

Prof. Jon Stevens, TAs: Marcus Meryfield, Nigel Zikri, Daniel Coelho

```
In [144]: import numpy as np
From scipy import integrate
import matplotlib.pyplot as plt
import astropy.units as un
```

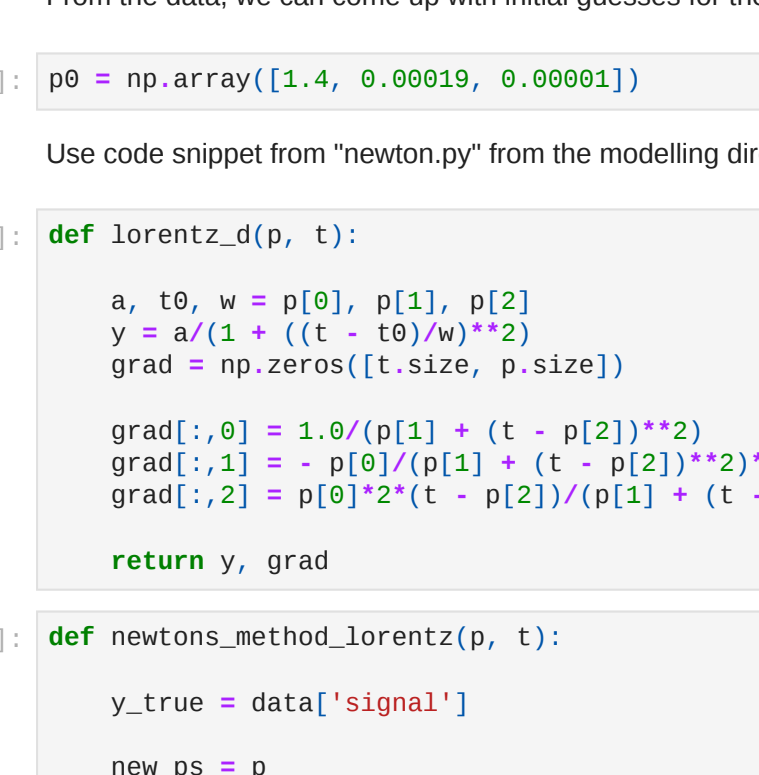
Problem 1

```
a)
We first load the data

In [4]: data = np.load('%sidebands.npy')
t = data['time']
d = data['signal']

In [5]: plt.plot(t, d)
plt.xlabel('Time [s]')
plt.ylabel('data value')
plt.title('Data to fit')
plt.xticks(rotation = 45)

Out[5]: (array([[-5.0e-05, 0.8e+08, 5.0e-05, 1.9e-04, 1.5e-04, 2.0e-04,
1.5e-04, 3.0e-04, 3.5e-04, 4.0e-04, 4.5e-04]]),
[Text(0, 0, ''),
Text(0, 0, ''),
Text(0, 0, ''),
Text(0, 0, ''),
Text(0, 0, ''),
Text(0, 0, ''),
Text(0, 0, ''),
Text(0, 0, ''),
Text(0, 0, ''),
Text(0, 0, ''])]

Data to fit


Now we would like to model the data as a single Lorentzian and use analytic derivatives. To do so, we use Newton's method to carry out the fit. We parametrize the Lorentzian as  $d = \frac{a}{1 + ((t - t_0)/\omega)^2}$ 
```

```
From the data, we can come up with initial guesses for the different parameters we will use: amplitude  $a \sim 1.4$ , peak location  $t_0 \sim 0.00019$  and peak spread  $\omega \sim 0.00001$ 

In [6]: p0 = np.array([1.4, 0.00019, 0.00001])

Use code snippet from 'newton.py' from the modelling directory, adapting it to the form of the model we are using for d:
```

```
In [7]: def lorentz_d(p, t):
a, t0, w = p[0], p[1], p[2]
y = a/(1 + ((t - t0)/w)**2)
grad = np.zeros((t.size, p.size))

grad[:,0] = 1.0/(p[1] + (t - p[2])**2)
grad[:,1] = -p[0]/(p[1]**3) * (t - p[2])**2**2
grad[:,2] = p[0]**2*(t - p[2])/(p[1]**3 + (t - p[2])**2)**2

return y, grad

In [8]: def newtons_method_lorentz(p, t):
y_true = data['signal']
new_ps = p

for i in range(100):
res = y_true - pred
res = np.matrix(res).T
grad = np.matrix(grad)
dp = np.linalg.pinv(grad.T*grad)*(grad.T*res)

for i in range(len(new_ps)):
new_ps[i] += dp[i]
new_ps = np.asarray(new_ps)

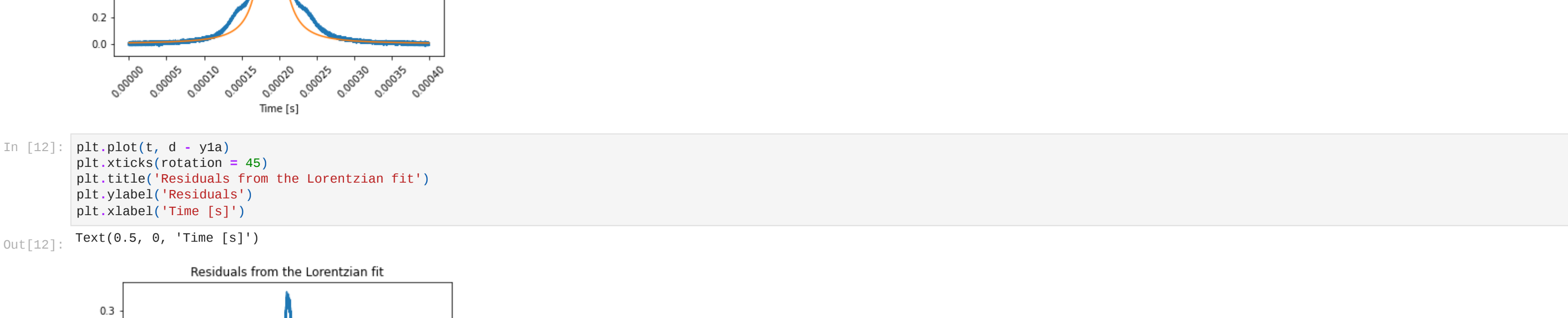
return new_ps

In [9]: p_newdta = newtons_method_lorentz(p0, t)

In [10]: yda, gradda = lorentz_d(p_newdta, t)

In [11]: plt.plot(t, d, label = 'True data')
plt.plot(t, yda, label = 'Lorentzian fit')
plt.xlabel('Time [s]')
plt.ylabel('Signal')
plt.title('Lorentzian fit over the data')
plt.xticks(rotation = 45)
print('Parameters found: a =', p_newdta[0], ', t_0 =', p_newdta[1], ', omega =', p_newdta[2])

Parameters found: a = 1.389999965497165, t_0 = 0.0001982626518466272, omega = -1.4139742183718972e-05
```



```
In [12]: plt.plot(t, d - yda)
plt.xticks(rotation = 45)
plt.title('Residuals from the Lorentzian fit')
plt.ylabel('Residuals')
plt.xlabel('Time [s]')
Text(0.5, 0, 'Time [s]')

Out[12]: Residuals from the Lorentzian fit

```

b) Now we estimate the noise in the data, and from it, estimate the error on our parameters

```
In [13]: sigma = np.std(d)
sc_sigma = sigma*np.sqrt(len(d))

In [14]: cov_inv = np.eye(len(t))
np.fill_diagonal(cov_inv, 1/(sc_sigma**2))

The following code chunk doesn't work. The simple first matrix multiplication to be done in the 'cov' definition kills the kernel after a minute or two. After all, those are 100 000 by 100 000 matrices, but I don't know if this was to be expected.

In [ ]: '''
cov = np.linalg.pinv(grada.T@cov_inv@grada)
errs = np.sqrt(np.diag(cov))
print('The error on a is:', errs[0])
print('The error on t0 is:', errs[1])
print('The error on w is:', errs[2])

I ended up managing to run the code to make the covariance matrix on another machine. So I just pushed the array on github and I load it here!
```

```
In [15]: cov = np.load("mycov.npy")
errs = np.sqrt(np.diag(cov))
print('The error on a is:', errs[0])
print('The error on t0 is:', errs[1])
print('The error on w is:', errs[2])

The error on a is: 5.617820322334366e-14
The error on t0 is: 1.797214219689857e-13
The error on w is: 3.674805385773e-10

The error seems to be particularly small compared to the accuracy of the fit to the data (c.f. previous plot, e.g. the error on the peak magnitude seems to be orders of magnitude more than just  $10^{-14}$ .)

c)
```

We repeat part a) but using numerical derivatives. First we define a derivative function, then we will have different f functions whose variable will be the different parameters. The derivative is taken with respect to the different parameters to have the gradients for each parameter (to do so, we use lambda functions. It helps in having to avoid defining many different functions. We only define three f functions that take as variable each parameter. We compute the gradient numerically with a derivative function). We take an arbitrarily small time step dt.

```
In [16]: dt = 10**(-8)
deriv = lambda f, t: (1/2*dt)*(f(t + dt) - f(t - dt))

In [17]: def lorentz_num(p, t):
a, t0, w = p[0], p[1], p[2]
y = a/(1 + ((t - t0)/w)**2)
f_a = lambda x: a/(1 + ((t - t0)/w)**2)
f_t0 = lambda y: a/(1 + ((t - y)/w)**2)
f_w = lambda z: a/(1 + ((t - t0)/z)**2)

grad = np.zeros((len(t), len(p)))
grad[:,0] = deriv(f_a, a)
grad[:,1] = deriv(f_t0, t0)
grad[:,2] = deriv(f_w, w)

return y, grad

In [18]: def newtons_method_lorentz_num(data, p, t, n):
y_true = data['signal']
new_ps = p

for i in range(n):
calc_grad = lorentz_num(new_ps, t)
res = y_true - calc_grad
res = np.matrix(res).T
grad = np.matrix(grad)
dp = np.linalg.pinv(grad.T*grad)*(grad.T*res)

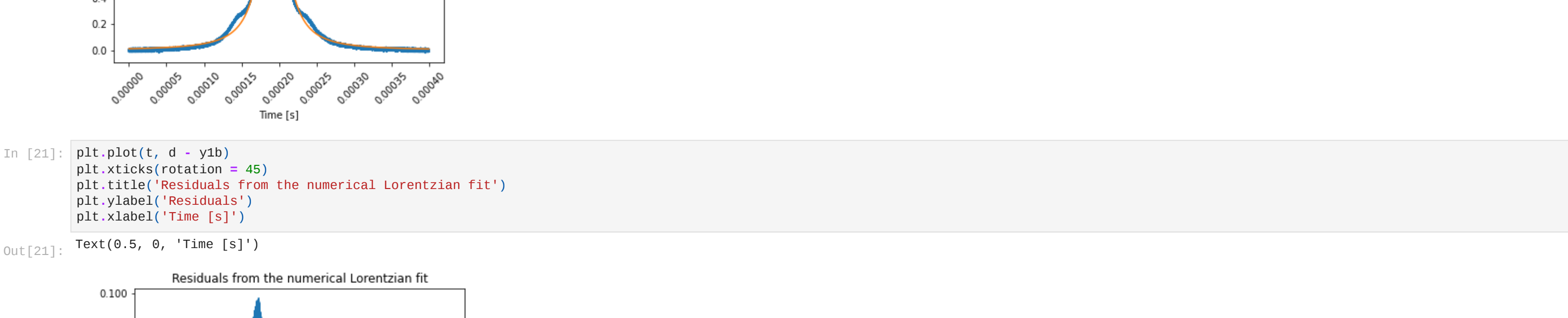
for i in range(len(new_ps)):
new_ps[i] += dp[i]
new_ps = np.asarray(new_ps)

return new_ps
```

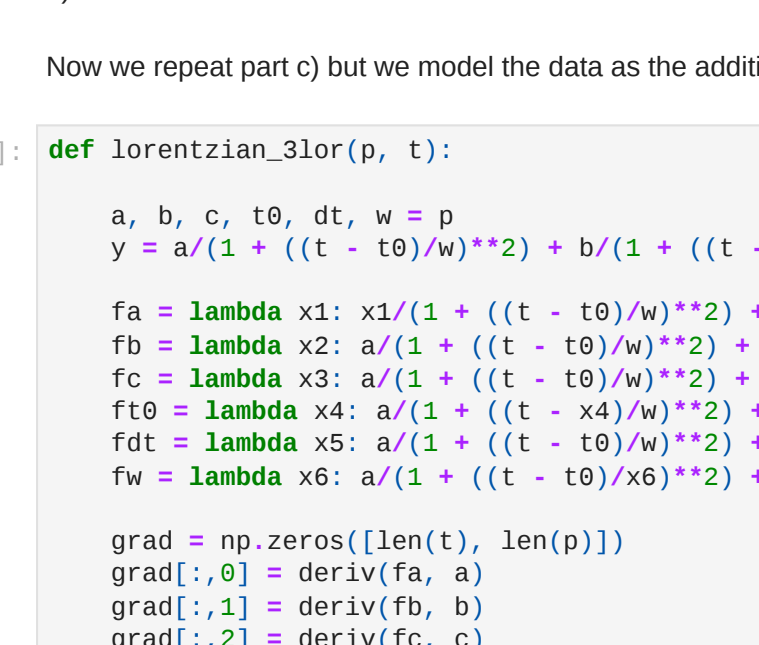
```
In [19]: p_newnb = newtons_method_lorentz_num(data, p0, t, 500)
y1b, grad1b = lorentz_d(p_newnb, t)
```

```
In [20]: plt.plot(t, d, label = "True data")
plt.plot(t, y1b, label = "Numerical Lorentzian fit")
plt.xlabel('Time [s]')
plt.ylabel('Signal')
plt.title('Numerical Lorentzian fit over the data')
plt.xticks(rotation = 45)
print('Parameters found: a =', p_newnb[0], ', t_0 =', p_newnb[1], ', omega =', p_newnb[2])

Parameters found: a = 1.422818682183835, t_0 = 0.0001923586492623834, omega = -1.792369075490326e-05
```



```
In [21]: plt.plot(t, d - y1b)
plt.xticks(rotation = 45)
plt.title('Residuals from the numerical Lorentzian fit')
plt.ylabel('Residuals')
plt.xlabel('Time [s]')
Text(0.5, 0, 'Time [s]')

Out[21]: Residuals from the numerical Lorentzian fit

```

The numerical derivatives approach yields similar parameters as in a), although the order of magnitude of discrepancy between the data and the fit seems to be lower than in a), as the comparison of the residuals' plot shows.

d) Now we repeat part a) but we model the data as the addition of three Lorentzian functions. Very similarly, we define our lambda functions for each parameter and we define the gradient numerically with our deriv function.

```
In [23]: def lorentzian_3lor(p, t):
a, b, c, t0, dt, w = p
y = a/(1 + ((t - t0)/w)**2) + b/(1 + ((t - t0 + dt)/w)**2) + c/(1 + ((t - t0 - dt)/w)**2)

fa = lambda x1: x1/(1 + ((t - t0)/w)**2) + b/(1 + ((t - t0 + dt)/w)**2) + c/(1 + ((t - t0 - dt)/w)**2)
fb = lambda x2: a/(1 + ((t - t0)/w)**2) + x2/(1 + ((t - t0 + dt)/w)**2) + c/(1 + ((t - t0 - dt)/w)**2)
fc = lambda x3: a/(1 + ((t - t0)/w)**2) + b/(1 + ((t - t0 + dt)/w)**2) + x3/(1 + ((t - t0 - dt)/w)**2)
fd = lambda x4: a/(1 + ((t - t0)/w)**2) + b/(1 + ((t - t0 + dt)/w)**2) + c/(1 + ((t - t0 - dt)/w)**2)
fe = lambda x5: a/(1 + ((t - t0)/w)**2) + b/(1 + ((t - t0 + dt)/w)**2) + c/(1 + ((t - t0 - dt)/w)**2)
fw = lambda x6: a/(1 + ((t - t0)/w)**2) + b/(1 + ((t - t0 + dt)/w)**2) + c/(1 + ((t - t0 - dt)/w)**2)

grad = np.zeros((len(t), len(p)))
grad[:,0] = deriv(fa, a)
grad[:,1] = deriv(fb, b)
grad[:,2] = deriv(fc, c)
grad[:,3] = deriv(fd, t0)
grad[:,4] = deriv(fe, dt)
grad[:,5] = deriv(fw, w)

return y, grad
```

```
In [24]: def newtons_method_3lor(data, p, t, n):
y_true = data['signal']
p_new = p

for i in range(n):
calc_grad = lorentzian_3lor(p_new, t)
res = y_true - calc_grad
res = np.matrix(res).T
grad = np.matrix(grad)
dp = np.linalg.pinv(grad.T*grad)*(grad.T*res)

for i in range(len(p_new)):
p_new[i] += dp[i]
p_new = np.asarray(p_new)

return p_new
```

We need to have estimates for the amplitude of the two other Lorentzian peaks, which from the data plot seem to be of order ~ 0.3 , and their distance from the main peak seems to be of order ~ 0.00005

```
In [25]: p0_3lor = np.array([1.5, 0.3, 0.3, 0.00019, 0.00005, 0.00001])
p_newd = newtons_method_3lor(data, p0_3lor, t, 100)
y1d, grad1d = lorentzian_3lor(p_newd, t)
```

```
In [26]: print("The best-fit parameters for the sum of Lorentzians using numerical derivatives are:")
print("a:", p_newd[0])
print("prob. accept:", p_newd[1])
print("c:", p_newd[2])
print("mu:", p_newd[3])
print("dt:", p_newd[4])
print("w:", p_newd[5])

The best-fit parameters for the sum of Lorentzians using numerical derivatives are:
a: 1.4429923932697661
b: 0.2639077098946761
c: 0.6647323080929792
t0: 0.0001923586492623835
dt: 4.4887426545412e-05
w: 1.606510942376226e-05
```

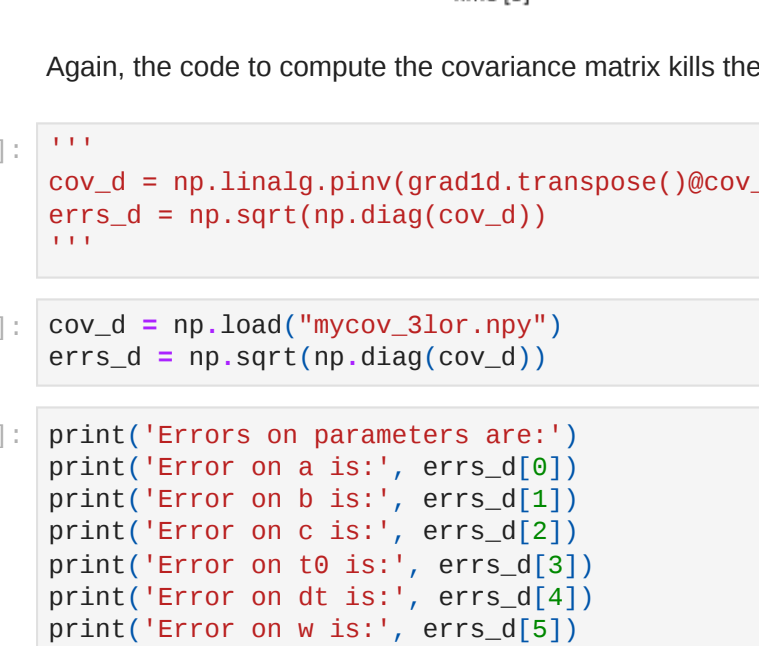
```
In [27]: plt.plot(t, d, label = "True data")
plt.plot(t, y1d, label = "Three Lorentzian fits")
plt.xlabel('Time [s]')
plt.ylabel('Signal')
plt.title('Three Lorentzian fits over the data')
plt.legend()
plt.xticks(rotation = 45)

Out[27]: (array([[-5.0e-05, 0.8e+08, 5.0e-05, 1.9e-04, 1.5e-04, 2.0e-04,
1.5e-04, 3.0e-04, 3.5e-04, 4.0e-04, 4.5e-04]]),
[Text(0, 0, ''),
Text(0, 0, ''),
Text(0, 0, ''),
Text(0, 0, ''),
Text(0, 0, ''),
Text(0, 0, ''),
Text(0, 0, ''),
Text(0, 0, ''),
Text(0, 0, ''),
Text(0, 0, '')])

Three Lorentzian fits over the data

```

```
In [28]: plt.plot(t, d - y1d)
plt.xticks(rotation = 45)
plt.title('Residuals from the 3 Lorentzian fits')
plt.ylabel('Residuals')
plt.xlabel('Time [s]')
Text(0.5, 0, 'Time [s]')

Out[28]: Residuals from the 3 Lorentzian fits

```

Again, the code to compute the covariance matrix kills the kernel. Once more, I run it on another machine, put it on github and load it:

```
In [ ]: '''
cov_d = np.linalg.pinv(grad1d.transpose()@cov_inv@grad1d)
errs_d = np.sqrt(np.diag(cov_d))

In [29]: cov_d = np.load("mycov_3lor.npy")
errs_d = np.sqrt(np.diag(cov_d))

In [30]: print('Errors on parameters are:')
print('Error on a is:', errs_d[0])
print('Error on b is:', errs_d[1])
print('Error on c is:', errs_d[2])
print('Error on t0 is:', errs_d[3])
print('Error on dt is:', errs_d[4])
print('Error on w is:', errs_d[5])

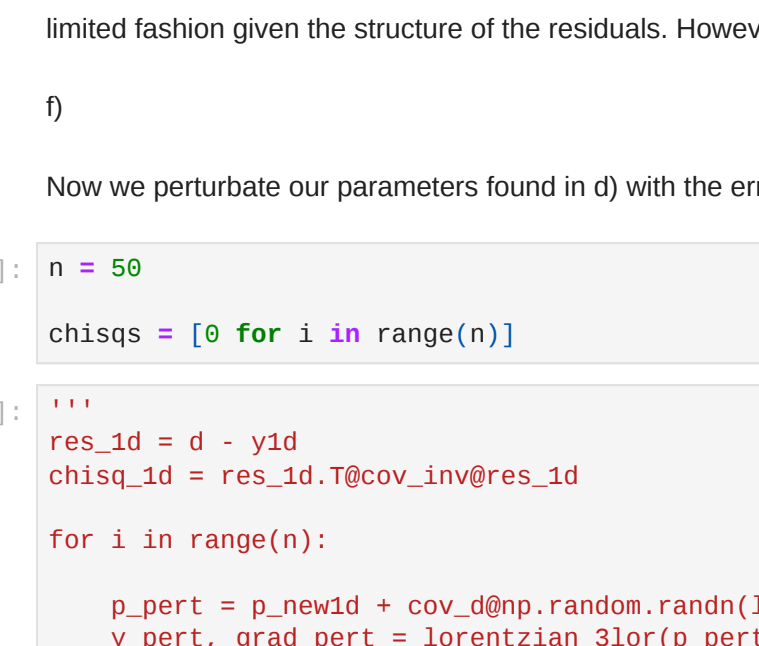
Errors on parameters are:
Error on a is: 1.984464049298022e-05
Error on b is: 1.816439139738888e-05
Error on c is: 1.778913384857236e-05
Error on t0 is: 2.250783892976696e-10
Error on dt is: 2.718178487099205e-09
Error on w is: 4.6931245203269164e-10

Again, the error on the parameters seems underestimated, although given the errors here are in general bigger than in a) and given the fit is much better than in a), that discrepancy does not seem to be as important.
```

e) Here we look at the residuals from the calculated fit found in d)

```
In [31]: plt.plot(t, d - y1d)
plt.title('Residuals from the 3 Lorentzian fits')
plt.xlabel('Time [s]')
plt.ylabel('Residuals')
plt.xticks(rotation = 45)

Out[31]: (array([[-5.0e-05, 0.8e+08, 5.0e-05, 1.9e-04, 1.5e-04, 2.0e-04,
1.5e-04, 3.0e-04, 3.5e-04, 4.0e-04, 4.5e-04]]),
[Text(0, 0, ''),
Text(0, 0, ''),
Text(0, 0, ''),
Text(0, 0, ''),
Text(0, 0, ''),
Text(0, 0, ''),
Text(0, 0, ''),
Text(0, 0, ''),
Text(0, 0, ''),
Text(0, 0, '')])

Residuals from the 3 Lorentzian fits

```

The look of the residuals does not look much random, rather it seems there is still some structure to it, especially near the peak. It is hence far to say that our model of the three Lorentzian could be upgraded, and that it models our data in a limited fashion given the structure of the residuals. However, our current model is not that bad, with the structure having an amplitude of order ~ 0.04 .

f) Now we perturbate our parameters found in d) with the errors and the covariance matrix. Then we can compute the χ^2 of each of those and if the model is well behaved, the χ^2 should be similar between the different realizations.

```
In [32]: n = 50
chisqs = [0 for i in range(n)]

In [33]: '''
res_id = d - y1d
chisq_id = res_id.T@cov_inv@res_id

for i in range(n):
p_newd = p_newd + cov_d@np.random.randn(len(p_newd))
y_perd, grad_perd = lorentzian_3lor(p_perd, t)
res_perd = d - y_perd.T
chisq[i] = res_perd.T@cov_inv@res_perd

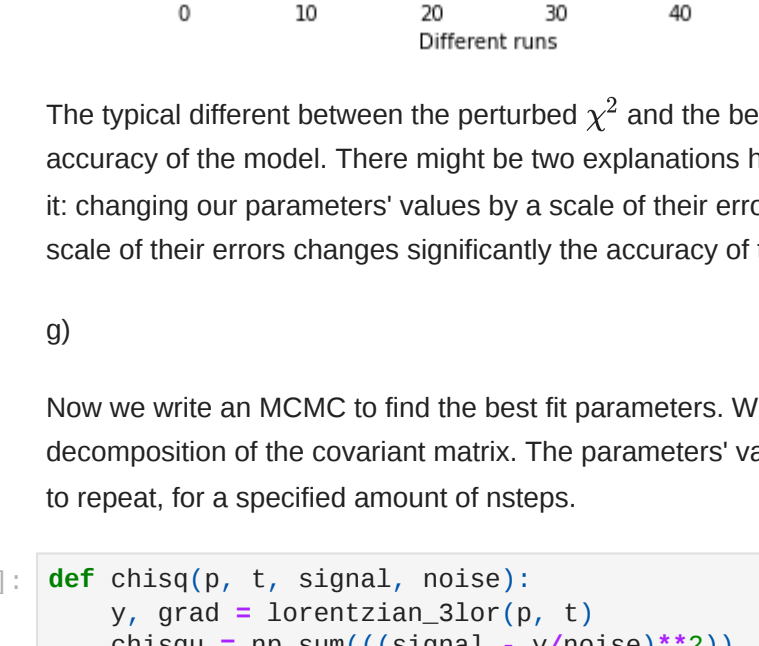
In [34]: 'Infer i in range(n): \n \n p_perd = p_newd + cov_d@np.random.randn(len(p_newd)) \n \n y_perd, grad_perd = lorentzian_3lor(p_perd, t) \n \n res_perd = d - y_perd.T \n \n chisq[i] = res_perd.T@cov_inv@res_perd \n \n

Again, my computer does not support running the code above. Another machine took approximately 1 hour to compute the 50  $\chi^2$ , and they are loaded from github here:
```

```
In [35]: chisq_id = np.load("chisq_id.npy")
chisqs = np.load("chisqs.npy")

In [43]: plt.plot(chisqs - chisq_id)
plt.xlabel('different runs')
plt.ylabel('difference in  $\chi^2$ chi^2s')
plt.title('difference in  $\chi^2$ chi^2s')
plt.xticks(rotation = 45)

Out[43]: (array([[-5.0e-05, 0.8e+08, 5.0e-05, 1.9e-04, 1.5e-04, 2.0e-04,
1.5e-04, 3.0e-04, 3.5e-04, 4.0e-04, 4.5e-04]]),
[Text(0, 0, ''),
Text(0, 0, ''),
Text(0, 0, ''),
Text(0, 0, ''),
Text(0, 0, ''),
Text(0, 0, ''),
Text(0, 0, ''),
Text(0, 0, ''),
Text(0, 0, ''),
Text(0, 0, '')])

Best-fit  $\chi^2$  - (perturbed  $\chi^2$ )

```

The typical difference between the perturbed χ^2 and the best-fit χ^2 is of order $\sim 1e-8$, which is very small. That means that changing our parameters by a scale proportional to the error on our parameters will only ever slightly change the accuracy of the model. There might be two explanations here: 1) the error scale found could be unusually small, which is why little to no difference is observed; 2) the error scale is perfectly fine, and the little difference observed here supports it; changing our parameters' values by a scale of their errors yield similar results, which is what should be expected. The problem would have been if the difference plotted was large. It would mean that changing our parameters' values on a scale of their errors changes significantly the accuracy of the model. This would question the choice of the error.

g) Now we use MCMC to find the best fit parameters. We begin with the set of parameters found in d). The step for each parameter is found through the covariance matrix. Each step contributes a little evolution found through the Cholesky decomposition of the covariant matrix. The parameters' values are then updated, which yields a new χ^2 value. The step is randomly accepted based on the difference between the previous and newly calculated χ^2 . The process then proceeds to repeat, for a specified amount of steps.

```
In [51]: def chisq(p, t, signal, noise):
y, grad = lorentzian_3lor(p, t)
chisq = np.sum((signal - y/noise)**2)
return chisq

In [60]: def MCMC(params, cov, nsteps, noise):
nparams = chisq(params, t, d, noise)
nparams = len(params)
chain = np.zeros((nsteps, nparams))
chi_vec = np.zeros(nsteps)
counts = 0
cholesky_mat = np.linalg.cholesky(cov)

for i in range(nsteps):
if i == int(nsteps/4):
print('25% Complete.')

if i == int(nsteps/2):
print('50% Complete.')

if i == int(3*nsteps/4):
print('75% Complete.')

dparams = cholesky_mat@np.random.randn(nparams)
trial_params = params + dparams
trial_chisq = chisq(trial_params, t, d, noise)
dchisq = trial_chisq - chi_0
prob_accept = np.exp(-0.5*dchisq)
accept = np.random.rand(1) < prob_accept

if accept:
counts += 1
params = trial_params
chi_0 = trial_chisq

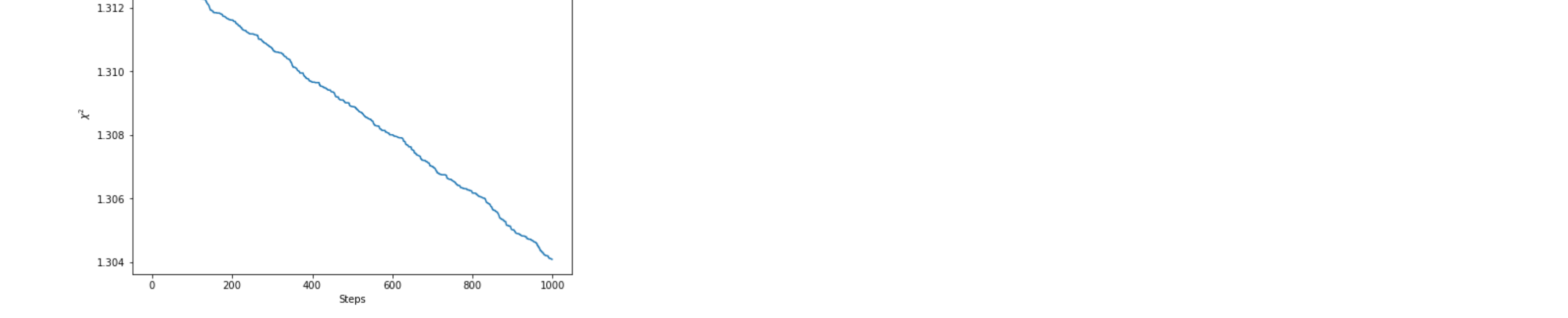
chain[i,:] = params
chi_vec[i] = chi_0

return chain, chi_vec
```

```
In [61]: chain1, chi_vec1 = MCMC(p_newd, cov_d, steps, sc_signals)
<ipython-input-68-b62fab38577>:38: RuntimeWarning: overflow encountered in exp
prob_accept = np.exp(-0.5*dchisq)
25% Complete.
40% Complete.
75% Complete.
Out of 1800 steps, 488 were accepted ( 27.1 %)
```

```
In [64]: plt.figure(figsize = (8,6))
plt.plot(chain1, chi_vec1)
plt.title('chi^2 evolution')
plt.xlabel('steps')
plt.ylabel('chi^2chi^2s')

Out[64]: Text(0, 0.5, '$\chi^2$chi^2s')
```

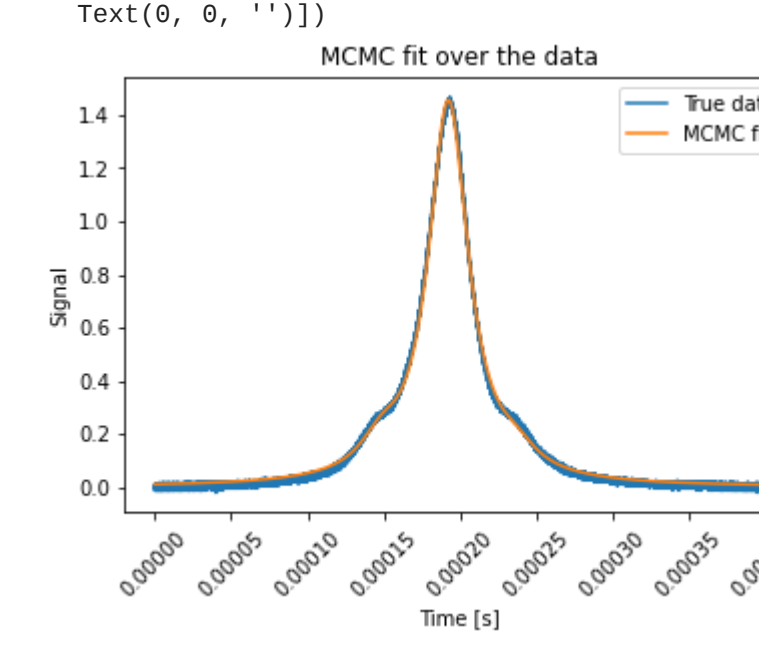


It doesn't seem like my chain converged. The χ^2 decreases linearly all throughout. I would have expected an important drop before stabilization, but it's not the case. Also, the χ^2 values are very large. I wonder why the chain behaved like this. I used with the initial guess parameters (which are more rough estimates). But the same was observed.

```
In [70]: params_mcmc = chain1[-1]
y1g, grad1g = lorentzian_3lor(params_mcmc, t)

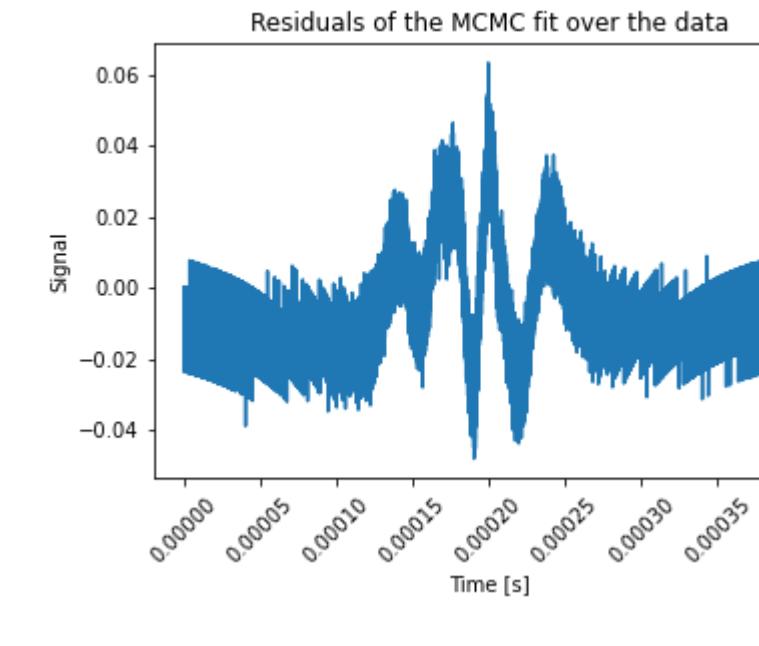
In [72]: plt.plot(t, d, label = "True data")
plt.plot(t, y1g, label = "MCMC fit")
plt.xlabel('Time [s]')
plt.ylabel('Signal')
plt.title('MCMC fit over the data')
plt.legend()
plt.xticks(rotation = 45)

Out[72]: (array([[-5.0e-05, 0.8e+08, 5.0e-05, 1.9e-04, 1.5e-04, 2.0e-04,
1.5e-04, 3.0e-04, 3.5e-04, 4.0e-04, 4.5e-04]]),
[Text(0, 0, ''),
Text(0, 0, ''),
Text(0, 0, ''),
Text(0, 0, ''),
Text(0, 0, ''),
Text(0, 0, ''),
Text(0, 0, ''),
Text(0, 0, ''),
Text(0, 0, ''),
Text(0, 0, '')])

MCMC fit over the data

```

```
In [74]: plt.plot(t, d - y1g)
plt.xlabel('Time [s]')
plt.ylabel('Residuals')
plt.title('Residuals of the MCMC fit over the data')
plt.xticks(rotation = 45)

Out[74]: (array([[-5.0e-05, 0.8e+08, 5.0e-05, 1.9e-04, 1.5e-04, 2.0e-04,
1.5e-04, 3.0e-04, 3.5e-04, 4.0e-04, 4.5e-04]]),
[Text(0, 0, ''),
Text(0, 0, ''),
Text(0, 0, ''),
Text(0, 0, ''),
Text(0, 0, ''),
Text(0, 0, ''),
Text(0, 0, ''),
Text(0, 0, ''),
Text(0, 0, ''),
Text(0, 0, '')])

Residuals of the MCMC fit over the data

```

b) To find the width of the cavity resonance, we can do $\frac{d\omega}{dZ} \sim 9$ GHz

With Newton's method, we found $dZ = 4.456716295451412e-05 \pm 2.718178487099205e-09 = 0.000044567 \pm 0.000000003$, and $\omega = 1.0065109423762263e-05 \pm 4.081245203269164e-10 = 0.000100651 \pm 0.000000004$ GHz,

which gives a cavity resonance of $Zd\omega = 24.967 \pm 0.002$ s (add relative errors on dZ and ω to find the relative error on the cavity resonance).