Pierre Navaro

Scientific computing with Python

for beginners

Publisher

Contents

Chapter 1

Introduction

1.1 History

- Project initiated by Guido Von Rossum in 1990
- Interpreted language written in C.
- Widely used in all domains (Web, Data Science, Scientific Computation).
- This is a high level language with a simple syntax.
- Python types are numerously and powerful.
- Bind Python with other languages is easy.
- You can perform a lot of operations with very few lines.
- Available on all platforms Unix, Windows, Mac OS X...
- Many libraries offer Python bindings.
- Python 2 is retired use only Python 3

1.2 Python distributions

Python packages are available with all linux distributions but you can get standalone bundles:

- Anaconda
- Enthought Tools Suite

- Astropy
- SAGEMATH
- Pyzo

1.3 Performances

Python is not fast... but: - Sometimes it is. - Most of operations are optimized. - Package like numpy can reduce the CPU time. - With Python you can save time to achieve your project.

Some advices: - Write your program with Python language. - If it is fast enough, be happy. - After profiling, optimize costly parts of your code.

"Premature optimization is the root of all evil" (Donald Knuth 1974)

1.4 Jupyter - Start The Notebook

Open the notebook

```
git clone https::://github.com/pnavaro/python-notebooks
cd python-notebooks/notebooks
jupyter notebook
```

You should see the notebook open in your browser. If not, go to http://localhost:8888

The Jupyter Notebook is an interactive environment for writing and running code. The notebook is capable of running code in a wide range of languages. However, each notebook is associated with Python3 kernel.

1.5 Code cells allow you to enter and run code

Make a copy of this notebook by using the File menu.

Run a code cell using `Shift-Enter` or pressing the
button in the toolbar above:
There are two other keyboard shortcuts for running code:

- `Alt-Enter` runs the current cell and inserts a new one below.
- `Ctrl-Enter` run the current cell and enters command mode.

1.6 Managing the Kernel

Code is run in a separate process called the Kernel. The Kernel can be interrupted or restarted. Try running the following cell and then hit the
button in the toolbar above.
The "Cell" menu has a number of menu items for running code in different ways. These includes:

- Run and Select Below
- Run and Insert Below
- Run All
- Run All Above
- Run All Below

1.7 Restarting the kernels

The kernel maintains the state of a notebook's computations. You can reset this state by restarting the kernel. This is done by clicking on the
in the toolbar above.
Check the documentation.

1.8 First program

- Print out the string ”Hello world!” and its type.
- Print out the value of `a` variable set to 6625 and its type.

```
In [1]: s = "Hello World!"
        print(type(s),s)

<class 'str'> Hello World!


In [2]: a = 6625
        print(type(a),a)

<class 'int'> 6625


In [3]: # a+s
```

1.9 Execute using python

```
In [4]: %%file hello.py

        s = "Hello World!"
        print(type(s),s)
        a = 6625
        print(type(a),a)

Writing hello.py


$ python3 hello.py
<class 'str'> Hello World!
<class 'int'> 6625
```

1.10 Execute with ipython

```
(my-env) $ ipython
Python 3.6.3 | packaged by conda-forge | (default, Nov  4 2017, 10:13:32)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.2.1 -- An enhanced Interactive Python. Type '?' for help.

In [1]: run hello.py
<class 'str'> Hello World!
<class 'int'> 6625

In [5]: %run hello.py

<class 'str'> Hello World!
<class 'int'> 6625
```

1.11 Python Types

- Most of Python types are classes, typing is dynamic.
- ; symbol can be used to split two Python commands on the same line.

```
In [6]: s = int(2010); print(type(s))
        s = 3.14; print(type(s))
        s = True; print(type(s))
        s = None; print(type(s))
        s = 1.0j; print(type(s))
        s = type(type(s)); print(type(s))
```

```
<class 'int'>
<class 'float'>
<class 'bool'>
<class 'NoneType'>
<class 'complex'>
<class 'type'>
```

1.12 Calculate with Python

```
In [7]: x = 45        # This is a comment!
        x += 2        # equivalent to x = x + 2
        print(x, x > 45)
```

```
47 True
```

```
In [8]: y = 2.5
        print("x+y=",x+y, type(x+y))  # Add float to integer, result will be a float
```

```
x+y= 49.5 <class 'float'>
```

```
In [9]: print(x*10/y)   # true division returns a float
        print(x*10//3)  # floor division discards the fractional part
```

```
188.0
156
```

```
In [10]: print( x % 8) # the % operator returns the remainder of the division
```

```
7
```

```
In [11]: print( f" x = {x:05d} ") # You can use C format rules to improve print output
```

```
 x = 00047
```

1.13 Multiple Assignment

- Variables can simultaneously get new values.
- Expressions on the right-hand side are all evaluated first before assignments take place.
- The right-hand side expressions are evaluated from the left to the right.

- Use it very carefully

```
In [12]: a = b = c = 1
         print(a, b, c)
```

```
1 1 1
```

```
In [13]: a, b, c = 1, 2, 3
         print (a, b, c)
```

```
1 2 3
```

```
In [14]: a, c = c, a     # Nice way to permute values
         print (a, b, c)
```

```
3 2 1
```

```
In [15]: a < b < c, a > b > c
```

```
Out[15]: (False, True)
```

1.14 input Function

- Value returned by input is a string.
- You must cast input call to get the type you want.

  ```
  name = input("Please enter your name: ")
  x = int(input("Please enter an integer: "))
  L = list(input("Please enter 3 integers "))
  ```

  Copy-pase code above in three different cells and print returned values.

Chapter 2

Strings

```
In [1]: word = "bonjour"
```

```
In [2]: print(word, len(word))
```

```
bonjour 7
```

Add a `.` to the variable and then press `<TAB>` to get all attached methods available.

```
In [3]: word.capitalize()
```

```
Out[3]: 'Bonjour'
```

After choosing your method, press `shift+<TAB>` to get interface.

```
In [4]: word.upper()
```

```
Out[4]: 'BONJOUR'
```

```
In [5]: help(word.replace) # or word.replace?
```

```
Help on built-in function replace:

replace(old, new, count=-1, /) method of builtins.str instance
    Return a copy with all occurrences of substring old replaced by new.

      count
        Maximum number of occurrences to replace.
        -1 (the default value) means replace all occurrences.

    If the optional argument count is given, only the first count occurrences are
    replaced.
```

```
In [6]: word.replace('o','O',1)
```

```
Out[6]: 'bOnjour'
```

2.1 Strings and `print` Function

Strings can be enclosed in single quotes ('...') or double quotes ("...") with the same result. can be used to escape quotes:

```
In [7]: print('spam eggs')          # single quotes
        print('doesn\'t')           # use \' to escape the single quote...
        print("doesn't")            # ...or use double quotes instead
        print('"Yes," he said.')    #
        print("\"Yes,\" he said.")
        print('"Isn\'t," she said.')
```

```
spam eggs
doesn't
doesn't
"Yes," he said.
"Yes," he said.
"Isn't," she said.
```

`print` function translates C special characters

```
In [8]: s = '\tFirst line.\nSecond line.'  # \n means newline \t inserts tab
        print(s)  # with print(), \n produces a new line
        print(r'\tFirst line.\nSecond line.')  # note the r before the quote
```

```
        First line.
Second line.
\tFirst line.\nSecond line.
```

2.2 String literals with multiple lines

```
In [9]: print("""\
        Usage: thingy [OPTIONS]
             -h                        Display this usage message
             -H hostname               Hostname to connect to
        """)
```

```
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
```

character removes the initial newline.

Strings can be concatenated (glued together) with the + operator, and repeated with *

```
In [10]: 3 * ("Re" + 2 * 'n' + 'es ')
```

```
Out[10]: 'Rennes Rennes Rennes '
```

Two or more string literals next to each other are automatically concatenated.

```
In [11]: text = ('Put several strings within parentheses '
                 'to have them joined together.')
         text
```

```
Out[11]: 'Put several strings within parentheses to have them joined together.'
```

Strings can be indexed, with the first character having index 0. There is no separate character type; a character is simply a string of size one

```
In [12]: word = 'Python @ ENSAI'
         print(word[0]) # character in position 0
         print(word[5]) # character in position 5
```

```
P
n
```

Indices may also be negative numbers, to start counting from the right

```
In [13]: print(word[-1])  # last character
         print(word[-2])  # second-last character
```

```
I
A
```

2.3 Slicing Strings

- Omitted first index defaults to zero,
- Omitted second index defaults to the size of the string being sliced.
- Step can be set with the third index

```
In [14]: print(word[:2])  # character from the beginning to position 2 (excluded)
         print(word[4:])  # characters from position 4 (included) to the end
         print(word[-2:]) # characters from the second-last (included) to the end
         print(word[::-1]) # This is the reversed string!
```

```
Py
on @ ENSAI
AI
IASNE @ nohtyP
```

```
In [15]: word[::2]
```

```
Out[15]: 'Pto  NA'
```

Python strings cannot be changed — they are immutable. If you need a different string, you should create a new or use Lists.

```
In [16]: word[0] = 'J'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)

<ipython-input-16-91a956888ca7> in <module>
----> 1 word[0] = 'J'


TypeError: 'str' object does not support item assignment
```

```
In [17]: ## Some string methods
         print(word.startswith('P'))
```

```
True
```

```
In [18]: print(*("\n"+w for w in dir(word) if not w.startswith('_')) )
```

```
capitalize
casefold
center
count
encode
endswith
expandtabs
find
format
format_map
index
isalnum
isalpha
isascii
isdecimal
isdigit
isidentifier
islower
isnumeric
isprintable
isspace
istitle
isupper
join
ljust
lower
lstrip
maketrans
partition
replace
rfind
rindex
rjust
rpartition
rsplit
rstrip
split
splitlines
startswith
strip
swapcase
title
translate
upper
zfill
```

2.4 Exercise

- Ask user to input a string.
- Print out the string length.
- Check if the last character is equal to the first character.
- Check if this string contains only letters.
- Check if this string is lower case.
- Check if this string is a palindrome. A palindrome is a word, phrase, number, or other sequence of characters which reads the same backward as forward.

```
In [19]: # %load solutions/strings/demo.py
```

Chapter 3

Python lists and tuples

- List is the most versatile Python data type to group values with others
- Can be written as a list of comma-separated values (items) between square brackets.
- Tuples are written between parenthesis. They are read-only lists.
- Lists can contain items of different types.
- Like strings, lists can be indexed and sliced.
- Lists also support operations like concatenation.

3.1 Indexing

```
In [1]: squares = [1, 4, 9, 16, 25]
        print(squares)
```

```
[1, 4, 9, 16, 25]
```

```
In [2]: print(squares[0])  # indexing returns the item
```

```
1
```

```
In [3]: squares[-1]
```

```
Out[3]: 25
```

```
In [4]: squares[-3:] # slicing returns a new list
```

```
Out[4]: [9, 16, 25]
```

```
In [5]: squares += [36, 49, 64, 81, 100]
        print(squares)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

- Unlike strings, which are immutable, lists are a mutable type.

```
In [6]: cubes = [1, 8, 27, 65, 125]  # something's wrong here
        cubes[3] = 64  # replace the wrong value, the cube of 4 is 64, not 65!
        print(cubes)
```

```
[1, 8, 27, 64, 125]
```

```
In [7]: cubes.append(216)  # add the cube of 6
        print(cubes)
```

```
[1, 8, 27, 64, 125, 216]
```

```
In [8]: cubes.remove(1)
        print(cubes)
```

```
[8, 27, 64, 125, 216]
```

3.2 Assignment

- You can change the size of the list or clear it entirely.
- The built-in function len() returns list size.
- It is possible to create lists containing other lists.

```
In [9]: letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
        letters[2:5] = ['C', 'D', 'E'] # replace some values
        print(letters)
```

```
['a', 'b', 'C', 'D', 'E', 'f', 'g']
```

```
In [10]: letters[2:5] = [] # now remove them
         print(letters)
```

```
['a', 'b', 'f', 'g']
```

```
In [11]: a = ['a', 'b', 'c']
         n = [1, 2, 3]
         x = [a, n]
```

```
In [12]: x
```

```
Out[12]: [['a', 'b', 'c'], [1, 2, 3]]
```

```
In [13]: x[0]
```

```
Out[13]: ['a', 'b', 'c']
```

```
In [14]: x[0][1], len(x)
```

```
Out[14]: ('b', 2)
```

3.3 Assignment, Copy and Reference

```
In [15]: a = [0, 1, 2, 3, 4]
         b = a
         print("b = ",b)
```

```
b =  [0, 1, 2, 3, 4]
```

```
In [16]: b[1] = 20        # Change one value in b
         print("a = ",a) # Y
```

```
a =  [0, 20, 2, 3, 4]
```

b is a reference to a, they occupy same space memory

```
In [17]: b = a[:] # assign a slice of a and you create a new list
         b[2] = 10
         print("b = ",b)
         print("a = ",a)
```

```
b =  [0, 20, 10, 3, 4]
a =  [0, 20, 2, 3, 4]
```

3.4 Some useful List Methods

```
In [18]: a = list("Python-2020")
         a
```

```
Out[18]: ['P', 'y', 't', 'h', 'o', 'n', '-', '2', '0', '2', '0']
```

```
In [19]: a.sort()
         a
```

```
Out[19]: ['-', '0', '0', '2', '2', 'P', 'h', 'n', 'o', 't', 'y']
```

```
In [20]: a.reverse()
         a
```

```
Out[20]: ['y', 't', 'o', 'n', 'h', 'P', '2', '2', '0', '0', '-']
```

```
In [21]: a.pop() #pop the last item and remove it from the list
         a
```

```
Out[21]: ['y', 't', 'o', 'n', 'h', 'P', '2', '2', '0', '0']
```

3.5 Dictionary

They are indexed by keys, which are often strings.

```
In [22]: person = dict(firstname="John", lastname="Smith", email="john.doe@domain.fr")
         person['size'] = 1.80
         person['weight'] = 70
```

```
In [23]: person
```

```
Out[23]: {'firstname': 'John',
          'lastname': 'Smith',
          'email': 'john.doe@domain.fr',
          'size': 1.8,
          'weight': 70}
```

```
In [24]: print(person.keys())
```

```
dict_keys(['firstname', 'lastname', 'email', 'size', 'weight'])
```

```
In [25]: print(person.items())
```

```
dict_items([('firstname', 'John'), ('lastname', 'Smith'), ('email', 'john.doe@domain.fr'), ('size', 1.8
```

3.6 Exercises

- Split the string ”python ENSAI 2020” into the list [”python”,”ENSAI”, 2020]
- Insert ”september” and value 7 before 2020 in the result list.
- Capitalize the first item to ”Python”
- Create a dictionary with following keys (meeting, month, day, year)
- Print out the items.
- Append the key ”place” to this dictionary and set the value to ”ENSAI”. `python ['python', 'ENSAI', '2020'] ['python', 'ENSAI', 'september', 7, '2020'] ['Python', 'ENSAI', 'september', 7, '2020'] {'course': 'Python','september': 'september','day': 7,'year': '2020','place': 'ENSAI'}`

Chapter 4

Control Flow Tools

4.1 While loop

- Don't forget the ':' character.
- The body of the loop is indented

```
In [1]: # Fibonacci series:
        # the sum of two elements defines the next
        a, b = 0, 1
        while b < 500:
            a, b = b, a+b
            print(round(b/a,5), end=",")
```

```
1.0,2.0,1.5,1.66667,1.6,1.625,1.61538,1.61905,1.61765,1.61818,1.61798,1.61806,1.61803,1.61804,
```

4.2 `if` Statements

`True`, `False`, `and`, `or`, `not`, `==`, `is`, `!=`, `is not`, `>`, `>=`, `<`, `<=`

```
In [2]: x = 42
        if x < 0:
            x = 0
            print('Negative changed to zero')
        elif x == 0:
            print('Zero')
        elif x == 1:
            print('Single')
        else:
            print('More')
```

```
More
```

switch or case statements don't exist in Python.

4.2.1 Exercise Collatz conjecture

Consider the following operation on an arbitrary positive integer: - If the number is even, divide it by two. - If the number is odd, triple it and add one.

The conjecture is that no matter what initial value of this integer, the sequence will always reach 1. - Test the Collatz conjecture for n = 100000. - How many steps do you need to reach 1 ?

4.3 Loop over an iterable object

We use for statement for looping over an iterable object. If we use it with a string, it loops over its characters.

```
In [3]: for c in "python":
            print(c)
```

```
p
y
t
h
o
n
```

```
In [4]: for word in "Python ENSAI september 7th 2020".split(" "):
            print(word, len(word))
```

```
Python 6
ENSAI 5
september 9
7th 3
2020 4
```

4.3.1 Exercise: Anagram

An anagram is word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.

Write a code that print True if s1 is an anagram of s2. To do it, remove every character present in both strings. Check you obtain two empty strings.

Hint: `s = s.replace(c,"",1)` removes the character `c` in string `s` one time.

```
s1 = "pascal obispo"
s2 = "pablo picasso"
..
True
```

4.4 Loop with range function

- It generates arithmetic progressions
- It is possible to let the range start at another number, or to specify a different increment.
- Since Python 3, the object returned by `range()` doesn't return a list to save memory space. `xrange` no longer exists.
- Use function list() to creates it.

```
In [5]: list(range(5))
```

```
Out[5]: [0, 1, 2, 3, 4]
```

```
In [6]: list(range(2, 5))
```

```
Out[6]: [2, 3, 4]
```

```
In [7]: list(range(-1, -5, -1))
```

```
Out[7]: [-1, -2, -3, -4]
```

```
In [8]: for i in range(5):
            print(i, end=' ')
```

```
0 1 2 3 4
```

4.4.1 Exercise Exponential

- Write some code to compute the exponential mathematical constant $e \simeq 2.718281828459045$ using the taylor series developed at 0 and without any import of external modules:

$$e \simeq \sum_{n=0}^{50} \frac{1}{n!}$$

4.5 `break` Statement.

```
In [9]: for n in range(2, 10):     # n = 2,3,4,5,6,7,8,9
            for x in range(2, n):  # x = 2, ..., n-1
                if n % x == 0:     # Return the division remain (mod)
                    print(n, " = ", x, "*", n//x)
                    break
                else:
                    print("%d is a prime number" % n)
                    break
```

```
3 is a prime number
4  =  2 * 2
5 is a prime number
6  =  2 * 3
7 is a prime number
8  =  2 * 4
9 is a prime number
```

4.5.1 `iter` Function

```
In [10]: course = """ Python september 7, 14 2020 ENSAI Rennes """.split()
         print(course)
```

```
['Python', 'september', '7,', '14', '2020', 'ENSAI', 'Rennes']
```

```
In [11]: iterator = iter(course)
         print(iterator.__next__())
```

```
Python
```

```
In [12]: print(iterator.__next__())
```

```
september
```

4.6 Defining Function: `def` statement

```
In [13]: def is_palindromic(s):
             "Return True if the input sequence is a palindrome"
             return s == s[::-1]


         is_palindromic("kayak")
```

```
Out[13]: True
```

- Body of the function start must be indented
- Functions without a return statement do return a value called `None`.

```
In [14]: def fib(n):
             """Print a Fibonacci series up to n."""
             a, b = 0, 1
             while a < n:
                  print(a, end=' ')  # the end optional argument is \n by default
                  a, b = b, a+b
             print("\n") # new line

         result = fib(2000)
         print(result) # is None
```

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597

None
```

4.7 Documentation string

- It's good practice to include docstrings in code that you write, so make a habit of it.

```
In [15]: def my_function( foo):
              """Do nothing, but document it.

              No, really, it doesn't do anything.
              """
              pass

         print(my_function.__doc__)
```

```
Do nothing, but document it.

     No, really, it doesn't do anything.
```

```
In [16]: help(my_function)
```

```
Help on function my_function in module __main__:

my_function(foo)
    Do nothing, but document it.

    No, really, it doesn't do anything.
```

4.8 Default Argument Values

```
In [17]: def f(a,b=5):
             return a+b

         print(f(1))
         print(f(b="a",a="bc"))
```

```
6
bca
```

Important warning: The default value is evaluated only once.

```
In [18]: def f(a, L=[]):
             L.append(a)
             return L

         print(f(1))
```

```
[1]
```

```
In [19]: print(f(2)) # L = [1]
```

```
[1, 2]
```

```
In [20]: print(f(3)) # L = [1,2]
```

```
[1, 2, 3]
```

4.9 Function Annotations

Completely optional metadata information about the types used by user-defined functions. These type annotations conforming to PEP 484 could be statically used by MyPy.

```
In [21]: def f(ham: str, eggs: str = 'eggs') -> str:
              print("Annotations:", f.__annotations__)
              print("Arguments:", ham, eggs)
              return ham + ' and ' + eggs

         f('spam')
         help(f)
         print(f.__doc__)
```

```
Annotations: {'ham': <class 'str'>, 'eggs': <class 'str'>, 'return': <class 'str'>}
Arguments: spam eggs
Help on function f in module __main__:

f(ham: str, eggs: str = 'eggs') -> str

None
```

4.10 Arbitrary Argument Lists

Arguments can be wrapped up in a tuple or a list with form *args

```
In [22]: def f(*args, sep=" "):
             print (args)
             return sep.join(args)

         print(f("big","data"))
```

```
('big', 'data')
big data
```

- Normally, these variadic arguments will be last in the list of formal parameters.
- Any formal parameters which occur after the *args parameter are 'keyword-only' arguments.

4.11 Keyword Arguments Dictionary

A final formal parameter of the form **name receives a dictionary.

```
In [23]: def add_contact(kind, *args, **kwargs):
             print(args)
             print("-" * 40)
             for key, value in keywords.items():
                 print(key, ":", value)
```

*name must occur before **name

```
In [24]: add_contact("John", "Smith",
                    phone="555 8765",
                    email="john.smith@python.org")
```

```
('Smith',)
----------------------------------------
```

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

<ipython-input-24-9581700007e6> in <module>
----> 1 add_contact("John", "Smith",
      2             phone="555 8765",
      3             email="john.smith@python.org")


<ipython-input-23-5a2bcf0288b1> in add_contact(kind, *args, **kwargs)
      2     print(args)
      3     print("-" * 40)
----> 4     for key, value in keywords.items():
      5         print(key, ":", value)


NameError: name 'keywords' is not defined
```

4.12 Lambda Expressions

Lambda functions can be used wherever function objects are required.

```
In [25]: f = lambda x : 2 * x + 2
         f(3)
```

```
Out[25]: 8
```

```
In [26]: taxicab_distance = lambda x_a,y_a,x_b,y_b: abs(x_b-x_a)+abs(y_b-y_a)
         print(taxicab_distance(3,4,7,2))
```

```
6
```

lambda functions can reference variables from the containing scope:

```
In [27]: def make_incrementor(n):
             return lambda x: x + n

         f = make_incrementor(42)
         f(0),f(1)
```

```
Out[27]: (42, 43)
```

4.13 Unpacking Argument Lists

Arguments are already in a list or tuple. They can be unpacked for a function call. For instance, the built-in range() function is called with the *-operator to unpack the arguments out of a list:

```
In [28]: def chessboard_distance(x_a, y_a, x_b, y_b):
             """
             Compute the rectilinear distance between
             point (x_a,y_a) and (x_b, y_b)
             """
             return max(abs(x_b-x_a),abs(y_b-y_a))

         coordinates = [3,4,7,2]
         chessboard_distance(*coordinates)
```

```
Out[28]: 4
```

In the same fashion, dictionaries can deliver keyword arguments with the **-operator:

```
In [29]: def parrot(voltage, state='a stiff', action='voom'):
              print("-- This parrot wouldn't", action, end=' ')
              print("if you put", voltage, "volts through it.", end=' ')
              print("E's", state, "!")

         d = {"voltage": "four million", "state": "bleedin' demised", "action": "VOOM"}
         parrot(**d)
```

```
-- This parrot wouldn't VOOM if you put four million volts through it. E's bleedin' demised !
```

4.14 Exercise: Time converter

Write 3 functions to manipulate hours and minutes : - Function minutes return minutes from (hours, minutes). - Function hours the inverse function that return (hours, minutes) from minutes. - Function add_time to add (hh1,mm1) and (hh2, mm2) two couples (hours, minutes). It takes 2 tuples of length 2 as input arguments and return the tuple (hh,mm).

```
print(minutes(6,15)) # 375
print(minutes(7,46)) # 466
print(add_time((6,15),(7,46)) # (14,01)
```

4.15 Functions Scope

- All variable assignments in a function store the value in the local symbol table.
- Global variables cannot be directly assigned a value within a function (unless named in a global statement).
- The value of the function can be assigned to another name which can then also be used as a function.

```
In [30]: pi = 1.
         def deg2rad(theta):
             pi = 3.14
             return theta * pi / 180.

         print(deg2rad(45))
         print(pi)
```

```
0.785
1.0
```

```
In [31]: def rad2deg(theta):
             return theta*180./pi

         print(rad2deg(0.785))
         pi = 3.14
         print(rad2deg(0.785))
```

```
141.3
45.0
```

```
In [32]: def deg2rad(theta):
             global pi
             pi = 3.14
             return theta * pi / 180

         pi = 1
         print(deg2rad(45))
```

```
0.785
```

```
In [33]: print(pi)
```

```
3.14
```

4.16 `enumerate` Function

```
In [34]: primes =  [1,2,3,5,7,11,13]
         for idx, ele in enumerate (primes):
             print(idx, " --- ", ele)
```

```
0  ---  1
1  ---  2
2  ---  3
3  ---  5
4  ---  7
5  ---  11
6  ---  13
```

4.16.1 Exercise: Caesar cipher

In cryptography, a Caesar cipher, is one of the simplest and most widely known encryption techniques. It is a type of substitution cipher in which each letter in the plaintext is replaced by a letter some fixed number of positions down the alphabet. For example, with a left shift of 3, D would be replaced by A, E would become B, and so on.

- Create a function `cipher` that take the plain text and the key value as arguments and return the encrypted text.
- Create a funtion `plain` that take the crypted text and the key value as arguments that return the deciphered text.

4.17 `zip` Builtin Function

Loop over sequences simultaneously.

```
In [35]: L1 = [1, 2, 3]
         L2 = [4, 5, 6]

         for (x, y) in zip(L1, L2):
             print (x, y, '--', x + y)
```

```
1 4 -- 5
2 5 -- 7
3 6 -- 9
```

4.18 List comprehension

- Set or change values inside a list
- Create list from function

```
In [36]: lsingle = [1, 3, 9, 4]
         ldouble = []
         for k in lsingle:
             ldouble.append(2*k)
         ldouble
```

```
Out[36]: [2, 6, 18, 8]
```

```
In [37]: ldouble = [k*2 for k in lsingle]
```

```
In [38]: [n*n for n in range(1,10)]
```

```
Out[38]: [1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
In [39]: [n*n for n in range(1,10) if n&1]
```

```
Out[39]: [1, 9, 25, 49, 81]
```

```
In [40]: [n+1 if n&1 else n//2 for n in range(1,10) ]
```

```
Out[40]: [2, 1, 4, 2, 6, 3, 8, 4, 10]
```

4.18.1 Exercise

Code a new version of cypher function using list comprehension.

Hints: - `s = ''.join(L)` convert the characters list `L` into a string `s`. - `L.index(c)` return the index position of `c` in list `L` - `"c".islower()` and `"C".isupper()` return `True`

4.19 map built-in function

Apply a function over a sequence.

```
In [41]: res = map(hex,range(16))
         print(res)
```

```
<map object at 0x7fefd46cb5e0>
```

Since Python 3.x, `map` process return an iterator. Save memory, and should make things go faster. Display result by using unpacking operator.

```
In [42]: print(*res)
```

```
0x0 0x1 0x2 0x3 0x4 0x5 0x6 0x7 0x8 0x9 0xa 0xb 0xc 0xd 0xe 0xf
```

4.20 map with user-defined function

```
In [43]: def add(x,y):
             return x+y

         L1 = [1, 2, 3]
         L2 = [4, 5, 6]
         print(*map(add,L1,L2))
```

```
5 7 9
```

- `map` is often faster than `for` loop

```
In [44]: M = range(10000)
         f = lambda x: x**2
         %timeit lmap = list(map(f,M))
```

```
3.3 ms ± 21.1 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
In [45]: M = range(10000)
         f = lambda x: x**2
         %timeit lfor = [f(m) for m in M]
```

```
3.66 ms ± 15.6 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

4.21 filter

creates a iterator of elements for which a function returns `True`.

```
In [46]: number_list = range(-5, 5)
         odd_numbers = filter(lambda x: x & 1 , number_list)
         print(*odd_numbers)
```

```
-5 -3 -1 1 3
```

- As `map`, `filter` is often faster than `for` loop

```
In [47]: M = range(1000)
         f = lambda x: x % 3 == 0
         %timeit lmap = filter(f,M)
```

```
186 ns ± 1.04 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)
```

```
In [48]: M = range(1000)
         %timeit lfor = (m for m in M if m % 3 == 0)
```

```
348 ns ± 3.33 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

4.22 Exercise with map:

Code a new version of your cypher function using map.

Hints: - Applied function must have only one argument, create a function called `shift` with the key value and use map.

4.23 Exercise with filter:

Create a function with a number n as single argument that returns True if n is a Kaprekar number. For example 45 is a Kaprekar number, because

$$45^2 = 2025$$

and

$$20 + 25 = 45$$

Use `filter` to give Kaprekar numbers list lower than 10000.

```
1, 9, 45, 55, 99, 297, 703, 999, 2223, 2728, 4879, 4950, 5050, 5292, 7272, 7777, 9999
```

4.24 Recursive Call

“‘python slideshow={”slide_type”: ”fragment”} def gcd(x, y): """ returns the greatest common divisor.""" if x == 0: return y else : return gcd(y % x, x)

gcd(12,16) “‘

4.25 Exercises

4.25.1 Factorial

- Write the function `factorial` with a recursive call

NB: Recursion is not recommended by Guido.

4.25.2 Minimum number of rooms required for lectures.

Given an array of time intervals (start, end) for classroom lectures (possibly overlapping), find the minimum number of rooms required.

For example, given Input:

```
lectures = ["9:00-10:30", "9:30-11:30","11:00-12:00","14:00-18:00", "15:00-16:00", "15:30-17:30", "16:0
```

should output 3.

4.25.3 Non-palindromic skinny numbers

non-palindromic squares remaining square when written backwards

$$\begin{array}{lcllcl} 10^2 & = & 100 & 01^2 & = & 001 \\ 13^2 & = & 169 & 31^2 & = & 961 \\ 102^2 & = & 10404 & 201^2 & = & 40401 \end{array}$$

4.25.4 Narcissistic number

A number is narcissistic if the sum of its own digits each raised to the power of the number of digits.
Example : $4150 = 4^5 + 1^5 + 5^5 + 0^5$ or $153 = 1^3 + 5^3 + 3^3$
Find narcissitic numbers with 3 digits

4.25.5 Happy number

- Given a number $n = n_0$, define a sequence $n_1, n_2, \ldots$ where n_{i+1} is the sum of the squares of the digits of n_i. Then n is happy if and only if there exists i such that $n_i = 1$.

For example, 19 is happy, as the associated sequence is:

$$\begin{array}{lcllcl} 1^2 & + & 9^2 & = & 82 \\ 8^2 & + & 2^2 & = & 68 \\ 6^2 & + & 8^2 & = & 100 \\ 1^2 & + & 0^2 + 0^2 & = & 1 \end{array}$$

- Write a function `ishappy(n)` that returns True if `n` is happy. - Write a function `happy(n)` that returns a list with all happy numbers $< n$.

```
happy(100) = [1, 7, 10, 13, 19, 23, 28, 31, 32, 44, 49, 68, 70, 79, 82, 86, 91, 94, 97]
```

4.25.6 Longuest increasing subsequence

Given N elements, write a program that prints the length of the longuest increasing subsequence whose adjacent element difference is one.
Examples:

```
a = [3, 10, 3, 11, 4, 5, 6, 7, 8, 12]
Output : 6
Explanation: 3, 4, 5, 6, 7, 8 is the longest increasing subsequence whose adjacent element differs by o
```

```
Input : a = [6, 7, 8, 3, 4, 5, 9, 10]
Output : 5
Explanation: 6, 7, 8, 9, 10 is the longest increasing subsequence
```

4.25.7 Polynomial derivative

- A Polynomial is represented by a Python list of its coefficients. [1,5,-4] => $1 + 5x - 4x^2$
- Write the function diff(P,n) that return the nth derivative Q
- Don't use any external package

```
diff([3,2,1,5,7],2) = [2, 30, 84]
diff([-6,5,-3,-4,3,-4],3) = [-24, 72, -240]
```

Chapter 5

Modules

If your Python program gets longer, you may want to split it into several files for easier maintenance. To support this, Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a module.

Run the cell below to create a file named fibo.py with several functions inside:

```
In [1]: %%file fibo.py
        """ Simple module with
            two functions to compute Fibonacci series """

        def fib1(n):
           """ write Fibonacci series up to n """
           a, b = 0, 1
           while b < n:
              print(b, end=', ')
              a, b = b, a+b

        def fib2(n):
            """ return Fibonacci series up to n """
            result = []
            a, b = 0, 1
            while b < n:
                result.append(b)
                a, b = b, a+b
            return result

        if __name__ == "__main__":
            import sys
            fib1(int(sys.argv[1]))
```

```
Writing fibo.py
```

You can use the function fib by importing fibo which is the name of the file without .py extension.

```
In [2]: import fibo
        print(fibo.__name__)
        print(fibo.__file__)
        fibo.fib1(1000)
```

```
fibo
/home/runner/work/python-notebooks/python-notebooks/notebooks/fibo.py
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987,
```

```
In [3]: %run fibo.py 1000

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987,

In [4]: help(fibo)

Help on module fibo:

NAME
    fibo

DESCRIPTION
    Simple module with
    two functions to compute Fibonacci series

FUNCTIONS
    fib1(n)
        write Fibonacci series up to n

    fib2(n)
        return Fibonacci series up to n

FILE
    /home/runner/work/python-notebooks/python-notebooks/notebooks/fibo.py
```

5.1 Executing modules as scripts

When you run a Python module with

```
$ python fibo.py <arguments>
```

the code in the module will be executed, just as if you imported it, but with the **name** set to "**main**". The following code will be executed only in this case and not when it is imported.

```
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

In Jupyter notebook, you can run the fibo.py python script using magic command.

```
In [5]: %run fibo.py 1000

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987,
```

The module is also imported.

```
In [6]: fib1(1000)

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987,
```

5.2 Different ways to import a module

```
import fibo
import fibo as f
from fibo import fib1, fib2
from fibo import *
```

- Last command with '*' imports all names except those beginning with an underscore (_). In most cases, do not use this facility since it introduces an unknown set of names into the interpreter, possibly hiding some things you have already defined.
- If a function with same name is present in different modules imported. Last module function imported replace the previous one.

```
In [7]: from numpy import sqrt
        from scipy import sqrt
        sqrt(-1)
```

```
<ipython-input-7-f3f47bc91153>:3: DeprecationWarning: scipy.sqrt is deprecated and will be removed in S
  sqrt(-1)
```

```
Out[7]: 1j
```

```
In [8]: from scipy import sqrt
        from numpy import sqrt
        sqrt(-1)
```

```
<ipython-input-8-8a25f477b688>:3: RuntimeWarning: invalid value encountered in sqrt
  sqrt(-1)
```

```
Out[8]: nan
```

```
In [9]: import numpy as np
        import scipy as sp

        print(np.sqrt(-1+0j), sp.sqrt(-1))
```

```
1j 1j
```

```
<ipython-input-9-235de4d5ffbb>:4: DeprecationWarning: scipy.sqrt is deprecated and will be removed in S
  print(np.sqrt(-1+0j), sp.sqrt(-1))
```

- For efficiency reasons, each module is only imported once per interpreter session. Therefore, if you change your modules, you must restart the interpreter – If you really want to test interactively after a long run, use :

  ```
  import importlib
  importlib.reload(modulename)
  ```

5.3 The Module Search Path

When a module is imported, the interpreter searches for a file named module.py in a list of directories given by the variable sys.path. - Python programs can modify sys.path - export the PYTHONPATH environment variable to change it on your system.

```
In [10]: import sys
         sys.path

Out[10]: ['/home/runner/work/python-notebooks/python-notebooks/notebooks',
          '/usr/share/miniconda3/envs/runenv/lib/python38.zip',
          '/usr/share/miniconda3/envs/runenv/lib/python3.8',
          '/usr/share/miniconda3/envs/runenv/lib/python3.8/lib-dynload',
          '',
          '/usr/share/miniconda3/envs/runenv/lib/python3.8/site-packages',
          '/usr/share/miniconda3/envs/runenv/lib/python3.8/site-packages/IPython/extensions',
          '/home/runner/.ipython']

In [11]: import collections
         collections.__path__

Out[11]: ['/usr/share/miniconda3/envs/runenv/lib/python3.8/collections']
```

`sys.path` is a list and you can append some directories:

```
In [12]: sys.path.append("/Users/navaro/python-notebooks/")
         print(sys.path)

['/home/runner/work/python-notebooks/python-notebooks/notebooks', '/usr/share/miniconda3/envs/runenv/li
```

When you import a module `foo`, following files are searched in this order:

- **foo.dll**, **foo.dylib** or **foo.so**
- **foo.py**
- **foo.pyc**
- **foo/___init___.py**

5.4 Packages

- A package is a directory containing Python module files.
- This directory always contains a file name ___init___.py

cluster `__init__.py`

5.5 Relative imports

These imports use leading dots to indicate the current and parent packages involved in the relative import.
In the sugiton module, you can use:

```
from . import cluster # import module in the same directory
from .. import base   # import module in parent directory
from ..ensemble import _forest # import module in another subdirectory of the parent directory
```

5.6 Reminder

Don't forget that importing * is not recommended

```
In [13]: sum(range(5),-1)

Out[13]: 9
```

```
In [14]: from numpy import *
         sum(range(5),-1)

Out[14]: 10

In [15]: del sum # delete imported sum function from numpy
         help(sum)

Help on built-in function sum in module builtins:

sum(iterable, /, start=0)
    Return the sum of a 'start' value (default: 0) plus an iterable of numbers

    When the iterable is empty, return the start value.
    This function is intended specifically for use with numeric values and may
    reject non-numeric types.


In [16]: import numpy as np
         help(np.sum)

Help on function sum in module numpy:

sum(a, axis=None, dtype=None, out=None, keepdims=<no value>, initial=<no value>, where=<no value>)
    Sum of array elements over a given axis.

    Parameters
    ----------
    a : array_like
        Elements to sum.
    axis : None or int or tuple of ints, optional
        Axis or axes along which a sum is performed.  The default,
        axis=None, will sum all of the elements of the input array.  If
        axis is negative it counts from the last to the first axis.

        .. versionadded:: 1.7.0

        If axis is a tuple of ints, a sum is performed on all of the axes
        specified in the tuple instead of a single axis or all the axes as
        before.
    dtype : dtype, optional
        The type of the returned array and of the accumulator in which the
        elements are summed.  The dtype of `a` is used by default unless `a`
        has an integer dtype of less precision than the default platform
        integer.  In that case, if `a` is signed then the platform integer
        is used while if `a` is unsigned then an unsigned integer of the
        same precision as the platform integer is used.
    out : ndarray, optional
        Alternative output array in which to place the result. It must have
        the same shape as the expected output, but the type of the output
        values will be cast if necessary.
    keepdims : bool, optional
        If this is set to True, the axes which are reduced are left
        in the result as dimensions with size one. With this option,
        the result will broadcast correctly against the input array.
```

```
    If the default value is passed, then `keepdims` will not be
    passed through to the `sum` method of sub-classes of
    `ndarray`, however any non-default value will be.  If the
    sub-class' method does not implement `keepdims` any
    exceptions will be raised.
initial : scalar, optional
    Starting value for the sum. See `~numpy.ufunc.reduce` for details.

    .. versionadded:: 1.15.0

where : array_like of bool, optional
    Elements to include in the sum. See `~numpy.ufunc.reduce` for details.

    .. versionadded:: 1.17.0

Returns
-------
sum_along_axis : ndarray
    An array with the same shape as `a`, with the specified
    axis removed.   If `a` is a 0-d array, or if `axis` is None, a scalar
    is returned.  If an output array is specified, a reference to
    `out` is returned.

See Also
--------
ndarray.sum : Equivalent method.

add.reduce : Equivalent functionality of `add`.

cumsum : Cumulative sum of array elements.

trapz : Integration of array values using the composite trapezoidal rule.

mean, average

Notes
-----
Arithmetic is modular when using integer types, and no error is
raised on overflow.

The sum of an empty array is the neutral element 0:

>>> np.sum([])
0.0

For floating point numbers the numerical precision of sum (and
``np.add.reduce``) is in general limited by directly adding each number
individually to the result causing rounding errors in every step.
However, often numpy will use a  numerically better approach (partial
pairwise summation) leading to improved precision in many use-cases.
This improved precision is always provided when no ``axis`` is given.
When ``axis`` is given, it will depend on which axis is summed.
Technically, to provide the best speed possible, the improved precision
```

```
is only used when the summation is along the fast axis in memory.
Note that the exact precision may vary depending on other parameters.
In contrast to NumPy, Python's ``math.fsum`` function uses a slower but
more precise approach to summation.
Especially when summing a large number of lower precision floating point
numbers, such as ``float32``, numerical errors can become significant.
In such cases it can be advisable to use `dtype="float64"` to use a higher
precision for the output.

Examples
--------
>>> np.sum([0.5, 1.5])
2.0
>>> np.sum([0.5, 0.7, 0.2, 1.5], dtype=np.int32)
1
>>> np.sum([[0, 1], [0, 5]])
6
>>> np.sum([[0, 1], [0, 5]], axis=0)
array([0, 6])
>>> np.sum([[0, 1], [0, 5]], axis=1)
array([1, 5])
>>> np.sum([[0, 1], [np.nan, 5]], where=[False, True], axis=1)
array([1., 5.])

If the accumulator is too small, overflow occurs:

>>> np.ones(128, dtype=np.int8).sum(dtype=np.int8)
-128

You can also start the sum with a value other than zero:

>>> np.sum([10], initial=5)
15
```

Chapter 6

Input and Output

- str() function return human-readable representations of values.
- repr() generate representations which can be read by the interpreter.
- For objects which don't have a particular representation for human consumption, str() will return the same value as repr().

```
In [1]: s = 'Hello, world.'
        str(s)
```

```
Out[1]: 'Hello, world.'
```

```
In [2]: l = list(range(4))
        str(l)
```

```
Out[2]: '[0, 1, 2, 3]'
```

```
In [3]: repr(s)
```

```
Out[3]: "'Hello, world.'"
```

```
In [4]: repr(l)
```

```
Out[4]: '[0, 1, 2, 3]'
```

```
In [5]: x = 10 * 3.25
        y = 200 * 200
        s = 'The value of x is ' + str(x) + ', and y is ' + repr(y) + '...'
        print(s)
```

```
The value of x is 32.5, and y is 40000…
```

repr() of a string adds string quotes and backslashes:

```
In [6]: hello = 'hello, world\n'
        hellos = repr(hello)
        hellos
```

```
Out[6]: "'hello, world\\n'"
```

The argument to repr() may be any Python object:

```
In [7]: repr((x, y, ('spam', 'eggs')))
```

```
Out[7]: "(32.5, 40000, ('spam', 'eggs'))"
```

```
In [8]: n = 7
        for x in range(1, n):
            for i in range(n):
                print(repr(x**i).rjust(i+2), end=' ') # rjust or center can be used
            print()
```

```
 1   1    1     1      1       1        1
 1   2    4     8     16      32       64
 1   3    9    27     81     243      729
 1   4   16    64    256    1024     4096
 1   5   25   125    625    3125    15625
 1   6   36   216   1296    7776    46656
```

```
In [9]: for x in range(1, n):
            for i in range(n):
                print("%07d" % x**i, end=' ')  # old C format
            print()
```

```
0000001 0000001 0000001 0000001 0000001 0000001 0000001
0000001 0000002 0000004 0000008 0000016 0000032 0000064
0000001 0000003 0000009 0000027 0000081 0000243 0000729
0000001 0000004 0000016 0000064 0000256 0001024 0004096
0000001 0000005 0000025 0000125 0000625 0003125 0015625
0000001 0000006 0000036 0000216 0001296 0007776 0046656
```

6.1 Usage of the `str.format()` method

```
In [10]: print('We are at the {} in {}!'.format('ENSAI', 'Rennes'))
```

```
We are at the ENSAI in Rennes!
```

```
In [11]: print('From {0} to  {1}'.format('September 7', 'September 14'))
```

```
From September 7 to  September 14
```

```
In [12]: print('It takes place at {place}'.format(place='Milon room'))
```

```
It takes place at Milon room
```

```
In [13]: import math
         print('The value of PI is approximately {:.7g}.'.format(math.pi))
```

```
The value of PI is approximately 3.141593.
```

6.2 Formatted string literals (Python 3.6)

```
In [14]: print(f'The value of PI is approximately {math.pi:.4f}.')
```

```
The value of PI is approximately 3.1416.
```

```
In [15]: name = "Fred"
         print(f"He said his name is {name}.")
         print(f"He said his name is {name!r}.")
```

```
He said his name is Fred.
He said his name is 'Fred'.
```

```
In [16]: f"He said his name is {repr(name)}."  # repr() is equivalent to !r
```

```
Out[16]: "He said his name is 'Fred'."
```

```
In [17]: width, precision = 10, 4
         value = 12.34567
         print(f"result: {value:{width}.{precision}f}")  # nested fields
```

```
result:    12.3457
```

```
In [18]: from datetime import *
         today = datetime(year=2017, month=1, day=27)
         print(f"{today:%B %d, %Y}")  # using date format specifier
```

```
January 27, 2017
```

6.3 Reading and Writing Files

`open()` returns a file object, and is most commonly used with file name and accessing mode argument.

```
In [19]: f = open('workfile.txt', 'w')
         f.write("1. This is a txt file.\n")
         f.write("2. \\n is used to begin a new line")
         f.close()
         !cat workfile.txt
```

```
1. This is a txt file.
2. \n is used to begin a new line
```

`mode` can be : - 'r' when the file will only be read, - 'w' for only writing (an existing file with the same name will be erased) - 'a' opens the file for appending; any data written to the file is automatically added to the end. - 'r+' opens the file for both reading and writing. - The mode argument is optional; 'r' will be assumed if it's omitted. - Normally, files are opened in text mode. - 'b' appended to the mode opens the file in binary mode.

```
In [20]: with open('workfile.txt') as f:
             read_text = f.read()
         f.closed
```

```
Out[20]: True
```

```
In [21]: read_text
```

```
Out[21]: '1. This is a txt file.\n2. \\n is used to begin a new line'
```

```
In [22]: lines= []
         with open('workfile.txt') as f:
             lines.append(f.readline())
             lines.append(f.readline())
             lines.append(f.readline())

         lines
```

```
Out[22]: ['1. This is a txt file.\n', '2. \\n is used to begin a new line', '']
```

- `f.readline()` returns an empty string when the end of the file has been reached.
- `f.readlines()` or `list(f)` read all the lines of a file in a list.

For reading lines from a file, you can loop over the file object. This is memory efficient, fast, and leads to simple code:

```
In [23]: with open('workfile.txt') as f:
             for line in f:
                 print(line, end='')
```

```
1. This is a txt file.
2. \n is used to begin a new line
```

6.3.1 Exercise: Wordcount Example

WordCount is a simple application that counts the number of occurrences of each word in a given input set.

- Use lorem module to write a text in the file "sample.txt"
- Write a function `words` with file name as input that returns a sorted list of words present in the file.
- Write the function `reduce` to read the results of words and sum the occurrences of each word to a final count, and then output the results as a dictionary `{word1:occurences1, word2:occurences2}`.
- You can check the results using piped shell commands:

  ```
  cat sample.txt | fmt -1 | tr [:upper:] [:lower:] | tr -d '.' | sort | uniq -c
  ```

```
In [24]: from lorem import text

         text()
```

```
Out[24]: 'Magnam eius ipsum est. Dolor etincidunt neque amet voluptatem sed. Non ut ipsum aliquam etinc
```

```
In [25]: def words( file ):
             """ Parse a file and returns a sorted list of words """
             pass

         words('sample.txt')
         #[('adipisci', 1),
         # ('adipisci', 1),
         # ('adipisci', 1),
         # ('aliquam', 1),
         # ('aliquam', 1),
```

```
In [26]: d = {}
         d['word1'] = 3
         d['word2'] = 2
         d
```

```
Out[26]: {'word1': 3, 'word2': 2}
```

```
In [27]: def reduce ( words ):
             """ Count the number of occurences of a word in list
             and return a dictionary """
             pass

         reduce(words('sample.txt'))
```

```
        #{'neque': 80),
        # 'ut': 80,
        # 'est': 76,
        # 'amet': 74,
        # 'magnam': 74,
        # 'adipisci': 73,
```

6.4 Saving structured data with json

- JSON (JavaScript Object Notation) is a popular data interchange format.
- JSON format is commonly used by modern applications to allow for data exchange.
- JSON can be used to communicate with applications written in other languages.

```
In [28]: import json
         json.dumps([1, 'simple', 'list'])
```

```
Out[28]: '[1, "simple", "list"]'
```

```
In [29]: x = dict(name="Pierre Navaro", organization="CNRS", position="IR")
         with open('workfile.json','w') as f:
             json.dump(x, f)
```

```
In [30]: with open('workfile.json','r') as f:
             x = json.load(f)
         x
```

```
Out[30]: {'name': 'Pierre Navaro', 'organization': 'CNRS', 'position': 'IR'}
```

```
In [31]: %cat workfile.json
```

```
{"name": "Pierre Navaro", "organization": "CNRS", "position": "IR"}
```

Use `ujson` for big data structures https://pypi.python.org/pypi/ujson

For common file formats used in data science (CSV, xls, feather, parquet, ORC, HDF, avro, ...) use packages like pandas or better pyarrow. It depends of what you want to do with your data but Dask and pyspark offer features to read and write (big) data files.

Chapter 7

Errors and Exceptions

There are two distinguishable kinds of errors: *syntax errors* and *exceptions*. - Syntax errors, also known as parsing errors, are the most common. - Exceptions are errors caused by statement or expression syntactically corrects. - Exceptions are not unconditionally fatal.

Exceptions in Python documentation

```
In [1]: 10 * (1/0)
```

```
        ---------------------------------------------------------------------------

        ZeroDivisionError                         Traceback (most recent call last)

        <ipython-input-1-0b280f36835c> in <module>
    ----> 1 10 * (1/0)


        ZeroDivisionError: division by zero
```

```
In [2]: 4 + spam*3
```

```
        ---------------------------------------------------------------------------

        NameError                                 Traceback (most recent call last)

        <ipython-input-2-c98bb92cdcac> in <module>
    ----> 1 4 + spam*3


        NameError: name 'spam' is not defined
```

```
In [3]: '2' + 2
```

```
        ---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-3-d2b23a1db757> in <module>
```

```
----> 1 '2' + 2


TypeError: can only concatenate str (not "int") to str
```

7.1 Handling Exceptions

- In example below, the user can interrupt the program with `Control-C` or the `stop` button in Jupyter Notebook.
- Note that a user-generated interruption is signalled by raising the **KeyboardInterrupt** exception.

```
In [4]: while True:
            try:
              x = int(input("Please enter a number: "))
              print(f' x = {x}')
              break
            except ValueError:
              print("Oops!  That was no valid number.  Try again...")


        ---------------------------------------------------------------------------

        StdinNotImplementedError                  Traceback (most recent call last)

        <ipython-input-4-d9e83eb78fd3> in <module>
          1 while True:
          2     try:
        ----> 3       x = int(input("Please enter a number: "))
          4       print(f' x = {x}')
          5       break


        /usr/share/miniconda3/envs/runenv/lib/python3.8/site-packages/ipykernel/kernelbase.py in raw_inp
        855         """
        856         if not self._allow_stdin:
        --> 857             raise StdinNotImplementedError(
        858                 "raw_input was called, but this frontend does not support input requests."
        859             )


        StdinNotImplementedError: raw_input was called, but this frontend does not support input reques
```

- A try statement may have more than one except clause
- The optional `else` clause must follow all except clauses.

```
In [5]: import sys

        def process_file(file):
            " Read the first line of f and convert to int and check if this integer is positive"
            try:
                i = int(open(file).readline().strip())
                print(i)
                assert i > 0
            except OSError as err:
```

```
            print(f"OS error: {err}")
        except ValueError:
            print("Could not convert data to an integer.")
        except:
            print("Unexpected error:", sys.exc_info()[0])

    # Create the file workfile.txt
    with open('workfile.txt','w') as f:
        f.write("foo")
        f.write("bar")
```

```
In [6]: process_file('workfile.txt') # catch exception return by int() call
```

```
Could not convert data to an integer.
```

```
In [7]: # Change permission of the file, workfile.txt cannot be read
        !chmod u-r workfile.txt
```

```
In [8]: process_file('workfile.txt') # catch exception return by open() call
```

```
OS error: [Errno 13] Permission denied: 'workfile.txt'
```

```
In [9]: # Let's delete the file workfile.txt
        !rm -f workfile.txt
```

```
In [10]: process_file('workfile.txt') # catch another exception return by open() call
```

```
OS error: [Errno 2] No such file or directory: 'workfile.txt'
```

```
In [11]: # Insert the value -1 at the top of workfile.txt
         !echo "-1" > workfile.txt
         %cat workfile.txt
```

```
-1
```

```
In [12]: process_file('workfile.txt') # catch exception return by assert()
```

```
-1
Unexpected error: <class 'AssertionError'>
```

7.2 Raising Exceptions

The raise statement allows the programmer to force a specified exception to occur.

```
In [13]: raise NameError('HiThere')
```

```
        ---------------------------------------------------------------------------

        NameError                                 Traceback (most recent call last)

        <ipython-input-13-72c183edb298> in <module>
    ----> 1 raise NameError('HiThere')


        NameError: HiThere
```

7.3 Defining Clean-up Actions

- The try statement has an optional clause which is intended to define clean-up actions that must be executed under all circumstances.
- A finally clause is always executed before leaving the try statement

```
In [14]: try:
             raise KeyboardInterrupt
         finally:
             print('Goodbye, world!')

Goodbye, world!
```

```
---------------------------------------------------------------------------

KeyboardInterrupt                         Traceback (most recent call last)

<ipython-input-14-7786db0b9fd1> in <module>
  1 try:
----> 2     raise KeyboardInterrupt
  3 finally:
  4     print('Goodbye, world!')


KeyboardInterrupt:
```

7.3.1 Wordcount Exercise

- Improve the function `reduce` to read the results of `words` by using the `KeyError` exception to fill in the dictionary.

Chapter 8

Classes

- Classes provide a means of bundling data and functionality together.
- Creating a new class creates a **new type** of object.
- Assigned variables are new **instances** of that type.
- Each class instance can have **attributes** attached to it.
- Class instances can also have **methods** for modifying its state.
- Python classes provide the class **inheritance** mechanism.

8.1 Use class to store data

- A empty class can be used to bundle together a few named data items.
- You can easily save this class containing your data in JSON file.

```
In [1]: class Car:
            pass

        mycar = Car()  # Create an empty car record

        # Fill the fields of the record
        mycar.brand = 'Peugeot'
        mycar.model = 308
        mycar.year = 2015

In [2]: mycar.__dict__

Out[2]: {'brand': 'Peugeot', 'model': 308, 'year': 2015}
```

8.2 namedtuple

```
In [3]: from collections import namedtuple

        Car = namedtuple('Car', 'brand, model, year')

In [4]: mycar = Car('Peugeot', 308, 2015)
        mycar

Out[4]: Car(brand='Peugeot', model=308, year=2015)

In [5]: mycar.year

Out[5]: 2015
```

```
In [6]: # Like tuples, namedtuples are immutable:
        mycar.model = 3008


        ---------------------------------------------------------------------------

        AttributeError                            Traceback (most recent call last)

        <ipython-input-6-09089c3eb392> in <module>
          1 # Like tuples, namedtuples are immutable:
    ----> 2 mycar.model = 3008


        AttributeError: can't set attribute


In [7]: class Car:

            "A simple example class Animal with its name, weight and age"

            def __init__(self, brand, model, year):  # constructor
                self.brand = brand
                self.model = model
                self.year = year

            def age(self):  # method
                import datetime
                now = datetime.datetime.now()
                return now.year - self.year

In [8]: mycar = Car('Peugeot', 308, 2015) # Instance
        print(f' {mycar.brand} {mycar.model} {mycar.year}')
        print(f' {mycar.age()} years old')

 Peugeot 308 2015
 5 years old


In [9]: mycar.year = 2017
        mycar.age()

Out[9]: 3
```

- `mycar` is an *instance* of Car Class.
- `mycar.age()` is a *method* of `Car` instance `mycar`.
- `brand` and `model` are attributes of `Car` instance `mycar`.

8.3 Convert method to attribute

Use the `property` decorator

```
In [10]: class Car:

             "A simple example class Car with its model, brand and year"

             def __init__(self, brand, model, year):  # constructor
                 self.model = model
```

```
                self.brand = brand
                self.year = year

            @property
            def age(self):  # method
                import datetime
                now = datetime.datetime.now()
                return now.year - self.year
```

```
In [11]: mycar = Car('Peugeot', 308, 2015)
         mycar.age  # age can now be used as an attribute
```

```
Out[11]: 5
```

```
In [12]: mycar.age = 3 # a protected attribute
```

```
         ---------------------------------------------------------------------------

         AttributeError                            Traceback (most recent call last)

         <ipython-input-12-d6fe717cdbfa> in <module>
    ----> 1 mycar.age = 3 # a protected attribute


         AttributeError: can't set attribute
```

8.4 The new Python 3.7 DataClass

```
In [13]: from dataclasses import dataclass

         @dataclass
         class Car:

             brand: str
             model: int
             year: int

             @property
             def age(self) -> int:
                 import datetime
                 now = datetime.datetime.now()
                 return now.year - self.year
```

```
In [14]: mycar = Car('Peugeot', 308, 2015)
         mycar
```

```
Out[14]: Car(brand='Peugeot', model=308, year=2015)
```

```
In [15]: myothercar = Car('BMW', "1 Series", 2009)
         myothercar
```

```
Out[15]: Car(brand='BMW', model='1 Series', year=2009)
```

8.5 Method Overriding

- Every Python classes has a `__repr__()` method used when you call `print()` function.

```
In [16]: class Car:
             """Simple example class with method overriding """

             def __init__(self, brand, model, year):
                 self.brand = brand
                 self.model = model
                 self.year = year

             def __repr__(self):
                 return f"{self.year} {self.brand} {self.model} {self.__class__.__name__}"
```

```
In [17]: mycar = Car('Peugeot', 308, 2015)
         print(mycar)
```

```
2015 Peugeot 308 Car
```

8.6 Inheritance

```
In [18]: class Rectangle():  # Parent class is defined here

             def __init__(self, width, height):
                 self.width, self.height = width, height
             @property
             def area(self):
                 return self.width * self.height

         class Square(Rectangle):

             def __init__(self, edge):
                 super().__init__(edge, edge)  # Call method in the parent class


         r = Rectangle(2, 3)
         print(f"Rectangle area \t = {r.area:7.3f}")
         s = Square(4)
         print(f"Square area \t = {s.area:7.3f}")
```

```
Rectangle area          =   6.000
Square area          =  16.000
```

8.7 Private Variables and Methods

```
In [19]: class DemoClass:
             " Demo class for name mangling "

             def public_method(self):
                 return 'public!'

             def __private_method(self):  # Note the use of leading underscores
                 return 'private!'
```

```
        object3 = DemoClass()

In [20]: object3.public_method()

Out[20]: 'public!'

In [21]: object3.__private_method()


        ---------------------------------------------------------------------------

        AttributeError                            Traceback (most recent call last)

        <ipython-input-21-5a4a6ba27511> in <module>
    ----> 1 object3.__private_method()


        AttributeError: 'DemoClass' object has no attribute '__private_method'


In [22]: [ s for s in dir(object3) if "method" in s]

Out[22]: ['_DemoClass__private_method', 'public_method']

In [23]: object3._DemoClass__private_method()

Out[23]: 'private!'

In [24]: object3.public_method

Out[24]: <bound method DemoClass.public_method of <__main__.DemoClass object at 0x7f5890247ee0>>
```

8.8 Use class as a Function.

```
In [25]: class Polynomial:

            " Class representing a polynom P(x) -> c_0+c_1*x+c_2*x^2+..."

            def __init__(self, coeffs):
               self.coeffs = coeffs

            def __call__(self, x):
               return sum(coef*x**exp for exp,coef in enumerate(self.coeffs))

         p = Polynomial([2,4,-1])
         p(2)

Out[25]: 6
```

8.8.1 Exercise: Polynomial

- Improve the class above called Polynomial by creating a method `diff(n)` to compute the nth derivative.
- Override the `__repr__()` method to output a pretty printing.

Hint: `f"{coeff:+d}"` forces to print sign before the value of an integer.

8.9 Operators Overriding

```
In [26]: class MyComplex:
             " Simple class representing a complex"
             width = 7
             precision = 3

             def __init__(self, real=0, imag=0):
                 self.real = real
                 self.imag = imag

             def __repr__(self):
                 return (f"({self.real:{self.width}.{self.precision}f},"
                         f"{self.imag:+{self.width}.{self.precision}f}j)")

             def __eq__(self, other):  # override '=='
                 return (self.real == other.real) and (self.imag == other.imag)

             def __add__(self, other):  # override '+'
                 return MyComplex(self.real+other.real, self.imag+other.imag)

             def __sub__(self, other):  # override '-'
                 return MyComplex(self.real-other.real, self.imag-other.imag)

             def __mul__(self, other):  # override '*'
                 if isinstance(other, MyComplex):
                     return MyComplex(self.real * other.real - self.imag * other.imag,
                                      self.real * other.imag + self.imag * other.real)

                 else:
                     return MyComplex(other*self.real, other*self.imag)
```

```
In [27]: u = MyComplex(0, 1)
         v = MyComplex(1, 0)
         print('u=', u, "; v=", v)
```

```
u= (  0.000, +1.000j) ; v= (  1.000, +0.000j)
```

```
In [28]: u+v, u-v, u*v, u==v
```

```
Out[28]: ((  1.000, +1.000j), ( -1.000, +1.000j), (  0.000, +1.000j), False)
```

We can change the *class* attribute precision.

```
In [29]: MyComplex.precision=2
         print(u.precision)
         print(u)
```

```
2
(   0.00,  +1.00j)
```

```
In [30]: v.precision
```

```
Out[30]: 2
```

We can change the *instance* attribute precision.

```
In [31]: u.precision=1
         print(u)
```

```
(    0.0,   +1.0j)
```

```
In [32]: print(v)
```

```
(   1.00,  +0.00j)
```

```
In [33]: MyComplex.precision=5
         u # set attribute keeps its value
```

```
Out[33]: (    0.0,   +1.0j)
```

```
In [34]: v # unset attribute is set to the new value
```

```
Out[34]: (1.00000,+0.00000j)
```

8.10 Rational example

```
In [35]: class Rational:
             " Class representing a rational number"

             def __init__(self, n, d):
                 assert isinstance(n, int) and isinstance(d, int)

                 def gcd(x, y):
                     if x == 0:
                         return y
                     elif x < 0:
                         return gcd(-x, y)
                     elif y < 0:
                         return -gcd(x, -y)
                     else:
                         return gcd(y % x, x)

                 g = gcd(n, d)
                 self.numer, self.denom = n//g, d//g

             def __add__(self, other):
                 return Rational(self.numer * other.denom + other.numer * self.denom,
                                 self.denom * other.denom)

             def __sub__(self, other):
                 return Rational(self.numer * other.denom - other.numer * self.denom,
                                 self.denom * other.denom)

             def __mul__(self, other):
                 return Rational(self.numer * other.numer, self.denom * other.denom)

             def __truediv__(self, other):
                 return Rational(self.numer * other.denom, self.denom * other.numer)

             def __repr__(self):
                 return f"{self.numer:d}/{self.denom:d}"
```

```
In [36]: r1 = Rational(2,3)
         r2 = Rational(3,4)
         r1+r2, r1-r2, r1*r2, r1/r2
```

```
Out[36]: (17/12, -1/12, 1/2, 8/9)
```

8.10.1 Exercise

Improve the class Polynomial by implementing operations: - Overrides '+' operator (**add**) - Overrides '-' operator (**neg**) - Overrides '==' operator (**eq**) - Overrides '*' operator (**mul**)

Chapter 9

Iterators

Most container objects can be looped over using a for statement:

```
In [1]: for element in [1, 2, 3]:
            print(element, end=' ')

1 2 3

In [2]: for element in (1, 2, 3):
            print(element, end=' ')

1 2 3

In [3]: for key in {'one': 1, 'two': 2}:
            print(key, end=' ')

one two

In [4]: for char in "123":
            print(char, end=' ')

1 2 3

In [5]: for line in open("environment.yml"):
            print(line, end= ' ')


        ---------------------------------------------------------------------------

        FileNotFoundError                         Traceback (most recent call last)

        <ipython-input-5-e8ea7e33e965> in <module>
    ----> 1 for line in open("environment.yml"):
          2     print(line, end= ' ')


        FileNotFoundError: [Errno 2] No such file or directory: 'environment.yml'
```

- The `for` statement calls `iter()` on the container object.
- The function returns an iterator object that defines the method `__next__()`
- To add iterator behavior to your classes:
  - Define an `__iter__()` method which returns an object with a `__next__()`.

  - If the class defines `__next__()`, then `__iter__()` can just return self.
  - The **StopIteration** exception indicates the end of the loop.

```
In [6]: s = 'abc'
        it = iter(s)
        it

Out[6]: <str_iterator at 0x7f350963c2e0>

In [7]: next(it), next(it), next(it)

Out[7]: ('a', 'b', 'c')

In [8]: class Reverse:
            """Iterator for looping over a sequence backwards."""

            def __init__(self, data):
                self.data = data
                self.index = len(data)

            def __iter__(self):
                return self

            def __next__(self):
                if self.index == 0:
                    raise StopIteration
                self.index = self.index - 1
                return self.data[self.index]

In [9]: rev = Reverse('spam')
        for char in rev:
            print(char, end='')
```

maps

```
In [10]: def reverse(data): # Python 3.6
             yield from data[::-1]

         for char in reverse('bulgroz'):
             print(char, end='')
```

zorglub

9.1 Generators

- Generators are a simple and powerful tool for creating iterators.
- Write regular functions but use the yield statement when you want to return data.
- the `__iter__()` and `__next__()` methods are created automatically.

```
In [11]: def reverse(data):
             for index in range(len(data)-1, -1, -1):
                 yield data[index]

In [12]: for char in reverse('bulgroz'):
             print(char, end='')
```

zorglub

9.1.1 Exercise

Generates a list of IP addresses based on IP range.

```
ip_range =
for ip in ip_range("192.168.1.0", "192.168.1.10"):
   print(ip)
```

```
192.168.1.0
192.168.1.1
192.168.1.2
...
```

9.2 Generator Expressions

- Use a syntax similar to list comprehensions but with parentheses instead of brackets.
- Tend to be more memory friendly than equivalent list comprehensions.

```
In [13]: sum(i*i for i in range(10))                 # sum of squares

Out[13]: 285

In [14]: %load_ext memory_profiler

In [15]: %memit doubles = [2 * n for n in range(10000)]

peak memory: 48.99 MiB, increment: 1.12 MiB


In [16]: %memit doubles = (2 * n for n in range(10000))

peak memory: 48.99 MiB, increment: 0.00 MiB


In [17]: # list comprehension
         doubles = [2 * n for n in range(10)]
         for x in doubles:
             print(x, end=' ')

0 2 4 6 8 10 12 14 16 18

In [18]: # generator expression
         doubles = (2 * n for n in range(10))
         for x in doubles:
             print(x, end=' ')

0 2 4 6 8 10 12 14 16 18
```

9.2.1 Exercise

The Chebyshev polynomials of the first kind are defined by the recurrence relation

$$T_o(x) = 1 \tag{9.1}$$
$$T_1(x) = x \tag{9.2}$$
$$T_{n+1} = 2xT_n(x) - T_{n-1}(x) \tag{9.3}$$

- Create a class `Chebyshev` that generates the sequence of Chebyshev polynomials

9.3 itertools

9.3.1 zip_longest

`itertools.zip_longest()` accepts any number of iterables as arguments and a fillvalue keyword argument that defaults to None.

```
In [19]: x = [1, 1, 1, 1, 1]
         y = [1, 2, 3, 4, 5, 6, 7]
         list(zip(x, y))
         from itertools import zip_longest
         list(map(sum,zip_longest(x, y, fillvalue=1)))

Out[19]: [2, 3, 4, 5, 6, 7, 8]
```

9.3.2 combinations

```
In [20]: loto_numbers = list(range(1,50))
```

A choice of 6 numbers from the sequence 1 to 49 is called a combination. The `itertools.combinations()` function takes two arguments—an iterable inputs and a positive integer n—and produces an iterator over tuples of all combinations of n elements in inputs.

```
In [21]: from itertools import combinations
         len(list(combinations(loto_numbers, 6)))

Out[21]: 13983816

In [22]: from math import factorial
         factorial(49)/ factorial(6) / factorial(49-6)

Out[22]: 13983816.0
```

9.3.3 permutations

```
In [23]: from itertools import permutations
         for s in permutations('dsi'):
             print( "".join(s), end=", ")

dsi, dis, sdi, sid, ids, isd,
```

9.3.4 count

```
In [24]: from itertools import count
         n = 2024
         for k in count(): # replace  k = 0; while(True) : k += 1
             if n == 1:
                 print(f"k = {k}")
                 break
             elif n & 1:
                 n = 3*n +1
             else:
                 n = n // 2

k = 112
```

9.3.5 cycle, islice, dropwhile, takewhile

```
In [25]: from itertools import cycle, islice, dropwhile, takewhile
         L = list(range(10))
         cycled = cycle(L)  # cycle through the list 'L'
         skipped = dropwhile(lambda x: x < 6 , cycled)  # drop the values until x==4
         sliced = islice(skipped, None, 20)  # take the first 20 values
         print(*sliced)
```

```
6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
```

```
In [26]: result = takewhile(lambda x: x > 0, cycled) # cycled begins to 4
         print(*result)
```

```
6 7 8 9
```

9.3.6 product

```
In [27]: ranks = ['A', 'K', 'Q', 'J', '10', '9', '8', '7']
         suits = [ '\u2660', '\u2665', '\u2663', '\u2666']
         cards = [(rank, suit) for rank in ranks for suit in suits]
         len(cards)
         from itertools import product
         cards = product(ranks, suits)
         print(*cards)
```

```
('A', ' ') ('A', ' ') ('A', ' ') ('A', ' ') ('K', ' ') ('K', ' ') ('K', ' ') ('K', ' ') ('Q', ' ') ('Q', ' '
```

Chapter 10

Multiprocessing

```
In [1]: from multiprocessing import cpu_count

        cpu_count()
```

```
Out[1]: 2
```

10.1 Map reduce example

```
In [2]: from time import sleep
        def delayed_square(x):
            sleep(1)
            return x*x
        data = list(range(8))
        data
```

```
Out[2]: [0, 1, 2, 3, 4, 5, 6, 7]
```

```
In [3]: %time sum(delayed_square(x) for x in data)
```

```
CPU times: user 2.33 ms, sys: 353 µs, total: 2.68 ms
Wall time: 8.01 s
```

```
Out[3]: 140
```

```
In [4]: %time sum(map(delayed_square,data))
```

```
CPU times: user 2.27 ms, sys: 296 µs, total: 2.57 ms
Wall time: 8.01 s
```

```
Out[4]: 140
```

We can process each `delayed_square` calls independently and in parallel. To accomplish this we'll apply that function across all list items in parallel using multiple processes.

10.2 Thread and Process: Differences

- A Process is an instance of a running program.
- Process may contain one or more threads, but a thread cannot contain a process.

- Process has a self-contained execution environment. It has its own memory space.
- Application running on your computer may be a set of cooperating processes.
- A Thread is made of and exist within a Process; every process has at least one.
- Multiple threads in a process share resources, which helps in efficient communication between threads.
- Threads can be concurrent on a multi-core system, with every core executing the separate threads simultaneously.

10.3 Multi-Processing vs Multi-Threading

10.3.1 Memory

- Each process has its own copy of the data segment of the parent process.
- Each thread has direct access to the data segment of its process.
- A process runs in separate memory spaces.
- A thread runs in shared memory spaces.

10.3.2 Communication

- Processes must use inter-process communication to communicate with sibling processes.
- Threads can directly communicate with other threads of its process.

10.3.3 Overheads

- Processes have considerable overhead.
- Threads have almost no overhead.

10.4 Multi-Processing vs Multi-Threading

10.4.1 Creation

- New processes require duplication of the parent process.
- New threads are easily created.

10.4.2 Control

- Processes can only exercise control over child processes.
- Threads can exercise considerable control over threads of the same process.

10.4.3 Changes

- Any change in the parent process does not affect child processes.
- Any change in the main thread may affect the behavior of the other threads of the process.

10.5 The Global Interpreter Lock (GIL)

- The Python interpreter is not thread safe.
- A few critical internal data structures may only be accessed by one thread at a time. Access to them is protected by the GIL.
- Attempts at removing the GIL from Python have failed until now. The main difficulty is maintaining the C API for extension modules.

- Multiprocessing avoids the GIL by having separate processes which each have an independent copy of the interpreter data structures.
- The price to pay: serialization of tasks, arguments, and results.

10.6 Multiprocessing (history)

- The multiprocessing allows the programmer to fully leverage multiple processors.
- The `Pool` object parallelizes the execution of a function across multiple input values.
- The if `__name__ == '__main__'` part is necessary.
  The next program does not work in a cell you need to save it and run with python in a terminal

```
python3 pool.py
```

```
In [5]: %%file pool.py

        from time import time, sleep

        from multiprocessing import Pool

        def delayed_square(x):
            sleep(1)
            return x*x

        if __name__ == '__main__': # Executed only on main process.
            start = time()
            data = list(range(8))
            with Pool() as p:
                result = sum(p.map(delayed_square, data))
            stop = time()
            print(f"result = {result} - Elapsed time {stop - start}")
```

```
Writing pool.py
```

```
In [6]: import sys
        !{sys.executable} pool.py
```

```
result = 140 - Elapsed time 4.026031732559204
```

10.7 Futures

The `concurrent.futures` module provides a high-level interface for asynchronously executing callables.

The asynchronous execution can be performed with threads, using ThreadPoolExecutor, or separate processes, using ProcessPoolExecutor. Both implement the same interface, which is defined by the abstract Executor class.

```
In [7]: %%file process_pool.py
        import os
        from time import time, sleep
        if os.name == "nt":
            from loky import ProcessPoolExecutor  # for Windows users
        else:
            from concurrent.futures import ProcessPoolExecutor

            from time import time, sleep
```

```
        def delayed_square(x):
            sleep(1)
            return x*x

        if __name__ == "__main__":
            start = time()
            data = list(range(8))
            with ProcessPoolExecutor() as pool:
                result = sum(pool.map(delayed_square, data))
            stop = time()
            print(f" result : {result} - elapsed time {stop - start}")
```

```
Writing process_pool.py
```

```
In [8]: !{sys.executable} process_pool.py
```

```
 result : 140 - elapsed time 4.011481523513794
```

```
In [9]: %%time
        from concurrent.futures import ThreadPoolExecutor

        e = ThreadPoolExecutor()

        results = list(e.map(delayed_square, range(8)))
```

```
CPU times: user 4.47 ms, sys: 38 µs, total: 4.51 ms
Wall time: 2 s
```

10.8 Asynchronous Future

While many parallel applications can be described as maps, some can be more complex. In this section we look at the asynchronous Future interface, which provides a simple API for ad-hoc parallelism. This is useful for when your computations don't fit a regular pattern.

10.8.1 Executor.submit

The `submit` method starts a computation in a separate thread or process and immediately gives us a `Future` object that refers to the result. At first, the future is pending. Once the function completes the future is finished.

We collect the result of the task with the `.result()` method, which does not return until the results are available.

```
In [10]: from time import sleep

         def slowadd(a, b, delay=1):
             sleep(delay)
             return a + b
```

```
In [11]: from concurrent.futures import ThreadPoolExecutor
         e = ThreadPoolExecutor(4)
         future = e.submit(slowadd, 1, 2)
         future
```

```
Out[11]: <Future at 0x7fce58271c70 state=running>
```

```
In [12]: future.result()
```

```
Out[12]: 3
```

Submit many tasks all at once and they be will executed in parallel.

```
In [13]: %%time
         results = [slowadd(i, i, delay=1) for i in range(8)]
```

```
CPU times: user 2.87 ms, sys: 0 ns, total: 2.87 ms
Wall time: 8.01 s
```

```
In [14]: %%time
         futures = [e.submit(slowadd, 1, 1, delay=1) for i in range(8)]
         results = [f.result() for f in futures]
```

```
CPU times: user 2.53 ms, sys: 551 µs, total: 3.09 ms
Wall time: 2 s
```

- Submit fires off a single function call in the background, returning a future.
- When you combine submit with a single for loop we recover the functionality of map.
- To collect your results, replace each of futures, `f`, with a call to `f.result()`
- Combine submit with multiple for loops and other general programming to get something more general than map.
- Sometimes, it did not speed up the code very much
- Threads and processes show some performance differences
- Use threads carefully, you can break your Python session.

Today most library designers are coordinating around the concurrent.futures interface, so it's wise to move over.

- Profile your code
- Used concurrent.futures.ProcessPoolExecutor for simple parallelism
- Gained some speed boost (but not as much as expected)
- Lost ability to diagnose performance within parallel code
- Describing each task as a function call helps use tools like map for parallelism
- Making your tasks fast is often at least as important as parallelizing your tasks.

10.8.2 Exercise: Pi computation

Parallelize this computation with a ProcessPoolExecutor. ThreadPoolExecutor is not usable because of `random` function calls.

```
In [15]: import time
         import random

         def compute_pi(n):
             count = 0
             for i in range(n):
                 x = random.random()
                 y = random.random()
                 if x*x + y*y <= 1:
                     count += 1
             return count
```

```
        elapsed_time = time.time()
        nb_simulations = 4
        n = 10**7
        result = [compute_pi(n) for i in range(nb_simulations)]
        pi = 4 * sum(result) / (n*nb_simulations)
        print(f"Estimated value of Pi : {pi:.8f} time : {time.time()-elapsed_time:.8f}")
```

```
Estimated value of Pi : 3.14116110 time : 12.66365552
```

10.8.3 Exercise

- Do the same computation using asynchronous future
- Implement a joblib version (see example below)

10.9 Parallel tools for Python

The parallel tools from standard library are very limited. You will have more powerful features with:

- Joblib provides a simple helper class to write parallel for loops using multiprocessing.
- Dask
- PySpark
- mpi4py

Chapter 11

Standard Library

11.1 Operating System Interface

```
In [1]: import os
        os.getcwd()      # Return the current working directory
```

```
Out[1]: '/home/runner/work/python-notebooks/python-notebooks/notebooks'
```

```
In [2]: %env CC='/usr/local/bin/gcc-7'
        os.environ['CC']='/usr/local/bin/gcc-7' # Change the default C compiler to gcc-7
        os.system('mkdir today') # Run the command mkdir in the system shell
```

```
env: CC='/usr/local/bin/gcc-7'
```

```
Out[2]: 0
```

```
In [3]: os.chdir('today')   # Change current working directory
        os.system('touch data.db') # Create the empty file data.db
```

```
Out[3]: 0
```

```
In [4]: import shutil
        shutil.copyfile('data.db', 'archive.db')
        if os.path.exists('backup.db'):  # If file backup.db exists
            os.remove('backup.db')       # Remove it
        shutil.move('archive.db', 'backup.db',)
        shutil.os.chdir('..')
```

11.2 File Wildcards

The glob module provides a function for making file lists from directory wildcard searches:

```
In [5]: import glob
        glob.glob('*.py')
```

```
Out[5]: ['process_pool.py', 'pool.py']
```

```
In [6]: def recursive_replace( root, pattern, replace ) :
            """
            Function to replace a string inside a directory
            root : directory
            pattern : searched string
```

```
    replace "pattern" by "replace"
    """
    for directory, subdirs, filenames in os.walk( root ):
      for filename in filenames:
        path = os.path.join( directory, filename )
        text = open( path ).read()
        if pattern in text:
          print('occurence in :' + filename)
          open(path,'w').write( text.replace( pattern, replace ) )
```

11.3 Command Line Arguments

These arguments are stored in the sys module's argv attribute as a list.

```
   writefile magic_args="-a demo.py" slideshow={"slide_type": "fragment"} import sys
print(sys.argv)
```

```
In [7]: %run demo.py one two three
```

```
    ---------------------------------------------------------------------------

    OSError                                   Traceback (most recent call last)

    /usr/share/miniconda3/envs/runenv/lib/python3.8/site-packages/IPython/core/magics/execution.py i
    702             fpath = arg_lst[0]
--> 703             filename = file_finder(fpath)
    704         except IndexError:


    /usr/share/miniconda3/envs/runenv/lib/python3.8/site-packages/IPython/utils/path.py in get_py_fi
    108     else:
--> 109         raise IOError('File `%r` not found.' % name)
    110


    OSError: File `'demo.py'` not found.


During handling of the above exception, another exception occurred:


    Exception                                 Traceback (most recent call last)

    <ipython-input-7-916df5e5b25f> in <module>
----> 1 get_ipython().run_line_magic('run', 'demo.py one two three')


    /usr/share/miniconda3/envs/runenv/lib/python3.8/site-packages/IPython/core/interactiveshell.py i
   2324                 kwargs['local_ns'] = self.get_local_scope(stack_depth)
   2325             with self.builtin_trap:
-> 2326                 result = fn(*args, **kwargs)
   2327             return result
   2328
```

```
<decorator-gen-59> in run(self, parameter_s, runner, file_finder)


/usr/share/miniconda3/envs/runenv/lib/python3.8/site-packages/IPython/core/magic.py in <lambda>
185     # but it's overkill for just that one bit of state.
186     def magic_deco(arg):
--> 187         call = lambda f, *a, **k: f(*a, **k)
188
189         if callable(arg):


/usr/share/miniconda3/envs/runenv/lib/python3.8/site-packages/IPython/core/magics/execution.py
712             if os.name == 'nt' and re.match(r"^'.*'$",fpath):
713                 warn('For Windows, use double quotes to wrap a filename: %run "mypath\\myfi
--> 714             raise Exception(msg)
715         except TypeError:
716             if fpath in sys.meta_path:


Exception: File `''demo.py''` not found.
```

11.4 Random

```
In [8]: import random
        random.choice(['apple', 'pear', 'banana'])
```

```
Out[8]: 'apple'
```

```
In [9]: random.sample(range(100), 10)   # sampling without replacement
```

```
Out[9]: [87, 96, 80, 42, 92, 46, 4, 22, 53, 28]
```

```
In [10]: random.random()    # random float
```

```
Out[10]: 0.3800633462797103
```

```
In [11]: random.randrange(6)    # random integer chosen from range(6)
```

```
Out[11]: 1
```

11.5 Statistics

```
In [12]: import statistics
         data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]
         statistics.mean(data)
```

```
Out[12]: 1.6071428571428572
```

```
In [13]: statistics.median(data)
```

```
Out[13]: 1.25
```

```
In [14]: statistics.variance(data)
```

```
Out[14]: 1.3720238095238095
```

11.6 Performance Measurement

```
In [15]: from timeit import Timer
         Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()

Out[15]: 0.02361907599998858

In [16]: Timer('a,b = b,a', 'a=1; b=2').timeit()

Out[16]: 0.019873652999990554

In [17]: %%timeit a=1; b=2
         a,b = b,a

20.3 ns ± 0.54 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)
```

The profile and pstats modules provide tools for identifying time critical sections in larger blocks of code.

11.7 Quality Control

One approach for developing high quality software is to write tests for each function.

- The doctest module provides a tool for scanning a module and validating tests embedded in a program's docstrings.
- This improves the documentation by providing the user with an example and it allows the doctest module to make sure the code remains true to the documentation:

```
In [18]: def average(values):
             """Computes the arithmetic mean of a list of numbers.

             >>> print(average([20, 30, 70]))
             40.0
             """
             return sum(values) / len(values)

         import doctest
         doctest.testmod()   # automatically validate the embedded tests

Out[18]: TestResults(failed=0, attempted=1)
```

11.8 Python's standard library is very extensive

- Containers and iterators: `collections`, `itertools`
- Internet access: `urllib`, `email`, `mailbox`, `cgi`, `ftplib`
- Dates and Times: `datetime`, `calendar`,
- Data Compression: `zlib`, `gzip`, `bz2`, `lzma`, `zipfile`, `tarfile`
- File formats: `csv`, `configparser`, `netrc`, `xdrlib`, `plistlib`
- Cryptographic Services: `hashlib`, `hmac`, `secrets`
- Structure Markup Processing Tools: `html`, `xml`

Check the The Python Standard Library

Chapter 12

Matplotlib

- Python 2D plotting library which produces figures in many formats and interactive environments.
- Tries to make easy things easy and hard things possible.
- You can generate plots, histograms, power spectra, bar charts, errorcharts, scatterplots, etc., with just a few lines of code.
- Check the Matplotlib gallery.
- For simple plotting the pyplot module provides a MATLAB-like interface, particularly when combined with IPython.
- Matplotlib provides a set of functions familiar to MATLAB users.

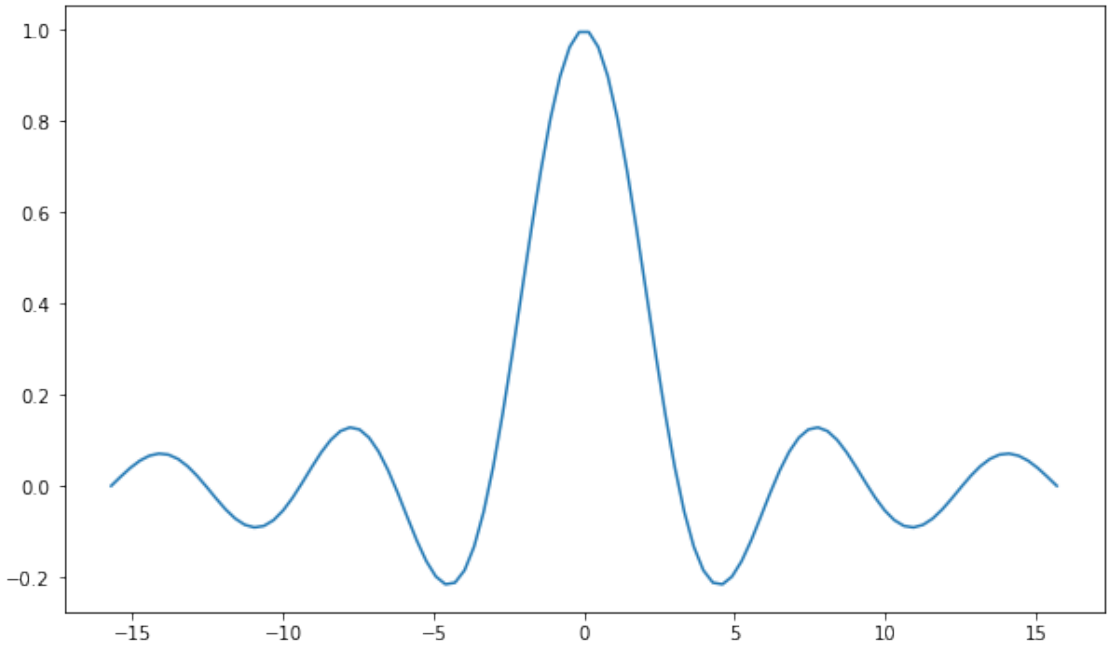
In this notebook we use some numpy command that will be explain more precisely later.

12.1 Line Plots

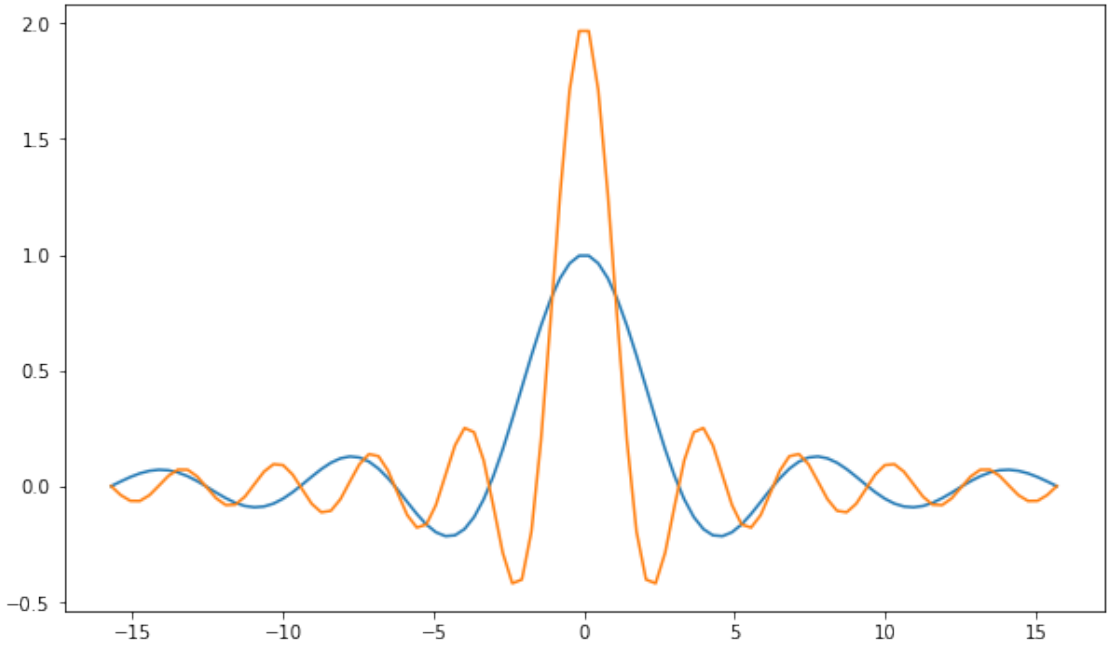
- `np.linspace(0,1,10)` return 10 evenly spaced values over $[0, 1]$.

```
In [1]: %matplotlib inline
        # inline can be replaced by notebook to get interactive plots
        import numpy as np
        import matplotlib.pyplot as plt
```

```
In [2]: plt.rcParams['figure.figsize'] = (10.0, 6.0) # set figures display bigger
        x = np.linspace(- 5*np.pi,5*np.pi,100)
        plt.plot(x,np.sin(x)/x);
```




```
In [3]: plt.plot(x,np.sin(x)/x,x,np.sin(2*x)/x);
```

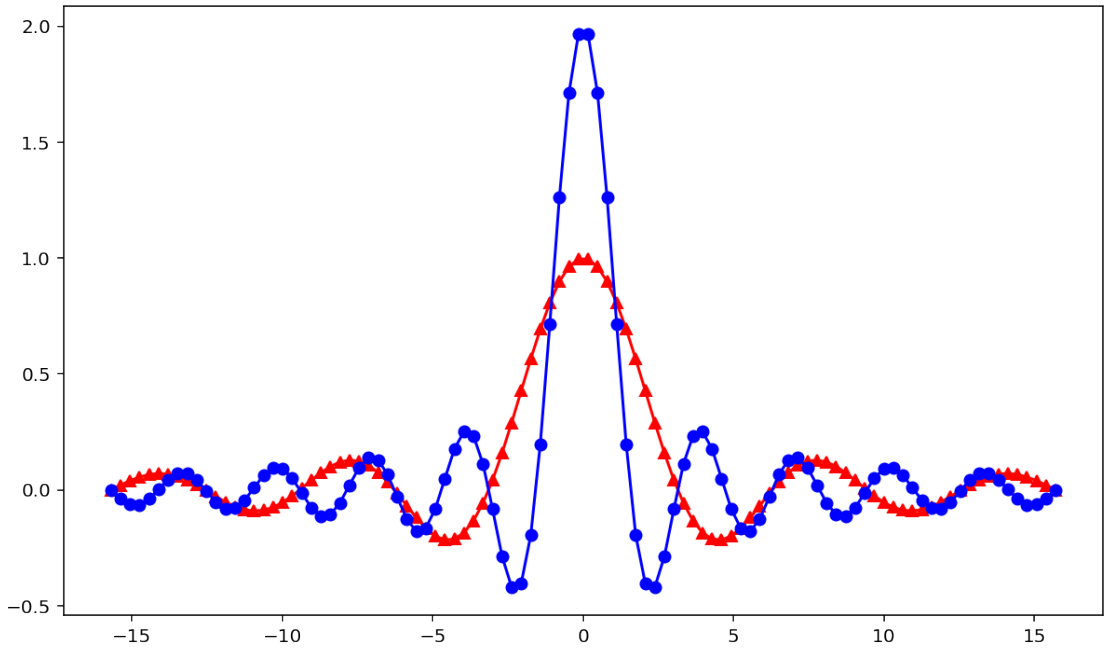


If you have a recent Macbook with a Retina screen, you can display high-resolution plot outputs. Running the next cell will give you double resolution plot output for Retina screens.

Note: the example below won't render on non-retina screens

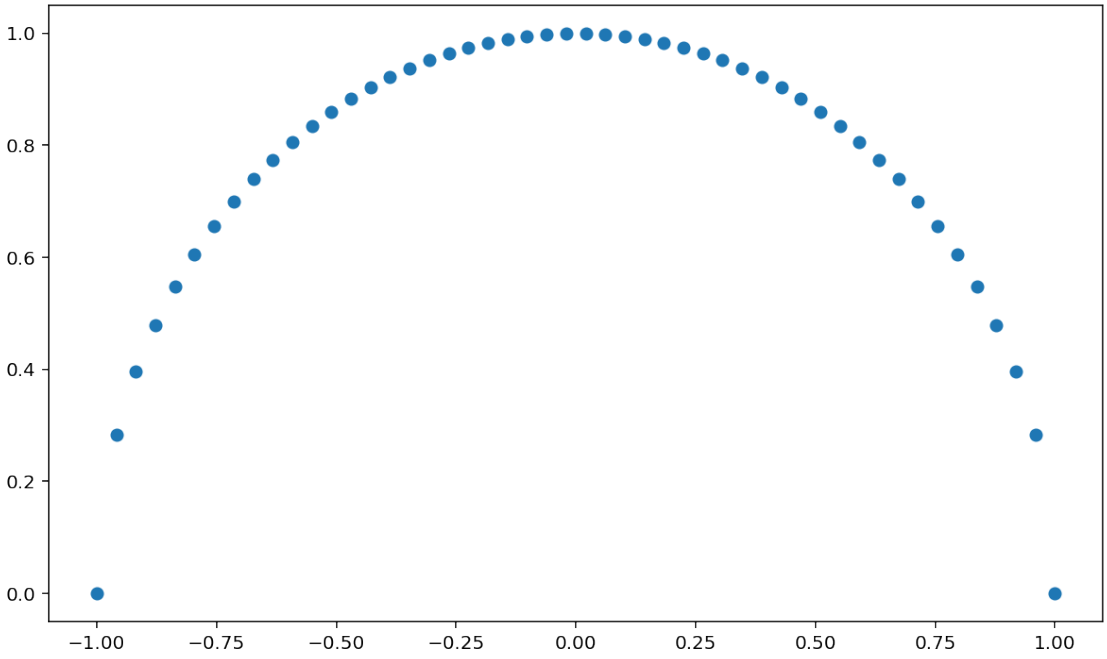
```
In [4]: %config InlineBackend.figure_format = 'retina'
```

```
In [5]: # red, dot-dash, triangles and blue, dot-dash, bullet
        plt.plot(x,np.sin(x)/x, 'r-^',x,np.sin(2*x)/x, 'b-o');
```



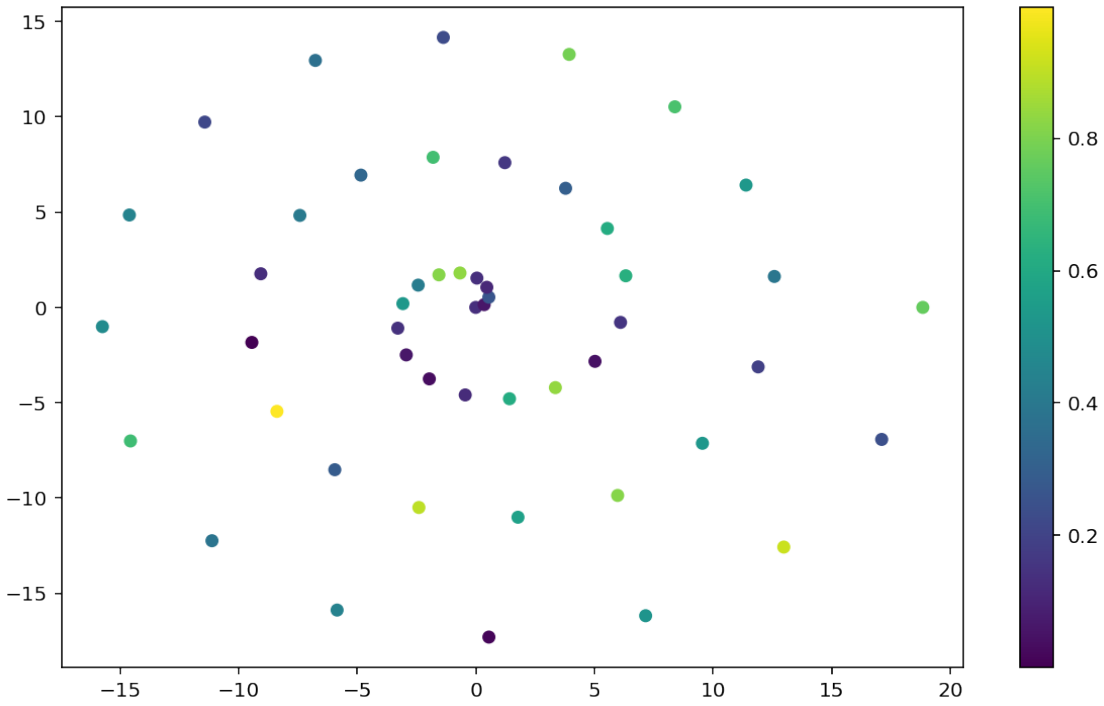
12.2 Simple Scatter Plot

```
In [6]: x = np.linspace(-1,1,50)
        y = np.sqrt(1-x**2)
        plt.scatter(x,y);
```



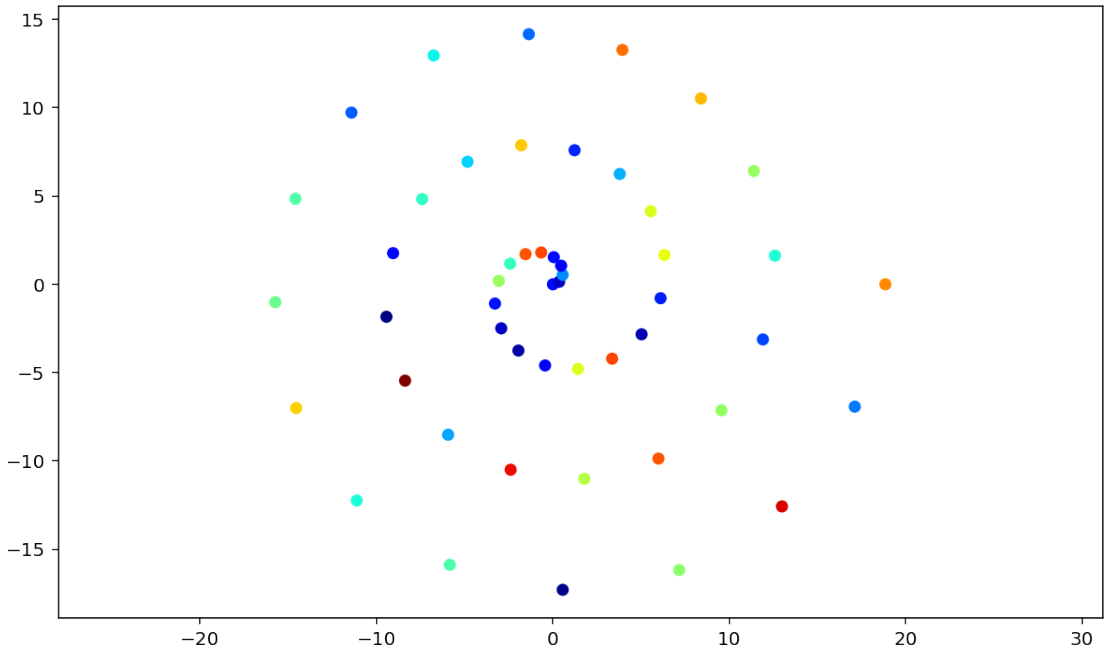
12.3 Colormapped Scatter Plot

```
In [7]: theta = np.linspace(0,6*np.pi,50) # 50 steps from 0 to 6 PI
        size = 30*np.ones(50) # array with 50 values set to 30
        z = np.random.rand(50) # array with 50 random values in [0,1]
        x = theta*np.cos(theta)
        y = theta*np.sin(theta)
        plt.scatter(x,y,size,z)
        plt.colorbar();
```



12.4 Change Colormap

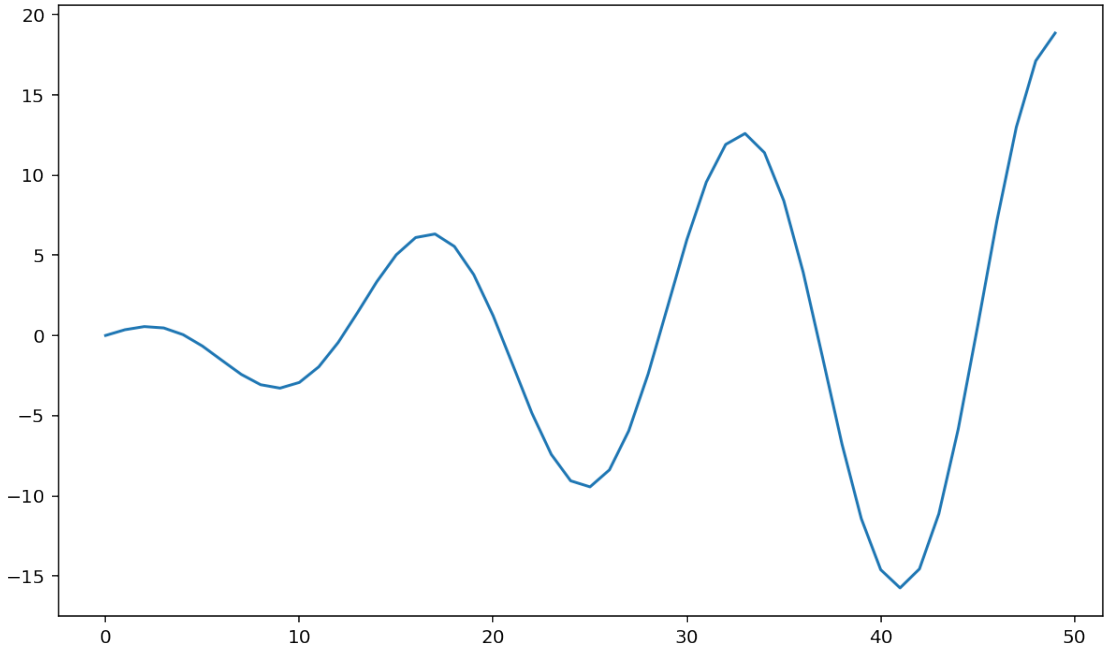
```
In [8]: fig = plt.figure() # create a figure
        ax = fig.add_subplot(1, 1, 1) # add a single plot
        ax.scatter(x,y,size,z,cmap='jet');
        ax.set_aspect('equal', 'datalim')
```

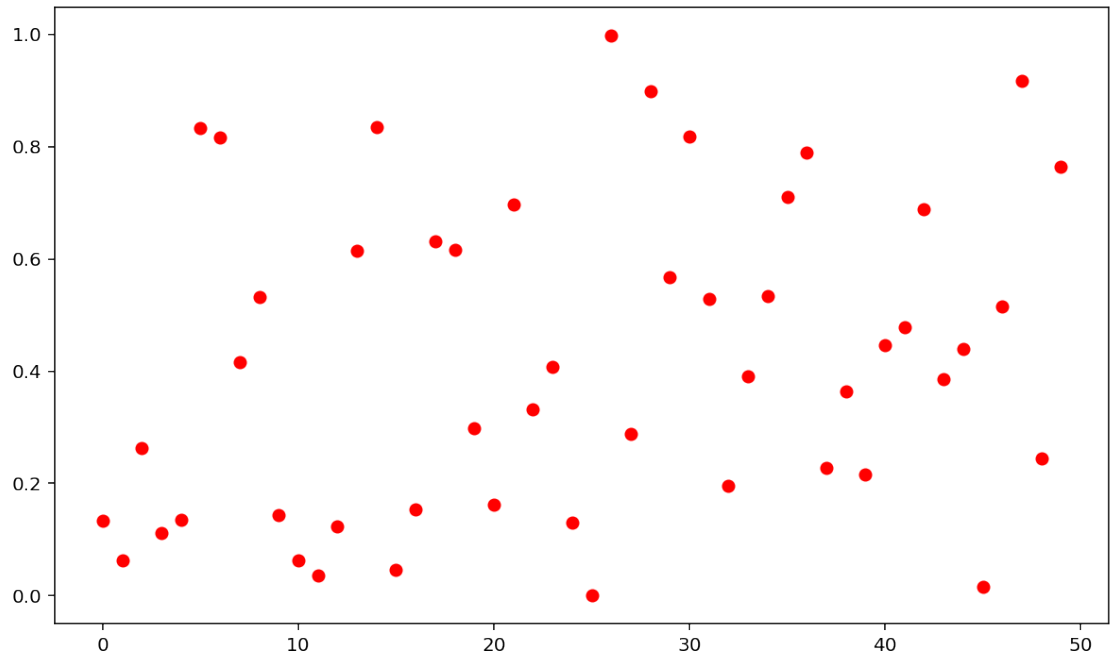


colormaps in matplotlib documentation.

12.5 Multiple Figures

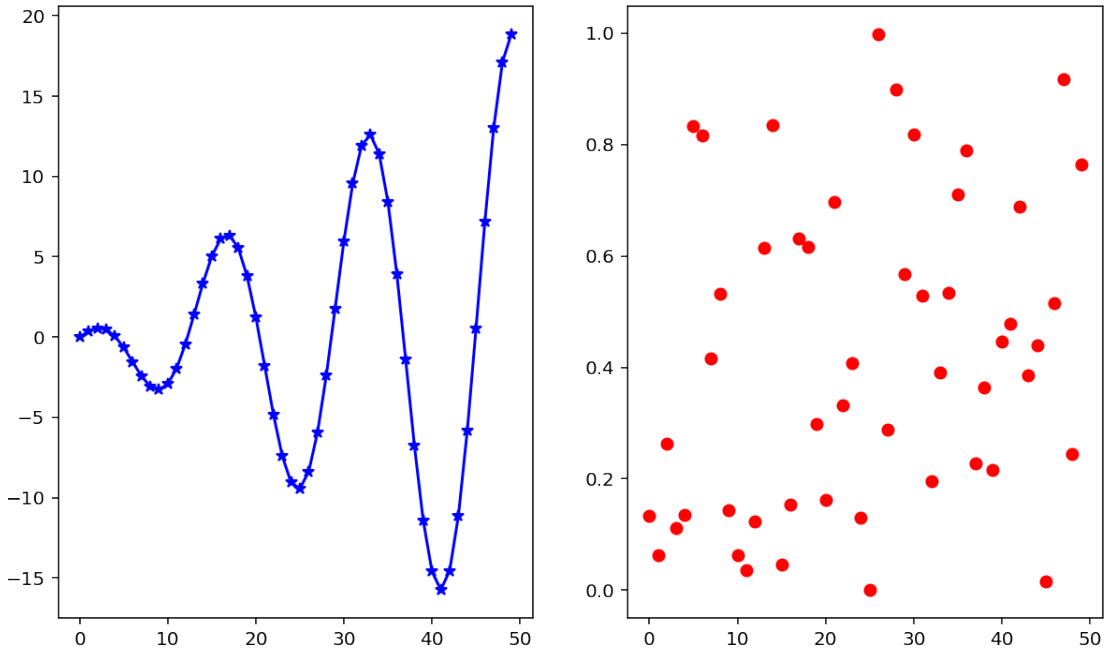
```
In [9]: plt.figure()
        plt.plot(x)
        plt.figure()
        plt.plot(z,'ro');
```





12.6 Multiple Plots Using `subplot`

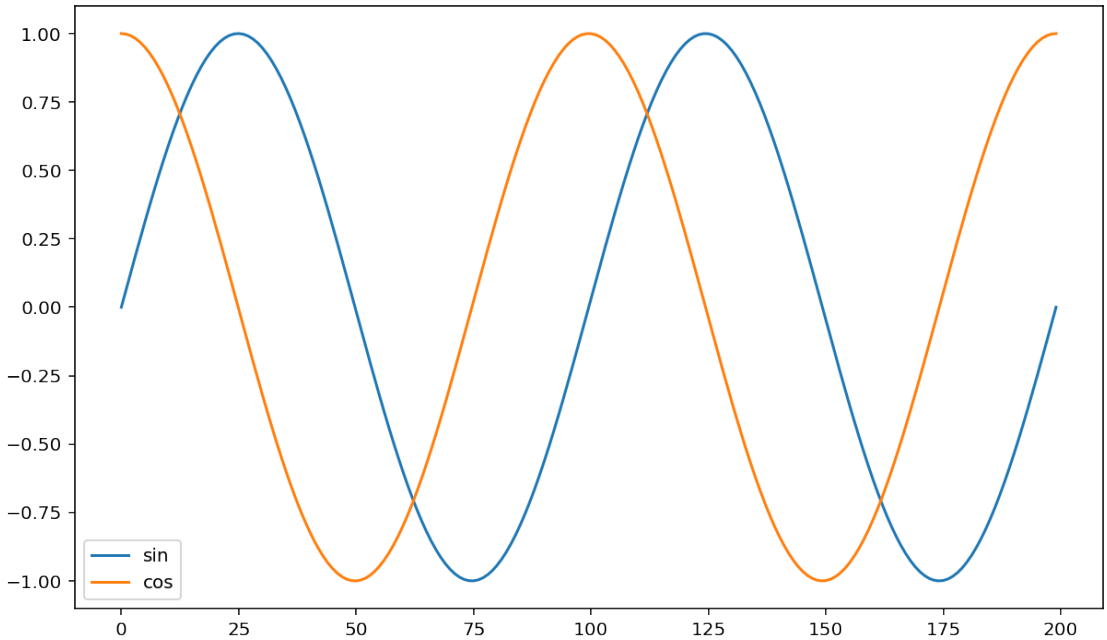
```
In [10]: plt.subplot(1,2,1) # 1 row 1, 2 columns, active plot number 1
         plt.plot(x,'b-*')
         plt.subplot(1,2,2) # 1 row 1, 2 columns, active plot number 2
         plt.plot(z,'ro');
```



12.7 Legends

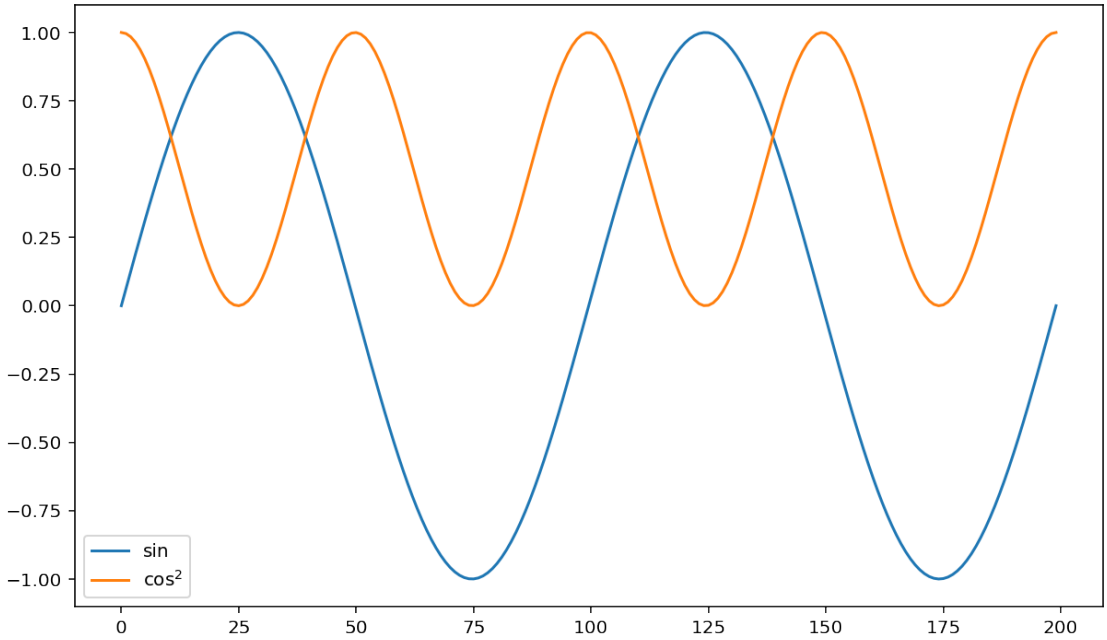
- Legends labels with plot

```
In [11]: theta =np.linspace(0,4*np.pi,200)
         plt.plot(np.sin(theta), label='sin')
         plt.plot(np.cos(theta), label='cos')
         plt.legend();
```



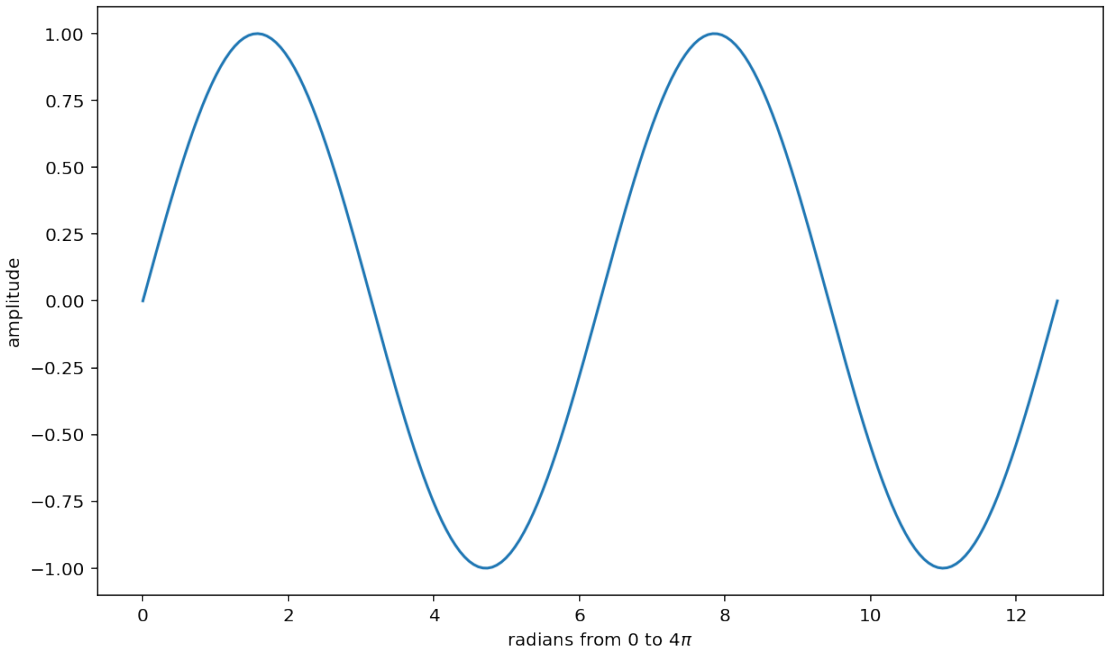
- Labelling with `legend`

```
In [12]: plt.plot(np.sin(theta))
         plt.plot(np.cos(theta)**2)
         plt.legend(['sin','$\cos^2$']);
```

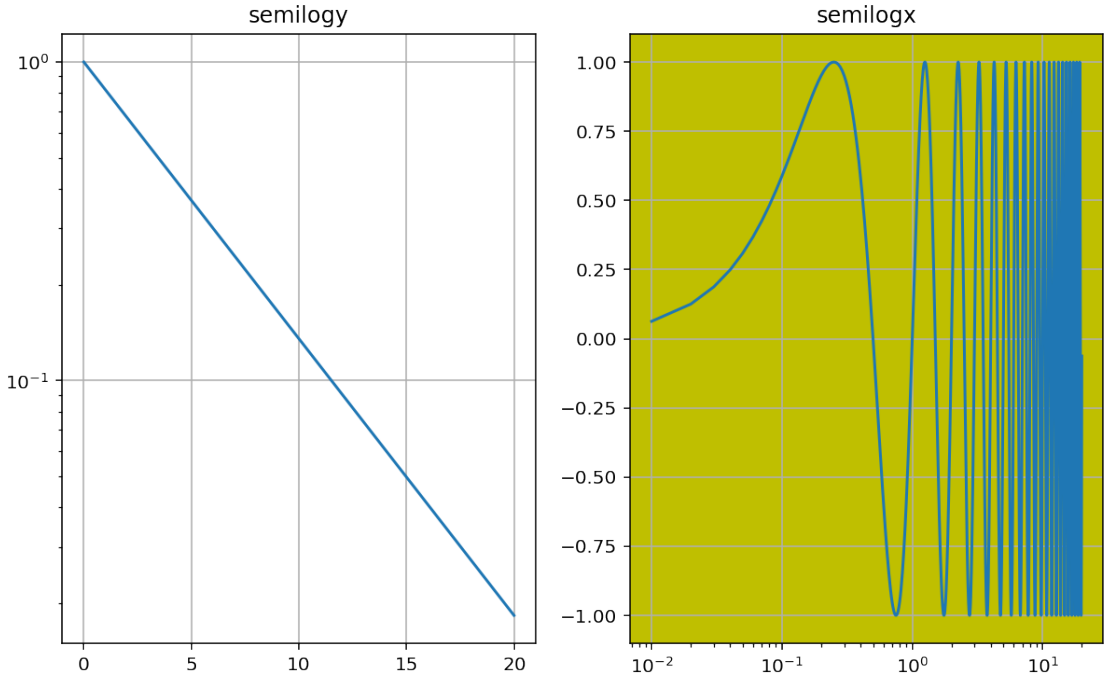


12.8 Titles and Axis Labels

```
In [13]: plt.plot(theta,np.sin(theta))
         plt.xlabel('radians from 0 to $4\pi$')
         plt.ylabel('amplitude');
```

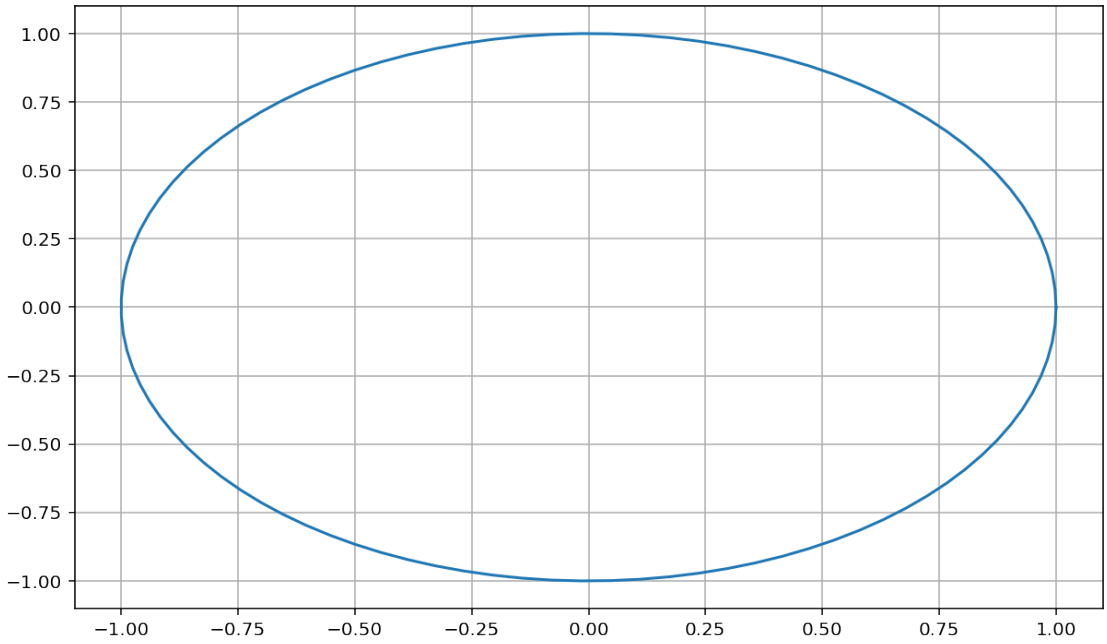



```
In [14]: t = np.arange(0.01, 20.0, 0.01)

         plt.subplot(121)
         plt.semilogy(t, np.exp(-t/5.0))
         plt.title('semilogy')
         plt.grid(True)

         plt.subplot(122,fc='y')
         plt.semilogx(t, np.sin(2*np.pi*t))
         plt.title('semilogx')
         plt.grid(True)
```



12.9 Plot Grid and Save to File

```
In [15]: theta = np.linspace(0,2*np.pi,100)
         plt.plot(np.cos(theta),np.sin(theta))
         plt.grid();
```



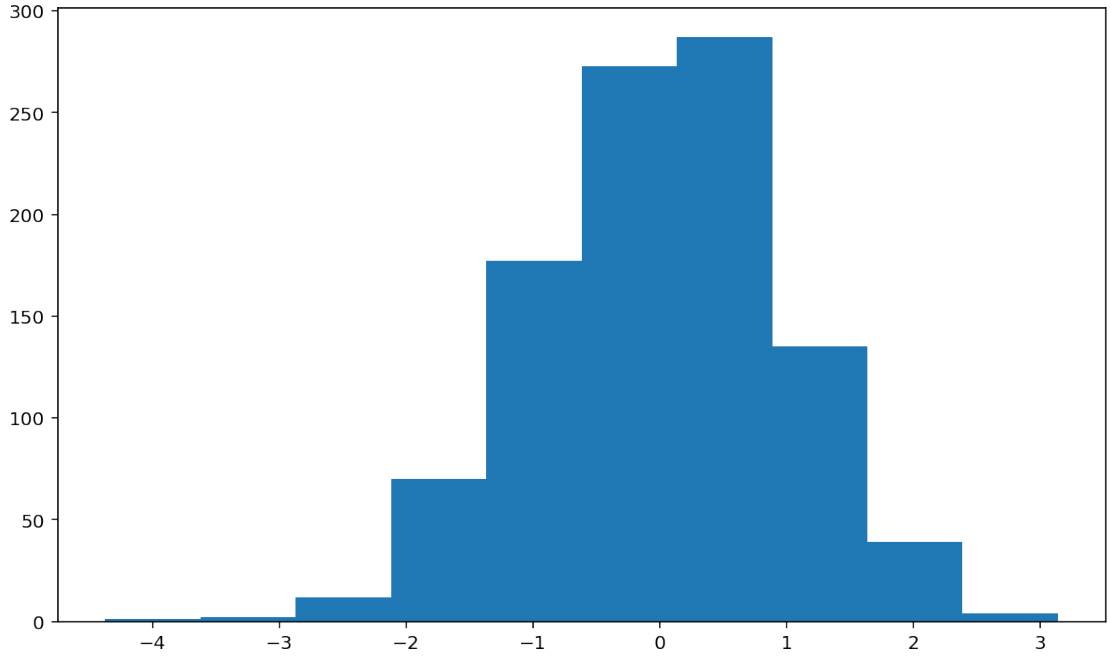
```
In [16]: plt.savefig('circle.png');
         %ls *.png
```

```
circle.png
```

```
<Figure size 720x432 with 0 Axes>
```

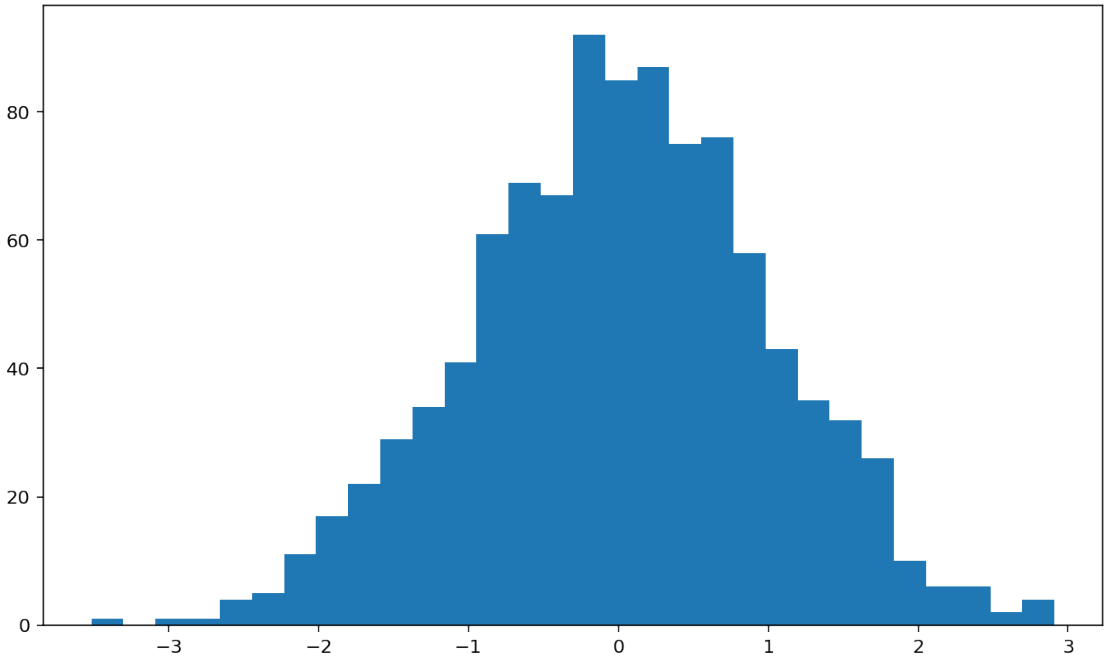
12.10 Histogram

```
In [17]: from numpy.random import randn
         plt.hist(randn(1000));
```



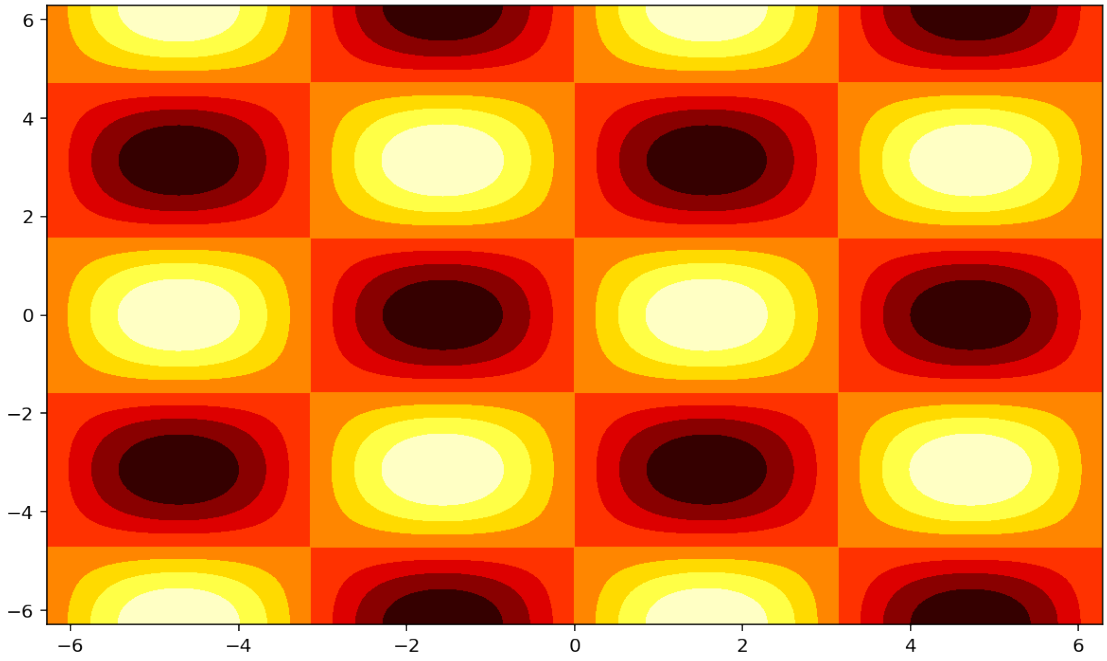
Change the number of bins and supress display of returned array with ;

```
In [18]: plt.hist(randn(1000), 30);
```



12.11 Contour Plot

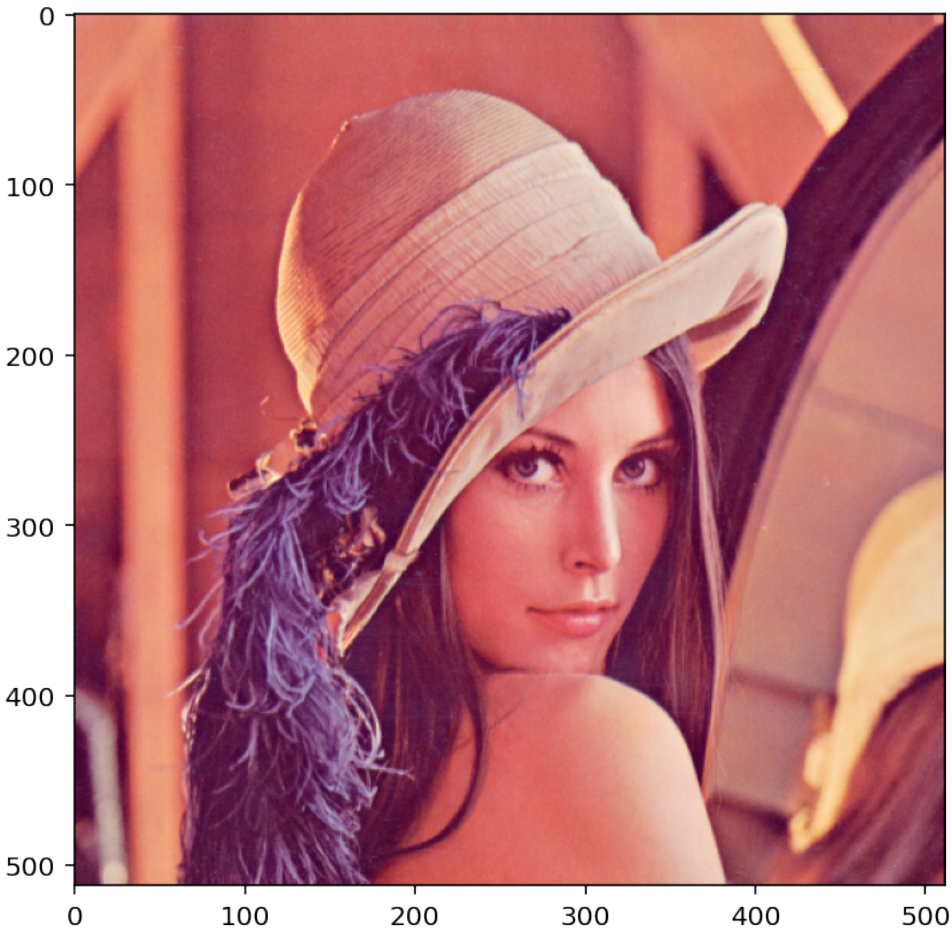
```
In [19]: x = y = np.arange(-2.0*np.pi, 2.0*np.pi+0.01, 0.01)
         X, Y = np.meshgrid(x, y)
         Z = np.sin(X)*np.cos(Y)

         plt.contourf(X, Y, Z,cmap=plt.cm.hot);
```



12.12 Image Display

```
In [20]: img = plt.imread("https://hackage.haskell.org/package/JuicyPixels-extra-0.1.0/src/data-examples/lenna.png"
         plt.imshow(img)
```

```
Out[20]: <matplotlib.image.AxesImage at 0x7f7fc36e1d00>
```



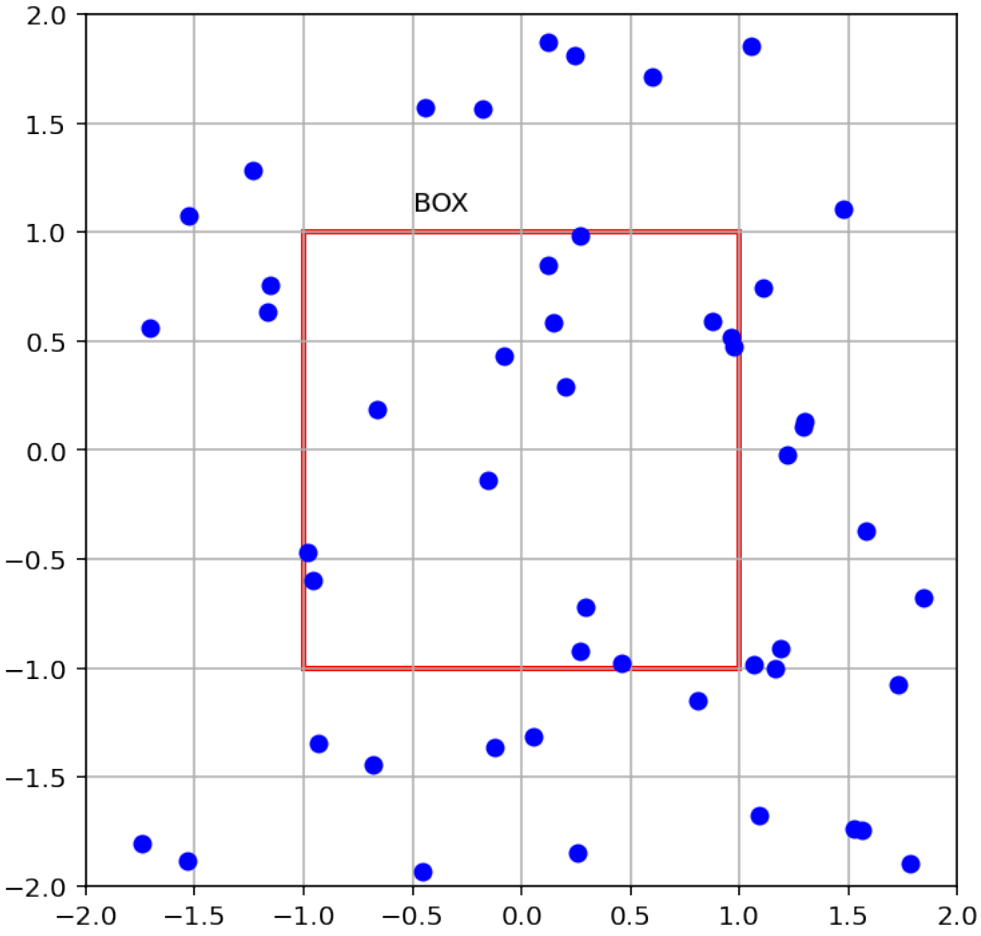
12.13 figure and axis

Best method to create a plot with many components

```
In [21]: fig = plt.figure()
         axis = fig.add_subplot(111, aspect='equal',
                                xlim=(-2, 2), ylim=(-2, 2))

         state = -0.5 + np.random.random((50, 4))
         state[:, :2] *= 3.9
```

```
bounds = [-1, 1, -1, 1]

particles = axis.plot(state[:,0], state[:,1], 'bo', ms=6)
rect = plt.Rectangle(bounds[::2],
                     bounds[1] - bounds[0],
                     bounds[3] - bounds[2],
                     ec='r', lw=2, fc='none')
axis.grid()
axis.add_patch(rect)
axis.text(-0.5,1.1,"BOX")
```

```
Out[21]: Text(-0.5, 1.1, 'BOX')
```



12.14 Exercises

Recreate the image my_plots.png using the *delicate_arch.png* file in *images* directory.

12.15 Alternatives

- bqplot : Jupyter Notebooks, Interactive.

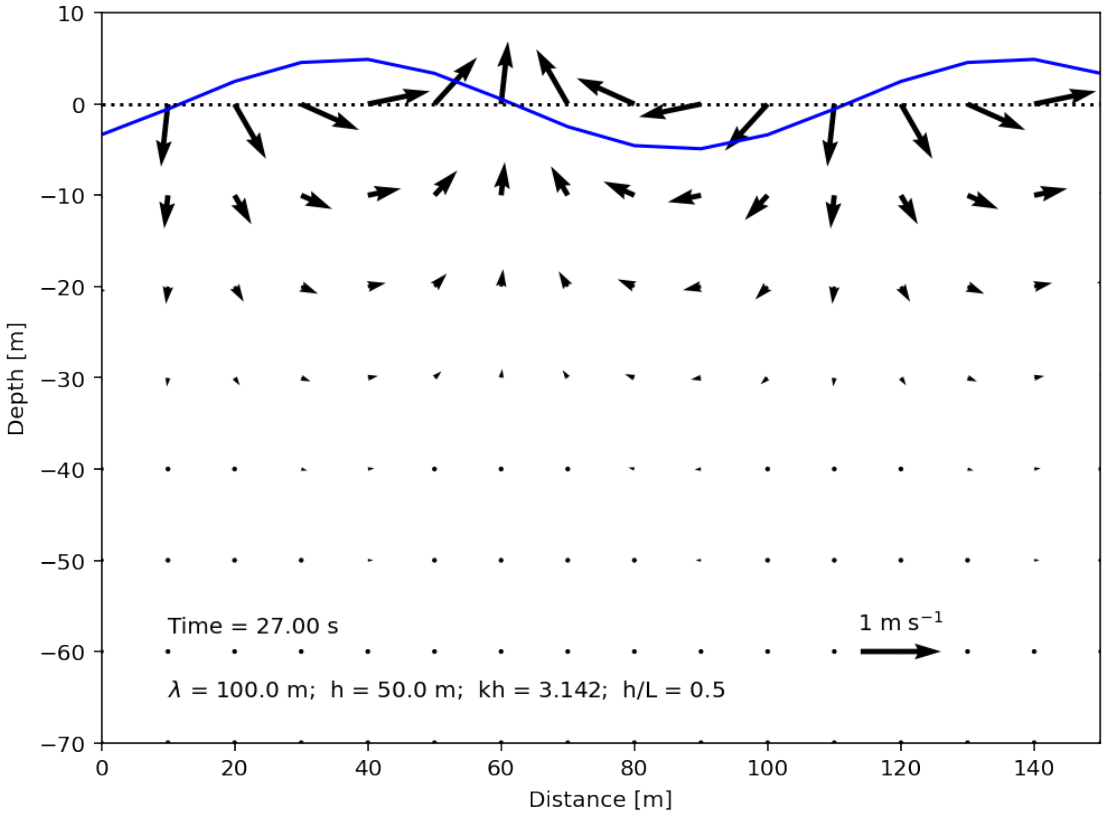
- seaborn : Statistics.
- toyplot : Nice graphes.
- bokeh : Interactive and Server mode.
- pygal : Charting
- Altair : Data science
- plot.ly : Data science and interactive
- Mayavi: 3D
- YT: Astrophysics (volume rendering, contours, particles).
- VisIt: Powerful, easy to use but heavy.
- Paraview: The most-used visualization application. Need high learning effort.
- PyVista: 3D plotting and mesh analysis through a streamlined interface for the Visualization Toolkit (VTK)
- Yellowbrick : Yellowbrick: Machine Learning Visualization

In [22]:
```
#example from Filipe Fernandes
#http://nbviewer.jupyter.org/gist/ocefpaf/9730c697819e91b99f1d694983e39a8f
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import animation

g = 9.81
denw = 1025.0  # Seawater density [kg/m**3].
sig = 7.3e-2  # Surface tension [N/m].
a = 1.0  # Wave amplitude [m].

L, h = 100.0, 50.0  # Wave height and water column depth.
k = 2 * np.pi / L
omega = np.sqrt((g * k + (sig / denw) * (k**3)) * np.tanh(k * h))
T = 2 * np.pi / omega
c = np.sqrt((g / k + (sig / denw) * k) * np.tanh(k * h))

# We'll solve the wave velocities in the `x` and `z` directions.
x, z = np.meshgrid(np.arange(0, 160, 10), np.arange(0, -80, -10),)
u, w = np.zeros_like(x), np.zeros_like(z)


def compute_vel(phase):
    u = a * omega * (np.cosh(k * (z+h)) / np.sinh(k*h)) * np.cos(k * x - phase)
    w = a * omega * (np.sinh(k * (z+h)) / np.sinh(k*h)) * np.sin(k * x - phase)
    mask = -z > h
    u[mask] = 0.0
    w[mask] = 0.0
    return u, w


def basic_animation(frames=91, interval=30, dt=0.3):
    fig = plt.figure(figsize=(8, 6))
    ax = plt.axes(xlim=(0, 150), ylim=(-70, 10))

    # Animated.
    quiver = ax.quiver(x, z, u, w, units='inches', scale=2)
    ax.quiverkey(quiver, 120, -60, 1,
                 label=r'1 m s$^{-1}$',
                 coordinates='data')
    line, = ax.plot([], [], 'b')

    # Non-animated.
    ax.plot([0, 150], [0, 0], 'k:')
```

```
    ax.set_ylabel('Depth [m]')
    ax.set_xlabel('Distance [m]')
    text = (r'$\lambda$ = %s m;  h = %s m;  kh = %2.3f;  h/L = %s' %
            (L, h, k * h, h/L))
    ax.text(10, -65, text)
    time_step = ax.text(10, -58, '')
    line.set_data([], [])

    def init():
        return line, quiver, time_step

    def animate(i):
        time = i * dt
        phase = omega * time
        eta = a * np.cos(x[0] * k - phase)
        u, w = compute_vel(phase)
        quiver.set_UVC(u, w)
        line.set_data(x[0], 5 * eta)
        time_step.set_text('Time = {:.2f} s'.format(time))
        return line, quiver, time_step

    return animation.FuncAnimation(fig, animate, init_func=init,
                                   frames=frames, interval=interval)
```

In [23]:
```
from IPython.display import HTML

HTML(basic_animation(dt=0.3).to_jshtml())
```

Out[23]: <IPython.core.display.HTML object>



12.16 References

- Simple examples with increasing difficulty https://matplotlib.org/examples/index.html
- Gallery https://matplotlib.org/gallery.html
- A matplotlib tutorial, part of the Lectures on Scientific Computing with Python by J.R. Johansson.
- NumPy Beginner | SciPy 2016 Tutorial | Alexandre Chabot LeClerc
- matplotlib tutorial by Nicolas Rougier from LORIA.
- 10 Useful Python Data Visualization Libraries for Any Discipline

Chapter 13

What provide Numpy to Python ?

- `ndarray` multi-dimensional array object
- derived objects such as masked arrays and matrices
- `ufunc` fast array mathematical operations.
- Offers some Matlab-ish capabilities within Python
- Initially developed by Travis Oliphant.
- Numpy 1.0 released October, 2006.
- The SciPy.org website is very helpful.
- NumPy fully supports an object-oriented approach.

13.1 Routines for fast operations on arrays.

```
- shape manipulation
- sorting
- I/O
- FFT
- basic linear algebra
- basic statistical operations
- random simulation
- statistics
- and much more...
```

13.2 Getting Started with NumPy

- It is handy to import everything from NumPy into a Python console:

  ```
  from numpy import *
  ```

- But it is easier to read and debug if you use explicit imports.

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

```
In [1]: import numpy as np
        print(np.__version__)
```

```
1.19.1
```

13.3 Why Arrays ?

- Python lists are slow to process and use a lot of memory.
- For tables, matrices, or volumetric data, you need lists of lists of lists... which becomes messy to program.

```
In [2]: from random import random
        from operator import truediv

In [3]: l1 = [random() for i in range(1000)]
        l2 = [random() for i in range(1000)]
        %timeit s = sum(map(truediv,l1,l2))
```

```
33.7 µs ± 423 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

```
In [4]: a1 = np.array(l1)
        a2 = np.array(l2)
        %timeit s = np.sum(a1/a2)
```

```
9.4 µs ± 79.7 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

13.4 Numpy Arrays: The `ndarray` class.

- There are important differences between NumPy arrays and Python lists:
  - NumPy arrays have a fixed size at creation.
  - NumPy arrays elements are all required to be of the same data type.
  - NumPy arrays operations are performed in compiled code for performance.
- Most of today's scientific/mathematical Python-based software use NumPy arrays.
- NumPy gives us the code simplicity of Python, but the operation is speedily executed by pre-compiled C code.

```
In [5]: a = np.array([0,1,2,3])  #  list
        b = np.array((4,5,6,7))  #  tuple
        c = np.matrix('8 9 0 1') #  string (matlab syntax)

In [6]: print(a,b,c)
```

```
[0 1 2 3] [4 5 6 7] [[8 9 0 1]]
```

13.5 Element wise operations are the "default mode"

```
In [7]: a*b,a+b
```

```
Out[7]: (array([ 0,  5, 12, 21]), array([ 4,  6,  8, 10]))
```

```
In [8]: 5*a, 5+a
```

```
Out[8]: (array([ 0,  5, 10, 15]), array([5, 6, 7, 8]))
```

```
In [9]: a @ b, np.dot(a,b)  # Matrix multiplication
```

```
Out[9]: (38, 38)
```

13.6 NumPy Arrays Properties

```
In [10]: a = np.array([1,2,3,4,5]) # Simple array creation

In [11]: type(a) # Checking the type

Out[11]: numpy.ndarray

In [12]: a.dtype # Print numeric type of elements

Out[12]: dtype('int64')

In [13]: a.itemsize # Print Bytes per element

Out[13]: 8

In [14]: a.shape # returns a tuple listing the length along each dimension

Out[14]: (5,)

In [15]: np.size(a), a.size # returns the entire number of elements.

Out[15]: (5, 5)

In [16]: a.ndim  # Number of dimensions

Out[16]: 1

In [17]: a.nbytes # Memory used

Out[17]: 40
```

- ** Always use `shape` or `size` for numpy arrays instead of `len` **
- `len` gives same information only for 1d array.

13.7 Functions to allocate arrays

```
In [18]: x = np.zeros((2,),dtype=('i4,f4,a10'))
         x

Out[18]: array([(0, 0., b''), (0, 0., b'')],
               dtype=[('f0', '<i4'), ('f1', '<f4'), ('f2', 'S10')])
```

`empty, empty_like, ones, ones_like, zeros, zeros_like, full, full_like`

13.8 Setting Array Elements Values

```
In [19]: a = np.array([1,2,3,4,5])
         print(a.dtype)

int64


In [20]: a[0] = 10 # Change first item value
         a, a.dtype

Out[20]: (array([10,  2,  3,  4,  5]), dtype('int64'))

In [21]: a.fill(0) # slighty faster than a[:] = 0
         a

Out[21]: array([0, 0, 0, 0, 0])
```

13.9 Setting Array Elements Types

```
In [22]: b = np.array([1,2,3,4,5.0]) # Last item is a float
         b, b.dtype

Out[22]: (array([1., 2., 3., 4., 5.]), dtype('float64'))

In [23]: a.fill(3.0)  # assigning a float into a int array
         a[1] = 1.5   # truncates the decimal part
         print(a.dtype, a)

int64 [3 1 3 3 3]


In [24]: a.astype('float64') # returns a new array containing doubles

Out[24]: array([3., 1., 3., 3., 3.])

In [25]: np.asfarray([1,2,3,4]) # Return an array converted to a float type

Out[25]: array([1., 2., 3., 4.])
```

13.10 Slicing x[lower:upper:step]

- Extracts a portion of a sequence by specifying a lower and upper bound.
- The lower-bound element is included, but the upper-bound element is **not** included.
- The default step value is 1 and can be negative.

```
In [26]: a = np.array([10,11,12,13,14])

In [27]: a[:2], a[-5:-3], a[0:2], a[-2:] # negative indices work

Out[27]: (array([10, 11]), array([10, 11]), array([10, 11]), array([13, 14]))

In [28]: a[::2], a[::-1]

Out[28]: (array([10, 12, 14]), array([14, 13, 12, 11, 10]))
```

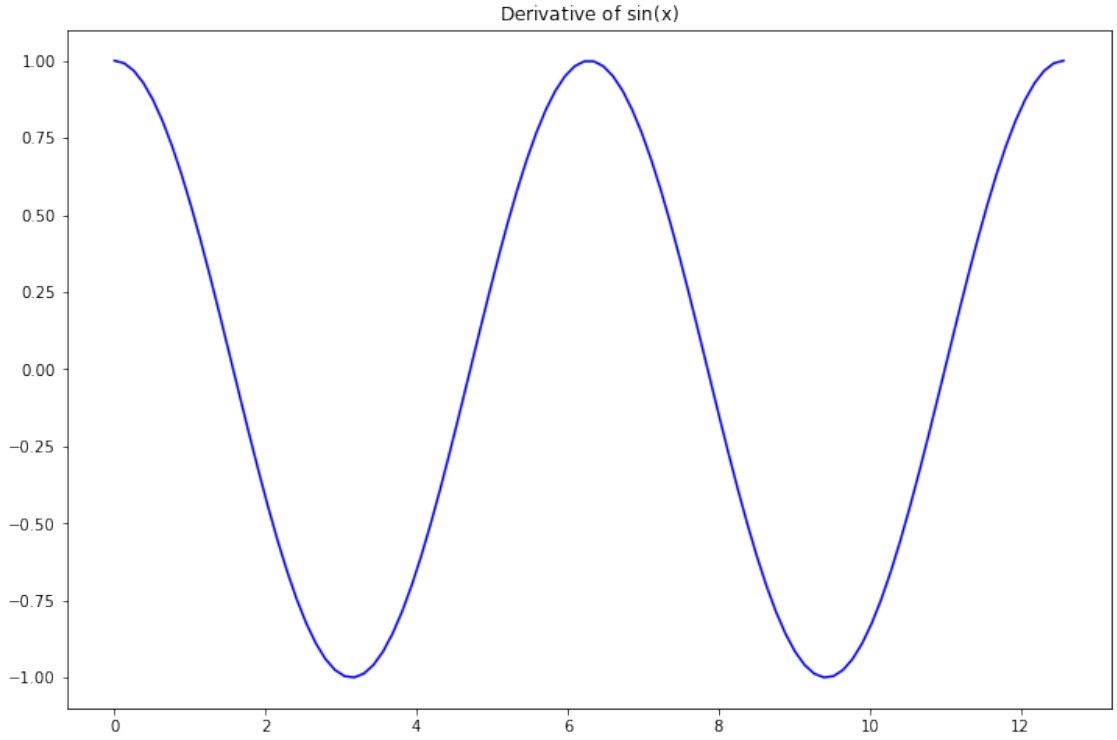
13.10.1 Exercise:

- Compute derivative of $f(x) = \sin(x)$ with finite difference method.

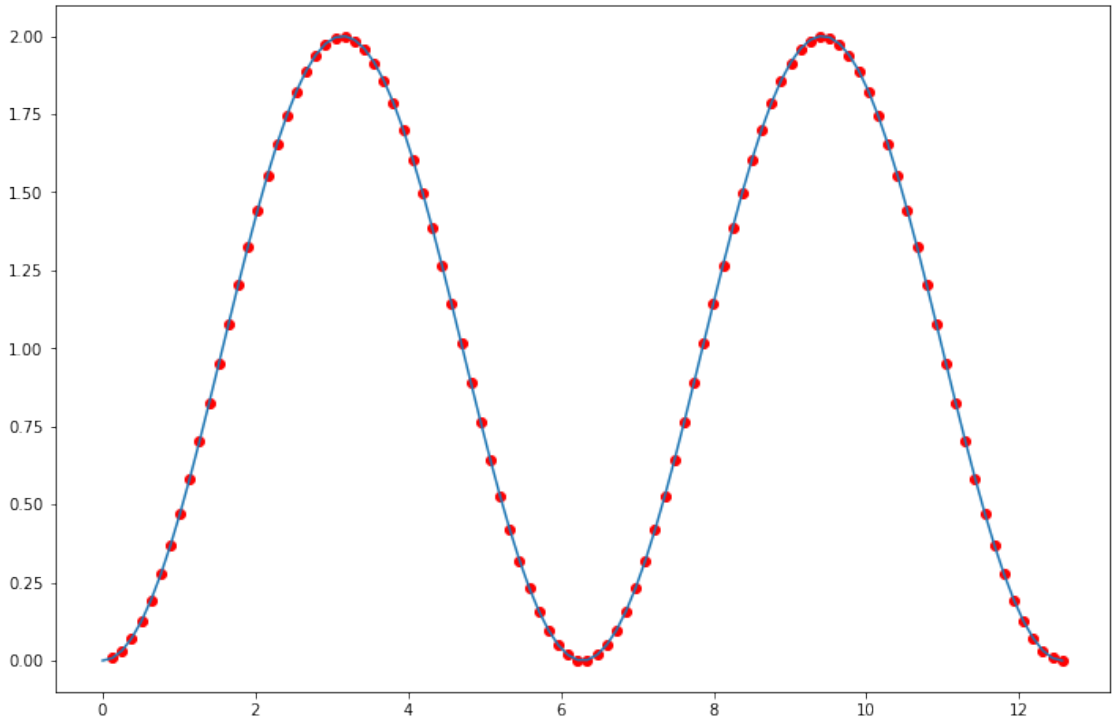
$$\frac{\partial f}{\partial x} \sim \frac{f(x+dx) - f(x)}{dx}$$

  derivatives values are centered in-between sample points.

```
In [29]: x, dx = np.linspace(0,4*np.pi,100, retstep=True)
         y = np.sin(x)

In [30]: %matplotlib inline
         import matplotlib.pyplot as plt
         plt.rcParams['figure.figsize'] = [12.,8.] # Increase plot size
         plt.plot(x, np.cos(x),'b')
         plt.title(r"$\rm{Derivative\ of}\ \sin(x)$");
```



In [31]:
```
# Compute integral of x numerically
avg_height = 0.5*(y[1:]+y[:-1])
int_sin = np.cumsum(dx*avg_height)
plt.plot(x[1:], int_sin, 'ro', x, np.cos(0)-np.cos(x));
```



13.11 Multidimensional array

```
In [32]: a = np.arange(4*3).reshape(4,3) # NumPy array
         l = [[0,1,2],[3,4,5],[6,7,8],[9,10,11]] # Python List

In [33]: print(a)
         print(l)

[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
[[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 10, 11]]


In [34]: l[-1][-1] # Access to last item

Out[34]: 11

In [35]: print(a[-1,-1])  # Indexing syntax is different with NumPy array
         print(a[0,0])    # returns the first item
         print(a[1,:])    # returns the second line

11
0
[3 4 5]


In [36]: print(a[1]) # second line with 2d array
         print(a[:,-1])  # last column

[3 4 5]
[ 2  5  8 11]
```

13.11.1 Exercise

- We compute numerically the Laplace Equation Solution using Finite Difference Method
- Replace the computation of the discrete form of Laplace equation with numpy arrays

$$T_{i,j} = \frac{1}{4}(T_{i+1,j} + T_{i-1,j} + T_{i,j+1} + T_{i,j-1})$$

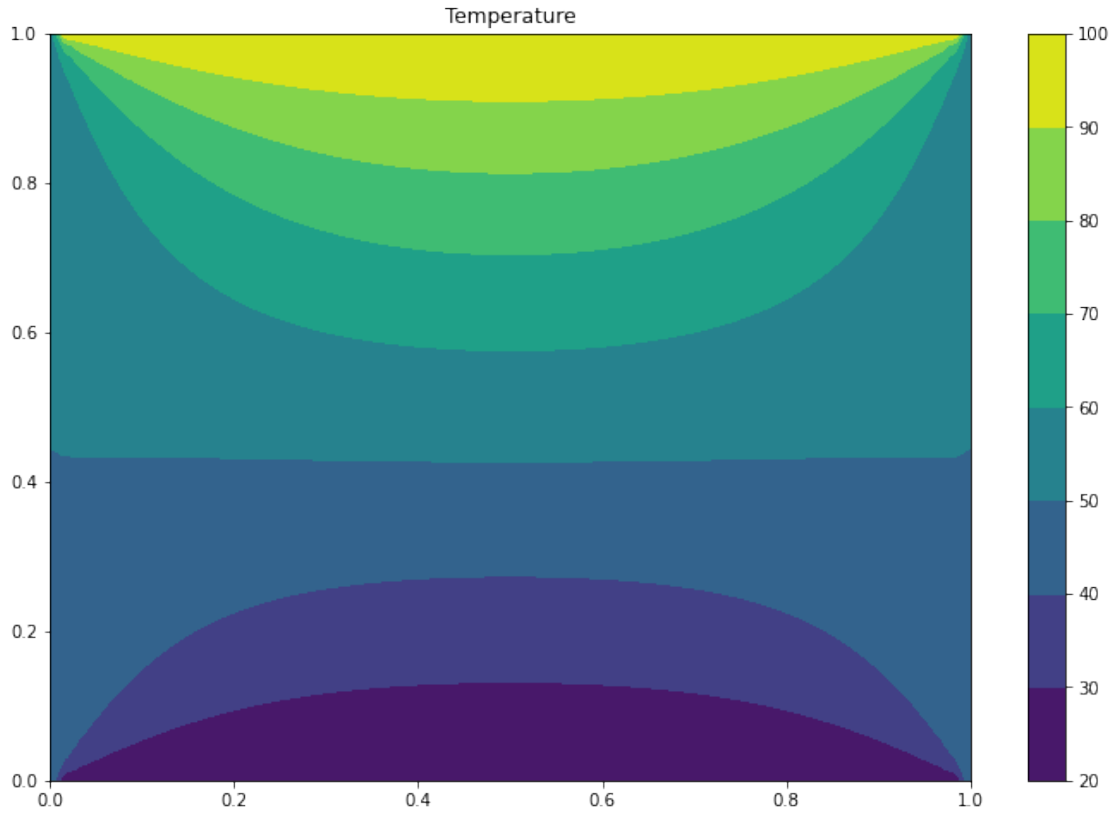
- The function numpy.allclose can help you to compute the residual.

```
In [37]: %%time
         # Boundary conditions
         Tnorth, Tsouth, Twest, Teast = 100, 20, 50, 50

         # Set meshgrid
         n, l = 64, 1.0
         X, Y = np.meshgrid(np.linspace(0,l,n), np.linspace(0,l,n))
         T = np.zeros((n,n))

         # Set Boundary condition
```

```
T[n-1:, :] = Tnorth
T[:1, :]   = Tsouth
T[:, n-1:] = Teast
T[:, :1]   = Twest

residual = 1.0
istep = 0
while residual > 1e-5 :
    istep += 1
    print ((istep, residual), end="\r")
    residual = 0.0
    for i in range(1, n-1):
        for j in range(1, n-1):
            T_old = T[i,j]
            T[i, j] = 0.25 * (T[i+1,j] + T[i-1,j] + T[i,j+1] + T[i,j-1])
            if T[i,j]>0:
                residual=max(residual,abs((T_old-T[i,j])/T[i,j]))


print()
print("iterations = ",istep)
plt.title("Temperature")
plt.contourf(X, Y, T)
plt.colorbar()
```

```
(2457, 1.0022293826789268e-05)
iterations =  2457
CPU times: user 38.4 s, sys: 238 ms, total: 38.6 s
Wall time: 38.3 s
```

Out[37]: <matplotlib.colorbar.Colorbar at 0x7fd86bfcf520>



13.12 Arrays to ASCII files

```
In [38]: x = y = z = np.arange(0.0,5.0,1.0)

In [39]: np.savetxt('test.out', (x,y,z), delimiter=',')   # X is an array
         %cat test.out
```

```
0.0000000000000000e+00,1.0000000000000000e+00,2.0000000000000000e+00,3.0000000000000000e+00,4.00
0.0000000000000000e+00,1.0000000000000000e+00,2.0000000000000000e+00,3.0000000000000000e+00,4.00
0.0000000000000000e+00,1.0000000000000000e+00,2.0000000000000000e+00,3.0000000000000000e+00,4.00
```

```
In [40]: np.savetxt('test.out', (x,y,z), fmt='%1.4e')   # use exponential notation
         %cat test.out
```

```
0.0000e+00 1.0000e+00 2.0000e+00 3.0000e+00 4.0000e+00
0.0000e+00 1.0000e+00 2.0000e+00 3.0000e+00 4.0000e+00
0.0000e+00 1.0000e+00 2.0000e+00 3.0000e+00 4.0000e+00
```

13.13 Arrays from ASCII files

```
In [41]: np.loadtxt('test.out')
```

```
Out[41]: array([[0., 1., 2., 3., 4.],
                [0., 1., 2., 3., 4.],
                [0., 1., 2., 3., 4.]])
```

- save: Save an array to a binary file in NumPy .npy format
- savez : Save several arrays into an uncompressed .npz archive
- savez_compressed: Save several arrays into a compressed .npz archive
- load: Load arrays or pickled objects from .npy, .npz or pickled files.

13.14 H5py

Pythonic interface to the HDF5 binary data format. h5py user manual

```
In [42]: import h5py as h5

         with h5.File('test.h5','w') as f:
             f['x'] = x
             f['y'] = y
             f['z'] = z

In [43]: with h5.File('test.h5','r') as f:
             for field in f.keys():
                 print(field+':',f[field].value)

x: [0. 1. 2. 3. 4.]
y: [0. 1. 2. 3. 4.]
z: [0. 1. 2. 3. 4.]


<ipython-input-43-7b8ae0576ec5>:3: H5pyDeprecationWarning: dataset.value has been deprecated. Use datas
  print(field+':',f[field].value)
```

13.15 Slices Are References

- Slices are references to memory in the original array.
- Changing values in a slice also changes the original array.

```
In [44]: a = np.arange(10)
         b = a[3:6]
         b  # `b` is a view of array `a` and `a` is called base of `b`

Out[44]: array([3, 4, 5])

In [45]: b[0] = -1
         a  # you change a view the base is changed.

Out[45]: array([ 0,  1,  2, -1,  4,  5,  6,  7,  8,  9])
```

- Numpy does not copy if it is not necessary to save memory.

```
In [46]: c = a[7:8].copy() # Explicit copy of the array slice
         c[0] = -1
         a

Out[46]: array([ 0,  1,  2, -1,  4,  5,  6,  7,  8,  9])
```

13.16 Fancy Indexing

```
In [47]: a = np.fromfunction(lambda i, j: (i+1)*10+j, (4, 5), dtype=int)
         a

Out[47]: array([[10, 11, 12, 13, 14],
                [20, 21, 22, 23, 24],
                [30, 31, 32, 33, 34],
                [40, 41, 42, 43, 44]])

In [48]: np.random.shuffle(a.flat) # shuffle modify only the first axis
         a

Out[48]: array([[10, 21, 11, 42, 32],
                [20, 13, 34, 22, 14],
                [30, 40, 43, 12, 41],
                [33, 24, 31, 44, 23]])

In [49]: locations = a % 3 == 0 # locations can be used as a mask
         a[locations] = 0 #set to 0 only the values that are divisible by 3
         a

Out[49]: array([[10,  0, 11,  0, 32],
                [20, 13, 34, 22, 14],
                [ 0, 40, 43,  0, 41],
                [ 0,  0, 31, 44, 23]])

In [50]: a += a == 0
         a

Out[50]: array([[10,  1, 11,  1, 32],
                [20, 13, 34, 22, 14],
                [ 1, 40, 43,  1, 41],
                [ 1,  1, 31, 44, 23]])
```

13.16.1 numpy.take

```
In [51]: a[1:3,2:5]

Out[51]: array([[34, 22, 14],
                [43,  1, 41]])

In [52]: np.take(a,[[6,7],[10,11]])  # Use flatten array indices

Out[52]: array([[13, 34],
                [ 1, 40]])
```

13.17 Changing array shape

```
In [53]: grid = np.indices((2,3)) # Return an array representing the indices of a grid.
         grid[0]

Out[53]: array([[0, 0, 0],
                [1, 1, 1]])

In [54]: grid[1]

Out[54]: array([[0, 1, 2],
                [0, 1, 2]])
```

```
In [55]: grid.flat[:] # Return a view of grid array

Out[55]: array([0, 0, 0, 1, 1, 1, 0, 1, 2, 0, 1, 2])

In [56]: grid.flatten() # Return a copy

Out[56]: array([0, 0, 0, 1, 1, 1, 0, 1, 2, 0, 1, 2])

In [57]: np.ravel(grid, order='C') # A copy is made only if needed.

Out[57]: array([0, 0, 0, 1, 1, 1, 0, 1, 2, 0, 1, 2])
```

13.18 Sorting

```
In [58]: a=np.array([5,3,6,1,6,7,9,0,8])
         np.sort(a) #. Return a view

Out[58]: array([0, 1, 3, 5, 6, 6, 7, 8, 9])

In [59]: a

Out[59]: array([5, 3, 6, 1, 6, 7, 9, 0, 8])

In [60]: a.sort() # Change the array inplace
         a

Out[60]: array([0, 1, 3, 5, 6, 6, 7, 8, 9])
```

13.19 Transpose-like operations

```
In [61]: a = np.array([5,3,6,1,6,7,9,0,8])
         b = a
         b.shape = (3,3) # b is a reference so a will be changed

In [62]: a

Out[62]: array([[5, 3, 6],
                [1, 6, 7],
                [9, 0, 8]])

In [63]: c = a.T # Return a view so a is not changed
         np.may_share_memory(a,c)

Out[63]: True

In [64]: c[0,0] = -1 # c is stored in same memory so change c you change a
         a

Out[64]: array([[-1,  3,  6],
                [ 1,  6,  7],
                [ 9,  0,  8]])

In [65]: c  # is a transposed view of a

Out[65]: array([[-1,  1,  9],
                [ 3,  6,  0],
                [ 6,  7,  8]])

In [66]: b  # b is a reference to a
```

```
Out[66]: array([[-1,  3,  6],
                [ 1,  6,  7],
                [ 9,  0,  8]])

In [67]: c.base  # When the array is not a view `base` return None

Out[67]: array([[-1,  3,  6],
                [ 1,  6,  7],
                [ 9,  0,  8]])
```

13.20 Methods Attached to NumPy Arrays

```
In [68]: a = np.arange(20).reshape(4,5)
         np.random.shuffle(a.flat)
         a

Out[68]: array([[ 7,  8,  9, 12, 17],
                [19, 16,  1, 10,  2],
                [ 3,  6, 11, 18, 15],
                [ 5, 13, 14,  4,  0]])

In [69]: a = (a - a.mean())/ a.std() # Standardize the matrix
         print(a)

[[-0.43355498 -0.26013299 -0.086711    0.43355498  1.30066495]
 [ 1.64750894  1.12724296 -1.47408695  0.086711   -1.30066495]
 [-1.12724296 -0.60697698  0.26013299  1.47408695  0.95382097]
 [-0.78039897  0.60697698  0.78039897 -0.95382097 -1.64750894]]


In [70]: np.set_printoptions(precision=4)
         print(a)

[[-0.4336 -0.2601 -0.0867  0.4336  1.3007]
 [ 1.6475  1.1272 -1.4741  0.0867 -1.3007]
 [-1.1272 -0.607   0.2601  1.4741  0.9538]
 [-0.7804  0.607   0.7804 -0.9538 -1.6475]]


In [71]: a.argmax() # max position in the memory contiguous array

Out[71]: 5

In [72]: np.unravel_index(a.argmax(),a.shape) # get position in the matrix

Out[72]: (1, 0)
```

13.21 Array Operations over a given axis

```
In [73]: a = np.arange(20).reshape(5,4)
         np.random.shuffle(a.flat)

In [74]: a.sum(axis=0) # sum of each column

Out[74]: array([40, 38, 54, 58])

In [75]: np.apply_along_axis(sum, axis=0, arr=a)
```

```
Out[75]: array([40, 38, 54, 58])

In [76]: np.apply_along_axis(sorted, axis=0, arr=a)

Out[76]: array([[ 4,  2,  1,  0],
                [ 5,  3,  7, 10],
                [ 6,  8, 14, 11],
                [12,  9, 15, 18],
                [13, 16, 17, 19]])
```

You can replace the `sorted` builtin fonction by a user defined function.

```
In [77]: np.empty(10)

Out[77]: array([0., 0., 1., 0., 1., 1., 0., 1., 0., 0.])

In [78]: np.linspace(0,2*np.pi,10)

Out[78]: array([0.    , 0.6981, 1.3963, 2.0944, 2.7925, 3.4907, 4.1888, 4.8869,
                5.5851, 6.2832])

In [79]: np.arange(0,2.+0.4,0.4)

Out[79]: array([0. , 0.4, 0.8, 1.2, 1.6, 2. ])

In [80]: np.eye(4)

Out[80]: array([[1., 0., 0., 0.],
                [0., 1., 0., 0.],
                [0., 0., 1., 0.],
                [0., 0., 0., 1.]])

In [81]: a = np.diag(range(4))
         a

Out[81]: array([[0, 0, 0, 0],
                [0, 1, 0, 0],
                [0, 0, 2, 0],
                [0, 0, 0, 3]])

In [82]: a[:,:,np.newaxis]

Out[82]: array([[[0],
                 [0],
                 [0],
                 [0]],

                [[0],
                 [1],
                 [0],
                 [0]],

                [[0],
                 [0],
                 [2],
                 [0]],

                [[0],
                 [0],
                 [0],
                 [3]]])
```

13.21.1 Create the following arrays

```
[100 101 102 103 104 105 106 107 108 109]
```

Hint: numpy.arange

```
[-2. -1.8 -1.6 -1.4 -1.2 -1. -0.8 -0.6 -0.4 -0.2 0.
0.2 0.4 0.6 0.8 1. 1.2 1.4 1.6 1.8]
```

Hint: numpy.linspace

```
[[ 0.001    0.00129155 0.0016681 0.00215443 0.00278256
     0.003593810.00464159 0.00599484 0.00774264 0.01]
```

Hint: numpy.logspace

```
[[ 0. 0. -1. -1. -1.]
 [ 0. 0.  0. -1. -1.]
 [ 0. 0.  0.  0. -1.]
 [ 0. 0.  0.  0.  0.]
 [ 0. 0.  0.  0.  0.]
 [ 0. 0.  0.  0.  0.]
 [ 0. 0.  0.  0.  0.]]
```

Hint: numpy.tri, numpy.zeros, numpy.transpose

```
[[ 0.  1.  2.  3. 4.]
 [-1.  0.  1.  2. 3.]
 [-1. -1.  0.  1. 2.]
 [-1. -1. -1.  0. 1.]
 [-1. -1. -1. -1. 0.]]
```

Hint: numpy.ones, numpy.diag

- Compute the integral numerically with Trapezoidal rule

$$I = \int_{-\infty}^{\infty} e^{-v^2} dv$$

with $v \in [-10; 10]$ and n=20.

13.22 Views and Memory Management

- If it exists one view of a NumPy array, it can be destroyed.

```
In [83]: big = np.arange(1000000)
         small = big[:5]
         del big
         small.base
```

```
Out[83]: array([     0,      1,      2, …, 999997, 999998, 999999])
```

- Array called `big` is still allocated.
- Sometimes it is better to create a copy.

```
In [84]: big = np.arange(1000000)
         small = big[:5].copy()
         del big
         print(small.base)
```

```
None
```

13.22.1 Change memory alignement

```
In [85]: del(a)
         a = np.arange(20).reshape(5,4)
         print(a.flags)
```

```
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : False
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False
  UPDATEIFCOPY : False
```

```
In [86]: b = np.asfortranarray(a) # makes a copy
         b.flags
```

```
Out[86]:   C_CONTIGUOUS : False
           F_CONTIGUOUS : True
           OWNDATA : True
           WRITEABLE : True
           ALIGNED : True
           WRITEBACKIFCOPY : False
           UPDATEIFCOPY : False
```

```
In [87]: b.base is a
```

```
Out[87]: False
```

You can also create a fortran array with array function.

```
In [88]: c = np.array([[1,2,3],[4,5,6]])
         f = np.asfortranarray(c)
```

```
In [89]: print(f.ravel(order='K')) # Return a 1D array using memory order
         print(c.ravel(order='K')) # Copy is made only if necessary
```

```
[1 4 2 5 3 6]
[1 2 3 4 5 6]
```

13.23 Broadcasting rules

Broadcasting rules allow you to make an outer product between two vectors: the first method involves array tiling, the second one involves broadcasting. The last method is significantly faster.

```
In [90]: n = 1000
         a = np.arange(n)
         ac = a[:, np.newaxis]   # column matrix
         ar = a[np.newaxis, :]   # row matrix
```

```
In [91]: %timeit np.tile(a, (n,1)).T * np.tile(a, (n,1))
```

```
9.93 ms ± 281 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
In [92]: %timeit ac * ar
```

```
1.66 ms ± 8.41 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

```
In [93]: np.all(np.tile(a, (n,1)).T * np.tile(a, (n,1)) == ac * ar)
```

```
Out[93]: True
```

13.24 Numpy Matrix

Specialized 2-D array that retains its 2-D nature through operations. It has certain special operators, such as $*$ (matrix multiplication) and $**$ (matrix power).

```
In [94]: m = np.matrix('1 2; 3 4') #Matlab syntax
         m
```

```
Out[94]: matrix([[1, 2],
                 [3, 4]])
```

```
In [95]: a = np.matrix([[1, 2],[ 3, 4]]) #Python syntax
         a
```

```
Out[95]: matrix([[1, 2],
                 [3, 4]])
```

```
In [96]: a = np.arange(1,4)
         b = np.mat(a) # 2D view, no copy!
         b, np.may_share_memory(a,b)
```

```
Out[96]: (matrix([[1, 2, 3]]), True)
```

```
In [97]: a = np.matrix([[1, 2, 3],[ 3, 4, 5]])
         a * b.T # Matrix vector product
```

```
Out[97]: matrix([[14],
                 [26]])
```

```
In [98]: m * a # Matrix multiplication
```

```
Out[98]: matrix([[ 7, 10, 13],
                 [15, 22, 29]])
```

13.25 StructuredArray using a compound data type specification

```
In [99]: data = np.zeros(4, dtype={'names':('name', 'age', 'weight'),
                                   'formats':('U10', 'i4', 'f8')})
         print(data.dtype)
```

```
[('name', '<U10'), ('age', '<i4'), ('weight', '<f8')]
```

```
In [100]: data['name'] = ['Pierre', 'Paul', 'Jacques', 'Francois']
          data['age'] = [45, 10, 71, 39]
          data['weight'] = [95.0, 75.0, 88.0, 71.0]
          print(data)
```

```
[('Pierre', 45, 95.) ('Paul', 10, 75.) ('Jacques', 71, 88.)
 ('Francois', 39, 71.)]
```

13.26 RecordArray

```
In [101]: data_rec = data.view(np.recarray)
          data_rec.age
```

```
Out[101]: array([45, 10, 71, 39], dtype=int32)
```

13.27 NumPy Array Programming

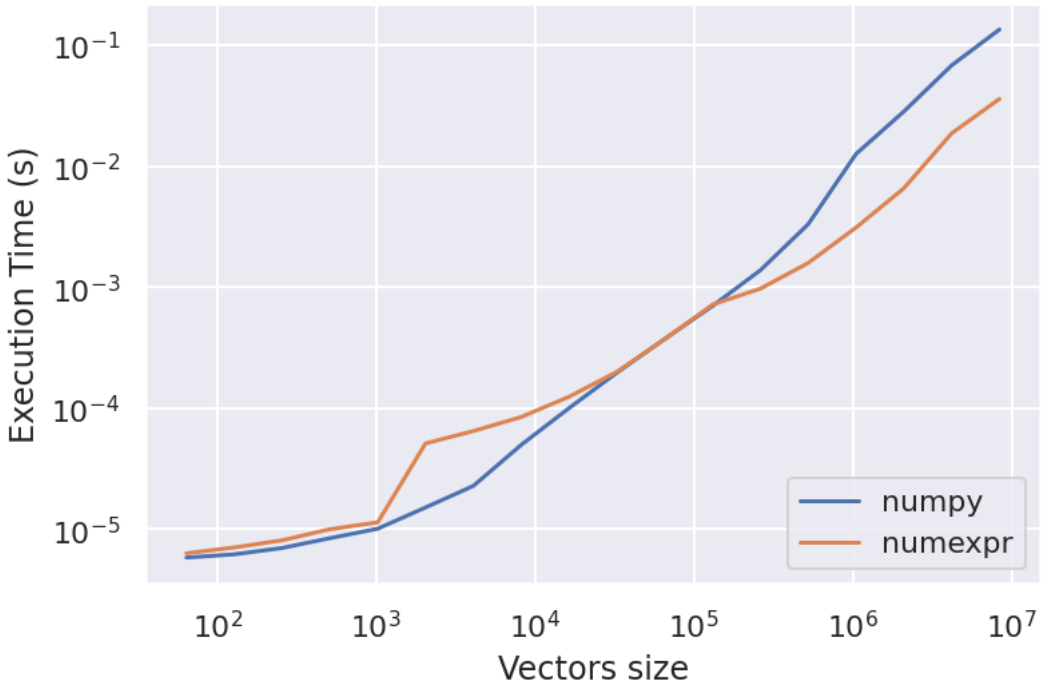
- Array operations are fast, Python loops are slow.
- Top priority: **avoid loops**
- It's better to do the work three times witharray operations than once with a loop.
- This does require a change of habits.
- This does require some experience.
- NumPy's array operations are designed to make this possible.

13.28 Fast Evaluation Of Array Expressions

- The `numexpr` package supplies routines for the fast evaluation of array expressions elementwise by using a vector-based virtual machine.
- Expressions are cached, so reuse is fast.

Numexpr Users Guide

```
In [102]: import numexpr as ne
          import numpy as np
          nrange = (2 ** np.arange(6, 24)).astype(int)

          t_numpy = []
          t_numexpr = []

          for n in nrange:
              a = np.random.random(n)
              b = np.arange(n, dtype=np.double)
              c = np.random.random(n)

              c1 = ne.evaluate("a ** 2 + b ** 2 + 2 * a * b * c ", optimization='aggressive')

              t1 = %timeit -oq -n 10 a ** 2 + b ** 2 + 2 * a * b * c
              t2 = %timeit -oq -n 10 ne.re_evaluate()

              t_numpy.append(t1.best)
              t_numexpr.append(t2.best)

          %matplotlib inline
          %config InlineBackend.figure_format = 'retina'
          import matplotlib.pyplot as plt
          import seaborn; seaborn.set()

          plt.loglog(nrange, t_numpy, label='numpy')
          plt.loglog(nrange, t_numexpr, label='numexpr')

          plt.legend(loc='lower right')
          plt.xlabel('Vectors size')
          plt.ylabel('Execution Time (s)');
```



13.29 References

- NumPy reference
- Getting the Best Performance out of NumPy
- Numpy by Konrad Hinsen

Chapter 14

Scipy

Scipy is the scientific Python ecosystem : - fft, linear algebra, scientific computation,... - scipy contains numpy, it can be considered as an extension of numpy. - the add-on toolkits Scikits complements scipy.

```
In [1]: %matplotlib inline
        %config InlineBackend.figure_format = 'retina'
        import matplotlib.pyplot as plt
        plt.rcParams['figure.figsize'] = (10,6)
```

```
In [2]: import numpy as np
        import scipy as sp
        np.sqrt(-1.), np.log(-2.)
```

```
<ipython-input-2-5c51392f47b6>:3: RuntimeWarning: invalid value encountered in sqrt
  np.sqrt(-1.), np.log(-2.)
<ipython-input-2-5c51392f47b6>:3: RuntimeWarning: invalid value encountered in log
  np.sqrt(-1.), np.log(-2.)
```

```
Out[2]: (nan, nan)
```

```
In [3]: sp.sqrt(-1.), sp.log(-2.)
```

```
<ipython-input-3-52b38bd19582>:1: DeprecationWarning: scipy.sqrt is deprecated and will be removed in S
  sp.sqrt(-1.), sp.log(-2.)
```

```
<ipython-input-3-52b38bd19582>:1: DeprecationWarning: scipy.log is deprecated and will be removed in Sc
  sp.sqrt(-1.), sp.log(-2.)
```

Out[3]: (1j, (0.6931471805599453+3.141592653589793j))

In [4]:
```
sp.exp(sp.log(-2.))
```

```
<ipython-input-4-021f9e257f98>:1: DeprecationWarning: scipy.log is deprecated and will be removed in Sc
  sp.exp(sp.log(-2.))
<ipython-input-4-021f9e257f98>:1: DeprecationWarning: scipy.exp is deprecated and will be removed in Sc
  sp.exp(sp.log(-2.))
```

Out[4]: (-2+2.4492935982947064e-16j)

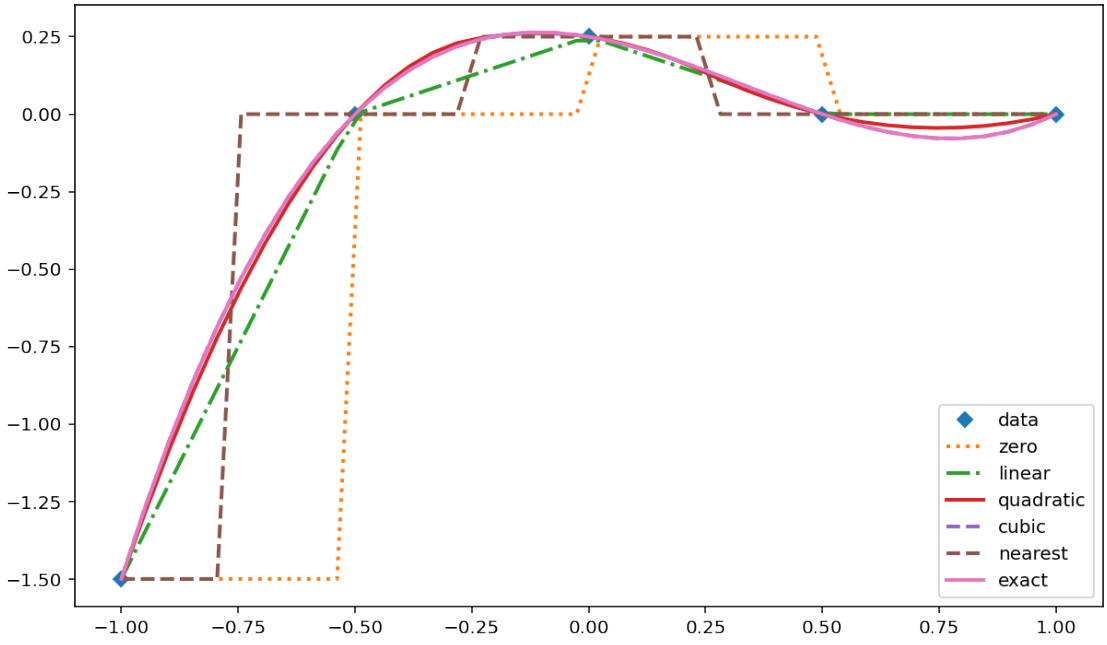
14.1 SciPy main packages

- `constants` : Physical and mathematical constants
- `fftpack` : Fast Fourier Transform routines
- `integrate` : Integration and ordinary differential equation solvers
- `interpolate` : Interpolation and smoothing splines
- `io` : Input and Output
- `linalg` : Linear algebra
- `signal` : Signal processing
- `sparse` : Sparse matrices and associated routines

In [5]:
```
from scipy.interpolate import interp1d
x = np.linspace(-1, 1, num=5)  # 5 points evenly spaced in [-1,1].
y = (x-1.)*(x-0.5)*(x+0.5)     # x and y are numpy arrays
f0 = interp1d(x,y, kind='zero')
f1 = interp1d(x,y, kind='linear')
f2 = interp1d(x,y, kind='quadratic')
f3 = interp1d(x,y, kind='cubic')
f4 = interp1d(x,y, kind='nearest')
```

In [6]:
```
xnew = sp.linspace(-1, 1, num=40)
ynew = (xnew-1.)*(xnew-0.5)*(xnew+0.5)
plt.plot(x,y,'D',xnew,f0(xnew),':', xnew, f1(xnew),'-.',
                 xnew,f2(xnew),'-',xnew ,f3(xnew),'--',
                 xnew,f4(xnew),'--',xnew, ynew, linewidth=2)
plt.legend(['data','zero','linear','quadratic','cubic','nearest','exact'],
           loc='best');
```

```
<ipython-input-6-bb75b01c8201>:1: DeprecationWarning: scipy.linspace is deprecated and will be removed
  xnew = sp.linspace(-1, 1, num=40)
```

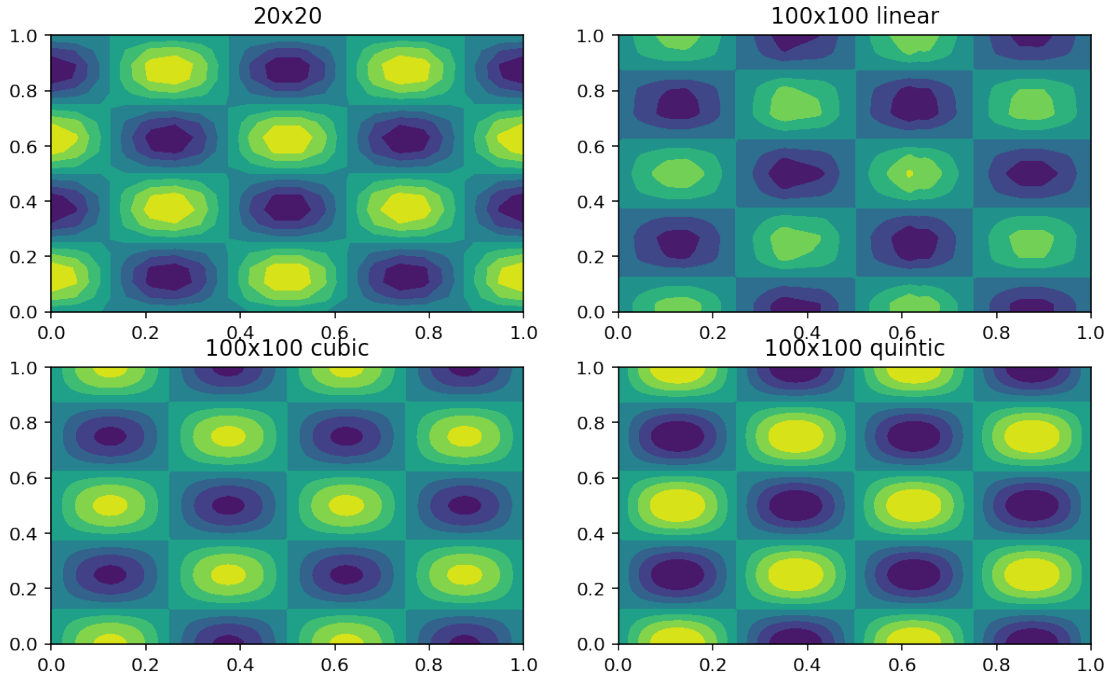



```
In [7]: from scipy.interpolate import interp2d
        x,y=sp.mgrid[0:1:20j,0:1:20j]  #create the grid 20x20
        z=sp.cos(4*sp.pi*x)*sp.sin(4*sp.pi*y) #initialize the field
        T1=interp2d(x,y,z,kind='linear')
        T2=interp2d(x,y,z,kind='cubic')
        T3=interp2d(x,y,z,kind='quintic')
```

```
<ipython-input-7-3b5ca11518b4>:3: DeprecationWarning: scipy.cos is deprecated and will be removed in Sc
  z=sp.cos(4*sp.pi*x)*sp.sin(4*sp.pi*y) #initialize the field
<ipython-input-7-3b5ca11518b4>:3: DeprecationWarning: scipy.sin is deprecated and will be removed in Sc
  z=sp.cos(4*sp.pi*x)*sp.sin(4*sp.pi*y) #initialize the field
```

```
In [8]: X,Y=sp.mgrid[0:1:100j,0:1:100j] #create the interpolation grid 100x100
        # complex -> number of points, float -> step size
        plt.figure(1)
        plt.subplot(221) #Plot original data
        plt.contourf(x,y,z)
        plt.title('20x20')
        plt.subplot(222) #Plot linear interpolation
        plt.contourf(X,Y,T1(X[:,0],Y[0,:]))
        plt.title('100x100 linear')
        plt.subplot(223) #Plot cubic interpolation
        plt.contourf(X,Y,T2(X[:,0],Y[0,:]))
        plt.title('100x100 cubic')
        plt.subplot(224) #Plot quintic interpolation
        plt.contourf(X,Y,T3(X[:,0],Y[0,:]))
        plt.title('100x100 quintic')
```

```
Out[8]: Text(0.5, 1.0, '100x100 quintic')
```



14.2 FFT : scipy.fftpack

- FFT dimension 1, 2 and n : fft, ifft (inverse), rfft (real), irfft, fft2 (dimension 2), ifft2, fftn (dimension n), ifftn.
- Discrete cosinus transform : dct
- Convolution product : convolve

```
In [9]: from numpy.fft import fft, ifft
        x = np.random.random(1024)
        %timeit ifft(fft(x))
```

```
40.7 µs ± 172 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

```
In [10]: from scipy.fftpack import fft, ifft
         x = np.random.random(1024)
         %timeit ifft(fft(x))
```

```
40.7 µs ± 427 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

14.3 Linear algebra : scipy.linalg

- Sovers, decompositions, eigen values. (same as numpy).
- Matrix functions : expm, sinm, sinhm,...

- Block matrices diagonal, triangular, periodic,...

```
In [11]: import scipy.linalg as spl
         b=sp.ones(5)
         A=sp.array([[1.,3.,0., 0.,0.],
                     [ 2.,1.,-4, 0.,0.],
                     [ 6.,1., 2,-3.,0.],
                     [ 0.,1., 4.,-2.,-3.],
                     [ 0.,0., 6.,-3., 2.]])
         print("x=",spl.solve(A,b,sym_pos=False)) # LAPACK ( gesv ou posv )
         AB=sp.array([[0.,3.,-4.,-3.,-3.],
                      [1.,1., 2.,-2., 2.],
                      [2.,1., 4.,-3., 0.],
                      [6.,1., 6., 0., 0.]])
         print("x=",spl.solve_banded((2,1),AB,b)) # LAPACK ( gbsv )
```

```
x= [-0.24074074  0.41358025 -0.26697531 -0.85493827  0.01851852]
x= [-0.24074074  0.41358025 -0.26697531 -0.85493827  0.01851852]
```

```
<ipython-input-11-4af1e65b6e1b>:2: DeprecationWarning: scipy.ones is deprecated and will be removed in S
  b=sp.ones(5)
<ipython-input-11-4af1e65b6e1b>:3: DeprecationWarning: scipy.array is deprecated and will be removed in
  A=sp.array([[1.,3.,0., 0.,0.],
<ipython-input-11-4af1e65b6e1b>:9: DeprecationWarning: scipy.array is deprecated and will be removed in
  AB=sp.array([[0.,3.,-4.,-3.,-3.],
```

```
In [12]: P,L,U = spl.lu(A) #  P A = L U
         np.set_printoptions(precision=3)
         for M in (P,L,U):
             print(M, end="\n"+20*"-"+"\n")
```

```
[[0. 1. 0. 0. 0.]
 [0. 0. 0. 1. 0.]
 [1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1.]
 [0. 0. 1. 0. 0.]]
--------------------
[[ 1.     0.     0.     0.     0.   ]
 [ 0.167  1.     0.     0.     0.   ]
 [ 0.     0.     1.     0.     0.   ]
 [ 0.333  0.235 -0.765  1.     0.   ]
 [ 0.     0.353  0.686  0.083  1.   ]]
--------------------
[[ 6.     1.     2.    -3.     0.   ]
 [ 0.     2.833 -0.333  0.5    0.   ]
 [ 0.     0.     6.    -3.     2.   ]
 [ 0.     0.     0.    -1.412  1.529]
 [ 0.     0.     0.     0.    -4.5  ]]
--------------------
```

14.4 CSC (Compressed Sparse Column)

- All operations are optimized
- Efficient ”slicing” along axis=1.
- Fast Matrix-vector product.

- Conversion to other format could be costly.

```
In [13]: import scipy.sparse as spsp
         row = sp.array([0,2,2,0,1,2])
         col = sp.array([0,0,1,2,2,2])
         data  = sp.array([1,2,3,4,5,6])
         Mcsc1 = spsp.csc_matrix((data,(row,col)),shape=(3,3))
         Mcsc1.todense()
```

```
<ipython-input-13-1852fd1225dd>:2: DeprecationWarning: scipy.array is deprecated and will be removed in
  row = sp.array([0,2,2,0,1,2])
<ipython-input-13-1852fd1225dd>:3: DeprecationWarning: scipy.array is deprecated and will be removed in
  col = sp.array([0,0,1,2,2,2])
<ipython-input-13-1852fd1225dd>:4: DeprecationWarning: scipy.array is deprecated and will be removed in
  data  = sp.array([1,2,3,4,5,6])
```

```
Out[13]: matrix([[1, 0, 4],
                 [0, 0, 5],
                 [2, 3, 6]])
```

```
In [14]: indptr  = sp.array([0,2,3,6])
         indices = sp.array([0,2,2,0,1,2])
         data    = sp.array([1,2,3,4,5,6])
         Mcsc2 = spsp.csc_matrix ((data,indices,indptr),shape=(3,3))
         Mcsc2.todense()
```

```
<ipython-input-14-31b5a43b6ca2>:1: DeprecationWarning: scipy.array is deprecated and will be removed in
  indptr  = sp.array([0,2,3,6])
<ipython-input-14-31b5a43b6ca2>:2: DeprecationWarning: scipy.array is deprecated and will be removed in
  indices = sp.array([0,2,2,0,1,2])
<ipython-input-14-31b5a43b6ca2>:3: DeprecationWarning: scipy.array is deprecated and will be removed in
  data    = sp.array([1,2,3,4,5,6])
```

```
Out[14]: matrix([[1, 0, 4],
                 [0, 0, 5],
                 [2, 3, 6]])
```

14.5 Dedicated format for assembling

- `lil_matrix` : Row-based linked list matrix. Easy format to build your matrix and convert to other format before solving.
- `dok_matrix` : A dictionary of keys based matrix. Ideal format for incremental matrix building. The conversion to csc/csr format is efficient.
- `coo_matrix` : coordinate list format. Fast conversion to formats CSC/CSR.

Lien vers la documentation scipy

14.6 Sparse matrices : scipy.sparse.linalg

- speigen, speigen_symmetric, lobpcg : (ARPACK).
- svd : (ARPACK).
- Direct methods (UMFPACK or SUPERLU) or Krylov based methods
- Minimization : lsqr and minres

For linear algebra: - Noobs: spsolve. - Intermmediate: dsolve.spsolve or isolve.spsolve - Advanced: splu, spilu (direct); cg, cgs, bicg, bicgstab, gmres, lgmres et qmr (iterative) - Boss: petsc4py et slepc4py.

14.7 LinearOperator

The LinearOperator is used for matrix-free numerical methods.

```
In [15]: import scipy.sparse.linalg as spspl
         def mv(v):
            return sp.array([2*v[0],3*v[1]])

         A=spspl.LinearOperator((2 ,2),matvec=mv,dtype=float )
         A
```

```
Out[15]: <2x2 _CustomLinearOperator with dtype=float64>
```

```
In [16]: A*sp.ones(2)
```

```
<ipython-input-16-8911e67c7fda>:1: DeprecationWarning: scipy.ones is deprecated and will be removed in S
  A*sp.ones(2)
<ipython-input-15-b76e52185657>:3: DeprecationWarning: scipy.array is deprecated and will be removed in
  return sp.array([2*v[0],3*v[1]])
```

```
Out[16]: array([2., 3.])
```

```
In [17]: A.matmat(sp.array([[1,-2],[3,6]]))
```

```
<ipython-input-17-80cc6cd3a514>:1: DeprecationWarning: scipy.array is deprecated and will be removed in
  A.matmat(sp.array([[1,-2],[3,6]]))
<ipython-input-15-b76e52185657>:3: DeprecationWarning: scipy.array is deprecated and will be removed in
  return sp.array([2*v[0],3*v[1]])
```

```
Out[17]: array([[ 2, -4],
                [ 9, 18]])
```

14.8 LU decomposition

```
In [18]: N = 50
         un = sp.ones(N)
         w = sp.rand(N+1)
         A = spsp.spdiags([w[1:],-2*un,w[:-1]],[-1,0,1],N,N) # tridiagonal matrix
         A = A.tocsc()
         b = un
         op = spspl.splu(A)
         op
```

```
<ipython-input-18-e97d13ba2dbc>:2: DeprecationWarning: scipy.ones is deprecated and will be removed in S
  un = sp.ones(N)
<ipython-input-18-e97d13ba2dbc>:3: DeprecationWarning: scipy.rand is deprecated and will be removed in S
  w = sp.rand(N+1)
```

```
Out[18]: <SuperLU at 0x7f3885705e40>
```

```
In [19]: x=op.solve(b)
         spl.norm(A*x-b)
```

```
Out[19]: 1.6279065094399114e-15
```

14.9 Conjugate Gradient

In [20]:
```
global k
k=0
def f(xk): # function called at every iterations
     global k
     print ("iteration {0:2d} residu = {1:7.3g}".format(k,spl.norm(A*xk-b)))
     k += 1

x,info=spspl.cg(A,b,x0=sp.zeros(N),tol=1.0e-12,maxiter=N,M=None,callback=f)
```

```
iteration  0 residu =    2.67
iteration  1 residu =    1.03
iteration  2 residu =   0.483
iteration  3 residu =   0.188
iteration  4 residu =  0.0848
iteration  5 residu =  0.0285
iteration  6 residu =  0.0119
iteration  7 residu = 0.00504
iteration  8 residu = 0.00167
iteration  9 residu = 0.000594
iteration 10 residu = 0.000225
iteration 11 residu = 8.78e-05
iteration 12 residu = 3.58e-05
iteration 13 residu = 1.32e-05
iteration 14 residu = 4.45e-06
iteration 15 residu = 1.42e-06
iteration 16 residu = 5.22e-07
iteration 17 residu = 2.23e-07
iteration 18 residu = 7.52e-08
iteration 19 residu = 2.22e-08
iteration 20 residu = 7.7e-09
iteration 21 residu = 2.23e-09
iteration 22 residu = 6.65e-10
iteration 23 residu = 1.95e-10
iteration 24 residu = 5.03e-11
iteration 25 residu = 1.19e-11
iteration 26 residu = 2.52e-12
```

```
<ipython-input-20-45fd1a29b4fe>:8: DeprecationWarning: scipy.zeros is deprecated and will be removed in
  x,info=spspl.cg(A,b,x0=sp.zeros(N),tol=1.0e-12,maxiter=N,M=None,callback=f)
```

14.10 Preconditioned conjugate gradient

In [21]:
```
pc=spspl.spilu(A,drop_tol=0.1)  # pc is an ILU decomposition
xp=pc.solve(b)
spl.norm(A*xp-b)
```

Out[21]: 0.4074669737861188

In [22]:
```
def mv(v):
    return pc.solve(v)
lo = spspl.LinearOperator((N,N),matvec=mv)
k = 0
x,info=spspl.cg(A,b,x0=sp.zeros(N),tol=1.e-12,maxiter=N,M=lo,callback=f)
```

```
iteration  0 residu =   0.356
iteration  1 residu =  0.0133
iteration  2 residu = 0.000589
iteration  3 residu = 2.75e-05
iteration  4 residu = 1.07e-06
iteration  5 residu = 2.98e-08
iteration  6 residu = 2.11e-09
iteration  7 residu = 4.03e-11
iteration  8 residu = 2.66e-12
```

```
<ipython-input-22-dff3da9ff400>:5: DeprecationWarning: scipy.zeros is deprecated and will be removed in
  x,info=spspl.cg(A,b,x0=sp.zeros(N),tol=1.e-12,maxiter=N,M=lo,callback=f)
```

14.11 Numerical integration

- quad, dblquad, tplquad,... Fortran library QUADPACK.

```
In [23]: import scipy.integrate as spi

         x2=lambda x: x**2
         4.**3/3  # int(x2) in [0,4]
```

```
Out[23]: 21.333333333333332
```

```
In [24]: spi.quad(x2,0.,4.)
```

```
Out[24]: (21.333333333333336, 2.368475785867001e-13)
```

14.12 Scipy ODE solver

It uses the Fortran ODEPACK library.

14.12.1 Van der Pol Oscillator

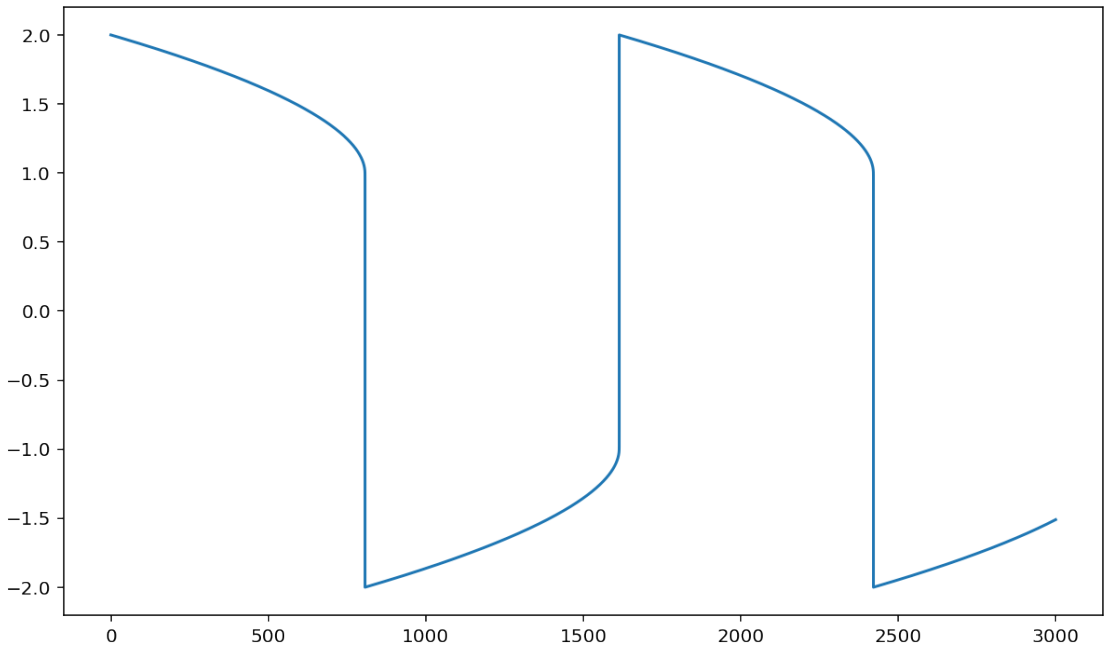
$$
\begin{aligned}
y_1'(t) &= y_2(t), \\
y_2'(t) &= 1000(1 - y_1^2(t))y_2(t) - y_1(t)
\end{aligned}
$$

$$ with $y_1(0) = 2 $ and $ y_2(0) = 0. $.

```
In [25]: import numpy as np
         import scipy.integrate as spi

         def vdp1000(y,t):
              dy=np.zeros(2)
              dy[0]=y[1]
              dy[1]=1000.*(1.-y[0]**2)*y[1]-y[0]
              return dy
```

```
In [26]: t0, tf =0,  3000
         N = 300000
         t, dt = np.linspace(t0,tf,N, retstep=True)
```

```
In [27]: y=spi.odeint(vdp1000,[2.,0.],t)
         plt.plot(t,y[:,0]);
```



14.13 Exercise

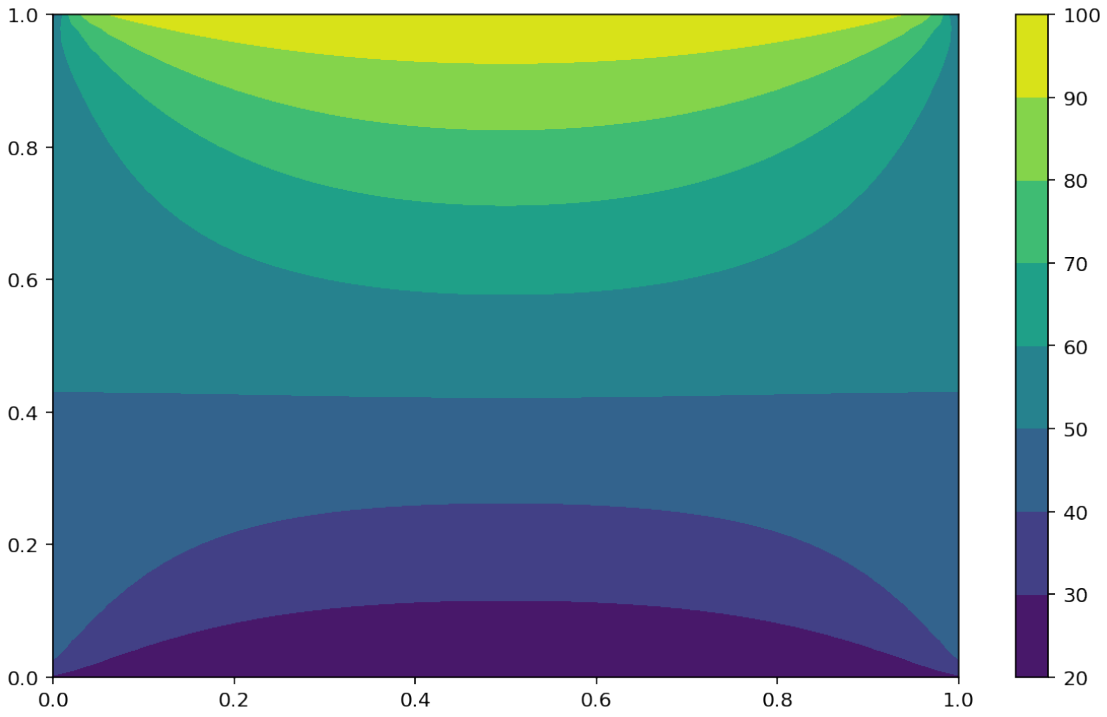
The following code solve the Laplace equation using a dense matrix. - Modified the code to use a sparse matrix

```
In [28]: %%time
         %matplotlib inline
         %config InlineBackend.figure_format = "retina"
         import numpy as np
         import matplotlib.pyplot as plt
         plt.rcParams['figure.figsize'] = (10,6)

         # Boundary conditions
         Tnorth, Tsouth, Twest, Teast = 100, 20, 50, 50

         # Set meshgrid
         n = 50
         l = 1.0
         h = l / (n-1)
         X, Y = np.meshgrid(np.linspace(0,l,n), np.linspace(0,l,n))
         T = np.zeros((n,n),dtype='d')

         # Set Boundary condition
         T[n-1:, :] = Tnorth / h**2
         T[:1, :] = Tsouth / h**2
         T[:, n-1:] = Teast / h**2
         T[:, :1] = Twest / h**2

         A = np.zeros((n*n,n*n),dtype='d')
         nn = n*n
         ii = 0
```

```
        for j in range(n):
            for i in range(n):
              if j > 0:
                 jj = ii - n
                 A[ii,jj] = -1
              if j < n-1:
                 jj = ii + n
                 A[ii,jj] = -1
              if i > 0:
                 jj = ii - 1
                 A[ii,jj] = -1
              if i < n-1:
                 jj = ii + 1
                 A[ii,jj] = -1
              A[ii,ii] = 4
              ii = ii+1


        U = np.linalg.solve(A,np.ravel(h**2*T))
        T = U.reshape(n,n)
        plt.contourf(X,Y,T)
        plt.colorbar()
```

```
CPU times: user 478 ms, sys: 44 ms, total: 522 ms
Wall time: 290 ms
```

Out[28]: <matplotlib.colorbar.Colorbar at 0x7f38843eb370>

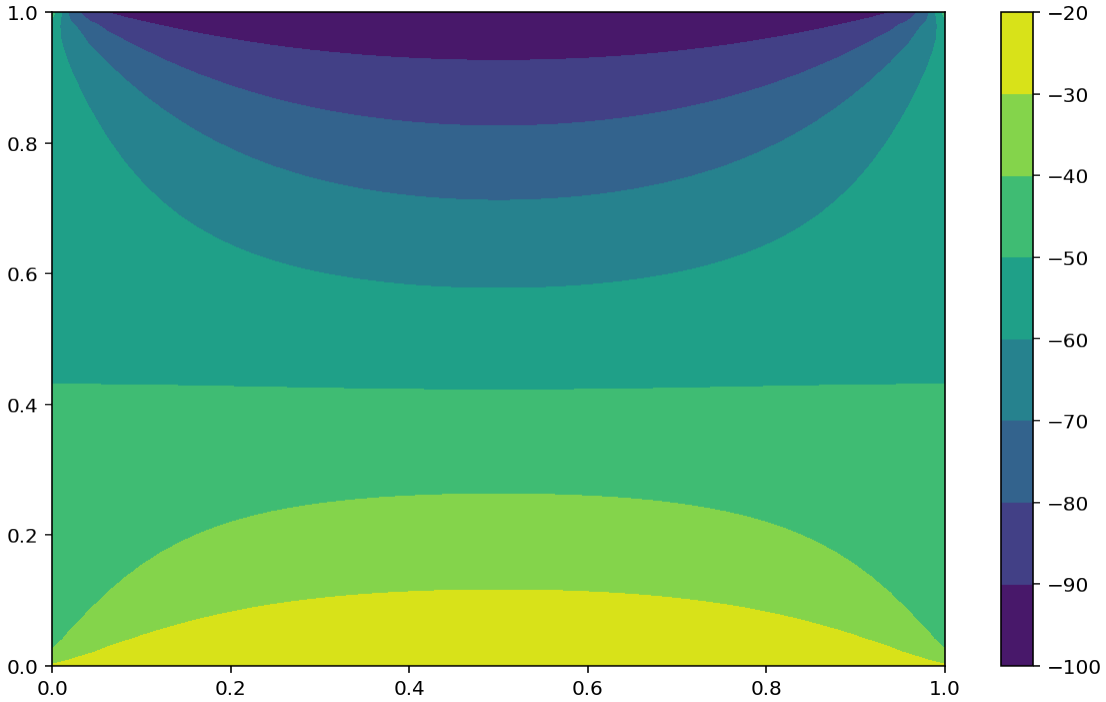


In [29]:
```
        %%time
        import scipy.sparse as spsp
```

```
import scipy.sparse.linalg as spspl

# Boundary conditions
Tnorth, Tsouth, Twest, Teast = 100, 20, 50, 50

# Set meshgrid
n = 50
l = 1.0
h = l / (n-1)
X, Y = np.meshgrid(np.linspace(0,l,n), np.linspace(0,l,n))
T = np.zeros((n,n),dtype='d')

# Set Boundary condition
T[n-1:, :]    = Tnorth / h**2
T[  :1, :]    = Tsouth / h**2
T[   :, n-1:] = Teast  / h**2
T[   :, :1]   = Twest  / h**2

bdiag = -4 * np.eye(n)
bup   = np.diag([1] * (n - 1), 1)
blow  = np.diag([1] * (n - 1), -1)
block = bdiag + bup + blow
# Creat a list of n blocks
blist = [block] * n
S = spsp.block_diag(blist)
# Upper diagonal array offset by -n
upper = np.diag(np.ones(n * (n - 1)), n)
# Lower diagonal array offset by -n
lower = np.diag(np.ones(n * (n - 1)), -n)
S += upper + lower

T = sp.linalg.solve(S,np.ravel(h**2*T))
plt.contourf(X,Y,T.reshape(n,n))
plt.colorbar();
```

```
CPU times: user 577 ms, sys: 112 ms, total: 689 ms
Wall time: 424 ms
```

Out[29]: <matplotlib.colorbar.Colorbar at 0x7f3884328460>


1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
−20
−30
−40
−50
−60
−70
−80
−90
−100

Chapter 15

Sympy

```
In [1]: %matplotlib inline
        %config InlineBackend.figure_format = "retina"
        import matplotlib.pyplot as plt
        import numpy as np
        import seaborn as sns
        sns.set()
        plt.rcParams['figure.figsize'] = (10.0, 6.0)
```

The function init_printing() will enable LaTeX pretty printing in the notebook for SymPy expressions.

```
In [2]: import sympy as sym
        from sympy import symbols, Symbol
        sym.init_printing()
```

```
In [3]: x= Symbol('x')
        (sym.pi + x)**2
```

Out[3]:

$$(x + \pi)^2$$

```
In [4]: alpha1, omega_2 = symbols('alpha1 omega_2')
        alpha1, omega_2
```

Out[4]:

$$(\alpha_1,\ \omega_2)$$

```
In [5]: mu, sigma = sym.symbols('mu sigma', positive = True)
        1/sym.sqrt(2*sym.pi*sigma**2)* sym.exp(-(x-mu)**2/(2*sigma**2))
```

Out[5]:

$$\frac{\sqrt{2}e^{-\frac{(-\mu+x)^2}{2\sigma^2}}}{2\sqrt{\pi}\sigma}$$

15.1 Why use sympy?

- Symbolic derivatives
- Translate mathematics into low level code
- Deal with very large expressions
- Optimize code using mathematics

Dividing two integers in Python creates a float, like 1/2 -> 0.5. If you want a rational number, use Rational(1, 2) or S(1)/2.

```
In [6]: x + sym.S(1)/2 , sym.Rational(1,4)
```

Out[6]:

$$\left(x + \frac{1}{2}, \frac{1}{4}\right)$$

```
In [7]: y = Symbol('y')
        x ^ y # XOR operator (True only if x != y)
```

Out[7]:

$$x \veebar y$$

```
In [8]: x**y
```

Out[8]:

$$x^y$$

SymPy expressions are immutable. Functions that operate on an expression return a new expression.

```
In [9]: expr = x + 1
        expr
```

Out[9]:

$$x + 1$$

```
In [10]: expr.subs(x, 2)
```

Out[10]:

$$3$$

```
In [11]: expr
```

Out[11]:

$$x + 1$$

15.1.1 Exercise: Lagrange polynomial

Given a set of $k+1$ data points :$(x_0, y_0), \ldots, (x_j, y_j), \ldots, (x_k, y_k)$ the Lagrange interpolation polynomial is:

$$L(x) := \sum_{j=0}^{k} y_j \ell_j(x)$$

ℓ_j are Lagrange basis polynomials:

$$\ell_j(x) := \prod_{\substack{0 \le m \le k \\ m \ne j}} \frac{x - x_m}{x_j - x_m}$$

We can demonstrate that at each point x_i, $L(x_i) = y_i$ so L interpolates the function.

- Compute the Lagrange polynomial for points

$$(-2, 21), (-1, 1), (0, -1), (1, -3), (2, 1)$$

15.2 Evaluate an expression

```
In [12]: sym.sqrt(2), sym.sqrt(2).evalf(7) # set the precision to 7 digits
```

Out[12]:

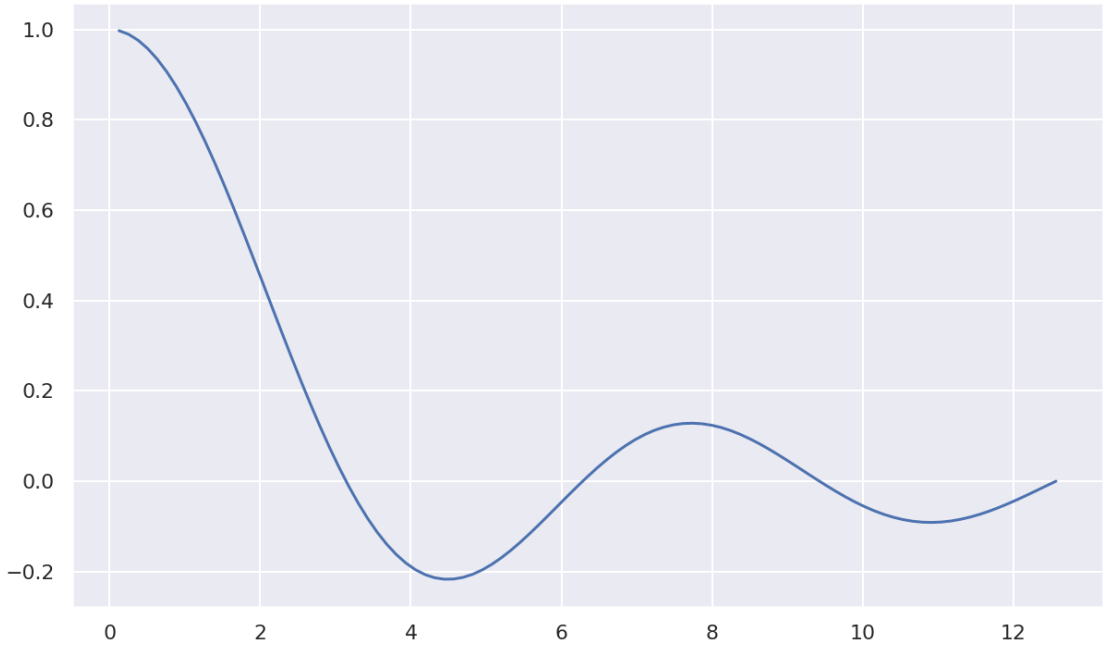
$$\left(\sqrt{2},\ 1.414214\right)$$

```
In [13]: from sympy import sin
         x = Symbol('x')
         expr = sin(x)/x
         expr.evalf(subs={x: 3.14})  # substitute the symbol x by Pi value
```

Out[13]:

0.00050721430461364

```
In [14]: from sympy.utilities.autowrap import ufuncify
         f = ufuncify([x], expr, backend='f2py')

         t = np.linspace(0,4*np.pi,100)
         plt.plot(t, f(t));
```



15.2.1 Exercise

- Plot the Lagrange polynomial computed above and interpolations points with matplotlib

15.3 Undefined functions and derivatives

Undefined functions are created with `Function()`. Undefined are useful to state that one variable depends on another (for the purposes of differentiation).

```
In [15]: from sympy import Function
         f = Function('f')
```

```
In [16]: f(x) + 1
```

Out[16]:

$$f(x) + 1$$

```
In [17]: from sympy import diff, sin, cos
         diff(sin(x + 1)*cos(y), x), diff(sin(x + 1)*cos(y), x, y), diff(f(x), x)
```

Out[17]:

$$\left(\cos(y)\cos(x+1),\ -\sin(y)\cos(x+1),\ \frac{d}{dx}f(x)\right)$$

```
In [18]: c, t = sym.symbols('t c')
         u = sym.Function('u')
         sym.Eq(diff(u(t,x),t,t), c**2*diff(u(t,x),x,2))
```

Out[18]:

$$\frac{\partial^2}{\partial c^2}u(c,x) = t^2\frac{\partial^2}{\partial x^2}u(c,x)$$

15.4 Matrices

```
In [19]: from sympy import Matrix
         Matrix([[1, 2], [3, 4]])*Matrix([x, y])
```

Out[19]:

$$\begin{bmatrix} x + 2y \\ 3x + 4y \end{bmatrix}$$

```
In [20]: x, y, z = sym.symbols('x y z')
         Matrix([sin(x) + y, cos(y) + x, z]).jacobian([x, y, z])
```

Out[20]:

$$\begin{bmatrix} \cos(x) & 1 & 0 \\ 1 & -\sin(y) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

15.5 Matrix symbols

SymPy can also operate on matrices of symbolic dimension $(n \times m)$. `MatrixSymbol("M", n, m)` creates a matrix M of shape $n \times m$.

```
In [21]: from sympy import MatrixSymbol, Transpose

         n, m = sym.symbols('n m', integer=True)
         M = MatrixSymbol("M", n, m)
         b = MatrixSymbol("b", m, 1)
         Transpose(M*b)
```

Out[21]:

$$(Mb)^T$$

```
In [22]: Transpose(M*b).doit()
```

Out[22]:

$$b^T M^T$$

15.6 Solving systems of equations

`solve` solves equations symbolically (not numerically). The return value is a list of solutions. It automatically assumes that it is equal to 0.

```
In [23]: from sympy import Eq, solve
         solve(Eq(x**2, 4), x)
```

Out[23]:

$$[-2,\ 2]$$

```
In [24]: solve(x**2 + 3*x - 3, x)
```

Out[24]:

$$\left[-\frac{3}{2} + \frac{\sqrt{21}}{2},\ -\frac{\sqrt{21}}{2} - \frac{3}{2}\right]$$

```
In [25]: eq1 = x**2 + y**2 - 4  # circle of radius 2
         eq2 = 2*x + y - 1  # straight line: y(x) = -2*x + 1
         solve([eq1, eq2], [x, y])
```

Out[25]:

$$\left[\left(\frac{2}{5} - \frac{\sqrt{19}}{5},\ \frac{1}{5} + \frac{2\sqrt{19}}{5}\right),\ \left(\frac{2}{5} + \frac{\sqrt{19}}{5},\ \frac{1}{5} - \frac{2\sqrt{19}}{5}\right)\right]$$

15.7 Solving differential equations

`dsolve` can (sometimes) produce an exact symbolic solution. Like `solve`, `dsolve` assumes that expressions are equal to 0.

```
In [26]: from sympy import Function, dsolve
         f = Function('f')
         dsolve(f(x).diff(x, 2) + f(x))
```

Out[26]:

$$f(x) = C_1 \sin(x) + C_2 \cos(x)$$

15.8 Code printers

The most basic form of code generation are the code printers. They convert SymPy expressions into over a dozen target languages.

```
In [27]: x = symbols('x')
         expr = abs(sin(x**2))
         expr
```

Out[27]:

$$\left|\sin\left(x^2\right)\right|$$

```
In [28]: sym.ccode(expr)
```

Out[28]: `'fabs(sin(pow(x, 2)))'`

```
In [29]: sym.fcode(expr, standard=2003, source_format='free')
```

Out[29]: `'abs(sin(x**2))'`

```
In [30]: from sympy.printing.cxxcode import cxxcode
         cxxcode(expr)
```

```
Out[30]: 'std::fabs(std::sin(std::pow(x, 2)))'
```

```
In [31]: sym.tanh(x).rewrite(sym.exp)
```

Out[31]:

$$\frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

```
In [32]: from sympy import sqrt, exp, pi
         expr = 1/sqrt(2*pi*sigma**2)* exp(-(x-mu)**2/(2*sigma**2))
         print(sym.fcode(expr, standard=2003, source_format='free'))
```

```
parameter (pi = 3.1415926535897932d0)
(1.0d0/2.0d0)*sqrt(2.0d0)*exp(-0.5d0*(-mu + x)**2/sigma**2)/(sqrt(pi)* &
      sigma)
```

15.9 Creating a function from a symbolic expression

In SymPy there is a function to create a Python function which evaluates (usually numerically) an expression. SymPy allows the user to define the signature of this function (which is convenient when working with e.g. a numerical solver in `scipy`).

```
In [33]: from sympy import log
         x, y = symbols('x y')
         expr = 3*x**2 + log(x**2 + y**2 + 1)
         expr
```

Out[33]:

$$3x^{2} + \log\left(x^{2} + y^{2} + 1\right)$$

```
In [34]: %timeit expr.subs({x: 17, y: 42}).evalf()
```

```
170 µs ± 1.32 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

```
In [35]: import math
         f = lambda x, y: 3*x**2 + math.log(x**2 + y**2 + 1)
         f(17, 42)
```

```
Out[35]: 874.6275443904885
```

```
In [36]: %timeit f(17, 42)
```

```
1.13 µs ± 16.3 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

Evaluate above expression numerically invoking the subs method followed by the evalf method can be quite slow and cannot be done repeatedly.

```
In [37]: from sympy import lambdify
         g = lambdify([x, y], expr, modules=['math'])
         g(17, 42)
```

Out[37]:
874.6275443904885

```
In [38]: %timeit g(17, 42)
```

```
1.12 µs ± 7 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

```
In [39]: xarr = np.linspace(17, 18, 5)
         h = lambdify([x, y], expr)  # lambdify return a python function
         out = h(xarr, 42)
         out.shape
```

Out[39]:
(5)

```
In [40]: z = z1, z2, z3 = symbols('z:3')
         expr2 = x*y*(z1 + z2 + z3)
         func2 = lambdify([x, y, z], expr2)
         func2(1, 2, (3, 4, 5))
```

Out[40]:
24

Behind the scenes lambdify constructs a string representation of the Python code and uses Python's eval function to compile the function.

15.10 SIR model

$$\frac{dS(t)}{dt} = -\beta S(t)I(t) \tag{15.1}$$

$$\frac{dI(t)}{dt} = \beta S(t)I(t) - \gamma I(t) \tag{15.2}$$

$$\frac{dR(t)}{dt} = \gamma I(t) \tag{15.3}$$

- S,I,R: ratio of suceptibles, infectious and recovered fraction of the population.
- t: time
- β : transmission coefficient.
- γ : healing rate.

We assume that total population is constant.

15.10.1 Solving the initial value problem numerically

We will now integrate this system of ordinary differential equations numerically using the `odeint` solver provided by `scipy`:

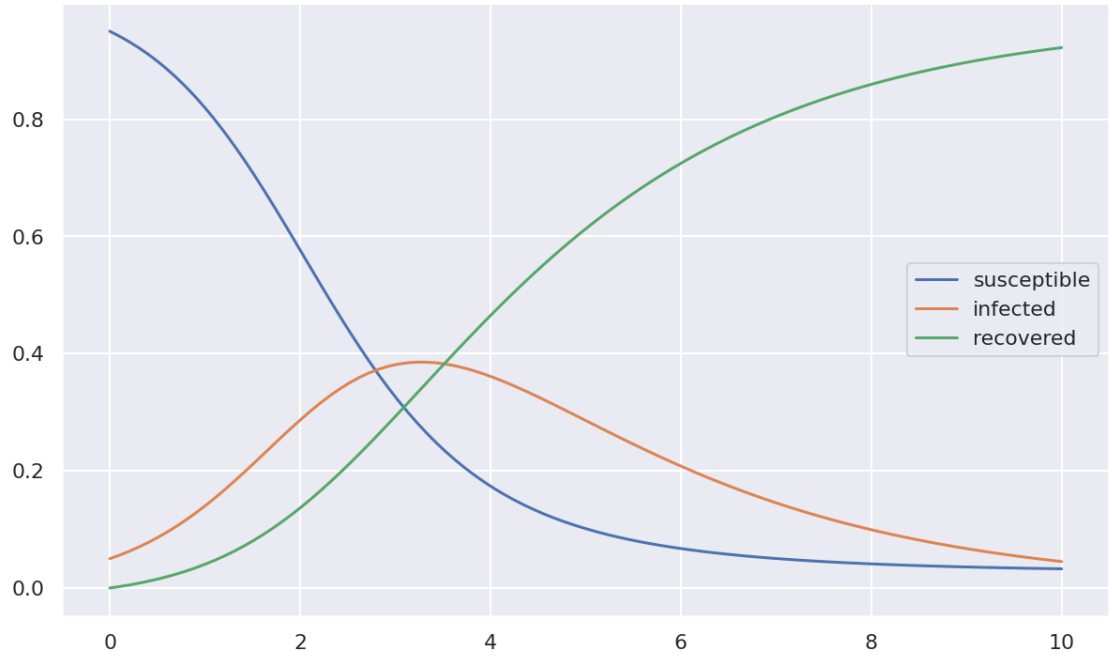
By looking at the documentation of odeint we see that we need to provide a function which computes a vector of derivatives ($\dot{\mathbf{y}} = [\frac{dy_1}{dt}, \frac{dy_2}{dt}, \frac{dy_3}{dt}]$). The expected signature of this function is:

```
f(y: array[float64], t: float64, *args: arbitrary constants) -> dydt: array[float64]
```

in our case we can write it as:

```
In [41]: def rhs(y, t, beta, gamma):
             rb = beta * y[0]*y[1]
             rg = gamma * y[1]
             return [- rb , rb - rg, rg]
```

```
In [42]: import scipy.integrate as spi
         tout = np.linspace(0, 10, 100)
         k_vals = 1.66, 0.4545455
         y0 = [0.95, 0.05, 0]
         yout = spi.odeint(rhs, y0, tout, k_vals)
         plt.plot(tout, yout)
         plt.legend(['susceptible', 'infected', 'recovered']);
```



We will construct the system from a symbolic representation. But at the same time, we need the `rhs` function to be fast. Which means that we want to produce a fast function from our symbolic representation. Generating a function from our symbolic representation is achieved through *code generation*.

1. Construct a symbolic representation from some domain specific representation using SymPy.
2. Have SymPy generate a function with an appropriate signature (or multiple thereof), which we pass on to the solver.

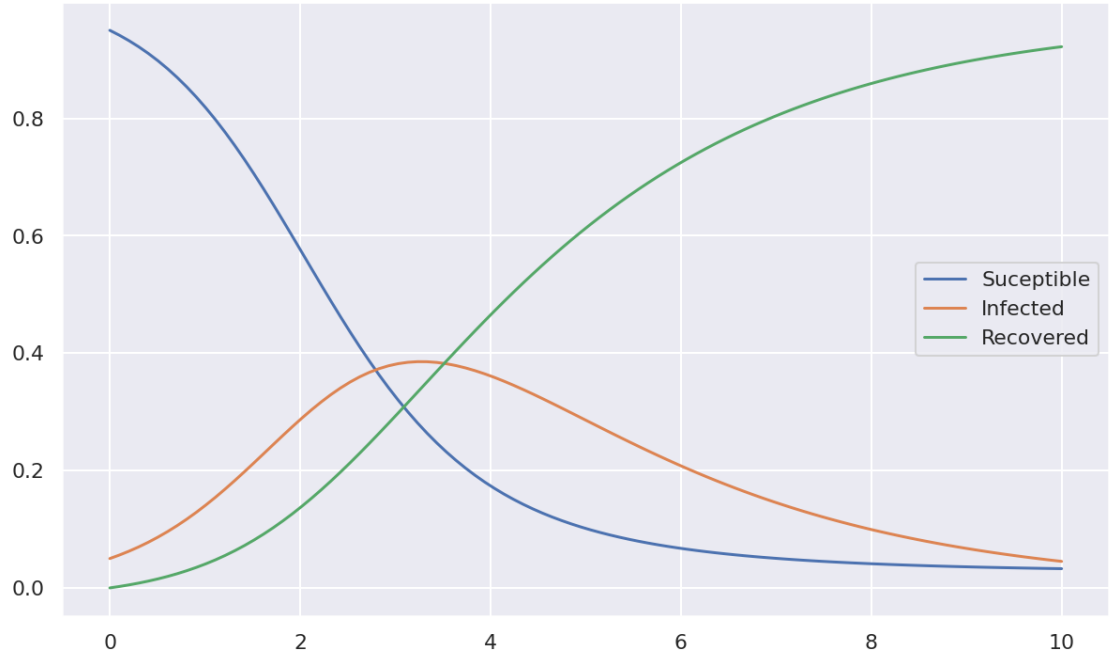
We will achieve (1) by using SymPy symbols (and functions if needed). For (2) we will use a function in SymPy called `lambdify` it takes a symbolic expressions and returns a function. In a later notebook, we will look at (1), for now we will just use `rhs` which we've already written:

```
In [43]: y, k = sym.symbols('y:3'), sym.symbols('beta gamma')
         ydot = rhs(y, None, *k)
         y, ydot
```

Out[43]:

$$((y_0,\ y_1,\ y_2),\ [-\beta y_0 y_1,\ \beta y_0 y_1 - \gamma y_1,\ \gamma y_1])$$

```
In [44]: f = sym.lambdify((y,t)+k, ydot)
         plt.plot(tout, spi.odeint(f, y0, tout, k_vals))
         plt.legend(['Suceptible', 'Infected', 'Recovered']);
```



In this example the gains of using a symbolic representation are arguably limited.
Let's take the same example with demography and n classes of subjects:

$$X_i = S_i, I_i, R_i \qquad i = 1 \dots n$$

$$\frac{dS_i}{dt} = \nu_i - \beta_i S_i I_i - \mu_i S_i + \sum_{j=1}^{n} m_{ji} S_j - \sum_{j=1}^{n} m_{ij} S_i \quad \frac{dI_i}{dt} = \beta_i S_i I_i - (\gamma_i + \mu_i) I_i + \sum_{j=1}^{n} m_{ij} I_j - \sum_{j=1}^{n} m_{ji} I_i \quad \frac{dR_i}{dt} = -\frac{dS_i}{dt} - \frac{dI_i}{dt}$$

- β : transmission coefficient
- γ : healing rate
- μ : mortality rate
- ν : birth rate

15.10.2 Exercise

- Create the symbolic matrix m, symbols $\nu_i, \mu_i, \beta_i, \gamma_i$ for $i = 0, 1, 2$ and y_j for $j = 0, 1, 2, \dots, 8$
- Write the system $\dot{y} = f(t, y, m, \nu, \mu, \beta, \gamma)$
- `lambdify` the f function.
- Solve the system with:

$$m = \begin{bmatrix} 0 & 0.01 & 0.01 \\ 0.01 & 0 & 0.01 \\ 0.01 & 0.01 & 0 \end{bmatrix}$$

$$
\begin{aligned}
t &= [0, 10] \text{ with } dt = 0.1 \\
\nu_i &= 0.0 \\
\mu_i &= 0.0 \\
\beta_i &= 1.66 \\
\gamma_i &= [0.4545, 0.3545, 0.2545] \\
S_i &= 0.95 \\
I_i &= 0.05 \\
R_i &= 0.0
\end{aligned}
$$

15.10.3 Exercise : Bezier curve

We want to compute and the draw the Bezier curve between the 3 points p_0, p_1, and p_2, The middle point p_1 position is arbitrary.

$$p0 = (1, 0); \qquad p1 = (x, y); \qquad p2 = (0, 1)$$

The $n + 1$ Bernstein basis polynomials of degree n are defined as

$$b_{i,n}(x) = \binom{n}{i} x^i \, (1 - x)^{n-i}, \quad i = 0, \ldots, n.$$

where $\binom{n}{i}$ is the binomial coefficient.
The Bezier curve is defined by a linear combination of Bernstein basis polynomials:

$$B_n(x) = \sum_{i=0}^{n} \beta_i b_{i,n}(x)$$

- With`sympy.binomial`, write a function `bpoly(t,n,i)` that returns the Bernstein basis polynomial $b_{i,n}(t)$.
- Compute the Berstein polynomial representing the Bezier curve between p_0, p_1, p_2. $\beta_i = 1$.
- Plot the Bezier Curve for 3 positions of $p_1 = (0, 0), (0.5, 0.5), (1, 1)$

15.11 Integrals quadrature

```
In [45]: from sympy.integrals.quadrature import *
         x, w = gauss_legendre(3, 5)
         x, w
```

Out[45]:
([−0.7746, 0, 0.7746], [0.55556, 0.88889, 0.55556])

```
In [46]: x, w = gauss_lobatto(3,12)
         x, w
```

Out[46]:
([−1, 0, 1], [0.333333333333, 1.33333333333, 0.333333333333])

15.12 References

- SciPy 2017 tutorial

Chapter 16

Call fortran from Python

```
In [1]: %matplotlib inline
        %config InlineBackend.figure_format = 'retina'
        import matplotlib.pyplot as plt
        import scipy.fftpack as sf
        import scipy.linalg as sl
        import numpy as np
        import warnings
        warnings.filterwarnings('ignore')
```

16.1 f2py

f2py is a part of Numpy and there are three ways to wrap Fortran with Python : - Write some fortran subroutines and just run f2py to create Python modules. - Insert special f2py directives inside Fortran source for complex wrapping. - Write a interface file (.pyf) to wrap Fortran files without changing them. f2py automatically generate the pyf template file that can be modified.

16.2 Simple Fortran subroutine to compute norm

16.2.1 Fortran 90/95 free format

```
In [2]: %%file euclidian_norm.f90
        subroutine euclidian_norm (a, b, c)
          real(8), intent(in) :: a, b
          real(8), intent(out) :: c
          c =        sqrt (a*a+b*b)
        end subroutine euclidian_norm
```

```
Writing euclidian_norm.f90
```

16.2.2 Fortran 77 fixed format

```
In [3]: %%file euclidian_norm.f
              subroutine euclidian_norm (a, b, c)
              real*8 a,b,c
        Cf2py intent(out) c
              c = sqrt (a*a+b*b)
              end
```

```
Writing euclidian_norm.f
```

16.3 Build extension module with f2py program

```
In [4]: import sys
        !{sys.executable} -m numpy.f2py --quiet -c euclidian_norm.f90 -m vect  --fcompiler=gnu95 --f90flags=-O3
```

16.3.1 Use the extension module in Python

```
In [5]: import vect
        c = vect.euclidian_norm(3,4)
        c
```

```
Out[5]: 5.0
```

```
In [6]: print(vect.euclidian_norm.__doc__) # Docstring is automatically generate
```

```
c = euclidian_norm(a,b)

Wrapper for ``euclidian_norm``.

Parameters
----------
a : input float
b : input float

Returns
-------
c : float
```

16.4 Fortran magic

- Jupyter extension that help to use fortran code in an interactive session.
- It adds a %%fortran cell magic that compile and import the Fortran code in the cell, using F2py.
- The contents of the cell are written to a .f90 file in the directory IPYTHONDIR/fortran using a filename with the hash of the code. This file is then compiled. The resulting module is imported and all of its symbols are injected into the user's namespace.

Documentation

```
In [7]: %load_ext fortranmagic
```

16.5 F2py directives

- F2PY introduces also some extensions to Fortran 90/95 language specification that help designing Fortran to Python interface, make it more "Pythonic".
- If editing Fortran codes is acceptable, these specific attributes can be inserted directly to Fortran source codes. Special comment lines are ignored by Fortran compilers but F2PY interprets them as normal lines.

```
In [8]: %%fortran
        subroutine euclidian_norm(a,c,n)
          integer :: n
          real(8),dimension(n),intent(in) :: a
          !f2py optional , depend(a) :: n=len(a)
          real(8),intent(out) :: c
```

```
          real(8) :: sommec
          integer :: i
          sommec = 0
          do i=1,n
            sommec=sommec+a( i )*a( i )
          end do
          c = sqrt (sommec)
        end subroutine euclidian_norm
```

```
In [9]: a=[2,3,4]  # Python list
        type(a)

Out[9]: list

In [10]: euclidian_norm(a)

Out[10]: 5.385164807134504

In [11]: a=np.arange(2,5)  # numpy array
         type(a)

Out[11]: numpy.ndarray

In [12]: euclidian_norm(a)

Out[12]: 5.385164807134504

In [13]: print(euclidian_norm.__doc__) # Documentation

c = euclidian_norm(a,[n])

Wrapper for ``euclidian_norm``.

Parameters
----------
a : input rank-1 array('d') with bounds (n)

Other Parameters
----------------
n : input int, optional
    Default: len(a)

Returns
-------
c : float
```

16.6 F2py directives

- `optional`: The corresponding argument is moved to the end.
- `required`: This is default. Use it to disable automatic optional setting.
- `intent(in | inout | out | hide)` , `intent(in)` is the default.
- `intent(out)` is implicitly translated to `intent(out,hide)`.
- `intent(copy)` and `intent(overwrite)` control changes for input arguments.
- `check` performs some assertions, it is often automatically generated.
- `depend`: f2py detects cyclic dependencies.

- `allocatable, parameter`
- `intent(callback), external`: for function as arguments.
- `intent(c)` C-type argument , array or function.
- C expressions: `rank, shape, len, size, slen`.

16.7 Callback

You can call a python function inside your fortran code

```
In [14]: %%fortran
         subroutine sum_f (f ,n, s)
           !Compute sum(f(i), i=1,n)
           external f
           integer, intent(in) :: n
           real, intent(out) :: s
           s = 0.0
           do i=1,n
             s=s+f(i)
           end do
         end subroutine sum_f

In [15]: def fonction(i) : # python function
             return i*i

         sum_f(fonction,3)

Out[15]: 14.0

In [16]: sum_f(lambda x :x**2,3) # lambda function

Out[16]: 14.0
```

16.8 Fortran arrays and Numpy arrays

Let's see how to pass numpy arrays to fortran subroutine.

```
In [17]: %%fortran --extra "-DF2PY_REPORT_ON_ARRAY_COPY=1"
         subroutine push( positions, velocities, dt, n)
           integer, intent(in) :: n
           real(8), intent(in) :: dt
           real(8), dimension(n,3), intent(in) :: velocities
           real(8), dimension(n,3) :: positions
           do i = 1, n
             positions(i,:) = positions(i,:) + dt*velocities(i,:)
           end do
         end subroutine push

In [18]: positions = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
         velocities = [[0, 1, 2], [0, 3, 2], [0, 1, 3]]

In [19]: import sys
         push(positions, velocities, 0.1)
         positions # memory is not updated because we used C memory storage

Out[19]: [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

During execution, the message "created an array from object" is displayed, because a copy of is made when passing multidimensional array to fortran subroutine.

```
In [20]: positions = np.array(positions, dtype='f8', order='F')
         push(positions, velocities, 0.1)
         positions # the memory is updated

Out[20]: array([[0. , 0.1, 0.2],
                [0. , 0.3, 0.2],
                [0. , 0.1, 0.3]])
```

16.9 Signature file

This file contains descriptions of wrappers to Fortran or C functions, also called as signatures of the functions. F2PY can create initial signature file by scanning Fortran source codes and catching all relevant information needed to create wrapper functions.

```
f2py vector.f90 -h vector.pyf
```

- vector.pyf

```
!    -*- f90 -*-
! Note: the context of this file is case sensitive.

subroutine euclidian_norm(a,c,n) ! in vector.f90
    real(kind=8) dimension(n),intent(in) :: a
    real(kind=8) intent(out) :: c
    integer optional,check(len(a)>=n),depend(a) :: n=len(a)
end subroutine euclidian_norm

! This file was auto-generated with f2py (version:2).
! See http://cens.ioc.ee/projects/f2py2e/
```

16.10 Wrap lapack function dgemm with f2py

- Generate the signature file

```
In [21]: %rm -f dgemm.f dgemm.pyf
         !wget -q http://ftp.mcs.anl.gov/pub/MINPACK-2/blas/dgemm.f

In [22]: # %load dgemm.f

In [23]: !{sys.executable} -m numpy.f2py -m mylapack --overwrite-signature -h dgemm.pyf dgemm.f

Reading fortran codes…
        Reading file 'dgemm.f' (format:fix,strict)
rmbadname1: Replacing "max" with "max_bn".
Post-processing…
        Block: mylapack
                        Block: dgemm
Post-processing (stage 2)…
Saving signatures to file "./dgemm.pyf"


!    -*- f90 -*-
! Note: the context of this file is case sensitive.

python module mylapack ! in
```

```
    interface  ! in :mylapack
        subroutine dgemm(transa,transb,m,n,k,alpha,a,lda,b,ldb,beta,c,ldc) ! in :mylapack:dgemm.f
            character*1 :: transa
            character*1 :: transb
            integer :: m
            integer :: n
            integer :: k
            double precision :: alpha
            double precision dimension(lda,*) :: a
            integer, optional,check(shape(a,0)==lda),depend(a) :: lda=shape(a,0)
            double precision dimension(ldb,*) :: b
            integer, optional,check(shape(b,0)==ldb),depend(b) :: ldb=shape(b,0)
            double precision :: beta
            double precision dimension(ldc,*) :: c
            integer, optional,check(shape(c,0)==ldc),depend(c) :: ldc=shape(c,0)
        end subroutine dgemm
    end interface
end python module mylapack

! This file was auto-generated with f2py (version:2).
! See http://cens.ioc.ee/projects/f2py2e/
```

```
In [24]: !{sys.executable} -m numpy.f2py --quiet -c dgemm.pyf -llapack
```

```
In [25]: import numpy as np
         import mylapack
         a = np.array([[7,8],[3,4],[1,2]])
         b = np.array([[1,2,3],[4,5,6]])
         print("a=",a)
         print("b=",b)
         assert a.shape[1] == b.shape[0]
         c = np.zeros((a.shape[0],b.shape[1]),'d',order='F')
         mylapack.dgemm('N','N',a.shape[0],b.shape[1],a.shape[1],1.0,a,b,1.0,c)
         print(c)
         np.all(c == a @ b) # check with numpy matrix multiplication
```

```
a= [[7 8]
 [3 4]
 [1 2]]
b= [[1 2 3]
 [4 5 6]]
[[39. 54. 69.]
 [19. 26. 33.]
 [ 9. 12. 15.]]
```

```
Out[25]: True
```

16.10.1 Exercise

- Modify the file dgemm.pyf to set all arguments top optional and keep only the two matrices as input.

```
In [26]: # %load solutions/fortran/dgemm2.pyf
```

- Build the python module

```
In [27]: !{sys.executable} -m numpy.f2py --quiet -c solutions/fortran/dgemm2.pyf -llapack --f90flags=-O3
```

```
In [28]: import mylapack2
         a = np.array([[7,8],[3,4],[1,2]])
         b = np.array([[1,2,3],[4,5,6]])
         c = mylapack2.dgemm(a,b)
         np.all( c == a @ b)
```

```
Out[28]: True
```

16.11 Check performance between numpy and mylapack

```
In [29]: a = np.random.random((512,128))
         b = np.random.random((128,512))
```

```
In [30]: %timeit c = mylapack2.dgemm(a,b)
```

```
1.64 ms ± 18.5 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

```
In [31]: %timeit c = a @ b
```

```
1.24 ms ± 29.2 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Fortran arrays allocated in a subroutine share same memory in Python

```
In [32]: %%fortran
         module f90module
            implicit none
            real(8), dimension(:),   allocatable :: farray
         contains
            subroutine init( n ) !Allocation du tableau farray
            integer, intent(in) :: n
            allocate(farray(n))
            end subroutine init
         end module f90module
```

```
In [33]: f90module.init(10)
         len(f90module.farray)
```

```
Out[33]: 10
```

Numpy arrays allocated in Python passed to Fortran are already allocated

```
In [34]: %%fortran
         module f90module
            implicit none
            real(8), dimension(:),   allocatable :: farray
         contains
            subroutine test_array( allocated_flag, array_size )
            logical, intent(out) :: allocated_flag
            integer, intent(out) :: array_size
            allocated_flag = allocated(farray)
            array_size = size(farray)
            end subroutine test_array
         end module f90module
```

```
In [35]: f90module.farray = np.random.rand(10).astype(np.float64)
         f90module.test_array()
```

```
Out[35]: (1, 10)
```

16.12 f2py + OpenMP

```
In [36]: %env OMP_NUM_THREADS=4

env: OMP_NUM_THREADS=4


In [37]: %%fortran
         subroutine hello( )
           integer :: i
           do i = 1, 4
             call sleep(1)
           end do
         end subroutine

In [38]: %%time
         hello()

CPU times: user 14.9 ms, sys: 8.43 ms, total: 23.3 ms
Wall time: 4 s


In [39]: %%fortran --f90flags "-fopenmp" --extra "-L/usr/local/lib -lgomp"
         subroutine hello_omp( )
           integer :: i
           !$OMP PARALLEL PRIVATE(I)
           !$OMP DO
           do i = 1, 4
             call sleep(1)
           end do
           !$OMP END DO
           !$OMP END PARALLEL

         end subroutine

In [40]: %%time
         hello_omp()

CPU times: user 6.44 ms, sys: 405 µs, total: 6.85 ms
Wall time: 1 s
```

16.13 Conclusions

16.13.1 pros

- Easy to use, it works with modern fortran, legacy fortran and also C.
- Works with common and modules and arrays dynamically allocated.
- Python function callback can be very useful combined with Sympy
- Documentation is automatically generated
- All fortran compilers are supported: GNU, Portland, Sun, Intel,...
- F2py is integrated in numpy library.

16.13.2 cons

- Derived types and fortran pointers are not well supported.
- Absolutely not compatible with fortran 2003-2008 new features (classes)
- f2py is maintained but not really improved. Development is stopped.

16.14 distutils

16.14.1 setup.py

```
from numpy.distutils.core import Extension, setup
ext1 = Extension(name = 'scalar',
                 sources = ['scalar.f'])
ext2 = Extension(name = 'fib2',
                 sources = ['fib2.pyf','fib1.f'])

setup(name = 'f2py_example', ext_modules = [ext1,ext2])
```

Compilation

```
python3 setup.py build_ext --inplace
```

16.15 Exercice: Laplace problem

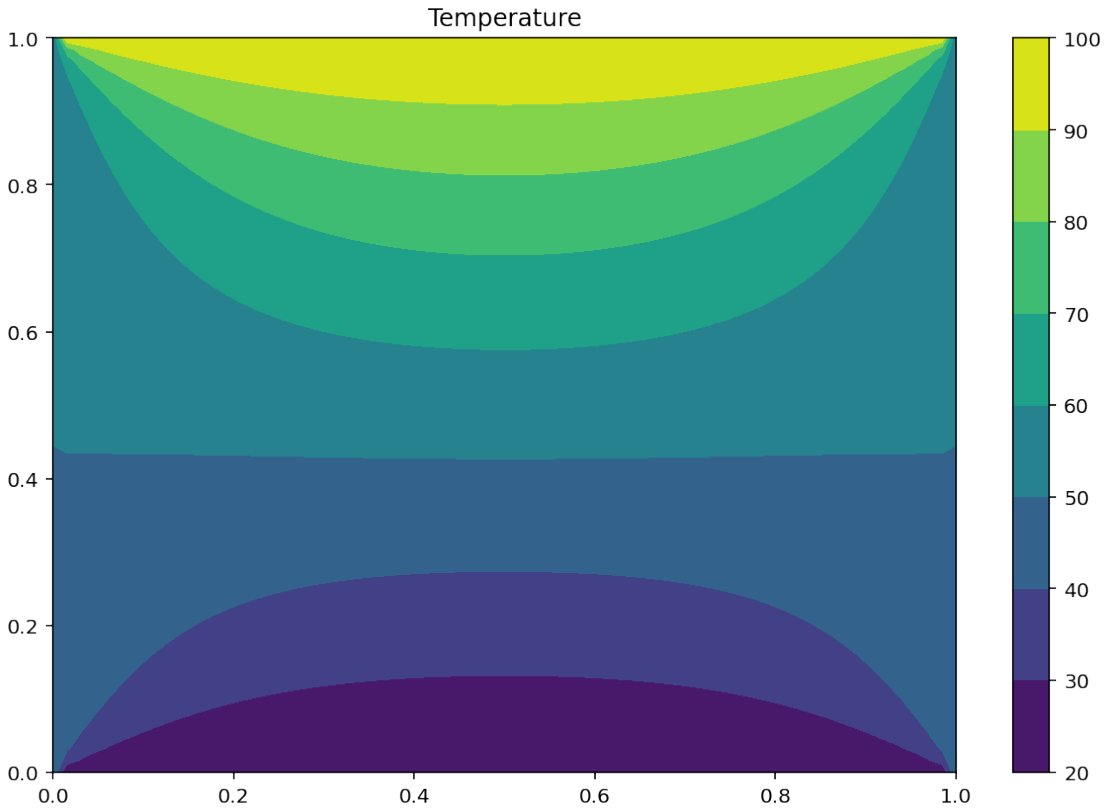
- Replace the `laplace` function by a fortran subroutine

```
In [41]: %%time
         %matplotlib inline
         %config InlineBackend.figure_format = 'retina'
         import numpy as np
         import matplotlib.pyplot as plt
         import itertools
         # Boundary conditions
         Tnorth, Tsouth, Twest, Teast = 100, 20, 50, 50

         # Set meshgrid
         n, l = 64, 1.0
         X, Y = np.meshgrid(np.linspace(0,l,n), np.linspace(0,l,n))
         T = np.zeros((n,n))

         # Set Boundary condition
         T[n-1:, :] = Tnorth
         T[:1, :] = Tsouth
         T[:, n-1:] = Teast
         T[:, :1] = Twest

         def laplace(T, n):
             residual = 0.0
             for i in range(1, n-1):
                 for j in range(1, n-1):
                     T_old = T[i,j]
                     T[i, j] = 0.25 * (T[i+1,j] + T[i-1,j] + T[i,j+1] + T[i,j-1])
                     if T[i,j]>0:
                         residual=max(residual,abs((T_old-T[i,j])/T[i,j]))
             return residual

         residual = 1.0
         istep = 0
         while residual > 1e-5 :
             istep += 1
             residual = laplace(T, n)
             print ((istep, residual), end="\r")

         print("iterations = ",istep)
```

```
        plt.rcParams['figure.figsize'] = (10,6.67)
        plt.title("Temperature")
        plt.contourf(X, Y, T)
        plt.colorbar()
```

```
iterations =  2457
CPU times: user 32 s, sys: 207 ms, total: 32.2 s
Wall time: 31.9 s
```

Out[41]: <matplotlib.colorbar.Colorbar at 0x7fdfdb7eb160>



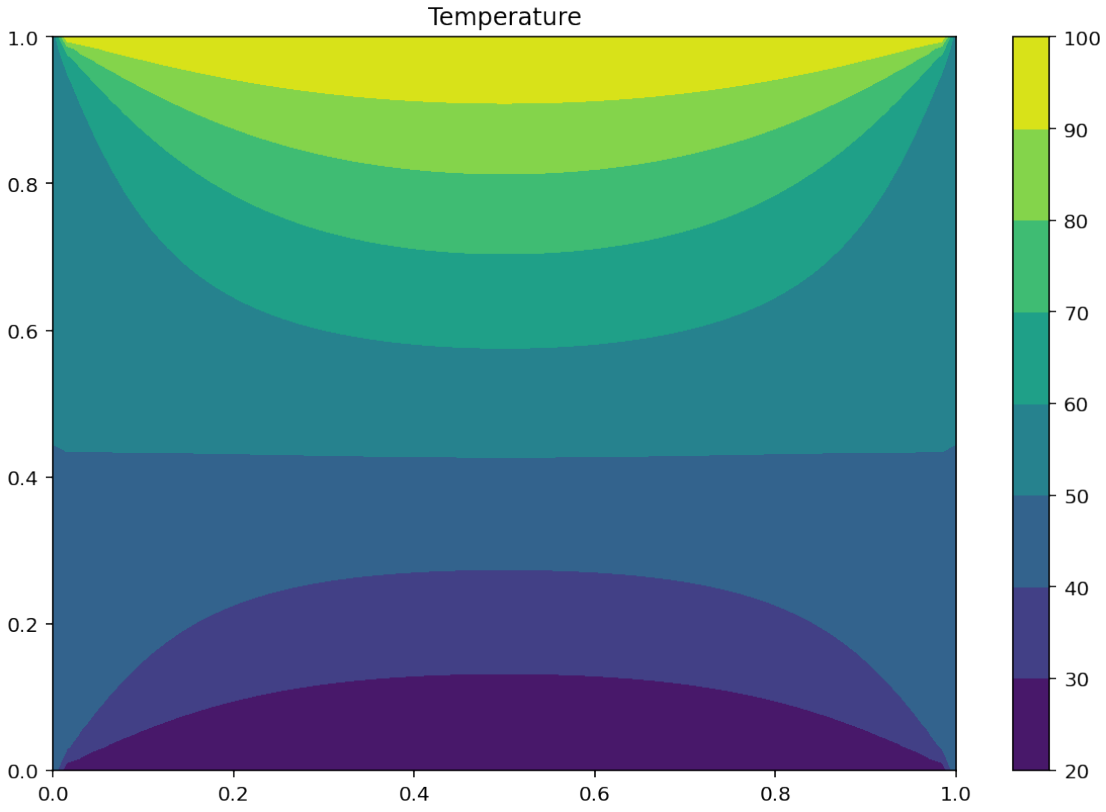
In [42]:
```
%%fortran
# %load solutions/fortran/laplace_fortran.F90
subroutine laplace_fortran( T, n, residual )

  real(8), intent(inout) :: T(0:n-1,0:n-1) ! Python indexing
  integer, intent(in)    :: n
  real(8), intent(out)   :: residual
  real(8) :: T_old

  residual = 0.0
  do i = 1, n-2
    do j = 1, n-2
        T_old = T(i,j)
        T(i, j) = 0.25 * (T(i+1,j) + T(i-1,j) + T(i,j+1) + T(i,j-1))
            if (T(i,j) > 0) then
```

```
                    residual=max(residual,abs((T_old-T(i,j))/T(i,j)))
                end if
            end do
          end do

        end subroutine laplace_fortran
```

In [43]:
```
%%time
%matplotlib inline
%config InlineBackend.figure_format = 'retina'
import numpy as np
import matplotlib.pyplot as plt
import itertools
# Boundary conditions
Tnorth, Tsouth, Twest, Teast = 100, 20, 50, 50

# Set meshgrid
n, l = 64, 1.0
X, Y = np.meshgrid(np.linspace(0,l,n), np.linspace(0,l,n))
T = np.zeros((n,n), order='F')  ## We need to declare a new order in memory

# Set Boundary condition
T[n-1:, :] = Tnorth
T[:1, :] = Tsouth
T[:, n-1:] = Teast
T[:, :1] = Twest

residual = 1.0
istep = 0
while residual > 1e-5 :
    istep += 1
    residual = laplace_fortran(T, n)
    print ((istep, residual), end="\r")

print()
print("iterations = ",istep)
plt.rcParams['figure.figsize'] = (10,6.67)
plt.title("Temperature")
plt.contourf(X, Y, T)
plt.colorbar()
```

```
(2457, 9.997295133247811e-06)
iterations =  2457
CPU times: user 502 ms, sys: 81.3 ms, total: 583 ms
Wall time: 415 ms
```

Out[43]: <matplotlib.colorbar.Colorbar at 0x7fdfdb51b910>



16.16 References

- Talk by E. Sonnendrücker
- SciPy
- Sagemath Documentation
- Hans Petter Langtangen. *Python Scripting for Computational Science.* Springer 2004

Chapter 17

Cython

```
In [1]: %matplotlib inline
        import matplotlib.pyplot as plt
        plt.rcParams['figure.figsize'] = (10,6)
        %config InlineBackend.figure_format = 'retina'
        import numpy as np
```

```
In [2]: import warnings
        warnings.filterwarnings("ignore")
```

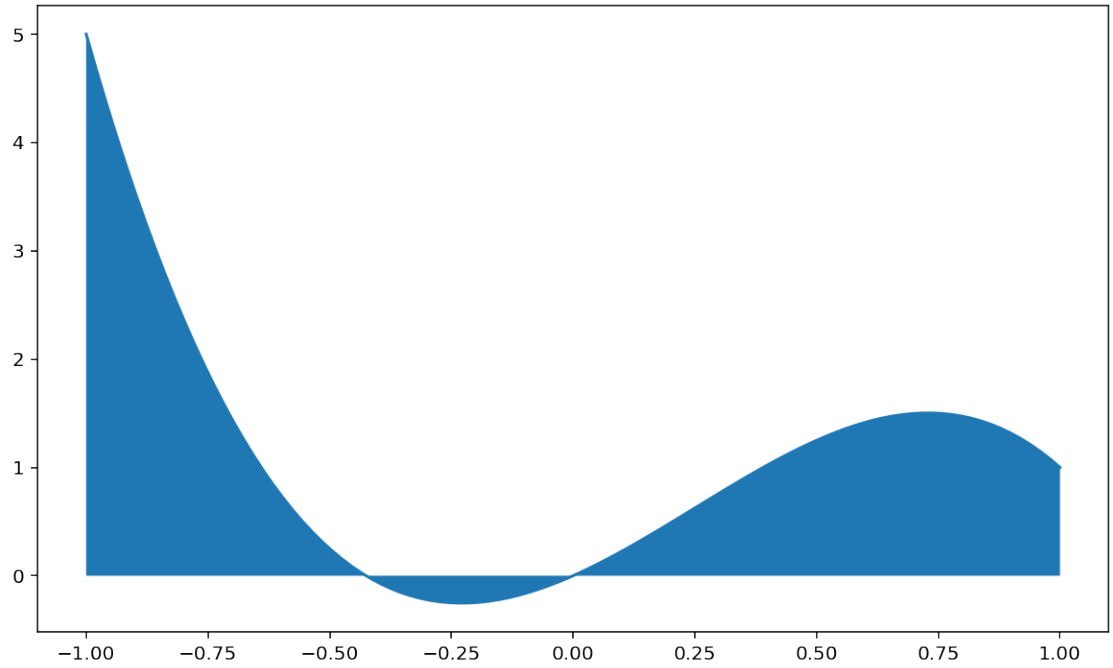
- Cython provides extra syntax allowing for static type declarations (remember: Python is generally dynamically typed)
- Python code gets translated into optimised C/C++ code and compiled as Python extension modules
- Cython allows you to write fast C code in a Python-like syntax.
- Furthermore, linking to existing C libraries is simplified.

- Pure Python Function

$f(x) = -2x^3 + 5x^2 + x,$

```
In [3]: def f(x):
            return -4*x**3 +3*x**2 +2*x

        x = np.linspace(-1,1,100)
```

```
ax = plt.subplot(1,1,1)
ax.plot(x, f(x))
ax.fill_between(x, 0, f(x));
```



we compute integral $\int_a^b f(x)dx$ numerically with N points.

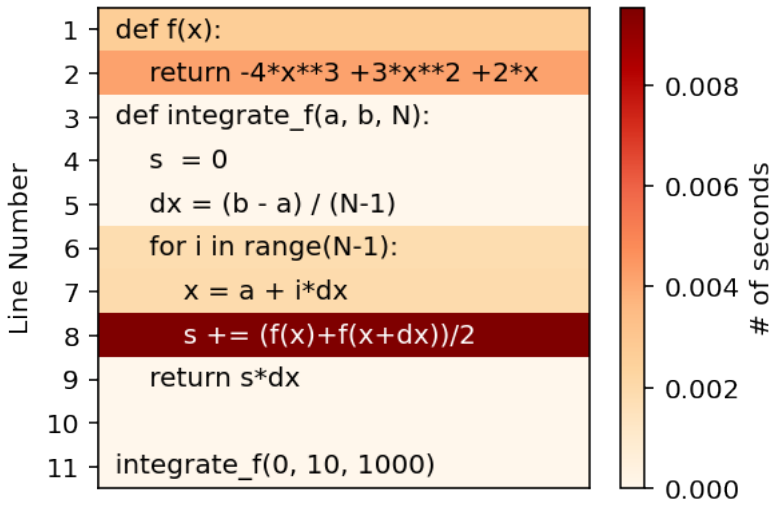
```
In [4]: def integrate_f_py(a,b,N):
            s  = 0
            dx = (b - a) / (N-1)
            for i in range(N-1): # we intentionally use the bad way to do this with a loop
                x = a + i*dx
                s += (f(x)+f(x+dx))/2
            return s*dx
```

```
In [5]: %timeit integrate_f_py(-1,1,10**3)
        print(integrate_f_py(-1,1,1000))
```

```
966 µs ± 8.58 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
2.0000040080120174
```

```
In [6]: %load_ext heat
```

```
In [7]: %%heat
        def f(x):
            return -4*x**3 +3*x**2 +2*x
        def integrate_f(a, b, N):
            s  = 0
            dx = (b - a) / (N-1)
            for i in range(N-1):
                x = a + i*dx
                s += (f(x)+f(x+dx))/2
            return s*dx

        integrate_f(0, 10, 1000)
```



- Pure C function

In [8]: %%file integral_f_c.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

#define NB_RUNS 1000

double f(double x) {
    return -4*x*x*x +3*x*x +2*x;
}

double integrate_f_c(double a, double b, int N) {
    double s  = 0;
    double dx = (b - a) / (N-1);
    for(int i=0; i<N-1; ++i){
        double x = a + i*dx;
        s += (f(x)+f(x+dx))/2.0;
    }
    return s*dx;
}

int main(int argc, char **argv)
{

  double a =  atof(argv[1]);
  double b =  atof(argv[2]);
  int N    =  atoi(argv[3]);
  double  res = 0;

  clock_t begin = clock();

  for (int i=0; i<NB_RUNS; ++i)
      res += integrate_f_c( a, b, N );
```

```
    clock_t end = clock();

    fprintf( stdout, "integral_f(%3.1f, %3.1f, %d) = %f \n", a, b, N, res / NB_RUNS );
    fprintf( stdout, "time = %e ms \n",  (double)(end - begin) / CLOCKS_PER_SEC );

    return 0;
}
```

```
Writing integral_f_c.c
```

```
In [9]: !gcc -O3 integral_f_c.c; ./a.out -1 1 1000
```

```
integral_f(-1.0, 1.0, 1000) = 2.000004
time = 5.494000e-03 ms
```

17.1 Cython compilation: Generating C code

Load Cython in jupyter notebook.

```
In [10]: %load_ext Cython
```

17.1.1 C Variable and Type definitions

In general, use `cdef` to declare C variables. The command :

```
$ cython -a mycode.pyx
```

outputs an html file. It shows what parts of your code are C, which parts are Python, and where C-Python conversion occurs.

```
In [11]: %%cython -a
         cdef int i, j = 2, k = 3      # assigning values at declaration
         i = 1                         # assigning values afterwards
         # avoid Python-C conversion! It's expensive:
         a = 5
         i = a
         # same with C-Python conversion:
         b = j
         print("a = %d" % a)
         print("i = %d" % i)
         print("b = %d" % b)
```

```
a = 5
i = 5
b = 2
```

```
Out[11]: <IPython.core.display.HTML object>
```

17.1.2 Another Python vs. Cython coloring guide

```
In [12]: %%cython -a
         cdef int m, n
         cdef double cy_total = 0.0
         for m in range(10):
             n = 2*m
```

```
            cy_total += n
        a, b = 0, 0
        py_total = 0.0
        for a in range(10):
            b = 2*a
            py_total += b
        print(cy_total, py_total)
```

```
90.0 90.0
```

Out[12]: <IPython.core.display.HTML object>

```
In [13]: %%cython -a
         cdef struct Grail:
             int age
             float volume
         cdef union Food:
             char *spam
             float *eggs
         cdef enum CheeseType:
             cheddar, edam,
             camembert
         cdef enum CheeseState:
             hard = 1
             soft = 2
             runny = 3
         cdef Grail holy
         holy.age    = 500
         holy.volume = 10.0
         print (holy.age, holy.volume)
```

```
500 10.0
```

Out[13]: <IPython.core.display.HTML object>

17.2 Cython Functions

Use **cdef** to define a Cython function.
- Cython function can accept either (inclusive) Python and C values as well as return either Python or C values, - *Within a Cython module* Python and Cython functions can call each other freely. However, only **Python** functions can be called from outside the module by Python code. (i.e. importing/exporting a Cython module into some Python code)

cpdef define a Cython function with a simple Python wrapper. However, when called from Cython the Cython / C code is called directly, bypassing the Python wrapper.

Writing pure code in Cython gives a small speed boost. Note that none of the code below is Cython-specific. Just add `.pyx` instead of `.py` extension.

```
In [14]: %%file cython_f_example.pyx
         def f(x):
             return -4*x**3 +3*x**2 +2*x
         def integrate_f(a, b, N):
             s  = 0
             dx = (b - a) / (N-1)
             for i in range(N-1):
                 x = a + i*dx
                 s += (f(x)+f(x+dx))/2
             return s*dx
```

```
Writing cython_f_example.pyx
```

17.3 Cython Compilation

- The .pyx source file is compiled by Cython to a .c file.
- The .c source file contains the code of a Python extension module.
- The .c file is compiled by a C compiler to a .so (shared object library) file which can be imported directly into a Python session.

17.3.1 Build with CMake

```
project(cython_f_example CXX)
include(UseCython)  # Load Cython functions
# Set C++ output
set_source_file_properties(cython_f_example.pyx PROPERTIES CYTHON_IS_CXX TRUE )
# Build the extension module
cython_add_module( modname cython_f_example.pyx cython_f_example.cpp )
```

17.3.2 C/C++ generation with cython application

```
cython -3 cython_f_example.pyx   # create the C file for Python 3
cython -3 --cplus cython_f_example.pyx  # create the C++ file for Python 3
```

17.3.3 build with a C/C++ compiler

To build use the Makefile:

```
CC=gcc
CFLAGS=`python-config --cflags`
LDFLAGS=`python-config --ldflags`
cython_f_example:
     ${CC} -c $@.c ${CFLAGS}
     ${CC} $@.o -o $@.so -shared ${LDFLAGS}
```

Import the module in Python session

```
import cython_f_example
```

17.4 pyximport

import Cython .pyx files as if they were .py files:

```
In [15]: import pyximport
         pyximport.install()
         import cython_f_example
         %timeit cython_f_example.integrate_f(-1,1,10**3)
         print(cython_f_example.integrate_f(-1,1,1000))
```

```
760 µs ± 4.58 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
2.0000040080120174
```

17.5 Building a Cython module using distutils

Create the setup.py script:

```
In [16]: %%file setup.py
         from distutils.core import setup
         from Cython.Build import cythonize

         setup(
           name = 'Cython Example Integrate f Function',
           ext_modules = cythonize("cython_f_example.pyx"),
         )
```

```
Writing setup.py
```

```
In [17]: %run setup.py  build_ext --inplace --quiet
```

```
Compiling cython_f_example.pyx because it changed.
[1/1] Cythonizing cython_f_example.pyx
running build_ext
building 'cython_f_example' extension
C compiler: gcc -pthread -B /usr/share/miniconda3/envs/runenv/compiler_compat -Wl,--sysroot=/ -Wsign-co

creating build
creating build/temp.linux-x86_64-3.8
compile options: '-I/usr/share/miniconda3/envs/runenv/include/python3.8 -c'
gcc: cython_f_example.c
gcc -pthread -shared -B /usr/share/miniconda3/envs/runenv/compiler_compat -L/usr/share/miniconda3/envs/
```

```
<Figure size 720x432 with 0 Axes>
```

```
In [18]: from cython_f_example import integrate_f
         %timeit integrate_f(-1,1,10**3)
         integrate_f(-1,1,10**3)
```

```
772 µs ± 12.3 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

```
Out[18]: 2.0000040080120174
```

17.6 Why is it faster with Cython ?

- Python code is interpreted at every execution to machine code.
- Compiled C code is already in machine code.
- C is a statically-typed language. It gives to the compiler more information which allows it to optimize both computations and memory access.
- To add two variables, Python checks the type before calling the right **add** function and store it to a value that can be new.
- C just add the variables and return the result.

17.7 Add Cython types

We coerce Python types to C types when calling the function. Still a ”Python function” so callable from the global namespace.

```
In [19]: %%cython
         def f(x):
             return -4*x**3 +3*x**2 +2*x
         def cy_integrate_f(double a, double b, int N):
             cdef int i
             cdef double s, x, dx
             s  = 0
             dx = (b - a) / (N-1)
             for i in range(N-1):
                 x = a + i*dx
                 s += (f(x)+f(x+dx))/2
             return s*dx
```

```
building '_cython_magic_8c0baa0b730cb71c660b134eef8e4b03' extension
C compiler: gcc -pthread -B /usr/share/miniconda3/envs/runenv/compiler_compat -Wl,--sysroot=/ -Wsign-co

compile options: '-I/usr/share/miniconda3/envs/runenv/include/python3.8 -c'
gcc: /home/runner/.cache/ipython/cython/_cython_magic_8c0baa0b730cb71c660b134eef8e4b03.c
gcc -pthread -shared -B /usr/share/miniconda3/envs/runenv/compiler_compat -L/usr/share/miniconda3/envs/
```

- typing the iterator variable i with C semantics, tells Cython to compile the for-loop to pure C code.
- typing a, s and dx is important as they are involved in arithmetic within the for-loop
- Cython type declarations can make the source code less readable
- Do not use them without good reason, i.e. only in performance critical sections.

```
In [20]: %timeit cy_integrate_f(-1,1,10**3)
         print(cy_integrate_f(-1,1,1000))
```

```
685 µs ± 9.54 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
2.0000040080120174
```

Finally, we integrate a Cython function instead of a Python function. This eliminates the Python-C conversion at the function call as seen above thus giving a pure Cython/C algorithm.

The primary downside is not being allowed to call the function `cy_f`, from Python unless `cpdef` is used.

```
In [21]: %%cython
         cdef double cy_f(double x):
             return -4*x**3 +3*x**2 +2*x
         def cycy_integrate_f(double a, double b, int N):
             cdef int i
             cdef double s, x, dx
             s  = 0
             dx = (b - a) / (N-1)
             for i in range(N-1):
                 x = a + i*dx
                 s += (cy_f(x)+cy_f(x+dx))/2
             return s*dx
```

```
building '_cython_magic_c253bb1961eda445c34ddc2d7b4ab8bc' extension
C compiler: gcc -pthread -B /usr/share/miniconda3/envs/runenv/compiler_compat -Wl,--sysroot=/ -Wsign-co

compile options: '-I/usr/share/miniconda3/envs/runenv/include/python3.8 -c'
gcc: /home/runner/.cache/ipython/cython/_cython_magic_c253bb1961eda445c34ddc2d7b4ab8bc.c
gcc -pthread -shared -B /usr/share/miniconda3/envs/runenv/compiler_compat -L/usr/share/miniconda3/envs/
```

```
In [22]: %timeit cycy_integrate_f(-1,1,10**3)
         print(cycy_integrate_f(-1,1,1000))
```

```
164 µs ± 614 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
2.0000040080120174
```

17.8 Exercise : Cythonize the trivial exponential function.

```
In [23]: %%cython -a
         def exp_python(x,terms=50):
             sum = 0.
             power = 1.
             fact = 1.
             for i in range(terms):
                 sum += power/fact
                 power *= x
                 fact *= i+1
             return sum
```

```
building '_cython_magic_a96a095f6c3175b142027432bac51f88' extension
C compiler: gcc -pthread -B /usr/share/miniconda3/envs/runenv/compiler_compat -Wl,--sysroot=/ -Wsign-co

compile options: '-I/usr/share/miniconda3/envs/runenv/include/python3.8 -c'
gcc: /home/runner/.cache/ipython/cython/_cython_magic_a96a095f6c3175b142027432bac51f88.c
gcc -pthread -shared -B /usr/share/miniconda3/envs/runenv/compiler_compat -L/usr/share/miniconda3/envs/
```

```
Out[23]: <IPython.core.display.HTML object>
```

```
In [24]: %timeit exp_python(1.,50)
```

```
4.02 µs ± 32.3 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

```
In [25]: %%cython

         # %load solutions/cython/exponential.pyx
         #cython: profile=False
         #cython: cdivision=True
         def exp_cython(double x, int terms = 50):
            cdef double sum
            cdef double power
            cdef double fact
            cdef int i
            sum = 0.
            power = 1.
            fact = 1.
            for i in range(terms):
               sum += power/fact
               power *= x
               fact *= i+1
            return sum
```

```
building '_cython_magic_f09447b346fe20b620ea715f5b8935ac' extension
C compiler: gcc -pthread -B /usr/share/miniconda3/envs/runenv/compiler_compat -Wl,--sysroot=/ -Wsign-co

compile options: '-I/usr/share/miniconda3/envs/runenv/include/python3.8 -c'
```

```
gcc: /home/runner/.cache/ipython/cython/_cython_magic_f09447b346fe20b620ea715f5b8935ac.c
gcc -pthread -shared -B /usr/share/miniconda3/envs/runenv/compiler_compat -L/usr/share/miniconda3/envs/
```

```
In [26]: %timeit exp_cython(1.,50)
```

```
339 ns ± 2.83 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

17.9 Cython and Numpy

The Numpy library contains many fast numerics routines. Their speed comes from manipulating the low-level C-arrays that the numpy.array object wraps rather than computing over slow Python lists. Using Cython one can access those low-level arrays and implement their own fast algorithms while allowing the easy interaction afforded by Python + Numpy.

The examples below are various implementations of the naive matrix multiplication algorithm. We will start with a pure Python implementation and then incrementally add structures that allow Cython to exploit the low-level speed of the numpy.array object.

17.9.1 Pure Python implementation compiled in Cython without specific optimizations.

```
In [27]: %%cython
         def matmul1(A, B, out=None):
             assert A.shape[1] == B.shape[0]
             for i in range(A.shape[0]):
                 for j in range(B.shape[1]):
                     s = 0
                     for k in range(A.shape[1]):
                         s += A[i,k] * B[k,j]
                     out[i,j] = s
             return out
```

```
building '_cython_magic_899b76c7c47e2932ff2e677e49192f5a' extension
C compiler: gcc -pthread -B /usr/share/miniconda3/envs/runenv/compiler_compat -Wl,--sysroot=/ -Wsign-co

compile options: '-I/usr/share/miniconda3/envs/runenv/include/python3.8 -c'
gcc: /home/runner/.cache/ipython/cython/_cython_magic_899b76c7c47e2932ff2e677e49192f5a.c
gcc -pthread -shared -B /usr/share/miniconda3/envs/runenv/compiler_compat -L/usr/share/miniconda3/envs/
```

17.9.2 Import numpy as a Cython module

We now take advantage of the ability to access the underlying C arrays in the `numpy.array` object from Cython, thanks to a special `numpy.pxd` file included with Cython. (The Cython developers worked closely with Numpy developers to make this optimal.)

To begin with, we have to `cimport` numpy: that is, import numpy as a **Cython** module rather than a **Python** module. To do so, simply type:

```
cimport numpy as np
```

Another important thing to note is the type of Numpy indexers. There is a special Numpy variable type used for `numpy.array` indices called `Py_ssize_t`. To take full advantage of the speedups that Cython can provide we should make sure to type the variables used for indexing as such.

```
In [28]: %%cython
         import numpy as np
         cimport numpy as np
         ctypedef np.float64_t dtype_t      # shorthand type. easy to change
         def matmul2(np.ndarray[dtype_t, ndim=2] A,
                     np.ndarray[dtype_t, ndim=2] B,
                     np.ndarray[dtype_t, ndim=2] out=None):
             cdef Py_ssize_t i, j, k
             cdef dtype_t s
             assert A.shape[1] == B.shape[0]
             for i in range(A.shape[0]):
                 for j in range(B.shape[1]):
                     s = 0
                     for k in range(A.shape[1]):
                         s += A[i,k] * B[k,j]
                     out[i,j] = s
             return out

building '_cython_magic_44357700cb4ec61887ddbee898be64d7' extension
C compiler: gcc -pthread -B /usr/share/miniconda3/envs/runenv/compiler_compat -Wl,--sysroot=/ -Wsign-co

compile options: '-I/usr/share/miniconda3/envs/runenv/lib/python3.8/site-packages/numpy/core/include -I
gcc: /home/runner/.cache/ipython/cython/_cython_magic_44357700cb4ec61887ddbee898be64d7.c
gcc -pthread -shared -B /usr/share/miniconda3/envs/runenv/compiler_compat -L/usr/share/miniconda3/envs/


In [29]: import numpy as np
         from timeit import timeit
         A = np.random.random_sample((64,64))
         B = np.random.random_sample((64,64))
         C = np.zeros((64,64))

In [30]: %timeit matmul1(A,B,C)

153 ms ± 1.62 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)


In [31]: %timeit matmul2(A,B,C)

327 µs ± 2.43 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

17.9.3 Tuning indexing

The array lookups are still slowed down by two factors: * Bounds checking is performed. * Negative indices are checked for and handled correctly.

The code doesn't use negative indices, and always access to arrays within bounds. We can add a decorator to disable bounds checking:

```
In [32]: %%cython
         cimport cython                                          # cython tools
         import numpy as np
         cimport numpy as np
         ctypedef np.float64_t dtype_t
         @cython.boundscheck(False) # turn off bounds-checking for entire function
         @cython.wraparound(False)  # turn off negative index wrapping for entire function
         def matmul3(np.ndarray[dtype_t, ndim=2] A,
                     np.ndarray[dtype_t, ndim=2] B,
```

```
                np.ndarray[dtype_t, ndim=2] out=None):
            cdef Py_ssize_t i, j, k
            cdef dtype_t s
            assert A.shape[1] == B.shape[0]
            for i in range(A.shape[0]):
                for j in range(B.shape[1]):
                    s = 0
                    for k in range(A.shape[1]):
                        s += A[i,k] * B[k,j]
                    out[i,j] = s
            return out

building '_cython_magic_75874d7a10a3fb62ace5ee487efe1f5a' extension
C compiler: gcc -pthread -B /usr/share/miniconda3/envs/runenv/compiler_compat -Wl,--sysroot=/ -Wsign-co

compile options: '-I/usr/share/miniconda3/envs/runenv/lib/python3.8/site-packages/numpy/core/include -I
gcc: /home/runner/.cache/ipython/cython/_cython_magic_75874d7a10a3fb62ace5ee487efe1f5a.c
gcc -pthread -shared -B /usr/share/miniconda3/envs/runenv/compiler_compat -L/usr/share/miniconda3/envs/
```

```
In [33]: %timeit matmul3(A,B,C)
```

```
287 µs ± 4.07 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

17.10 Cython Build Options

- boundcheck(True,False) : array bounds checking
- wraparound(True,False) : negative indexing.
- initializedcheck(True,False): checks that a memoryview is initialized
- nonecheck(True,False) : Check if one argument is None
- overflowcheck(True,False) : Check if int are too big
- cdivision(True,False) : If False, adjust the remainder and quotient operators C types to match those of Python ints. Could be very effective when it is set to True.
- profile (True / False) : Write hooks for Python profilers into the compiled C code. Default is False.

Cython Compiler directives

17.11 Numpy objects with external C program.

Note that this can actually be slower because the C function is not the best implementation of matrix multiplication. Call cblas with same technique is an interesting exercise.

```
In [34]: %%file mydgemm.c
         void my_dgemm( int m, int n, int k,
                        double a[m][n], double b[n][k], float c[m][k] )
         {
           double ab = 0;
           for( int j = 0 ; j < m ; j++ ) {
             for( int i = 0 ; i < k ; i++ ) {
               for( int l = 0 ; l < n ; l++ ){
                 ab += a[j][l] * b[l][i];
               }
               c[j][i] = ab ;
               ab = 0;
             }
```

```
    }
}
```

```
Writing mydgemm.c
```

- The `np.ndarray[double, ndim=2, mode="c"]` assures that you get a C-contiguous numpy array of doubles
- The `&input[0,0]` passed in the address of the beginning of the data array.

```
In [35]: from pyximport import install
         import os
         here = os.getcwd()
```

```
In [36]: %%cython -I {here}
         # do not forget to change the file path
         cdef extern from "mydgemm.c":
             void my_dgemm (int m, int n, int k,
                            double *A, double *B, double *C)
         cimport cython
         import numpy as np
         cimport numpy as np
         ctypedef np.float64_t dtype_t
         @cython.boundscheck(False)
         @cython.wraparound(False)
         def matmul4(np.ndarray[dtype_t, ndim=2, mode="c"] A,
                     np.ndarray[dtype_t, ndim=2, mode="c"] B,
                     np.ndarray[dtype_t, ndim=2, mode="c"] C=None):
             cdef int m = A.shape[0]
             cdef int n = A.shape[1]
             cdef int k = B.shape[1]
             cdef dtype_t s

             my_dgemm(m, n, k, &A[0,0], &B[0,0], &C[0,0])

             return C
```

```
building '_cython_magic_3182af7b83c2474db8f1f7bb74b9f4d7' extension
C compiler: gcc -pthread -B /usr/share/miniconda3/envs/runenv/compiler_compat -Wl,--sysroot=/ -Wsign-co

compile options: '-I/usr/share/miniconda3/envs/runenv/lib/python3.8/site-packages/numpy/core/include -I
gcc: /home/runner/.cache/ipython/cython/_cython_magic_3182af7b83c2474db8f1f7bb74b9f4d7.c
gcc -pthread -shared -B /usr/share/miniconda3/envs/runenv/compiler_compat -L/usr/share/miniconda3/envs/
```

```
In [37]: %timeit matmul4(A,B,C)
```

```
274 µs ± 4 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

17.11.1 Exercise : Find prime numbers < 10000

```
In [38]: # %load solutions/cython/is_prime0.py

         def is_prime0(n):
             if n < 4: return True
             if n % 2 == 0 : return False
             k = 3
             while k*k <= n:
```

```
            if n % k == 0: return False
            k += 2
        return True
```

```
In [39]: [ p for p in range(20) if is_prime0(p)]
```

```
Out[39]: [0, 1, 2, 3, 5, 7, 11, 13, 17, 19]
```

```
In [40]: L = list(range(10000))
         %timeit [ p for p in L if is_prime0(p)]
```

```
8.13 ms ± 95.8 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
In [41]: %%cython
         def is_prime1(n):
             if n < 4: return True
             if n % 2 == 0 : return False
             k = 3
             while k*k <= n:
                 if n % k == 0: return False
                 k += 2
             return True
```

```
building '_cython_magic_f81b23461181c5d78ab3700de598d3b5' extension
C compiler: gcc -pthread -B /usr/share/miniconda3/envs/runenv/compiler_compat -Wl,--sysroot=/ -Wsign-co

compile options: '-I/usr/share/miniconda3/envs/runenv/lib/python3.8/site-packages/numpy/core/include -I
gcc: /home/runner/.cache/ipython/cython/_cython_magic_f81b23461181c5d78ab3700de598d3b5.c
gcc -pthread -shared -B /usr/share/miniconda3/envs/runenv/compiler_compat -L/usr/share/miniconda3/envs/
```

```
In [42]: [ p for p in range(20) if is_prime1(p)]
```

```
Out[42]: [0, 1, 2, 3, 5, 7, 11, 13, 17, 19]
```

```
In [43]: %timeit [p  for p in L if is_prime1(p)]
```

```
4.01 ms ± 59.8 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

17.11.2 Add Cython types without modifying the Python Code

```
In [44]: %%cython
         import cython
         @cython.locals(n=int, k=int)
         def is_prime2(n):
             if n < 4: return True
             if n % 2 == 0 : return False
             k = 3
             while k*k <= n:
                 if n % k == 0: return False
                 k += 2
             return True
```

```
building '_cython_magic_745b858a4456e55af5f92e69b9968cd3' extension
C compiler: gcc -pthread -B /usr/share/miniconda3/envs/runenv/compiler_compat -Wl,--sysroot=/ -Wsign-co

compile options: '-I/usr/share/miniconda3/envs/runenv/lib/python3.8/site-packages/numpy/core/include -I
gcc: /home/runner/.cache/ipython/cython/_cython_magic_745b858a4456e55af5f92e69b9968cd3.c
gcc -pthread -shared -B /usr/share/miniconda3/envs/runenv/compiler_compat -L/usr/share/miniconda3/envs/
```

```
In [45]: [ p for p in range(20) if is_prime2(p)]
```

```
Out[45]: [0, 1, 2, 3, 5, 7, 11, 13, 17, 19]
```

```
In [46]: %timeit [p for p in L if is_prime2(p) ]
```

```
661 µs ± 3.29 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

17.11.3 Cython function

```
In [47]: %%cython
         import cython
         cdef bint is_prime3(int n):
             if n < 4: return True
             if n % 2 == 0: return False
             cdef int k = 3
             while k*k <= n:
                 if n % k == 0: return False
                 k += 2
             return True
         def prime_list(L):
             return [p for p in L if is_prime3(p)]
```

```
building '_cython_magic_666b225df8666611ff2af72a85a03e9d' extension
C compiler: gcc -pthread -B /usr/share/miniconda3/envs/runenv/compiler_compat -Wl,--sysroot=/ -Wsign-co

compile options: '-I/usr/share/miniconda3/envs/runenv/lib/python3.8/site-packages/numpy/core/include -I
gcc: /home/runner/.cache/ipython/cython/_cython_magic_666b225df8666611ff2af72a85a03e9d.c
gcc -pthread -shared -B /usr/share/miniconda3/envs/runenv/compiler_compat -L/usr/share/miniconda3/envs/
```

```
In [48]: prime_list(list(range(20)))
```

```
Out[48]: [0, 1, 2, 3, 5, 7, 11, 13, 17, 19]
```

```
In [49]: %timeit prime_list(L)
```

```
361 µs ± 5.36 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

```
In [50]: %%cython
         import cython
         from numpy cimport ndarray
         import numpy

         cdef bint is_prime3(int n):
             if n < 4: return True
             if n % 2 == 0: return False
             cdef int k = 3
             while k*k <= n:
                 if n % k == 0: return False
                 k += 2
             return True

         def prime_array(ndarray[int, ndim=1] L):
             cdef ndarray[int, ndim=1] res = ndarray(shape=(L.shape[0]),dtype=numpy.int32)
             cdef int i
             for i in range(L.shape[0]):
                 res[i] = is_prime3(L[i])
             return L[res==1]
```

```
building '_cython_magic_2a5b09b5e9c574f8e7fa68917eb3edc0' extension
C compiler: gcc -pthread -B /usr/share/miniconda3/envs/runenv/compiler_compat -Wl,--sysroot=/ -Wsign-co

compile options: '-I/usr/share/miniconda3/envs/runenv/lib/python3.8/site-packages/numpy/core/include -I
gcc: /home/runner/.cache/ipython/cython/_cython_magic_2a5b09b5e9c574f8e7fa68917eb3edc0.c
gcc -pthread -shared -B /usr/share/miniconda3/envs/runenv/compiler_compat -L/usr/share/miniconda3/envs/
```

```
In [51]: import numpy as np
         prime_array(np.arange(20,dtype=np.int32))
```

```
Out[51]: array([ 0,  1,  2,  3,  5,  7, 11, 13, 17, 19], dtype=int32)
```

```
In [52]: npL = numpy.array(L,dtype=np.int32)
         %timeit prime_array(npL)
```

```
362 µs ± 6.9 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

17.12 Using Parallelism

- Cython supports native parallelism via OpenMP
- by default, Python's Global Interpreter Lock (GIL) prevents that several threads use the Python interpreter simultaneously
- to use this kind of parallelism, the GIL must be released

If you have a default compiler with openmp support you can use this magic command in your notebook.

```
%%cython --compile-args=-fopenmp --link-args=-fopenmp
```

```
In [53]: %%file cython_omp.pyx
         import cython
         from cython.parallel cimport parallel, prange  # import parallel functions
         import numpy as np
         from numpy cimport ndarray

         cdef bint is_prime4(int n) nogil:      #release the gil
             if n < 4: return True
             if n % 2 == 0: return False
             cdef int k = 3
             while k*k <= n:
                 if n % k == 0: return False
                 k += 2
             return True

         @cython.boundscheck(False)
         def prime_array_omp(ndarray[int, ndim=1] L):
             cdef ndarray[int, ndim=1] res = ndarray(shape=(L.shape[0]),dtype=np.int32)
             cdef Py_ssize_t i
             with nogil, parallel(num_threads=4):
                 for i in prange(L.shape[0]):     #Parallel loop
                     res[i] = is_prime4(L[i])
             return L[res==1]
```

```
Writing cython_omp.pyx
```

To use the OpenMP support, you need to enable OpenMP. For gcc this can be done as follows in a setup.py:

```
In [54]: %%file setup.py
         from distutils.core import setup
         from distutils.extension import Extension
         from Cython.Build import cythonize
         import os, sys
         import numpy

         if sys.platform == "darwin": # for omp, use gcc installed with brew
             os.environ["CC"]="gcc-10"
             os.environ["CXX"]="g++-10"

         ext_modules = [
             Extension(
                 "cython_omp",
                 ["cython_omp.pyx"],
                 extra_compile_args=['-fopenmp'],
                 extra_link_args=['-fopenmp'],
                 include_dirs=[numpy.get_include()]
             )
         ]

         setup(
             name='Cython OpenMP Example',
             ext_modules=cythonize(ext_modules),
         )
         # python setup.py build_ext --inplace

Overwriting setup.py


In [55]: %run setup.py build_ext --inplace --quiet

Compiling cython_omp.pyx because it changed.
[1/1] Cythonizing cython_omp.pyx
running build_ext
building 'cython_omp' extension
C compiler: gcc -pthread -B /usr/share/miniconda3/envs/runenv/compiler_compat -Wl,--sysroot=/ -Wsign-co

compile options: '-I/usr/share/miniconda3/envs/runenv/lib/python3.8/site-packages/numpy/core/include -I
extra options: '-fopenmp'
gcc: cython_omp.c
gcc -pthread -shared -B /usr/share/miniconda3/envs/runenv/compiler_compat -L/usr/share/miniconda3/envs/


In [56]: from cython_omp import prime_array_omp

In [57]: prime_array_omp(np.arange(20,dtype=np.int32))

Out[57]: array([ 0,  1,  2,  3,  5,  7, 11, 13, 17, 19], dtype=int32)

In [58]: %timeit prime_array_omp(npL)

275 µs ± 13.8 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

17.13 References

- Cython documentation

- An Interactive Introduction to Cython by Chris Swierczewski
- Introduction To Python by Michael Kraus
- Cython by Xavier Juvigny
- Cython: C-Extensions for Python, Wiki
- Kurt W. Smith
  - Cython A Guide for Python Programmers
  - Cython: Blend the Best of Python and C++ | SciPy 2015 Tutorial | Kurt Smith
  - Cython: Speed up Python and NumPy, Pythonize C, C++, and Fortran, SciPy2013 Kurt W. Smith
  - SciPy 2017 - Cython by
  - Cython Book examples
- Parallel computing in Cython/threads - Neal Hughes

Chapter 18

Numba

```
In [1]: import numpy as np
```

- Numba is a compiler for Python array and numerical functions.
- Numba generates optimized machine code from pure Python code with a few simple annotations
- Python code is just-in-time optimized to performance similar as C, C++ and Fortran, without having to switch languages or Python interpreters.
- The code is generated on-the-fly for CPU (default) or GPU hardware.

18.1 Python decorator

A decorator is used to modify a function or a class. A reference to a function "func" or a class "C" is passed to a decorator and the decorator returns a modified function or class. The modified functions or classes usually contain calls to the original function "func" or class "C".

```
In [2]: def timeit(function):
            def wrapper(*args, **kargs):
                import time
                t1 = time.time()
                result = function(*args, **kargs)
                t2 = time.time()
                print("execution time", t2-t1)
                return result
            return wrapper

        @timeit
        def f(a, b):
            return a + b

        print(f(1, 2))
```

```
execution time 7.152557373046875e-07
3
```

18.2 First example

```
In [3]: from numba import jit
        @jit
        def sum(a, b):
            return a + b
```

- Compilation will be deferred until the first function execution.
- Numba will infer the argument types at call time.

```
In [4]: sum(1, 2), sum(1j, 2)
```

```
Out[4]: (3, (2+1j))
```

```
In [5]: x = np.random.rand(10)
        y = np.random.rand(10)
        sum(x, y)
```

```
Out[5]: array([0.86501866, 0.86999041, 0.90119998, 1.8825215 , 0.93424875,
               1.664764  , 0.89403158, 0.60210291, 0.80069788, 0.60922009])
```

18.3 Performance

```
In [6]: x = np.random.rand(10000000)
```

```
In [7]: %timeit x.sum() # Numpy
```

```
7.21 ms ± 100 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
In [8]: @jit
        def numba_sum(x):
            res= 0
            for i in range(x.size):
                res += x[i]
            return res
```

```
In [9]: %timeit numba_sum(x)
```

```
13.4 ms ± 104 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

18.4 Numba methods

```
In [10]: @jit
         def jit_sum(a, b):
             return a + b
```

```
In [11]: jit_sum.inspect_types() # jit_sum has not been compiled
```

```
In [12]: jit_sum(1, 2) # call it once with ints
         jit_sum.inspect_types()
```

```
jit_sum (int64, int64)
--------------------------------------------------------------------------------
# File: <ipython-input-10-ebda2b2f7dda>
# --- LINE 1 ---

@jit

# --- LINE 2 ---

def jit_sum(a, b):
```

```
    # --- LINE 3 ---
    # label 0
    #   a = arg(0, name=a)  :: int64
    #   b = arg(1, name=b)  :: int64
    #   $6binary_add.2 = a + b  :: int64
    #   del b
    #   del a
    #   $8return_value.3 = cast(value=$6binary_add.2)  :: int64
    #   del $6binary_add.2
    #   return $8return_value.3

    return a + b


================================================================================


In [13]: jit_sum(1., 2.) # call it once with doubles
         jit_sum.inspect_types()

jit_sum (int64, int64)
--------------------------------------------------------------------------------
# File: <ipython-input-10-ebda2b2f7dda>
# --- LINE 1 ---

@jit

# --- LINE 2 ---

def jit_sum(a, b):

    # --- LINE 3 ---
    # label 0
    #   a = arg(0, name=a)  :: int64
    #   b = arg(1, name=b)  :: int64
    #   $6binary_add.2 = a + b  :: int64
    #   del b
    #   del a
    #   $8return_value.3 = cast(value=$6binary_add.2)  :: int64
    #   del $6binary_add.2
    #   return $8return_value.3

    return a + b


================================================================================
jit_sum (float64, float64)
--------------------------------------------------------------------------------
# File: <ipython-input-10-ebda2b2f7dda>
# --- LINE 1 ---

@jit

# --- LINE 2 ---
```

```
def jit_sum(a, b):

    # --- LINE 3 ---
    # label 0
    #   a = arg(0, name=a)  :: float64
    #   b = arg(1, name=b)  :: float64
    #   $6binary_add.2 = a + b  :: float64
    #   del b
    #   del a
    #   $8return_value.3 = cast(value=$6binary_add.2)  :: float64
    #   del $6binary_add.2
    #   return $8return_value.3

    return a + b


================================================================================
```

- `jit_sum.inspect_llvm()` returns a dict with llvm representation.

LLVM is a library that is used to construct, optimize and produce intermediate and/or binary machine code.

- `jit_sum.inspect_asm()` returns a dict with assembler information.

```
In [14]: jit_sum.py_func(1, 2) # call origin python function without numba process
```

```
Out[14]: 3
```

18.5 Types coercion

Tell Numba the function signature you are expecting.

```
In [15]: @jit(['int32[:](int32[:], int32[:])','int32(int32, int32)'])
         def product(a, b):
             return a*b
```

```
In [16]: product(2, 3), product(2.2, 3.2)
```

```
Out[16]: (6, 6)
```

```
In [17]: a = np.arange(10, dtype=np.int32)
         b = np.arange(10, dtype=np.int32)
         product(a, b)
```

```
Out[17]: array([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81], dtype=int32)
```

```
In [18]: a = np.random.random(10) # Numpy arrays contain double by default
         b = np.random.random(10)
         try:
             product(a, b)
         except TypeError as e:
             print("TypeError:",e)
```

```
TypeError: No matching definition for argument type(s) array(float64, 1d, C), array(float64, 1d, C)
```

18.6 Numba types

```
void,
intp, uintp,
intc, uintc,
int8, uint8, int16, uint16, int32, uint32, int64, uint64,
float32, float64,
complex64, complex128.
```

18.6.1 Arrays

```
float32[:]
float64[:, :]
```

18.7 Numba compilation options

- ** nopython ** : Compilation fails if you use pure Python objects.
- ** nogil ** : release Python's global interpreter lock (GIL).
- ** cache ** : Do not recompile the function each time you invoke a Python program.
- ** parallel ** : experimental feature that automatically parallelizes must be used in conjunction with nopython=True:

18.8 Inlining

Numba-compiled functions can call other compiled functions. The function calls may even be inlined in the native code, depending on optimizer heuristics.

```
In [19]: import math
         from numba import njit

         @njit
         def square(x):
             return x ** 2

         @njit
         def hypot(x, y):
             return math.sqrt(square(x) + square(y)) # square function is inlined
```

```
In [20]: hypot(2., 3.)
```

```
Out[20]: 3.605551275463989
```

18.9 @vectorize decorator

- Numba's vectorize allows Python functions taking scalar input arguments to be used as NumPy ufuncs.
- Write your function as operating over input scalars, rather than arrays. Numba will generate the surrounding loop (or kernel) allowing efficient iteration over the actual inputs.

18.9.1 Two modes of operation:

1. Eager mode: If you pass one or more type signatures to the decorator, you will be building a Numpy universal function (ufunc).
2. Call-time mode: When not given any signatures, the decorator will give you a Numba dynamic universal function (DUFunc) that dynamically compiles a new kernel when called with a previously unsupported input type.

```
In [21]: from numba import vectorize, float64, float32, int32, int64

         @vectorize([float64(float64, float64)])
         def f(x, y):
             return x + y
```

If you pass several signatures, beware that you have to pass most specific signatures before least specific ones (e.g., single-precision floats before double-precision floats)

```
In [22]: @vectorize([int32(int32, int32),
                     int64(int64, int64),
                     float32(float32, float32),
                     float64(float64, float64)])
         def f(x, y):
             return x + y

In [23]: a = np.arange(6)
         f(a, a)

Out[23]: array([ 0,  2,  4,  6,  8, 10])

In [24]: a = np.linspace(0, 1, 6)
         f(a, a)

Out[24]: array([0. , 0.4, 0.8, 1.2, 1.6, 2. ])

In [25]: a = np.linspace(0, 1+1j, 6)
         f(a, a)


         ---------------------------------------------------------------------------

         TypeError                                 Traceback (most recent call last)

         <ipython-input-25-b196490ab338> in <module>
           1 a = np.linspace(0, 1+1j, 6)
     ----> 2 f(a, a)


         TypeError: ufunc 'f' not supported for the input types, and the inputs could not be safely coer
```

18.9.2 Why not using a simple iteration loop using the @jit decorator?

The answer is that NumPy ufuncs automatically get other features such as reduction, accumulation or broadcasting.

```
In [26]: a = np.arange(12).reshape(3, 4)
         a

Out[26]: array([[ 0,  1,  2,  3],
                [ 4,  5,  6,  7],
                [ 8,  9, 10, 11]])

In [27]: f.reduce(a, axis=0)

Out[27]: array([12, 15, 18, 21])

In [28]: f.reduce(a, axis=1)
```

```
Out[28]: array([ 6, 22, 38])

In [29]: f.accumulate(a)

Out[29]: array([[ 0,  1,  2,  3],
                [ 4,  6,  8, 10],
                [12, 15, 18, 21]])

In [30]: f.accumulate(a, axis=1)

Out[30]: array([[ 0,  1,  3,  6],
                [ 4,  9, 15, 22],
                [ 8, 17, 27, 38]])
```

18.10 The vectorize() decorator supports multiple ufunc targets:

- **cpu** *Single-threaded CPU* : small data sizes (approx. less than 1KB), no overhead.
- **parallel** *Multi-core CPU* : medium data sizes (approx. less than 1MB), small overhead.
- **cuda** *CUDA GPU* big data sizes (approx. greater than 1MB), significant overhead.

18.11 The @guvectorize decorator

- It allows you to write ufuncs that will work on an arbitrary number of elements of input arrays, and take and return arrays of differing dimensions.

```
In [31]: from numba import guvectorize
         @guvectorize([(int64[:], int64[:], int64[:])], '(n),()->(n)')
         def g(x, y, res):
             for i in range(x.shape[0]):
                 res[i] = x[i] + y[0]  # adds the scalar y to all elements of x
```

This decorator has two arguments: - the declaration (n),()->(n) tells NumPy that the function takes a n-element one-dimension array, a scalar (symbolically denoted by the empty tuple ()) and returns a n-element one-dimension array; - the list of supported concrete signatures as in @vectorize; here we only support int64 arrays.

18.12 Automatic parallelization with @jit

- Setting the parallel option for jit() enables this experimental Numba feature.
- **Array Expressions like element-wise or point-wise array operations are supported.**
  - unary operators: + - ~
  - binary operators: + - * / /? % | >> ^ << & ** //
  - comparison operators: == != < <= > >=
  - Numpy ufuncs that are supported in nopython mode.
  - Numpy reduction functions sum and prod.
- Numpy array creation functions zeros, ones, and several random functions (rand, randn, ranf, random_sample, sample, random, standard_normal, chisquare, weibull, power, geometric, exponential, poisson, rayleigh, normal, uniform, beta, binomial, f, gamma, lognormal, laplace, randint, triangular).

Numpy dot function between a matrix and a vector, or two vectors. In all other cases, Numba's default implementation is used.

Multi-dimensional arrays are also supported for the above operations when operands have matching dimension and size. The full semantics of Numpy broadcast between arrays with mixed dimensionality or size is not supported, nor is the reduction across a selected dimension.

http://numba.pydata.org/numba-doc/latest/user/parallel.html

18.13 Explicit Parallel Loops

Another experimental feature of this module is support for explicit parallel loops. One can use Numba's prange instead of range to specify that a loop can be parallelized. The user is required to make sure that the loop does not have cross iteration dependencies except the supported reductions. Currently, reductions on scalar values are supported and are inferred from in-place operations. The example below demonstrates a parallel loop with a reduction (A is a one-dimensional Numpy array):

```
In [32]: from numba import njit, prange
         @njit(parallel=True)
         def prange_test(A):
             s = 0
             for i in prange(A.shape[0]):
                 s += A[i]
             return s
```

18.14 Exercise

- Optimize the Laplace equation solver with numba.

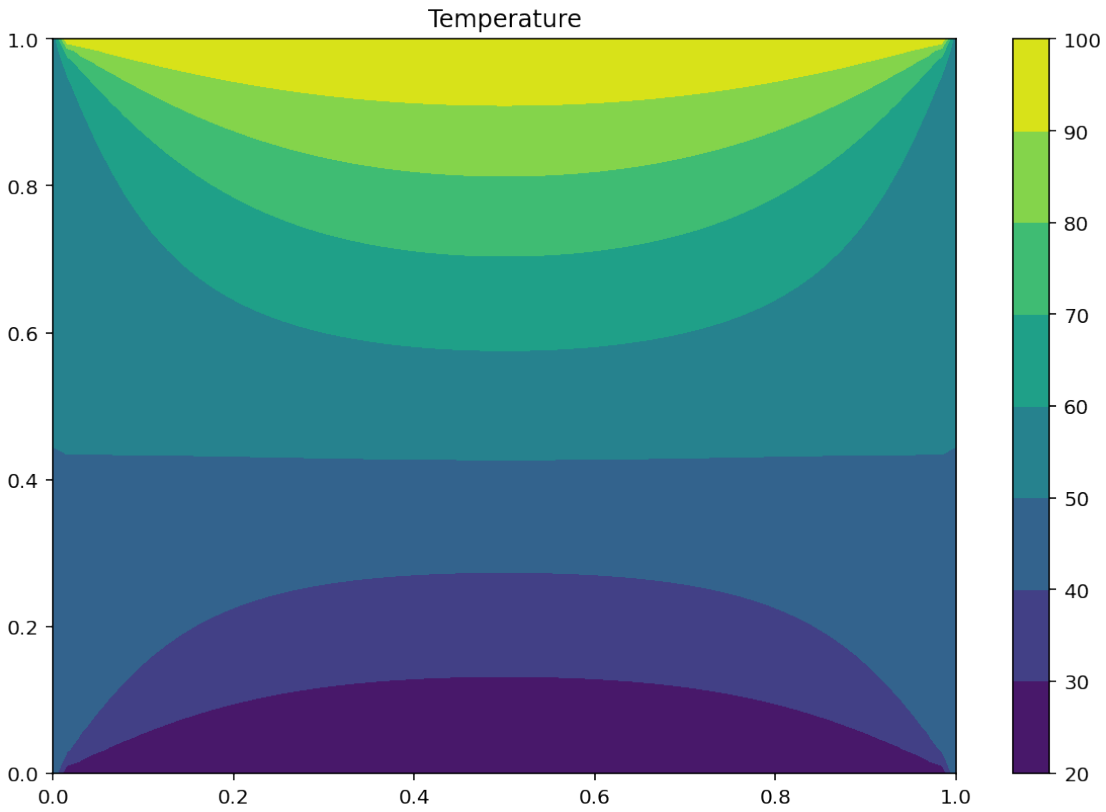
  1. Use only @jit
  2. Try to use @jit(nopython=True) option
  3. Optimize the laplace function with the right signature.
  4. Try to parallelize.

```
In [33]: %%time
         %matplotlib inline
         %config InlineBackend.figure_format = 'retina'
         import numpy as np
         import matplotlib.pyplot as plt
         import itertools
         from numba import jit, float64
         # Boundary conditions
         Tnorth, Tsouth, Twest, Teast = 100, 20, 50, 50

         # Set meshgrid
         n, l = 64, 1.0
         X, Y = np.meshgrid(np.linspace(0,l,n), np.linspace(0,l,n))
         T = np.zeros((n,n))

         # Set Boundary condition
         T[n-1:, :] = Tnorth
         T[:1, :] = Tsouth
         T[:, n-1:] = Teast
         T[:, :1] = Twest

         def laplace(T, n):
             residual = 0.0
             for i in range(1, n-1):
                 for j in range(1, n-1):
                     T_old = T[i,j]
                     T[i, j] = 0.25 * (T[i+1,j] + T[i-1,j] + T[i,j+1] + T[i,j-1])
                     if T[i,j]>0:
                         residual=max(residual,abs((T_old-T[i,j])/T[i,j]))
             return residual

         residual = 1.0
         istep = 0
```

```
        while residual > 1e-5 :
            istep += 1
            residual = laplace(T, n)
            print ((istep, residual), end="\r")

        print("\n iterations = ",istep)
        plt.rcParams['figure.figsize'] = (10,6.67)
        plt.title("Temperature")
        plt.contourf(X, Y, T)
        plt.colorbar()
```

```
(2457, 9.997295133247811e-06)
 iterations =  2457
CPU times: user 31.8 s, sys: 238 ms, total: 32 s
Wall time: 31.8 s
```

Out[33]: <matplotlib.colorbar.Colorbar at 0x7ffbb50dd190>



18.15 Vectorize performance

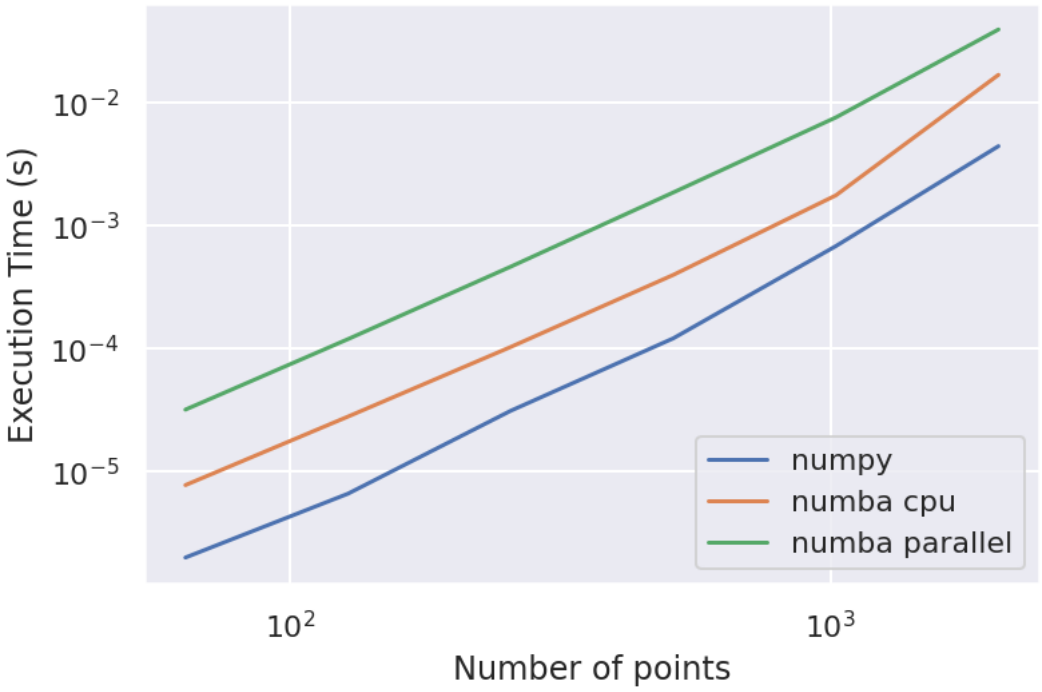
In [34]:
```
import socket
import numpy as np
from numba import vectorize
```

```
        @vectorize(['float64(float64, float64)'], target="cpu", cache=True, nopython=True)
        def cpu_add(a, b):
           return a + b

        @vectorize(['float64(float64, float64)'], target="parallel", cache=True, nopython=True)
        def parallel_add(a, b):
           return a + b

        if socket.gethostname() == "gpu-irmar.insa-rennes.fr":
            @vectorize(['float64(float64, float64)'], target="cuda", cache=True, nopython=True)
            def parallel_add(a, b):
               return a + b
```

```
/usr/share/miniconda3/envs/runenv/lib/python3.8/site-packages/numba/np/ufunc/parallel.py:363: NumbaWarni
  warnings.warn(problem)
```

```
In [35]: %matplotlib inline
         import matplotlib.pyplot as plt
         import seaborn; seaborn.set()
         import progressbar
         Nrange = (2 ** np.arange(6, 12)).astype(int)

         t_numpy = []
         t_numba_cpu = []
         t_numba_parallel = []

         bar = progressbar.ProgressBar()

         for N in bar(Nrange):
             # Initialize arrays

             A = np.ones(N*N, dtype=np.float32).reshape(N,N)
             B = np.ones(A.shape, dtype=A.dtype)
             C = np.empty_like(A, dtype=A.dtype)

             t1 = %timeit -oq C = A + B
             t2 = %timeit -oq C = cpu_add(A, B)
             t3 = %timeit -oq C = parallel_add(A, B)

             t_numpy.append(t1.best)
             t_numba_cpu.append(t2.best)
             t_numba_parallel.append(t3.best)

         plt.loglog(Nrange, t_numpy, label='numpy')
         plt.loglog(Nrange, t_numba_cpu, label='numba cpu')
         plt.loglog(Nrange, t_numba_parallel, label='numba parallel')
         plt.legend(loc='lower right')
         plt.xlabel('Number of points')
         plt.ylabel('Execution Time (s)');
```

```
100% (6 of 6) |############################| Elapsed Time: 0:01:58 Time:  0:01:58
```



18.16 References

- Numba by Loic Gouarin
- Numba Documentation
- Numbapro
- Numba examples

Chapter 19

Semi-Lagrangian method

Let us consider an abstract scalar advection equation of the form

$$\frac{\partial f}{\partial t} + a(x,t) \cdot \nabla f = 0.$$

The characteristic curves associated to this equation are the solutions of the ordinary differential equations

$$\frac{dX}{dt} = a(X(t),t)$$

We shall denote by $X(t,x,s)$ the unique solution of this equation associated to the initial condition $X(s) = x$.

The classical semi-Lagrangian method is based on a backtracking of characteristics. Two steps are needed to update the distribution function f^{n+1} at t^{n+1} from its value f^n at time t^n : 1. For each grid point x_i compute $X(t^n; x_i, t^{n+1})$ the value of the characteristic at t^n which takes the value x_i at t^{n+1}. 2. As the distribution solution of first equation verifies

$$f^{n+1}(x_i) = f^n(X(t^n; x_i, t^{n+1})),$$

we obtain the desired value of $f^{n+1}(x_i)$ by computing $f^n(X(t^n; x_i, t^{n+1})$ by interpolation as $X(t^n; x_i, t^{n+1})$ is in general not a grid point.

Eric Sonnendrücker - Numerical methods for the Vlasov equations

```
In [1]: %matplotlib inline
        %config InlineBackend.figure_format = 'retina'
        import matplotlib.pyplot as plt
        plt.rcParams['figure.figsize'] = (10.0, 6.0)
```

```
In [2]: # Disable the pager for lprun
        from IPython.core import page
        page.page = print
```

19.1 Bspline interpolator

- De Boor's Algorithm - Wikipedia

19.1.1 Numpy

```
In [3]: def bspline_python(p, j, x):
            """Return the value at x in [0,1[ of the B-spline with
            integer nodes of degree p with support starting at j.
            Implemented recursively using the de Boor's recursion formula"""
            assert (x >= 0.0) & (x <= 1.0)
```

```
            assert (type(p) == int) & (type(j) == int)
            if p == 0:
                if j == 0:
                    return 1.0
                else:
                    return 0.0
            else:
                w = (x - j) / p
                w1 = (x - j - 1) / p
            return w * bspline_python(p - 1, j, x) + (1 - w1) * bspline_python(p - 1, j + 1, x)
```

```
In [4]: import numpy as np
        from scipy.fftpack import fft, ifft

        class BSplineNumpy:

            """ Class to compute BSL advection of 1d function """

            def __init__(self, p, xmin, xmax, ncells):
                assert p & 1 == 1  # check that p is odd
                self.p = p
                self.ncells = ncells
                # compute eigenvalues of degree p b-spline matrix
                self.modes = 2 * np.pi * np.arange(ncells) / ncells
                self.deltax = (xmax - xmin) / ncells

                self.eig_bspl = bspline_python(p, -(p + 1) // 2, 0.0)
                for j in range(1, (p + 1) // 2):
                    self.eig_bspl += bspline_python(p, j - (p + 1) // 2, 0.0) * 2 * np.cos(j * self.modes)

                self.eigalpha = np.zeros(ncells, dtype=complex)

            def interpolate_disp(self, f, alpha):
                """compute the interpolating spline of degree p of odd degree
                of a function f on a periodic uniform mesh, at
                all points xi-alpha"""
                p = self.p
                assert (np.size(f) == self.ncells)
                # compute eigenvalues of cubic splines evaluated at displaced points
                ishift = np.floor(-alpha / self.deltax)
                beta = -ishift - alpha / self.deltax
                self.eigalpha.fill(0.)
                for j in range(-(p-1)//2, (p+1)//2 + 1):
                    self.eigalpha += bspline_python(p, j-(p+1)//2, beta) * np.exp((ishift+j)*1j*self.modes)

                # compute interpolating spline using fft and properties of circulant matrices
                return np.real(ifft(fft(f) * self.eigalpha / self.eig_bspl))
```

19.1.2 Interpolation test

sin function after a displacement of alpha

```
In [5]: def interpolation_test(BSplineClass):
            """ Test to check interpolation"""
            n = 64
            cs = BSplineClass(3,0,1,n)
            x = np.linspace(0,1,n, endpoint=False)
            f = np.sin(x*4*np.pi)
            alpha = 0.2
```

```
            return np.allclose(np.sin((x-alpha)*4*np.pi), cs.interpolate_disp(f, alpha))


        interpolation_test(BSplineNumpy)
```

```
Out[5]: True
```

19.1.3 Profiling the code

```
In [6]: %load_ext line_profiler
```

```
In [7]: n =1024
        cs = BSplineNumpy(3,0,1,n)
        x = np.linspace(0,1,n, endpoint=False)
        f = np.sin(x*4*np.pi)
        alpha = 0.2;
```

```
In [8]: %lprun -s -f cs.interpolate_disp -T lp_results.txt cs.interpolate_disp(f, alpha);
```

```
Timer unit: 1e-06 s

Total time: 0.001006 s
File: <ipython-input-4-c5af2f880bfd>
Function: interpolate_disp at line 22

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    22                                           def interpolate_disp(self, f, alpha):
    23                                               """compute the interpolating spline of degree p
    24                                               of a function f on a periodic uniform mesh, at
    25                                               all points xi-alpha"""
    26         1          2.0      2.0      0.2          p = self.p
    27         1          9.0      9.0      0.9          assert (np.size(f) == self.ncells)
    28                                                   # compute eigenvalues of cubic splines evaluate
    29         1          9.0      9.0      0.9          ishift = np.floor(-alpha / self.deltax)
    30         1          2.0      2.0      0.2          beta = -ishift - alpha / self.deltax
    31         1         16.0     16.0      1.6          self.eigalpha.fill(0.)
    32         5          9.0      1.8      0.9          for j in range(-(p-1)//2, (p+1)//2 + 1):
    33         4        803.0    200.8     79.8              self.eigalpha += bspline_python(p, j-(p+1),
    34
    35                                                   # compute interpolating spline using fft and pr
    36         1        156.0    156.0     15.5          return np.real(ifft(fft(f) * self.eigalpha / sc

*** Profile printout saved to text file 'lp_results.txt'.
```

19.1.4 Fortran

Replace the bspline computation by a fortran function, call it **bspline_fortran**.

```
In [9]: %load_ext fortranmagic
```

```
In [10]: %%fortran
         recursive function bspline_fortran(p, j, x) result(res)
             integer :: p, j
             real(8) :: x, w, w1
             real(8) :: res
```

```
            if (p == 0) then
                if (j == 0) then
                    res = 1.0
                    return
                else
                    res = 0.0
                    return
                end if
            else
                w = (x - j) / p
                w1 = (x - j - 1) / p
            end if

            res = w * bspline_fortran(p-1,j,x) &
            +(1-w1)*bspline_fortran(p-1,j+1,x)

        end function bspline_fortran
```

In [11]:
```
import numpy as np
from scipy.fftpack import fft, ifft

class BSplineFortran:

    def __init__(self, p, xmin, xmax, ncells):
        assert p & 1 == 1  # check that p is odd
        self.p = p
        self.ncells = ncells
        # compute eigenvalues of degree p b-spline matrix
        self.modes = 2 * np.pi * np.arange(ncells) / ncells
        self.deltax = (xmax - xmin) / ncells

        self.eig_bspl = bspline_fortran(p, -(p+1)//2, 0.0)
        for j in range(1, (p+1)//2):
            self.eig_bspl += bspline_fortran(p, j-(p+1)//2,0.0)*2*np.cos(j*self.modes)

        self.eigalpha = np.zeros(ncells, dtype=complex)

    def interpolate_disp(self, f, alpha):
        """compute the interpolating spline of degree p of odd degree
        of a function f on a periodic uniform mesh, at
        all points xi-alpha"""
        p = self.p
        assert (np.size(f) == self.ncells)
        # compute eigenvalues of cubic splines evaluated at displaced points
        ishift = np.floor(-alpha / self.deltax)
        beta = -ishift - alpha / self.deltax
        self.eigalpha.fill(0.)
        for j in range(-(p-1)//2, (p+1)//2 + 1):
            self.eigalpha += bspline_fortran(p, j-(p+1)//2, beta) * np.exp((ishift+j)*1j*self.modes)

        # compute interpolating spline using fft and properties of circulant matrices
        return np.real(ifft(fft(f) * self.eigalpha / self.eig_bspl))
```

In [12]:
```
interpolation_test(BSplineFortran)
```

Out[12]: True

19.1.5 Numba

Create a optimized function of bspline python function with Numba. Call it bspline_numba.

In [13]:
```
# %load solutions/landau_damping/bspline_numba.py
from numba import jit,  int32, float64
from scipy.fftpack import fft, ifft

@jit("float64(int32,int32,float64)",nopython=True)
def bspline_numba(p, j, x):

        """Return the value at x in [0,1[ of the B-spline with
        integer nodes of degree p with support starting at j.
        Implemented recursively using the de Boor's recursion formula"""

        assert ((x >= 0.0) & (x <= 1.0))
        if p == 0:
            if j == 0:
                return 1.0
            else:
                return 0.0
        else:
            w = (x-j)/p
            w1 = (x-j-1)/p
        return w * bspline_numba(p-1,j,x)+(1-w1)*bspline_numba(p-1,j+1,x)
```

In [14]:
```
class BSplineNumba:

    def __init__(self, p, xmin, xmax, ncells):
        assert p & 1 == 1  # check that p is odd
        self.p = p
        self.ncells = ncells
        # compute eigenvalues of degree p b-spline matrix
        self.modes = 2 * np.pi * np.arange(ncells) / ncells
        self.deltax = (xmax - xmin) / ncells

        self.eig_bspl = bspline_numba(p, -(p+1)//2, 0.0)
        for j in range(1, (p + 1) // 2):
            self.eig_bspl += bspline_numba(p,j-(p+1)//2,0.0)*2*np.cos(j*self.modes)

        self.eigalpha = np.zeros(ncells, dtype=complex)

    def interpolate_disp(self, f, alpha):
        """compute the interpolating spline of degree p of odd degree
        of a function f on a periodic uniform mesh, at
        all points xi-alpha"""

        p = self.p
        assert (np.size(f) == self.ncells)
        # compute eigenvalues of cubic splines evaluated at displaced points
        ishift = np.floor(-alpha / self.deltax)
        beta = -ishift - alpha / self.deltax
        self.eigalpha.fill(0.)
        for j in range(-(p-1)//2, (p+1)//2+1):
            self.eigalpha += bspline_numba(p, j-(p+1)//2, beta)*np.exp((ishift+j)*1j*self.modes)

        # compute interpolating spline using fft and properties of circulant matrices
        return np.real(ifft(fft(f) * self.eigalpha / self.eig_bspl))
```

In [15]:
```
interpolation_test(BSplineNumba)
```

Out[15]: True

19.1.6 Pythran

In [16]:
```
import pythran
```

In [17]:
```
%load_ext pythran.magic
```

In [18]:
```
# %load solutions/landau_damping/bspline_pythran.py

#pythran export bspline_pythran(int,int,float64)
def bspline_pythran(p, j, x):
    if p == 0:
        if j == 0:
            return 1.0
        else:
            return 0.0
    else:
        w = (x-j)/p
        w1 = (x-j-1)/p
    return w * bspline_pythran(p-1,j,x)+(1-w1)*bspline_pythran(p-1,j+1,x)
```

In [19]:
```
class BSplinePythran:

    def __init__(self, p, xmin, xmax, ncells):
        assert p & 1 == 1  # check that p is odd
        self.p = p
        self.ncells = ncells
        # compute eigenvalues of degree p b-spline matrix
        self.modes = 2 * np.pi * np.arange(ncells) / ncells
        self.deltax = (xmax - xmin) / ncells

        self.eig_bspl = bspline_pythran(p, -(p+1)//2, 0.0)
        for j in range(1, (p + 1) // 2):
            self.eig_bspl += bspline_pythran(p,j-(p+1)//2,0.0)*2*np.cos(j*self.modes)

        self.eigalpha = np.zeros(ncells, dtype=complex)

    def interpolate_disp(self, f, alpha):
        """compute the interpolating spline of degree p of odd degree
        of a function f on a periodic uniform mesh, at
        all points xi-alpha"""

        p = self.p
        assert (f.size == self.ncells)
        # compute eigenvalues of cubic splines evaluated at displaced points
        ishift = np.floor(-alpha / self.deltax)
        beta = -ishift - alpha / self.deltax
        self.eigalpha.fill(0.)
        for j in range(-(p-1)//2, (p+1)//2+1):
            self.eigalpha += bspline_pythran(p, j-(p+1)//2, beta)*np.exp((ishift+j)*1j*self.modes)

        # compute interpolating spline using fft and properties of circulant matrices
        return np.real(ifft(fft(f) * self.eigalpha / self.eig_bspl))
```

In [20]:
```
interpolation_test(BSplinePythran)
```

Out[20]: True

19.1.7 Cython

- Create **bspline_cython** function.

In [21]:
```
%load_ext cython
```

In [22]:
```
%%cython -a
def bspline_cython(p, j, x):
        """Return the value at x in [0,1[ of the B-spline with
        integer nodes of degree p with support starting at j.
        Implemented recursively using the de Boor's recursion formula"""
        assert (x >= 0.0) & (x <= 1.0)
        assert (type(p) == int) & (type(j) == int)
        if p == 0:
            if j == 0:
                return 1.0
            else:
                return 0.0
        else:
            w = (x - j) / p
            w1 = (x - j - 1) / p
        return w * bspline_cython(p - 1, j, x) + (1 - w1) * bspline_cython(p - 1, j + 1, x)
```

Out[22]: <IPython.core.display.HTML object>

In [23]:
```
%%cython
import cython
import numpy as np
cimport numpy as np
from scipy.fftpack import fft, ifft

@cython.cdivision(True)
cdef double bspline_cython(int p, int j, double x):
        """Return the value at x in [0,1[ of the B-spline with
        integer nodes of degree p with support starting at j.
        Implemented recursively using the de Boor's recursion formula"""
        cdef double w, w1
        if p == 0:
            if j == 0:
                return 1.0
            else:
                return 0.0
        else:
            w = (x - j) / p
            w1 = (x - j - 1) / p
        return w * bspline_cython(p-1,j,x)+(1-w1)*bspline_cython(p-1,j+1,x)

class BSplineCython:

    def __init__(self, p, xmin, xmax, ncells):
        self.p = p
        self.ncells = ncells
        # compute eigenvalues of degree p b-spline matrix
        self.modes = 2 * np.pi * np.arange(ncells) / ncells
        self.deltax = (xmax - xmin) / ncells

        self.eig_bspl = bspline_cython(p,-(p+1)//2, 0.0)
        for j in range(1, (p + 1) // 2):
            self.eig_bspl += bspline_cython(p,j-(p+1)//2,0.0)*2*np.cos(j*self.modes)
```

```
                self.eigalpha = np.zeros(ncells, dtype=complex)

            @cython.boundscheck(False)
            @cython.wraparound(False)
            def interpolate_disp(self,  f,  alpha):
                """compute the interpolating spline of degree p of odd degree
                of a function f on a periodic uniform mesh, at
                all points xi-alpha"""
                cdef Py_ssize_t j
                cdef int p = self.p
                # compute eigenvalues of cubic splines evaluated at displaced points
                cdef int ishift = np.floor(-alpha / self.deltax)
                cdef double beta = -ishift - alpha / self.deltax
                self.eigalpha.fill(0)
                for j in range(-(p-1)//2, (p+1)//2+1):
                    self.eigalpha += bspline_cython(p,j-(p+1)//2,beta)*np.exp((ishift+j)*1j*self.modes)

                # compute interpolating spline using fft and properties of circulant matrices
                return np.real(ifft(fft(f) * self.eigalpha / self.eig_bspl))
```

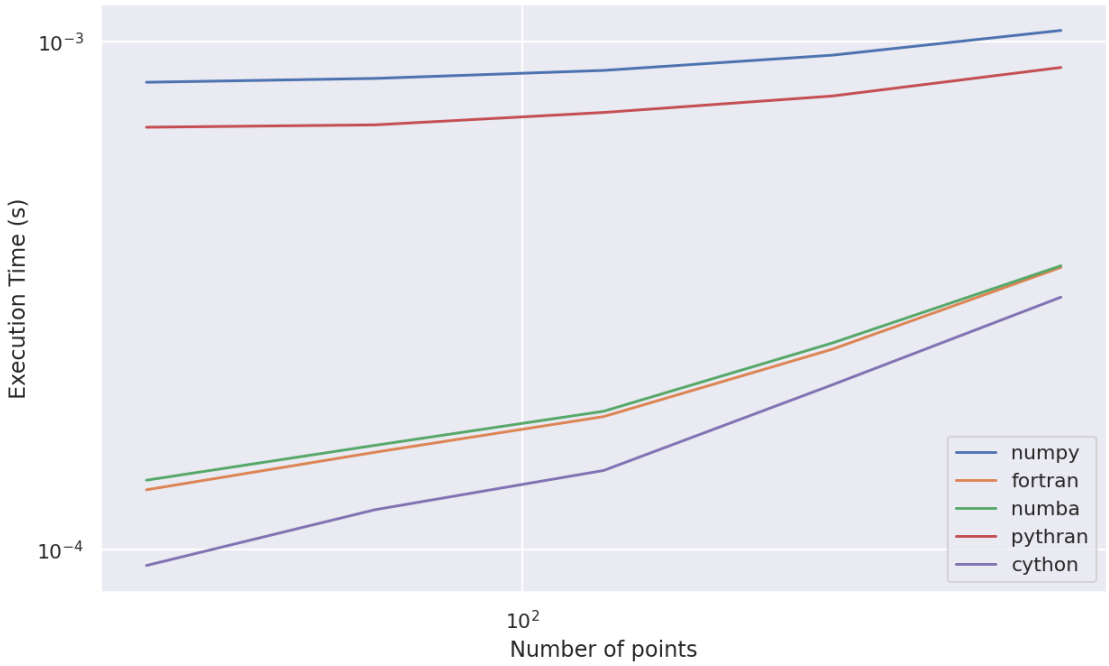
In [24]:
```
interpolation_test(BSplineCython)
```

Out[24]: True

In [25]:
```
import seaborn; seaborn.set()
from tqdm.notebook import tqdm
Mrange = (2 ** np.arange(5, 10)).astype(int)

t_numpy = []
t_fortran = []
t_numba = []
t_pythran = []
t_cython = []

for M in tqdm(Mrange):
    x = np.linspace(0,1,M, endpoint=False)
    f = np.sin(x*4*np.pi)
    cs1 = BSplineNumpy(5,0,1,M)
    cs2 = BSplineFortran(5,0,1,M)
    cs3 = BSplineNumba(5,0,1,M)
    cs4 = BSplinePythran(5,0,1,M)
    cs5 = BSplineCython(5,0,1,M)

    alpha = 0.1
    t1 = %timeit -oq cs1.interpolate_disp(f, alpha)
    t2 = %timeit -oq cs2.interpolate_disp(f, alpha)
    t3 = %timeit -oq cs3.interpolate_disp(f, alpha)
    t4 = %timeit -oq cs4.interpolate_disp(f, alpha)
    t5 = %timeit -oq cs5.interpolate_disp(f, alpha)

    t_numpy.append(t1.best)
    t_fortran.append(t2.best)
    t_numba.append(t3.best)
    t_pythran.append(t4.best)
    t_cython.append(t5.best)

plt.loglog(Mrange, t_numpy, label='numpy')
plt.loglog(Mrange, t_fortran, label='fortran')
plt.loglog(Mrange, t_numba, label='numba')
plt.loglog(Mrange, t_pythran, label='pythran')
```

```
        plt.loglog(Mrange, t_cython, label='cython')
        plt.legend(loc='lower right')
        plt.xlabel('Number of points')
        plt.ylabel('Execution Time (s)');
```

```
HBox(children=(FloatProgress(value=0.0, max=5.0), HTML(value='')))
```



19.2 Vlasov-Poisson equation

We consider the dimensionless Vlasov-Poisson equation for one species with a neutralizing background.

$$\frac{\partial f}{\partial t} + v \cdot \nabla_x f + E(t,x) \cdot \nabla_v f = 0, \; -\Delta\phi = 1 - \rho, E = -\nabla\phi\rho(t,x) = \int f(t,x,v)dv.$$

- Vlasov Equation - Wikipedia

```
In [26]: BSpline = dict(numpy=BSplineNumpy,
                        fortran=BSplineFortran,
                        cython=BSplineCython,
                        numba=BSplineNumba,
                        pythran=BSplinePythran)
```

```
        class VlasovPoisson:

            def __init__(self, xmin, xmax, nx, vmin, vmax, nv, opt='numpy'):

                # Grid
```

```
        self.nx = nx
        self.x, self.dx = np.linspace(xmin, xmax, nx, endpoint=False, retstep=True)
        self.nv = nv
        self.v, self.dv = np.linspace(vmin, vmax, nv, endpoint=False, retstep=True)

        # Distribution function
        self.f = np.zeros((nx,nv))

        # Interpolators for advection
        BSplineClass = BSpline[opt]
        self.cs_x = BSplineClass(3, xmin, xmax, nx)
        self.cs_v = BSplineClass(3, vmin, vmax, nv)

        # Modes for Poisson equation
        self.modes = np.zeros(nx)
        k =  2* np.pi / (xmax - xmin)
        self.modes[:nx//2] = k * np.arange(nx//2)
        self.modes[nx//2:] = - k * np.arange(nx//2,0,-1)
        self.modes += self.modes == 0 # avoid division by zero

    def advection_x(self, dt):
        for j in range(self.nv):
            alpha = dt * self.v[j]
            self.f[j,:] = self.cs_x.interpolate_disp(self.f[j,:], alpha)

    def advection_v(self, e, dt):
        for i in range(self.nx):
            alpha = dt * e[i]
            self.f[:,i] = self.cs_v.interpolate_disp(self.f[:,i], alpha)

    def compute_rho(self):
        rho = self.dv * np.sum(self.f, axis=0)
        return  rho - rho.mean()

    def compute_e(self, rho):
        # compute Ex using that ik*Ex = rho
        rhok = fft(rho)/self.modes
        return np.real(ifft(-1j*rhok))

    def run(self, f, nstep, dt):
        self.f = f
        nrj = []
        self.advection_x(0.5*dt)
        for istep in tqdm(range(nstep)):
            rho = self.compute_rho()
            e = self.compute_e(rho)
            self.advection_v(e, dt)
            self.advection_x(dt)
            nrj.append( 0.5*np.log(np.sum(e*e)*self.dx))

        return nrj
```

19.3 Landau Damping

Landau damping - Wikipedia

```
In [27]: from time import time
```

```
        elapsed_time = {}
        fig, axes = plt.subplots()
        for opt in ('numpy', 'fortran', 'numba', 'cython','pythran'):

            # Set grid
            nx, nv = 32, 64
            xmin, xmax = 0.0, 4*np.pi
            vmin, vmax = -6., 6.

            # Create Vlasov-Poisson simulation
            sim = VlasovPoisson(xmin, xmax, nx, vmin, vmax, nv, opt=opt)

            # Initialize distribution function
            X, V = np.meshgrid(sim.x, sim.v)
            eps, kx = 0.001, 0.5
            f = (1.0+eps*np.cos(kx*X))/np.sqrt(2.0*np.pi)* np.exp(-0.5*V*V)

            # Set time domain
            nstep = 600
            t, dt = np.linspace(0.0, 60.0, nstep, retstep=True)

            # Run simulation
            etime = time()
            nrj = sim.run(f, nstep, dt)
            print(" {0:12s} : {1:.4f} ".format(opt, time()-etime))

            # Plot energy
            axes.plot(t, nrj, label=opt)


        axes.plot(t, -0.1533*t-5.50)
        plt.legend();
```

```
HBox(children=(FloatProgress(value=0.0, max=600.0), HTML(value='')))
```

```
 numpy        : 15.6995
```

```
HBox(children=(FloatProgress(value=0.0, max=600.0), HTML(value='')))
```

```
 fortran      : 7.3522
```

```
HBox(children=(FloatProgress(value=0.0, max=600.0), HTML(value='')))
```

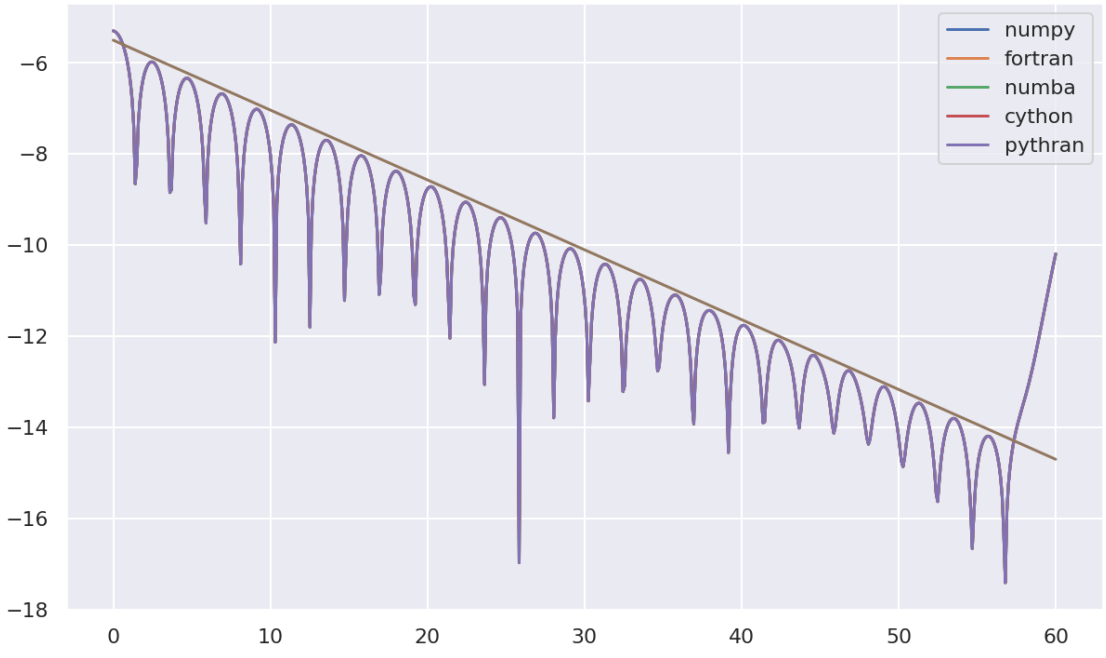
```
 numba        : 7.9384
```

```
HBox(children=(FloatProgress(value=0.0, max=600.0), HTML(value='')))
```

```
 cython       : 4.8745
```

```
HBox(children=(FloatProgress(value=0.0, max=600.0), HTML(value='')))
```

```
pythran      : 12.6075
```



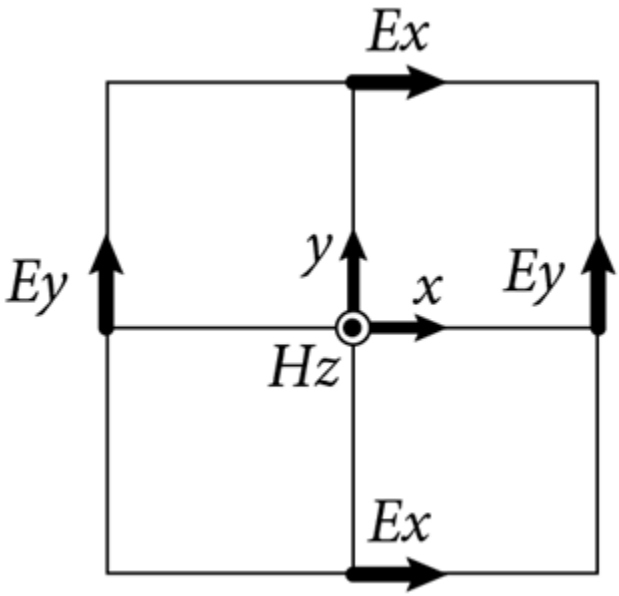
19.4 References

- Optimizing Python with NumPy and Numba

Chapter 20

Maxwell solver in two dimensions with FDTD scheme

$$\frac{\partial H_z}{\partial t} = \frac{\partial E_x}{\partial y} - \frac{\partial E_y}{\partial x}; \qquad \frac{\partial E_x}{\partial t} = \frac{\partial H_z}{\partial y}; \qquad \frac{\partial E_y}{\partial t} = -\frac{\partial H_z}{\partial x}$$



$$H_z\big|^{n+1/2}_{i+1/2,j+1/2} = H_z\big|^{n-1/2}_{i+1/2,j+1/2} + \frac{dt}{dy}\left(E_x\big|^{n}_{i+1/2,j+1} - E_x\big|^{n}_{i+1/2,j}\right) - \frac{dt}{dx}\left(E_y\big|^{n}_{i+1,j+1/2} - E_y\big|^{n}_{i,j+1/2}\right)$$

$$E_x\big|^{n+1}_{i+1/2,j} = E_x\big|^{n}_{i+1/2,j} + \frac{dt}{dy}\left(H_z\big|^{n+1/2}_{i+1/2,j+1/2} - H_z\big|^{n+1/2}_{i-1/2,j-1/2}\right)$$

$$E_y\big|^{n+1}_{i,j+1/2} = E_y\big|^{n}_{i,j+1/2} - \frac{dt}{dx}\left(H_z\big|^{n+1/2}_{i+1/2,j+1/2} - H_z\big|^{n+1/2}_{i-1/2,j+1/2}\right)$$

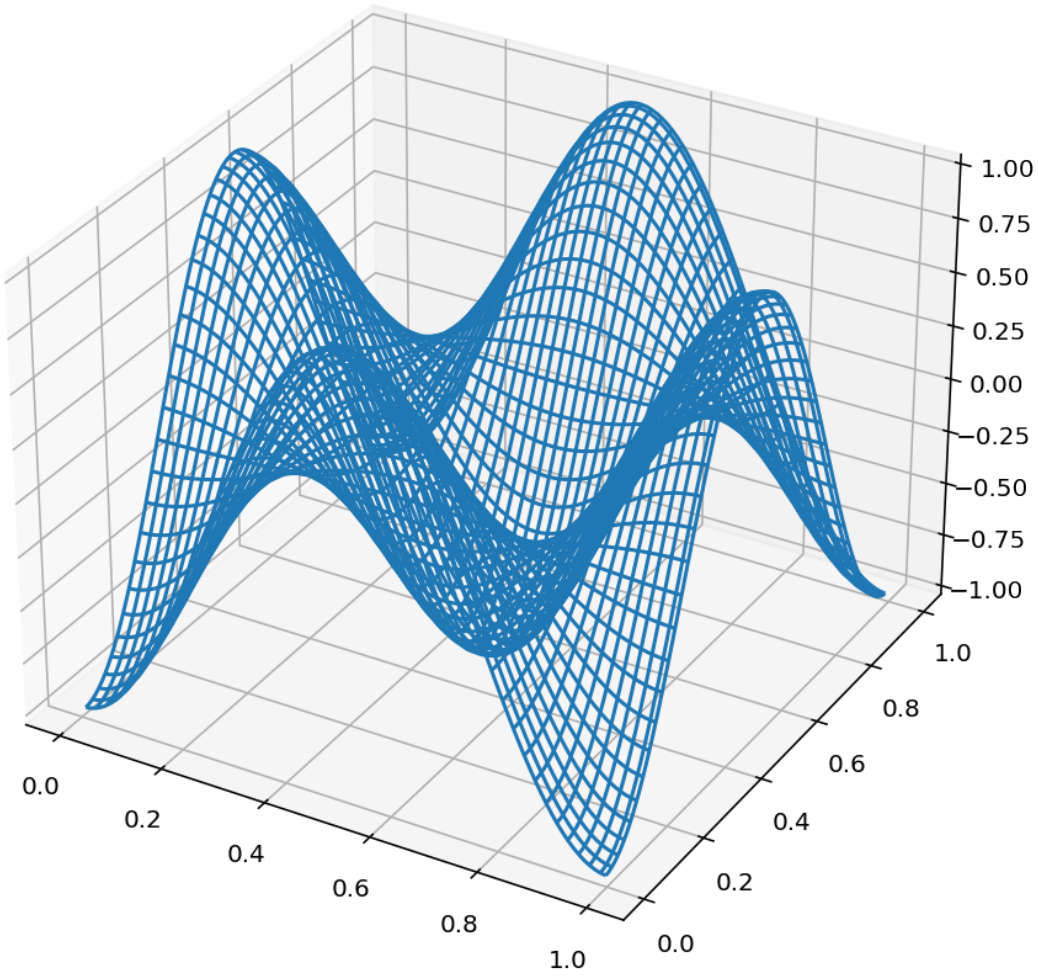
Description of the scheme

In [1]:
```
%matplotlib inline
%config InlineBackend.figure_format = 'retina'

import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import axes3d
```

```
import matplotlib.animation as animation
from IPython.display import HTML

plt.rcParams['figure.figsize'] = (10,6)
```

In [2]:
```
# Mesh parameters
nx, ny = 101, 101
vx, dx = np.linspace(0, 1, nx, endpoint=True, retstep=True)
vy, dy = np.linspace(0, 1, ny, endpoint=True, retstep=True)

#Initialize Ex, Ey when time = 0
ex = np.zeros((nx-1, ny), dtype=np.double)
ey = np.zeros((nx, ny-1), dtype=np.double)
nbiter = 500   # time loop size
dt = 0.001     # time step
m, n = 2, 2
omega = np.sqrt((m*np.pi)**2+(n*np.pi)**2)
# Create the staggered grid for Bz
x, y = np.meshgrid(0.5*(vx[:-1]+vx[1:]), 0.5*(vy[:-1]+vy[1:]))
```

In [3]:
```
fig = plt.figure()
ax = axes3d.Axes3D(fig)

#Initialize Bz when time = - dt / 2
hz = - np.cos(m*np.pi*y) * np.cos(n*np.pi*x) * np.cos(omega*(-0.5*dt))
wframe = ax.plot_wireframe(x, y, hz, rstride=2, cstride=2)
ax.set_zlim(-1,1);
```



20.1 numpy

```
In [4]: def faraday( ex, ey, hz ) :
            "faraday equation Bz(t+dt/2) -> Bz(t-dt/2) + dt f(E(t))"
            return hz + dt * ((ex[:, 1:]-ex[:, :-1]) / dy - (ey[1:, :]-ey[:-1, :]) / dx)

        def ampere_maxwell( hz, ex, ey):
            " Ampere-Maxwell equation E(t+dt) -> E(t) + dt g(Bz(t+dt/2)) "
            ex[:, 1:-1] += dt*(hz[:, 1:]-hz[:, :-1]) / dy
            ey[1:-1, :] += - dt*(hz[1:, :]-hz[:-1, :]) / dx

            # periodic boundary conditions
            ex[:, 0] += dt*(hz[:, 0]-hz[:, -1]) / dy
            ex[:, -1] = ex[:, 0]
            ey[0, :] += - dt*(hz[0, :]-hz[-1, :]) / dx
            ey[-1, :] = ey[0, :]
```

```
            return ex, ey
```

```
In [5]: def update(i, ax, fig):
            ax.cla()

            global ex, ey, hz

            hz = faraday( ex, ey, hz)
            ex, ey = ampere_maxwell( hz, ex, ey)

            wframe = ax.plot_wireframe(x, y, hz, rstride=2, cstride=2)
            ax.set_zlim(-1, 1)
            return wframe,
```

```
In [6]: ani = animation.FuncAnimation(fig, update,
                                       frames=range(200),
                                       fargs=(ax, fig), interval=100)
```

```
In [7]: %%time
        HTML(ani.to_html5_video())
```

```
CPU times: user 27.9 s, sys: 669 ms, total: 28.6 s
Wall time: 29 s
```

```
Out[7]: <IPython.core.display.HTML object>
```

```
In [8]: %%time

        from tqdm.notebook import tqdm

        nx, ny = 512, 512
        vx, dx = np.linspace(0, 1, nx, endpoint=True, retstep=True)
        vy, dy = np.linspace(0, 1, ny, endpoint=True, retstep=True)

        ex = np.zeros((nx-1, ny), dtype=np.double)
        ey = np.zeros((nx, ny-1), dtype=np.double)
        dt = 0.001     # time step
        m, n = 2, 2
        omega = np.sqrt((m*np.pi)**2+(n*np.pi)**2)
        x, y = np.meshgrid(0.5*(vx[:-1]+vx[1:]), 0.5*(vy[:-1]+vy[1:]))

        hz = - np.cos(m*np.pi*y) * np.cos(n*np.pi*x) * np.cos(omega*(-0.5*dt))

        for t in tqdm(range(1000)):

            hz = faraday( ex, ey, hz)
            ex, ey = ampere_maxwell( hz, ex, ey)
```

```
HBox(children=(FloatProgress(value=0.0, max=1000.0), HTML(value='')))
```

```
CPU times: user 6.13 s, sys: 32.1 ms, total: 6.17 s
Wall time: 6.13 s
```

```
In [9]: %load_ext fortranmagic
```

20.2 fortran

In [10]: %%fortran

```
subroutine faraday_fortran( ex, ey, bz, dx, dy, dt, nx, ny)
implicit none

real(8), intent(in) :: ex(nx-1,ny)
real(8), intent(in) :: ey(nx,ny-1)
real(8), intent(inout) :: bz(nx-1,ny-1)
integer, intent(in) :: nx, ny
real(8), intent(in) :: dx, dy, dt

integer :: i, j
real(8) :: dex_dx, dey_dy
real(8) :: dex_dy, dey_dx

do j=1,ny-1
do i=1,nx-1
   dex_dy  = (ex(i,j+1)-ex(i,j)) / dy
   dey_dx  = (ey(i+1,j)-ey(i,j)) / dx
   bz(i,j) = bz(i,j) +  dt * (dex_dy - dey_dx)
end do
end do

end subroutine faraday_fortran
```

In [11]: %%fortran

```
subroutine amperemaxwell_fortran(ex, ey, bz, dx, dy, dt, nx, ny)

    implicit none
    integer, intent(in):: nx, ny
    real(8), intent(in):: dx, dy, dt
    real(8), dimension(nx-1, ny-1), intent(inout) :: bz
    real(8), dimension(nx-1, ny),   intent(inout) :: ex
    real(8), dimension(nx, ny-1),   intent(inout) :: ey
    integer:: i, j
    real(8):: dbz_dx, dbz_dy
    real(8), parameter:: csq = 1d0

    do i = 1, nx-1
       dbz_dy = (bz(i, 1)-bz(i, ny-1)) / dy ! periodic BC
       ex(i, 1) = ex(i, 1) + dt*csq*dbz_dy
       ex(i, ny) = ex(i, 1)
    end do

    do j = 1, ny-1
       dbz_dx = (bz(1,j)-bz(nx-1,j)) / dx  ! periodic BC
       ey(1,j) = ey(1,j) - dt*csq*dbz_dx
       ey(nx,j) = ey(1,j)
    end do

    do j=2,ny-1
        do i=1,nx-1
           dbz_dy = (bz(i,j)-bz(i,j-1)) / dy
           ex(i,j) = ex(i,j) + dt*csq*dbz_dy
        end do
    end do
```

```
            do j=1,ny-1
                do i=2,nx-1
                   dbz_dx = (bz(i,j)-bz(i-1,j)) / dx
                   ey(i,j) = ey(i,j) - dt*csq*dbz_dx
                end do
            end do

        end subroutine amperemaxwell_fortran
```

In [12]:
```
%%time

from tqdm.notebook import tqdm

ex.fill(0.0)
ey.fill(0.0)
hz = - np.cos(m*np.pi*y) * np.cos(n*np.pi*x) * np.cos(omega*(-0.5*dt))
ex = np.asfortranarray(ex)
ey = np.asfortranarray(ey)
hz = np.asfortranarray(hz)

for t in tqdm(range(1000)):

    faraday_fortran( ex, ey, hz, dx, dy, dt, nx, ny)
    amperemaxwell_fortran(ex, ey, hz, dx, dy, dt, nx, ny)
```

```
HBox(children=(FloatProgress(value=0.0, max=1000.0), HTML(value='')))


CPU times: user 2.96 s, sys: 20 ms, total: 2.98 s
Wall time: 2.96 s
```

Chapter 21

Gray-Scott Model

```
In [1]: import numpy as np
```

```
In [2]: %config InlineBackend.figure_format = 'retina'
```

The reaction-diffusion system described here involves two generic chemical species U and V, whose concentration at a given point in space is referred to by variables u and v. As the term implies, they react with each other, and they diffuse through the medium. Therefore the concentration of U and V at any given location changes with time and can differ from that at other locations.

The overall behavior of the system is described by the following formula, two equations which describe three sources of increase and decrease for each of the two chemicals:

$$\frac{\partial u}{\partial t} = D_u \Delta u - uv^2 + F(1-u)$$
$$\frac{\partial v}{\partial t} = D_v \Delta v + uv^2 - (F+k)v$$

The laplacian is computed with the following numerical scheme

$$\Delta u_{i,j} \approx u_{i,j-1} + u_{i-1,j} - 4u_{i,j} + u_{i+1,j} + u_{i,j+1}$$

The classic Euler scheme is used to integrate the time derivative.

21.1 Initialization

u is 1 everywhere et v is 0 in the domain except in a square zone where $v = 0.25$ and $ u = 0.5$. This square located in the center of the domain is $[0, 1] \times [0, 1]$ with a size of 0.2.

```
In [3]: def init(n):

            u = np.ones((n+2,n+2))
            v = np.zeros((n+2,n+2))

            x, y = np.meshgrid(np.linspace(0, 1, n+2), np.linspace(0, 1, n+2))

            mask = (0.4<x) & (x<0.6) & (0.4<y) & (y<0.6)

            u[mask] = 0.50
            v[mask] = 0.25

            return u, v
```

21.2 Boundary conditions

We assume that the domain is periodic.

```
In [4]: def periodic_bc(u):
            u[0, :] = u[-2, :]
            u[-1, :] = u[1, :]
            u[:, 0] = u[:, -2]
            u[:, -1] = u[:, 1]
```

21.3 Laplacian

```
In [5]: def laplacian(u):
            """
            second order finite differences
            """
            return (                  u[ :-2, 1:-1] +
                    u[1:-1, :-2] - 4*u[1:-1, 1:-1] + u[1:-1, 2:] +
                                +   u[2:  , 1:-1] )
```

21.4 Gray-Scott model

```
In [6]: def grayscott(U, V, Du, Dv, F, k):

            u, v = U[1:-1,1:-1], V[1:-1,1:-1]

            Lu = laplacian(U)
            Lv = laplacian(V)

            uvv = u*v*v
            u += Du*Lu - uvv + F*(1 - u)
            v += Dv*Lv + uvv - (F + k)*v

            periodic_bc(U)
            periodic_bc(V)
```

21.5 Visualization

Nous utiliserons les données suivantes.

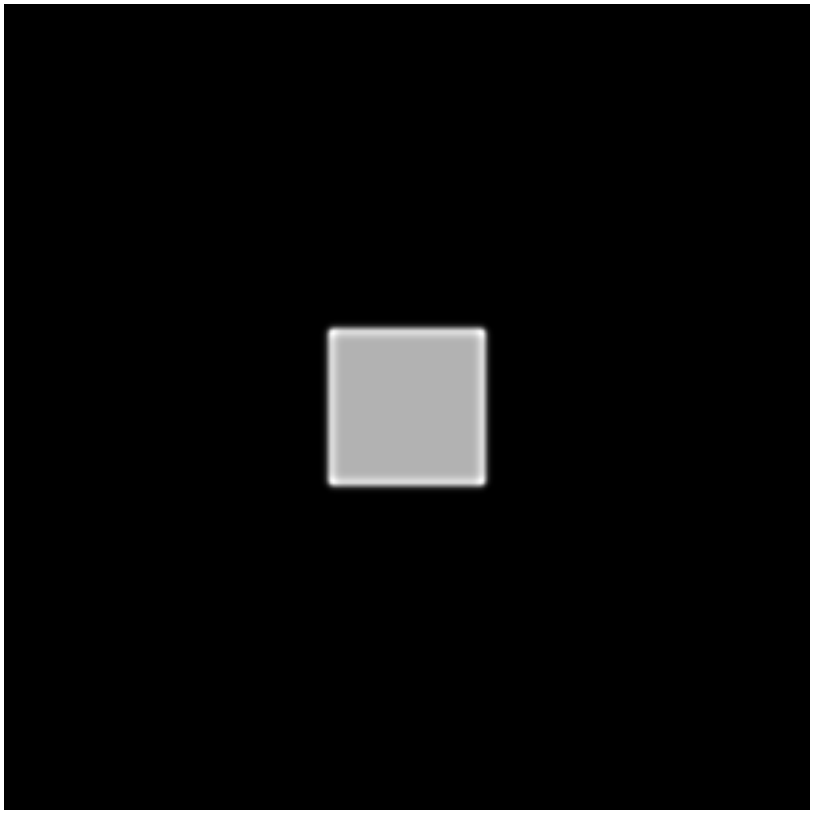
```
In [7]: Du, Dv = .1, .05
        F, k = 0.0545, 0.062
```

```
In [8]: %%time
        from tqdm.notebook import tqdm
        from PIL import Image
        U, V = init(300)

        def create_image():
            global U, V
            for t in range(40):
                grayscott(U, V, Du, Dv, F, k)
            V_scaled = np.uint8(255*(V-V.min()) / (V.max()-V.min()))
            return V_scaled

        def create_frames(n):
```

```
            return [create_image() for i in tqdm(range(n))]

        frames = create_frames(500)
```

```
HBox(children=(FloatProgress(value=0.0, max=500.0), HTML(value='')))


CPU times: user 1min 7s, sys: 25.3 s, total: 1min 33s
Wall time: 1min 32s
```

```
In [9]: from ipywidgets import interact, IntSlider

        def display_sequence(iframe):

            return Image.fromarray(frames[iframe])

        interact(display_sequence,
                 iframe=IntSlider(min=0,
                                  max=len(frames)-1,
                                  step=1,
                                  value=0,
                                  continuous_update=True))
```



```
Out[9]: <function __main__.display_sequence(iframe)>
```

```
In [10]: import imageio
         frames_scaled = [np.uint8(255 * frame) for frame in frames]
         imageio.mimsave('images/movie.gif', frames_scaled, format='gif', fps=60)

In [11]: from IPython.display import HTML
         HTML('<img src="images/movie.gif">')

Out[11]: <IPython.core.display.HTML object>
```

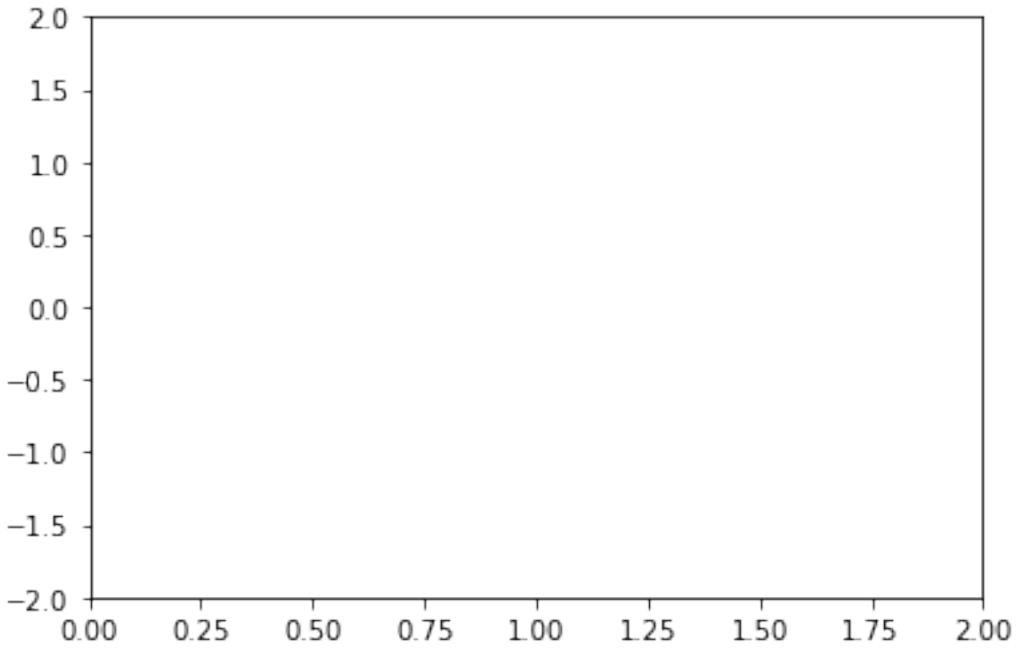
21.6 References

- Reaction-Diffusion by the Gray-Scott Model: Pearson's Parametrization

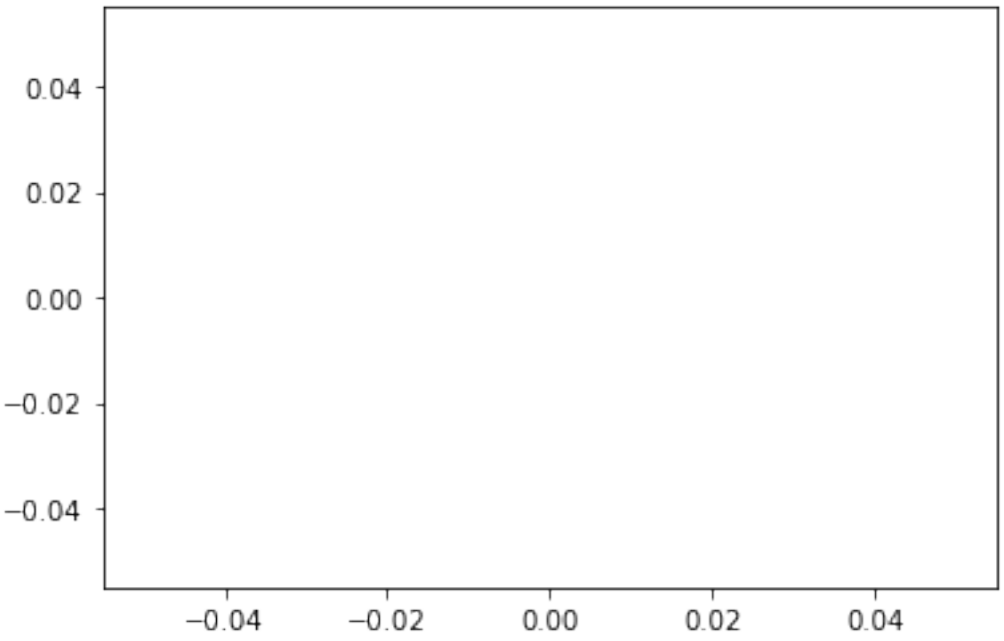
Chapter 22

Animation with matplotlib

```
In [1]: %matplotlib inline

In [2]: import numpy as np
        import matplotlib.pyplot as plt

        from matplotlib import animation, rc
        from IPython.display import HTML
        from matplotlib.animation import FuncAnimation

In [3]: fig, ax = plt.subplots()

        ax.set_xlim(( 0, 2))
        ax.set_ylim((-2, 2))

        line, = ax.plot([], [], lw=2)
```



```
In [4]: def init():
            line.set_data([], [])
            return (line,)

In [5]: 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,)

In [6]: anim = animation.FuncAnimation(fig, animate, init_func=init,
                                       frames=100, interval=20,
                                       blit=True)

In [7]: HTML(anim.to_html5_video())

Out[7]: <IPython.core.display.HTML object>

In [8]: HTML(anim.to_jshtml())

Out[8]: <IPython.core.display.HTML object>

In [9]: fig = plt.figure()
        ax1 = fig.add_subplot(1,1,1)
        xdata, ydata = [], []
        line1, = plt.plot([], [], 'r-', animated=True)
```




```
In [10]: def init():
             ax1.set_xlim((0,1))
             ax1.set_ylim((-1,1))
             return line1,

         def update(frame):
```

```
            xdata.append(frame)
            ydata.append(np.sin(8*np.pi*frame))
            line1.set_data(xdata, ydata)

            return line1,

        ani = FuncAnimation(fig, update, frames=np.linspace(0, 1.0, 100),
                            init_func=init, blit=True)

        plt.rc('animation', html='html5')
        ani
```

Out[10]: <matplotlib.animation.FuncAnimation at 0x7f5c426730d0>

In [ ]: