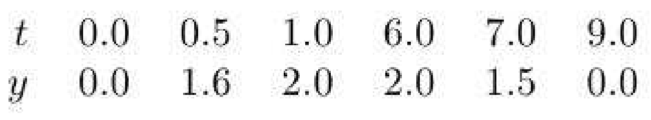
UNIVERSITY OF SUSSEX
Scientific Computing
Tutor: Dr. Ilian Iliev, Office: Pev 3 4A5

Problem Sheet 4 (Problem 2 will be assessed)

1. An experiment has produced the following data:



t	0.0	0.5	1.0	6.0	7.0	9.0
y	0.0	1.6	2.0	2.0	1.5	0.0

Figure 1: Data for Problem 1.

We wish to interpolate the data with a smooth curve in the hope of obtaining reasonable values of y for values of t between the points at which measurements were taken.

(a) Using the in-build `scipy.interpolate.barycentric_interpolate`, determine the polynomial of degree five that interpolates the given data, and make a smooth plot of it over the range $0 \leq t \leq 9$, marking the input data points with circles.

(b) Similarly, use the in-build Python function `interp1d` to determine a cubic spline that interpolates the given data, and make a smooth plot of it over the same range as in (a).

(c) Which interpolant seems to give more reasonable values between the given data points? Can you explain why each curve behaves the way it does?

(d) Might piecewise linear interpolation be a better choice for these particular data? Why?

Solution:

(a) We start by setting up the data points into 2 vectors:

```
t=np.array([0.0,0.5,1.0,6.0,7.0,9.0])
y=np.array([0.0,1.6,2.0,2.0,1.5,0.0])
```

Then we set up the times where the interpolations will be evaluated and call the interpolating polynomial function:

```
t0=np.arange(0.,9.,0.01)

y0=interpolate.barycentric_interpolate(t,y,t0)
```

This evaluates the interpolating polynomial at the points $t0$ and since there are 6 points, the polynomial is of $6 - 1 = 5$th order, as required.

We can plot the results using e.g.:

```
p1,=pl.plot(t0,y0)
p2,=pl.plot(t,y,'o')
```

(b) The `interp1d` function is called with the input date points t and y, for cubic splines, where we also need to supply the points x0 at which the spline should be evaluated, and then we plot the result:

```
ci=interpolate.interp1d(t,y,kind='cubic')

p3,=pl.plot(t0,ci(t0))
```

(c) It could be debated if either of the above interpolations in (a) and (b) provide a good representation of the underlying data. However, clearly the polynomial data oscillates considerably (this is called 'Runge phenomenon' after the mathematician who discovered it), without any support for such behaviour in the data points. The cubic spline interpolation looks more sensible, but lack of data, especially in the middle region means that we do not really know if there is a peak there as the cubic splines siggest.

(d) Given the above issues discussed in (c), it is indeed likely that piecewise linear interpolation (linear spline) could be better option here, given the lack of data in the middle region.

We can get the linear splines by calling the same `interp1d` function, but indicating we want linear splines to be used, and again plot the result on the same plot:

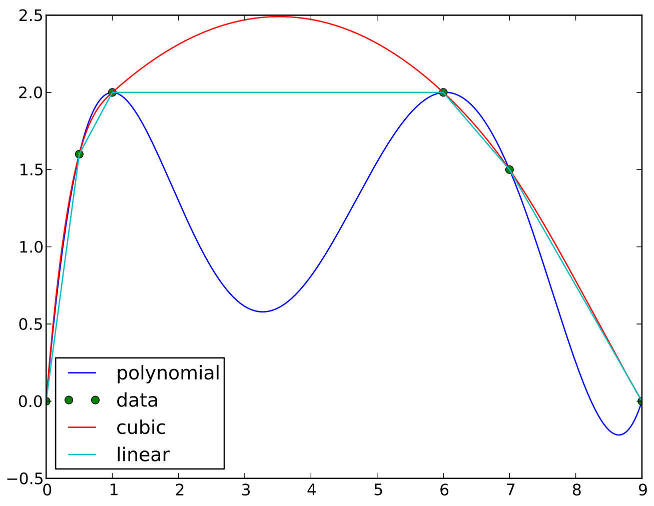
```
li=interpolate.interp1d(t,y,kind='linear')

p4,=pl.plot(t0,li(t0))
```

Finally, we can add a legend to the plot and show it (if script is used):

```
pl.legend([p1,p2,p3,p4],["polynomial","data","cubic","linear"],loc=3)

pl.show()
```

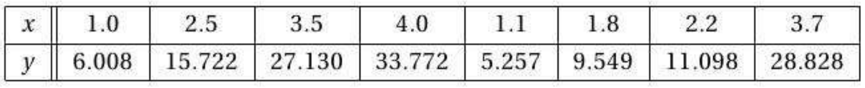


This results in the plot:

Clearly, the linear and cubic splines yield quite similar results where there are sufficient data points, but differ where data lacks. The polynomial interpolation does not behave well here (but is much better when used for interpolating smooth, well-behaved functions).

2. **Fitting data points:**

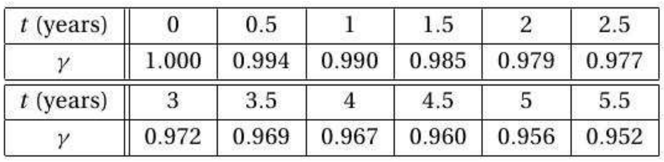
   (a) Fit a straight line and a quadratic to the data below using `curve_fit` function from `scipy.optimize`. Which fits better? (Hint: Compare the standard deviations.)



x	1.0	2.5	3.5	4.0	1.1	1.8	2.2	3.7
y	6.008	15.722	27.130	33.772	5.257	9.549	11.098	28.828

Figure 2: Data for straight line and quadratic fitting.

   (b) The intensity of radiation of a radioactive substance was measured at half-year intervals. The (noisy) results were:
   where γ is the relative intensity of radiation. Knowing that radioactivity decays exponentially with time, $\gamma(t) = ae^{-bt}$, estimate the radioactive half-life of the substance.



t (years)	0	0.5	1	1.5	2	2.5	3	3.5	4	4.5	5	5.5
γ	1.000	0.994	0.990	0.985	0.979	0.977	0.972	0.969	0.967	0.960	0.956	0.952

Figure 3: Data for exponential decay fitting.

[30]

Solution:

We first have to import the needed packages and set up the data points given in the problem as arrays, and we also define the points where the fits will be evaluated for plotting:

```
import numpy as np
import scipy
from scipy import optimize
from scipy.optimize import curve_fit
import pylab as pl

x=np.linspace(0,4,500)
z=np.linspace(0,5.5,500)

x1=np.array([1.0, 2.5, 3.5, 4.0, 1.1, 1.8, 2.2, 3.7])
y1=np.array([6.008, 15.722, 27.130, 33.772, 5.257, 9.549,
             11.098, 28.828])

x2=np.array([ 0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 5.5])
y2=np.array([ 1.000, 0.994, 0.990, 0.985, 0.979, 0.977, 0.972, 0.969,
 0.967, 0.960, 0.956, 0.952])
```

We then set out fitting functions themselves:

```
def f1(x,a,b):
    return a*x+b

def f2(x,a,b,c):
    return a*x**2+b*x+c
```

```
def f3(x,a,b):
    return a*x*np.exp(b*x)
```

We can then fit the data using the in-build function, as suggested in class:

```
p1,c1=curve_fit(f1,x1,y1)
p2,c2=curve_fit(f2,x1,y1)
p3,c3=curve_fit(f3,x2,y2)
```

The standard deviations of each fit are calculated using the formula given in class (and the textbook):

```
sigma1=np.sqrt(sum((y1-(p1[0]*x1+p1[1]))**2)/(len(y1)-2.))
sigma2=np.sqrt(sum((y1-(p2[0]*x1**2+p2[1]*x1+p2[2]))**2)/(len(y1)-3.))

print 'sigmas', sigma1,sigma2

sigma3=np.sqrt(sum((y2-(p3[0]*np.exp(-p3[1]*x2)))**2)/(len(y2)-2))

print 'sigma exp', sigma3

print 'parameters', p3
print 'half-time of decay (years)', np.log(2)/p3[1]

print 'check',p3[0]*np.exp(-p3[1]*(np.log(2)/p3[1]))
```

Results are:

```
sigmas 2.24356382796 0.812927961054
sigma exp 0.00124457508761
parameters [ 0.99843145  0.00864506]
half-time of decay (years) 80.1783792901
check 0.499215727019
```

Clearly, the quadratic fit to the first dataset is much better than the linear fit. The exponential one to dataset 2 is quite good. The half-life is easily calculated by setting the abundance to 0.5, which gives half-life of about 80 years.

Finally, we plot the results from all fits along with the data points, e.g. like this:

```
pl.scatter(x1,y1)
pl.plot(x,p1[0]*x+p1[1])
pl.plot(x,p2[0]*x**2+p2[1]*x+p2[2])
```

for part (a) and:

```
pl.scatter(x2,y2)
pl.plot(z,p3[0]*np.exp(-p3[1]*z))
```

for part (b).

Results are:

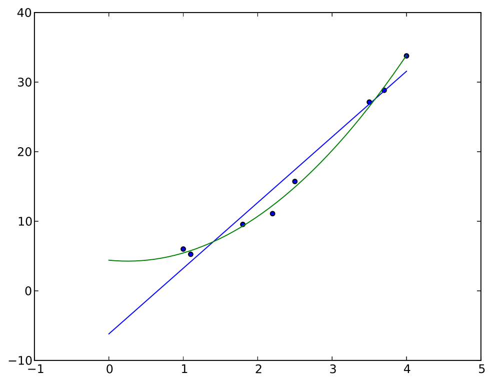


Figure 4: Linear and quadratic fits to dataset 1.

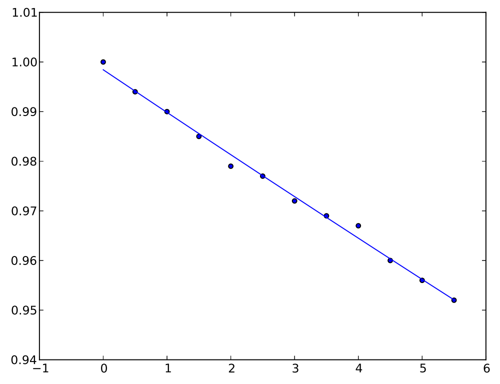


Figure 5: Exponential fit to dataset 2.