How to use numpy to solve problems

numpy: Advantages of C++ speed but with lots of great python features and ease of use. Used mostly for matrix math and manipulation and operations on lists/arrays

Lots of great resources out there, here is just one:https://www.datacamp.com/community/tutorials/python-numpy-tutorial

Let's look at some numpy examples together, now

On arrays

Some basic numpy examples

Let's go over this all together

```
>>> import numpy as np
>>> a=np.zeros(3)
>>> print(a)
[ 0.  0.  0.]
>>> a[0] = 1
>>> a[1] = 5
>>> a[2] = 6.6
>>> print(a)
[ 1.   5.   6.6]
>>> b=np.zeros_like(a)
>>> print(b)
[ 0.  0.  0.]
>>> c=[1,2,3,4]
>>> d=np.zeros_like(c)
>>> print(c)
[1, 2, 3, 4]
>>> print(d)
[0 0 0 0]
>>> q=np.ones(5)
>>> print(q)
[ 1.  1.  1.  1.  1.]
```

So far this doesn't look that magical or useful, right?

Some basic numpy examples


```
>>> q=np.ones(5)
>>> print(q)
[ 1.  1.  1.  1.  1.]
>>> s=q+1
>>> print(s)
[ 2.  2.  2.  2.  2.]
>>> t=s+q+5.5
>>> print(t)
[ 8.5  8.5  8.5  8.5  8.5]
```

Huh, that's pretty clever!

Make an array starting at 0, ending at 5 (inclusive), with 19 spaces

```
>>> print(np.linspace(0,5,19))
[ 0.          0.27777778  0.55555556  0.83333333  1.11111111  1.38888889
  1.66666667  1.94444444  2.22222222  2.5         2.77777778  3.05555556
  3.33333333  3.61111111  3.88888889  4.16666667  4.44444444  4.72222222
  5.        ]
```

Make an array starting at 0, ending at 5 (exclusive), 0.2 increments

```
>>> np.arange(0,5,0.2)
array([ 0. ,  0.2,  0.4,  0.6,  0.8,  1. ,  1.2,  1.4,  1.6,  1.8,  2. ,
        2.2,  2.4,  2.6,  2.8,  3. ,  3.2,  3.4,  3.6,  3.8,  4. ,  4.2,
        4.4,  4.6,  4.8])
```

Some basic numpy examples


```
>>> values=np.arange(0,5,0.2)
>>> print(values)
[ 0.   0.2  0.4  0.6  0.8  1.   1.2  1.4  1.6  1.8  2.   2.2  2.4  2.6  2.8
  3.   3.2  3.4  3.6  3.8  4.   4.2  4.4  4.6  4.8]
>>> len(values)
25
>>> values[22] = 666
>>> print(values)
[  0.00000000e+00   2.00000000e-01   4.00000000e-01   6.00000000e-01
   8.00000000e-01   1.00000000e+00   1.20000000e+00   1.40000000e+00
   1.60000000e+00   1.80000000e+00   2.00000000e+00   2.20000000e+00
   2.40000000e+00   2.60000000e+00   2.80000000e+00   3.00000000e+00
   3.20000000e+00   3.40000000e+00   3.60000000e+00   3.80000000e+00
   4.00000000e+00   4.20000000e+00   6.66000000e+02   4.60000000e+00
   4.80000000e+00]
>>> values[0:25:2]
array([  0.00000000e+00,   4.00000000e-01,   8.00000000e-01,
         1.20000000e+00,   1.60000000e+00,   2.00000000e+00,
         2.40000000e+00,   2.80000000e+00,   3.20000000e+00,
         3.60000000e+00,   4.00000000e+00,   6.66000000e+02,
         4.80000000e+00])
>>> values[0:5]
array([ 0. ,  0.2,  0.4,  0.6,  0.8])
```

Size of the list

Change one value, look what happens

Print every other value!

Print the first 5 values

On arrays


```
>>> from numpy import array
>>> song = array(['hip','hop'])
>>> song
array(['hip', 'hop'], dtype='<U3')
```

Apologies if you get this, and if you don’t, it’s a bad joke :)

What did we do on the last slide?

Not sure how do mince python code, only my words. But we did **slicing**

Slicing

Get the values starting at item 0, ending at item 25, skipping by 2

```
>>> values[0:25:2]
array([  0.00000000e+00,   4.00000000e-01,   8.00000000e-01,
         1.20000000e+00,   1.60000000e+00,   2.00000000e+00,
         2.40000000e+00,   2.80000000e+00,   3.20000000e+00,
         3.60000000e+00,   4.00000000e+00,   6.66000000e+02,
         4.80000000e+00])
>>> values[0:5]
array([ 0. ,  0.2,  0.4,  0.6,  0.8])
```

Get the values starting at 0, ending at item 5 (skipping by 1 is implied). Remember that the first entry is index zero!

Note that in values[a:b:c], a is inclusive (start with it), b is exclusive (end before you get to it)

More slicing (phew!)

Very powerful feature!!!


```
>>> array=np.linspace(1,10,10)
>>> array
array([  1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.,  10.])
>>> array[1:5]
array([ 2.,  3.,  4.,  5.])
>>> array[1:5:1]
array([ 2.,  3.,  4.,  5.])
>>> array[2:2]
array([], dtype=float64)
>>> array[:2]
array([ 1.,  2.])
>>> array[0:2]
array([ 1.,  2.])
>>> array[0:3:2]
array([ 1.,  3.])
>>> array[0:]
array([  1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.,  10.])
>>> array[-1:]
array([ 10.])
>>> array[-1:0]
array([], dtype=float64)
>>> array[-1:0:-1]
array([ 10.,   9.,   8.,   7.,   6.,   5.,   4.,   3.,   2.])
>>> array[-2::2]
array([ 9.])
>>> array[-2::-2]
array([ 9.,  7.,  5.,  3.,  1.])
```

Start from entry 1 (not 0), go up to entry 5 (exclusive)

The 1 is implied, here it is explicit

Start at 2, go to 2 (exclusive) = empty

Start at beginning, go up to 2 (exclusive)

Now we explicitly show start at entry 0

Go from 0 to 3 (exclusive), count by 2

From 0 to end

Start at end, go to end (exclusive)

Start at end, go to beginning (exclusive) = null

Start at end to beginning (exclusive), count backwards

Start at second to last, to beginning (exclusive), count by 2

Start at second to last, go to beginning (exclusive), count backwards by 2

What about 2d?


```
>>> array=np.zeros((3,2))
>>> array2=np.ones((3,2))
>>> sum = array+array2
>>> print ("Array:\n",array, "\nArray2:\n", array2, "\nAnd sum:\n",sum)
Array:
 [[ 0.  0.]
 [ 0.  0.]
 [ 0.  0.]]
Array2:
 [[ 1.  1.]
 [ 1.  1.]
 [ 1.  1.]]
And sum:
 [[ 1.  1.]
 [ 1.  1.]
 [ 1.  1.]]
>>> print(5*sum-3.3*array2)
[[ 1.7  1.7]
 [ 1.7  1.7]
 [ 1.7  1.7]]
>>> a2=np.zeros_like(sum)
>>> a2
array([[ 0.,  0.],
       [ 0.,  0.],
       [ 0.,  0.]])
```

Can combine to extra dimensions!

Printing works as expected, and you can do other math, too!

zeros_like can be very useful here

Other useful features


```
>>> print(a2.shape)
(3, 2)
>>> array=5*sum-3.3*array2
>>> array[1,0] = 12
>>> array
array([[  1.7,   1.7],
       [ 12. ,   1.7],
       [  1.7,   1.7]])
>>> array.T
array([[  1.7,  12. ,   1.7],
       [  1.7,   1.7,   1.7]])
>>> array[0,0] = np.pi
>>> array[0,1] = np.pi/2
>>> np.sin(array)
array([[  1.22464680e-16,   1.00000000e+00],
       [ -5.36572918e-01,   9.91664810e-01],
       [  9.91664810e-01,   9.91664810e-01]])
>>> np.log(array)
array([[ 1.14472989,  0.45158271],
       [ 2.48490665,  0.53062825],
       [ 0.53062825,  0.53062825]])
>>> np.exp(array)
array([[  2.31406926e+01,   4.81047738e+00],
       [  1.62754791e+05,   5.47394739e+00],
       [  5.47394739e+00,   5.47394739e+00]])
```

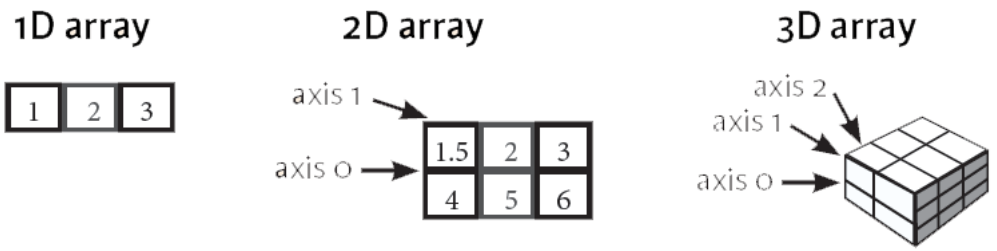
Can use sin, cos, log, etc. Lots of great features. Note that you *could* loop over each entry to do that, but numpy is fast! (Libraries compiled outside of python)

Other useful features

```
>>> array = np.arange(12)
>>> array
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
>>> b = array.reshape(6,2)
>>> print(b)
[[ 0  1]
 [ 2  3]
 [ 4  5]
 [ 6  7]
 [ 8  9]
 [10 11]]
>>> c = array.reshape(3,2,2)
>>> print(c)
[[[ 0  1]
  [ 2  3]]

 [[ 4  5]
  [ 6  7]]

 [[ 8  9]
  [10 11]]]
>>> c[2][0][1]
9
```

Reshaping can be useful. Also note that we can have more than 2d for arrays!

That being said, the arrays are still stored in a 1d contiguous chunk of memory :)



https://www.datacamp.com/community/tutorials/python-numpy-tutorial

Other useful features


```
>>> c = np.zeros(25)
>>> d = c.reshape(5,5)
>>> d
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
>>> import random as rand
>>> for ix in range(5):
...   for iy in range(5):
...     d[ix][iy] = rand.randint(0,100)
...
>>> print(d)
[[ 38.  90.  49.  18.   5.]
 [ 96.  25.  35.  96.  84.]
 [ 59.  81.  80.  94.  44.]
 [ 22.  45.  56.  19.  27.]
 [ 60.  73.  27.  36.  22.]]
>>> d[1:3,2:]
array([[ 35.,  96.,  84.],
       [ 80.,  94.,  44.]])
>>>
```

Random integer between 0 and 99

Careful: x is first index, y is second index

Slice assignment


```
>>> d
array([[  14.,   75.,   84.,    7.,   65.],
       [  27.,   95.,   44.,  123.,    1.],
       [  38.,   26.,   15.,   50.,   95.],
       [  37.,   55.,   74.,    8.,   74.],
       [  59.,    9.,   43.,   58.,   98.]])
>>> d[0:2,2:4] = 123
>>> d
array([[  14.,   75.,  123.,  123.,   65.],
       [  27.,   95.,  123.,  123.,    1.],
       [  38.,   26.,   15.,   50.,   95.],
       [  37.,   55.,   74.,    8.,   74.],
       [  59.,    9.,   43.,   58.,   98.]])
```

Let's look careful at this

On broadcasting


```
>>> q=np.ones(5)
>>> print(q)
[ 1.  1.  1.  1.  1.]
>>> s=q+1
>>> print(s)
[ 2.  2.  2.  2.  2.]
>>> t=s+q+5.5
>>> print(t)
[ 8.5  8.5  8.5  8.5  8.5]
```

Let's look more carefully at this previous example. It looked like numpy magic! But this is an example of broadcasting

```
>>> a = np.array([1.0, 2.0, 3.0])
>>> b = 2.0
>>> a * b
array([2.,  4.,  6.])
```

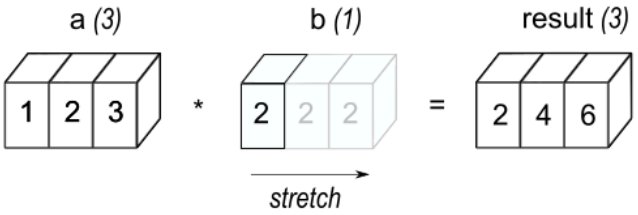


Figure 1

In the simplest example of broadcasting, the scalar b is stretched to become an array of same shape as a so the shapes are compatible for element-by-element multiplication.

https://numpy.org/doc/stable/user/basics.broadcasting.html

On broadcasting


```
>>> q=np.ones(5)
>>> print(q)
[ 1.  1.  1.  1.  1.]
>>> s=q+1
>>> print(s)
[ 2.  2.  2.  2.  2.]
>>> t=s+q+5.5
>>> print(t)
[ 8.5  8.5  8.5  8.5  8.5]
```

Here is our previous numpy code

```
>>> q=[1,1,1,1,1]
>>> s=[0,0,0,0,0]
>>> t=[0,0,0,0,0]
>>> for i in range(len(q)):
...     s[i] = q[i]+1
...     t[i] = s[i]+q[i]+5.5
...
>>> s
[2, 2, 2, 2, 2]
>>> t
[8.5, 8.5, 8.5, 8.5, 8.5]
```

This is a non-numpy version. It gives the same result, and aside from being less pretty why do we care?

Remember how python works


```
>>> q=[1,1,1,1,1]
>>> s=[0,0,0,0,0]
>>> t=[0,0,0,0,0]
>>> for i in range(len(q)):
...     s[i] = q[i]+1
...     t[i] = s[i]+q[i]+5.5
...
>>> s
[2, 2, 2, 2, 2]
>>> t
[8.5, 8.5, 8.5, 8.5, 8.5]
```

Python is dynamically typed. Each time Python sees a line of code, it needs to check the type and make sure of its scope. This is SLOW and happens at every iteration of the loop

Remember that C++ is a strongly typed language. The above checks are made during compilation (you store vectors of float, or int, not a vector of unknown type!) That makes C++ much faster. NumPy is essentially a wrapper pointing Python to strongly typed C code!

So why do we need broadcasting?


```
>>> q=np.ones(5)
>>> print(q)
[ 1.  1.  1.  1.  1.]
>>> s=q+1
>>> print(s)
[ 2.  2.  2.  2.  2.]
>>> t=s+q+5.5
>>> print(t)
[ 8.5  8.5  8.5  8.5  8.5]
```

Want to avoid loops over objects that don't have definite type

Numpy broadcasting allows smaller arrays to be repeated to minimize the number of loops in Python of unknown type. All the time-consuming things happen in C behind the scenes!

The smaller array is "repeated" multiple times until it has the same shape as the larger one, though what actually happens is it gets used multiple times so as not to increase memory usage

We won't focus on optimizing code inside and with numpy (that's another course!) but good to keep in mind

Let's check this

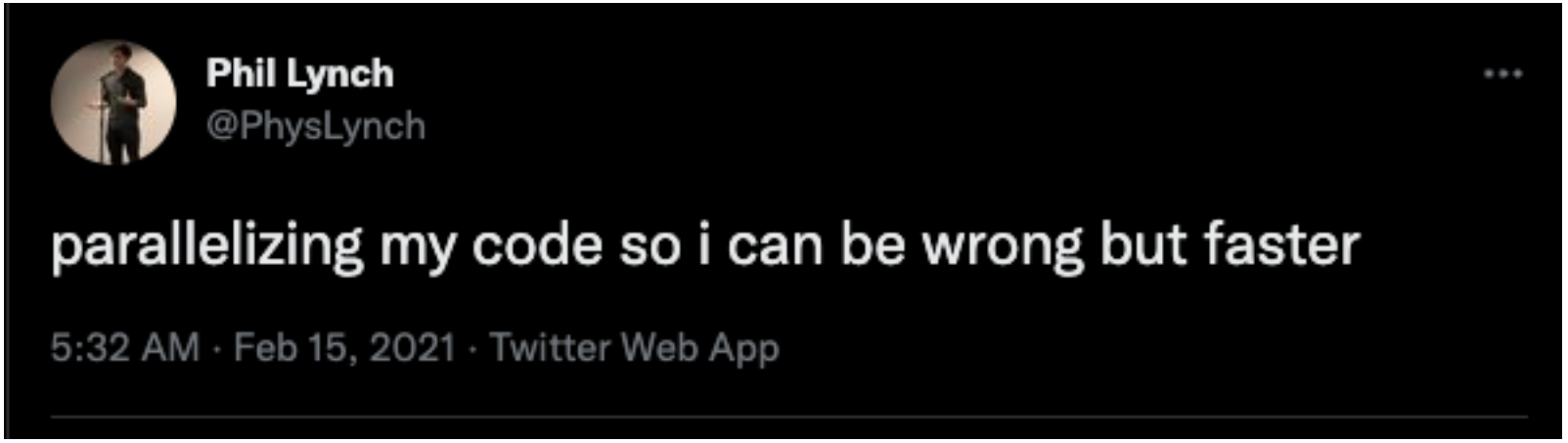

```
import time
import numpy as np

def calcFunc(n):
  vec1=np.random.randn(n)
  vec2=np.random.randn(n)
  constant=np.random.randn(1)
  val = 0
  valnp = 0

  tic = time.perf_counter()
  for i in range(n):
    val += constant*vec1[i]+vec2[i]
  toc = time.perf_counter()

  t = toc-tic

  tic = time.perf_counter()
  valnp = np.sum(constant*vec1+vec2)
  toc = time.perf_counter()
  tnp = toc-tic
  print("Calculated ",val[0], "and with np = ",valnp,"in ",t,"seconds", "and using np ", tnp, ", so speedup = ",t/tnp," for n =",n)

calcFunc(10**1)
calcFunc(10**2)
calcFunc(10**3)
calcFunc(10**4)
calcFunc(10**5)
calcFunc(10**6)
calcFunc(10**7)
```

```
Calculated  -0.9793656227385931 and with np =  -0.9793656227385926 in  0.0001613800000086485 seconds and using np  5.200999999033229e-05 , so speedup =  3.102864834428881  for n = 10
Calculated  7.259448074155098 and with np =  7.259448074155107 in  0.0007643830000176877 seconds and using np  8.1238999996458e-05 , so speedup =  9.409064612452326  for n = 100
Calculated  48.830690625379724 and with np =  48.83069062537972 in  0.006735426999995298 seconds and using np  0.011183783999996422 , so speedup =  0.6022493817832545  for n = 1000
Calculated  -173.03515893766095 and with np =  -173.03515893766155 in  0.1614091030000111 seconds and using np  0.00030696199999624696 , so speedup =  525.8276366520435  for n = 10000
Calculated  92.45334815719491 and with np =  92.45334815719202 in  1.8474122690000172 seconds and using np  0.00782565099999033 , so speedup =  236.07138486016052  for n = 100000
Calculated  -3038.741038383363 and with np =  -3038.741038383328 in  14.484037419999993 seconds and using np  0.00725014899998655 , so speedup =  1997.7572074762688  for n = 1000000
Calculated  -8055.281695791467 and with np =  -8055.281695791857 in  40.901180083000014 seconds and using np  0.062466465999989396 , so speedup =  654.7701943472672  for n = 10000000
```

Other advantages

By making arithmetic the same type as much as possible, we can parallelize the code (perform multiple operations all at once). This is challenging if every operation is different

Numpy will not parallelize your code. There are many tools to do this (using multiple threads/cores or even TPUs or GPUs), which we will not have time to cover this semester but are super important on their own



https://twitter.com/PhysLynch/status/1361277073399103488

Operations on lists and not just numpy arrays


```
>>> q=np.arange(12)
>>> q
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
>>> sum(q)
66
>>> min(q)
0
>>> q[0] = 12
>>> min(q)
1
>>> max(q)
12
>>> q
array([12,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
>>> sorted(q)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

More fun with numpy


```
[18] a=np.sin(a)
     print(a)
```

```
[[ 0.          0.84147098  0.90929743  0.14112001 -0.7568025  -0.95892427
  -0.2794155   0.6569866   0.98935825  0.41211849]
 [ 0.84147098  0.90929743  0.14112001 -0.7568025  -0.95892427 -0.2794155
   0.6569866   0.98935825  0.41211849 -0.54402111]
 [ 0.90929743  0.14112001 -0.7568025  -0.95892427 -0.2794155   0.6569866
   0.98935825  0.41211849 -0.54402111 -0.99999021]
 [ 0.14112001 -0.7568025  -0.95892427 -0.2794155   0.6569866   0.98935825
   0.41211849 -0.54402111 -0.99999021 -0.53657292]
 [-0.7568025  -0.95892427 -0.2794155   0.6569866   0.98935825  0.41211849
  -0.54402111 -0.99999021 -0.53657292  0.42016704]
 [-0.95892427 -0.2794155   0.6569866   0.98935825  0.41211849 -0.54402111
  -0.99999021 -0.53657292  0.42016704  0.99060736]
 [-0.2794155   0.6569866   0.98935825  0.41211849 -0.54402111 -0.99999021
  -0.53657292  0.42016704  0.99060736  0.65028784]
 [ 0.6569866   0.98935825  0.41211849 -0.54402111 -0.99999021 -0.53657292
   0.42016704  0.99060736  0.65028784 -0.28790332]
 [ 0.98935825  0.41211849 -0.54402111 -0.99999021 -0.53657292  0.42016704
   0.99060736  0.65028784 -0.28790332 -0.96139749]
 [ 0.41211849 -0.54402111 -0.99999021 -0.53657292  0.42016704  0.99060736
   0.65028784 -0.28790332 -0.96139749 -0.75098725]]
```

```
[23] selection = a > 0
```

Boolean array based on requirements

```
[24] selection
```

```
array([[False,  True,  True,  True, False, False, False,  True,  True,
         True],
       [ True,  True,  True, False, False, False,  True,  True,  True,
        False],
       [ True,  True, False, False, False,  True,  True,  True, False,
        False],
       [ True, False, False, False,  True,  True,  True, False, False,
        False],
       [False, False, False,  True,  True,  True, False, False, False,
         True],
       [False, False,  True,  True,  True, False, False, False,  True,
         True],
       [False,  True,  True,  True, False, False, False,  True,  True,
         True],
       [ True,  True,  True, False, False, False,  True,  True,  True,
        False],
       [ True,  True, False, False, False,  True,  True,  True, False,
        False],
       [ True, False, False, False,  True,  True,  True, False, False,
        False]])
```

Use boolean array as an index!!!

```
[25] a[selection]
```

```
array([0.84147098, 0.90929743, 0.14112001, 0.6569866 , 0.98935825,
       0.41211849, 0.84147098, 0.90929743, 0.14112001, 0.6569866 ,
       0.98935825, 0.41211849, 0.90929743, 0.14112001, 0.6569866 ,
       0.98935825, 0.41211849, 0.14112001, 0.6569866 , 0.98935825,
       0.41211849, 0.6569866 , 0.98935825, 0.41211849, 0.42016704,
       0.6569866 , 0.98935825, 0.41211849, 0.42016704, 0.99060736,
       0.6569866 , 0.98935825, 0.41211849, 0.42016704, 0.99060736,
       0.65028784, 0.6569866 , 0.98935825, 0.41211849, 0.42016704,
       0.99060736, 0.65028784, 0.98935825, 0.41211849, 0.42016704,
       0.99060736, 0.65028784, 0.41211849, 0.42016704, 0.99060736,
       0.65028784])
```

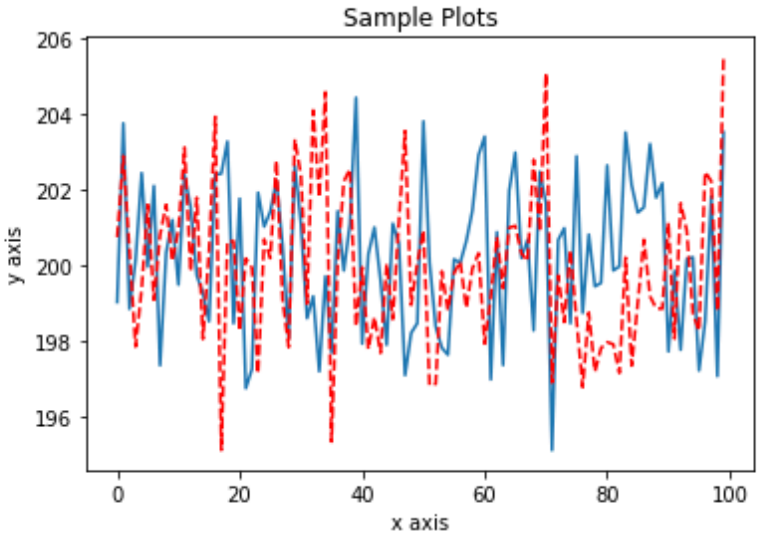
On arrays

```
>>> from numpy import array
>>> thing = array(['sun','shine'])
>>> thing
array(['sun', ‘shine'], dtype='<U3')
```

Is this any better?

On plotting

```
[11] import numpy as np
     from matplotlib import pyplot as plt

     ys = 200 + 2*np.random.randn(100)
     ys2 = 200 - 2*(np.random.randn(100))
     xs = [x for x in range(len(ys))]

     plt.plot(xs, ys, '-')
     plt.plot(xs, ys2, 'r--')

     plt.xlabel("x axis")
     plt.ylabel("y axis")
     plt.title("Sample Plots")
     plt.show()
```



Really useful tutorials and basic information on how to use matplotlib for plotting in python in the textbook (see most of chapter 3!). We'll be using it quite often. Please read and make sure you follow it all. Not too complicated, and you are not obligated to use it but I strongly recommend that you do so. We can also go over some examples now in colab

Plotting a 2d variable


```
import numpy as np
a = np.linspace(0,9,10).reshape(10,1)
print(a)
```

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

```
[5] b = np.linspace(0,9,10)
    print(b)
```

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

```
[6] a = a + b
```

```
[7] print(a)
```

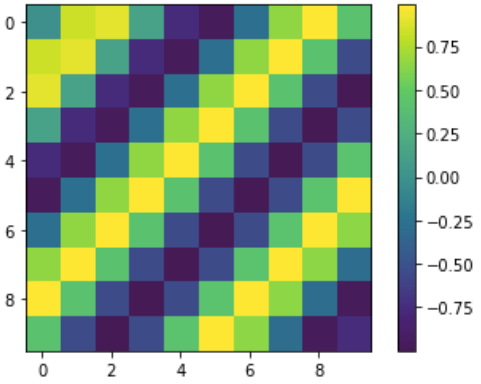
```
[[ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9.]
 [ 1.  2.  3.  4.  5.  6.  7.  8.  9. 10.]
 [ 2.  3.  4.  5.  6.  7.  8.  9. 10. 11.]
 [ 3.  4.  5.  6.  7.  8.  9. 10. 11. 12.]
 [ 4.  5.  6.  7.  8.  9. 10. 11. 12. 13.]
 [ 5.  6.  7.  8.  9. 10. 11. 12. 13. 14.]
 [ 6.  7.  8.  9. 10. 11. 12. 13. 14. 15.]
 [ 7.  8.  9. 10. 11. 12. 13. 14. 15. 16.]
 [ 8.  9. 10. 11. 12. 13. 14. 15. 16. 17.]
 [ 9. 10. 11. 12. 13. 14. 15. 16. 17. 18.]]
```

```
[18] a=np.sin(a)
     print(a)
```

```
[[ 0.          0.84147098  0.90929743  0.14112001 -0.7568025  -0.95892427
  -0.2794155   0.6569866   0.98935825  0.41211849]
 [ 0.84147098  0.90929743  0.14112001 -0.7568025  -0.95892427 -0.2794155
   0.6569866   0.98935825  0.41211849 -0.54402111]
 [ 0.90929743  0.14112001 -0.7568025  -0.95892427 -0.2794155   0.6569866
   0.98935825  0.41211849 -0.54402111 -0.99999021]
 [ 0.14112001 -0.7568025  -0.95892427 -0.2794155   0.6569866   0.98935825
   0.41211849 -0.54402111 -0.99999021 -0.53657292]
 [-0.7568025  -0.95892427 -0.2794155   0.6569866   0.98935825  0.41211849
  -0.54402111 -0.99999021 -0.53657292  0.42016704]
 [-0.95892427 -0.2794155   0.6569866   0.98935825  0.41211849 -0.54402111
  -0.99999021 -0.53657292  0.42016704  0.99060736]
 [-0.2794155   0.6569866   0.98935825  0.41211849 -0.54402111 -0.99999021
  -0.53657292  0.42016704  0.99060736  0.65028784]
 [ 0.6569866   0.98935825  0.41211849 -0.54402111 -0.99999021 -0.53657292
   0.42016704  0.99060736  0.65028784 -0.28790332]
 [ 0.98935825  0.41211849 -0.54402111 -0.99999021 -0.53657292  0.42016704
   0.99060736  0.65028784 -0.28790332 -0.96139749]
 [ 0.41211849 -0.54402111 -0.99999021 -0.53657292  0.42016704  0.99060736
   0.65028784 -0.28790332 -0.96139749 -0.75098725]]
```

```
import matplotlib.pyplot as plt
plt.imshow(a)
plt.colorbar()
```

```
<matplotlib.colorbar.Colorbar at 0x7f3892d0bc50>
```



Stepping back and thinking about something else, briefly

Does anyone here know Hubble's Law? (Without googling it or sneaking a peak at later slides)

Edwin Hubble

Astronomer who made critical contributions to our understanding of the universe. Moved to nearby Wheaton when he was 11, and attended University of Chicago

https://en.wikipedia.org/wiki/Edwin_Hubble

Edwin Hubble and his famous paper

https://www.pnas.org/content/pnas/15/3/168.full.pdf

A RELATION BETWEEN DISTANCE AND RADIAL VELOCITY AMONG EXTRA-GALACTIC NEBULAE

By Edwin Hubble

Mount Wilson Observatory, Carnegie Institution of Washington

Communicated January 17, 1929

https://en.wikipedia.org/wiki/Triangulum_Galaxy

Triangulum Galaxy (NGC 598)

TABLE 1

NEBULAE WHOSE DISTANCES HAVE BEEN ESTIMATED FROM STARS INVOLVED OR FROM MEAN LUMINOSITIES IN A CLUSTER

OBJECT	m_s	r	v	m_t	M_t
S. Mag.	..	0.032	+ 170	1.5	−16.0
L. Mag.	..	0.034	+ 290	0.5	17.2
N. G. C. 6822	..	0.214	− 130	9.0	12.7
598	..	0.263	− 70	7.0	15.1
221	..	0.275	− 185	8.8	13.4
224	..	0.275	− 220	5.0	17.2
5457	17.0	0.45	+ 200	9.9	13.3
4736	17.3	0.5	+ 290	8.4	15.1
5194	17.3	0.5	+ 270	7.4	16.1
4449	17.8	0.63	+ 200	9.5	14.5
4214	18.3	0.8	+ 300	11.3	13.2
3031	18.5	0.9	− 30	8.3	16.4
3627	18.5	0.9	+ 650	9.1	15.7
4826	18.5	0.9	+ 150	9.0	15.7
5236	18.5	0.9	+ 500	10.4	14.4
1068	18.7	1.0	+ 920	9.1	15.9
5055	19.0	1.1	+ 450	9.6	15.6
7331	19.0	1.1	+ 500	10.4	14.8
4258	19.5	1.4	+ 500	8.7	17.0
4151	20.0	1.7	+ 960	12.0	14.2
4382	..	2.0	+ 500	10.0	16.5
4472	..	2.0	+ 850	8.8	17.7
4486	..	2.0	+ 800	9.7	16.8
4649	..	2.0	+1090	9.5	17.0
Mean					−15.5

m_s = photographic magnitude of brightest stars involved.
r = distance in units of 10^6 parsecs. The first two are Shapley's values.
v = measured velocities in km./sec. N. G. C. 6822, 221, 224 and 5457 are recent determinations by Humason.
m_t = Holetschek's visual magnitude as corrected by Hopmann. The first three objects were not measured by Holetschek, and the values of m_t represent estimates by the author based upon such data as are available.
M_t = total visual absolute magnitude computed from m_t and r.

Edwin Hubble and his famous paper

https://www.pnas.org/content/pnas/15/3/168.full.pdf

A RELATION BETWEEN DISTANCE AND RADIAL VELOCITY AMONG EXTRA-GALACTIC NEBULAE

By Edwin Hubble

Mount Wilson Observatory, Carnegie Institution of Washington

Communicated January 17, 1929

1 pc = 3.1 x 10^{16} m
Positive v implies away from us, negative toward us

TABLE 1

NEBULAE WHOSE DISTANCES HAVE BEEN ESTIMATED FROM STARS INVOLVED OR FROM MEAN LUMINOSITIES IN A CLUSTER

OBJECT	m_s	r	v	m_t	M_t
S. Mag.	..	0.032	+ 170	1.5	−16.0
L. Mag.	..	0.034	+ 290	0.5	17.2
N. G. C. 6822	..	0.214	− 130	9.0	12.7
598	..	0.263	− 70	7.0	15.1
221	..	0.275	− 185	8.8	13.4
224	..	0.275	− 220	5.0	17.2
5457	17.0	0.45	+ 200	9.9	13.3
4736	17.3	0.5	+ 290	8.4	15.1
5194	17.3	0.5	+ 270	7.4	16.1
4449	17.8	0.63	+ 200	9.5	14.5
4214	18.3	0.8	+ 300	11.3	13.2
3031	18.5	0.9	− 30	8.3	16.4
3627	18.5	0.9	+ 650	9.1	15.7
4826	18.5	0.9	+ 150	9.0	15.7
5236	18.5	0.9	+ 500	10.4	14.4
1068	18.7	1.0	+ 920	9.1	15.9
5055	19.0	1.1	+ 450	9.6	15.6
7331	19.0	1.1	+ 500	10.4	14.8
4258	19.5	1.4	+ 500	8.7	17.0
4151	20.0	1.7	+ 960	12.0	14.2
4382	..	2.0	+ 500	10.0	16.5
4472	..	2.0	+ 850	8.8	17.7
4486	..	2.0	+ 800	9.7	16.8
4649	..	2.0	+1090	9.5	17.0
Mean					−15.5

m_s = photographic magnitude of brightest stars involved.
r = distance in units of 10^6 parsecs. The first two are Shapley's values.
v = measured velocities in km./sec. N. G. C. 6822, 221, 224 and 5457 are recent determinations by Humason.
m_t = Holetschek's visual magnitude as corrected by Hopmann. The first three objects were not measured by Holetschek, and the values of m_t represent estimates by the author based upon such data as are available.
M_t = total visual absolute magnitude computed from m_t and r.

Triangulum Galaxy (NGC 598)

What did Hubble find?

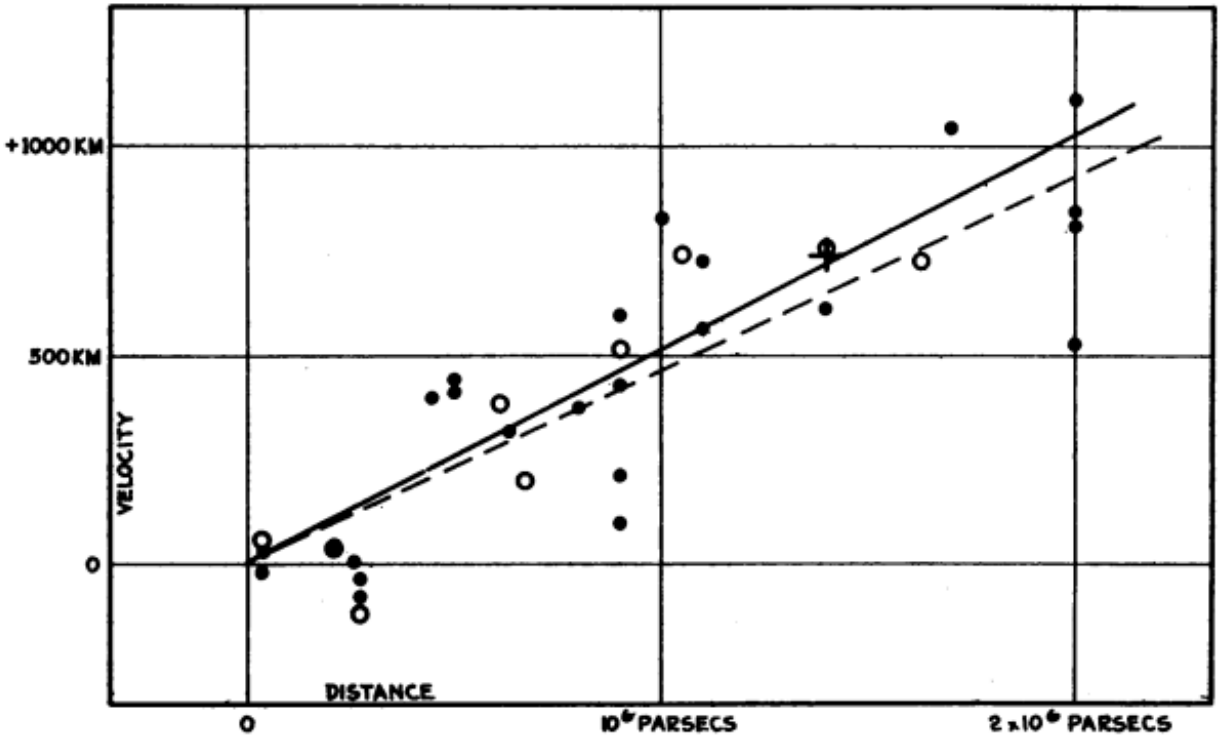


FIGURE 1

Velocity-Distance Relation among Extra-Galactic Nebulae.

Radial velocities, corrected for solar motion, are plotted against distances estimated from involved stars and mean luminosities of nebulae in a cluster. The black discs and full line represent the solution for solar motion using the nebulae individually; the circles and broken line represent the solution combining the nebulae into groups; the cross represents the mean velocity corresponding to the mean distance of 22 nebulae whose distances could not be estimated individually.

Clear linear relationship! How does one fit such curves, though?

Linear fits

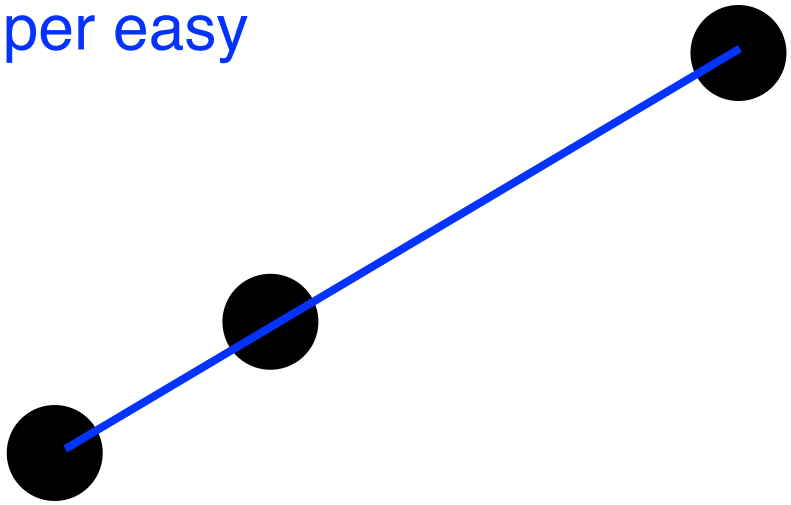
Well, that doesn't help, we can't fit
a straight line to 0 or 1 point!

Linear fits

Well, with 2 points, this is easy

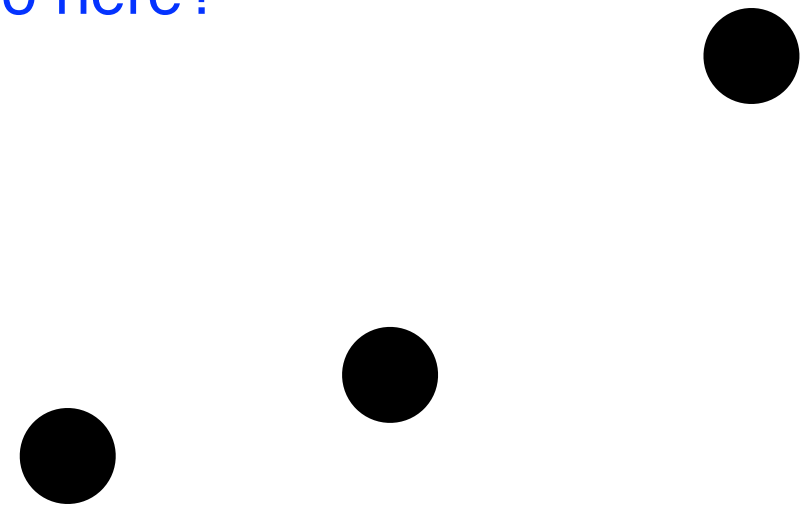
Linear fits

IF the 3 points line up and are colinear, this is also super easy



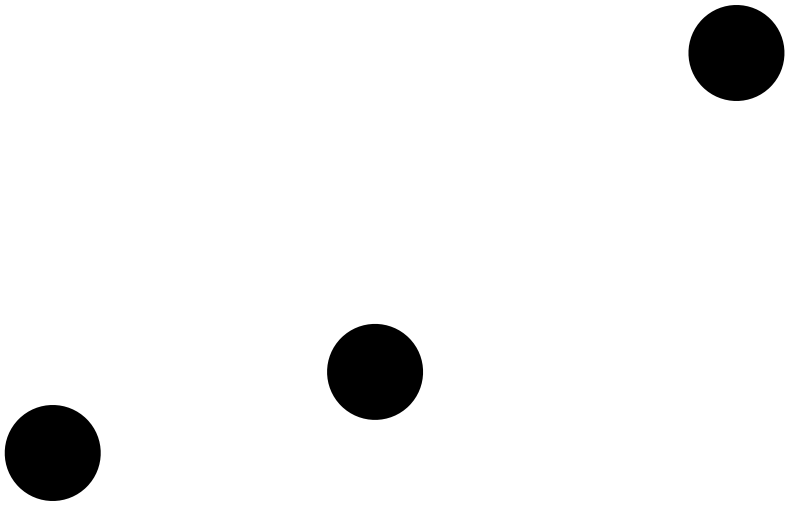
Linear fits

Uh oh, what do we do here?



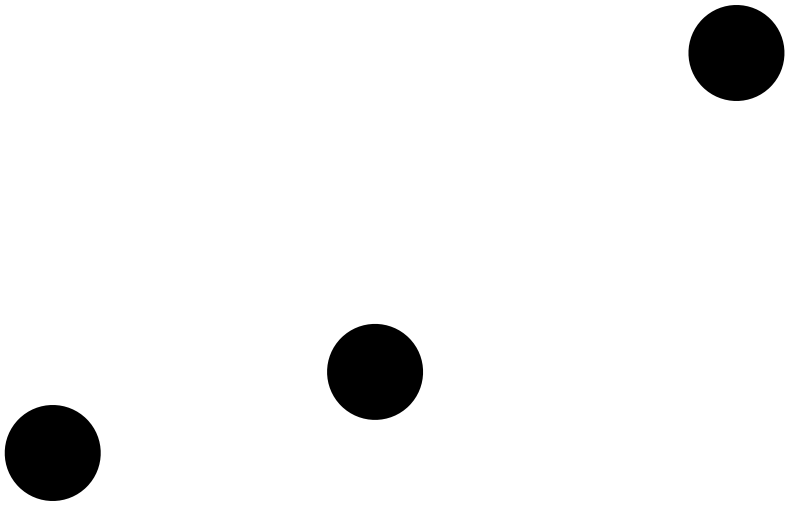
Well, there are a few reasons why our points might not all be colinear

Linear fits



Maybe each point is the result of statistical sampling. We repeat some measurement/sampling over and over; if so, this is easy to estimate and understand

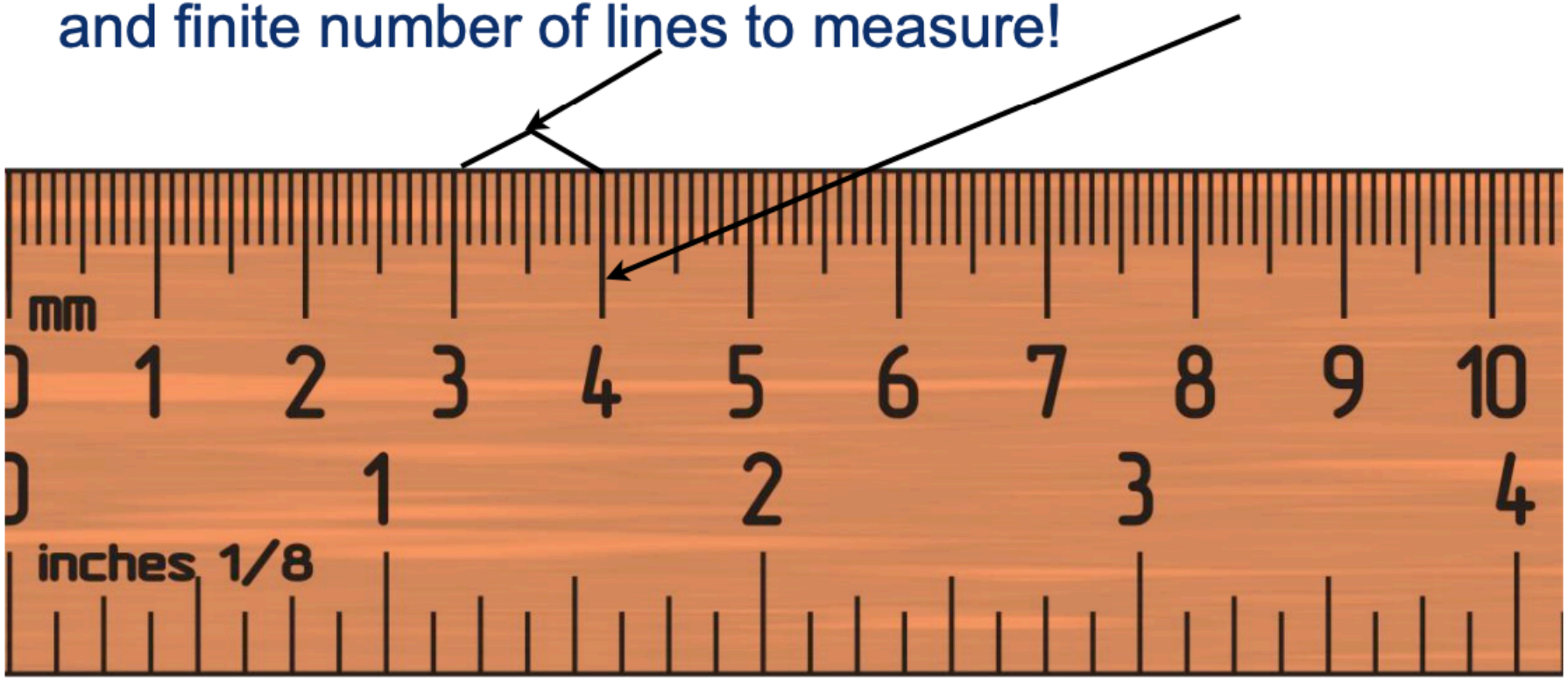
Linear fits



We can also have systematic uncertainties (maybe our ruler is not correct, maybe our stop watch is wrong, maybe there is human error, or otherwise). This is trickier.

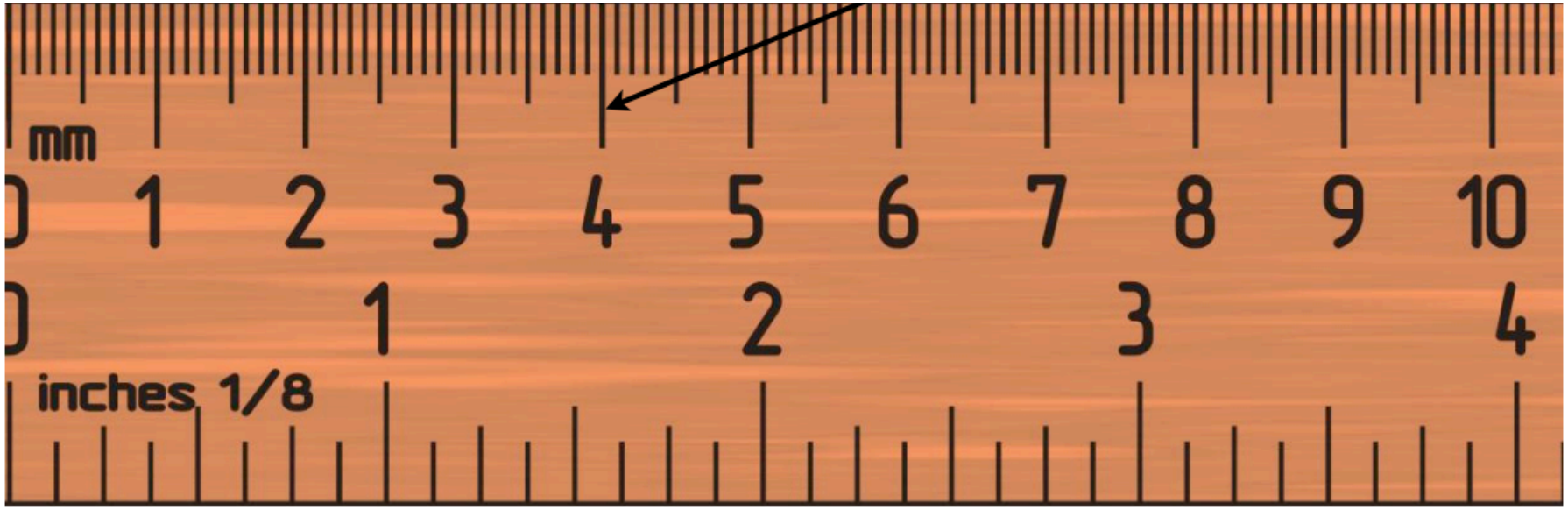
On uncertainties — Nice summary from Sal

- Example : measuring distance with a ruler
  - Systematic limitation : ruler has finite width of the lines, and finite number of lines to measure!



  - Statistical variation : you can try to repeat the same measurement over and over to get a better estimate of the “true” value

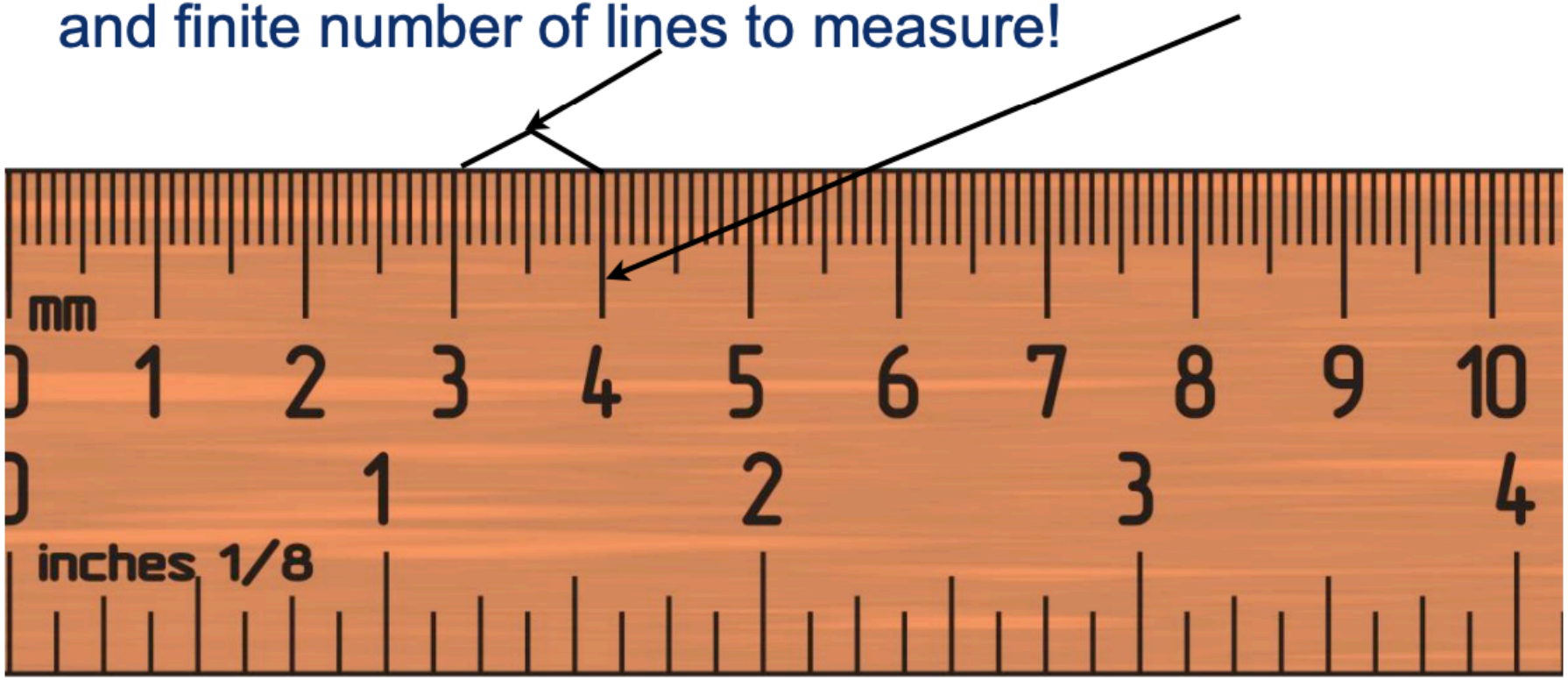
Repeat more measurements and this uncertainty shrinks!



– Statistical variation : you can try to repeat the same measurement over and over to get a better estimate of the "true" value

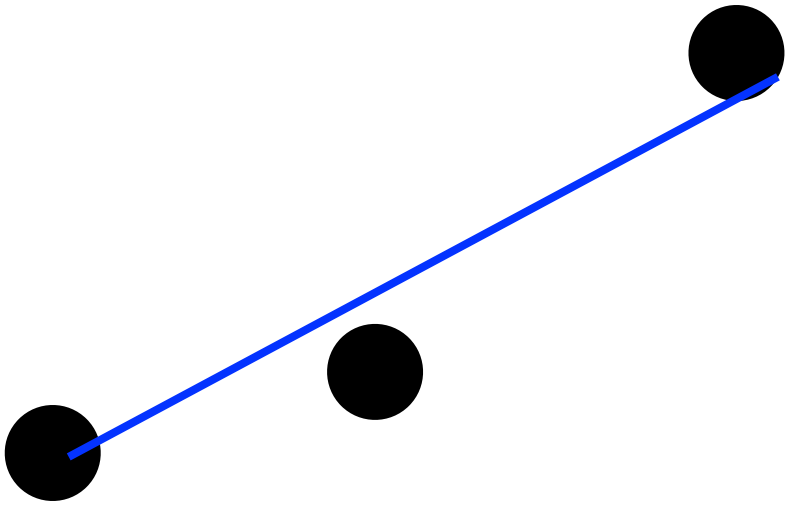
On uncertainties — Nice summary from Sal

- Example : measuring distance with a ruler
  - Systematic limitation : ruler has finite width of the lines, and finite number of lines to measure!



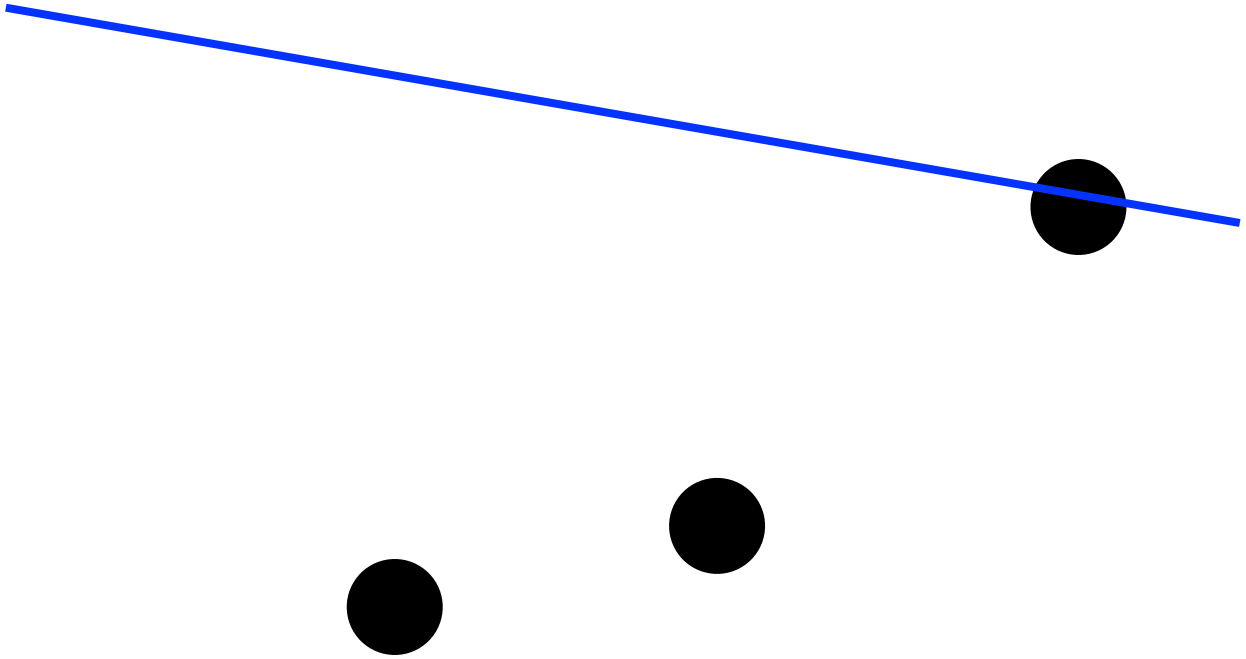
Unfortunately, more trials doesn't help with this. And it's harder to estimate!

Linear fits



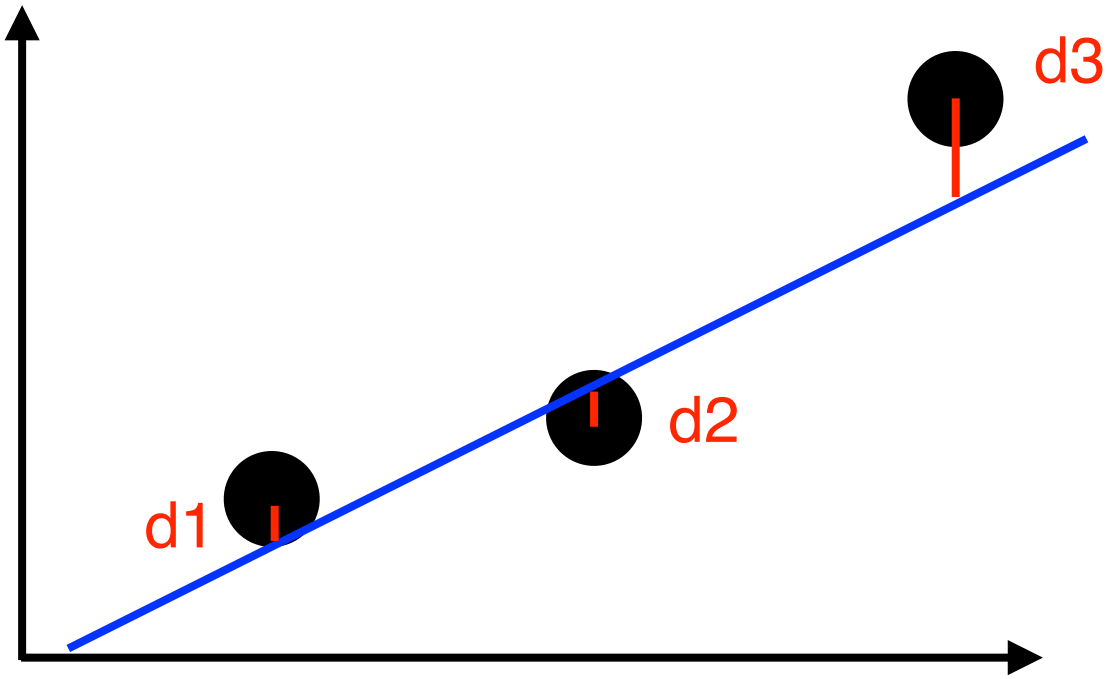
This looks like a reasonable guess and a good fit (of course, we will be more mathematically rigorous shortly)

How to do our linear fits?



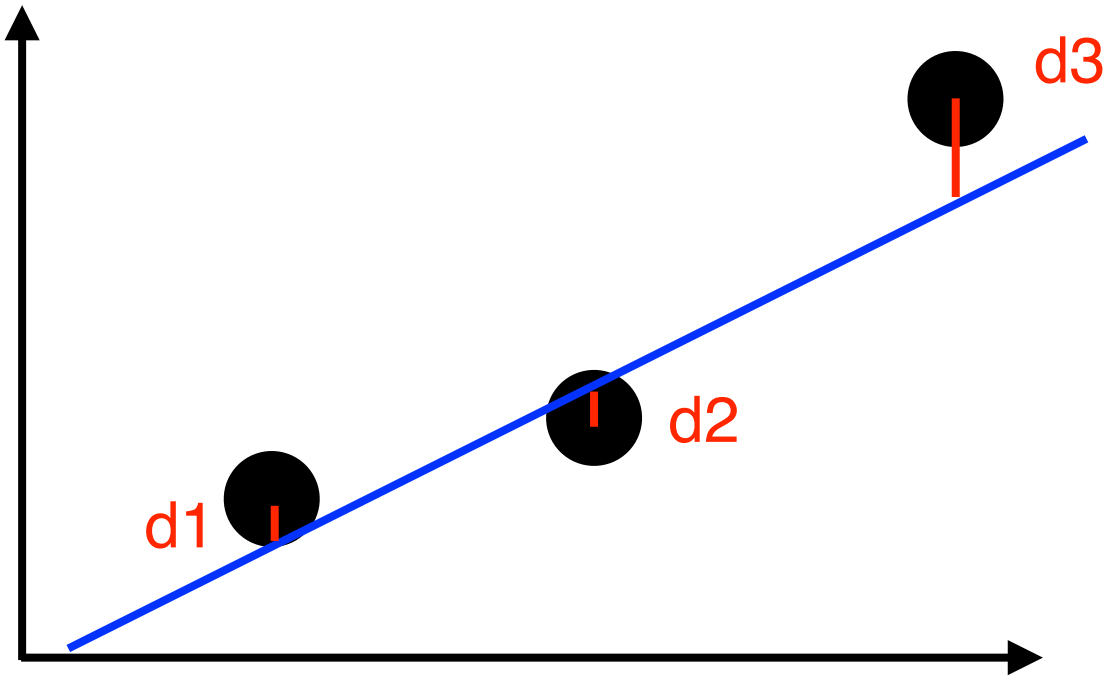
This is clearly not a good fit!

Why does the first line/fit look better?



Assume the errors are all the same (this assumption will be relaxed next). Our straight line is given by $y(x) = a+bx$, so that our prediction for point x_i is $y_{pred,i} = a+bx_i$ and then the distance between this and the measured value y_i is given by $y_i - y_{pred,i} = y_i - a- bx_i$

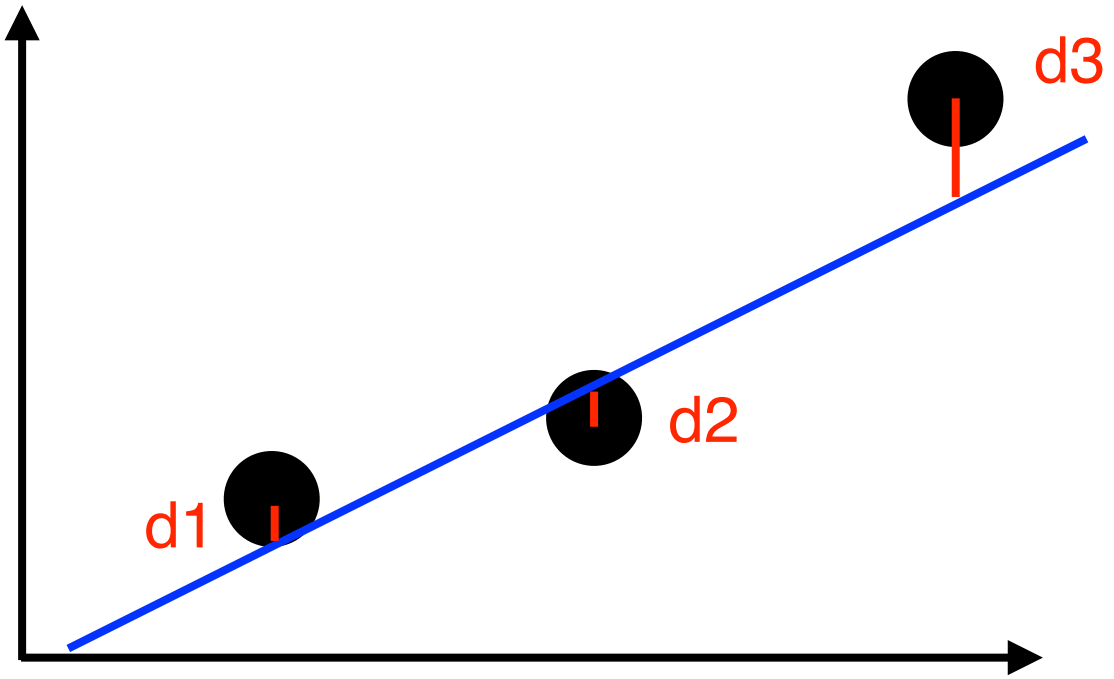
Why does the first line/fit look better?



The distance between the observed and measured value for point i is y_i - a- bx_i. Our total "distance" squared is then:

$$f(a,b) = \sum_i (y_i - a - bx_i)^2$$

How to get best-fit parameters



We want to **minimize** the following quantity:

$$f(a,b) = \sum_i (y_i - a - bx_i)^2$$

How to get best-fit parameters

We want to **minimize** the following quantity:

$$f(a,b) = \sum_i (y_i - a - bx_i)^2$$

$$\frac{\partial f}{\partial a} = -2\sum_i (y_i - a - bx_i)$$

$$\frac{\partial f}{\partial b} = -2\sum_i x_i(y_i - a - bx_i)$$

Two equations, two unknowns (a,b)

How to get best-fit parameters

$$\frac{\partial f}{\partial a} = -2\sum_i (y_i - a - bx_i) = 0$$

$$\frac{\partial f}{\partial b} = -2\sum_i x_i(y_i - a - bx_i) = 0$$

$$\sum_i (y_i - a - bx_i) = 0$$

$$\sum_i x_i(y_i - a - bx_i) = 0$$

$$\sum_i y_i - Na - b\sum x_i = 0$$

$$\sum_i x_i y_i - a\sum x_i - b\sum x_i^2 = 0$$

How to get best-fit parameters

$$\sum_i y_i - Na - b\sum x_i = 0$$

$$\sum_i x_i y_i - a\sum x_i - b\sum x_i^2 = 0$$

$$a = \frac{1}{N}\sum y_i - b\frac{1}{N}\sum x_i$$

$$a = \frac{1}{\sum x_i}\sum x_i y_i - b\frac{1}{\sum x_i}\sum x_i^2$$

How to get best-fit parameters

$$a = \frac{1}{N}\sum y_i - b\frac{1}{N}\sum x_i$$

$$a = \frac{1}{\sum x_i}\sum x_i y_i - b\frac{1}{\sum x_i}\sum x_i^2$$

$$s_y/N - (b/N)s_x = (s_{xy}/s_x) - b(s_{xx}/s_x)$$

$$b(s_{xx}/s_x - s_x/N) = s_{xy}/s_x - s_y/N$$

$$b(Ns_{xx} - s_x^2) = Ns_{xy} - s_x s_y$$

$$b = (Ns_{xy} - s_x s_y)/(Ns_{xx} - s_x * s_x)$$

How to get best-fit parameters

$$a = \frac{1}{N}\sum y_i - b\frac{1}{N}\sum x_i$$

$$a = \frac{1}{\sum x_i}\sum x_i y_i - b\frac{1}{\sum x_i}\sum x_i^2$$

$$s_y/N - a = (b/N)s_x$$

$$s_{xy}/s_x - a = (b/s_x)s_{xx}$$

$$b = s_y/s_x - aN/s_x = s_{xy}/s_{xx} - as_x/s_{xx}$$

$$s_y/s_x - s_{xy}/s_{xx} = a(N/s_x - s_x/s_{xx})$$

$$a = (s_y/s_x - s_{xy}/s_{xx})/(N/s_x - s_x/s_{xx})$$

$$a = (s_y * s_{xx} - s_{xy} * s_x)/(N * s_{xx} - s_x^2)$$

How to get best-fit parameters

$$a = (s_y * s_{xx} - s_{xy} * s_x)/(N * s_{xx} - s_x^2)$$

$$b = (N * s_{xy} - s_x * s_y)/(N * s_{xx} - s_x^2)$$

Just a few simple sums to
calculate, pretty straightforward!

Is that the full picture?

We also want to know three additional, very important quantities:

1) The uncertainty on a
2) The uncertainty on b
3) The uncertainty per degree of freedom of the fit (aka the goodness of the fit). For our linear fit we have two constraints so the variance of the fit, or goodness of fit, is:

$$\sigma^2 = \frac{1}{n-2}\sum (y_i - y(x_i))^2$$

And what do we do if the uncertainty on each point is not equal? Minimize the chi2 instead!

$$\chi^2(a,b) = \sum \left(\frac{y_i - a - bx_i}{\sigma_i} \right)^2$$

More complicated fits

And what do we do if the uncertainty on each point is not equal? Minimize the chi2 instead!

$$\chi^2(a,b) = \sum \left(\frac{y_i - a - bx_i}{\sigma_i} \right)^2$$

If the uncertainties on each point are constant, then minimizing the above gives our previous result. If not, it provides more weight in the fit to points with smaller error, and less weight in the fit to points with larger error

Using point-by-point errors

Uncertainties on parameters!

$$b = \frac{1}{S_{tt}} \sum_{i=0}^{n-1} \frac{t_i y_i}{\sigma_i} , \qquad a = \frac{S_y - S_x b}{S} \qquad \sigma_a^2 = \frac{1}{S}\left(1 + \frac{S_x^2}{S S_{tt}}\right) , \qquad \sigma_b^2 = \frac{1}{S_{tt}}$$

with

$$t_i = \frac{1}{\sigma_i}\left(x_i - \frac{S_x}{S}\right) , \qquad S_{tt} = \sum_{i=0}^{n-1} t_i^2 ,$$

$$S = \sum_{i=0}^{n-1} \frac{1}{\sigma_i^2} , \qquad S_x = \sum_{i=0}^{n-1} \frac{x_i}{\sigma_i^2} , \qquad S_y = \sum_{i=0}^{n-1} \frac{y_i}{\sigma_i^2}$$

And goodness of fit :

$$\chi^2(a,b)/ndof = \frac{1}{n-2} \sum \left(\frac{y_i - a - b x_i}{\sigma_i} \right)^2$$

Using point-by-point errors

And goodness of fit :

$$\chi^2(a,b)/ndof = \frac{1}{n-2}\sum\left(\frac{y_i - a - bx_i}{\sigma_i}\right)^2$$

If our data really follow a linear distribution AND we have estimated our errors correctly, we should get chi2/ndf values ~1. Either way, we can use the chi2 and n to calculate a p-value indicating our goodness of fit

Other types of fits

How do we fit other curves to our data? One obvious thing to do is to transform the data so that it is a straight line! For example:

$$y = Ae^{Bx} \rightarrow \ln y = \ln A + Bx$$

So if we fit ln(y) as a function of x, we can use the same tools as before, just remembering that B is the slope and our constant is ln(A)

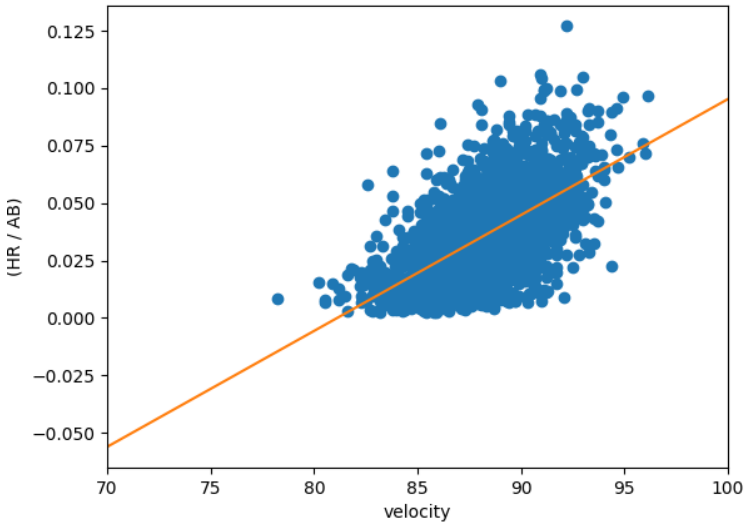
Uncertainties with alternative fits

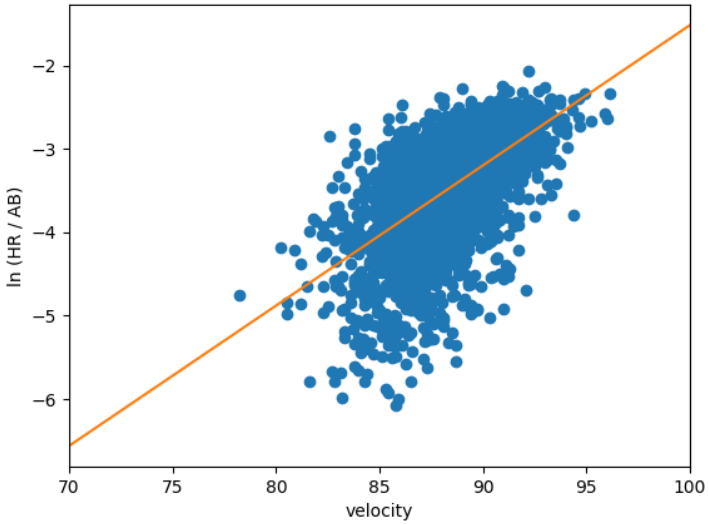
$$y = Ae^{Bx} \rightarrow \ln y = \ln A + Bx$$

$$y' = f(y) \rightarrow \sigma_{y'}^2 = \left(\frac{\partial f}{\partial y}\right)^2 \sigma_y^2$$

$$y' = \ln y = \ln A + Bx \rightarrow \sigma_{y'} = \frac{\sigma_y}{|y|}$$

Example from baseball

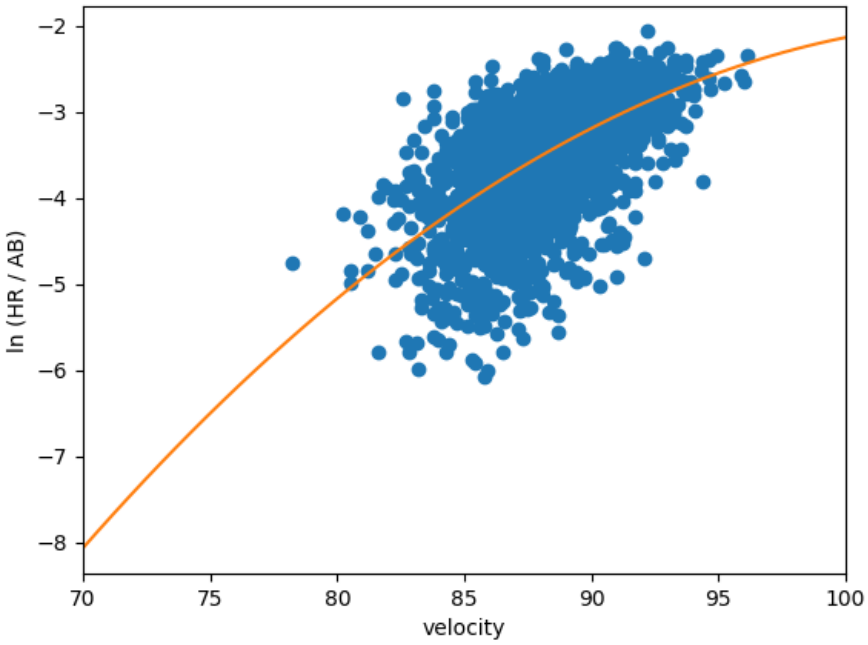




The harder you hit the ball, the more home runs you should hit. Is a linear fit really right? Exponential may be better (need to do goodness of fit checks to decide this!)

Also, quick Q: What is wrong with my plots? (Good check for you to make…)

Example from baseball



Try fitting a 2nd order polynomial to the log (equivalent to $y=Ae^{Bx+Cx**2}$

Aside from goodness of fit tests, how might you determine if this is a better fit?

How to fit a more general function?

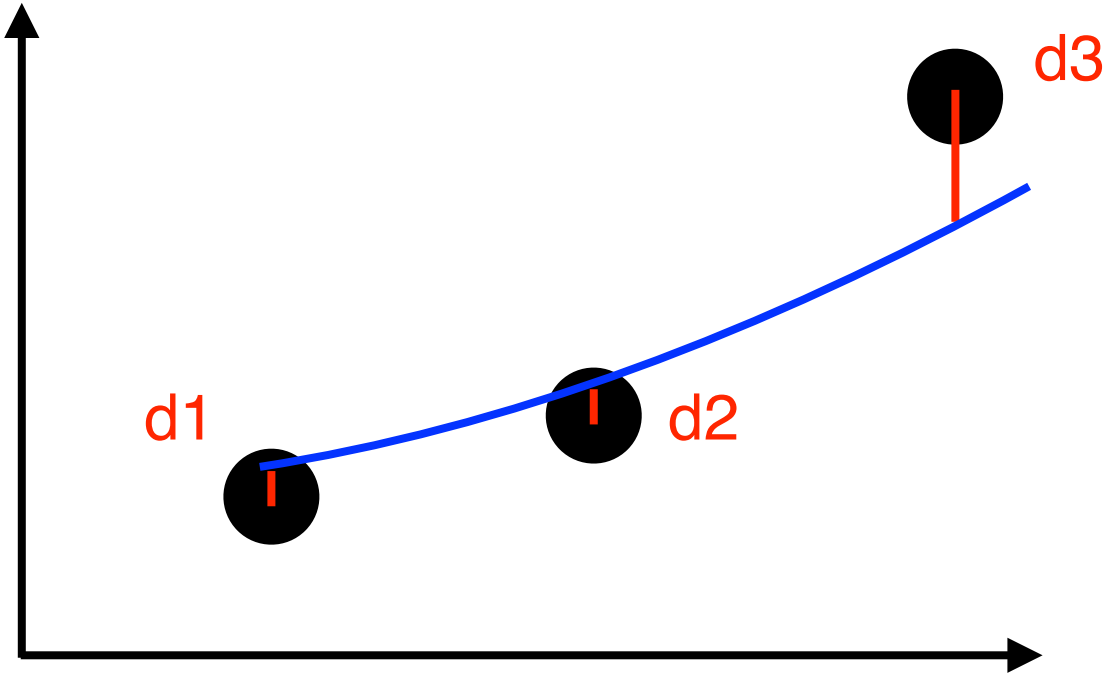
$$y = y(x; \vec{p})$$

y is a function of x, with vector of parameters **p**

$$\chi^2(\vec{p}) = \sum_i \left(\frac{y_i - y(x; \vec{p})}{\sigma_i} \right)^2$$

Minimize this for **p**

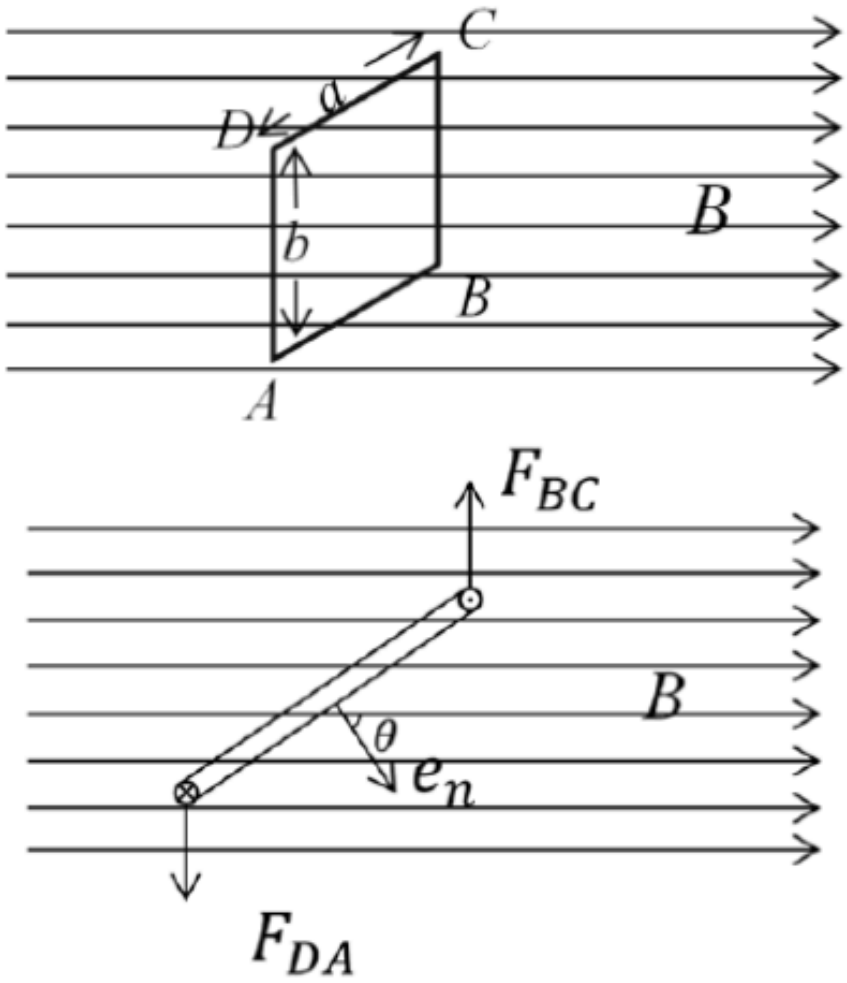
Still minimizing the total distance between prediction and observation



Switching topics…

Following 2110.04673

For the CS folks here… it's OK, don't panic :)



Current I running through the loop. Biot-Savart law tells us that there is a force F on each of the b length arms, with magnitude bIB, and these are in opposite direction so there is a torque τ on the coil. Perpendicular distance between the two forces is $a \sin\theta$, so

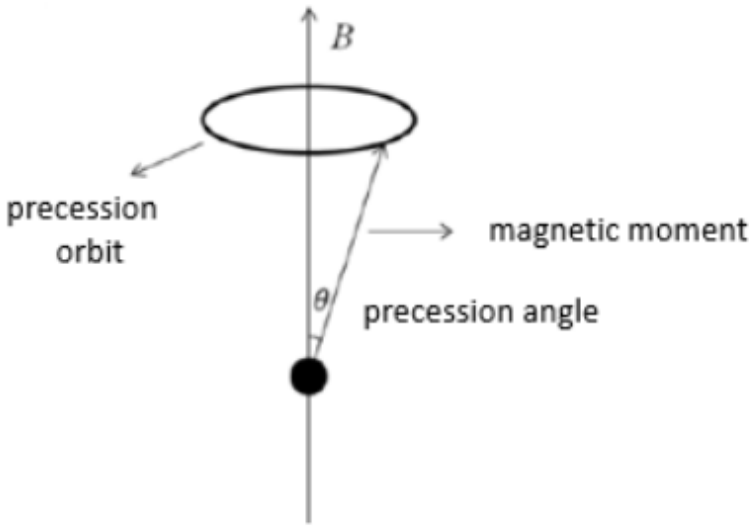
$$\tau = abIB \sin\theta = AIB \sin\theta$$

(A=ab)

$$|\tau| = IA\vec{e}_n \times \vec{B} = \vec{M} \times \vec{B}$$

$\vec{M} = IA\vec{e}_n$ with $\vec{e}_n$ the unit vector normal to the coil

How do we measure this in particle physics?



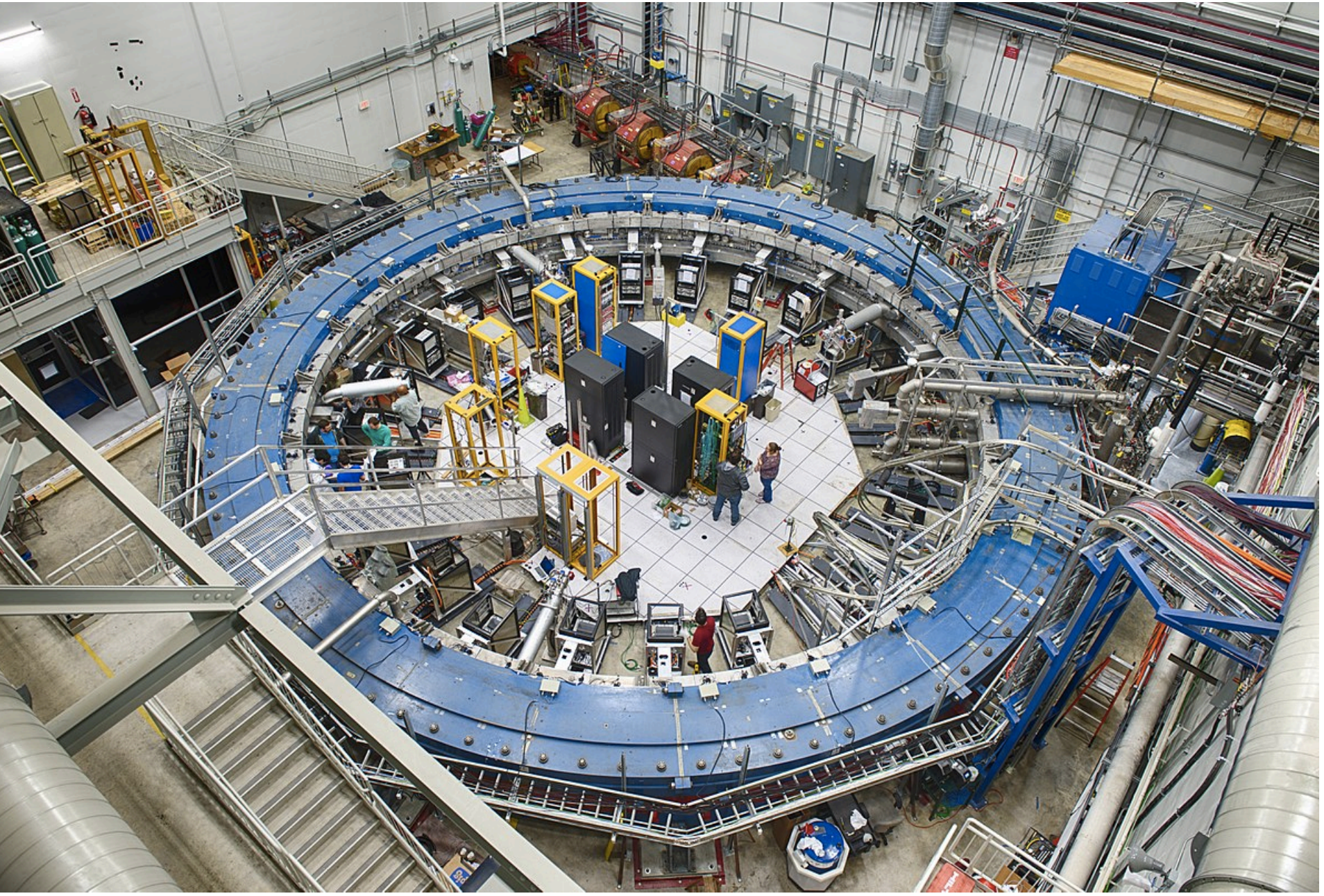
Analogous to a spinning top - spins rotates about the magnetic field

$$\frac{d\vec{S}}{dt} = \vec{L} = \vec{M} \times \vec{B} = g\frac{e}{2m}\vec{S} \times \vec{B}.$$

$$\omega = g\frac{e}{2m}B$$

Challenge - we are in the lab frame, not the rest frame of the muon!

g-2 at FNAL



How do we measure this, more fully?

Following 2110.04673

$$\frac{d\vec{S}}{dt} = (\vec{\omega}_c + \vec{\omega}_a) \times \vec{S},$$

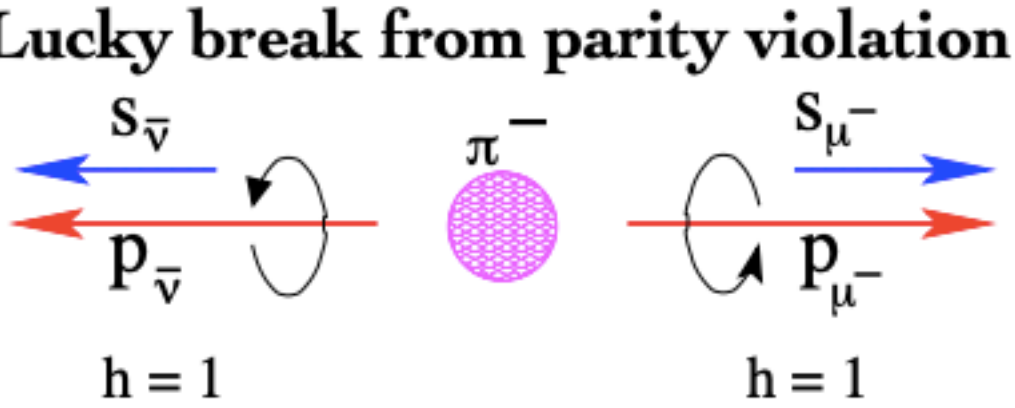
$$\vec{\omega}_c = -\frac{e}{\gamma m}\left(\vec{B} + \frac{\gamma^2}{\gamma^2 - 1}\frac{\vec{E} \times \vec{v}}{c^2}\right),$$

$$\vec{\omega}_a = -\frac{e}{m}\left[a\vec{B} - a\left(\frac{\gamma}{\gamma + 1}\frac{\vec{v} \cdot \vec{B}}{c^2}\right)\vec{v} + \left(a - \frac{1}{\gamma^2 - 1}\right)\frac{\vec{E} \times \vec{v}}{c^2}\right]$$

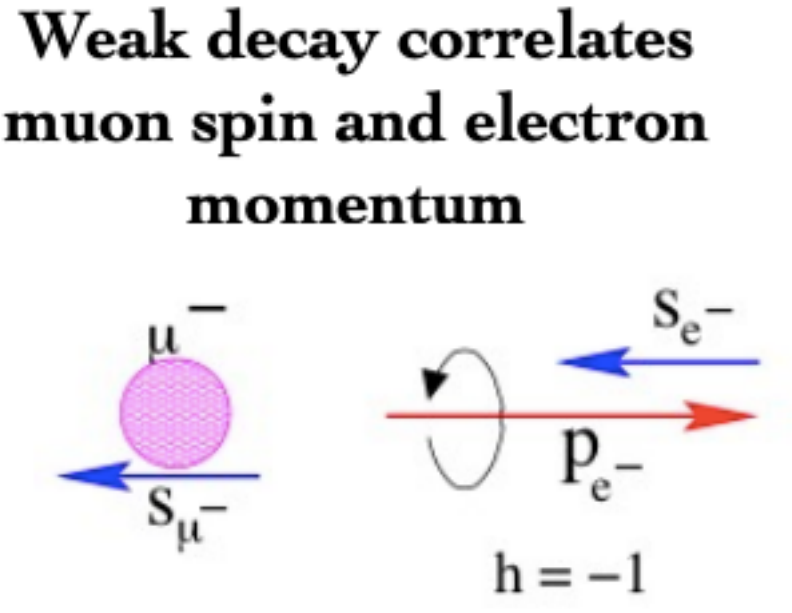
Pick the velocity giving "magic γ factor" to simplify ω_a. The other challenge is precisely measuring the **B** field and ensuring that it is uniform. Not easy! Use careful NMR probes and other tools

Muons come from pion decays. Thanks to parity violation, these are fully polarized!

https://indico.cern.ch/event/276476/contributions/1620139/attachments/501953/693162/g_minus_2_fewbuildins.pdf

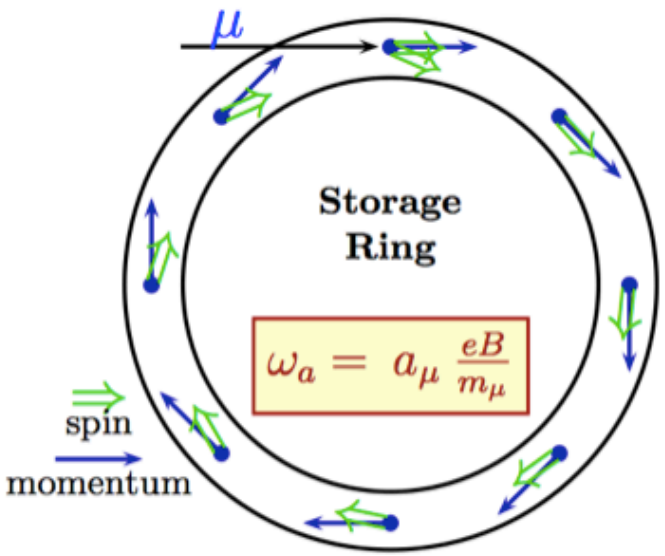


And there is a correlation between muon's spin and electron momentum from its decay



g-2 at FNAL

Spin precesses because $g_\mu \neq 2$

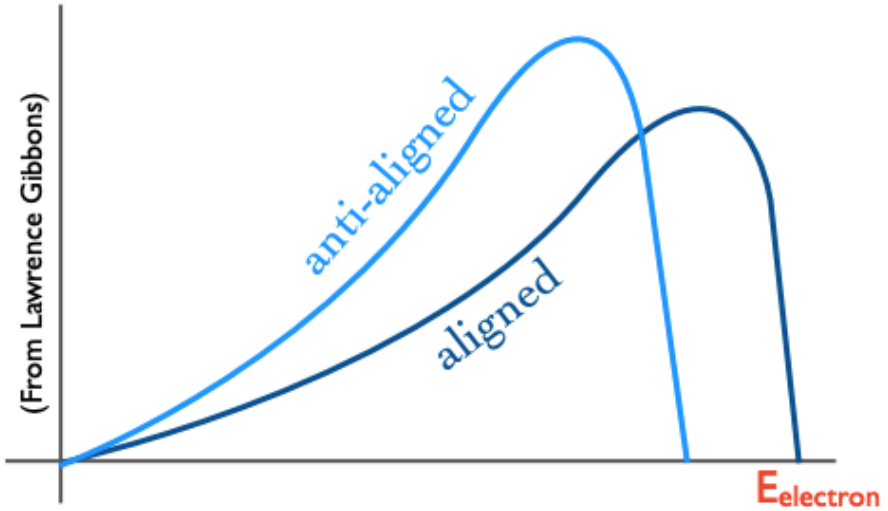


actual precession × 2

https://indico.cern.ch/event/276476/contributions/1620139/attachments/501953/693162/g_minus_2_fewbuildins.pdf

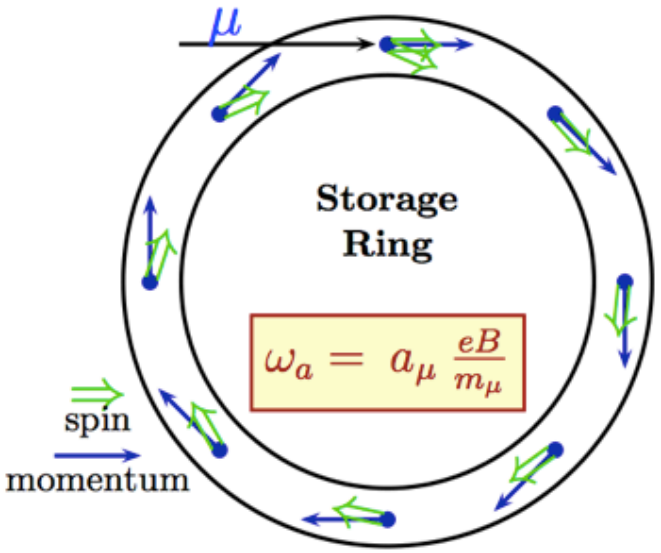
3. Measure Muon Spin Precession → 3. Measure Electron Energy

Harder electron spectrum when spin and momentum are aligned



g-2 at FNAL

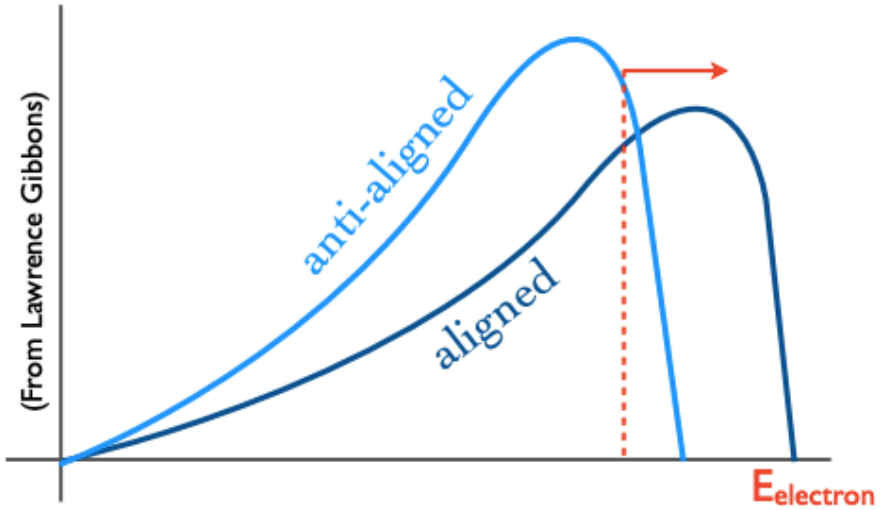
Spin precesses because $g_\mu \neq 2$



actual precession × 2

3. Measure Muon Spin Precession → 3. Measure Electron Energy

Harder electron spectrum when spin and momentum are aligned



Count N(e) above fixed threshold.
Oscillation rate ∝ a_μ

https://indico.cern.ch/event/276476/contributions/1620139/attachments/501953/693162/g_minus_2_fewbuildins.pdf

Not an easy thing to calculate at higher orders!

1805.01944

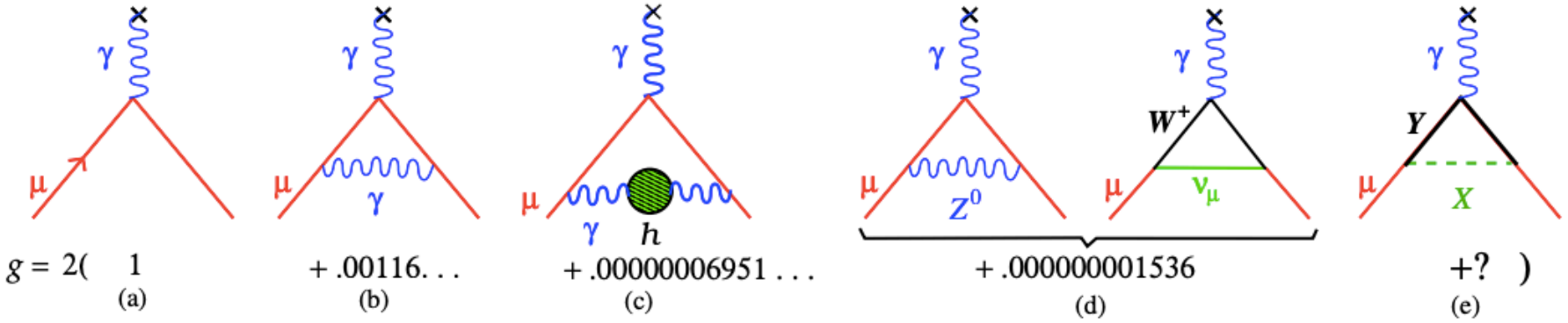


Figure 1: The Feynman graphs showing contributions to g for each of the Standard Model forces, ordered by size: (a) The Dirac interaction. (b) The lowest-order QED term $\alpha/2\pi$, which dominates the value of the anomaly. (c) The hadronic vacuum polarization contribution. (d) The lowest-order electroweak contributions. (The one-loop Higgs contribution is negligible.) (e) Potential contribution from new BSM particles X and Y.

Big challenge for evaluating the potential for BSM physics!
Measurement already accurate to 0.14 ppm!

3.2 The results of the measurements

The BNL E821 result is [2]

$$a_\mu^{\rm BNL} = 116592091(63) \times 10^{-11}. \tag{3.9}$$

The Fermilab result combined with the BNL result is [1]

$$a_\mu^{\rm Exp} = a_\mu^{\rm BNL+FNAL} = 116592061(41) \times 10^{-11}. \tag{3.10}$$

The SM predition

$$a_\mu^{\rm SM} = 116591810(43) \times 10^{-11}, \tag{3.11}$$

consists of the following contributions [3–6]

$$a_\mu^{\rm QED} = 116584718.9(1) \times 10^{-11}, \tag{3.12}$$

5 loops!

$$a_\mu^{\rm EW} = 153.6(1) \times 10^{-11}, \tag{3.13}$$

$$a_\mu^{\rm HVP,\ LO} = 6931(40) \times 10^{-11}, \tag{3.14}$$

$$a_\mu^{\rm HVP,\ NLO} = -98.3(7) \times 10^{-11}, \tag{3.15}$$

$$a_\mu^{\rm HVP,\ NNLO} = 12.4(1) \times 10^{-11}, \tag{3.16}$$

$$a_\mu^{\rm HLBL} + a_\mu^{\rm HLBL,\ NLO} = 92(18) \times 10^{-11}, \tag{3.17}$$

where the main uncertainties come from the hadronic contributions in HVP and HLBL. The deviation between the experiment and SM is

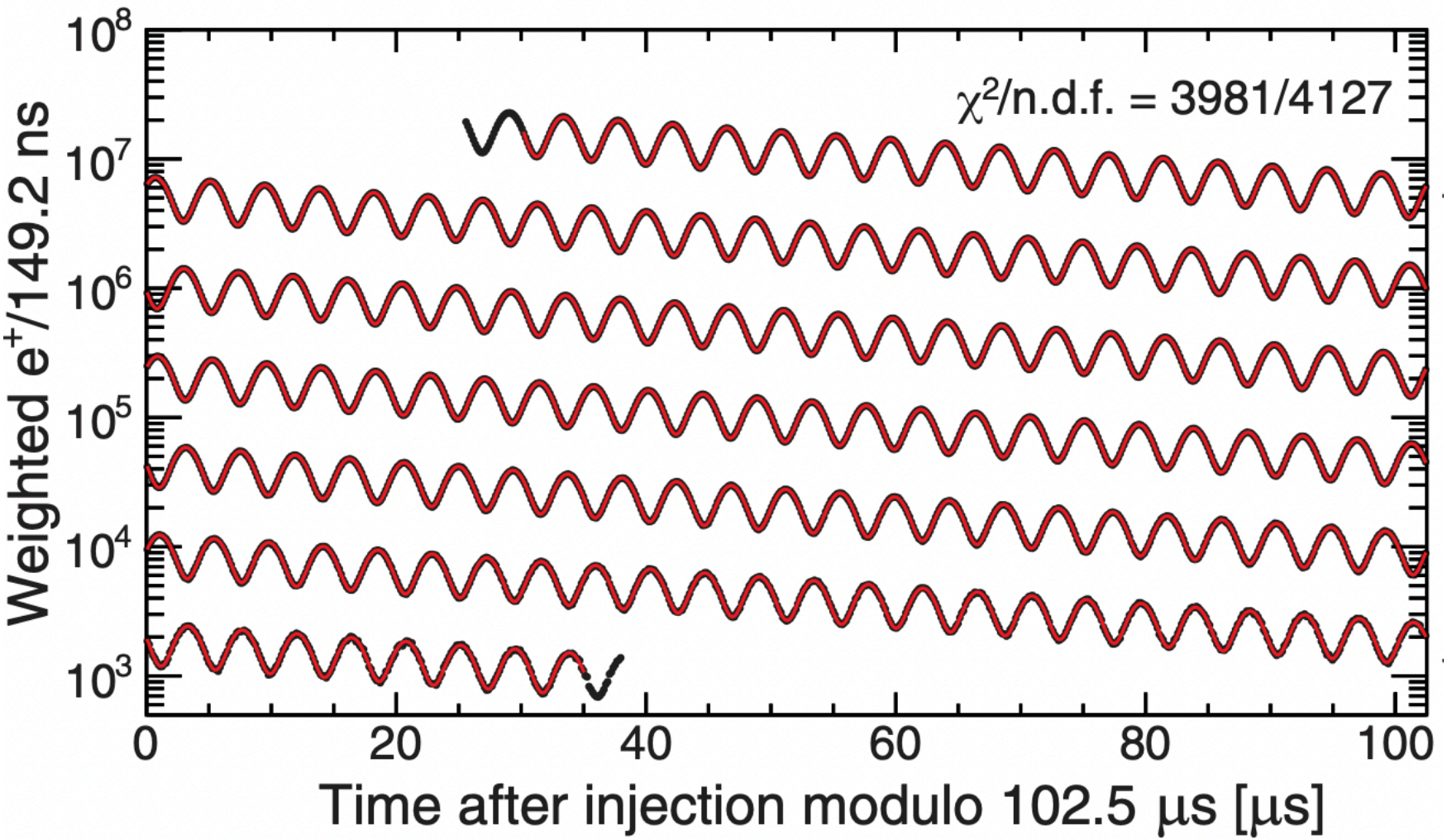
4.2σ !

$$a_\mu^{\rm Exp} - a_\mu^{\rm SM} = 251(59) \times 10^{-11}, \tag{3.18}$$

What are the fit results?

$$N(t) = N_0 \eta_N(t) e^{-t/\gamma\tau_\mu} \times \{1 + A\eta_A(t) \cos[\omega_a^m t + \varphi_0 + \eta_\phi(t)]\}$$

The number of measured decays oscillates as a cos term, but with an exponential term as muons decay!

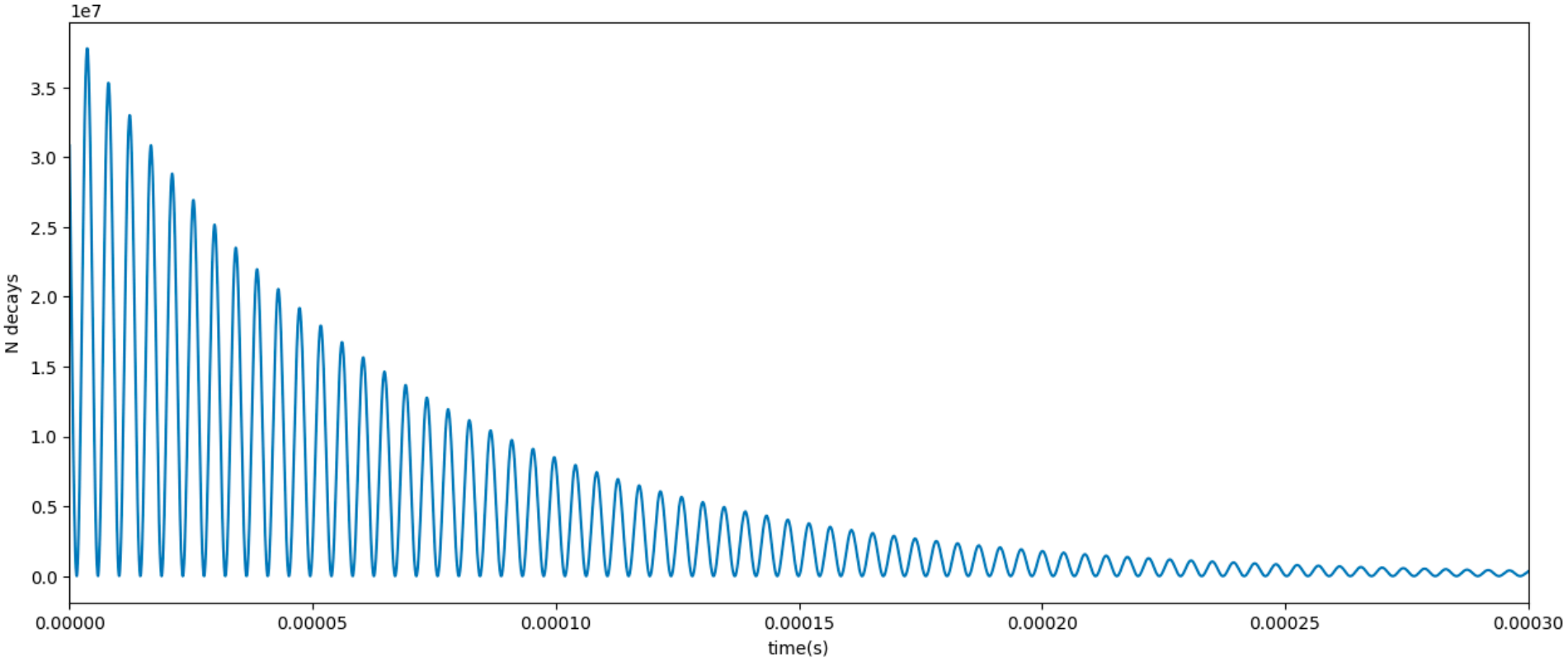


How to fit data like this?

2308.06230

I generated data with a simplified fit model (the actual analysis has a lot of corrections to these terms that we won't account for here!)

$$N(t) = N_0 e^{-t/\tau}(1 + cos(\omega t + \delta))$$



What is our χ^2?

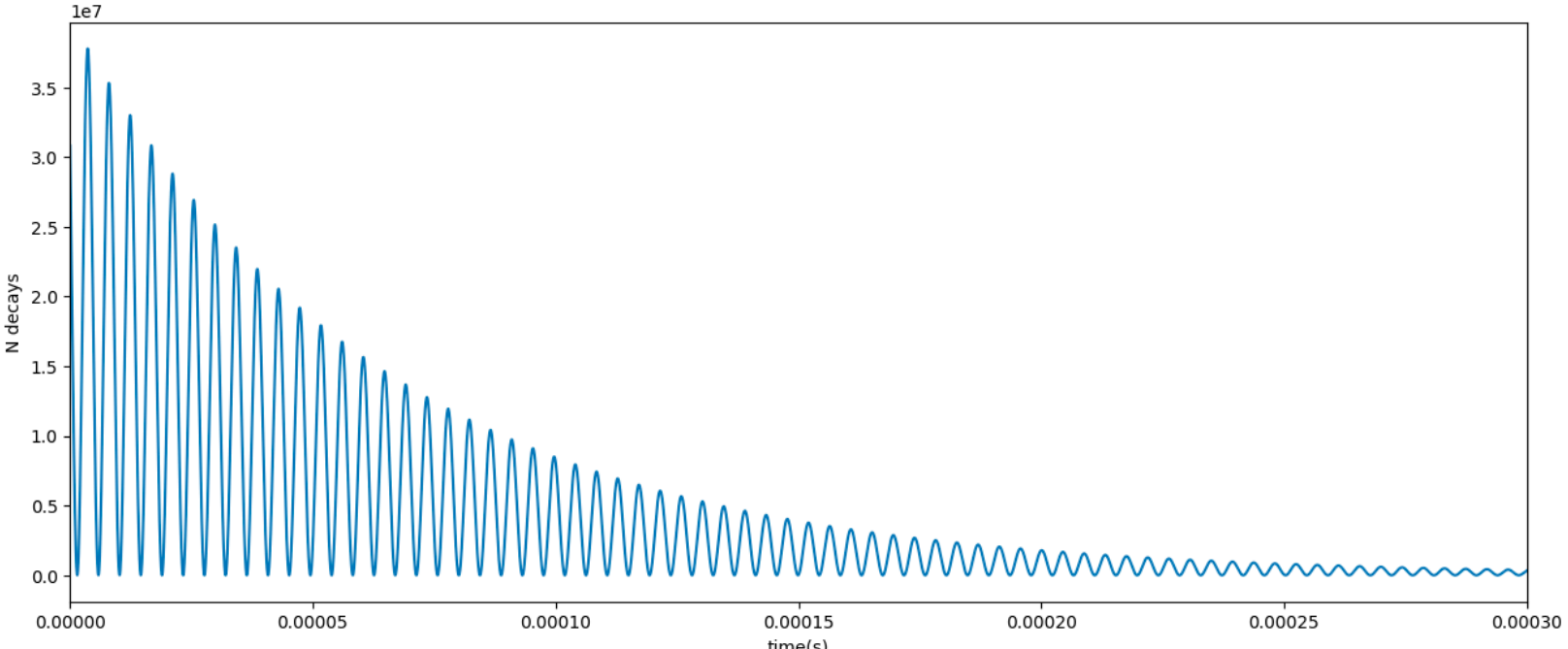
2308.06230

Use a $\chi^2(N_0, \omega, \delta) = \sum \frac{(y(t) - N(t))^2}{\sqrt{y(t)}^2} = \frac{(y(t) - N(t))^2}{y(t)} =$

What is in the denominator? It is the uncertainty on a count N, which from Poisson statistics goes as the sqrt(N)

$N(t) = N_0 e^{-t/\tau}(1 + cos(\omega t + \delta))$

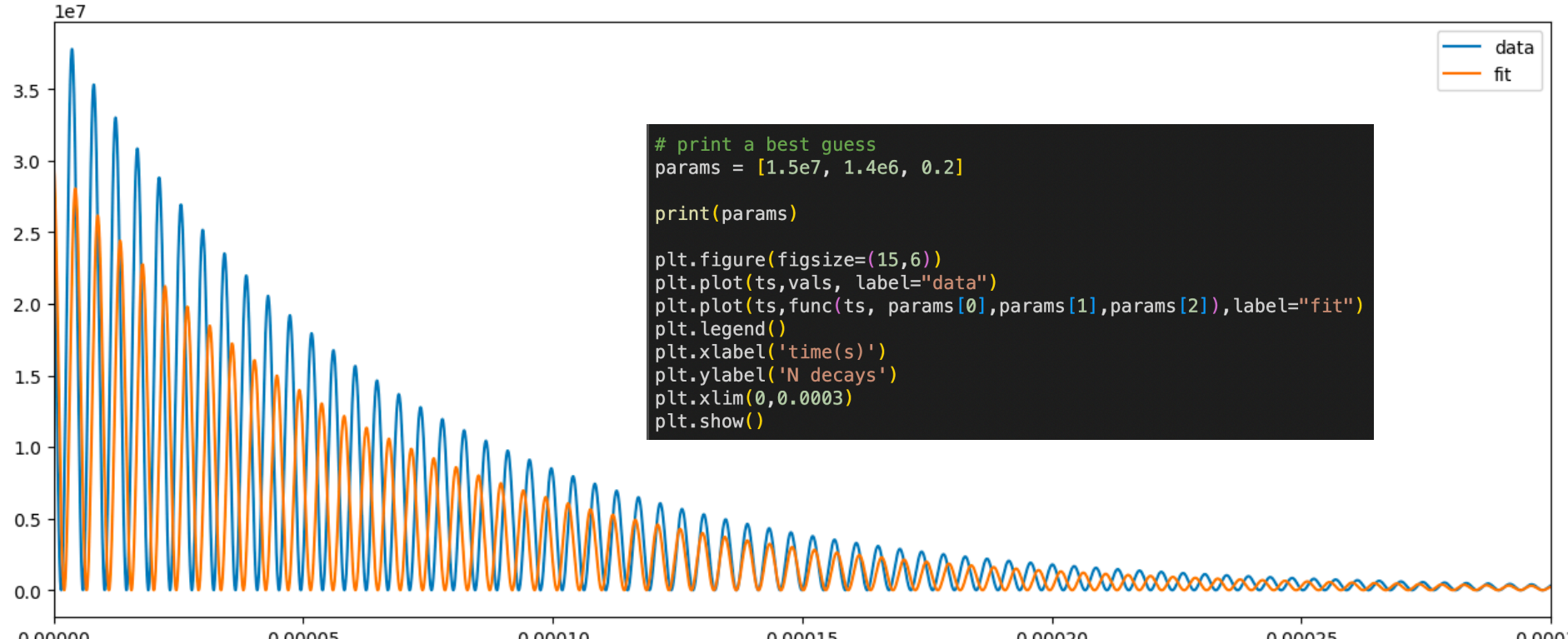
We want to minimize the χ^2 over all 3 parameters! (The lifetime τ is known and fixed)



So how do we move in this space?

Calculate gradient of the χ^2 - and move in the opposite direction! A few tricks: 1) We want to take small steps, 2) We need an initial guess that is close to the true fit values, 3) We want to take smaller and smaller steps as our fit progresses

Clearly not a great fit, but not a bad first guess



Calculating partial derivatives

Use a $\chi^2(N_0, \omega, \delta) = \sum \frac{(y(t) - N(t))^2}{y(t)}$

$N(t) = N_0 e^{-t/\tau}(1 + cos(\omega t + \delta))$

What is the gradient? Calculate partial derivatives!

$$\frac{\partial \chi^2}{\partial N_0} = \sum -2 \frac{(y(t) - N(t))(e^{-t/\tau}(1 + \cos(\omega t + \delta))}{y(t)}$$

$$\frac{\partial \chi^2}{\partial \omega} = \sum 2 \frac{(y(t) - N(t))(N_0 t e^{-t/\tau} \sin(\omega t + \delta)}{y(t)}$$

$$\frac{\partial \chi^2}{\partial \delta} = \sum 2 \frac{(y(t) - N(t))(N_0 e^{-t/\tau} \sin(\omega t + \delta)}{y(t)}$$

Calculating partial derivatives

Note - we can calculate these analytically! But we can also do this computationally (we'll get there in a bit!)

Use a $\chi^2(N_0, \omega, \delta) = \sum \frac{(y(t) - N(t))^2}{y(t)}$

$N(t) = N_0 e^{-t/\tau}(1 + cos(\omega t + \delta))$

What is the gradient? Calculate partial derivatives!

$$\frac{\partial \chi^2}{\partial N_0} = \sum -2 \frac{(y(t) - N(t))(e^{-t/\tau}(1 + \cos(\omega t + \delta))}{y(t)}$$

$$\frac{\partial \chi^2}{\partial \omega} = \sum 2 \frac{(y(t) - N(t))(N_0 t e^{-t/\tau} \sin(\omega t + \delta)}{y(t)}$$

$$\frac{\partial \chi^2}{\partial \delta} = \sum 2 \frac{(y(t) - N(t))(N_0 e^{-t/\tau} \sin(\omega t + \delta)}{y(t)}$$

Calculating partial derivatives


```
from numpy import sqrt
tau = 64.4e-6
def func(t,N0,omega,phase):
  return N0*np.exp(-t/tau)*(1+np.cos(omega*t+phase))

def calcChi2(xs,ys,params,reduced = True): ### params = N0, omega, phase
  chi2 = 0
  for x,y in zip(xs,ys):
    expected = func(x,params[0],params[1],params[2])
    if(y < 1e-6): chi2 += 9 ### placeholder
    else: chi2 += ((y-expected)**2)/y
  if (reduced): return chi2/len(xs)
  else: return chi2

def gradient(xs,ys,params):
  gradN0 = 0
  gradOmega = 0
  gradDelta = 0
  for x,y in zip(xs,ys):
    denom = y
    if (denom < 1e-6): denom = 9
    gradN0 += -2*(y-func(x,params[0],params[1],params[2]))*(np.exp(-x/tau)*(1+np.cos(params[1]*x+params[2])))/denom
    gradOmega += 2*(y-func(x,params[0],params[1],params[2]))*(params[0]*x*np.exp(-x/tau)*np.sin(params[1]*x+params[2]))/denom
    gradDelta += 2*(y-func(x,params[0],params[1],params[2]))*(params[0]*np.exp(-x/tau)*np.sin(params[1]*x+params[2]))/denom
  return(gradN0,gradOmega,gradDelta)

params = [N0,omega,phase]
chi2 = calcChi2(ts,vals,params)
print(chi2)
```

Lorentz-boosed lifetime of muon in lab frame (in s)

Our fit function!

Need to be careful if denominator gets too small - χ^2 method breaks down!

The challenge with these fits

Initial value, close to true values!

```
params = [2.0e7, 1.439e6, 1.0]
#params = [2.0e7, 1439337.1142358, 1.0] ### true values

print(params,calcChi2(ts,vals,params))

lr0 = 1e2
lr1 = 1e-4
lr2 = 1e-17
decay = 0.996

chi2s = []
N0s = []
omegas = []
phis = []
gs0 = []
gs1 = []
gs2 = []
for iter in range(1000):
  chi2 = calcChi2(ts,vals,params)
  if (iter%100 == 0): print(iter,chi2,params)
  gradients = gradient(ts,vals,params)
  chi2s.append(chi2)
  N0s.append(params[0])
  omegas.append(params[1])
  phis.append(params[2])
  gs0.append(gradients[0])
  gs1.append(gradients[1])
  gs2.append(gradients[2])

  #params[0] = params[0] - lr0*gradients[0]
  params[1] = params[1] - lr1*gradients[1]
  #params[2] = params[2] - lr2*gradients[2]
```

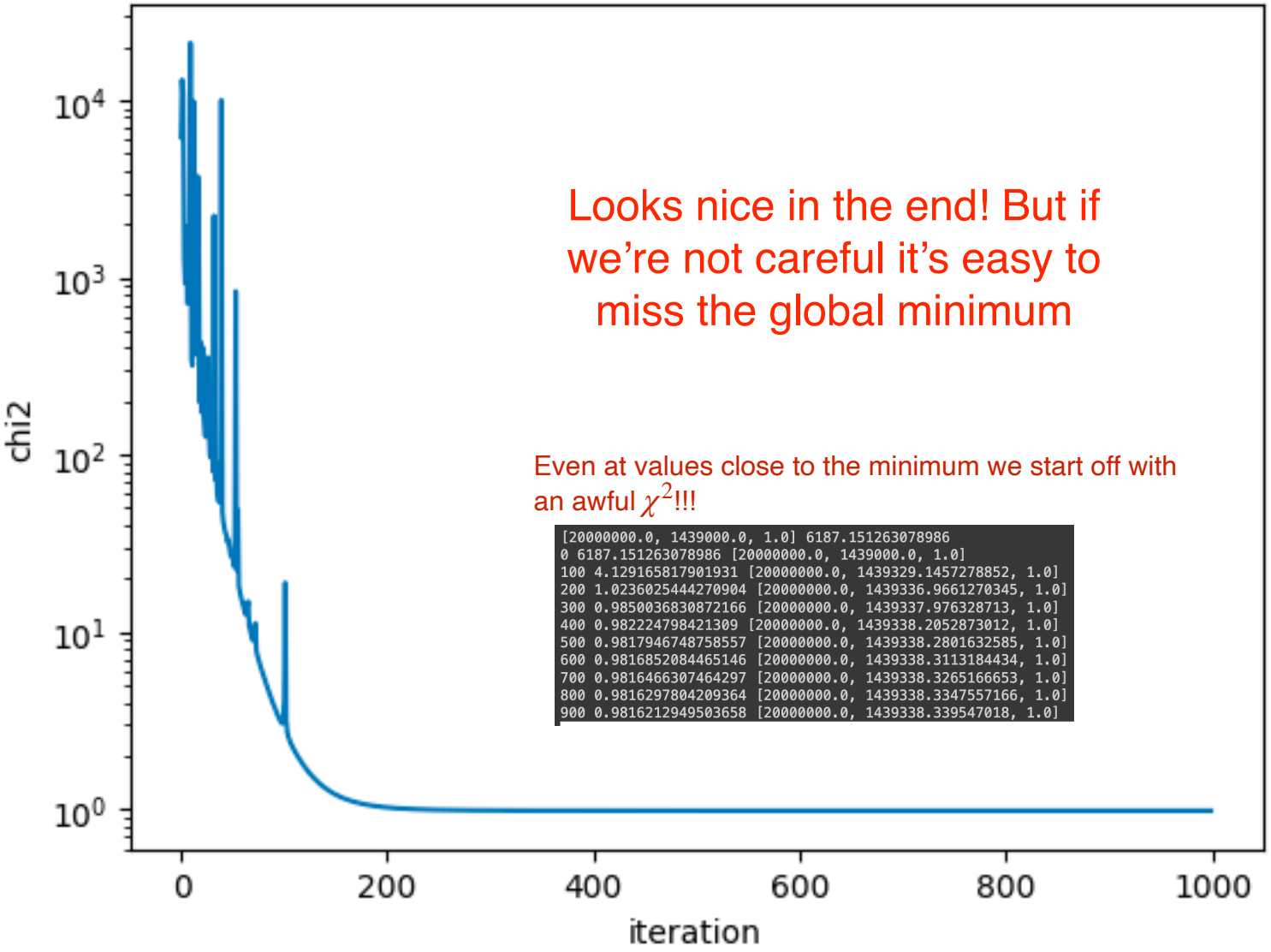
No set rule for these! Small but not too small that we never reach the global minimum

Fits with trig functions are challenging - we really should do a Fourier analysis instead (later chapters)! So let's fit just for ω

Make our steps smaller as we approach the minimum

```
lr0 = lr0*decay
lr1 = lr1*decay
lr2 = lr2*decay
```

Our fit result



Looks nice in the end! But if we're not careful it's easy to miss the global minimum

Even at values close to the minimum we start off with an awful χ^2!!!

```
[20000000.0, 1439000.0, 1.0] 6187.151263078986
0 6187.151263078986 [20000000.0, 1439000.0, 1.0]
100 4.129165817901931 [20000000.0, 1439329.1457278852, 1.0]
200 1.0236025444270904 [20000000.0, 1439336.9661270345, 1.0]
300 0.9850036830872166 [20000000.0, 1439337.976328713, 1.0]
400 0.982224798421309 [20000000.0, 1439338.2052873012, 1.0]
500 0.9817946748758557 [20000000.0, 1439338.2801632585, 1.0]
600 0.9816852084465146 [20000000.0, 1439338.3113184434, 1.0]
700 0.9816466307464297 [20000000.0, 1439338.3265166653, 1.0]
800 0.9816297804209364 [20000000.0, 1439338.3347557166, 1.0]
900 0.9816212949503658 [20000000.0, 1439338.339547018, 1.0]
```

What other kinds of fits can we do?

If we normalize our fit function, then for any set of parameters we know the probability that the fit function gave any one data point - it's the function itself! Call that the probability of the data point

Given a set of parameters for our normalized fit function, we can multiply the probabilities for each of our data points to give the joint probability (our likelihood, L) of all data!

Maximum likelihood fits

Multiplying probabilities is OK, but we end up with very large or small numbers. And it's hard to maximize the joint probability in many cases. Do make this easier we can take the log of the likelihood and maximize that instead probability with respect to all parameters (the natural log, ln, is a monotonically increasing function, so this still works!)

This is a cool way to answer the question: "For which parameter value does the observed data have the biggest probability?"

Let's give a concrete example

Fit data to function $f(x) = kx^2e^{cx}$, where k is a normalization term . Imagine that c is negative and our data of N points goes from 0 to infinity (in practice we will cut this off at some large value). The constant k is not a free parameter but is really k(c), as we must set k to normalize our fit function

Then $L = \Pi_i kx^2e^{cx}$, and $\ln L = \ln \Pi kx_i^2e^{cx_i}$

$\ln L = [\Sigma_i \ln(k) + \ln(x_i^2) + cx_i] = N \ln k + 2\Sigma_i ln(x_i) + c\Sigma_i x_i$

If we want to maximize lnL we can also maximize ln L/N

ln L/N = ln k + 2<ln(x)> + c<x>. Looks simple… but what is k?

We can estimate k(c) numerically for a given c, but thankfully Wikipedia gives us this analytical answer!

$$\int xe^{cx}\,dx = e^{cx}\left(\frac{cx-1}{c^2}\right) \qquad \text{for } c \neq 0;$$

$$\int x^2e^{cx}\,dx = e^{cx}\left(\frac{x^2}{c} - \frac{2x}{c^2} + \frac{2}{c^3}\right)$$

$$\int x^ne^{cx}\,dx = \frac{1}{c}x^ne^{cx} - \frac{n}{c}\int x^{n-1}e^{cx}\,dx$$

Let's normalize our fit function

$$\int_0^\infty kx^2e^{cx} = 0 - ke^0\left(0/c - 0/c^2 + 2/c^3\right) = -2k/c^3 = 1,$$

So k = -c³/2 and $f(x) = -\frac{c^3}{2}x^2e^{cx}$. Then:

ln L/N = ln (-c³/2) + 2<ln(x)> + c<x>.

$$\int xe^{cx}\,dx = e^{cx}\left(\frac{cx-1}{c^2}\right) \qquad \text{for } c \neq 0;$$

$$\int x^2e^{cx}\,dx = e^{cx}\left(\frac{x^2}{c} - \frac{2x}{c^2} + \frac{2}{c^3}\right)$$

$$\int x^ne^{cx}\,dx = \frac{1}{c}x^ne^{cx} - \frac{n}{c}\int x^{n-1}e^{cx}\,dx$$

Let's maximize ln(L/N)

$$\int_0^\infty kx^2e^{cx} = 0 - ke^0\left(0/c - 0/c^2 + 2/c^3\right) = -2k/c^3 = 1,$$

So k = -c³/2 and $f(x) = -\frac{c^3}{2}x^2e^{cx}$. Then:

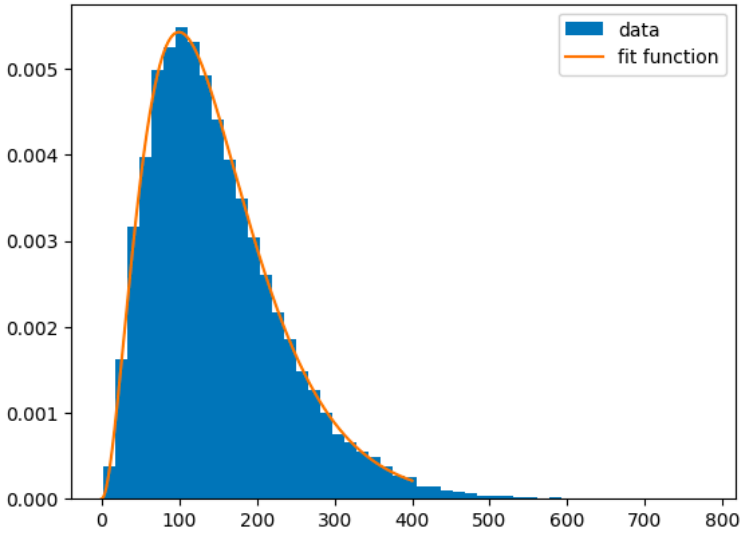
ln L/N = ln(-c³/2) + 2<ln(x)> + c<x>. Given a set of data, the only free parameter is c. How to maximize ln(L/N) for this data? The derivative with respect to c but must be zero!

$\frac{\partial(lnL/N)}{\partial c}$ = 3/c + <x> = 0 → c = -3/<x>, which is good, it's negative as we wanted! Is this going to minimize or maximize the probability? Take the second derivative:

$\frac{\partial^2(lnL/N)}{\partial c^2}$ = -3/c², which is negative, so it's a maximum!

Let's try it out


```
import matplotlib.pyplot as plt
import numpy as np
from google.colab import files
uploaded = files.upload()### first fit data
points = np.genfromtxt('fitdata.txt')
N = points.size
average = np.sum(points)/N
c_estimated = -3/average
print("c estimated = ",c_estimated)
min=0
max=400

def func_fixed(xs):
    return -c_estimated*c_estimated*c_estimated/(2)*xs*xs*np.exp(c_estimated*xs)


plt.hist(points,bins = 50, density = True,label="data")
plt.plot(np.linspace(min,max,1000), np.frompyfunc(func_fixed, 1, 1)(np.linspace(min,max,1000)),label="fit function")
plt.legend()
plt.show()
```



Cool! We didn't even need to do any work. Sometimes we can't get an analytical answer like this but then we can scan across parameters and find the values that maximize the log-likelihood (using some techniques from this and later chapters)

Let's try another one

Fit data to function $f(x) = kx^2 e^{-(ax^2+bx)}$, where k is a normalization term and a > 0 . Imagine that data of N points goes from -infinity to infinity (in practice we will cut this off at some values). The constant k is not a free parameter but is really k(a,b), as we must set k to normalize our fit function

This is a different way to write a Gaussian centered not at zero!

Then $L = \Pi_i k x_i^2 e^{-(ax_i^2+bx_i)}$, and $\ln L = \ln \Pi_i k x_i^2 e^{-(ax_i^2+bx_i)}$

$$\ln L = \Sigma_i \ln(k) + \ln(x_i^2) - ax_i^2 - bx_i =$$
$$N \ln k + 2N < \ln(x) > - aN < x^2 > - bN < x >$$

If we want to maximize lnL we can also maximize ln L/N
ln L/N = ln k + 2<ln(x)> -a<x²> - b<x>. What is k?

Let's normalize our second fit function

$$\int_0^\infty kx^2 e^{-(ax^2+bx)} = k\sqrt{\frac{\pi}{a}}e^{b^2/(4a)}$$

So k = $\sqrt{\frac{a}{\pi}}e^{-b^2/(4a)}$ and $f(x) = \sqrt{\frac{a}{\pi}}e^{-b^2/(4a)}x^2 e^{-(ax^2+bx)}$

We had:

ln L/N = ln k + 2<ln(x)> -a<x²> - b<x>

So ln L/N = 0.5(ln a - lnπ) + $\frac{-b^2}{4a}$ + 2<ln(x)> -a<x²> - b<x>

How do we maximize this? BOTH partial derivatives should be zero!

Let's normalize our second fit function

$$\ln L/N = 0.5(\ln a - \ln\pi) + \frac{-b^2}{4a} + 2\langle \ln(x) \rangle - a\langle x^2 \rangle - b\langle x \rangle$$

$$\frac{\partial lnL/N}{\partial a} = 0.5/a + \frac{b^2}{4a^2} - \langle x^2 \rangle = 0$$

$$\frac{\partial lnL/N}{\partial b} = \frac{-b}{2a} - \langle x \rangle = 0$$

Rewrite the first of this: $0.5a + b^2/4 - a^2\langle x^2 \rangle = 0$

From the second: $b = -2a\langle x \rangle$

Plus this back in: $0.5a + a^2\langle x \rangle^2 - a^2\langle x^2 \rangle = 0$. We don't want a=0

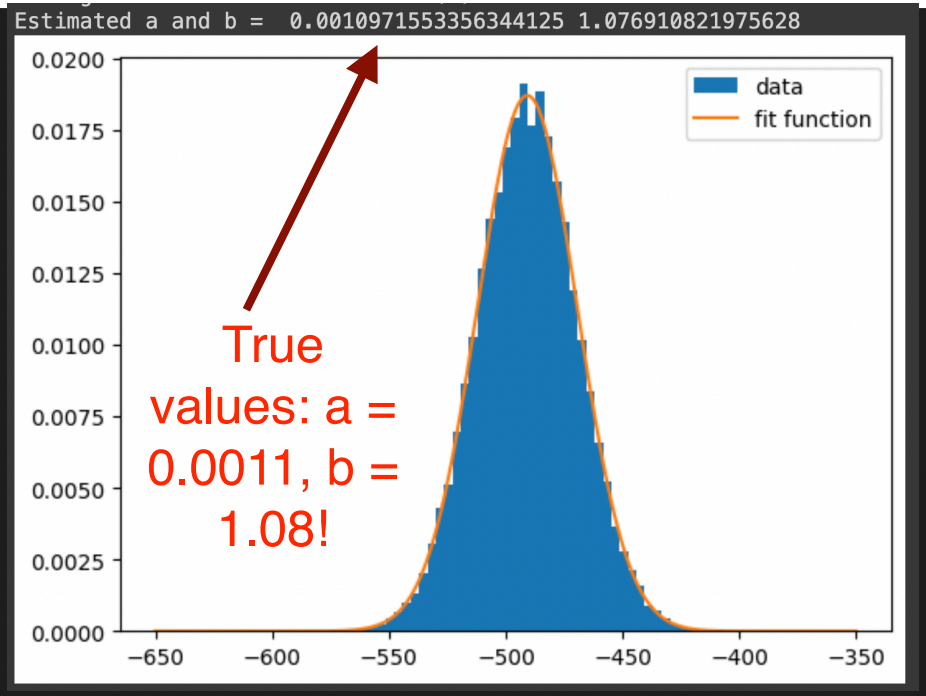
So $0.5 + a\langle x \rangle^2 - a\langle x^2 \rangle = 0$, or **$a = 1./(2*(\langle x^2 \rangle - \langle x \rangle^2))$**

And then **$b = -\langle x \rangle/(\langle x^2 \rangle - \langle x \rangle^2))$**

Trying out our second fit

```
import matplotlib.pyplot as plt
import numpy as np
from google.colab import files

def func_fixed(xs):
    k = np.sqrt(np.pi/a)*np.exp(b*b/(4*a))
    return (1./k)*np.exp(-(a*xs*xs + b*xs))

uploaded = files.upload()### first fit data
points = np.genfromtxt('fitdata3.txt')

min=-650
max=-350

N = points.size
avgx = np.sum(points)/N
avgx2 = np.sum(points*points)/N
a = 1./(2*(avgx2 - avgx*avgx))
b = -avgx/(avgx2 - avgx*avgx)
print("Estimated a and b = ",thisa,thisb)

plt.hist(points,bins = 50, density = True,label="data")
plt.plot(np.linspace(min,max,1000), np.frompyfunc(func_fixed, 1, 1)(np.linspace(min,max,1000)),label="fit function")
plt.legend()
plt.show()
```

```
Estimated a and b =  0.0010971553356344125 1.076910821975628
```



It's great when we can do this analytically! We won't have time to try fits where that isn't possible, for a second semester of this course? :)

Homework #2 (page 1/2)

https://classroom.github.com/a/D7PwP7rF

1) Write **pseudocode** to take a set of N (x,y) points, each with associated errors, and return the best fit straight line, errors on parameters, and chi2/ndf. Include any error checking that might be needed
2) Read the data from Hubble's original paper and plot it
3) Write actual code to determine the best-fit line and plot that on the same axis, assuming constant errors. How does your value for Hubble's constant (the slope) compare with current best estimates of ~70 (km/s)/Mpc? Plot that current best estimate on the same axis. Add legends and axis labels
4) Do the same for Problem 3.8 in the textbook. Then redo the problem assuming a constant fractional error of 2% on the y values in the data. Give the chi2/ndf and errors on a,b and h in this new version. How do your values of a,b and h change? Then redo this again assuming a constant fractional error of 1% on the values in the data added in quadrature with a 0.03V error. How do your chi2/ndf values change? How do the errors on your parameters change? Providing proper errors on your data is important!

Homework #2 (continued)

5) For PHYS510 only: On the course git page you will find a comma-separated file containing several thousand entries for baseball players over the past 5 years. Plot the batting average (the fraction of the time a batter succeeds) as a function of average launch angle (angle at which the ball leaves the baseball bat with respect to the horizontal). Fit this data to a second order polynomial and show your fit on the data. You can use numerical recipes you find on the web or elsewhere for this. What does your 2nd order fit show as the best average angle to succeed (ie to have a hit)? And what does it show as the drop in expected average if a ball is hit with a shift of 10 degrees from optimal?

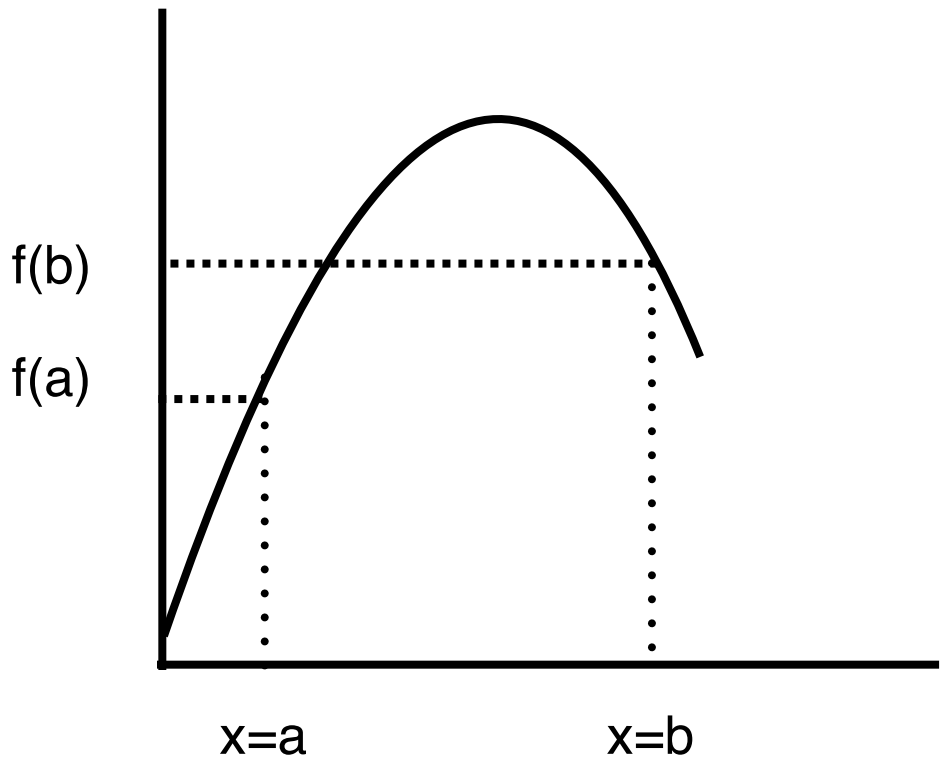
   If you are using python you may want to investigate the csv package. If you want to use colab, you will need to figure out how to get the data into colab. I suggest uploading to your private google drive:

```
from google.colab import drive
drive.mount('/content/drive')
!ls "/content/drive/My Drive/baseball_stats.csv” ## to change if file moves
f = open('/content/drive/My Drive//baseball_stats.csv”) # something like this
```

On to integrals!

The integral of f(x) = cos(x) = sin(x), easy we're all done, right? What happens if f(x) does not have an analytic answer? We must **evaluate it numerically!**

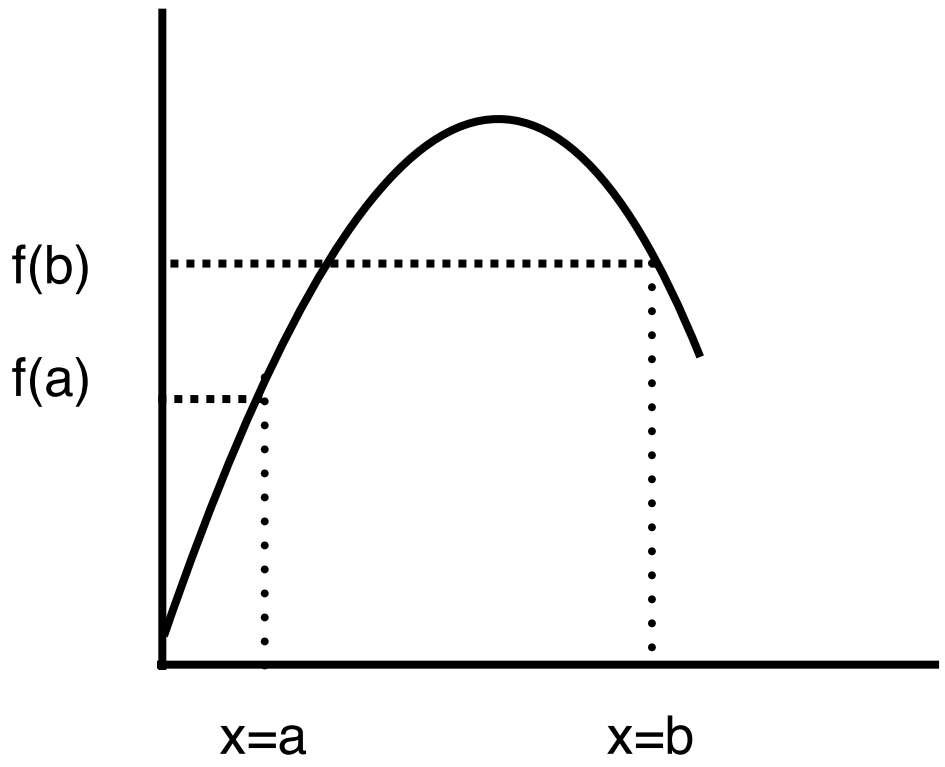
What's the simplest thing we can do?



Obviously not a really great approximation, but it's the lowest order thing we can do

$$\int_a^b f(x')dx' \sim f(b)(b-a) \sim f(a)(b-a)$$

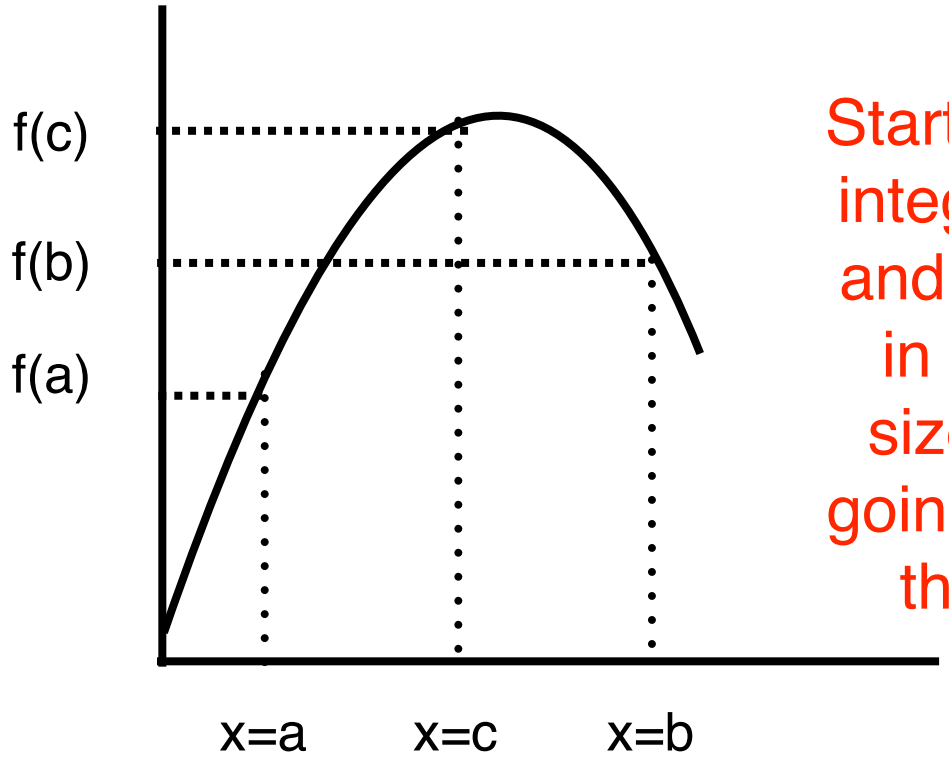
What's the simplest thing we can do?



Still not so great, but maybe slightly better. We use the average value of the function at the endpoints.

$$\int_a^b f(x')dx' \sim 0.5 * [f(b) + f(a)](b - a)$$

What's the simplest thing we can do?



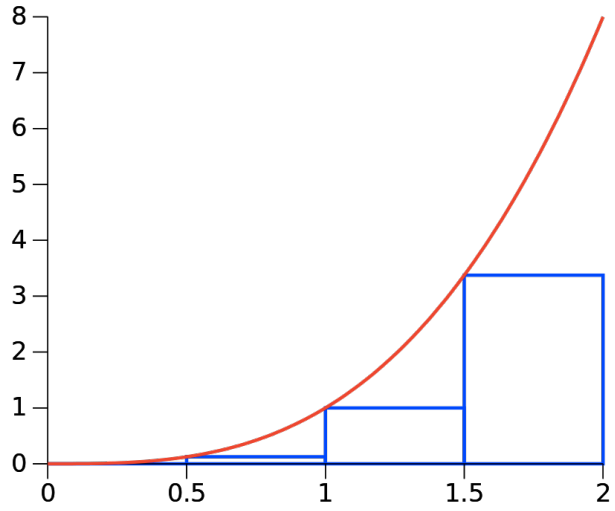
Start to break up the integral into smaller and smaller pieces, in the limit of the size of the pieces going to zero we get the exact value

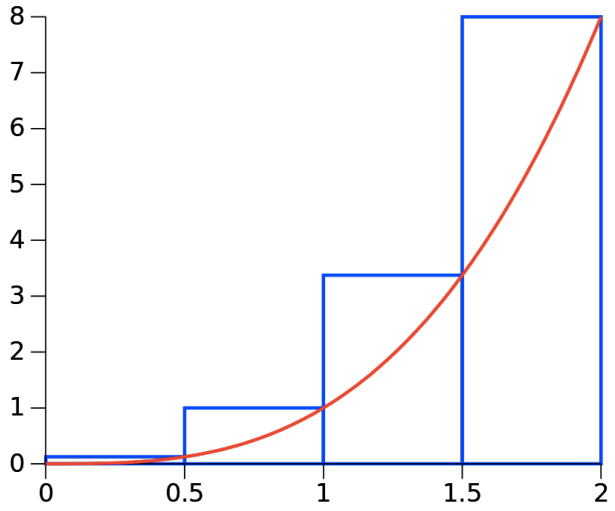
Why was this not so great? Because we know the function changes between b and a. So let's get back to the definition of an integral and break things up into smaller pieces

$$\int_a^b f(x')dx' \sim \frac{f(b)+f(a)+f(c)}{3}(b-a)$$

What's the simplest thing we can do?

https://en.wikipedia.org/wiki/Riemann_sum





These are two ways to integrate x^3 with 4 subdivisions. On the left is the "left" sum and on the right is the "right" sum. The difference is whether you use the function at the left endpoint or the right endpoint. Both are equally valid

$$\int_a^b f(x')dx' \sim h\sum_{i=2}^{N} f_i \sim \sum_{i=1}^{N-1} f_i$$

What is the error on this method?

Focus on the integral in one region between the (k-1)th and kth points. We Taylor expand the function of interest

$$\int_{x_{k-1}}^{x_k} f(x)dx = \int_{x_{k-1}}^{x_k} f(x_{k-1})dx + \int_{x_{k-1}}^{x_k} (x - x_{k-1})f'(x_{k-1})dx + \int_{x_{k-1}}^{x_k} \frac{1}{2}(x - x_{k-1})^2 f''(x_{k-1})dx + \ldots$$

Now we change variables, noting that the width of our rectangles is $x_k - x_{k-1}$ = to h (the width of one slice):

$$\int_{x_{k-1}}^{x_k} f(x)dx = f(x_{k-1})h + f'(x_{k-1})(\frac{1}{2}x_k^2 - \frac{1}{2}x_{k-1}^2 - x_k x_{k-1} + x_{k-1}^2) +$$
$$\frac{1}{2}f''(x_{k-1})(\frac{1}{3}x_k^3 - \frac{1}{3}x_{k-1}^3 - x_{k-1}x_k^2 + x_{k-1}^3 + x_{k-1}^2 x_k - x_{k-1}^3) + \ldots$$

$$\int_{x_{k-1}}^{x_k} f(x)dx = f(x_{k-1})h + f'(x_{k-1})(\frac{1}{2}x_k^2 + \frac{1}{2}x_{k-1}^2 - x_k x_{k-1}) +$$
$$\frac{1}{2}f''(x_{k-1})(\frac{1}{3}x_k^3 - \frac{1}{3}x_{k-1}^3 - x_{k-1}x_k^2 + x_{k-1}^2 x_k) + \ldots$$

What is the error on this method?

$$\int_{x_{k-1}}^{x_k} f(x)dx = f(x_{k-1})h + f'(x_{k-1})(\frac{1}{2}x_k^2 + \frac{1}{2}x_{k-1}^2 - x_k x_{k-1}) +$$
$$\frac{1}{2}f''(x_{k-1})(\frac{1}{3}x_k^3 - \frac{1}{3}x_{k-1}^3 - x_{k-1}x_k^2 + x_{k-1}^2 x_k)$$

$$\int_{x_{k-1}}^{x_k} f(x)dx = f(x_{k-1})h + \frac{f'(x_{k-1})}{2}(x_k - x_{k-1})^2 + \frac{f''(x_{k-1})}{6}(x_k - x_{k-1})^3 + \ldots$$

$$\int_{x_{k-1}}^{x_k} f(x)dx = f(x_{k-1})h + \frac{f'(x_{k-1})}{2}h^2 + \frac{f''(x_{k-1})}{6}h^3 + \ldots$$

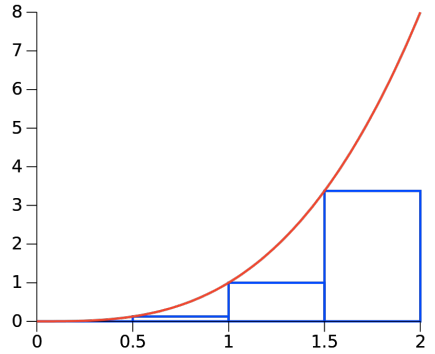
This is our estimate! So it is correct up to order h and the smaller h is the better we do

That was the left-hand rule

Recall that for left-hand sums we had:

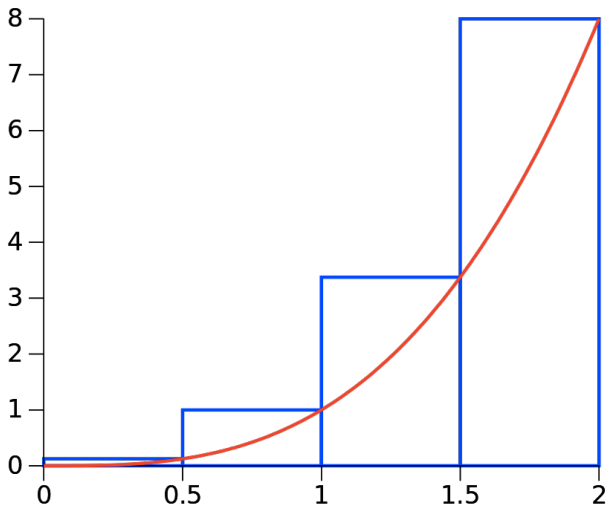
$$\int_{x_{k-1}}^{x_k} f(x)dx = f(x_{k-1})h + \frac{f'(x_{k-1})}{2}h^2 + \frac{f''(x_{k-1})}{6}h^3 + \ldots$$

If slope is positive, our rectangular rule misses a positive piece!



What about the right-hand rule….

If slope is positive, our rectangular rule now overestimates things! Let's see why…



Right hand sums

Expand about x_k now…

$$\int_{x_{k-1}}^{x_k} f(x)dx = \int_{x_{k-1}}^{x_k} f(x_k)dx + \int_{x_{k-1}}^{x_k} (x-x_k)f'(x_k)dx + \int_{x_{k-1}}^{x_k} \frac{1}{2}(x-x_k)^2 f''(x_k)dx + \ldots$$

$$\int_{x_{k-1}}^{x_k} f(x)dx = f(x_k)h + f'(x_k)(\frac{1}{2}x_k^2 - \frac{1}{2}x_{k-1}^2 - x_k^2 + x_k x_{k-1}) +$$

$$\frac{1}{2}f''(x_k)(\frac{1}{3}x_k^3 - \frac{1}{3}x_{k-1}^3 - x_{k-1}x_k^2 + x_{k-1}^3 + x_{k-1}^2 x_k - x_{k-1}^3) + \ldots$$

$$\int_{x_{k-1}}^{x_k} f(x)dx = f(x_k)h - \frac{f'(x_k)}{2}h^2 + \frac{f''(x_k)}{6}h^3 + \ldots$$

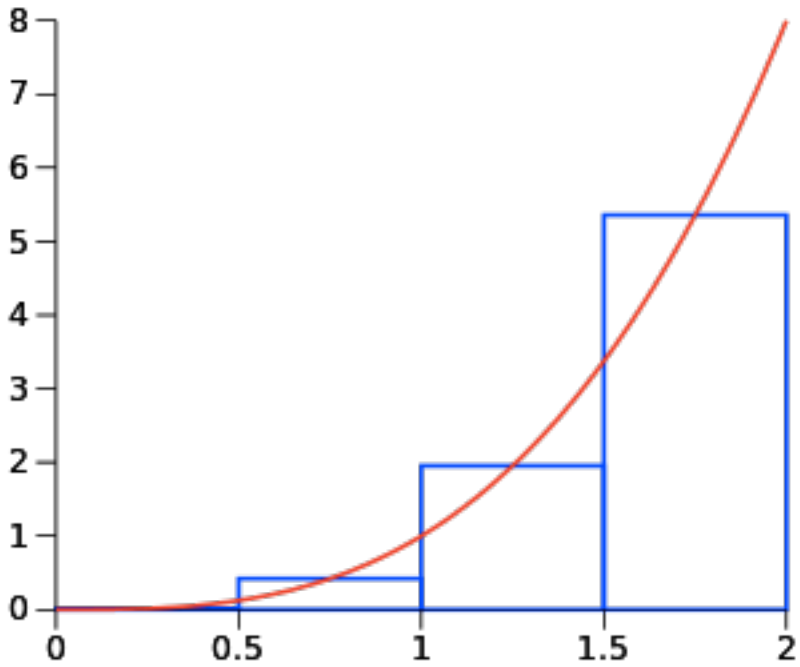
Also correct to order h but now with opposite sign in f’

Midpoint rule

Why do we take the value of the function at either end? Why not use the mid-point value, giving the midpoint rule?

Can consider this as the sum of a left-handed rectangular rule and a right-handed rectangular rule. The O(h) errors then cancel and the integral is exact to $O(h^2)$! That is nice because it means we don't need h to be as small —> less computation/time needed for the same error!

https://en.wikipedia.org/wiki/Riemann_sum



Midpoint rule h² term

$$\int_{x_{k-1}}^{x_k} f(x)dx = \int_{x_{k-1}}^{x_k} f(\frac{x_{k-1}+x_k}{2})dx + \int_{x_{k-1}}^{x_k} (x - \frac{x_{k-1}+x_k}{2}) f'(\frac{x_{k-1}+x_k}{2})dx + \ldots$$

$$\int_{x_{k-1}}^{x_k} f(x)dx = hf(\frac{x_{k-1}+x_k}{2}) + f'(\frac{x_{k-1}+x_k}{2}) \left(\frac{x_k^2}{2} - \frac{x_{k-1}^2}{2} + (\frac{x_{k-1}+x_k}{2})x_{k-1} - (\frac{x_{k-1}+x_k}{2})x_k \right) + \mathcal{O}h^3$$

$$\int_{x_{k-1}}^{x_k} f(x)dx = hf(\frac{x_{k-1}+x_k}{2}) + f'(\frac{x_{k-1}+x_k}{2}) \left(\frac{x_k^2}{2} - \frac{x_k^2}{2} - \frac{x_{k-1}^2}{2} + \frac{x_{k-1}^2}{2} + \frac{x_k x_{k-1}}{2} - \frac{x_k x_{k-1}}{2} \right) + \mathcal{O}h^3 \ldots$$

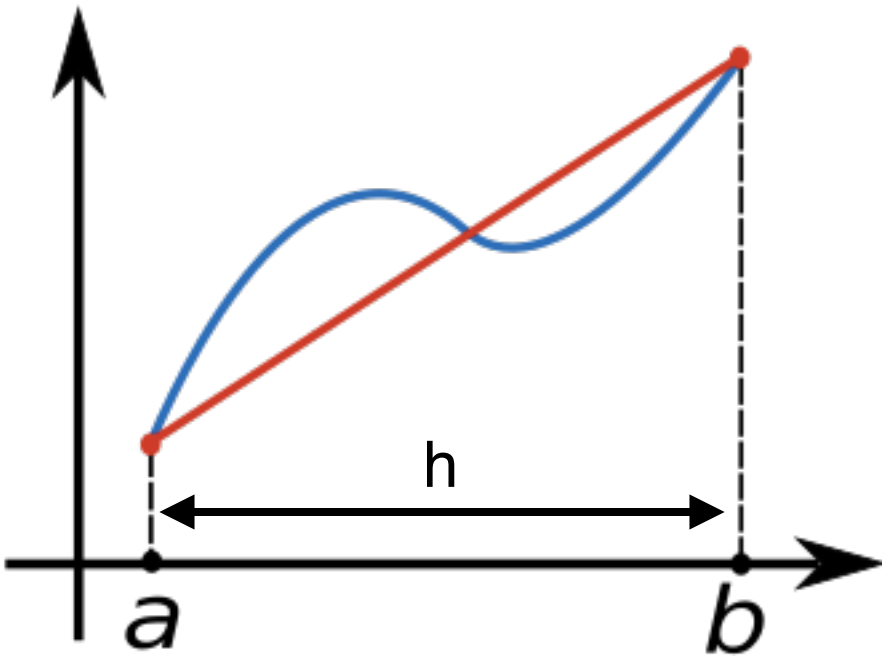
ZERO!

Exact to order h²!

Trapezoidal rule

https://en.wikipedia.org/wiki/Trapezoidal_rule

Instead of picking one value or another, why not use the average of our two (left and right) values?



Integral in one sub-division:

$$\int_a^b f(x)dx \sim \frac{f(a)+f(b)}{2}h$$

Trapezoidal rule

https://en.wikipedia.org/wiki/Trapezoidal_rule

Integral over all bins:

Error goes as h^2

$$\int_{x_0}^{x_N} f(x)dx \sim \frac{f(x_0)+f(x_1)}{2}h + \frac{f(x_1)+f(x_2)}{2}h + \frac{f(x_2)+f(x_3)}{2}h + \ldots.\frac{f(x_{N-1})+f_(x_N)}{2}h$$

With the exception of the first and last terms, each piece enters twice! One as the end of a trapezoid, and again as the beginning of another

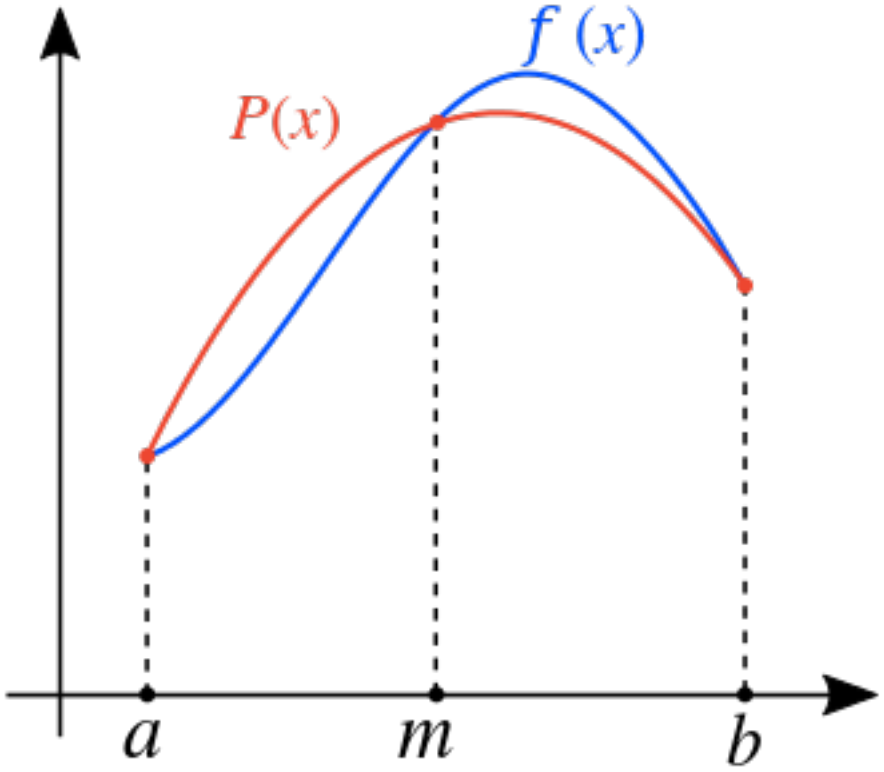
$$\int_{x_0}^{x_N} f(x)dx \sim h\left[\frac{f(0)+f(N)}{2} + \sum_{k=1}^{k=N-1} f(x_k)\right]$$

$$\int_{x_0}^{x_N} f(x)dx \sim h\left[\frac{f(0)+f(N)}{2} + \sum_{k=1}^{k=N-1} f(x_0+kh)\right]$$

Simpson's Rule

https://en.wikipedia.org/wiki/Simpson's_rule

f(x) is what we try to integrate. Can approximate it in any narrow window with a 2nd order polynomial (a quadratic), which requires having 3 points, not two, giving P(x), which we know how to integrate

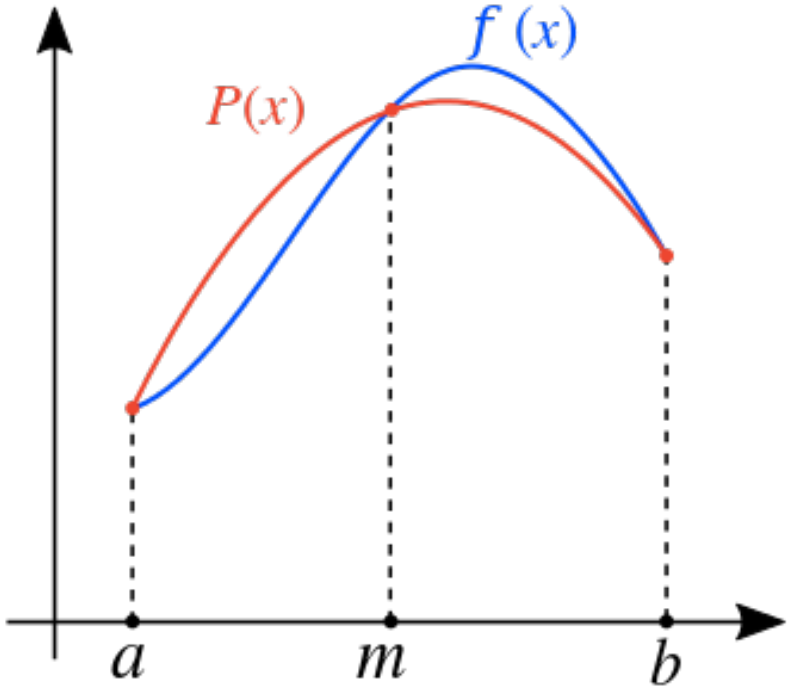


Clear here that the trapezoid rule would not give as good of a fit as P(x)

Simpson's Rule derivation

https://en.wikipedia.org/wiki/Simpson's_rule

Place our 3 points at x=-h, x=0 and x=+h (can then shift/ slide this window and results to any points later on)



$$f(x) = Ax^2 + Bx + C$$

$$f(-h) = Ah^2 - Bh + C$$

$$f(0) = C$$

$$f(h) = Ah^2 + Bh + C$$

$$f(0) = C$$

$$\frac{f(h) + f(-h) - 2C}{2h^2} = A \rightarrow \frac{f(h) + f(-h) - 2f(0)}{2h^2} = A$$

$$\frac{f(h) - f(-h)}{2h} = B$$

Simpson's Rule derivation

$$f(0) = C$$

$$\frac{f(h) + f(-h) - 2f(0)}{2h^2} = A$$

$$\frac{f(h) - f(-h)}{2h} = B$$

$$\int_{-h}^{h} (Ax^2 + Bx + C)dx = \frac{1}{3}Ax^3|_{-h}^{h} + \frac{1}{2}Bx^2|_{-h}^{h} + Cx|_{-h}^{h}$$

$$\int_{-h}^{h} (Ax^2 + Bx + C)dx = \frac{2}{3}Ah^3 + 2Ch$$

Simpson's Rule derivation

$$f(0) = C$$

$$\frac{f(h) + f(-h) - 2f(0)}{2h^2} = A$$

$$\frac{f(h) - f(-h)}{2h} = B$$

$$\int_{-h}^{h} (Ax^2 + Bx + C)dx = \frac{h}{3}\left(f(h) + f(-h) - 2f(0)\right) + 2hf(0)$$

$$\int_{-h}^{h} (Ax^2 + Bx + C)dx = \frac{h}{3}\left(f(h) + f(-h) + 4f(0)\right)$$

How to use Simpson's Rule

$$\int_{-h}^{h} (Ax^2 + Bx + C)dx = \frac{h}{3}\left(f(h) + f(-h) + 4f(0)\right)$$

$$\int_{a}^{b} f(x)dx =$$

If we want to evaluate the integral from a to b, we can break that into windows of size 2h and then apply Simpson's rule for each window

$$\frac{h}{3}([f(a) + 4 * f(a+h) + f(a+2h)] +$$ First window

$$[f(a+2h) + 4 * f(a+3h) + f(a+4h)] +$$ Second window

$$[f(a+4h) + 4 * f(a+5h) + f(a+6h)] + \ldots$$ Third window

$$[f(b-2h) + 4 * f(b-h) + f(b)]$$ Last window

How to use Simpson's Rule

$$\int_a^b f(x)dx =$$

Combine like terms

$$\int_a^b f(x)dx = \frac{h}{3}([f(a) + 4 * f(a+h) + 2 * f(a+2h) + 4 * f(a+3h) + 2 * f(a+4h) + ...+$$
$$2 * f(b-2h) + 4 * f(b-h) + f(b)]$$

This particular implementation requires an EVEN number of intervals, and that the function is evaluated at an ODD number of points (need three points on each!)

Correct to order h^4, error goes as h^5!

Thinking about how to improve on this

We know that we can do better as we make h smaller and smaller…. but the difference in predicted integral between evaluation with window size h and 2h (or 3h or nh) tells us something about the error on our method since we know how the error should scale! Similarly, if the integration result is stable when you shrink the window size, you are probably close to the correct value

Adaptive integration. And we can be clever by "reusing" our points from previous integration estimates

Romberg integration

If we stick to our trapezoidal rule, we know that our integral at any point I_1 goes as h^2. If we have an adaptive method and shrink h in half, then in the new integration (I_2) the error should go down by a factor of 4.

We're still evaluating the same integral I, so for some unknown c

$$I = I_1 + (ch_1)^2 = I_2 + (ch_2)^2$$

$$I = I_1 + (c*2*h_2)^2 = I_1 + 4(ch_2)^2 = I_2 + (ch_2)^2$$

$$I_2 - I_1 = 4(ch_2)^2 - (ch_2)^2 = 3(ch_2)^2$$

The error on I_2 is $(c*h_2)^2 = 1/3 * (I_2 - I_1)$

So the difference between the last stage of integration and this stage of integration tells us the size of the error we can expect

Romberg integration

The error on I_2 is $(c*h_2)^2 = 1/3 * (I_2 - I_1)$

So we can add this term back in to our integral:

$I_2 \rightarrow I_2 + 1/3 * (I_2 - I_1)$

And now our integral is exact to not only 2nd order but also 3rd order (odd orders cancel) and has $O(h^4)$ errors!

So we can say that

$I = I_N + O(h^4) = I_N + kh^4$

But what happens if we had evaluated this in the previous step with an h twice as big? Then

$I = I_{N-1} + O(h^4) = I_{N-1} + k(2h)^4 = I_{N-1} + 16k*h^4$

So we can equate the two and find that

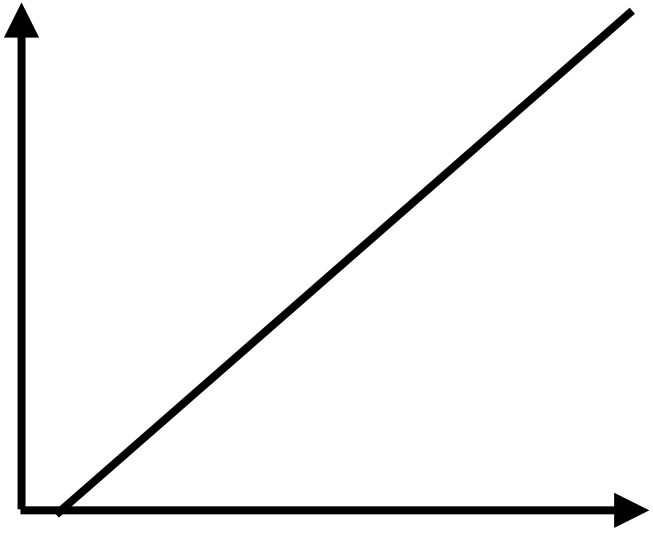
$I = I_N + O(h^4) = I_2 + 1/15*(I_2-I_1)$

And now our integral is exact to $O(h^6)$!

Romberg integration

We can repeat this process and at the Nth step we can do N evaluations of the integral. What we are doing in some sense is a series expansion in h. If the integral isn't doing anything crazy, we can see how it keeps changing as we change the size of h and use this to refine our prediction. This prediction then goes into the next piece of the series expansion. So we can gain a lot for every little computation time!

Higher-order methods



This straight line can be perfectly approximated by trapezoidal integrals. Similarly, a quadratic can be perfectly approximated by Simpson integrals! Why not use higher-order terms? We can, of course

Integrating to infinity

How do we integrate to infinity??? We can change variables. Instead of integrating x we can integrate $z=x/(1+x)$ or $x=z(1-z)$. Need to be careful of change of variables in the integral, of course.
And if we want to start the integral not at 0 but a, we can use $z=(x-a)/(1+x-a)$

There are lots of possibilities here

Let's look at some example code


```
### nice code from Sal's Computational Physics course, we can reuse these functions as necessary
import numpy as np

def simpson(f, a, b, n):
    """Approximates the definite integral of f from a to b by
    the composite Simpson's rule, using n subintervals.
    From http://en.wikipedia.org/wiki/Simpson's_rule
    """
    h = (b - a) / n
    i = np.arange(0,n)

    s = f(a) + f(b)
    s += 4 * np.sum( f( a + i[1::2] * h ) )
    s += 2 * np.sum( f( a + i[2:-1:2] * h ) )

    return s * h / 3

def trapezoid(f, a, b, n):
    """Approximates the definite integral of f from a to b by
    the composite trapezoidal rule, using n subintervals.
    From http://en.wikipedia.org/wiki/Trapezoidal_rule
    """
    h = (b - a) / n
    s = f(a) + f(b)
    i = np.arange(0,n)
    s += 2 * np.sum( f(a + i[1:] * h) )
    return s * h / 2
```

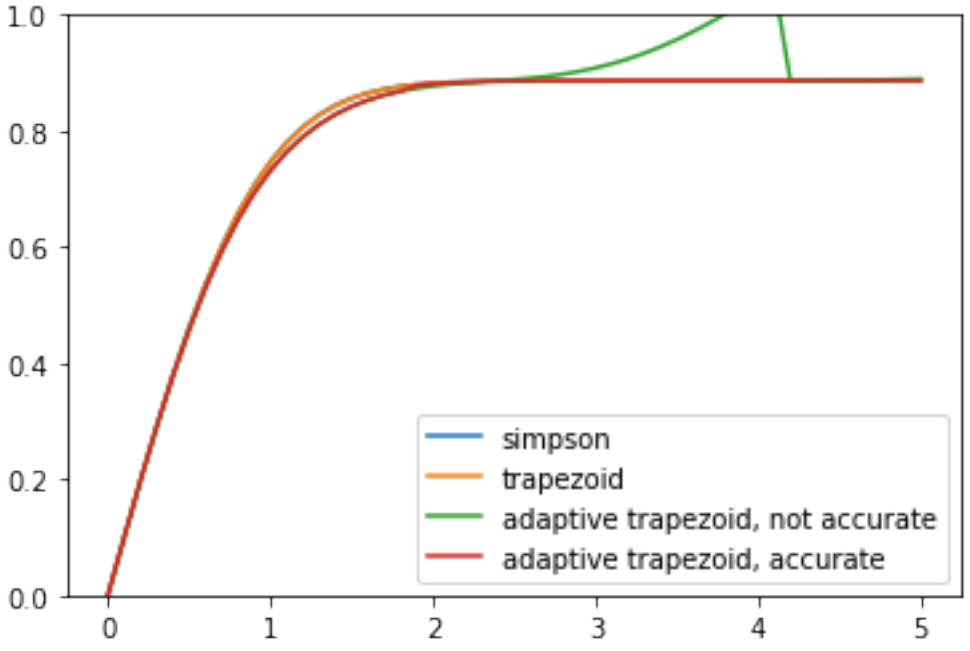
Let's look at some example code


```
def adaptive_trapezoid(f, a, b, acc, output=False):
    """
    Uses the adaptive trapezoidal method to compute the definite integral
    of f from a to b to desired accuracy acc.
    """
    old_s = np.inf
    h = b - a
    n = 1
    s = (f(a) + f(b)) * 0.5
    if output == True :
        print ("N = " + str(n+1) + ",  Integral = " + str( h*s ))
    while abs(h * (old_s - s*0.5)) > acc :
        old_s = s
        for i in np.arange(n) :
            s += f(a + (i + 0.5) * h)
        n *= 2.
        h *= 0.5
        if output == True :
            print ("N = " + str(n) + ",  Integral = " + str( h*s ))
    return h * s
```

Gaussian error function, Exercise 5.3


```
### Exercise 5.3 from the book, this is the integral of a Gaussian. Let's check with different methods
from numpy import exp
from numpy import linspace
from pylab import plot,ylim,show,legend

xmin = 0.0 ### min start
xmax = 5.0 ### max we integrate out to
N = 10 ### steps for integration, small number!
steps=50 ### steps to draw, not steps for integration

def f(t):
  return exp(-t*t)

xpoints = linspace(xmin,xmax,steps+1)
ypoints_simpson = []
ypoints_trapezoid = []
ypoints_adaptive_low = []
ypoints_adaptive_high = []
accuracy_low = 0.5
accuracy_high = 0.05
for x in xpoints:
  ypoints_simpson.append(simpson(f,xmin,x,N))
  ypoints_trapezoid.append(trapezoid(f,xmin,x,N))
  ypoints_adaptive_low.append(adaptive_trapezoid(f,xmin,x,accuracy_low))
  ypoints_adaptive_high.append(adaptive_trapezoid(f,xmin,x,accuracy_high))

plot(xpoints,ypoints_simpson,label="simpson")
plot(xpoints,ypoints_trapezoid,label="trapezoid")
plot(xpoints,ypoints_adaptive_low,label="adaptive trapezoid, not accurate")
plot(xpoints,ypoints_adaptive_high,label="adaptive trapezoid, accurate")
ylim(0.0,1.0)
legend()
show()
```

Gaussian error function, Exercise 5.3



Exercise 5.9, Debye's theory of solids

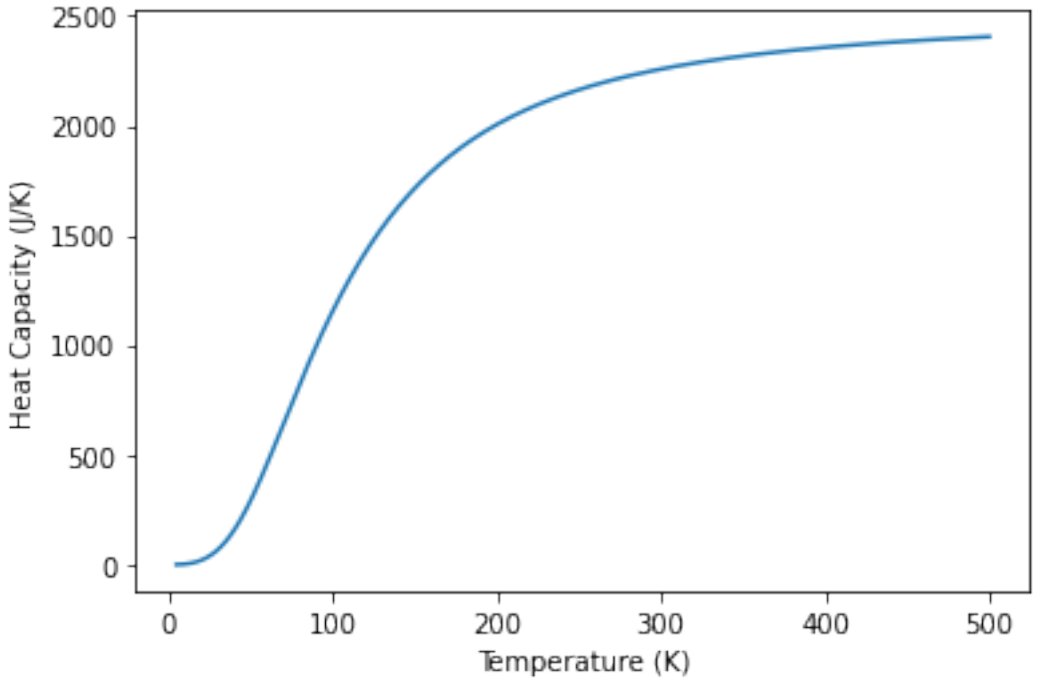
$$C_V = 9V\rho k_B \left(\frac{T}{\theta_D}\right)^3 \int_0^{\theta_D/T} \frac{x^4 e^x}{(e^x - 1)^2} dx$$

Plot the heat capacity for 1000 cubic centimeters of aluminum from 5K to 500 K

```
### Exercise 5.9
from math import exp,expm1
from pylab import plot,show,xlabel,ylabel

def simpson_simple(f, a, b, n):
  h = (b - a) / n
  s1 = 0.0
  for k in range(1,n,2):
    s1 += f(a+k*h)
  s2 = 0.0
  for k in range(2,n,2):
    s2 += f(a+k*h)

  s = f(a) + f(b)
  s += 4 * s1
  s += 2 * s2
  return s * h / 3


V = 0.001 ### Volume in cubic meters
rho = 6.022e28 ### Number density of aluminum
thetaD = 428.0 ### Debye temperature of aluminum
kB = 1.38065e-23 ### Boltzmann's constant
N = 500

# Integrand
def f(x):
  return(x**4)*exp(x)/(expm1(x)**2) ### expm1 is useful here for small x!
```

```
### Function to compute cv, using simpson
def cv(T):
  a = 0.000001 ### can't set to zero, or we divide by zero
  b = thetaD/T
  return 9*V*rho*kB*((T/thetaD)**3)*simpson_simple(f,a,b,N)

### main program, and plotting
tpoints = []
cpoints = []

for T in range(5,501):
  tpoints.append(T)
  cpoints.append(cv(T))

plot(tpoints,cpoints)
xlabel("Temperature (K)")
ylabel("Heat Capacity (J/K)")
show()
```

Exercise 5.9, Debye's theory of solids



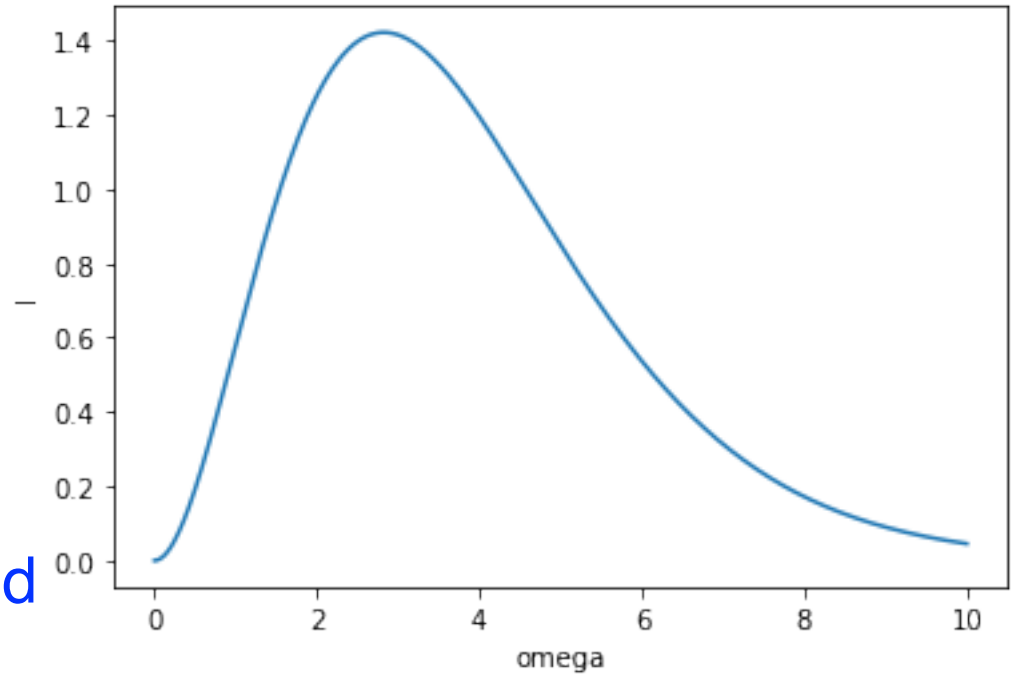
Let's discuss expm1, why integral didn't start from zero, and why we needed to redefine how we apply Simpson for integration

Exercise 5.12, Stefan-Boltzmann constant

$$I(\omega) = \frac{\hbar}{4\pi^2 c^2} \frac{\omega^3}{e^{\hbar\omega/k_B T} - 1}$$

According to Planck's theory, this is the amount of energy per second emitted by a blackbody object in the frequency range ω to ω+dω

```
# Exercise 5.12
from math import exp,expm1,pow
from pylab import plot,show,xlabel,ylabel

def funcI(omega):
  return pow(omega,3)/(expm1(omega))

N=500
xpoints = linspace(0.000001,10,N)
ypoints = []
for x in xpoints:
  ypoints.append(funcI(x))

plot(xpoints,ypoints)
xlabel("omega")
ylabel("I")
show()
```



Goes to zero for low and high frequencies (no ultraviolet catastrophe!)

Exercise 5.12, Stefan-Boltzmann constant

$$I(\omega) = \frac{\hbar}{4\pi^2 c^2} \frac{\omega^3}{e^{\hbar\omega/k_B T} - 1}$$

Rate that the object emits blackbody radiation:

$$W = \int_0^\infty I(\omega) d\omega = \int_0^\infty \frac{\hbar}{4\pi^2 c^2} \frac{\omega^3}{e^{\hbar\omega/k_B T} - 1} d\omega$$

$$x = \hbar\omega/k_B T, dx = \hbar/k_B T d\omega$$

$$W = \int_0^\infty \frac{\hbar}{4\pi^2 c^2} \frac{(x k_B T/\hbar)^3}{e^x - 1} k_B T/\hbar dx$$

$$W = \frac{k_B^4 T^4}{4\pi^2 c^2 \hbar^3} \int_0^\infty \frac{x^3}{e^x - 1} dx$$

Exercise 5.12, Stefan-Boltzmann constant


```
# Exercise 5.12 continued
from math import exp,expm1,pow
from pylab import plot,show,xlabel,ylabel

def funcI(omega):
  return pow(omega,3)/(expm1(omega))

def simpson_simple(f, a, b, n):
  h = (b - a) / n
  s1 = 0.0
  for k in range(1,n,2):
    s1 += f(a+k*h)
  s2 = 0.0
  for k in range(2,n,2):
    s2 += f(a+k*h)

  s = f(a) + f(b)
  s += 4 * s1
  s += 2 * s2
  return s * h / 3

print(simpson_simple(funcI,0.0001,10,100))
print(simpson_simple(funcI,0.0001,100,100))
print(simpson_simple(funcI,0.0001,200,100))
print(simpson_simple(funcI,0.0001,10,1000))
print(simpson_simple(funcI,0.0001,100,1000))
print(simpson_simple(funcI,0.0001,200,1000))
print(simpson_simple(funcI,0.0001,200,20000))
print(simpson_simple(funcI,0.0001,400,40000))
```

```
6.431923556355499
6.5106034223890665
6.733703321472937
6.431921896947455
6.493941068815315
6.4939660670977934
6.493939402433141
6.4939394024331305
```

Let's look at these results and discuss them. And then use this to calculate the Stefan-Boltzmann-constant

Exercise 5.12, Stefan-Boltzmann constant

$$W = \frac{k_B^4 T^4}{4\pi^2 c^2 \hbar^3} \int_0^\infty \frac{x^3}{e^x - 1} dx = \sigma T^4 = \frac{k_B^4}{4\pi^2 c^2 \hbar^3} 6.493939 T^4$$

Let's look at these results and discuss them. And then use this to calculate the Stefan-Boltzmann-constant

k_B = 1.3806 x 10^{-23} J/K

c = 2.9979 x 10^{8} m/s

hbar = 1.054572 x 10^{-34} Js

Plug in: σ=5.670x10^{-8} $Wm^{-2}K^{-4}$

(agrees with actual value!)

Integration in more than one dimension

In principle evaluate in one dimension and then in the other, though this gets ugly if limits of integration in one variable depend on the other variable. We will discuss one way to sample a higher-dimensional space in later chapters when we get to Monte Carlo methods

Let's look at a simple integral

$$\int_0^1 x^{-\frac{2}{3}}\,dx = \frac{1}{-\frac{2}{3}+1}x^{1/3}|_0^1 = \frac{1}{1/3}x^{1/3}|_0^1 = 3$$

But what happens if we try and use any of our previous methods to evaluate this? Note that they all include the endpoints of the integral. But x^{-2/3} at x=0 is not something that we can evaluate!

More generally, we'd like to be able to evaluate our integral for any arbitrary non-uniform set of N points. We saw before that we can approximate an integral with 2 points via a straight line, 3 points via a 2nd order polynomial, and more generally, N points with a polynomial of degree N-1

Method of interpreting polynomials

Let's imagine we have sampled the function we want to integrate at N points x_k. Define:

$$\phi_k(x) = \Pi_{m=1..N, m\neq k} \frac{x - x_m}{x_k - x_m}$$

Consider this polynomial for our set of N points. If we evaluate it at any of the points $x=x_m$ we get 1 if m==k and 0 if m is not k. Let's check this:

$$\phi_k(x_m) = \delta_{km}$$

$$\Phi(x) = \sum_{k=1}^{N} f(x_k)\phi_k(x) = \sum_{k=1}^{N} f(x_k)\delta_{km} = f(x_m)$$

So Φ is the polynomial that fits through all our points!

Method of interpreting polynomials

$$\Phi(x) = \sum_{k=1}^{N} f(x_k)\phi_k(x) = \sum_{k=1}^{N} f(x_k)\delta_{km} = f(x_m)$$

Use the polynomial to estimate our integral!

$$\int_a^b f(x)dx \sim \int_a^b \Phi(x)dx = \int_a^b f(x_k)\phi_k(x)dx = \sum_{k=1}^{N} f(x_k) \int_a^b \phi_k(x)dx$$

This gives us the weights we need to use for each point, but unfortunately there is no simple closed form for the weights. But we can do this once and then reuse them over and over. Which is what we will do. See appendix of book for tool to do this

Gaussian integration

Will use this code from textbook online resources

```
# Gaussian Integration code from the textbook online resource
#######################################################################
#
# Functions to calculate integration points and weights for Gaussian
# quadrature
#
# x,w = gaussxw(N) returns integration points x and integration
#           weights w such that sum_i w[i]*f(x[i]) is the Nth-order
#           Gaussian approximation to the integral int_{-1}^1 f(x) dx
# x,w = gaussxwab(N,a,b) returns integration points and weights
#           mapped to the interval [a,b], so that sum_i w[i]*f(x[i])
#           is the Nth-order Gaussian approximation to the integral
#           int_a^b f(x) dx
#
# This code finds the zeros of the nth Legendre polynomial using
# Newton's method, starting from the approximation given in Abramowitz
# and Stegun 22.16.6.  The Legendre polynomial itself is evaluated
# using the recurrence relation given in Abramowitz and Stegun
# 22.7.10.  The function has been checked against other sources for
# values of N up to 1000.  It is compatible with version 2 and version
# 3 of Python.
#
# Written by Mark Newman <mejn@umich.edu>, June 4, 2011
# You may use, share, or modify this file freely
#
#######################################################################
```

```
from numpy import ones,copy,cos,tan,pi,linspace

def gaussxw(N):

    # Initial approximation to roots of the Legendre polynomial
    a = linspace(3,4*N-1,N)/(4*N+2)
    x = cos(pi*a+1/(8*N*N*tan(a)))

    # Find roots using Newton's method
    epsilon = 1e-15
    delta = 1.0
    while delta>epsilon:
        p0 = ones(N,float)
        p1 = copy(x)
        for k in range(1,N):
            p0,p1 = p1,((2*k+1)*x*p1-k*p0)/(k+1)
        dp = (N+1)*(p0-x*p1)/(1-x*x)
        dx = p1/dp
        x -= dx
        delta = max(abs(dx))

    # Calculate the weights
    w = 2*(N+1)*(N+1)/(N*N*(1-x*x)*dp*dp)

    return x,w

def gaussxwab(N,a,b):
    x,w = gaussxw(N)
    return 0.5*(b-a)*x+0.5*(b+a),0.5*(b-a)*w
```

Gaussian integration

```
# We added this one
def integrateGauss(N,a,b,f):
    integral = 0.0
    ## get the sample points and optimal weights
    x,w = gaussxwab(N,a,b)

    # Now we use these in the integration
    for k in range(N):
      integral = integral+w[k]*f(x[k])
    return integral
```

```
# Use the above to integrate our problematic function
from math import pow
def f(x):
  return pow(x,-2/3.)

N = 1000
a = 0.0
b = 1.0

integral = integrateGauss(N,a,b,f)

print(integral)
```

```
2.981489262465544
```

Adaptive trapezoidal rule (5.20)

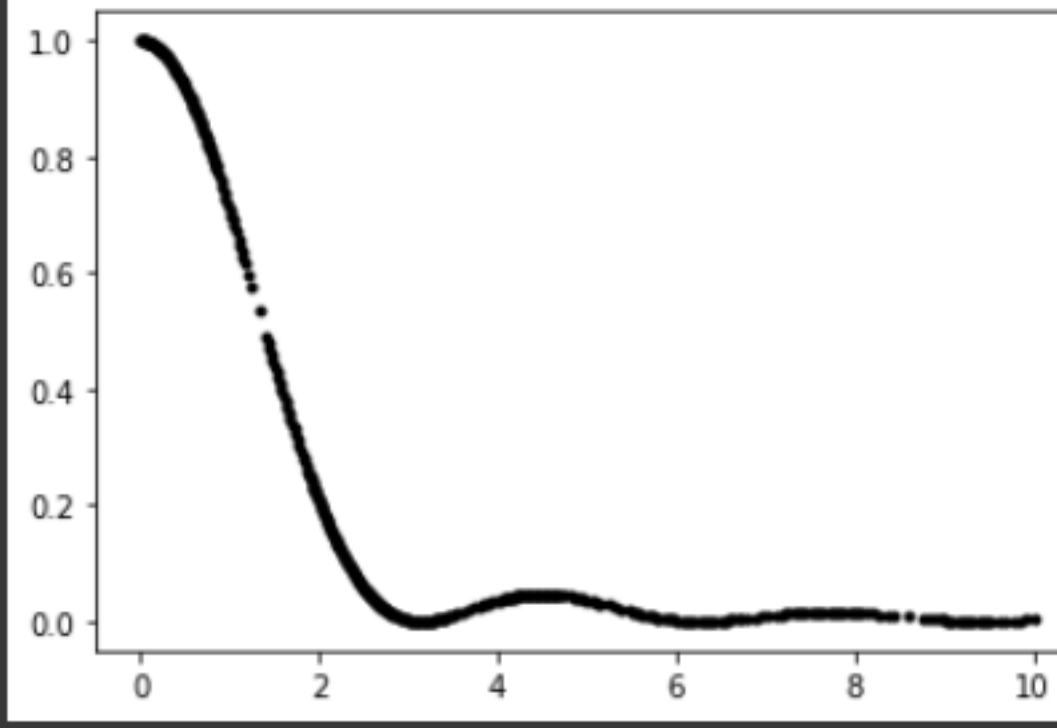
```
# 5.20

from numpy import sin
from matplotlib.pyplot import plot,show

a = 0.0
b = 10.0
epsilon = 1e-4
delta = epsilon/(b-a)

# The points we use, start with ends only
xpoints = [a,b]

# Integrand
def f(x):
  if x == 0.0:
    return 1.0
  return (sin(x)/x)**2
```

$$I = \int_0^{10} \frac{\sin^2 x}{x^2} dx$$

Adaptive trapezoidal rule (5.20)

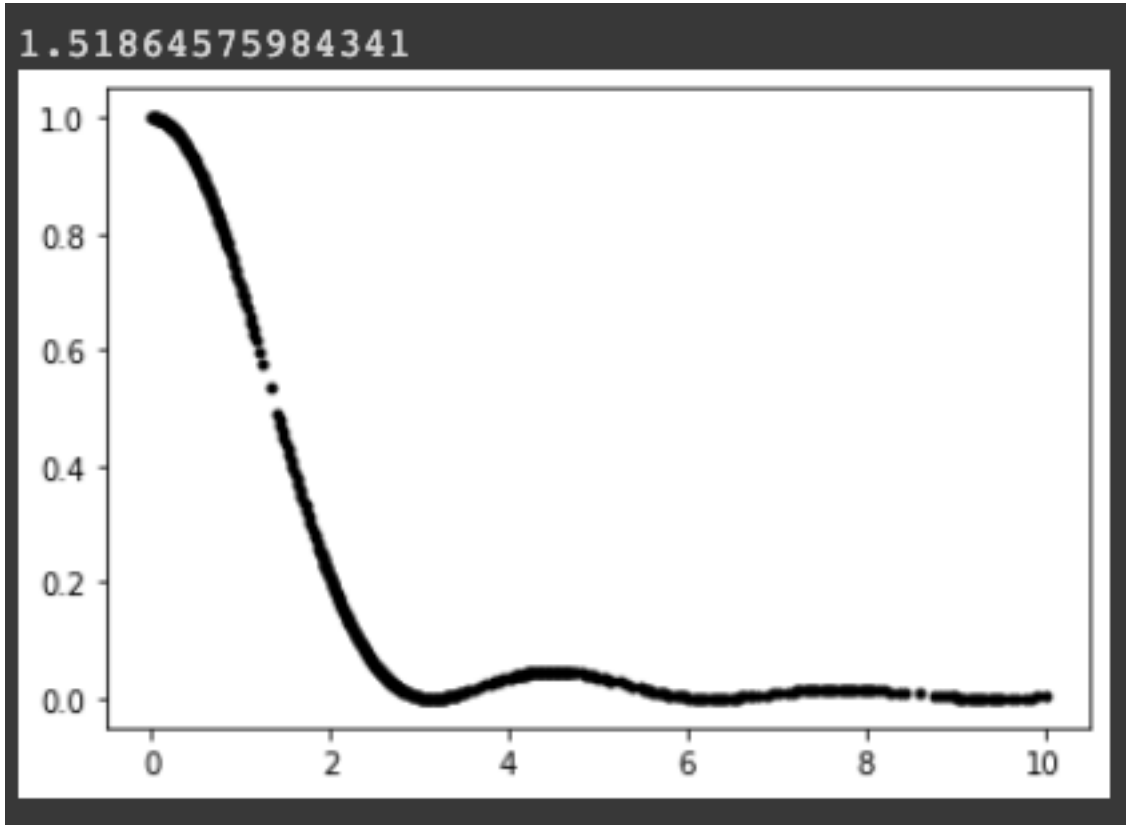
```
# Recursive function to do one step
# We pass the values of the function so we don't have to calculate them more than once, which speeds things up!

def step(x1,x2,f1,f2):
  h = x2-x1
  xm = 0.5*(x1+x2)
  xpoints.append(xm)
  fm = f(xm)

  # Calculate two estimates and the error
  I1 = h*(f1+f2)/2
  I2 = h*(f1+2*fm+f2)/4
  if abs(I2-I1)/3 < (delta*h): ### we are done wi
    return h*(f1+4*fm+f2)/6 # use improved simpso

  # Point was too large, divide into two parts an
  I1 = step(x1,xm,f1,fm) # First part
  I2 = step(xm,x2,fm,f2) # Second part
  return I1+I2 # return the sum of the two pieces

# Main program
fa = f(a)
fb = f(b)
print(step(a,b,fa,fb))

fpoints = []
for x in xpoints:
  fpoints.append(f(x))
plot(xpoints,fpoints,"k.")
show()
```

```
1.51864575984341
```



Adaptive trapezoidal rule (5.20)

$$I = \int_0^{10} \frac{\sin^2 x}{x^2} dx$$



Can see the integral having wider binning where the function follows a straight line, and narrow binning with more curvature!

Derivatives

We can evaluate this in the same sort of way as we did integrals, though we typically won't use as many fancy techniques

$$\frac{df}{dx} = f' = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Exact

$$f' \sim \frac{f(x+h) - f(x)}{h}$$

Forward difference

$$f' \sim \frac{f(x) - f(x-h)}{h}$$

Backward difference

Forward vs backward difference equally valid in most cases

Error on derivatives

$$f(x+h) = f(x) + hf'(x) + \frac{1}{2}h^2 f''(x) + \ldots \rightarrow$$

$$f'(x) = \frac{f(x+h) - f(x)}{h} - \frac{1}{2}hf''(x) + \ldots$$

So our estimate has errors of O(h). As the book points out, though, if we make h smaller, we are subtracting two numbers that will be close together, so if h is too small we again get large errors!

Central differences

Central difference: We don't actually look at f(x)!

$$\frac{df}{dx} \sim \frac{f(x+h/2) - f(x-h/2)}{h}$$

$$f(x+h/2) = f(x) + \frac{h}{2}f'(x) + (\frac{h}{2})^2\frac{1}{2!}f''(x) + (\frac{h}{2})^3\frac{1}{3!}f'''(x) + \ldots$$

$$f(x+h/2) = f(x) + \frac{h}{2}f'(x) + \frac{h^2}{8}f''(x) + \frac{h^3}{48}f'''(x) + \ldots$$

$$f(x-h/2) = f(x) - \frac{h}{2}f'(x) + (\frac{h}{2})^2\frac{1}{2!}f''(x) - (\frac{h}{2})^3\frac{1}{3!}f'''(x) + \ldots$$

$$f(x-h/2) = f(x) - \frac{h}{2}f'(x) + \frac{h^2}{8}f''(x) - \frac{h^3}{48}f'''(x) + \ldots$$

Central differences

$$\frac{df}{dx} \sim \frac{f(x+h/2) - f(x-h/2)}{h}$$

$$f(x+h/2) = f(x) + \frac{h}{2}f'(x) + \frac{h^2}{8}f''(x) + \frac{h^3}{48}f'''(x) + \ldots$$

$$f(x-h/2) = f(x) - \frac{h}{2}f'(x) + \frac{h^2}{8}f''(x) - \frac{h^3}{48}f'''(x) + \ldots$$

Now subtract one from the other

$$f(x+h/2) - f(x-h/2) = hf'(x) + \frac{h^3}{24}f'''(x) + \ldots$$

$$f'(x) = \frac{f(x+h/2) - f(x-h/2)}{h} - \frac{h^2}{24}f'''(x) + \ldots$$

Central differences

Central difference: We don't actually look at f(x)! Correct to Order(h) now - an improvement! And error Order(h^2)

$$\frac{df}{dx} \sim \frac{f(x+h/2) - f(x-h/2)}{h}$$

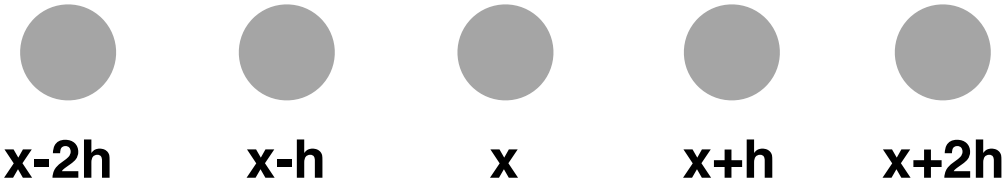
Note that we require points at +/- h/2 from the point of interest, so we have to be able to make the functional evaluation at that point!

Higher-order approximations

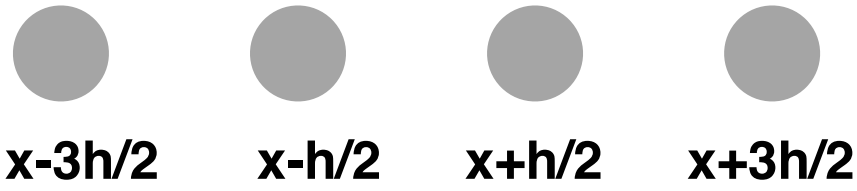
Why not use more points to evaluate the derivative, just like we did for the integral?

We can already think about this in terms of whether we use an even or odd number of points

Odd number of points (such as 5) includes the use of f(x), f(x±h), … f(x±nh)



Even number of points (such as 4) includes the use of f(x±h/2), f(x±3h/2), … f(x±nh±1/2)



Second derivatives

Remember that:

$$\frac{df}{dx} \sim \frac{f(x+h/2) - f(x-h/2)}{h}$$

The above is evaluated at x. What if we evaluate it at x+h/2 and x-h/2?

$$\frac{df}{dx}(x+h/2) \sim \frac{f(x+h/2+h/2) - f(x+h/2-h/2)}{h}$$

$$\frac{df}{dx}(x+h/2) \sim \frac{f(x+h) - f(x)}{h}$$

$$\frac{df}{dx}(x-h/2) \sim \frac{f(x-h/2+h/2) - f(x-h/2-h/2)}{h}$$

$$\frac{df}{dx}(x-h/2) \sim \frac{f(x) - f(x-h)}{h}$$

Second derivatives

We just found:

$$\frac{df}{dx}(x+h/2) \sim \frac{f(x+h)-f(x)}{h}$$

$$\frac{df}{dx}(x-h/2) \sim \frac{f(x)-f(x-h)}{h}$$

Central difference:

$$\frac{df}{dx} \sim \frac{f(x+h/2)-f(x-h/2)}{h}$$

Take derivative again:

$$\frac{d^2f}{dx^2}(x) = f''(x) \sim \frac{f'(x+h/2)-f'(x-h/2)}{h}$$

Second derivatives

We just found:

$$\frac{df}{dx}(x+h/2) \sim \frac{f(x+h)-f(x)}{h}$$

$$\frac{df}{dx}(x-h/2) \sim \frac{f(x)-f(x-h)}{h}$$

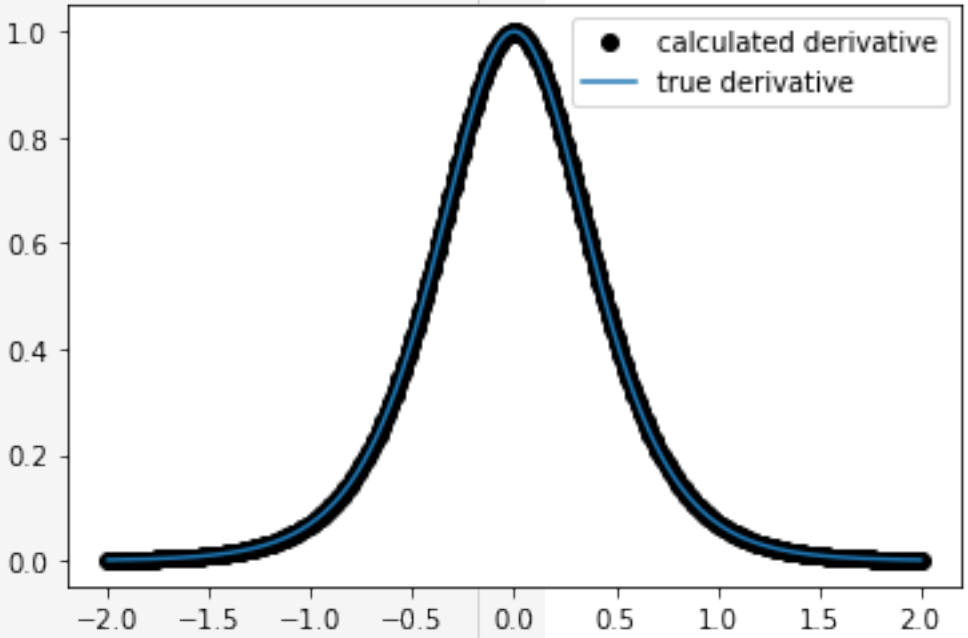
$$\frac{d^2f}{dx^2}(x) = f''(x) \sim \frac{f'(x+h/2)-f'(x-h/2)}{h}$$

$$f''(x) \sim \frac{f(x+h)-f(x)-f(x)+f(x-h)}{h}$$

$$f''(x) \sim \frac{f(x+h)-2f(x)+f(x-h)}{h}$$

Exercise 5.15 from text


```
# Exercise 5.15
# f(x) = 1+0.5*tanh(2x)
# f'(x) = 0.5*d/dx(tanh(2x)) = 0.5*2*(1-tanh^2(2x)) = 1-tanh^2(2x) = 1/(cosh^2(2x))
# Calculate f'(x) and compare to true value using central difference derivatives

from math import tanh,cosh
from numpy import linspace
from matplotlib.pyplot import plot,show,legend

a=-2.0
b=2.0
N = 500 ## plotting points we calculate at
h = 1e-8 ## h size used for derivatives

def f(x):
  return 1+0.5*tanh(2*x)

def fprime_true(x):
  return 1/cosh(2*x)**2

xpoints = linspace(a,b,N)
dpoints_estimated = [] ### derivatives, estimated
dpoints_true = [] ### derivatives, analytic true

for x in xpoints:
  df = (f(x+0.5*h)-f(x-0.5*h))/h
  dpoints_estimated.append(df)
  dpoints_true.append(fprime_true(x))

plot(xpoints,dpoints_estimated,"ko",label="calculated derivative")
plot(xpoints,dpoints_true,label="true derivative")
legend()
show()
```



Exercise 5.21 from text (first parts)


```
# Exercise 5.21 electric field of a charge distribution

from math import sqrt,pi
from numpy import empty
from pylab import quiver,show,figure,axis,xlim,ylim,imshow,cm,colorbar

C = 1.0 # charge
d = 0.1 # separation
e0 = 8.8542e-12 # permittivity free space constant
L = 1.0 #size of square
M = 100 # number of points on a side
a = L/M # lattice spacing
epsilon = 1.e-12 #small number
Elimit = 4e10 # Max size of E vector, make things easier to see

# Create array to hold results
phi = empty([M+1,M+1],float)

# Calculate positions of the two points of charge
x1 = (L+d)/2
y1 = L/2
x2 = (L-d)/2
y2 = L/2
```

Exercise 5.21 from text (first parts)

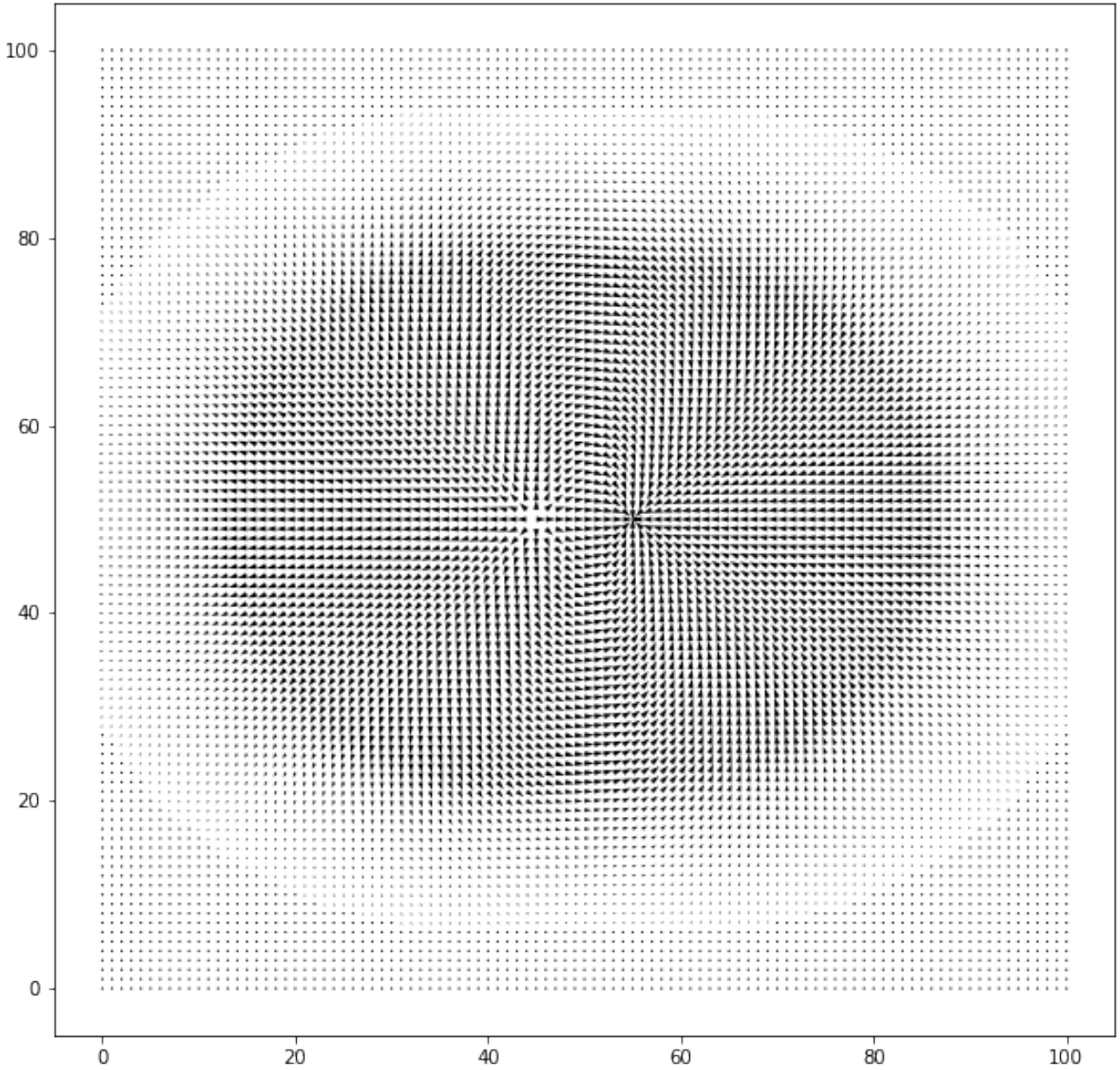

```
# Calculate potential
for i in range(M+1):
  y = i*a
  for j in range(M+1):
    x = j*a

    # Calculate potential at this point
    r1 = sqrt((x-x1)**2+(y-y1)**2)
    r2 = sqrt((x-x2)**2+(y-y2)**2)
    if r1 < epsilon: ## if too close potential is large, cut it off, positive
      phi[i,j] = 1./epsilon
    elif r2 < epsilon: ### it too close to other one
      phi[i,j] = -1./epsilon
    else:
      phi[i,j] = (1/r1-1/r2)*C/(4*pi*e0)

# We have potential, now get the field
Ex = empty([M+1,M+1],float)
Ey = empty([M+1,M+1],float)
```

Exercise 5.21 from text (first parts)


```
# Calculate x components
for i in range(M+1):

  # Edges first
  Ex[i,0] =  (phi[i,1]-phi[i,0])/a
  Ex[i,M] =  (phi[i,M]-phi[i,M-1])/a

  # Now the interior
  for j in range(1,M):
    Ex[i,j] = (phi[i,j+1]-phi[i,j-1])/(2*a)

# Calculate y components now
for j in range(M+1):

  # Edges first
  Ey[0,j] =  (phi[1,j]-phi[0,j])/a
  Ey[M,j] =  (phi[M,j]-phi[M-1,j])/a

  # Now the interior
  for i in range(1,M):
    Ey[i,j] = (phi[i+1,j]-phi[i-1,j])/(2*a)
```

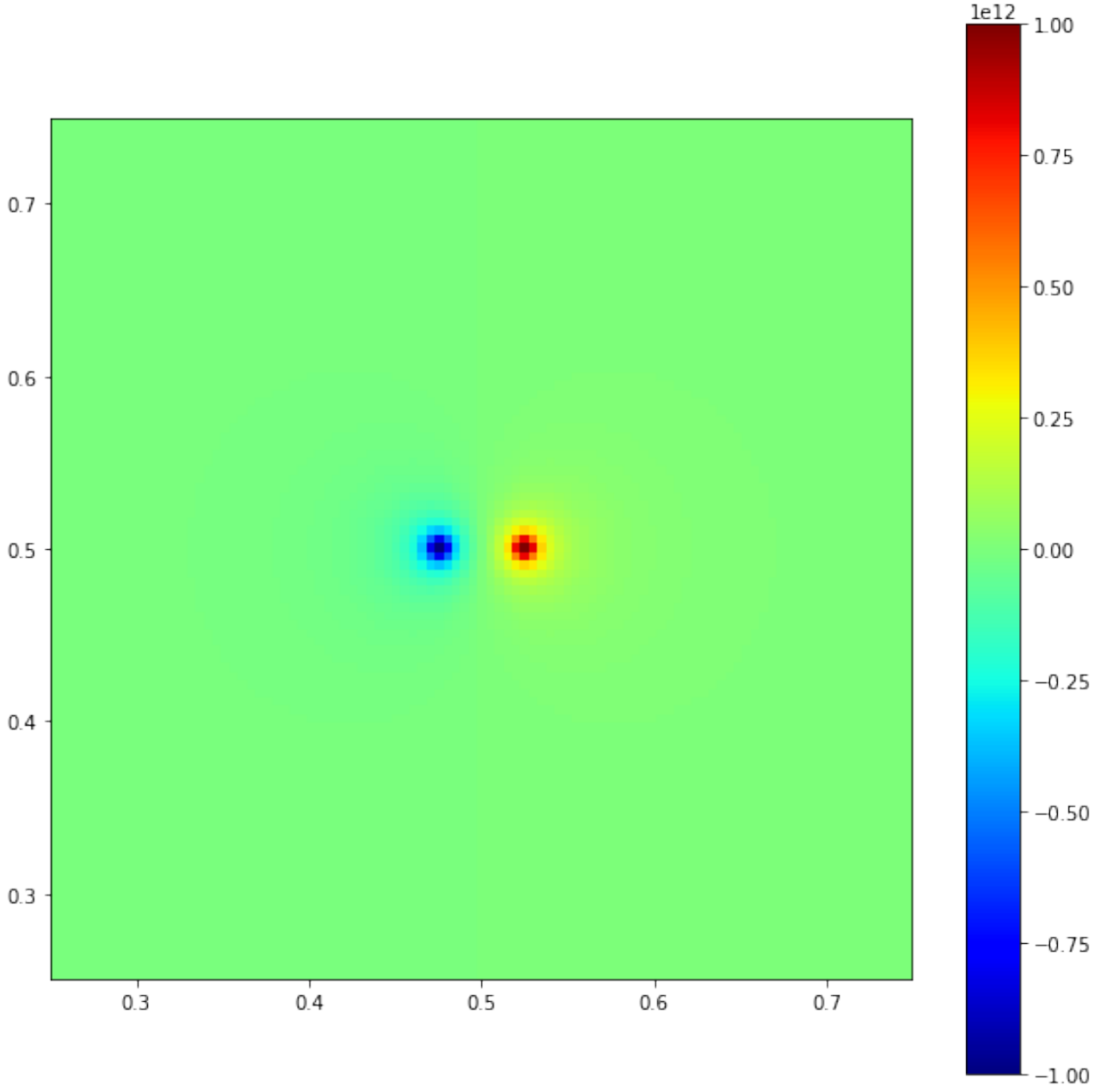
```
# Limit the magnitude for visibility
for i in range(M+1):
  for j in range(M+1):
    E = sqrt(Ex[i,j]**2+Ey[i,j]**2)
    ratio = E/Elimit
    if ratio > 1:
      Ex[i,j] /= ratio
      Ey[i,j] /= ratio

figure(figsize=(10,10))
quiver(Ex,Ey)
show()
figure(figsize=(10,10))
imshow(phi,cmap=cm.jet,extent=[0.25,0.75*L,0.25,0.75*L])
colorbar()
show()
```

Exercise 5.21 from text (first parts)



The electric field. Neat! Looks as expected

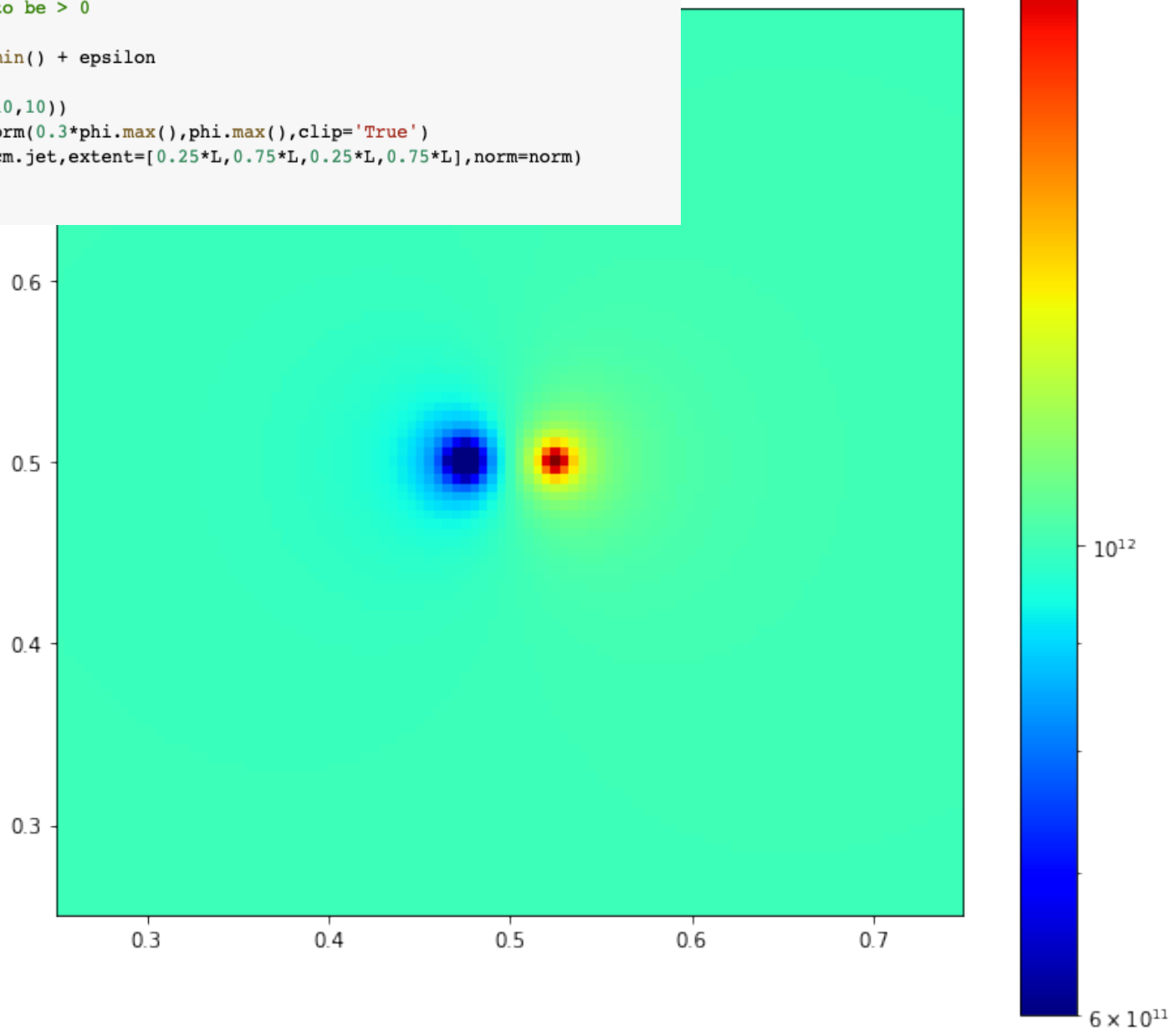
Exercise 5.21 from text (first parts)



The potential also looks as expected. But this is hard to visualize.

Exercise 5.21 from text (first parts)


```
### We can add any constant value to the potential and it doesn't matter.
### Shift the minimum to be just above zero so we can plot on a log scale
### Add epsilon to be > 0

phi = phi - phi.min() + epsilon

figure(figsize=(10,10))
norm=colors.LogNorm(0.3*phi.max(),phi.max(),clip='True')
imshow(phi,cmap=cm.jet,extent=[0.25*L,0.75*L,0.25*L,0.75*L],norm=norm)
colorbar()
show()
```



Homework #3

https://classroom.github.com/a/h9MpLPwV

Exercises 5.4 (suggest vmax = 0.005), 5.10, 5.11. For 5.11 plot on the same axis the result from z=3m and also z=1m and z=6m. Explain what you see

PHYS 510 only: Also 5.14. For this problem, you will need to do a two-dimensional Gaussian integration. Treat both dimensions as independent of one another and then the integration is straightforwards. And also note that the limits of your integration are the same so you only have to call gausswxab once!