access columns of a random matrix.

Assignment 25 Resolution of the identity
Generate a random matrix $\boldsymbol{M}$ with complex elements that have both their real and imaginary parts in the interval $(0, 1)$. Add this matrix to its Hermitian adjoint. Find its eigenvalues and eigenvectors $\boldsymbol{u}_i$.

1. Numerically verify Eq. (8.52).

Hint: see Table 37

Table 38: Computing the eigenvalues and eigenvectors of a random Hermitian matrix

```
import numpy as np
n = 10
A = np.random.random([n,n]) + 1j * np.random.random([n,n])
A = A + A.conj().T
evalA, evecA = np.linalg.eigh(A)
```

Assignment 26 Cauchy interlace theorem

The Cauchy interlace theorem states that the eigenvalues of a Hermitian matrix $\boldsymbol{A}$ of order n are interlaced with those of all of its principal submatrices of order $n-1$, submatrices obtained by removing from $\boldsymbol{A}$ row and column i with $i = 1, \ldots, n$. Interlaced means that if the respective eigenvalues are $\lambda_1, \ldots, \lambda_n$ and $\lambda'_1, \ldots, \lambda'_{-1}n$, then

$$\lambda_i \leq \lambda'_i \leq \lambda_{i+1} \text{ with } i = 1, \ldots, n-1. \tag{8.66}$$

Write a program that verifies Cauchy's interlace theorem of the eigenvalues of a Hermitian matrix. Do this by generating a random Hermitian matrix and comparing its eigenvalues with those of the matrix obtained by dropping its last row and column. The program must output a `True` or `False`.

Hint: see Table 38.

Assignment 27 Square root of a Hermitian matrix

Choose a value of $n \approx 10$. Generate a random $n \times n$ matrix $\boldsymbol{R}$ with elements the real and imaginary parts of which are $U(0,1)$ random numbers. Define $\boldsymbol{Q} = \boldsymbol{R}\boldsymbol{R}^{\dagger}$.

1. Show that $\boldsymbol{Q}$ is positive definite, i.e., for any non-vanishing column vector $\boldsymbol{v}$

$$\boldsymbol{v}^{\dagger}\boldsymbol{Q}\boldsymbol{v} > 0. \tag{8.67}$$

2. Use the spectral decomposition of $\boldsymbol{Q}$ to construct the matrix $\boldsymbol{Q}^{\frac{1}{2}}$

3. Use %timeit to time the difference in speed of implementing Eq. (8.58) with two matrix multiplications (@) or with the row/column wise multiplication ($*$) as shown in the example in the following hint.

Hint: see Tables 39 and 8 for Hadamard-like row- and column-wise matrix multiplications.

Assignment 28 Matrix squaring

Let $\boldsymbol{A}$ be a random, symmetric real matrix. Generate

$$\boldsymbol{A}_1 = \boldsymbol{A}/\left\|\boldsymbol{A}\right\|_{\infty}, \boldsymbol{A}^2/\left\|\boldsymbol{A}^2\right\|_{\infty}, \ldots, \boldsymbol{A}^{2n}/\left\|\boldsymbol{A}^{2n}\right\|_{\infty}, \ldots \tag{8.68}$$

where $\left\|\boldsymbol{A}\right\|_{\infty}$ is the absolute value of the largest element of the matrix $\boldsymbol{A}$. What is the limit of this sequence?

8.4 Bra-ket notation

Points in space exist independently of the coordinate system one chooses to specify their coordinates. The same is true for the representation one chooses to describe the state of a quantum mechanical system. Dirac introduced the coordinate free bra-ket notation to make the independence of the representation explicit for quantum mechanical systems. The bra-ket notation also makes it simple to discuss coordinate transformations that allow one to go from one representation to another.

In Dirac notation a column vector $\boldsymbol{x}$ represents a ket $|\boldsymbol{x}\rangle$. The row vector $\boldsymbol{x}^{\dagger}$ together with its matrix multiplication to the right represents the bra $\langle\boldsymbol{x}|$. By including the matrix multiplication, the row vector actually is a linear function defined on the linear space of kets. The ket and bra spaces have the same mathematical structure. The relationship between the two is known as *duality*.

The inner product of Eq. (8.11) takes the form

$$\langle\boldsymbol{x}|\boldsymbol{y}\rangle = (\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{x}^{\dagger}\boldsymbol{y}. \tag{8.69}$$

Clearly the notation we're using is slightly redundant.

The inner product is independent of the coordinate system, that is the basis of choice; see Eq. (8.31). As is discussed in many books on quantum mechanics, an orthonormal set of eigenkets $|\boldsymbol{u}_i\rangle$ of Hermitian operators defines a basis, aka representation. Given a complete orthonormal basis $|\boldsymbol{\alpha}_k\rangle$, $k = 1, 2, \ldots, n$, that is a set satisfying

$$\sum_{k=1}^{n}|\boldsymbol{\alpha}_k\rangle\langle\boldsymbol{\alpha}_k| = \mathbb{1}, \tag{8.70}$$

$$\langle\boldsymbol{\alpha}_k|\boldsymbol{\alpha}_l\rangle = \delta_{kl}, \tag{8.71}$$

Table 39: Hadamard-like row- and column-wise matrix multiplication implemented by NumPy

```
>>> import numpy as np
>>> x = np.array([[1,2,3], [4,5,6]])
>>> a = np.array([2,3,4])
>>> print('x*a\n', x * a)
x*a
 [[ 2  6 12]
 [ 8 15 24]]
>>> print('a*x\n', a * x)
a*x
 [[ 2  6 12]
 [ 8 15 24]]
>>> print('x @ diag(a)\n', x @ np.diag(a) )
x @ diag(a)
 [[ 2  6 12]
 [ 8 15 24]]
>>> b = np.array([[3],[4]])
>>> print('b*x\n', b * x)
b*x
 [[ 3  6  9]
 [16 20 24]]
>>> print('x*b\n', x * b)
x*b
 [[ 3  6  9]
 [16 20 24]]
>>> c = np.array([3,4])
>>> print('diag(c) @ x\n', np.diag(c) @ x )
diag(c) @ x
 [[ 3  6  9]
 [16 20 24]]
```

In the $\boldsymbol{\alpha}$ basis othonormality takes the form

$$\langle \boldsymbol{u}_i | \boldsymbol{u}_j \rangle = \sum_{k=1}^{n} u_{ik}^* u_{kj} = \delta_{ij} \tag{8.72}$$

where $u_{ik}^* = \langle \boldsymbol{u}_i | \boldsymbol{\alpha}_k \rangle^* = \langle \boldsymbol{\alpha}_k | \boldsymbol{u} \rangle$ and $u_{kj} = \langle \boldsymbol{\alpha}_k | \boldsymbol{u}_j \rangle$. Once again, completeness of the $\boldsymbol{u}$ basis takes the form:

$$\sum_{i=1}^{n} |\boldsymbol{u}_i \rangle \langle \boldsymbol{u}_i | = \mathbb{1} \tag{8.73}$$

in bra-ket notation. Multiplying this expression through by $|\boldsymbol{\alpha}_l\rangle$ give the transformation from the $\boldsymbol{\alpha}$ to the u basis

$$\sum_{i=1}^{n} |\boldsymbol{u}_i \rangle \langle \boldsymbol{u}_i | \boldsymbol{\alpha}_j \rangle = |\boldsymbol{\alpha}_j \rangle, \tag{8.74}$$

where the $\langle \boldsymbol{u}_i | \boldsymbol{\alpha}_j \rangle$ are elements of a unitary matrix.

8.5 Infinite bases

The inner product defined for finite sequences of complex numbers in the normed linear space l^2 with elements in $\mathbb{C}^n$ can be generalized to infinite sequences of complex numbers x_i with finite norm defined as

$$\|\boldsymbol{x}\| = \sqrt{\sum_{i=1}^{\infty} |x_i|^2} < \infty \tag{8.75}$$

The inner product becomes

$$(\boldsymbol{x}, \boldsymbol{y}) = \sum_{i=1}^{\infty} x_i^* y_i. \tag{8.76}$$

The Cauchy-Schwarz-Bunyakovsky inequality[53] guarantees that the inner product

$$\|(\boldsymbol{x}, \boldsymbol{y})\| \leq \|\boldsymbol{x}\| \, \|\boldsymbol{y}\| \tag{8.77}$$

is finite.

Analogously one can define the normed linear space L^2. It consists of complex-valued functions f such that

$$\|f\| = \sqrt{\int_{-\infty}^{\infty} |f(x)|^2} < \infty \tag{8.78}$$

The inner product of L^2 is

$$(f, g) = \int_{-\infty}^{\infty} f(x)^* g(x) \, dx \tag{8.79}$$

The next generalization is to extend L^2 so that it includes generalized functions (aka distributions) such as the Dirac δ-function.[54]

9 Fourier transforms

9.1 Sequences

Consider N equally spaced points on the unit circle in the complex plane $e^{ik2\pi/N}$, with $k = 0, 1, \ldots, N-1$, the N solutions of the equation $z^N = 1$. Rotation symmetry over an angle of $2\pi/N$ about the origin implies that for any two integers l and l'.

$$\frac{1}{N} \sum_{k=0}^{N-1} \left(e^{i2\pi kl/N} \right) e^{i2\pi kl'/N} = \delta_{ll'} \tag{9.1}$$

---

[53]For proof see Wikipedia. *Cauchy–Schwarz inequality*. URL: https://en.wikipedia.org/wiki/CauchySchwarz_inequality#Proofs

[54]M. J. Lighthill. *Fourier Analysis and Genralized functions*. Cambridge Universtity Press.

The discrete Fourier transform of a sequence $(f_0, f_1, \ldots, f_{N-1})$ is defined as $(g_0, g_1, \ldots, g_{N-1})$, where

$$g_k = \frac{1}{\sqrt{N}} \sum_{l=0}^{N-1} e^{-2\pi ikl/N} f_l \text{ with } k = 0, \ldots, N-1. \tag{9.2}$$

The right-hand side of this equation does not change if we replace k by $k \pm N, k \pm 2N, \cdots$. This allows us to extend the definition of the g_k to any integer k.

The inverse transform follows from Eq. (9.1)

$$f_l = \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} e^{2\pi ikl/N} g_k \text{ with } l = 0, \ldots, N-1. \tag{9.3}$$

9.2 Periodic functions

Suppose $f(x)$ is periodic in x with period $2L$, i.e., $f(x + 2L) = f(x)$. $f(x)$ can be expanded as follows:

$$f(x) = \sum_{n=-\infty}^{\infty} e^{in\pi x/n} g_n. \tag{9.4}$$

The basis functions $e^{in\pi x/n}$ satisfy the othonormality relation

$$\frac{1}{2L} \int_{-L}^{L} \left(e^{im\pi x/L} \right)^* e^{in\pi x/L} \, dx = \delta_{m,n}. \tag{9.5}$$

The g_n are obtained by using the othonormality relations. This leads to

$$g_n = \frac{1}{2L} \int_{-L}^{L} f(x) e^{-in\pi x/L} \, dx, \text{ with } n \in \mathbb{N}. \tag{9.6}$$

If we substitute this last equation into Eq. (9.4) we get

$$\begin{aligned} f(x) &= \sum_{n=-\infty}^{\infty} \left(\frac{1}{2L} \int_{-L}^{L} f(x') e^{-in\pi x'/L} \, dx' \right) e^{in\pi x/L} \\ &= \int_{-L}^{L} f(x') \left(\frac{1}{2L} \sum_{-n}^{N} e^{2\pi in(x-x')/L} \right) dx'. \end{aligned} \tag{9.7}$$

If the left-hand side of this equation is to reproduce $f(x)$ for any function f it must be true that

$$\frac{1}{2L} \sum_{n=-\infty}^{\infty} e^{2\pi in(x-x')/L} = \delta(x - x'), \tag{9.8}$$

which, as follows from Eq. (8.6), is equivalent to

$$\frac{1}{2\pi} \sum_{n=-\infty}^{\infty} e^{in(x-x')} = \delta(x - x'). \tag{9.9}$$

This relationship expresses the completeness of the discrete, infinite set of exponential functions on the finite interval $(-\pi, \pi)$ and their periodic extensions to $(-\infty, \infty)$.

For real functions it may be computationally advantageous to write

$$f(x) = \frac{1}{2}a_0 + \sum_{n=1}^{\infty} a_n \cos nx + \sum_{n=1}^{\infty} a_n \sin nx. \tag{9.10}$$

By expressing the trigonometric in terms of the exponential functions one gets:

$$\left. \begin{aligned} a_n &= c_n + c_{-n} && \text{with } n \in \mathbb{N}_0, \\ b_n &= i(c_n - c_{-n}) && \text{with } n \in \mathbb{N}. \end{aligned} \right\} \tag{9.11}$$

9.2.1 Assignments

Assignment 29 δ-sequence

A δ-sequence of functions $\delta_N(x)$ converging to the Dirac δ-function with $N = 1, 2, \ldots$ can be obtained by truncating the doubly infinite series in Eq. (9.9), that is, $\sum_{-\infty}^{\infty} \to \sum_{-N}^{N}$. Illustrate this graphically using the Python plotting library.

9.3 Non-periodic functions

Let $f(x)$ be a function defined for $-\infty < x < \infty$. Define a periodic function $f_l(x)$ with period $2L$ so that $f_L(x) = f(x)$ for $-L < x < L$, as illustrated in Fig. 9.

Figure 9: Periodic function $f_l(x)$ defined in terms of non-periodic $f(x)$
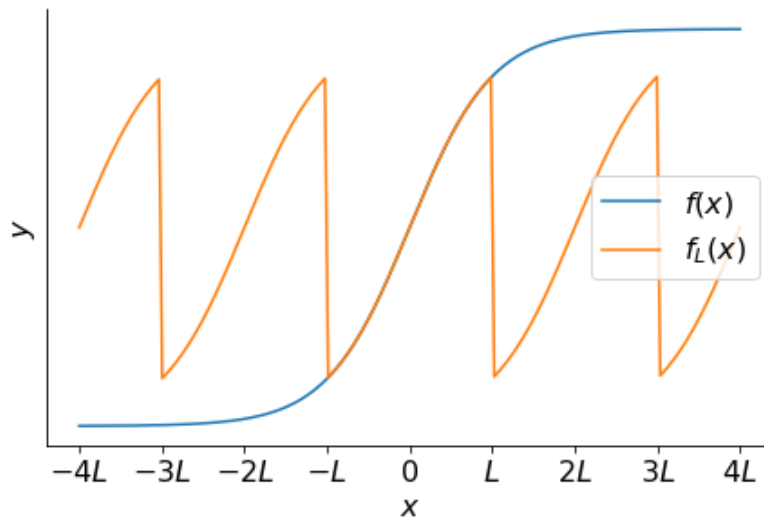
Because f_L is periodic we can expand it as follows

$$f_L(x) = \frac{1}{2L} \sum_{n=-\infty}^{\infty} e^{in x \pi / L}, g_L\left(\frac{n}{2L}\right), \tag{9.12}$$

where the notation assumes that the functions in the sequence $g_1, \ldots, g_N$ in Eqs. (9.2) and 9.3) are neighboring values of a function $g(x)$.

For $L \to \infty$ the Riemann sum on the right-hand side of Eq. (9.12) approaches an integral, which we assume exists. This gives

$$f(x) = \int_{-\infty}^{\infty} e^{2\pi i x y} g(y)\, dy, \tag{9.13}$$

$$g(y) = \int_{-\infty}^{\infty} e^{-2\pi i x y} f(x)\, dx. \tag{9.14}$$

In physics people tend to use a different but equivalent and more symmetric convention to define the Fourier transform

$$f(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{i x y} g(y)\, dy, \tag{9.15}$$

$$g(y) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{-i x y} f(x)\, dx. \tag{9.16}$$

This convention results in a unitary Fourier transformation. In quantum mechanics this conserves probability. There are also different conventions for the sign of in the exponential. Following the same procedure as in Eq. (9.7), we find the following completeness (closure) relation

$$\frac{1}{2\pi} \int_{-\infty}^{\infty} e^{i x (y - y')}\, dx = \delta(y - y'). \tag{9.17}$$

For the finite case we have

$$\frac{1}{N} \sum_{k=0}^{N-1} e^{2\pi i k(l-m)/N} = \delta_{lm} \text{ with } l, m = 0, 1, \ldots, n-1. \tag{9.18}$$

It is useful to write Eq. (9.18) in matrix form. Define

$$\boldsymbol{U} = (U_{kl})_{k,l=0}^{N-1} \text{ with } U_{kl} = \frac{e^{2\pi i k l/N}}{\sqrt{N}}. \tag{9.19}$$

Then Eq. (9.18) takes the form

$$\mathbf{U}\mathbf{U}^\dagger = \mathbf{U}^\dagger\mathbf{U} = \mathbb{1} \tag{9.20}$$

with $\mathbb{1}$ the $N \times N$ identity matrix, with ones along the diagonal and zeros elsewhere.

By introducing column vectors $\boldsymbol{f}$ and $\boldsymbol{g}$ with components f_k and g_l respectively we can put Eqs. (9.2) and (9.3) in the form

$$\boldsymbol{f} = \mathbf{U}\boldsymbol{g}, \tag{9.21}$$
$$\boldsymbol{g} = \mathbf{U}^\dagger\boldsymbol{f}. \tag{9.22}$$

Let $\mathcal{F}[f]$ be the Fourier transform of f

$$\mathcal{F}[f](\omega) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{-i\omega t} f(t) \, dt. \tag{9.23}$$

The following relationship is known as Parseval's relation

$$(f, g) = \int_{-\infty}^{\infty} f(x)^* g(x) \, dx = \int_{-\infty}^{\infty} \mathcal{F}[f](\omega)^* \mathcal{F}[g](\omega) \, d\omega = (\mathcal{F}[f], \mathcal{F}[g]). \tag{9.24}$$

This relationship is a consequence of the unitarity of the Fourier transformation. For periodic functions, a similar relation holds, except that the second integral becomes an infinite sum. For finite sequences, both integrals become finite sums.

9.4 Convolution

Let's restrict ourselves for real functions f and g defined on the interval $(-\infty, \infty)$. The convolution of these functions is $f * g$ and is defined as

$$(f * g)(x) = \int_{-\infty}^{\infty} f(x') g(x - x') \, dx'. \tag{9.25}$$

Convolution is linear and also commutative

$$f * g = g * f, \tag{9.26}$$

which follows by a simple change of variable. Convolution is associative

$$(f * g) * h = f * (g * h). \tag{9.27}$$

The area under the curve of the convolution is the product of the area under the individual factors

$$\int_{-\infty}^{\infty} (f * g)(x) \, dx = \int_{-\infty}^{\infty} \left[\int_{-\infty}^{\infty} f(x') g(x - x') \, dx' \right] dx = \left[\int_{-\infty}^{\infty} f(x) \, dx \right] \left[\int_{-\infty}^{\infty} f(x') \, dx' \right] \tag{9.28}$$

As far as convolution is concerned, the δ-function plays the role of unity in the sense that

$$f * \delta = \delta * f \tag{9.29}$$

Now let's look at how Fourier transformation and convolution are related

$$h(x) \equiv \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(x' - x) g(x') \, dx' \tag{9.30}$$

$$= \frac{1}{2\pi} \int_{-\infty}^{\infty} g(x') \left[\int_{-\infty}^{\infty} \mathcal{F}[f](k) e^{-ik(xx-x')} \, dk \right] dx' \tag{9.31}$$

$$= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \mathcal{F}[f](k) \left[\int_{-\infty}^{\infty} g(x') e^{ikx'} \, dx' \right] e^{-ikx} \, dk \tag{9.32}$$

$$= \frac{1}{\sqrt{2\pi}} \int_{\infty}^{\infty} \mathcal{F}[f](k) \mathcal{F}[g](k) e^{-ikx} \, dk. \tag{9.33}$$

By inverse Fourier transformation and use of Eq. (9.17) this becomes

$$\int_{-\infty}^{\infty} e^{ikx} h(x)\, dx = \sqrt{2\pi} \mathcal{F}[h](k) = \mathcal{F}[f](k)\mathcal{F}[g](k). \tag{9.34}$$

In other words, apart from ever present $\sqrt{2\pi}$, which depends on the convention used in its definition, the Fourier transform of a convolution is the product of the convolved functions. A direct consequence of this is that the convolution of two Gaussians is another Gaussian. More precisely, with[55]

$$\mathcal{N}(x, u, \sigma^2) = \frac{e^{-\frac{(x-\mu)^2}{2\sigma^2}}}{\sqrt{2\pi\sigma^2}} \tag{9.35}$$

we find that

$$(\mathcal{N}(\mu_1, \sigma_1^2) * \mathcal{N}(\mu_2, \sigma_2))(x) = \mathcal{N}(x, \mu_1 + \mu_2, \sigma_1^2 + \sigma_2^2), \tag{9.36}$$

which follows directly from Eq. (9.34) and

$$\mathcal{F}[\mathcal{N}(\mu, \sigma^2)](k) = \frac{e^{ik\mu - \frac{1}{2}k^2\sigma^2}}{\sqrt{2\pi\sigma^2}}. \tag{9.37}$$

To perform the Fourier transformation for the latter equation, complete the square as discussed in reference to Eq. (6.22).

9.4.1 Example: image reconstruction

All of the above can be generalized to two and higher dimensions. Let's consider the case of a two-dimensional picture that has been blurred. The image of a point should be a point at $\boldsymbol{r} = (x, y)$ but suppose it has become a blur $G(\boldsymbol{r} - \boldsymbol{r}')$ for example

$$G(\boldsymbol{r} - \boldsymbol{r}') = \frac{1}{2\pi\sigma^2} e^{-\frac{(x-x')^2 + (y-y')^2}{2\sigma^2}} \tag{9.38}$$

Instead of the true image $f(\boldsymbol{r})$ we obtain the blurred image

$$b(\boldsymbol{r}) = \int G(\boldsymbol{r}' - \boldsymbol{r}) f(\boldsymbol{r})\, d\boldsymbol{r}. \tag{9.39}$$

i.e. $b = G * f$. It follows immediately from the convolution theorem that in reciprocal Fourier space image reconstruction becomes division.

$$f \propto \mathcal{F}^{-1}\left[\frac{\mathcal{F}[b]}{\mathcal{F}[G]} \right] \tag{9.40}$$

We recover the ideal, no-blur case, in the $\sigma^2 \to 0$ limit. In that case, G becomes a δ function and its Fourier transform a constant, i.e. the infinite variance limit of a Gaussian.

9.5 Vibrating string

The equation for a vibrating one-dimensional string with excursion $u(x, t)$ at position x at time t is

$$\frac{\partial^2 u}{\partial x^2} = \frac{1}{v^2} \frac{\partial^2 u}{\partial t^2}. \tag{9.41}$$

We assume that the string is fixed at both ends and that its shape $u(x, 0)$ velocity $u_t(x, 0) = 0$ are given at $t = 0$. Time and position can be transformed linearly so that in redefined dimensionless new variables x and t the problem and it boundary conditions take the form:

$$\frac{\partial^2 u}{\partial x^2} = \frac{\partial^2 u}{\partial t^2}, \tag{9.42}$$

$$u(0, t) = u(1, t) = 0 \text{ for all } t, \tag{9.43}$$

$$u(x, 0) = \begin{cases} 2x \text{ if} & 0 \le x \le \frac{1}{2}, \\ 2(x-1) \text{ if} & \frac{1}{2} \le x \le 1, \end{cases} \tag{9.44}$$

$$u_t(x, t) \equiv \frac{\partial u(x, t)}{\partial t} = 0 \text{ at } t = 0, \tag{9.45}$$

---

[55] Compare with $\mathcal{N}(x; \mu, \sigma^2)$ as defined in Eq. (6.11).

The functions $\exp[i(kx + \omega t)]$ form a complete set in which one can expand the solutions to Eq. (9.42). Substitution in this equation shows that

$$k^2 + \omega^2 = 0. \tag{9.46}$$

The only combination of exponentials that satisfies the boundary conditions at $x = 0$ and $x = 1$ in Eq. (9.43), and the initial contidion at $t = 0$ in Eq. (9.45), is to have

$$u(x,t) = \sum_{k=1}^{\infty} a_k \sin(\pi k x) \cos(\pi k t). \tag{9.47}$$

The constants a_k follow from the known function $u(x,0)$. For this purpose use the orthonormality relation

$$\int_0^1 \sqrt{2} \sin(\pi m x) \sqrt{2} \sin(\pi n x)\, dx = \delta_{mn}. \tag{9.48}$$

This gives

$$a_k = \int_0^1 \sqrt{2}\sin(\pi k x) u(x,0)\, dx = \int_0^{\frac{1}{2}} \sqrt{2}\sin(\pi k x)(2x)\, dx + \int_{\frac{1}{2}}^1 \sqrt{2}\sin(\pi k x)[2(1-x)]\, dx = \frac{4\sqrt{2}\sin\left(\frac{1}{2}\pi k\right)}{\pi^2 n^2}. \tag{9.49}$$

Note that $\sin(\frac{1}{2}k\pi)$ vanishes for even values of k, equals 1 for $k = 1, 5, 9, \ldots$ and equals -1 for $k = 3, 7, 11, \ldots$.

9.5.1 Assignments

Assignment 30 Generalize and animate vibrating sting

1. Let $0 < x_0 < 1$. Generalize the solution of the vibrating string problem to the case of a string that at time $t = 0$ increases linearly from $x = 0$ to $x = x_0$—with $x_0 \neq \frac{1}{2}$ unlike in Eq. (9.44)—and decreases linearly from $x = x_0$ to $x = 1$.

2. Animate the motion of the string for $t > 0$ through several cycles.

Hint: Table 40 shows how to produce a movie of a simple animated line plot.

```
"""
Matplotlib Animation Example

author: Jake Vanderplas
email: vanderplas@astro.washington.edu
website: http://jakevdp.github.com
license: BSD
Please feel free to use and modify this, but keep the above information. Thanks!
"""

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

# First set up the figure, the axis, and the plot element we want to animate
fig = plt.figure()
ax = plt.axes(xlim=(0, 2), ylim=(-2, 2))
line, = ax.plot([], [], lw=2)

# initialization function: plot the background of each frame
def init():
    line.set_data([], [])
    return line,

# animation function.  This is called sequentially
def animate(i):
    x = np.linspace(0, 2, 1000)
    y = np.sin(2 * np.pi * (x - 0.01 * i))
    line.set_data(x, y)
    return line,

# call the animator.  blit=True means only re-draw the parts that have changed.
anim = animation.FuncAnimation(fig, animate, init_func=init,
                               frames=200, interval=20, blit=True)

# save the animation as an mp4.  This requires ffmpeg or mencoder to be
# installed.  The extra_args ensure that the x264 codec is used, so that
# the video can be embedded in html5.  You may need to adjust this for
# your system: for more information, see
# http://matplotlib.sourceforge.net/api/animation_api.html
anim.save('basic_animation.mp4', fps=30, extra_args=['-vcodec', 'libx264'])

plt.show()
from IPython.display import HTML
HTML(anim.to_html5_video())
```

9.6 Diffusion

Start from a density that evolves in time in one spatial dimension $\rho(x, t)$. The diffusion equation is

$$\frac{\partial \rho}{\partial t} = D \frac{\partial^2 \rho}{\partial x^2}. \tag{9.50}$$

Define the operator $\mathbf{H}$ as

$$\mathbf{H} = D \frac{\partial^2}{\partial x^2}. \tag{9.51}$$

For an infinitesimal time interval, using the definition of the time derivative, we have

$$\rho(x, t + \delta) - \rho(x, t) = \delta t \mathbf{H} \rho \tag{9.52}$$

so that

$$\rho(x, t + \delta t) = (\mathbb{1} + \delta t\, \mathbf{H})\rho(x, t). \tag{9.53}$$

In other words, the operator $1 + \delta t\, \mathbf{H})$ generates infinitesimal time translations, *i.e.*, it evolves the density by a time step δt. To evolve the density over a finite time t we can apply evolve the N times over a time t/n and consider the $n \to \infty$ limit

$$\lim_{n \to \infty} (\mathbb{1} + \frac{t}{n}\mathbf{H})^n = e^{t\mathbf{H}} \equiv \mathbf{G}(t). \tag{9.54}$$

The result is

$$\rho(x, t) = \mathbf{G}(t)\rho(x, 0), \tag{9.55}$$

which is pretty useless unless we figure out what the operator $\mathbf{G}$, aka the Green's function of the diffusion equation.

A simple way to evaluate $\mathbf{G}$ is to use Dirac's bra-ket notation: $\rho(x, t) \equiv \langle x | \rho(t) \rangle$ Write

$$\langle x | \rho(t) \rangle = \int_{-\infty}^{\infty} \langle x | \mathbf{G}(t) | x' \rangle \, dx' \, \langle x' | \rho(0) \rangle, \tag{9.56}$$

which is obtained from Eq. (9.55) by inserting the following resolution of the identity operator

$$\mathbb{1} = \int_{-\infty}^{\infty} |x\rangle \, dx \, \langle x| \tag{9.57}$$

In other words,

$$\langle x | \mathbf{G}(t) | x' \rangle = \langle x | e^{t\mathbf{H}} | x' \rangle. \tag{9.58}$$

Now use that in the position representation $-i\partial/\partial x$ represents the momentum operator $\mathbf{p}$, which has eigenstates satisfying

$$\boldsymbol{p}|p\rangle = p|p\rangle. \tag{9.59}$$

Note that the dimensionless momentum *operator*, as it is called in the context of quantum mechanics, is denoted by $\boldsymbol{p}$ and its *eigenvalues* by $-\infty < p < \infty$. We'll use the following mathematical relations that can be found in any quantum mechanics book:[56]

$$\langle x | p \rangle = \frac{e^{ipx}}{\sqrt{2\pi}} \tag{9.60}$$

$$\langle p | x \rangle = \frac{e^{-ipx}}{\sqrt{2\pi}} \tag{9.61}$$

and, momentum space analog of Eq. (9.57)

$$\mathbb{1} = \int_{-\infty}^{\infty} |p\rangle \, dp \, \langle p|. \tag{9.62}$$

Use this in Eq. (9.58) to obtain

$$\langle x | \mathbf{G}(t) | x' \rangle = \int_{-\infty}^{\infty} dp \int_{-\infty}^{\infty} dp' \, \langle x | p \rangle \langle p | e^{-tD\mathbf{p}^2} | p' \rangle \langle p' | x' \rangle. \tag{9.63}$$

The operator $\mathbf{p}$ is diagonal in the momentum representation, as are functions of $\mathbf{p}$. As a consequence,

$$\langle p | e^{-tD\mathbf{p}^2} | p' \rangle = e^{-tDp^2} \delta(p - p') \tag{9.64}$$

Now substitute Eqs. (9.61-9.61) and Eq. (9.64) into Eq. (9.63), complete the square in the exponent, transform the coordinates so that the integral takes the form

$$\frac{1}{2\pi} \int_{-\infty}^{\infty} e^{x^2/2} \, dx = 1 \tag{9.65}$$

---

[56] In the context of quantum mechanics $\mathbb{1} - i\boldsymbol{p}dx/\hbar$ plays the role of the infinitesimal translation operator with $\boldsymbol{p}$ where $\boldsymbol{p}$ is the usual dimensionful momentum. In dimensionless, mathematical terms this is nothing but a restatement of Taylor's theorem $f(x + \Delta x) = f(x) + f'(x)\Delta x + \mathcal{O}\left((\Delta x)^2\right)$, which can also be written as $f(x + \Delta x) = \exp(\Delta x d/dx)f(x)$.

The result is

$$G(x, x'; t) = \frac{1}{\sqrt{2\pi Dt}} \exp\left(\frac{-(x - x')^2}{4Dt}\right),$$ (9.66)

which describes the time evolution of a system described by a $\delta(x - x')$ at time $t = 0$. The evolution of a general initial mass distribution $\rho(x, 0)$ at $t = 0$ takes the form of the convolution

$$\rho(x, t) = \int_{-\infty}^{\infty} G(x, x'; t) \rho(x', 0) \, dx.$$ (9.67)

9.6.1   Drift-diffusion

Diffusion Eq. (9.50) is the continuum limit of the drunken sailor problem who has no sense of direction whatsoever. In that case the distance covered is proportional to the square root of time, which corresponds to a vanishing speed of propagation. This is a manifestation of the central limit theorem, as is Eq. (9.66).

Drift-diffusion is the case in which the sailor still has a sense of direction and moves with a non-zero velocity. In the continuum limit this leads to the equation

$$\frac{\partial \rho}{\partial t} = D \frac{\partial^2 \rho}{\partial x^2} + v \frac{\partial \rho}{\partial x}.$$ (9.68)

By transforming to a frame of reference that moves along with the drift velocity v it follows that $\rho(x - tv)$ satisfies the diffusion equation Eq. (9.50).

The corresponding Green's function is

$$G_{\mathrm{DD}}(x, x'; t) = \frac{1}{\sqrt{2\pi Dt}} \exp\left(\frac{-(x - vt - x')^2}{4Dt}\right),$$ (9.69)

as follows form Eq. (9.66).

10   Divide-and-conquer algorithms

10.1   Multiplication

One of the simplest examples of a divide-and-conquer algorithm is its application to multiplication of two integers. The point is that one can do better than the standard multiplication algorithm that is taught in elementary school. If you multiply two n-digit numbers you have to multiply each digit of one by each digit of the other. That takes n^2 operations even if we ignore the n additions that you have to perform.

Consider the multiplication of two n-bit numbers x and y. Write $x = a2^{n/2} + b$ and $y = c2^{n/2} + d$ where a, b, c and d are $\frac{1}{2}n$-bit numbers. From

$$xy = (a2^{n/2} + b)(c2^{n/2} + d) = ac2^n + (ad + bc)2^{n/2} + bd$$ (10.1)

it follows that $z = xy$ can be computed from the following program

```
u = (a + b) * (c + d)
v = a * c
w = b * d
z = v * 2^n + (u - v - w) * 2^{n/2} + w
```

The multiplications by powers of two can be implemented by simple bit-shift operation operations each of which takes an amount of time on the order $\frac{1}{2}n$ in units of the time it takes to perform such elementary computer operations. This shows that three $\frac{1}{2}n$-bit multiplications suffice. The process can be done repeatedly. The resulting recursion relation is:

$$T(n) = 3T(n/2) + kn,$$ (10.2)

where the kn represents the bit-shift and addition operations.

For simplicity we assume that n is a power of 2. To find the solution of the recursion relation in Eq. (10.2 first consider the homogeneous equation

$$T_0(n) = 3T_0(n/2) = 3^2 T_0(n/2^2) = \cdots = 3^b T_0(n/2^b)$$ (10.3)

which follows by choosing b so that $2^b = n$. This gives

$$T_0(n) = 3^{\log_2 n} T_0(1) = 2^{\log_2 3 + \log_2 n} T_0(1) = n^{\log_2 3} T_0(1) \tag{10.4}$$

To obtain equation Eq. (10.3) from Eq. (10.2) subtract $t(n) = an$ with a chosen so that

$$t(n) = 3t(n/2) + kn, \tag{10.5}$$

that is with $a = -2k$.

The result is

$$T(n) = n^{\log_2 3} T_0(1) - 2kn \tag{10.6}$$

Finally, choose $T_0(1)$ so that $T(1) = k$. This gives

$$T(n) = 3kn^{\log_2 3} - 2kn. \tag{10.7}$$

Because $\log_2 3 \approx 1.59$, this grows more slowly with n than (n^2).

A similar approach works for matrix multiplication by dividing matrices into block.[57]

10.2 Fast Fourier transform

The divide-and-conquer algorithm that produces the fast Fourier transform works in a similar way.

Suppose that we want to expand a periodic function f as

$$f(x) = \sum_{j=0}^{N-1} g_j e^{ijx} \tag{10.8}$$

in terms of the values of f at $2\pi k$, for $k = 0, 1, \ldots, N-1$. In other words,

$$c_j = \frac{1}{N} \sum_{k=0}^{N-1} \alpha_k \omega^{jk}, \tag{10.9}$$

with

$$\omega = e^{2\pi i/N} \text{ and } \alpha_k = \frac{1}{N} f\left(\frac{2\pi k}{N}\right) \tag{10.10}$$

Implemented straightforwardly with Horner's rule, each of the N coefficients c_j will take N complex multiplications, a total of N^2. Fast Fourier transform, as we'll see takes only $2N \log_2 N$ complex multiplications. For $N = 2^{10} = 1024$ the latter is about 50 times faster.

Again we consider the simple case in which $N = 2^n$ is an integral power of 2. Write $k = 2l$ if k is even and $k = 2l + 1$ if k is odd, so that $0 \le l \le 2^{n-1}$. Then we can write

$$c_j = \sum_{l=0}^{2^{n-1}} \alpha_{2l} (\omega^2)^{jl} + \sum_{l=0}^{2^{n-1}} \alpha_{2l+1} (\omega^2)^{jl} \omega^j \tag{10.11}$$

This expression is a linear combination of two Fourier transforms involving $\frac{1}{2}N$ instead of the original N terms, a property that once again can be exploited recursively. To see this in detail write $j = r2^{n-1} + j_1$ with $j_1 = j \mod 2^{n-1}$, the remainder of j divided by 2^{n-1}, so that $r = 0$ or $r = 1$. Because $\omega^N = 1$ we have

$$(\omega^2)^{jl} = (\omega^2)^{j_1 l} \tag{10.12}$$

We can now write Eq. (10.11) as

$$c_j = d_{j_1}^{(1)} + \omega^j d_{j_1}^{(2)} \tag{10.13}$$

with

$$\left. \begin{array}{l} d_{j_1}^{(1)} = \sum_{l=0}^{2^{n-1}} \alpha_{2l} (\omega^2)^{jl} \\ d_{j_1}^{(2)} = \sum_{l=0}^{2^{n-1}} \alpha_{2l+1} (\omega^2)^{jl} \end{array} \right\} \text{ with } j_1 = 0, 1, \ldots, 2^{n-1} \tag{10.14}$$

The essence of the fast Fourier transform is shown in Fig. 10. The full transform corresponds to the crosses and dots. The latter correspond to the first term on the right-hand side of Eq. (10.13) and the

---

[57] For details see W. H. Press et al. *2.11 Is Matrix Inversion an N^3 Process?* URL: http://phys.uri.edu/~nigh/NumRec/bookfpdf/f2-11.pdf

crosses to the second term. The displacement of one with respect to the other comes from the factor ω^j in the latter.

Figure 10: Points ω^j with $j = 0, \ldots, 2^5$ on the unit circle in the complex plane: fast Fourier transform illustrated for the case $N = 2^5$ as it is divided into two 2^4 point transforms. See Table 41 for the code to generate this picture.
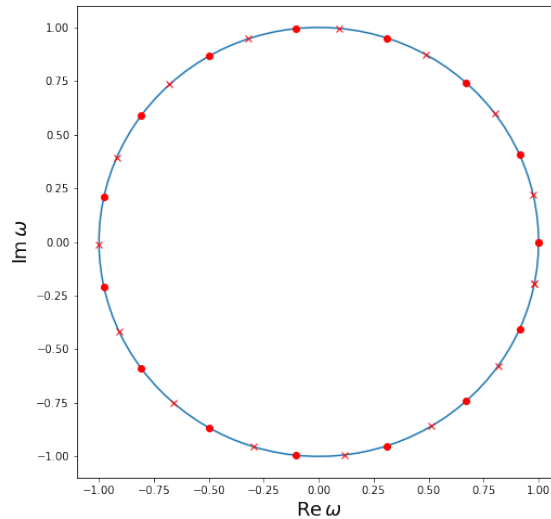
Table 41: Python code to generate Fig. 10

```python
import numpy as np
from matplotlib import pyplot as plt
def x(phi, r):
    return r*np.cos(phi)
def y(phi, r):
    return -r*np.sin(phi)
plt.figure(figsize=(8,8))
npnt =100
phi_range = np.linspace(0, 2*np.pi, npnt)
n = 4
tpn = 2**n
phi_range1 = np.linspace(0, 2*np.pi, tpn)
s = np.pi/tpn
phi_range2 = np.linspace(s, s+ 2*np.pi, tpn)
plt.plot(x(phi_range, 1), y(phi_range, 1))
plt.plot(x(phi_range1, 1), y(phi_range1, 1), 'ro')
plt.plot(x(phi_range2, 1), y(phi_range2, 1), 'rx')
plt.xlabel('Re$\,\omega$',fontsize = 18)
plt.ylabel('Im$\,\omega$',fontsize = 18)
plt.savefig('fft-32.png')
# avoid losing edges with plt.savefig('fft-32.png', bbox_inches='tight')
plt.show()
```

The conclusion is that the fast Fourier transform has the following divide-and-conquer recursion relation:

$$T(n) = 2T(\frac{n}{2}) + 2n \tag{10.15}$$

The method we use to solve Eq. (10.2) fails in this case, but it can be recovered by solving

$$T(n, x) = 2^x T(\frac{n}{2}) + 2n \tag{10.16}$$

and taking the limit of $x \to 1$ after the initial condition $T(1, x) = k$ has been imposed.

The analog of Eq. (10.5) in this case, once again assuming that $t(n) = an$, reads

$$t(n) = n^x t(n/2) + 2n \tag{10.17}$$

an equation that is satisfied for $a = 4/(2^x - 1)$. This expression for a diverges as $x \to 1$, which shows why the case $x = 1$ has to be dealt with carefully. The homogeneous equation

$$T_0(n, x) = 2^x T_0(n/2, x) \tag{10.18}$$

has the solution $T_0(n, x) = n^x T_0(1, x)$. Adding to this the solution of Eq. (10.17) and imposing the condition $T(1, x) = k$ gives

$$T(n, x) = \frac{[k(2^x - 2) + 4]n^x - 4n}{2^x - 2} \tag{10.19}$$

The limit $x \to 1$ follows by using l'Hospital's rule:

$$T(n) = n\left(k + \frac{2\log(n)}{\log(2)}\right) \tag{10.20}$$

10.2.1 Assignments

Assignment 31 Circulant matrix

1. A circulant matrix $\boldsymbol{A}$ is of the cyclic form

$$\boldsymbol{A} = \begin{pmatrix} a_0 & a_1 & a_2 & \ldots & a_{n-1} \\ a_{n-1} & a_0 & a_1 & \ldots & a_{n-2} \\ a_{n-2} & a_{n-1} & a_0 & \ldots & a_{n-3} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_1 & a_2 & \cdots & a_{n-1} & a_0 \end{pmatrix} \tag{10.21}$$

   In other words, each row is obtained from the previous one by cyclic permutation by one step to the right.

2. Show that $\boldsymbol{U}$ as defined in Eq. (9.19) diagonalizes a circulant matrix.[58]

3. Choose n an integral power of two. Choose as the first row of the $n \times n$ matrix $\boldsymbol{A}$ the sequence $(0, 1, \ldots, n - 1)$. Row $k + 1$ of $\boldsymbol{A}$ is obtained from row k by a circular right shift, that is a cyclic permutation of one step to the right. In Python this can be implemented by `np.roll` Do `help(np.roll)` for details.

4. Use `numpy.linalg.eigvals` to obtain the eigenvalues of $\boldsymbol{A}$.

5. Use fast Fourier transformation, `numpy.fft.fft` to obtain the eigenvalues of $\boldsymbol{A}$.

6. Choose a big value of n, say 2^{10} and check that the eigenvalues obtained by both methods agree.

7. Use %timeit to compare the time "dumb" diagonalization takes compared to fast Fourier transformation.

Hint: to show that two set of eigenvalues are the same you can use `numpy.linalg.norm` or `numpy.allclose` Both of those will fail if the two sets are not in the same order. `numpy.sort` can be used for this. This routine uses lexicographical ordering denoted by $\prec$. That is if z_1 and z_2 are two complex numbers

$$z_1 \prec z_2 \Leftrightarrow (\Re z_1 < \Re z_2 \text{ or } \Im z_1 < \Im z_2 \text{ if } \Re z_1 = \Re z_2) \tag{10.22}$$

This kind of ordering produces some unexpected results when used in sorting. In an attempt to show that two sequences of complex number are the same, it only works correctly if the number are rounded sufficiently so that small rounding differences do not determine the order. See Table 42.

---

[58] This is nothing but a simple example of the famous Bloch theorem of solid state physics.

Table 42: Rounding elements of an array

```
>>> import numpy as np
>>> a = 1.11 * np.array([1,2,3,4,5])
>>> np.round(a,1)
array([1.1, 2.2, 3.3, 4.4, 5.6])
>>> np.round(a,0)
array([1., 2., 3., 4., 6.])
```

10.3   A drunken sailor on Escher-Penrose stairs

Eq. (9.50 describes the continuum limit movement of a gang of dead drunks who from one step to the next can't remember which way the are going, Brownian motion if you like. Eq. (9.68) describes the same on an incline that imposes a bias in one direction, which results in a drift.

In this section we consider the drift-diffusion problem with two differences: the motion takes place on a disrete lattice and we assume periodic boundary conditions. In other words, we're considering the motion of a drunk monks on a set of Escher-Penrose stairs, as shown in Fig. 11.

Figure 11: M.C. Escher's Penrose stairs: ascending and descending
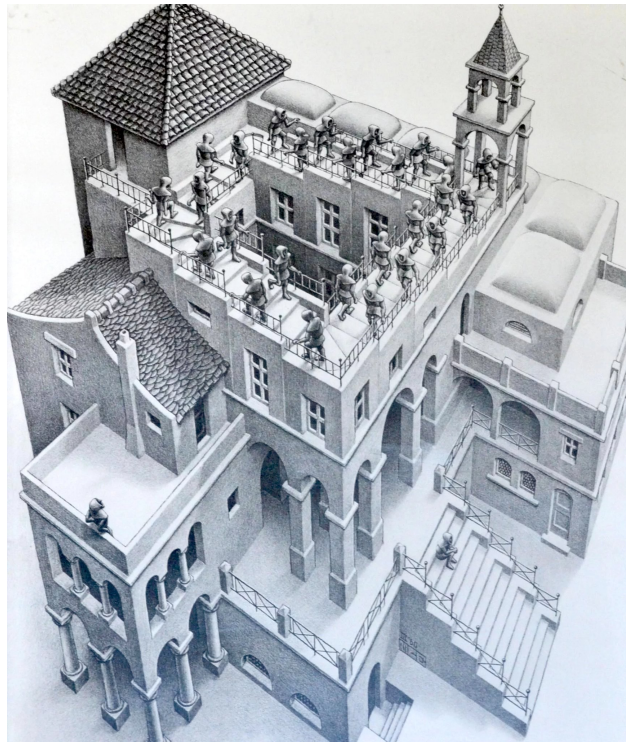
The analog of Eq. (9.50) in discrete space and continuous time t is

$$\frac{\partial \boldsymbol{\rho}}{\partial t} = \boldsymbol{H}\boldsymbol{\rho}, \tag{10.23}$$

where $\boldsymbol{\rho}$ is a column vector with ρ_k representing the fraction of the total mass concentrated on lattice site $k = 1, \ldots, N$. We assume periodic boundary (Escher-Penrose) conditions identifying sites and matrix indices $k = 0$ and $k = N + 1$. The $\boldsymbol{H}$ is a circulant matrix defined by

$$H_{ij} = \begin{cases} -2 & \text{if } i = j \\ 1 - v & \text{if } j = i + 1 \\ 1 + v & \text{if } j = i - 1 \end{cases} \tag{10.24}$$

with $i, j = 1, \ldots, N$ In other words, for $v = \frac{1}{2}$ and $N = 5$ we have

$$
\boldsymbol{H} = \begin{pmatrix}
-2 & \frac{1}{2} & 0 & 0 & \frac{3}{2} \\
\frac{3}{2} & -2 & \frac{1}{2} & 0 & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots \\
\frac{1}{2} & 0 & 0 & \frac{3}{2} & -2
\end{pmatrix},
\tag{10.25}
$$

which, as explained in section 3.5, is a finite-difference approximation of the continuum drift-diffusion Eq. (9.68).

If the intial conditions is a given $\boldsymbol{\rho}(0)$, the solution of Eq. (10.23) is

$$
\boldsymbol{\rho}(t) = e^{t\boldsymbol{H}}\boldsymbol{\rho}(0).
\tag{10.26}
$$

The exponentiated matrix on the right can be obtained by Eq. (8.63), or equivalently by the more explicit Eq. (8.64).

The fact that $\boldsymbol{H}$ is a circulant matrix makes it easy to diagonalize the matrix by Fourier transformation. The latter is particularly simple in this case, because each row of the matrix contains only 3 non-zero elements. To find the eigenvalues of a circulant matrix see assigment 31 on page 74.

10.4   Assignments

Assignment 32 Drift-diffusion animation

1. Choose some reasonable value of N, say $N = 32$ or some other, nearby power of 2. The latter because powers of 2 work well for fast Fourier transformation; see assignment 31 on page 74.

2. Choose

$$
\rho(0)_i = \begin{cases} 1 \text{ if } i = 1 \\ 0 \text{ if } i \neq 1 \end{cases}
\tag{10.27}
$$

3. Animate $\rho(t)$ using Eq. (10.26) and by evaluating $e^{t\boldsymbol{H}}$ using Fourier transformation.

Hints: 1. `from scipy.linalg import circulant, expm` will provide you with

(a) `circulant` which constructs a circulant matrix from its first row; and

(b) `expm` which exponentiates an arbitrary matrix; use this to check your program.

2. Table 40 shows how to use `matplotlib` to program the animation

11   Appendix

11.1   Meshgrid

The `meshgrid` method provides a mechanism to evaluate a function efficiently for every point of a lattice. The usual convention of labeling matrix elements M_{ij} is that i labels the rows and increases going downward; j labels the columns and increases going to the right. Typically, if (x, y) is a point in a Cartesian coordinate system x increases going to the right and y upward. To accomplish either one of the these `meshgrid` has an option `indexing`. The effect is illustrated in Tables 43 and 44. The function `map` maps the coordinates x and y onto a single number $1000x + y$ in which x and y are easily recognizable for sufficiently small values of x and y. The result of `indexing` option is just a transposition, but it is important to know which is which.

In the examples, in Tables 43 and 44, `map` is applied to two arguments arrays in two Numpy arrays `x` and `y`. The evaluation is done for pairs from corresponding locations in the arrays. For this it is necessary that the arrays have the same number of elements, but the arrays must also have the same shape. Suppose the arguments are `a = np.array([1,2,3,4])` and `b = np.array([[1,2],[3,4]])`. Clearly the number of elements is 4 in both cases, but the shapes are different. As a consequence, Python generates an error message: "`ValueError: operands could not be broadcast together with shapes (4,) (2,2).`"

Table 43: `meshgrid`— with matrix style indexing. Note that `map` is applied to corresponding pairs of its array arguments

```
[1]: import numpy as np
     x = np.linspace(11,15,5)
     y = np.linspace(21,25,5)
     map = lambda x,y: 1000 * x + y
     xy = map(x,y)
     print(xy)
```

```
[11021. 12022. 13023. 14024. 15025.]
```

```
[2]: grid_i,grid_j = np.meshgrid(x,y,indexing='ij')
     print('grid_i:\n',grid_i)
     print('grid_j:\n',grid_j)
```

```
grid_i:
 [[11. 11. 11. 11. 11.]
 [12. 12. 12. 12. 12.]
 [13. 13. 13. 13. 13.]
 [14. 14. 14. 14. 14.]
 [15. 15. 15. 15. 15.]]
grid_j:
 [[21. 22. 23. 24. 25.]
 [21. 22. 23. 24. 25.]
 [21. 22. 23. 24. 25.]
 [21. 22. 23. 24. 25.]
 [21. 22. 23. 24. 25.]]
```

```
[3]: print('map_ij:\n',map(grid_i,grid_j))
```

```
map_ij:
 [[11021. 11022. 11023. 11024. 11025.]
 [12021. 12022. 12023. 12024. 12025.]
 [13021. 13022. 13023. 13024. 13025.]
 [14021. 14022. 14023. 14024. 14025.]
 [15021. 15022. 15023. 15024. 15025.]]
```

Table 44: `meshgrid`— with Cartesian coordinate style indexing. Note that the function `map` is applied to corresponding pairs of its array arguments

```
[1]: import numpy as np
     x = np.linspace(11,15,5)
     y = np.linspace(21,25,5)
     map = lambda x,y: 1000 * x + y
     xy = map(x,y)
     print(xy)
```

```
[11021. 12022. 13023. 14024. 15025.]
```

```
[2]: grid_x,grid_y = np.meshgrid(x,y,indexing='xy')
     print('grid_x:\n',grid_x)
     print('grid_y:\n',grid_y)
```

```
grid_x:
 [[11. 12. 13. 14. 15.]
 [11. 12. 13. 14. 15.]
 [11. 12. 13. 14. 15.]
 [11. 12. 13. 14. 15.]
 [11. 12. 13. 14. 15.]]
grid_y:
 [[21. 21. 21. 21. 21.]
 [22. 22. 22. 22. 22.]
 [23. 23. 23. 23. 23.]
 [24. 24. 24. 24. 24.]
 [25. 25. 25. 25. 25.]]
```

```
[3]: print('map_xy:\n',map(grid_x,grid_y))
```

```
map_xy:
 [[11021. 12021. 13021. 14021. 15021.]
 [11022. 12022. 13022. 14022. 15022.]
 [11023. 12023. 13023. 14023. 15023.]
 [11024. 12024. 13024. 14024. 15024.]
 [11025. 12025. 13025. 14025. 15025.]]
```

11.2   Kernel error

To fix kernel error:

- look at kernel error message;

- jupyter kernelspec list: and

- in the python3 directory edit kernel.json so it agrees with the installed version of python3.

11.3   Starting Jupyter Notebook remotely

Running Jupyter Notebook remotely can be done via SSH port forwarding.

1. Local command line (at home): ssh -N -L localhost:8888:localhost:8889 nigh@phys.uri.edu

2. Local command line: ssh -X -Y nigh@phys.uri.edu

3. Remote command line: jupyter notebook –no-browser –port=8889 This produces:

```
The Jupyter Notebook is running at:
[I 14:12:48.368 NotebookApp] http://localhost:8889/?
token=dcf63bc73193c109f303d502dd83628b6149d9fce59500d5
```

4. In local browser paste: http://localhost:8888/?
token=dcf63bc73193c109f303d502dd83628b6149d9fce59500d5

Note that the first ssh command connects the local UNIX socket localhost:8888 to the remote UNIX socket localhost:8889. Hence the only difference in 3 and 4.

12 Figures, tables and bibliography

12.1 List of Figures

12.2 List of Tables

12.3 Bibliography

Apostol, T. M. "An Elementary View of Euler's Summation Formula". In: *The American Mathematical Monthly* 106.5 (1999), pp. 409–418. DOI: 10.1080/00029890.1999.12005063.

Dahlquist, G. and N. M. Åke Björk. *Numerical Methods*. Prentice-Hall, 1974.

Documentation, P. *Generate pseudo-random numbers*. URL: https://docs.python.org/3/library/random.html.

Edwards, H. M. *Riemanns' Zeta Function*. URL: https://archive.org/details/riemannszetafunc00edwa_0.

Gantmacher, F. R. *The Theory of Matrices, Vol. I and II*. American Mathematical Society, 1998.

Given a function of one variable, return a local minimum. Scipy. URL: https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.brent.html.

Golub, G. H. and C. F. Van Loan. *Matrix Computations, 4th edition*. Johns Hopkins University Press, 2013.

Higham, N. J. *Functions of Matrices: Theory and Computation*. DOI: https://doi.org/10.1137/1.9780898717778. URL: https://epdf.pub/functions-of-matrices-theory-and-computation339cc971f225e5cd9d369.html.

Hunter, J. D. "Matplotlib: A 2D graphics environment". In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.

Johansson, F. et al. *mpmath: a Python library for arbitrary-precision floating-point arithmetic (version 1.1.0)*. Dec. 2018. URL: http://mpmath.org/.

Johansson, F. et al. *mpmath: a Python library for arbitrary-precision floating-point arithmetic (version 1.1.0)*. Dec. 2018. URL: http://mpmath.org/doc/current.

Jupyter/iPython Notebook Quick Start Guide. URL: https://jupyter-notebook-beginner-guide.readthedocs.io/en/latest/what_is_jupyter.html.

Khalid, M. Y. U. *Python Tips*. URL: https://book.pythontips.com/en/latest.

Lighthill, M. J. *Fourier Analysis and Genralized functions*. Cambridge Universtity Press.

Marsaglia, G. and W. W. Tsang. "The Monty Python method for generating random variables". In: *ACM Trans. Math. Softw.* 24 (3 Sept. 1998), pp. 341–350. ISSN: 0098-3500. DOI: http://doi.acm.org/10.1145/292395.292453. URL: http://doi.acm.org/10.1145/292395.292453.

Matplotlib: Visualization with Python—Gallery. URL: https://matplotlib.org/stable/gallery/index.html.

Mehta, P. et al. "A high-bias, low-variance introduction to Machine Learning for physicists". In: *Physics Reports* 810 (2019). A high-bias, low-variance introduction to Machine Learning for physicists, pp. 1 –124. ISSN: 0370-1573. DOI: https://doi.org/10.1016/j.physrep.2019.03.001. URL: http://www.sciencedirect.com/science/article/pii/S0370157319300766.

Numerical Recipes: The Art of Scientific Computing. Third Edition. URL: http://www.numerical.recipes/.

NumPy.org. URL: https://numpy.org/doc/stable/reference/generated/numpy.linspace.html.

— *data types*. URL: https://numpy.org/doc/stable/user/basics.types.html.

Pandas: getting started. pandas. URL: https://pandas.pydata.org/pandas-docs/stable/getting_started/tutorials.html.

Press, W. H. et al. *15.4 General Linear Least Squares*. URL: http://phys.uri.edu/~nigh/NumRec/bookfpdf/f15-4.pdf.

— *2.11 Is Matrix Inversion an N^3 Process?* URL: http://phys.uri.edu/~nigh/NumRec/bookfpdf/f2-11.pdf.

— *Numerical Recipes: 10 Minimization or Maximization of Functions*. URL: http://phys.uri.edu/~nigh/NumRec/bookfpdf/f10-0.pdf.

— *Numerical Recipes: 1.2 Error, Accuracy, and Stability*. URL: http://phys.uri.edu/~nigh/NumRec/bookfpdf/f1-2.pdf#page=2.

— *Numerical Recipes: 4.2 Integration of Functions—Elementary Algorithms*. URL: http://phys.uri.edu/~nigh/NumRec/bookfpdf/f4-2.pdf.

— *Numerical Recipes: 5.6 Quadratic and Cubic Equations*. URL: http://phys.uri.edu/~nigh/NumRec/bookfpdf/f5-6.pdf.

— *Numerical Recipes: Chapter 7. Random Numbers*. URL: http://phys.uri.edu/~nigh/NumRec/bookfpdf/f7-0.pdf.

PyFormat.info. *Python formatting with practical examples*. URL: https://pyformat.info/#simple.

python.org. *3. Data model: 1. Objects, values, types*. URL: https://docs.python.org/3.9/reference/datamodel.html.

— *9.2 Python Scopes and Namespaces*. URL: https://docs.python.org/3.9/tutorial/classes.html#a-word-about-names-and-objects.

SageMath.org. URL: https://www.sagemath.org.

SciPy.org. *Special Functions*. URL: https://docs.scipy.org/doc/scipy/reference/special.html.

SymPy. *List Comprehensions*. URL: https://docs.python.org/2/tutorial/datastructures.html#list-comprehensions.

SymPy.org. URL: https://www.sympy.org/en/index.html.

The IPython Development Team. *Built-in magic commands*. URL: https://ipython.readthedocs.io/en/stable/interactive/magics.html.

VanderPlas, J. *The Basics of NumPy Arrays*. URL: https://jakevdp.github.io/PythonDataScienceHandbook/02.02-the-basics-of-numpy-arrays.html.

w3schools. *Python String Formatting: Multiple Values*. URL: https://www.w3schools.com/python/python_string_formatting.asp.

— *Python Tutorial*. URL: https://www.w3schools.com/python/.

Weideman, J. A. C. "Numerical Integration of Periodic Functions: A Few Examples". In: *The American Mathematical Monthly* 109.1 (2002), pp. 21–36. DOI: 10.2307/2695765. URL: http://www.jstor.org/stable/2695765.

Wikipedia. *Cauchy–Schwarz inequality*. URL: https://en.wikipedia.org/wiki/CauchySchwarz_inequality#Proofs.

— *Central limit theorem: generalized theorem*. URL: https://en.wikipedia.org/wiki/Central_limit_theorem#Generalized_theorem.

— *Exponent bias*. URL: https://en.wikipedia.org/wiki/Exponent_bias.

Wikipedia. *Hadamard product (matrices)*. URL: https://en.wikipedia.org/wiki/Hadamard_product_(matrices).

— *Hypergeometric function*. URL: https://en.wikipedia.org/wiki/Hypergeometric_function.

Wilkinson, J. H. *The Algebraic Eigenvalue Problem*. Oxford University Press, 1988.

13 Index

periodic boundary conditions, 75
pointer, 9
polynomial, 26, 27, 31–33
position representation, 70
printoptions, 13, 14, 24
probability density function, 41–43
projection matrix, 56
pseudo random numbers, *see* pseudo random numbers
bers
pyplot, 23, 25, 30, 43, 73
Python integer, 19

quadratic convergence, 21
quadratic equation, 18

random number generator, 25
random numbers, 24, 46
random.normal, 47
recursion, 20, 23–25
regula falsi, 26
rejection method, 48, 50
repeated Richardson extrapolation, 37
representation, 61
resolution of the identity, 58
rest term, 40
Richardson extrapolation, 37–40
Riemann sum, 65
Riemann zeta function, 40

SageMath, 17
savefig, 23, 30, 73
scope, 10
script, 12
secant, 26
secant method, 27
secular equation, 56
seed, 24
sesquilinearity, 53
Shanks transformation, 38
shebang, 12
sign bit, 18
spectral decomposition, 58, 61
SSH port forwarding, 78
standard deviation, 41, 42
standard normal distribution, 42–44, 47
steepest decent, 29
stochastic independence, 42
symbolic computer mathematics, 17
symmetric matrix, 54
SymPy, 17
system time, 24

Taylor series, 21, 32, 42
%timeit, 61, 74
timing, 14
tranformation method, 49
transformation method, 48–50
transpose, 53
transposition, 7
trapezoidal rule, 38

triangular distribution, 50
truncation, 8
truncation errors, 8
type, 9
types, 8

U(0,1), 24, 61
unbiased estimator, 45
uncertainty, 46
unitary matrix, 54

value, 9
variance, 41–43, 45
vectorize, 58
vibrating sting, 67

zeroth moment, 41