Scientific programming in Python

Outline of course

The aims for this course are

- Learn the basics of Python
- Learn the basics of C
- Prepare you for research projects.

Format of course

The course consists of

- **Lectures**.
- You will solve **Exercises** in the classroom.
- You will do **tutorials** (i.e. run/understand programs that I have made).

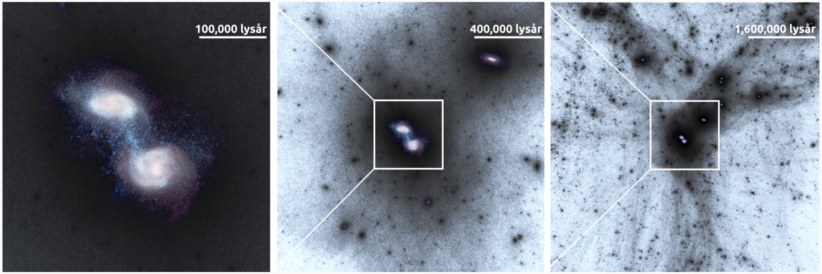
Introduction

Why do we need to learn a programming language?

- It makes it easy (or at least possible) to work on large datasets.
- It can be used to produce publishable plots and figures.

Example 1: Visualisation of numerical simulations

Galaxy mergers in a hydrodynamical computer simulation



Example 2: Spectroscopical observations

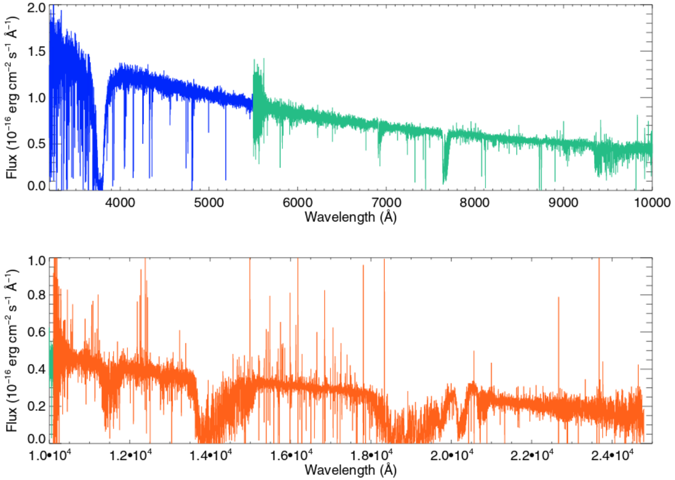


Fig. 1. The flux-calibrated X-shooter spectrum of GRB 090926A. *Top*: ultraviolet/blue and visual arms. *Bottom*: near-infrared arm. The absolute flux calibration may be not accurate due to slit losses.

Why Python?

- Python is a free, *easy-to-use* programming language.
- Many software-packages available.
- Supported by the astrophysics community (and other communities as well, e.g. particle physics)

Python is simpler than C!

In C:

```
#include <stdio.h>
int main()
{
 printf(”Hello, World!”);
 return 0;
}
```

In Python:

```
print(”Hello, World!”)
```

It is faster to write Python than C

Packages for Python

- **NumPy**: A package to work with arrays and matrices. It includes modules for Fourier transforms, linear algebra, random number generation, etc.
- **Matplotlib**: A package for plotting.
- **astropy**: astronomy/astrophysics related computations.
- **TensorFlow and PyTorch**: Machine Learning and Neural Networks.
- many more...

How to write and run pythoncode

How to write and run pythoncode

- Run python in shell/terminal.
- Notebooks (e.g. Jupyter).
- Microsoft Visual Studio is good (copilot).
- Docker.

If you would like to specialize in numerical astrophysics you should be able to work with all such tools... It is important that you learn to run python in a shell/terminal.

Shell/terminal (my preferred way)

- Run python from shell.
- Use a text editor to edit code (for example Atom or vi).

IPython is useful when working in shell

- IPython is an interactive Python shell/terminal. `https://ipython.org/`
- Open it by typing ipython in the terminal.
- In IPython (and when using the terminal) the TAB key is your friend:

Jupyter

- Works in a browser (that can both be good or bad)
- Works equally well on all platforms.

Some examples

Example: Conversion of density units

And it is also to convert between density units:

```
import astropy.units as u
Density = 2 * u.Msun / u.kpc**3
print(Density.to(u.g/u.cm**3))
```

Output:

```
1.35362568e−31 g / cm3
```

... such convenient libraries reduce the number of mistakes/bugs in programs!

Example: astropy.units

astropy.units can also convert between units. E.g. from kpc to cm:

```
import astropy.units as u
Length = 32.0*u.kpc
print(Length.to(u.cm))
```

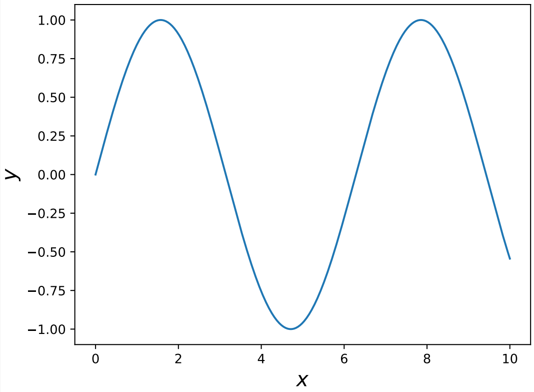
Output:

```
9.874168260695014e+22 cm
```

Matplotlib

A program that plots $\sin(x)$

```
import matplotlib.pyplot as plt
import numpy

x = numpy.linspace(0.0,10.0,1000)
plt.plot(x,numpy.sin(x),'–')
plt.xlabel('$x$',fontsize=16)
plt.ylabel('$y$',fontsize=16)
plt.show()
```



It is a good idea to become familiar with numpy and matplotlib! It is the aim of the next couple of weeks of this course.

Arithmetic operations, strings, lists

Arithmetic operations $+ - \times /$

Program:

```
a = 5
b = 3
c=a+b
print(c)
c=a-b
print(c)
c=a*b
print(c)
c=a/b
print(c)
```

Output:

```
8
2
15
1.6666666666666667
```

The $**$ operator

- To calculate x^3 in Python we write x**3.
- do not type $x^\wedge 3$, it does something else (a bitwise xor, which you will most likely never need)

The // operator

- // is used for integer division. For example 2//3 is equal to 0.

Note, the division operator in Python3 differs from Python2, C, and essentially all other programming languages.

Strings

Program:

```
a = 'John'
b = "Deere"
c=a+b
print(c)
c=a+' '+b
print(c)
```

Output:

```
JohnDeere
John Deere
```

Strings and numbers

Program:

```
a = ′John′
b = ″Deere″
N = 6215
c=a+′ ′+b+′ ′+str(N)
print(c)
```

Output:

```
John Deere 6215
```

Lists I

Program:

```
MyList = [0,1,2,3,4,5]
print(MyList*2)
```

Output:

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

Lists II

In lists you can have elements of different types. E.g. integers and text strings:

Program:

```
MyList = [0,1,2,3,4,5,'This is a string']
print(MyList)
```

Output:

```
[0, 1, 2, 3, 4, 5, 'This is a string']
```

Lists III. The append function.

Program:

```
MyList = [0,1,2,3,4,5,’This is a string’]
MyList.append(6)
print(MyList)
```

Output:

```
[0, 1, 2, 3, 4, 5, ’This is a string’, 6.0]
```

Loops over lists

Program:

```
MyList = [0,1,2,3,4,5,'This is a string']
for item in MyList:
        print(item)
```

Output:

```
0
1
2
3
4
5
This is a string
```

comparison operators

$==$	equal
$!=$	not equal
$<$	smaller than
$>$	greater than
$<=$	smaller than or equal
$>=$	greater than or equal

Note that $=$ and $==$ work completely differents

See more on `https://www.tutorialspoint.com/python3/comparison_operators_example.htm`.

if statements

```
MyList = [0,1.0,2,'This is a string']
for item in MyList:
    if type(item)==str:
        print('This is a string')
    elif type(item)==float:
        print('This is a float')
    elif type(item)==int:
        print('This is an integer')
```

Output:

```
This is an integer
This is a float
This is an integer
This is a string
```

It is tutorial time!

Go through the tutorial Learn the Basics on `https://www.learnpython.org` You can do this in a browser.

Plotting with Python – Matplotlib

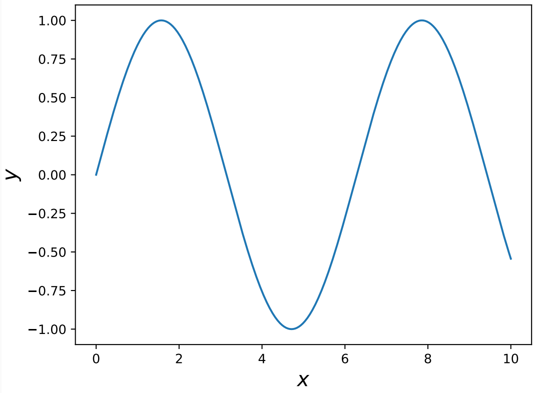
Aim for this lecture

- This will enable you to do plots of publishing quality (for a MSc thesis or publications).
- Go through examples (partially based on `https://matplotlib.org/stable/tutorials/pyplot.html`).

Matplotlib (example from last week)

A program that plots $\sin(x)$

```
import matplotlib.pyplot as plt
import numpy

x = numpy.linspace(0.0,10.0,1000)
plt.plot(x,numpy.sin(x),'−')
plt.xlabel('$x$',fontsize=16)
plt.ylabel('$y$',fontsize=16)
plt.show()
```



A slightly more advanced example

We now also set a plot title, xlimits and ylimits

```
import matplotlib.pyplot as plt
import numpy

x = numpy.linspace(0.0,4*numpy.pi,1000)

plt.plot(x,numpy.sin(x),'-',color='red',lw=1)
plt.plot(x,numpy.cos(x),'--',color='blue',lw=2)
plt.plot(x,numpy.cos(x+numpy.pi/4.0),':',color='black',lw=2)

plt.xlabel('$x$',fontsize=16)
plt.ylabel('$y$',fontsize=16)
plt.title('Plot of function')
plt.xlim((0.0,4*numpy.pi))
plt.ylim((-1.05,1.05))
plt.show()
```

Legends

We add a legend

```
import matplotlib.pyplot as plt
import numpy

x = numpy.linspace(0.0,4*numpy.pi,1000)

plt.plot(x,numpy.sin(x),'-',color='red',label='sin(x)')
plt.plot(x,numpy.cos(x),'--',color='blue',label='cos(x)')
plt.plot(x,numpy.cos(x+numpy.pi/4.0),':',color='black',label=r'$cos(x+\pi/4.0)$')

plt.xlabel('$x$',fontsize=16)
plt.ylabel('$y$',fontsize=16)
plt.title('Plot of function')
plt.xlim((0.0,4*numpy.pi))
plt.ylim((-1.05,1.05))
plt.legend()
plt.show()
```

Legends II

An even nicer legend!

```
import matplotlib.pyplot as plt
import numpy

x = numpy.linspace(0.0,4*numpy.pi,1000)

plt.plot(x,numpy.sin(x),'-',color='red',label='sin(x)')
plt.plot(x,numpy.cos(x),'--',color='blue',label='cos(x)')
plt.plot(x,numpy.cos(x+numpy.pi/4.0),':',color='black',label=r'$cos(x+\pi/4.0)$')

plt.xlabel('$x$',fontsize=16)
plt.ylabel('$y$',fontsize=16)
plt.title('Plot of function')
plt.xlim((0.0,4*numpy.pi))
plt.ylim((-1.05,1.05))

plt.legend(prop=dict(size=14), numpoints=1, ncol=1,frameon=True,loc=2,handlelength=2.0)
plt.show()
```

See documentation about plt.legend() on
`https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.legend.html`

Matplotlib documentation

The Matplotlib webpage is your friend! `https://matplotlib.org/stable/plot_types/index` and `https://matplotlib.org/stable/api/pyplot_summary.html`.

In the following we will go through a selection of Matplotlib commands.

plt.savefig

With plt.savefig we can save a plot to e.g. png and pdf.

```
import matplotlib.pyplot as plt
import numpy

x = numpy.linspace(0.0,4*numpy.pi,1000)

plt.plot(x,numpy.sin(x),'-',color='red',label='sin(x)')
plt.plot(x,numpy.cos(x),'--',color='blue',label='cos(x)')
plt.plot(x,numpy.cos(x+numpy.pi/4.0),':',color='black',label=r'$cos(x+\pi/4.0)$')

plt.xlabel('$x$',fontsize=16)
plt.ylabel('$y$',fontsize=16)
plt.title('Plot of function')
plt.xlim((0.0,4*numpy.pi))
plt.ylim((-1.05,1.05))

plt.legend(prop=dict(size=14), numpoints=1, ncol=1,frameon=True,loc=2,handlelength=2.0)
plt.savefig('OurPlot.png')
plt.savefig('OurPlot.pdf')
plt.show()
```

plt.subplot

Multipanel figures with plt.subplot

```
import matplotlib.pyplot as plt
import numpy

x = numpy.linspace(0.0,4*numpy.pi,1000)

plt.subplot(1,3,1)
plt.title('sin(x)')
plt.plot(x,numpy.sin(x),'-',color='red',label='sin(x)')
plt.subplot(1,3,2)
plt.title('cos(x)')
plt.plot(x,numpy.cos(x),'--',color='blue',label='cos(x)')
plt.subplot(1,3,3)
plt.title(r'$cos(x+\pi/4)$')
plt.plot(x,numpy.cos(x+numpy.pi/4.0),':',color='black',label=r'$cos(x+\pi/4.0)$')
plt.show()
```

Plotting of datapoints

Plotting of points

```
import matplotlib.pyplot as plt
import numpy

x = numpy.linspace(0.0,4*numpy.pi,25)

plt.subplot(1,3,1)
plt.title('sin(x)')
plt.plot(x,numpy.sin(x),'o',color='red',label='sin(x)')
plt.subplot(1,3,2)
plt.title('cos(x)')
plt.plot(x,numpy.cos(x),'x',color='blue',label='cos(x)')
plt.subplot(1,3,3)
plt.title(r'$cos(x+\pi/4)$')
plt.plot(x,numpy.cos(x+numpy.pi/4.0),'<',color='black',label=r'$cos(x+\pi/4.0)$')
plt.show()
```

See Markers on `https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.plot.html`.

plt.fill_between

Filling area between lines

```
import matplotlib.pyplot as plt
import numpy

x = numpy.linspace(0.0,4*numpy.pi,1000)

plt.plot(x,numpy.sin(x),'−',color='black',lw=3)
plt.plot(x,numpy.cos(x),'−−',color='grey',lw=3)
plt.fill_between(x,numpy.sin(x),y2=numpy.cos(x),color='red')
plt.show()
```

See `https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.fill_between.html`.

log-axes

With plt.loglog and plt.semilogx and semilogy we can make an axis logarithmic

```
import matplotlib.pyplot as plt
import numpy

x = numpy.linspace(1.0,10.0,1000)

plt.subplot(1,3,1)

plt.plot(x,numpy.exp(x),'−')
plt.semilogy()

plt.subplot(1,3,2)

plt.plot(x,numpy.exp(x),'−')
plt.loglog()

plt.subplot(1,3,3)

plt.plot(x,numpy.exp(x),'−')

plt.show()
```

Histograms (plt.hist)

We can create histograms with plt.hist

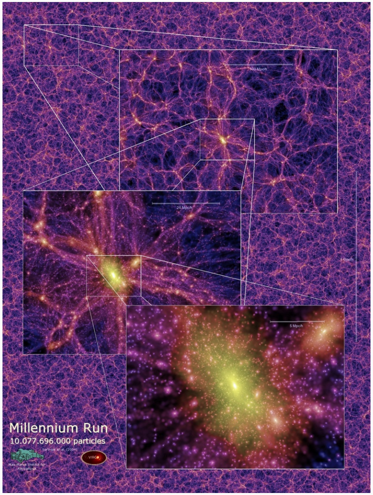
```
import matplotlib.pyplot as plt
import numpy

mu, sigma = 100, 15
x = mu + sigma * numpy.random.randn(10000)

plt.hist(x, 50, facecolor='green')
plt.show()
```

See `https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.hist.html`.

Random numbers and Numpy

Random numbers

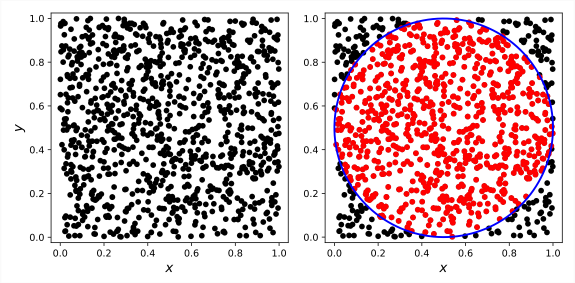
Example I: Cosmological structure simulations



Dark matter is modelled as collisionless particles sampled from an initial distribution, as observed in the early Universe.

Example II: Estimation of π

By randomly drawing points from a uniform distribution with $0 \le x < 1$ and $0 \le y < 1$, we can estimate π with a Monte Carlo method.



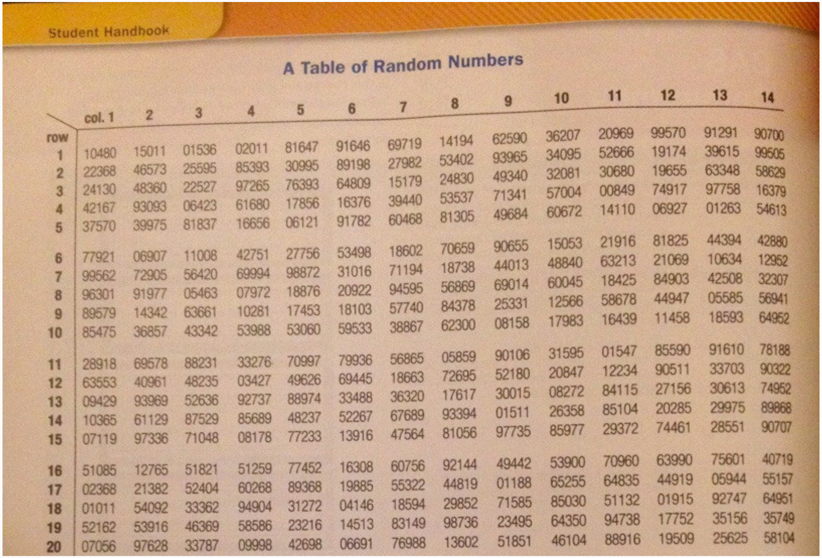
The ratio of area covered by the blue circle and a square containing all points is:

$$\frac{A_{\text{circle}}}{A_{\text{box}}} = \frac{\pi 0.5^2}{1^2}.$$

The points are equally distributed in the box, so $A_{\text{circle}}/A_{\text{box}} \approx N_{\text{circle}}/N_{\text{box}}$, so

$$\boxed{\pi \approx \frac{4A_{\text{circle}}}{A_{\text{box}}} = \frac{4N(\text{red})}{N(\text{red}) + N(\text{black})}}.$$

Schaum's Outline of Mathematical Handbook of Formulas and Tables



Student Handbook

A Table of Random Numbers

row	col. 1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	10480	15011	01536	02011	81647	91646	69719	14194	62590	36207	20969	99570	91291	90700
2	22368	46573	25595	85393	30995	89198	27982	53402	93965	34095	52666	19174	39615	99505
3	24130	48360	22527	97265	76393	64809	15179	24830	49340	32081	30680	19655	63348	58629
4	42167	93093	06423	61680	17856	16376	39440	53537	71341	57004	00849	74917	97758	16379
5	37570	39975	81837	16656	06121	91782	60468	81305	49684	60672	14110	06927	01263	54613
6	77921	06907	11008	42751	27756	53498	18602	70659	90655	15053	21916	81825	44394	42880
7	99562	72905	56420	69994	98872	31016	71194	18738	44013	48840	63213	21069	10634	12952
8	96301	91977	05463	07972	18876	20922	94595	56869	69014	60045	18425	84903	42508	32307
9	89579	14342	63661	10281	17453	18103	57740	84378	25331	12566	58678	44947	05585	56941
10	85475	36857	43342	53988	53060	59533	38867	62300	08158	17983	16439	11458	18593	64952
11	28918	69578	88231	33276	70997	79936	56865	05859	90106	31595	01547	85590	91610	78188
12	63553	40961	48235	03427	49626	69445	18663	72695	52180	20847	12234	90511	33703	90322
13	09429	93969	52636	92737	88974	33488	36320	17617	30015	08272	84115	27156	30613	74952
14	10365	61129	87529	85689	48237	52267	67689	93394	01511	26358	85104	20285	29975	89868
15	07119	97336	71048	08178	77233	13916	47564	81056	97735	85977	29372	74461	28551	90707
16	51085	12765	51821	51259	77452	16308	60756	92144	49442	53900	70960	63990	75601	40719
17	02368	21382	52404	60268	89368	19885	55322	44819	01188	65255	64835	44919	05944	55157
18	01011	54092	33362	94904	31272	04146	18594	29852	71585	85030	51132	01915	92747	64951
19	52162	53916	46369	58586	23216	14513	83149	98736	23495	64350	94738	17752	35156	35749
20	07056	97628	33787	09998	42698	06691	76988	13602	51851	46104	88916	19509	25625	58104

Random numbers in Python I

Program:

```
import random
print(random.random())
print(random.random())
print(random.random())
print(random.random())
print(random.random())
```

Output:

```
0.610668054627
0.91141768662
0.949963972579
0.0934526506271
0.669123796498
```

The random.random() function returns random floating point numbers between 0 and 1 ($0 \leq x < 1$).

Random numbers in Python II

To create 10000 random numbers we could simply write

```
import random
for i in range(10000):
        print(random.random())
```

The random number seed

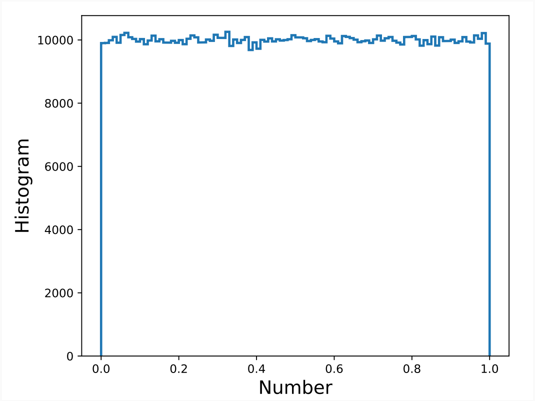
The random number seed can be set as follows:

```
import random
random.seed(1344)
for i in range(10000):
        print(random.random())
```

- When using a random number seed the program gives the same output every time you run it.
- If no random seed is specified, python calculates one based on the time.
- It is easier to debug programs, if a random seed is used (because errors are reproducible).

Random numbers

- It is impossible to create truly random numbers on a computer. Computers are deterministic!
- What we mean with random numbers, is a number sequence that follows a uniform distribution.



NumPy

Numpy

NumPy is the fundamental package for scientific computing with Python. It contains among other things:

- a powerful N-dimensional array object
- Fast functions (written in C)
- tools for integrating C/C++ and Fortran code
- useful linear algebra, Fourier transform, and random number capabilities

Numpy arrays I

```
import numpy
Array = numpy.array([1,2,3])
print(Array)
print(2*Array)
print(numpy.sin(2*Array))
print(numpy.exp(2*Array))
```

Output:

```
[1 2 3]
[2 4 6]
[ 0.90929743 −0.7568025 −0.2794155 ]
[ 7.3890561 54.59815003 403.42879349]
```

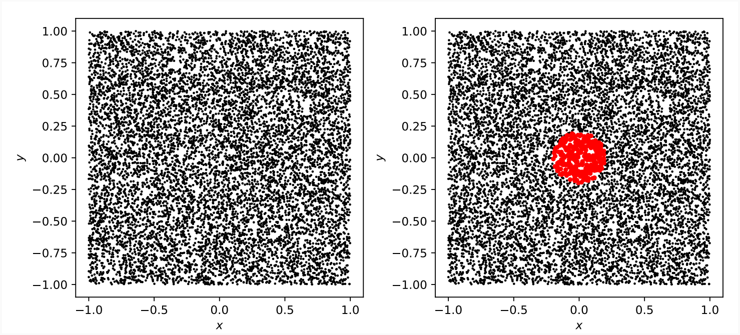
Numpy arrays II

- Calculations with numpy arrays are fast, because calculations are carried out with compiled C-code.
- Python lists are usually $\sim 20 - 50$ times slower.

Numpy arrays III: The numpy.where function

The **numpy.where** function can be used to select subsets of arrays:

```
x = numpy.random.random(10000)*2−1
y = numpy.random.random(10000)*2−1
r = numpy.sqrt(x**2+y**2)
x_central_points = x[numpy.where(r<0.2)]
y_central_points = y[numpy.where(r<0.2)]
```



Numpy arrays IIII: The numpy.where function

The **numpy.where** function can be used to select subsets of arrays:

```
x = numpy.random.random(10000)*2−1
y = numpy.random.random(10000)*2−1
r = numpy.sqrt(x**2+y**2)
print('Number of particles total:', x.shape[0] )
print('Number of particles with r<0.2:', x[numpy.where(r<0.2)].shape[0] )
```

Output:

```
Number of particles total: 10000
Number of particles with r<0.2: 318
```

Attributes of numpy arrays

Each numpy vector/matrix is technically a *Python-class* named **ndarray**. It has e.g. the following attributes:

- ndarray.**ndim**: the number of axes (dimensions) of the array.
- ndarray.**shape**: the dimensions of the array.
- ndarray.**size**: the total number of elements of the array.
- ndarray.**dtype**: an object describing the type of the elements in the array. e.g. numpy.int32, numpy.int16, and numpy.float64.
- ndarray.**itemsize**: the size in bytes of each element of the array.

Example: reshaping of arrays

We can reshape a vector with four entries to an array of size (2,2) with the following:

```
import numpy
A = numpy.array([1,2,3,4])
A.shape = (2,2)
```

Tutorials and exercise

- Go through `https://numpy.org/doc/stable/user/quickstart.html`
- Exercises about random numbers.

Dictionaries

Dictionaries

A dictionary is a flexible way to store and retrieve data. See for example:

```
StellarMass = {}
StellarMass['MilkyWay'] = 8.0e10
StellarMass['Andromeda'] = 1.0e11
StellarMass['M33'] = 5.0e10
StellarMass['M51'] = 6e10

GalaxyType = {}
GalaxyType['MilkyWay'] = 'Our home'
GalaxyType['Andromeda'] = 'Spiral'
GalaxyType['M33'] = 'Spiral'
GalaxyType['M51'] = 'Merger'

DistanceFromUs = {}
DistanceFromUs['MilkyWay'] = 8.0
DistanceFromUs['Andromeda'] = 1200.0
DistanceFromUs['M33'] = 840.0
DistanceFromUs['M51'] = 7100.0

Keys = StellarMass.keys()
for key in Keys:
    print('We now plot %s (%s), which has a stellar mass of %2.2g and is %f kpc away from us'
        %(key,GalaxyType[key],StellarMass[key],DistanceFromUs[key]))
```

See **DictExample.py**.

Data types in Numpy

What is a datatype?

```
import numpy as np
A = np.arange(10)
print(A)
print('The type of A is',A.dtype)
A = 1.0*A
print(A)
print('The type of A is now',A.dtype)
```

Output:

```
[0 1 2 3 4 5 6 7 8 9]
The type of A is int64
[0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
The type of A is now float64
```

By multiplying with a floating number, an array of 64 bit integers is automatically converted to 64 bit floats.

Data types in Numpy

- The most commonly used data types are numpy.**float64** and numpy.**int64**!

With dtype you can set the type of an array

example:

```
import numpy
A=numpy.arange(10,dtype=numpy.int32)
```

If dtype is not set, it will be a 64 bit int (which is normally good)

Integers (and np.iinfo)

```
import numpy as np
print(np.iinfo(np.int64))
print(np.iinfo(np.uint64))
```

Output:

```
Machine parameters for int64
min = −9223372036854775808
max = 9223372036854775807

Machine parameters for uint64
min = 0
max = 18446744073709551615
```

Floats (and np.finfo)

With the following code you can see properties of floats

```
import numpy as np
print(np.finfo(np.float32))
print(np.finfo(np.float64))
```

A standard 64 bit float

```
import numpy
A = numpy.array([1.0],dtype=numpy.float64)
print( (A+1e15)−1e15 )
print( (A+1e16)−1e16 )
```

Output:

```
1.0
0.0
```

A 32 bit float

```
import numpy
A = numpy.array([1.0],dtype=numpy.float32)
print( (A+1e7)−1e7 )
print( (A+1e8)−1e8 )
```

Output:

```
1.0
0.0
```

Boolean types

Can be True or False. (1 or 0).

```
import numpy
A = numpy.arange(10)
B = A>5
print(A)
print(B)
print(B.dtype)
```

Output:

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

Takeaway message

- **float64** works if 15 decimal precision is sufficient, and we are working with numbers within $10.0^{\pm 308}$.
- **int64** works for $-9223372036854775808 \leq i \leq 9223372036854775807$.
- **uint64** works for $0 \leq i \leq 18446744073709551615$.
- **float32**, **int32** or **uint32** are useful, if you would like work with large amount of data (and if you do not need high precision).
- **bool** is for true / false.

Some references

- A technical standard for floating numbers
  `https://en.wikipedia.org/wiki/IEEE_754`

Input and output

Writing to a file

The following line prints the $0, 1, 2, 3$ and $\sin(0), \sin(1), \sin(2), \sin(3)$ to a file:

```
import numpy
file = open("File.txt","w")
for i in range(4):
    file.write('%d\t%f\n'%(i,numpy.sin(i)))
file.close()
```

open is a built in python function, **File.txt** is the name of the file we are reading, and **w** indicates that we would like to write to the file.

With string formatters

```
import numpy
file = open("File.txt","w")
for i in range(4):
    file.write("{}\t{}\n".format(i,numpy.sin(i)))
file.close()
```

Reading a file

The following line reads and prints every line in a file:

```
file = open(”File.txt”,”r”)
for line in file:
    print(line)
```

The open function

The second argument for **open** can be:

′r′ : use for reading
′w′ : use for writing
′x′ : use for creating and writing to a new file
′a′ : use for appending to a file
′r+′ : use for reading and writing to the same file

Example

```
import numpy, time

#Let us generate an array with 3e6 floats
x = numpy.random.random(3000000)
y = numpy.random.random(3000000)

#Let us first try and save and read with for loops
tstart = time.time()
f = open('Ascii_with_for_loop.txt','w')
for i in range(len(x)):
    f.write("%.18f %.18f\n"%(x[i],y[i]))
f.close()

List_x_read = []
List_y_read = []

f = open('Ascii_with_for_loop.txt','r')
for line in f:
    tmp = line.split(' ')
    List_x_read.append( float(tmp[0]) )
    List_y_read.append( float(tmp[1]) )
f.close()
print('With for loops and lists writing/reading took %.4f seconds.'%(time.time()-tstart))
```

Output: With for loops and lists writing/reading took 7.4066 seconds.

numpy.loadtxt and numpy.savetxt

numpy.loadtxt and **numpy.savetxt** are convenient function to save and store tables with numbers. They are built to work with simple tables like this one:

```
0.244624351457276457 0.530129348064506289
0.970292362788436447 0.056219423310213679
0.965128610756553429 0.810915113222641093
0.944834695135362224 0.430878051584567490
0.981248369436128587 0.937609667152707438
```

Example

```
import numpy
import time

#Let us generate two arrays with 3e6 floats
x = numpy.random.random(3000000)
y = numpy.random.random(3000000)

#We now save the data to an ascii file, read it again and measure the time it takes
tstart = time.time()
numpy.savetxt('Asciifile1.txt',[x,y])
x1,y1 =numpy.loadtxt('Asciifile1.txt')
print('With ascii files it took %.4f seconds'%(time.time()-tstart))
```

Output

```
With ascii files it took 8.0046 seconds
```

Binary files and the h5py package

- Instead of saving data to a text/ascii file, it is more efficient to save the data to *binary file*. *A binary file is computer-readable but not human-readable. In contrast, text files are stored in a form that is human-readable.*
- Recommendation: Always use binary files for large data sets (e.g. when you have GBs or TBs or data).
- The *hdf group* has created the h5py package, which can be used to read and write binary files (hdf5-files).

Example

```
import numpy, time, h5py
x = numpy.random.random(3000000)
y = numpy.random.random(3000000)

tstart = time.time()
f = h5py.File('Outputfile.hdf5','w')
dset = f.create_dataset('x', x.shape, dtype=x.dtype, data = x)
dset = f.create_dataset('y', x.shape, dtype=x.dtype, data = y)
f.close()

f=h5py.File('Outputfile.hdf5','r')
x2 = f['x'][()]
y2 = f['y'][()]
f.close()
print('With hdf5 files it took %.4f seconds.'%(time.time()-tstart))
```

Output: With hdf5 files it took 0.0330 seconds.

Note: until a few years ago you wrote .value instead of [()]

Recommendations!

- Numpy's loadtxt and savetxt are convenient for small datasets.
- Input/Output to binary files is faster and better for larger datafiles. HDF5 is a good package for binary input/output!

FITS files via pyfits

fits is the standard data format in observational astronomy.

The PyFITS module is a Python library providing access to FITS files. FITS (Flexible Image Transport System) is a portable file standard widely used in the astronomy community to store images and tables. See `https://pyfits.readthedocs.io/en/latest/`.

import astropy.io.fits as fits

pandas

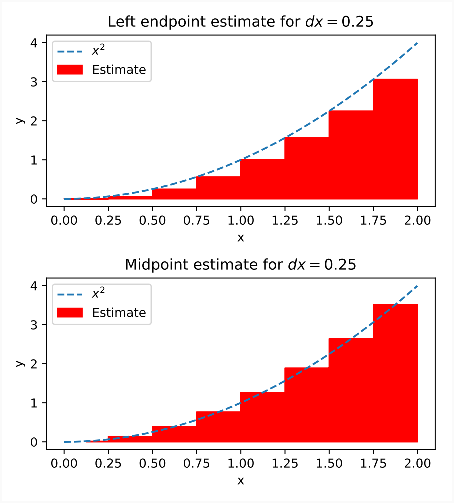
pandas is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language. See `https://pandas.pydata.org/`.

See tutorial on `https://www.learnpython.org/en/Pandas_Basics`.

Numerical integration

Visualisation of the methods

Let us calculate the integral, $\int_0^2 x^2 dx$. We will compare two methods:



Numerical integration, reminder!

We define $f(x) = x^2$. The *left endpoint method* estimate is,

$$\int_0^2 x^2 dx \approx \sum_{i=0}^{N-1} f(i \cdot \Delta x) \cdot \Delta x,$$

where $\Delta x = 2/N$.

For the *midpoint method*, we estimate

$$\int_0^2 x^2 dx \approx \sum_{i=0}^{N-1} f([i + 0.5] \cdot \Delta x) \cdot \Delta x.$$

Example

With the left endpoint method, we get

```
import numpy
dx = 0.01
x = numpy.arange(0.0,2.0,dx)
y = x**2
IntegralValue = numpy.sum(y*dx)
print('Numerical estimate of integral is %8.8g. Analytical solution is %8.8g.'\
%(IntegralValue,1.0/3.0*2.0**3))
```

```
Numerical estimate of integral is 2.6467. Analytical solution is 2.6666667.
```

Example

With the midpoint method, we get

```
import numpy
dx = 0.01
x = numpy.arange(0.0,2.0,dx)
x += dx/2
y = x**2
IntegralValue = numpy.sum(y*dx)
print('Numerical estimate of integral is %8.8g. Analytical solution is %8.8g.'\
%(IntegralValue,1.0/3.0*2.0**3))
```

```
Numerical estimate of integral is 2.66665. Analytical solution is 2.6666667.
```

Example: Simpson integration

```
import numpy
import scipy.integrate
dx = 0.01
x = numpy.arange(0.0,2.0,dx)
y = x**2
IntegralValue = scipy.integrate.simps(y,dx=dx)
print('With Simpsons method we get %8.8g. Analytical solution is %8.8g.'\
%(IntegralValue,1.0/3.0*2.0**3))
```

```
With Simpsons method we get 2.6268665. Analytical solution is 2.6666667.
```

See more on

`https://docs.scipy.org/doc/scipy/tutorial/integrate.html`.

Example: trapezoidal rule

```
import numpy
import scipy.integrate
dx = 0.01
x = numpy.arange(0.0,2.0,dx)
y = x**2
IntegralValue = scipy.integrate.trapz(y,dx=dx)
print('With Simpsons method we get %8.8g. Analytical solution is %8.8g.'\
%(IntegralValue,1.0/3.0*2.0**3))
```

See more on

`https://docs.scipy.org/doc/scipy/tutorial/integrate.html`.

Example: Integration of a function

```
import numpy
import scipy.integrate

def f(t):
    return t**2

Result = scipy.integrate.romberg(f,0.0,2.0)
print('With Rombergs method for a function we get %18.18g'%Result)

Result = scipy.integrate.quad(f,0.0,2.0)
print('With the quad method for a function we get %18.18g +/- %5.5g'%(Result[0],Result[1]))
```

Output:

```
With Rombergs method for a function we get 2.66666666666666652
With the quad method for a function we get 2.66666666666666696 +/- 2.9606e-14
```

See more about Romberg integration https://en.wikipedia.org/wiki/Romberg's_method and the quad method https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.quad.html#scipy.integrate.quad.

Scipy is an extension of Numpy

SciPy is a free and open-source Python library used for scientific computing and technical computing.

SciPy contains modules for optimization, linear algebra, integration, interpolation, special functions, FFT, signal and image processing, ODE solvers and other tasks common in science and engineering.

SciPy builds on the NumPy array object and is part of NumPy [...]

See `https://en.wikipedia.org/wiki/SciPy`.

Classes

About classes

- A class is a programming object – it usually consists of a set of functions and variables *working together*.
- It is especially good for creating reusable modular code.
- ... But it is also very counter-intuitive. The learning curve is steep!

How is a class defined and used?

```
#we define the class
class HelloWorld:
    def __init__(self):
        print('Hello world.')
    def PrintAgain(self):
        print('Hello world again.')

A = HelloWorld()#We initialize
A.PrintAgain()#We print hello world again
A.PrintAgain()#...and again
```

A slightly more advanced example

```
#we define the class
class HelloWorld:
    def __init__(self):
        self.N=1
        print('Hello world. This is time number',self.N,'we say hello.')

    def PrintAgain(self):
        self.N += 1
        print('Hello world. This is time number',self.N,'we say hello.')

A = HelloWorld()#We initialize
A.PrintAgain()#We print hello world
A.PrintAgain()#...and again
A.PrintAgain()#...and again
```

A class consists of variables and functions that work together. The functions in HelloWorld (*init* and *PrintAgain*) know about the variable, *N*. *self* is (a pointer to) the class itself.

Three examples

1. **Example.py** is an example showing how to read, rebin and plot an observational spectrum.
2. **Example2.py** and **ClassSpectrum.py** does the same with a class.
3. **Example3.py** shows how two different spectra can easily be plotted.

Conclusions about classes

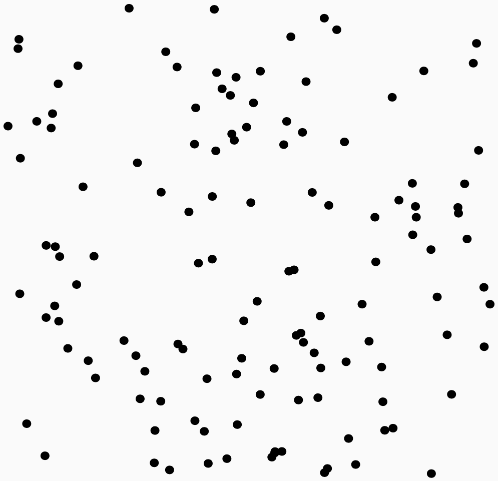
- *Classes* are good for creating re-usable code. – we demonstrated that with a code reading spectra.
- They use a high abstraction level.
- If you are not experienced writing classes, it is often best to first write a normal code (without classes), and then later re-implement the code using classes.

N-body simulations

N particles interacting only through gravity

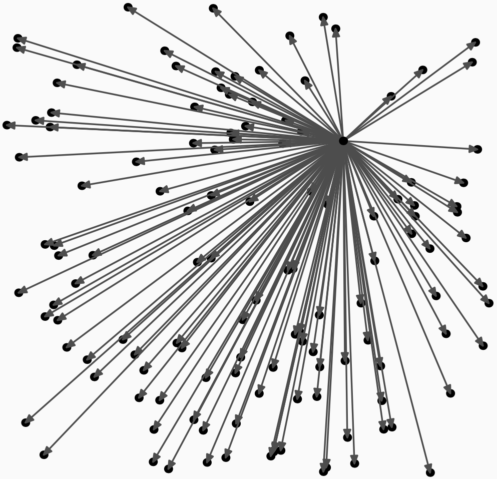
Force according to Newton's law of gravitation :

$F_{ij} = Gm_im_j/r_{ij}^2$



N particles interacting only through gravity

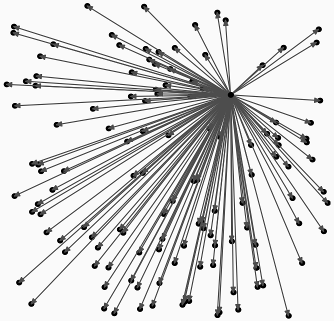
Force according to Newton's law of gravitation :
$F_{ij} = Gm_im_j/r_{ij}^2$



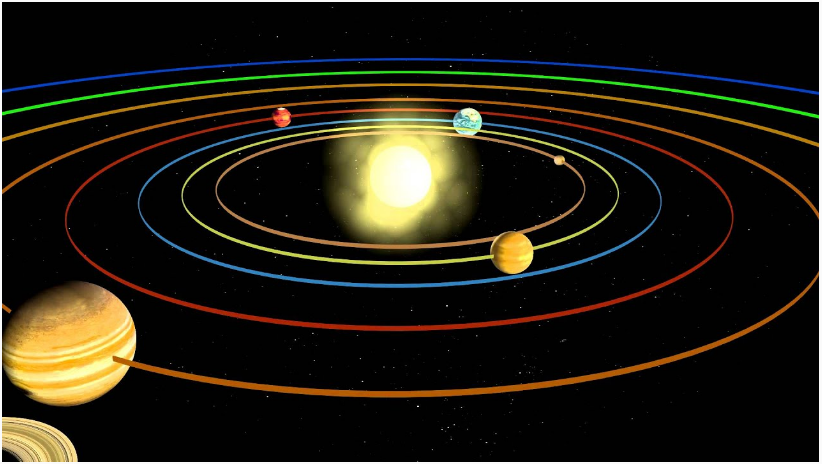
N particles interacting only through gravity

Force according to Newton's law of gravitation :
$F_{ij} = Gm_i m_j / r_{ij}^2$.

- The vector form of Newton's law is, $\boldsymbol{F}_{ij} = -\frac{Gm_i m_j}{r_{ij}^2} \frac{\boldsymbol{r}_{ij}}{r_{ij}}$, where $\boldsymbol{r}_{ij} \equiv \boldsymbol{r}_i - \boldsymbol{r}_j$,
- and the total force on particle i is $\boldsymbol{F}_i = \sum_{j \neq i} \boldsymbol{F}_{ij}$



Example I: the solar system



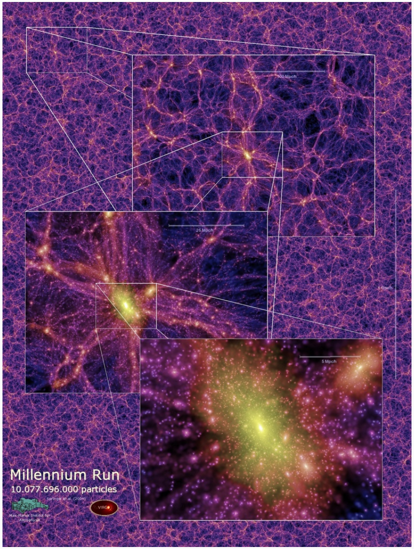
Example II: The orbit of a comet (here Hale-Bopp from 1997)

Example III: Stars in Globular Clusters (here is the M13 cluster in Hercules)

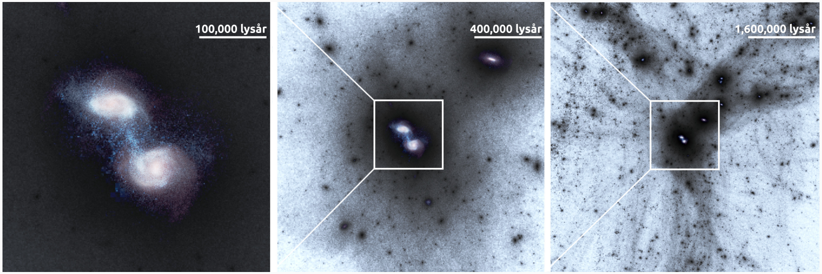
Example III: Stars in Globular Clusters (here is M13 shown through an amateur telescope)

Example IV: Stars in galaxy mergers (Mice galaxies shown)

Example V: Dark matter in the Universe (Millennium simulation)



Example VI: Merger simulations (my own work!)



Our first *N*-body code. A planet orbiting the sun

In this example we calculate the orbit of a planet around the Sun. See the following programs:

- **Nbody.py**: A simple *N*-body code.

Our first *N*-body code. A planet orbiting the sun

In this example we calculate the orbit of a planet around the Sun. See the following programs:

- **Nbody.py**: A simple *N*-body code.
- **Nbody1.py**: Same algorithm, but here we improve Matplotlib visualisation.

Our first *N*-body code. A planet orbiting the sun

In this example we calculate the orbit of a planet around the Sun. See the following programs:

- **Nbody.py**: A simple *N*-body code.
- **Nbody1.py**: Same algorithm, but here we improve Matplotlib visualisation.
- **Nbody2.py**: Error analysis

Our first N-body code. A planet orbiting the sun

In this example we calculate the orbit of a planet around the Sun. See the following programs:

- **Nbody.py**: A simple N-body code.
- **Nbody1.py**: Same algorithm, but here we improve Matplotlib visualisation.
- **Nbody2.py**: Error analysis
- **Nbody3.py**: Error analysis. We implement the *Leapfrog* algorithm, which gives better accuracy.

Our first *N*-body code. A planet orbiting the sun

In this example we calculate the orbit of a planet around the Sun. See the following programs:

- **Nbody.py**: A simple *N*-body code.
- **Nbody1.py**: Same algorithm, but here we improve Matplotlib visualisation.
- **Nbody2.py**: Error analysis
- **Nbody3.py**: Error analysis. We implement the *Leapfrog* algorithm, which gives better accuracy.
- **Nbody4.py – Nbody6.py**: Different initial velocities.

Euler integration – the first timestep

The first timestep can be written as

$$\begin{aligned} \mathbf{x}_1 &= \mathbf{x}_0 + \mathbf{v}_0 \Delta t, \\ \text{Evaluate } \mathbf{a}_1 &\text{ based on } \mathbf{x}_1, \\ \mathbf{v}_1 &= \mathbf{v}_0 + \mathbf{a}_1 \Delta t. \end{aligned}$$

Here a subscript indicates the number of a timestep, i.e. 0 corresonds to $t = 0$ and n corresponds to $t = n \cdot \Delta t$. With these three operations we have hence evolved the system from $t = 0$ to $t = \Delta t$.

Euler integration – the n'th timestep

In general, the n'th timestep is

$$\begin{aligned} \mathbf{x}_{n+1} &= \mathbf{x}_n + \mathbf{v}_n \Delta t, \\ \text{Evaluate } \mathbf{a}_{n+1} &\text{ based on } \mathbf{x}_{n+1}, \\ \mathbf{v}_{n+1} &= \mathbf{v}_n + \mathbf{a}_{n+1} \Delta t. \end{aligned}$$

It can be proven that this method is accurate to first order.

Leapfrog – initialisation

In the Leapfrog algorithm a higher accuracy (2nd order) is obtained by making a velocity offset at the initial time:

$$\begin{aligned} &\text{Evaluate } \mathbf{a}_0 \text{ based on } \mathbf{x}_0, \\ &\mathbf{v}_{1/2} = \mathbf{v}_0 + \mathbf{a}_0 \Delta t/2. \end{aligned}$$

After the initialisation the first timestep can be performed:

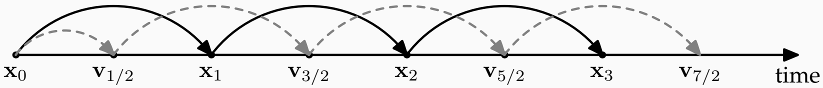
$$\begin{aligned} &\mathbf{x}_1 = \mathbf{x}_0 + \mathbf{v}_{1/2} \Delta t. \\ &\text{Evaluate } \mathbf{a}_1 \text{ based on } \mathbf{x}_1, \\ &\mathbf{v}_{3/2} = \mathbf{v}_{1/2} + \mathbf{a}_1 \Delta t. \end{aligned}$$

Leapfrog – n'th timestep

The n'th timestep can be written as:

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \mathbf{v}_{n+1/2}\Delta t,$$
$$\text{Evaluate } \mathbf{a}_{n+1} \text{ based on } \mathbf{x}_{n+1},$$
$$\mathbf{v}_{n+3/2} = \mathbf{v}_{n+1/2} + \mathbf{a}_{n+1}\Delta t.$$

This figure illustrates how the position and velocity for a particle are evolved with the leapfrog integrator.



Literature about integration of orbits

See Section 9.6 of the Feyman lectures: `https://www.feynmanlectures.caltech.edu/I_09.html`.

More examples

In this example we calculate the orbit of a planet around the Sun. See the following programs:

- **Nbody5.py**: A smaller initial velocity.

More examples

In this example we calculate the orbit of a planet around the Sun. See the following programs:

- **Nbody5.py**: A smaller initial velocity.
- **Nbody6.py**: $v > v_{\text{esc}} = \sqrt{2GM/r}$.

N-body simulations II

Structure of an N-body code

1. Generation of **initial conditions**.
2. Simulation **parameters** (e.g. setting G, time-step and maximum time).
3. The **main loop**, where equations of motion are integrated.

Initial conditions

We create a particle distribution with

$$\rho(r) = \frac{\rho_0}{r/a}\frac{1}{[1+r/a]^3}, \tag{1}$$

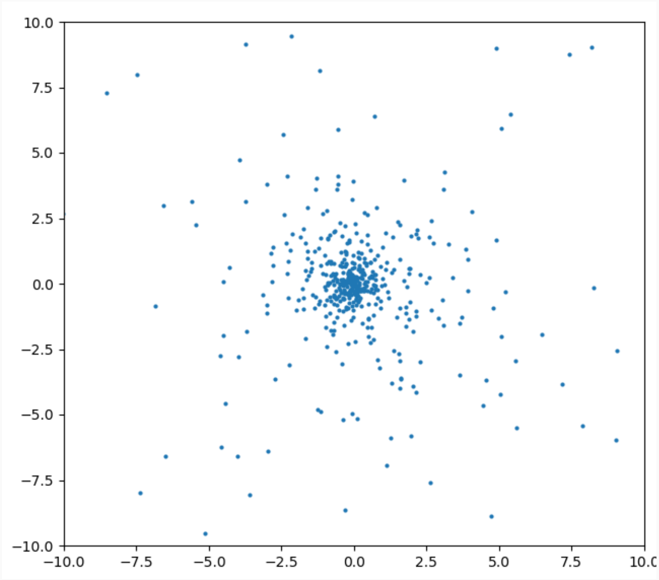
$$M(r) = M_0\frac{(r/a)^2}{(1+r/a)^2}. \tag{2}$$

where $\rho_0 = M_0/[2\pi a^3]$. M_0 is the total mass and a is the scale radius.

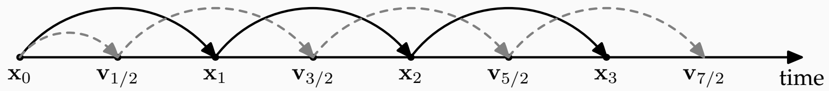
To determine the initial radius af a particle, a uniform random number between 0 and M_0 is drawn and then we can solve for r in Eq. 2.

Each particle is given a speed between 0 and $0.1 v_{\text{esc}}$ (this is an arbitrary choice).

The ICs



We use a Leapfrog integrator



Gravitational force calculation

We determine the vectors

$$\Delta x_{ij} = x_i - x_j, \tag{3}$$

$$\Delta y_{ij} = y_i - y_j, \tag{4}$$

$$\Delta z_{ij} = z_i - z_j. \tag{5}$$

And based on this determine $\mathbf{F}_{ij}$, $\mathbf{F}_i \equiv \sum_j \mathbf{F}_{ij}$, and a_x, a_y, a_z and the gravitational potential $V_i = -\sum_j Gm_j/r_{ij}$.

Gravitational softening

Instead of

$$F_{ij} = \frac{Gm_i m_j}{r_{ij}^2}, \tag{6}$$

we write

$$F_{ij} = \frac{Gm_i m_j}{r_{ij}^2 + \epsilon^2}, \tag{7}$$

where ϵ is the gravitational softening, which ensures numerical stability, so we do not divide with 0, when particles are near each other. Such a softening is always used in simulations of collisionless systems.

Class GravitySimulation

See **GravitySimulationClass.py**.

Class GravitySimulation

With the GravitySimulationClass we can run a simulation with 512 particles like this:

```
import GravitySimulationClass

C = GravitySimulationClass.GravitySimulation(OutputPrefix='MartinsSimulation')
C.InitializeHernquistHalo(Nparticles = 512)
C.RunSimulation(dt=0.01,tmax=10.0)
```

Three simulations with different random seeds

We can run three simulations with different random seeds (i.e. different realisations) like this:

```
import GravitySimulationClass

List_RandomSeed = [32941,42349,439205]

for i in List_RandomSeed:
   C = GravitySimulationClass.GravitySimulation(OutputPrefix='Seed%d'%i,RandomSeed=i)
   C.InitializeHernquistHalo(Nparticles = 128)
   C.RunSimulation(dt=0.01,tmax=10.0)
```

Four simulations with a different number of particles

We can run three simulations with different number of particles

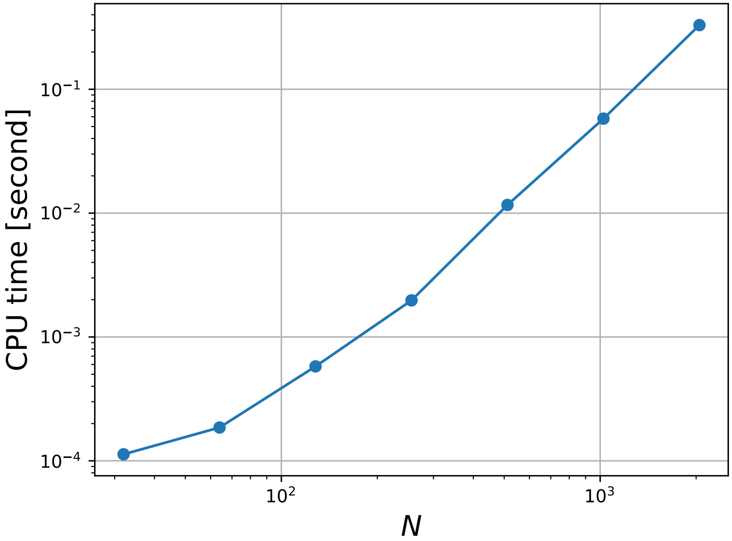
```
import GravitySimulationClass

List_N = [32,64,128,256]

for i in List_N:
   C = GravitySimulationClass.GravitySimulation(OutputPrefix='N%d'%i)
   C.InitializeHernquistHalo(Nparticles = i)
   C.RunSimulation(dt=0.01,tmax=10.0)
```

N-body simulations III

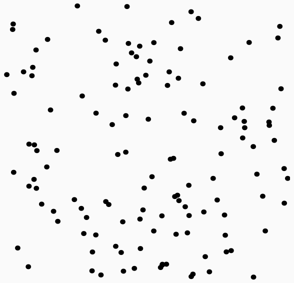
Wrap up of exercise

- Run simulations with $N = 32, 64, 128, 256, 512, 1024$ and 2048. Use $dt = 0.01$ and $tmax = 2$. Measure the mean time it takes to do a calculation of accelerations in each case, and plot (N,mean time to calculate acceleration) – use plt.loglog() to make axes logarithmic. Show that the time of an acceleration calculation scales approximately as N^2.

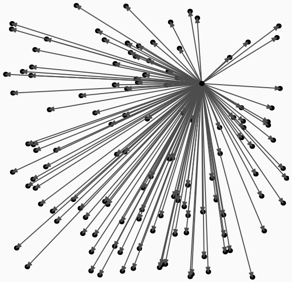
Scaling of this N-body code. CPU time $\propto N^2$



Improving the algorithm for force/acceleration calculation



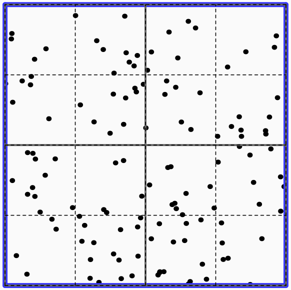
Improving the algorithm for force/acceleration calculation



Such a *direct summation* calculation consists of summing $N \times (N-1) \simeq N^2$ numbers.

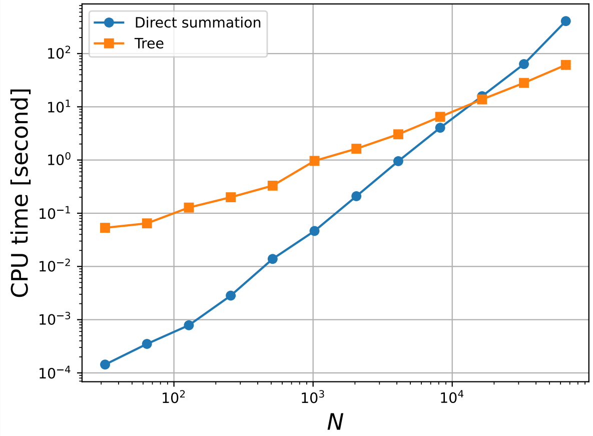
This is why we see that the time it takes to calculate the accelerations scales as time$\propto N^2$.

Improving the algorithm for force/acceleration calculation



A *tree gravity algorithm* recursively divides the simulation box into sub-boxes. We define $\theta \equiv L_n/r$, where L_n is the size of the box after n refinements and r is the distance between a particle and a box. If $\theta > 0.5$ we refine further. If $\theta \leq 0.5$ we caluclate the force on a particle based on the mass and center-of-mass of the box. Here the force calculation scales as time$\propto N \log N$.

Scaling in simulations



Here, I compare a tree code vs. direct summation for the gravitational force calculation.

Outlook

- Read more about tree codes here: `https://ui.adsabs.harvard.edu/abs/1986Natur.324..446B/abstract`
- and here `https://ui.adsabs.harvard.edu/abs/2005MNRAS.364.1105S/abstract`
- Consider following MSc course, **Advanced comp. astrophys. – Conecpt/Applications" 2V2S Module 765 Pfrommer/Pawloski**