Guillaume Payeur (260929164)

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
%matplotlib inline
import matplotlib as mpl
mpl.rcParams['figure.dpi'] = 200
plt.rcParams.update({"text.usetex": True})
```

Q1

a)

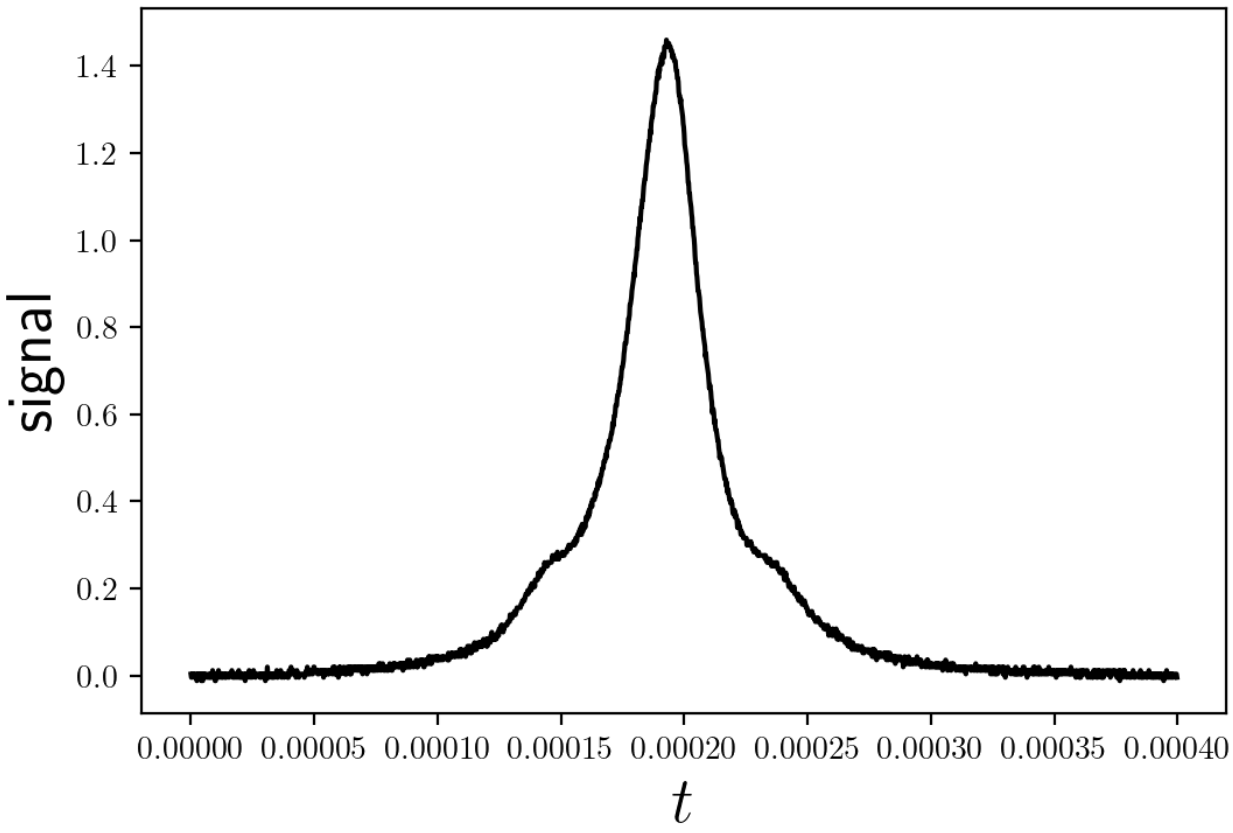
Begin by loading and plotting the raw data

In [2]:
```
# Loading data
stuff=np.load('mcmc/sidebands.npz')
t=stuff['time'][::50]
d=stuff['signal'][::50]
```

Note: I am keeping 1 data point out of 50 as means to lower memory requirements. The data is still very densly sampled.

In [3]:
```
plt.plot(t,d,color='black')
plt.xlabel('$t$',fontsize=20)
plt.ylabel('signal',fontsize=20)
```

Out[3]:
```
Text(0, 0.5, 'signal')
```



Fit the amplitude a, width w and center t_0 of the Lorentzian using Newton's method with initial guess $a = 1.4, w = 0.0001, t_0 = 0.0002$.

In [4]:

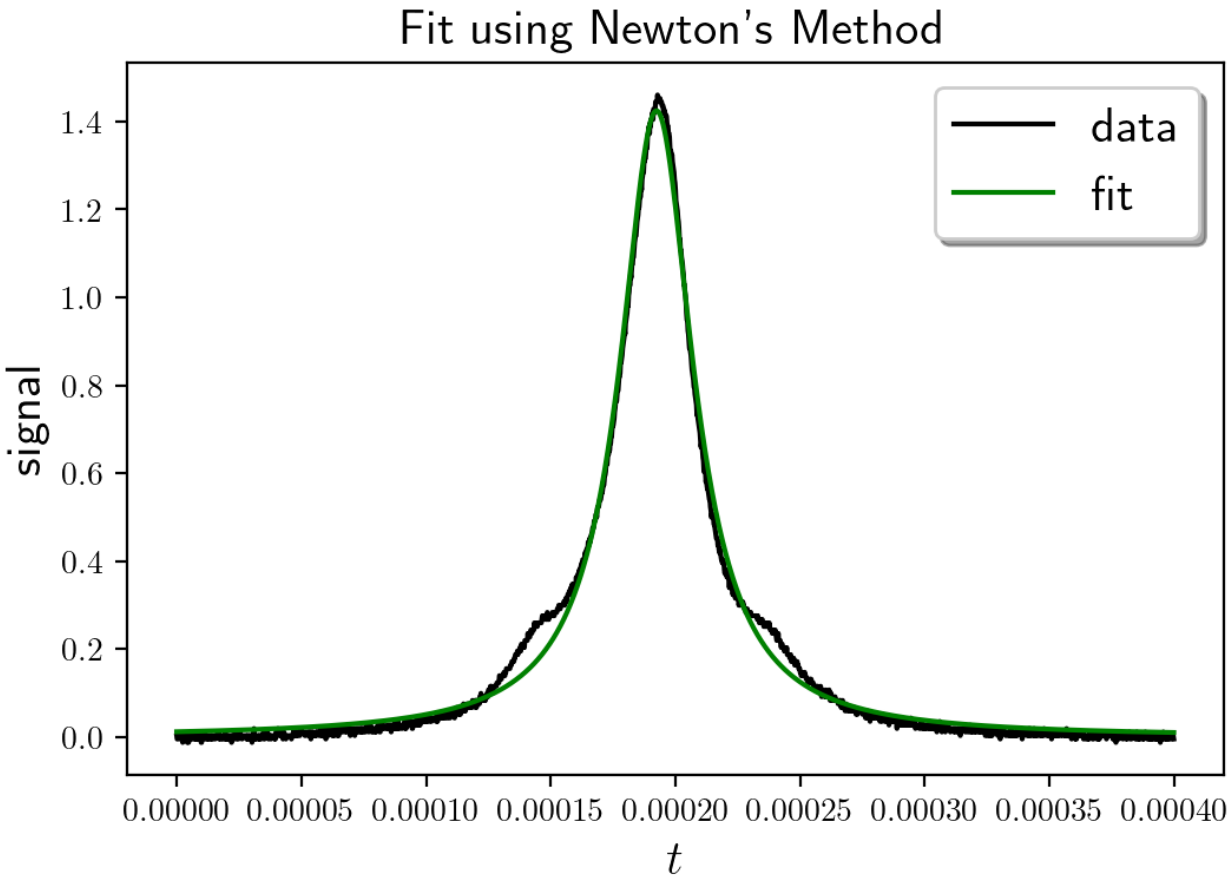
```
# Initial guess for parameters
a = 1.4
w = 0.0001
t0 = 0.0002
m = np.array([a,w,t0])

# Doing 20 itterations of Newton's method
for i in range(20):
    # Computing residuals
    Am = m[0]/(1+((t-m[2])**2)/(m[1]**2))
    r = d-Am

    # Computing gradient of A
    A_a = 1/(1+((t-m[2])**2)/(m[1]**2))
    A_w = (2*m[0]*(t-m[2])**2)/((m[1]**3)*((1+((t-m[2])**2)/(m[1]**2)))**2)
    A_t0 = (2*m[0]*(t-m[2]))/((m[1]**2)*(1+((t-m[2])**2/(m[1])**2))**2)
    A_m = np.zeros((A_a.shape[0],3))
    A_m[:,0] = A_a
    A_m[:,1] = A_w
    A_m[:,2] = A_t0

    # Computing delta m
    delta_m = np.linalg.inv(A_m.T@A_m)@A_m.T@r
    m = m + delta_m

# Printing best fit parameters
print(m)
```

```
[1.42416244e+00 1.78925605e-05 1.92360899e-04]
```

Making a plot of the data and the fit to make sure it worked

In [5]:
```
plt.plot(t,d,color='black',label='data')
plt.plot(t,m[0]/(1+((t-m[2])**2)/(m[1]**2)),color='green',label='fit')
plt.xlabel('$t$',fontsize=15)
plt.ylabel('signal',fontsize=15)
plt.title("Fit using Newton's Method",fontsize=15)
plt.legend(loc=0,frameon=True,shadow=True,fontsize=15)
```

Out[5]:
```
<matplotlib.legend.Legend at 0x179ada545c8>
```



The fit looks good enough to conclude that Newton's Method worked. The parameters we got are

$$a = 1.42 \tag{1}$$
$$w = 1.79\text{x}10^{-5} \tag{2}$$
$$t_0 = 1.92\text{x}10^{-4} \tag{3}$$

b)

We begin by estimating the uncertainty on each data point. For that we pick datapoints near $t = 0$ where the curve is almost flat, especially on short time scales, and find the square root of their variance. Under the assumption that the curve is flat, the variance is purely due to noise, hence this gives an estimate of the error.

In [6]:
```
sigma = np.std(d[0:int(d.shape[0]/10)])
print(sigma)
```

```
0.004970451303634731
```

So we approximate the uncertainty on each data point as being 0.005. Now we can find the uncertainty on the model parameters

In [7]:
```
# Initial parameters
a = 1.4
w = 0.0001
t0 = 0.0002
m = np.array([a,w,t0])

# Doing 20 itterations of Newton's method
for i in range(20):
    # Computing residuals
    Am = m[0]/(1+((t-m[2])**2)/(m[1]**2))
    r = d-Am

    # Computing gradient of A
    A_a = 1/(1+((t-m[2])**2)/(m[1]**2))
    A_w = (2*m[0]*(t-m[2])**2)/((m[1]**3)*((1+((t-m[2])**2)/(m[1]**2)))**2)
    A_t0 = (2*m[0]*(t-m[2]))/((m[1]**2)*(1+((t-m[2])**2/(m[1])**2))**2)
    A_m = np.zeros((A_a.shape[0],3))
    A_m[:,0] = A_a
    A_m[:,1] = A_w
    A_m[:,2] = A_t0

    # Computing delta m
    delta_m = np.linalg.inv(A_m.T@A_m)@A_m.T@r
    m = m + delta_m

# Calculating uncertainty on best fit parameters
sigma = 0.005
Ninv = np.eye(t.shape[0])/(sigma**2)
cov = np.linalg.inv(A_m.T@Ninv@A_m)

# Printing uncertainty on best fit parameters
print(np.sqrt(np.diag(cov)))
```

```
[5.96668976e-04 1.06125158e-08 7.49409704e-09]
```

So the fit parameters with uncertainty are

$$a = 1.42 \pm 5.97\text{x}10^{-4} \tag{4}$$
$$w = 1.79\text{x}10^{-5} \pm 1.06\text{x}10^{-8} \tag{5}$$
$$t_0 = 1.92\text{x}10^{-4} \pm 7.49\text{x}10^{-8} \tag{6}$$

But note that we shouldn't trust these parameters or these errors since the model is not a good fit to the data: clearly the residuals are correlated without need to plot them.

c)

We redo part a) with numerical derivatives.

In [8]:
```
# Function to take numerical derivative of fun at x.
def ndiff(fun,x):
    # machine precision
    eps = 1e-16
    delta = np.sqrt(eps)

    # Calculating derivative
    deriv = 1/(2*delta)*(fun(x+delta)-fun(x-delta))
    return deriv

# Initial guess for parameters
a = 1.4
w = 0.0001
t0 = 0.0002
m = np.array([a,w,t0])

# Doing 20 itterations of Newton's method
for i in range(20):
    # Computing residuals
    Am = m[0]/(1+((t-m[2])**2)/(m[1]**2))
    r = d-Am

    # Computing gradient of A numerically
    def fun_a(a):
        return a/(1+((t-m[2])**2)/(m[1]**2))
    A_a = ndiff(fun_a,m[0])
    def fun_w(w):
        return m[0]/(1+((t-m[2])**2)/(w**2))
    A_w = ndiff(fun_w,m[1])
    def fun_t0(t0):
        return m[0]/(1+((t-t0)**2)/(m[1]**2))
    A_t0 = ndiff(fun_t0,m[2])

    A_m = np.zeros((A_a.shape[0],3))
    A_m[:,0] = A_a
    A_m[:,1] = A_w
    A_m[:,2] = A_t0

    # Computing delta m
    delta_m = np.linalg.inv(A_m.T@A_m)@A_m.T@r
    m = m + delta_m

# Printing best fit parameters
print(m)
```

```
[1.42416244e+00 1.78925605e-05 1.92360899e-04]
```

The parameters we got now are

$$a = 1.42 \tag{7}$$
$$w = 1.79\text{x}10^{-5} \tag{8}$$
$$t_0 = 1.92\text{x}10^{-4} \tag{9}$$

and the difference between them and the parameters we got before is less than 1 time the standard error on the parameters determined before, so the difference is not statistically significant. Both ways of proceeding worked equally well.

d)

We repeat part c) but with three Lorentzians

In [9]:

```
# Function to take numerical derivative of fun at x
def ndiff(fun,x):
    # machine precision
    eps = 1e-16
    delta = np.sqrt(eps)

    # Calculating derivative
    deriv = 1/(2*delta)*(fun(x+delta)-fun(x-delta))
    return deriv

# Initial guess for parameters
a = 1.42
b = 0.1
c = 0.1
w = 0.000018
t0 = 0.000192
dt = 0.00004
m = np.array([a,b,c,w,t0,dt])

# Doing 20 itterations of Newton's method
for i in range(20):
    # Computing residuals
    Am = m[0]/(1+((t-m[4])**2)/(m[3]**2)) + m[1]/(1+((t-m[4]+m[5])**2)/(m[3]**2)) + m[2
    r = d-Am

    # Computing gradient of A numerically
    def fun_a(a):
        return a/(1+((t-m[4])**2)/(m[3]**2)) + m[1]/(1+((t-m[4]+m[5])**2)/(m[3]**2)) +
    A_a = ndiff(fun_a,m[0])
    def fun_b(b):
        return m[0]/(1+((t-m[4])**2)/(m[3]**2)) + b/(1+((t-m[4]+m[5])**2)/(m[3]**2)) +
    A_b = ndiff(fun_b,m[1])
    def fun_c(c):
        return m[0]/(1+((t-m[4])**2)/(m[3]**2)) + m[1]/(1+((t-m[4]+m[5])**2)/(m[3]**2))
    A_c = ndiff(fun_c,m[2])
    def fun_w(w):
        return m[0]/(1+((t-m[4])**2)/(w**2)) + m[1]/(1+((t-m[4]+m[5])**2)/(w**2)) + m[2
    A_w = ndiff(fun_w,m[3])
    def fun_t0(t0):
        return m[0]/(1+((t-t0)**2)/(m[3]**2)) + m[1]/(1+((t-t0+m[5])**2)/(m[3]**2)) + m
    A_t0 = ndiff(fun_t0,m[4])
    def fun_dt(dt):
        return m[0]/(1+((t-m[4])**2)/(m[3]**2)) + m[1]/(1+((t-m[4]+dt)**2)/(m[3]**2)) +
    A_dt = ndiff(fun_dt,m[5])

    A_m = np.zeros((A_a.shape[0],6))
    A_m[:,0] = A_a
    A_m[:,1] = A_b
    A_m[:,2] = A_c
    A_m[:,3] = A_w
    A_m[:,4] = A_t0
    A_m[:,5] = A_dt

    # Computing delta m
```

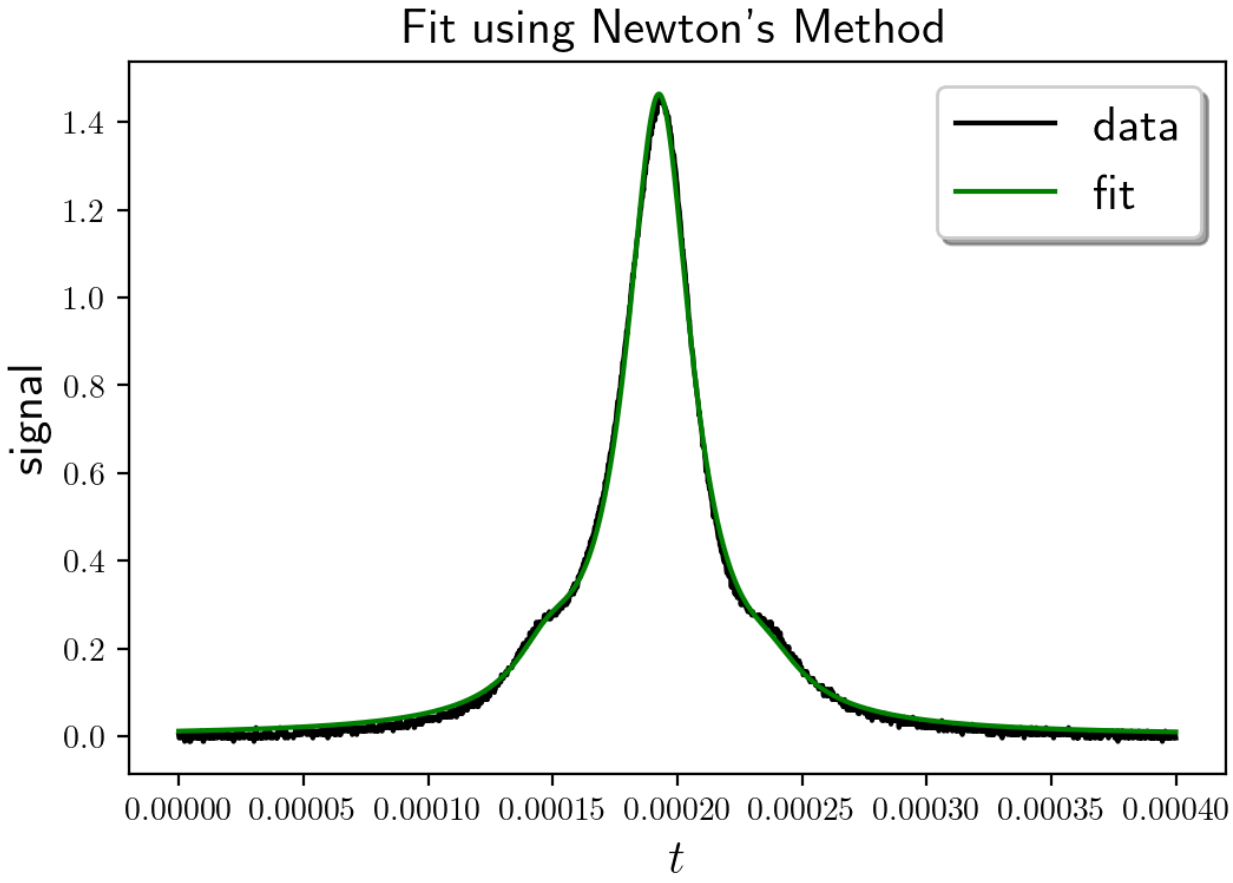
```
    delta_m = np.linalg.inv(A_m.T@A_m)@A_m.T@r
    m = m + delta_m

# Printing best fit parameters
print(m)
```

```
[1.44461789e+00 1.04728584e-01 6.54256989e-02 1.60245012e-05
 1.92580498e-04 4.45420709e-05]
```

Plotting the fit to make sure it worked

In [10]:
```
plt.plot(t,d,color='black',label='data')
plt.plot(t,m[0]/(1+((t-m[4])**2)/(m[3]**2)) + m[1]/(1+((t-m[4]+m[5])**2)/(m[3]**2)) + m
plt.xlabel('$t$',fontsize=15)
plt.ylabel('signal',fontsize=15)
plt.title("Fit using Newton's Method",fontsize=15)
plt.legend(loc=0,frameon=True,shadow=True,fontsize=15)
```

Out[10]: <matplotlib.legend.Legend at 0x179ad0278c8>



The fit looks better than before. The parameters we got are

$$a = 1.44 \tag{10}$$
$$b = 1.05\text{x}10^{-1} \tag{11}$$
$$c = 6.54\text{x}10^{-2} \tag{12}$$
$$w = 1.60\text{x}10^{-5} \tag{13}$$
$$t_0 = 1.93\text{x}10^{-4} \tag{14}$$
$$dt = 4.45\text{x}10^{-5} \tag{15}$$

Now we find the uncertainties as we did before

In [11]:

```
# Function to take numerical derivative of fun at x
def ndiff(fun,x):
    # machine precision
    eps = 1e-16
    delta = np.sqrt(eps)

    # Calculating derivative
    deriv = 1/(2*delta)*(fun(x+delta)-fun(x-delta))
    return deriv

# Initial guess for parameters
a = 1.42
b = 0.1
c = 0.1
w = 0.000018
t0 = 0.000192
dt = 0.00004
m = np.array([a,b,c,w,t0,dt])

# Doing 20 itterations of Newton's method
for i in range(20):
    # Computing residuals
    Am = m[0]/(1+((t-m[4])**2)/(m[3]**2)) + m[1]/(1+((t-m[4]+m[5])**2)/(m[3]**2)) + m[2
    r = d-Am

    # Computing gradient of A numerically
    def fun_a(a):
        return a/(1+((t-m[4])**2)/(m[3]**2)) + m[1]/(1+((t-m[4]+m[5])**2)/(m[3]**2)) +
    A_a = ndiff(fun_a,m[0])
    def fun_b(b):
        return m[0]/(1+((t-m[4])**2)/(m[3]**2)) + b/(1+((t-m[4]+m[5])**2)/(m[3]**2)) +
    A_b = ndiff(fun_b,m[1])
    def fun_c(c):
        return m[0]/(1+((t-m[4])**2)/(m[3]**2)) + m[1]/(1+((t-m[4]+m[5])**2)/(m[3]**2))
    A_c = ndiff(fun_c,m[2])
    def fun_w(w):
        return m[0]/(1+((t-m[4])**2)/(w**2)) + m[1]/(1+((t-m[4]+m[5])**2)/(w**2)) + m[2
    A_w = ndiff(fun_w,m[3])
    def fun_t0(t0):
        return m[0]/(1+((t-t0)**2)/(m[3]**2)) + m[1]/(1+((t-t0+m[5])**2)/(m[3]**2)) + m
    A_t0 = ndiff(fun_t0,m[4])
    def fun_dt(dt):
        return m[0]/(1+((t-m[4])**2)/(m[3]**2)) + m[1]/(1+((t-m[4]+dt)**2)/(m[3]**2)) +
    A_dt = ndiff(fun_dt,m[5])

    A_m = np.zeros((A_a.shape[0],6))
    A_m[:,0] = A_a
    A_m[:,1] = A_b
    A_m[:,2] = A_c
    A_m[:,3] = A_w
    A_m[:,4] = A_t0
    A_m[:,5] = A_dt

    # Computing delta m
    delta_m = np.linalg.inv(A_m.T@A_m)@A_m.T@r
    m = m + delta_m
```

```
# Calculating uncertainty on best fit parameters
Ninv = np.eye(t.shape[0])/(sigma**2)
cov = np.linalg.inv(A_m.T@Ninv@A_m)

# Printing uncertainty on best fit parameters
print(np.sqrt(np.diag(cov)))
```

```
[6.47021717e-04 6.16052665e-04 6.03327894e-04 1.36395647e-08
 7.62930004e-09 9.12622319e-08]
```

So the fit parameters with uncertainty are

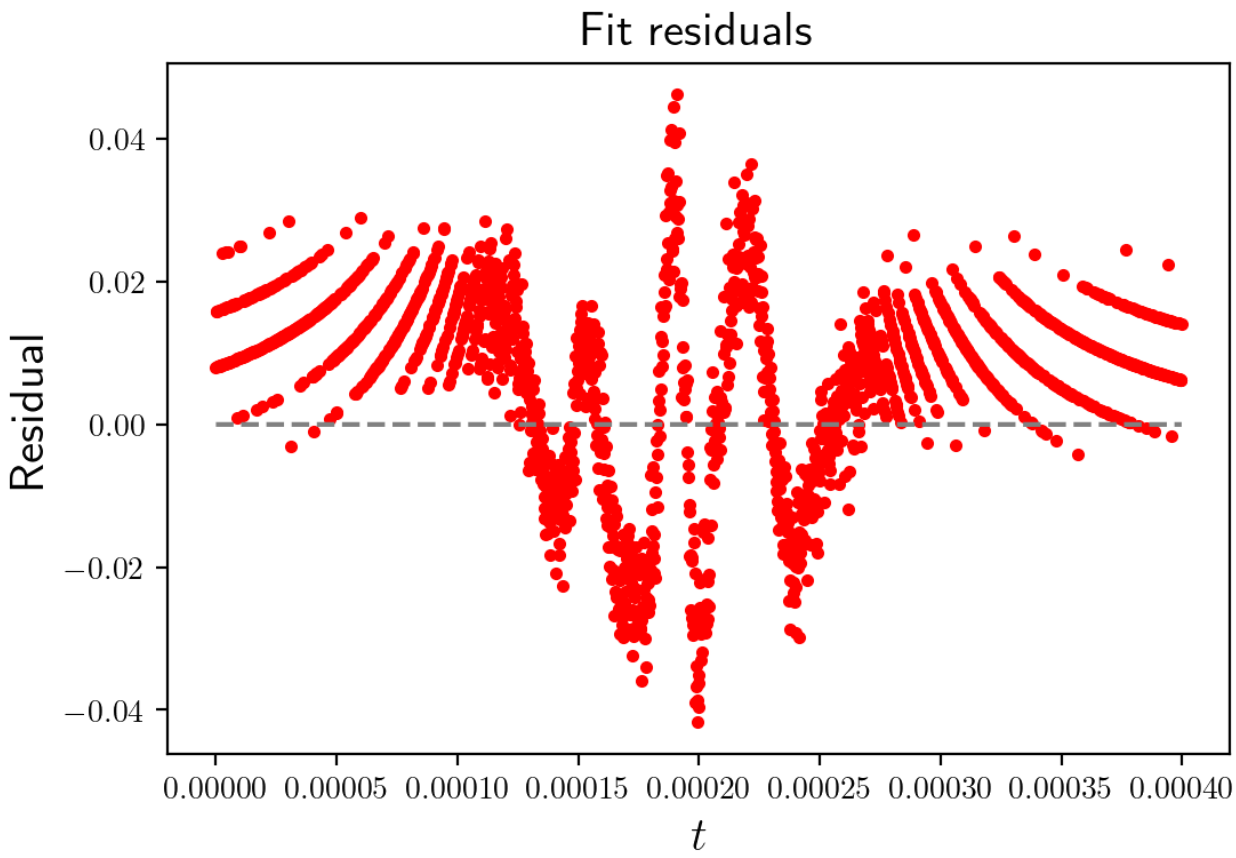
$$a = 1.44 \pm 6.47\text{x}10^{-4} \tag{16}$$
$$b = 1.04\text{x}10^{-1} \pm 6.16\text{x}10^{-4} \tag{17}$$
$$c = 6.45\text{x}10^{-2} \pm 6.03\text{x}10^{-4} \tag{18}$$
$$w = 1.61\text{x}10^{-5} \pm 1.36\text{x}10^{-8} \tag{19}$$
$$t_0 = 1.93\text{x}10^{-4} \pm 7.63\text{x}10^{-8} \tag{20}$$
$$dt = 4.46\text{x}10^{-5} \pm 9.13\text{x}10^{-7} \tag{21}$$

e)

Now we plot the residuals.

In [12]:
```
plt.plot(t,m[0]/(1+((t-m[4])**2)/(m[3]**2)) + m[1]/(1+((t-m[4]+m[5])**2)/(m[3]**2)) + m
plt.plot(t,t*0,color='gray',ls='--')
plt.xlabel('$t$',fontsize=15)
plt.ylabel('Residual',fontsize=15)
plt.title("Fit residuals",fontsize=15)
```

Out[12]:
```
Text(0.5, 1.0, 'Fit residuals')
```



The residuals are clearly correlated, so no, the error bars I got by assuming the data are independent with uniform variance are not a complete description of the data.

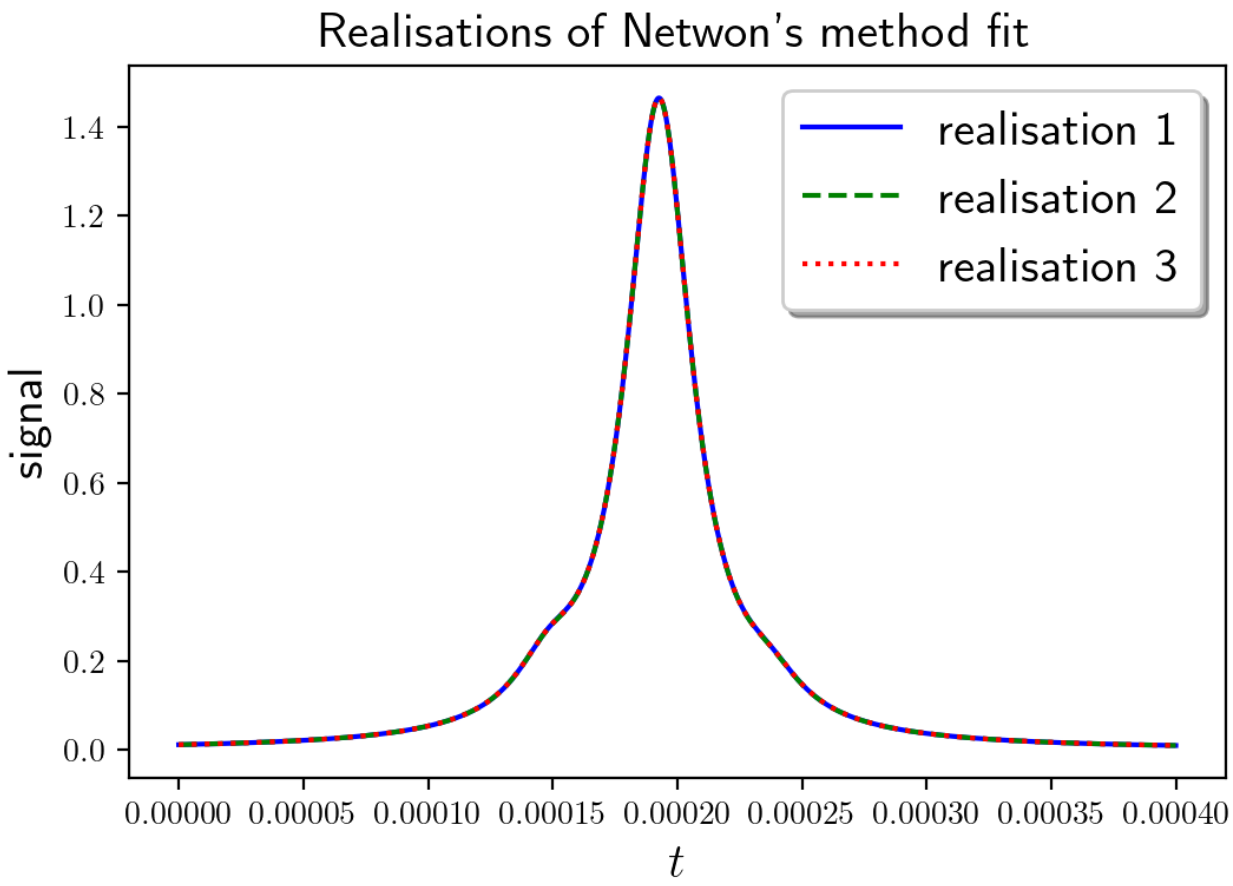
f)

We generate realisations for the parameter errors using Cholesky decomposition. Here I plot three realisations on top of each other

In [13]:
```
L = np.linalg.cholesky(cov)
n1 = L@np.random.randn(6)
n2 = L@np.random.randn(6)
n3 = L@np.random.randn(6)
m1 = m+n1
m2 = m+n2
m3 = m+n3
```

In [14]:
```
plt.plot(t,m1[0]/(1+((t-m1[4])**2)/(m1[3]**2)) + m1[1]/(1+((t-m1[4]+m1[5])**2)/(m1[3]**
plt.plot(t,m2[0]/(1+((t-m2[4])**2)/(m2[3]**2)) + m2[1]/(1+((t-m2[4]+m2[5])**2)/(m2[3]**
plt.plot(t,m3[0]/(1+((t-m3[4])**2)/(m3[3]**2)) + m3[1]/(1+((t-m3[4]+m3[5])**2)/(m3[3]**
plt.xlabel('$t$',fontsize=15)
plt.ylabel('signal',fontsize=15)
plt.title("Realisations of Netwon's method fit",fontsize=15)
plt.legend(loc=0,frameon=True,shadow=True,fontsize=15)
```

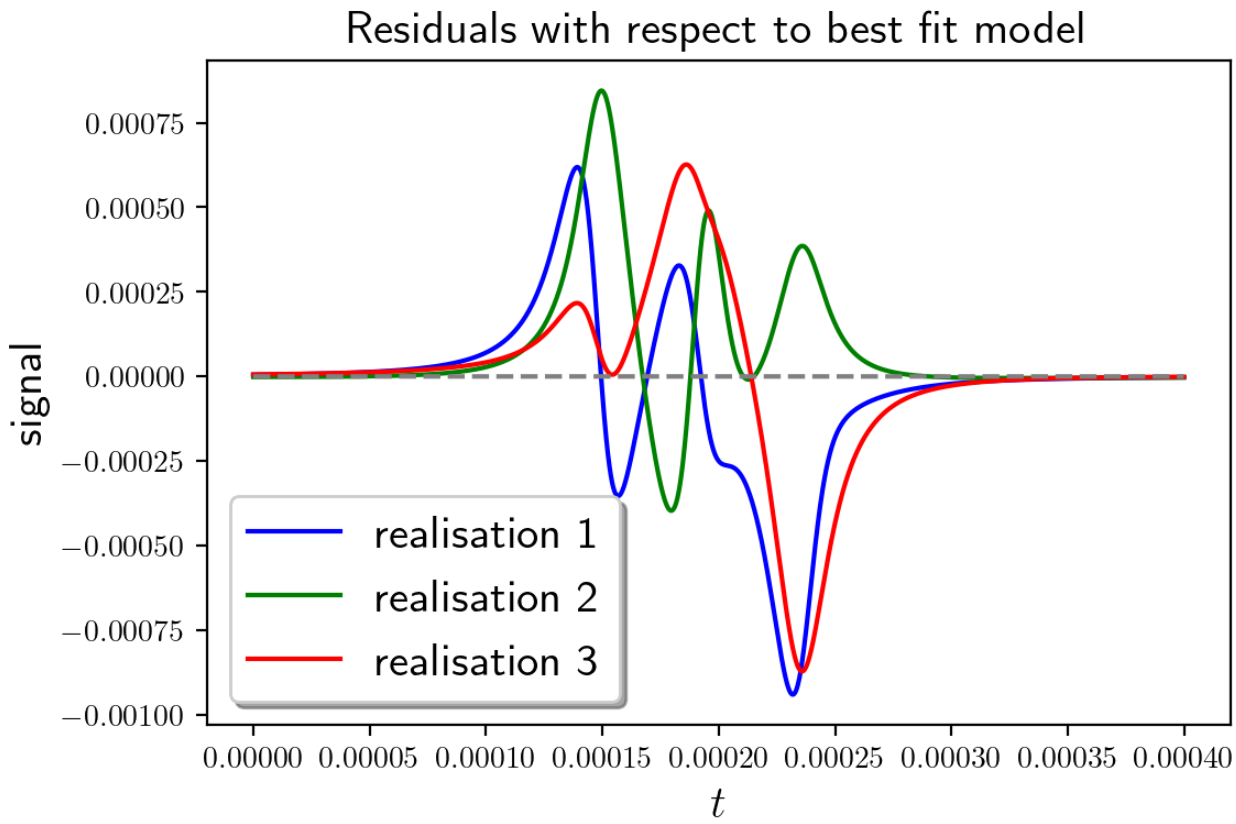
Out[14]:
```
<matplotlib.legend.Legend at 0x179ad15af08>
```



We can't really see the difference, but here's the residuals (with respect to the best fit model)

In [15]:
```
d_best_fit = m[0]/(1+((t-m[4])**2)/(m[3]**2)) + m[1]/(1+((t-m[4]+m[5])**2)/(m[3]**2)) +
d_1 = m1[0]/(1+((t-m1[4])**2)/(m1[3]**2)) + m1[1]/(1+((t-m1[4]+m1[5])**2)/(m1[3]**2)) +
d_2 = m2[0]/(1+((t-m2[4])**2)/(m2[3]**2)) + m2[1]/(1+((t-m2[4]+m2[5])**2)/(m2[3]**2)) +
d_3 = m3[0]/(1+((t-m3[4])**2)/(m3[3]**2)) + m3[1]/(1+((t-m3[4]+m3[5])**2)/(m3[3]**2)) +

plt.plot(t,d_best_fit-d_1,color='blue',label='realisation 1')
plt.plot(t,d_best_fit-d_2,color='green',label='realisation 2')
plt.plot(t,d_best_fit-d_3,color='red',label='realisation 3')
plt.plot(t,t*0,color='gray',ls='--')
plt.xlabel('$t$',fontsize=15)
plt.ylabel('signal',fontsize=15)
plt.title("Residuals with respect to best fit model",fontsize=15)
plt.legend(loc=0,frameon=True,shadow=True,fontsize=15)
```

Out[15]:
```
<matplotlib.legend.Legend at 0x179ad206a88>
```



now I generate more realisations of the model and calculate the χ^2 every time. I take the mean value of the increase in χ^2.

In [16]:

```
L = np.linalg.cholesky(cov)
# array to contain each value of chi2
chi2_diff = np.zeros((100))
for i in range(100):
    # creating realisation of model parameters
    n0 = L@np.random.randn(6)
    m0 = m+n0
    # creating realisation of model
    d0 = m0[0]/(1+((t-m0[4])**2)/(m0[3]**2)) + m0[1]/(1+((t-m0[4]+m0[5])**2)/(m0[3]**2)
    chi2_best_fit = np.sum((d_best_fit-d)**2/sigma**2)/(t.shape[0]-6)
    # calculating chi2 for the realisation of the model
    chi2_realisation = np.sum((d0-d)**2/sigma**2)/(t.shape[0]-6)
    # calculating chi2 difference wrt best fit
    chi2_diff[i] = np.abs(chi2_best_fit-chi2_realisation)
print(np.mean(chi2_diff))
```

```
0.0030335580411422216
```

The typical difference in χ^2 is 0.003, which is much less than 1. You would normally expect the typical difference to be closer to 1, but here the model is still not a good fit to the data (residuals are highly correlated, χ^2 is much larger than 1), so all bets are off.

g)

Now we do the same fit but using an MCMC. I'm adapting code written in class by Prof. Sievers. The step size is set to be the standard error on the fit parameters as found in d).

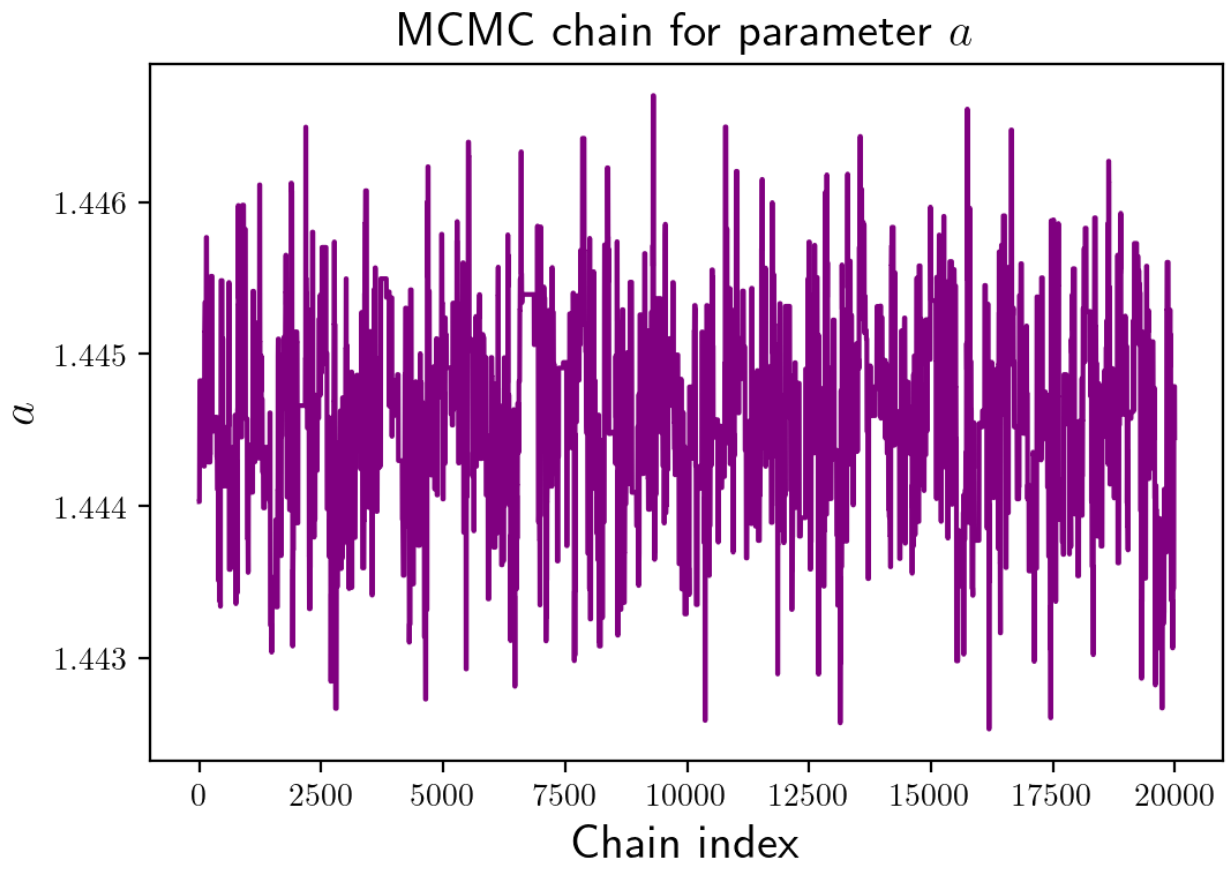
In [17]:
```
# Function to compute the model given model parameters
def three_lorentzian(m,t):
    return m[0]/(1+((t-m[4])**2)/(m[3]**2)) + m[1]/(1+((t-m[4]+m[5])**2)/(m[3]**2)) + m

# Function to compute the chi2 for a given model
def chisq(m,x,y,noise):
    pred=three_lorentzian(m,t)
    return np.sum(((y-pred)/noise)**2)

# Function to make an MCMC chain
def mcmc(pars,step_size,x,y,fun,noise,nstep=1000):
    # Initial chi2
    chi_cur=fun(pars,x,y,noise)
    # Making array to hold chains and chi2
    npar=pars.shape[0]
    chain=np.zeros([nstep,npar])
    chivec=np.zeros(nstep)

    for i in range(nstep):
        trial_pars=pars+step_size*np.random.randn(npar)
        trial_chisq=fun(trial_pars,x,y,noise)
        delta_chisq=trial_chisq-chi_cur
        accept_prob=np.exp(-0.5*delta_chisq)
        accept=np.random.rand(1)<accept_prob
        if accept:
            pars=trial_pars
            chi_cur=trial_chisq
        chain[i,:]=pars
        chivec[i]=chi_cur
    return chain,chivec
```

In [18]:
```
pars = np.array([1.44403041e+00, 1.03678060e-01, 6.44510546e-02, 1.60517917e-05, 1.9258
step_size = np.array([9.14128147e-04, 8.70429036e-04, 8.52345103e-04, 1.93273356e-08, 1
chain, chivec = mcmc(pars,step_size,t,d,chisq,sigma,nstep=20000)
```

To check that the MCMC converged, I plot the chain corresponding to every parameter.

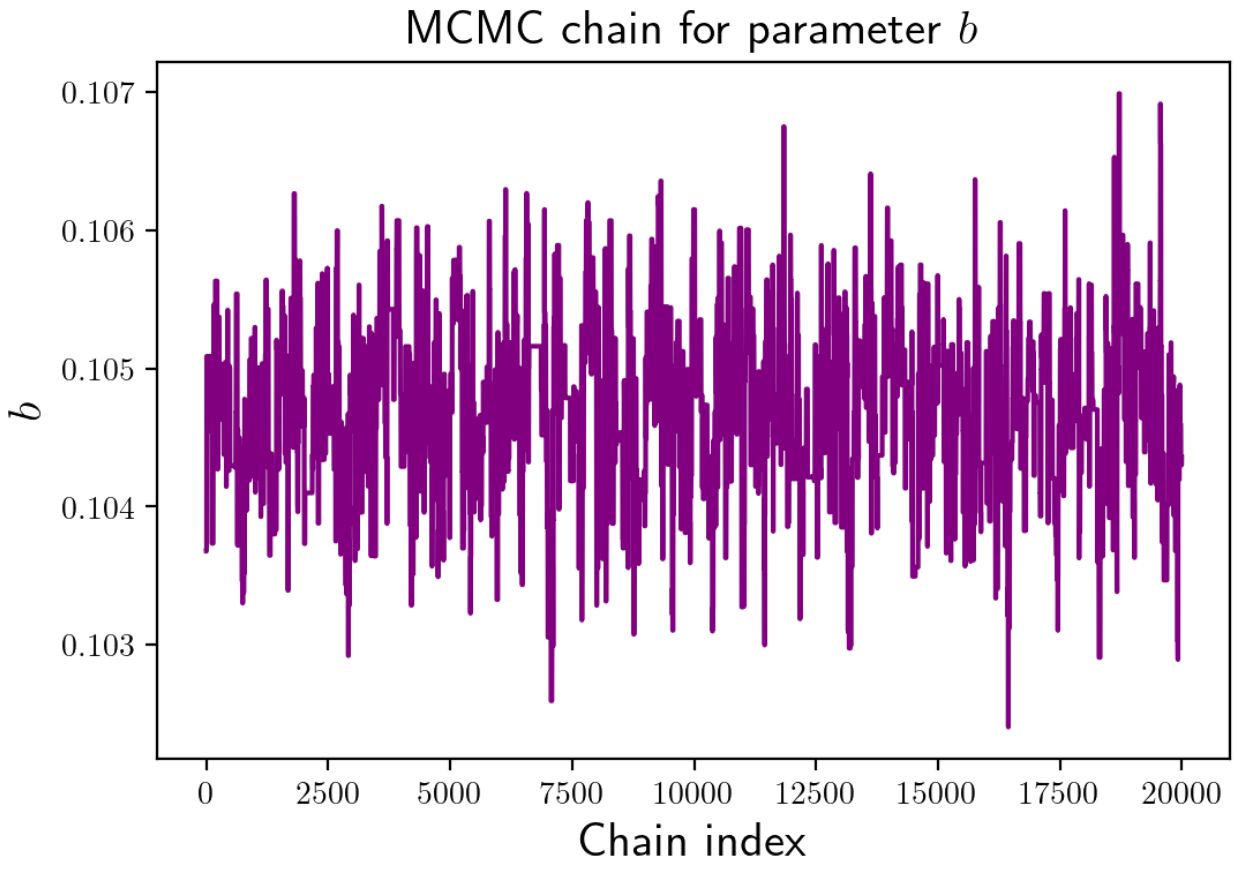
In [19]:
```
plt.plot(chain[:,0],color='purple')
plt.xlabel('Chain index',fontsize=15)
plt.ylabel('$a$',fontsize=15)
plt.title('MCMC chain for parameter $a$',fontsize=15)
```

Out[19]:
```
Text(0.5, 1.0, 'MCMC chain for parameter $a$')
```



In [20]:
```
plt.plot(chain[:,1],color='purple')
plt.xlabel('Chain index',fontsize=15)
plt.ylabel('$b$',fontsize=15)
plt.title('MCMC chain for parameter $b$',fontsize=15)
```

Out[20]:
```
Text(0.5, 1.0, 'MCMC chain for parameter $b$')
```

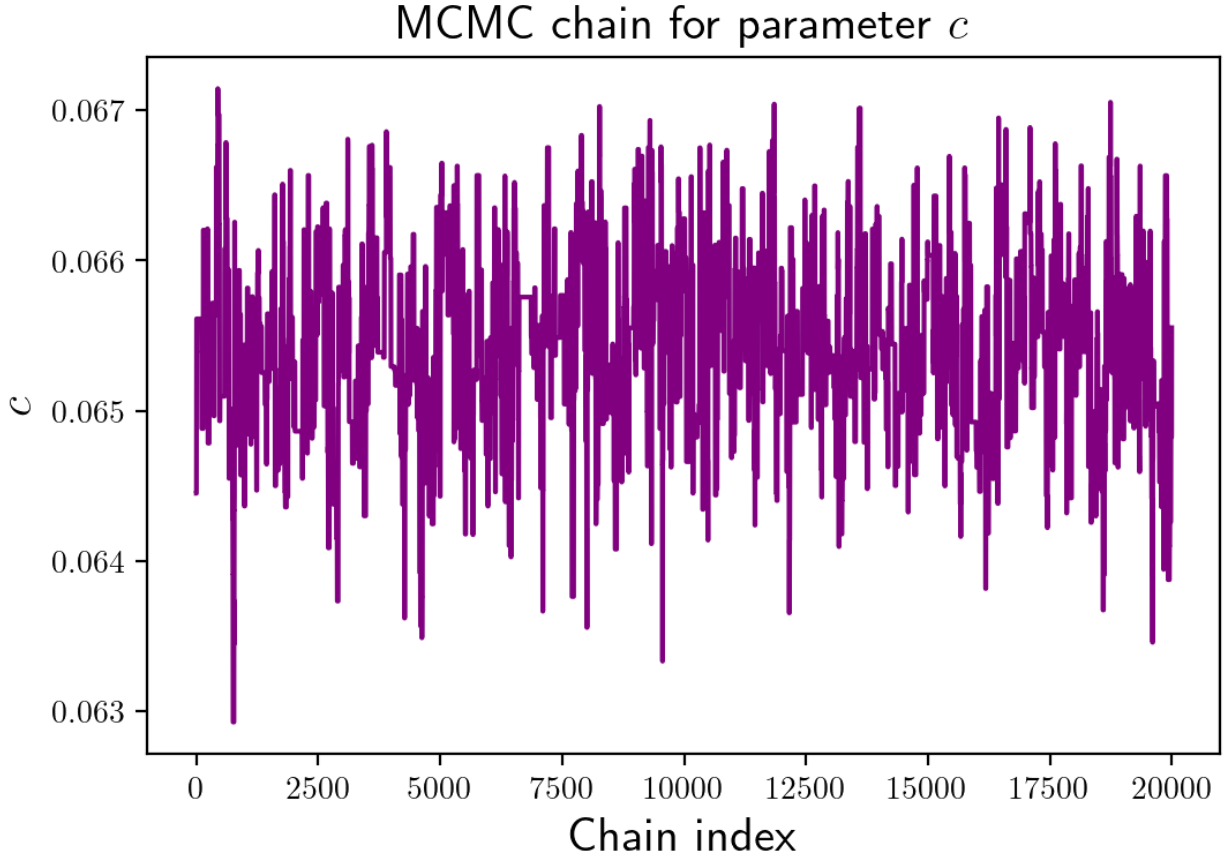


In [21]:

```
plt.plot(chain[:,2],color='purple')
plt.xlabel('Chain index',fontsize=15)
plt.ylabel('$c$',fontsize=15)
plt.title('MCMC chain for parameter $c$',fontsize=15)
```

Out[21]:

```
Text(0.5, 1.0, 'MCMC chain for parameter $c$')
```

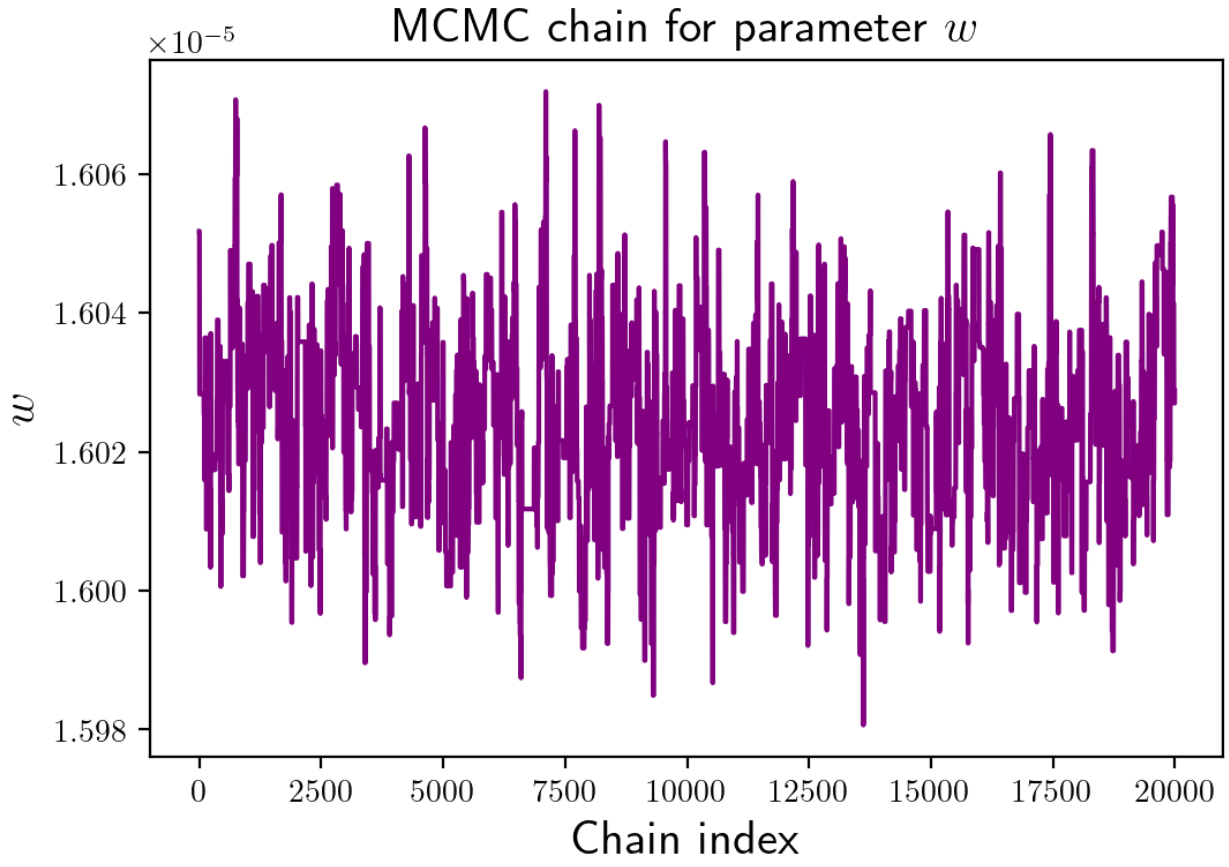


In [22]:

```
plt.plot(chain[:,3],color='purple')
plt.xlabel('Chain index',fontsize=15)
plt.ylabel('$w$',fontsize=15)
plt.title('MCMC chain for parameter $w$',fontsize=15)
```

Out[22]:

```
Text(0.5, 1.0, 'MCMC chain for parameter $w$')
```

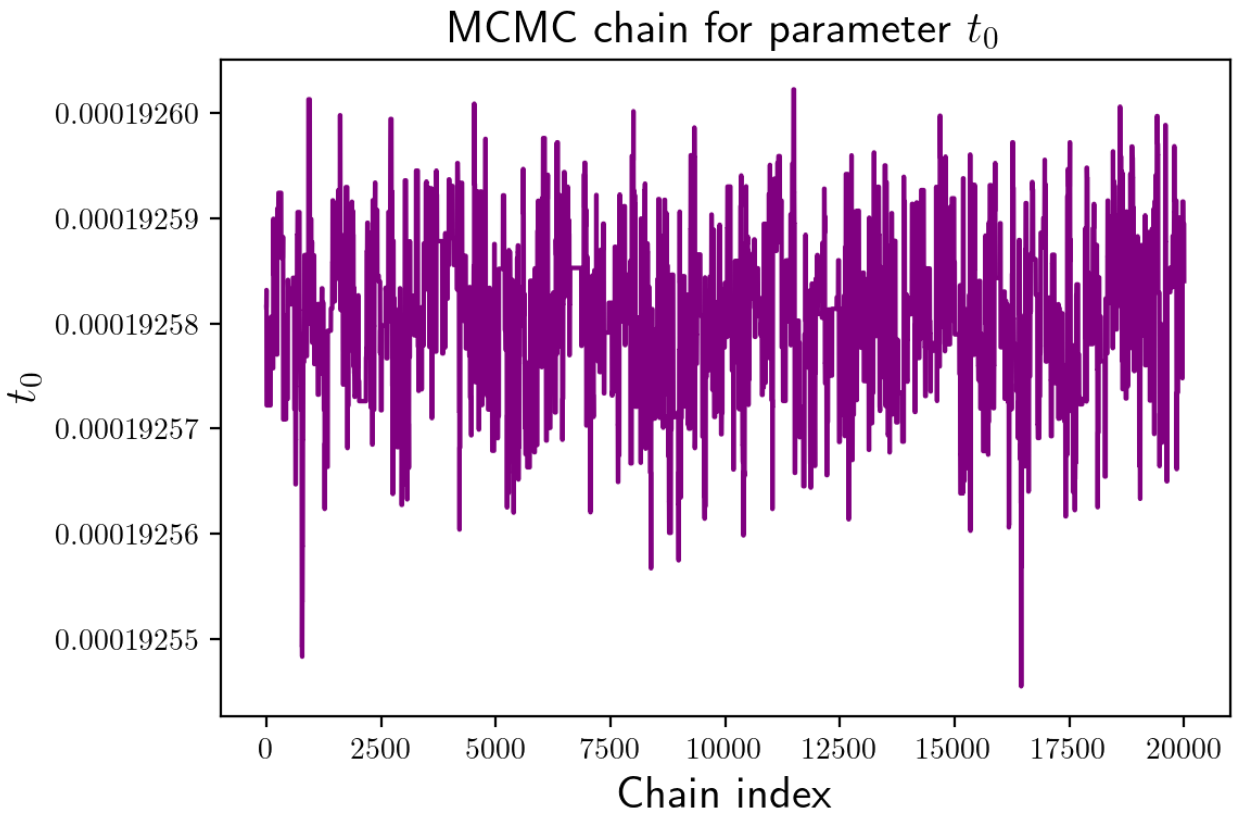


In [23]:

```
plt.plot(chain[:,4],color='purple')
plt.xlabel('Chain index',fontsize=15)
plt.ylabel('$t_0$',fontsize=15)
plt.title('MCMC chain for parameter $t_0$',fontsize=15)
```

Out[23]:

```
Text(0.5, 1.0, 'MCMC chain for parameter $t_0$')
```

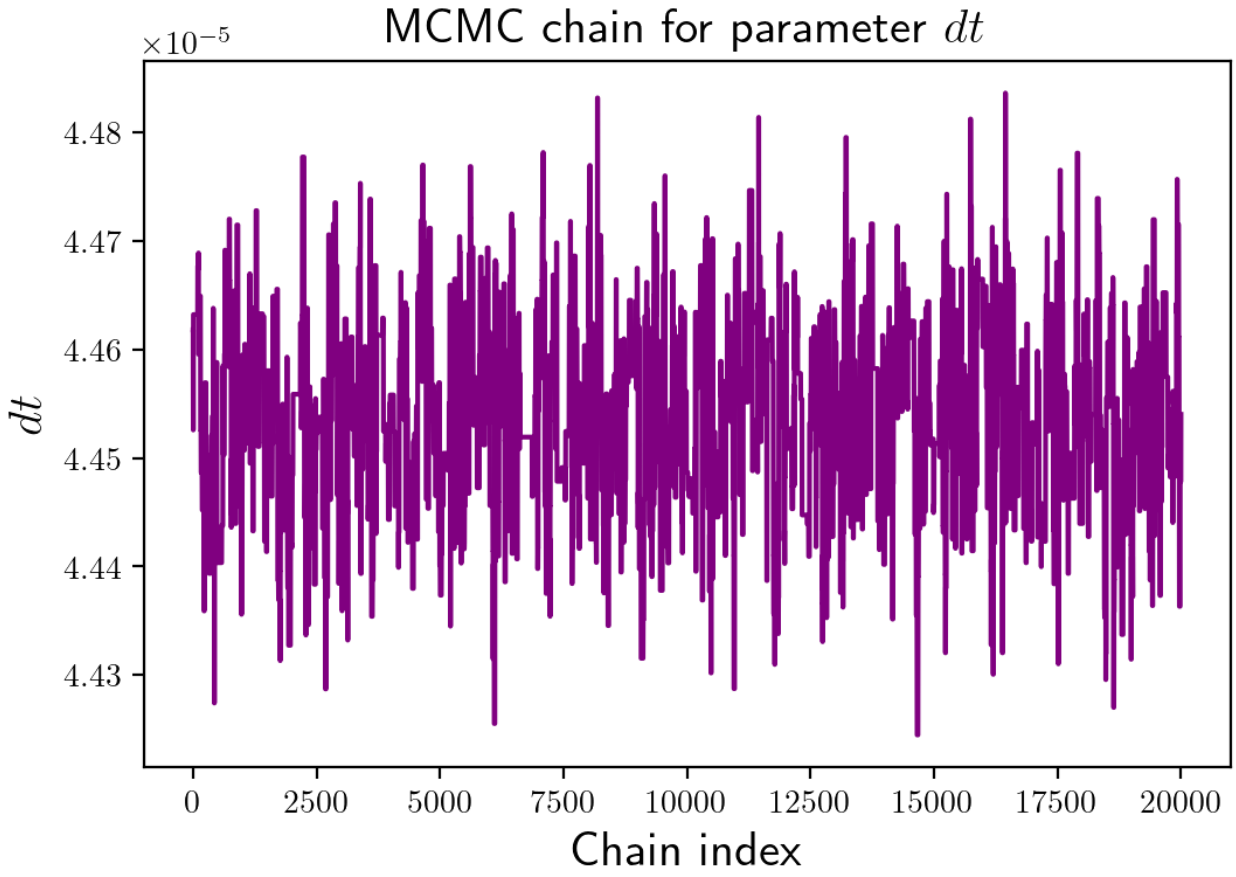


In [24]:

```
plt.plot(chain[:,5],color='purple')
plt.xlabel('Chain index',fontsize=15)
plt.ylabel('$dt$',fontsize=15)
plt.title('MCMC chain for parameter $dt$',fontsize=15)
```

Out[24]:

```
Text(0.5, 1.0, 'MCMC chain for parameter $dt$')
```



All the chains seems to have converged (no apparent structure on the scale of the entire chain). We now get new fit parameters and errors by finding the mean and standard deviation of the samples from the chain. We exclude the first 1000 samples to make sure all samples considered are at equilibrium.

In [25]:

```
# Compute means and standard deviations
a = np.mean(chain[:,0])
e_a = np.std(chain[:,0])
b = np.mean(chain[:,1])
e_b = np.std(chain[:,1])
c = np.mean(chain[:,2])
e_c = np.std(chain[:,2])
w = np.mean(chain[:,3])
e_w = np.std(chain[:,3])
t0 = np.mean(chain[:,4])
e_t0 = np.std(chain[:,4])
dt = np.mean(chain[:,5])
e_dt = np.std(chain[:,5])

# Print parameters and errors
print(a,e_a)
print(b,e_b)
print(c,e_c)
print(w,e_w)
print(t0,e_t0)
print(dt,e_dt)
```

```
1.4446020177563317 0.0006630618868887974
0.10470396154291853 0.0006287251446168117
```

```
0.0654419118731151 0.0005913198777459008
1.602442184774399e-05 1.3933302842027707e-08
0.0001925807879472429 7.599560984787617e-09
4.4538816733190186e-05 9.064898332239274e-08
```

So the parameters we get with the MCMC are

$$a = 1.44 \pm 6.66\text{x}10^{-4} \tag{22}$$
$$b = 1.04\text{x}10^{-1} \pm 6.29\text{x}10^{-4} \tag{23}$$
$$c = 6.45\text{x}10^{-2} \pm 5.91\text{x}10^{-4} \tag{24}$$
$$w = 1.61\text{x}10^{-5} \pm 1.39\text{x}10^{-8} \tag{25}$$
$$t_0 = 1.93\text{x}10^{-4} \pm 7.60\text{x}10^{-8} \tag{26}$$
$$dt = 4.46\text{x}10^{-5} \pm 9.06\text{x}10^{-8} \tag{27}$$

The uncertainties are slightly different but not by much. The fractional differences are on the order of 1/10.

h)

The width of the cavity is then given by

$$\text{width} = \left(\frac{dt}{w}\right) 9 \text{ GHz} \tag{28}$$

which I compute below

In [26]:
```
width = dt/w*9
e_width = width*(e_w/w)
print(width,e_width)
```

```
25.014902528614193 0.02175056397083404
```

So the width of the vacity resonance is

$$\text{width} = (25.01 \pm 0.02) \text{ GHz} \tag{29}$$

In [ ]:
```

```