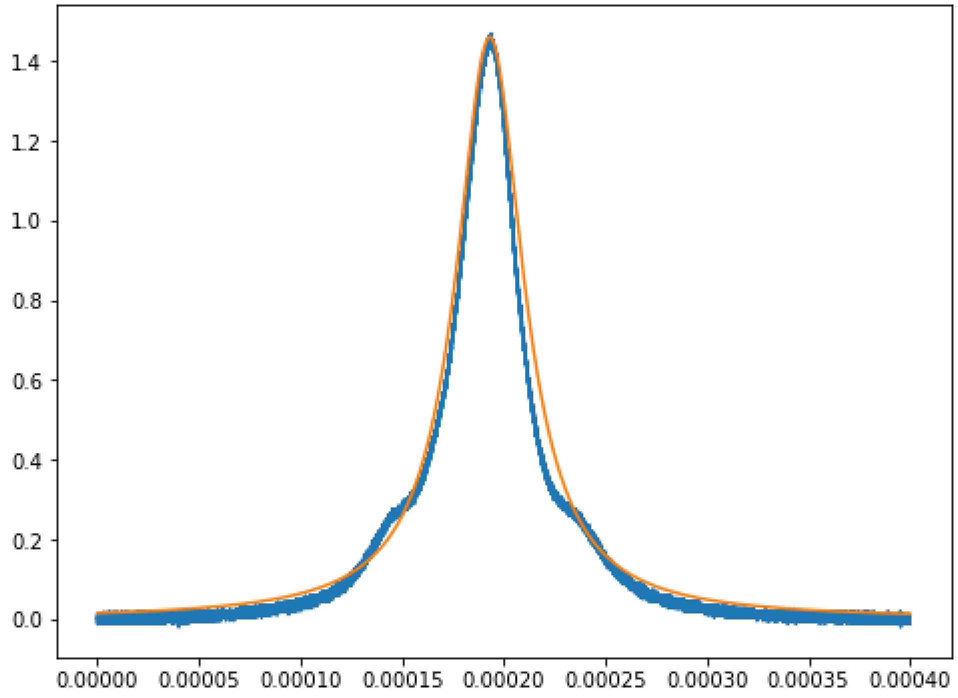
Problem Set 4

1) We will do a non-linear least-squares problem. Look at the file sidebands.npz in the mcmc directory. This file contains data from one of your TA's who is trying to measure the width of a resonance in an optical cavity. To do this, they send in a laser with sidebands (so, most of the laser power is at a central frequency ν plus a bit of power at $\nu + d\nu$ and $\nu - d\nu$). They use a piezo to mechanically push on the cavity, which in turn alters the resonant frequency of the cavity. You can assume that over the stretch of data we use, the piezo shifts the cavity resonant frequency linearly in time. You can load the data with *e.g.*:

```
stuff=np.load('sidebands.npz')
t=stuff['time']
d=stuff['signal']
```

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt

data = np.load("mcmc/sidebands.npz")
t = data['time']
d = data['signal']

#parameter guesses
a=1.46
t0=0.000193
w=0.00002

plt.figure(figsize=(8,6))
plt.plot(t,d)
plt.plot(t, a/(1+((t-t0)**2)/(w**2)))
plt.show()
```

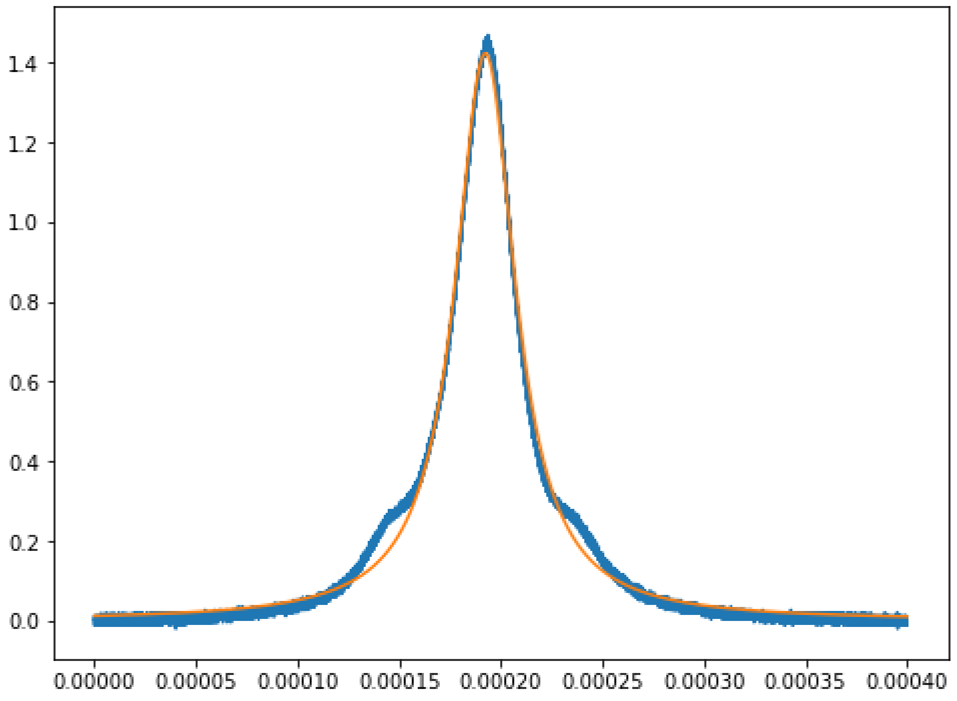


a) To start, model the data as a single Lorentzian and use analytic derivatives. Please use Newton's method (or Levenberg-Marquardt if you prefer) to carry out the fit. What are your best-fit parameters for the amplitude, width, and center? Please parameterize the Lorentzian as

$$d = \frac{a}{1 + (t - t_0)^2/w^2}$$

In [2]:
```
import numpy as np
import matplotlib.pyplot as plt

data = np.load("mcmc/sidebands.npz")
t = data['time']
d = data['signal']

def single_lorentz(params, t):
    a = params[0]
    t0 = params[1]
    w = params[2]
    d = a/(1+((t-t0)**2)/(w**2)) #lorentzian equation
    ddda = 1/(1+((t-t0)**2)/(w**2)) #partial derivative w.r.t each parameter
    dddt0 = (2*a*(t-t0))/(w**2*(((t-t0)**2)/(w**2)+1)**2)
    dddw = (2*a*(t-t0)**2)/(w**3*(((t-t0)**2)/(w**2)+1)**2)
    A = np.empty([len(t), len(params)]) #creating A and filling it with our
    A[:,0] = ddda #partial derivatives in each row
    A[:,1] = dddt0
    A[:,2] = dddw
    return d, A

#param guess
guess = [1.46, 0.000193, 0.00002]

#Newton's method
for iter in range(10):
    pred, A = single_lorentz(guess,t)
    resid = d-pred
    LHS = A.T@A
    RHS = A.T@resid
    dg = np.linalg.inv(LHS)@RHS
    guess += dg

print('The best fit parameters are:\na={}\nt0={}\nw={}'.format(round(guess[0],2),

Newt_fit, A = single_lorentz(guess, t)

plt.figure(figsize=(8,6))
plt.plot(t,d)
plt.plot(t,Newt_fit)
plt.show()
```

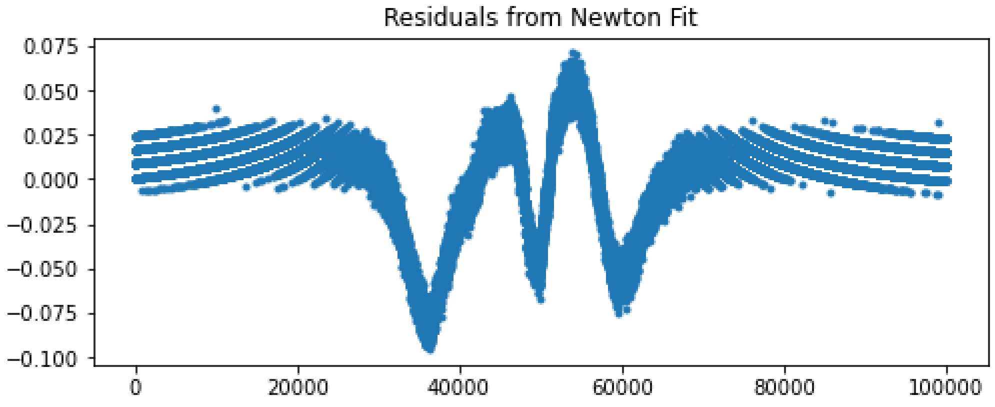
```
The best fit parameters are:
a=1.42
t0=0.000192
w=1.79e-05
```



b) Estimate the noise in the data, and use that to estimate the errors in your parameters.

In [6]:
```
import numpy as np
import matplotlib.pyplot as plt

#noise estimation
noise = np.mean(abs(Newt_fit-d))
print('We can estimate the noise in the data to be', round(noise,4))

#error in each parameter
errmat = np.sqrt(np.diag(np.linalg.inv(LHS)*noise**2)) #this addition of noise**2
err_a = errmat[0] #covariance matrix comes from the fact that we ignored N initic
err_t0 = errmat[1] #calculating the LHS, so we need to add it back in now that we
err_w = errmat[2] #an estimate for it

print('The error estimates for each parameter are:\na:{}\nt0:{}\nw:{}'.format(rou

plt.figure(figsize=(8,3))
plt.title('Residuals from Newton Fit')
plt.plot(Newt_fit-d, '.')
plt.show()
```

```
We can estimate the noise in the data to be 0.0194
The error estimates for each parameter are:
a:0.00033
t0:4.1e-09
w:5.8e-09
```



As we can see from the graph above, there is clearly structure in the residuals, meaning that the equation we chose for our fit was not good for the data we gathered. Because of this, our estimation for the noise in the data will not be accurate.

c) Repeat part a), but use *numerical* derivatives. I suggest you use a helper function that accepts an input function (and any ancillary data/arguments you want to pass it) and returns the derivatives of that function with respect to the model parameters. Are your answers statistically significantly different from your answers in a)?

In [ ]:
```
'''
import numpy as np
import matplotlib.pyplot as plt

def partial_derivs(f, t, params):
    #dx = np.diag(1e-16**(1/3)*params) #finding optimal stepsize for each paramet
    #print(dx[0])
    #print(params)
    #A = np.empty([len(t), len(params)])
    #A[:,0] = (f(t, params+dx[0])-f(t, params))/dx[0][0]
    #for i in range(len(params)):
    #    A[:,i] = (f(t, params+dx[i])-f(t, params))/dx[i][i]
    #return A

def lorentz(t, params):
    return params[0]/(1+((t-params[1])**2)/(params[2]**2))

def single_lorentz_num(params, t):
    a = params[0]
    t0 = params[1]
    w = params[2]
    d = a/(1+((t-t0)**2)/(w**2))
    A = partial_derivs(lorentz, t, params)
    return d, A

#param guess
guess = np.array([1.46, 0.000193, 0.00002])

for iter in range(10):
    pred, A = single_lorentz_num(guess,t)
    resid = d-pred
    LHS = A.T@A
    RHS = A.T@resid
    dg = np.linalg.inv(LHS)@RHS
    guess += dg

print('The best fit parameters are:\na={}\nt0={}\nw={}'.format(round(guess[0],2),

Newt_fit_num, A = single_lorentz_num(guess, t)

plt.figure(figsize=(8,6))
plt.plot(t,d)
plt.plot(t,Newt_fit_num)
plt.show()
'''
```

I couldn't figure out how to find the partial derivatives numerically, so for d) I just used the analytic derivatives.

d) Repeat part c), but now model the data as the sum of three Lorentzians. The width of all three Lorentzians should be the same, and the separation of the side peaks from the main peak should be equal, *i.e.*:

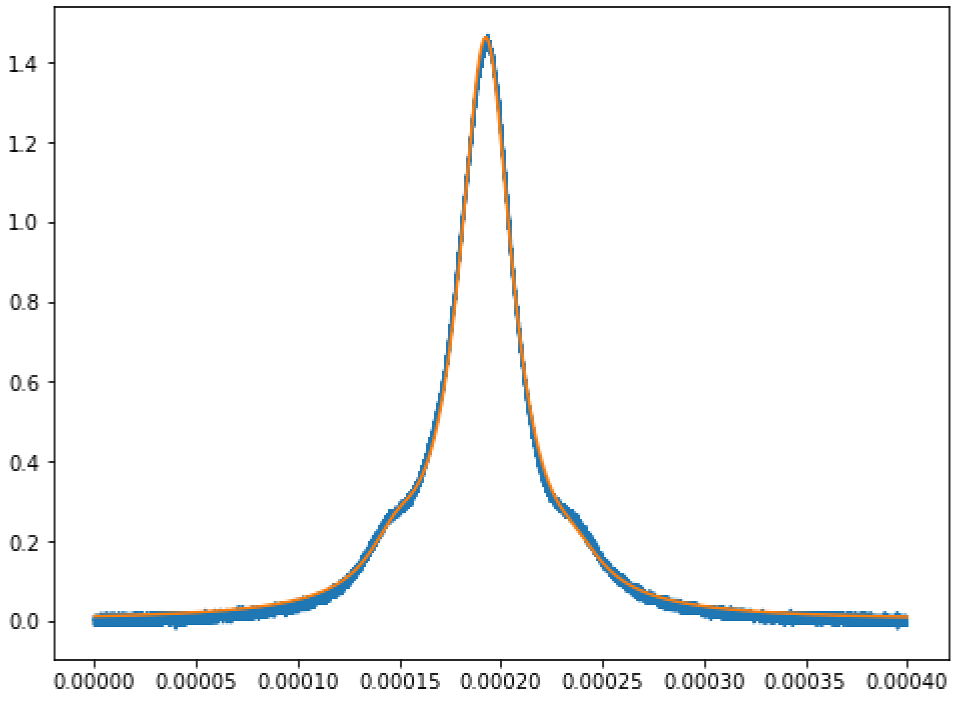
$$d = \frac{a}{1 + (t - t_0)^2/w^2} + \frac{b}{1 + (t - t_0 + dt)^2/w^2} + \frac{c}{1 + (t - t_0 - dt)^2/w^2}$$

You should have sensible guesses for a, t_0, w from earlier, so you should try to estimate reasonable initial guesses for b, c, dt. What are your errors on these parameters?

In [78]:
```
import numpy as np
import matplotlib.pyplot as plt

data = np.load("mcmc/sidebands.npz")
t = data['time']
d = data['signal']

def trip_lorentz(params, t):
    a = params[0]
    b = params[1]
    c = params[2]
    t0 = params[3]
    w = params[4]
    dt = params[5]

    y = a/(1+((t-t0)**2)/(w**2))+b/(1+((t-t0+dt)**2)/(w**2))+c/(1+((t-t0-dt)**2)/
    dyda = 1/(1+((t-t0)**2)/(w**2)) #partial derivatives for each parameter
    dydb = 1/(1+((t-t0+dt)**2)/(w**2))
    dydc = 1/(1+((t-t0-dt)**2)/(w**2))
    dydt0 = 2*a*(t-t0)/(w**2*((t-t0)**2/w**2+1)**2)+2*b*(t-t0+dt)/(w**2*((t-t0+dt
    dydw = 2*a*(t-t0)**2/(w**3*((t-t0)**2/w**2+1)**2)+2*b*(t-t0+dt)**2/(w**3*((t-
    dyddt = 2*c*(t-t0-dt)/(w**2*((t-t0-dt)**2/w**2+1)**2)-2*b*(t-t0+dt)/(w**2*((t

    A = np.empty([len(t), len(params)]) #making A like last time
    A[:,0] = dyda
    A[:,1] = dydb
    A[:,2] = dydc
    A[:,3] = dydt0
    A[:,4] = dydw
    A[:,5] = dyddt
    return y, A

#param guess
guess = [1.46, 0.2, 0.2, 0.000193, 0.000015, 0.00004]

#Newton's Method
for iter in range(10):
    pred, A = trip_lorentz(guess,t)
    resid = d-pred
    LHS = A.T@A
    RHS = A.T@resid
    dg = np.linalg.inv(LHS)@RHS
    guess += dg

#error in each parameter
errmat = np.sqrt(np.diag(np.linalg.inv(LHS)*noise**2))
err_a = errmat[0]
err_b = errmat[1]
err_c = errmat[2]
err_t0 = errmat[3]
err_w = errmat[4]
err_dt = errmat[5]

print('The best fit parameters are:\na={}±{}\nb={}±{}\nc={}±{}\nt0={}±{}\nw={}±{}

fit, A = trip_lorentz(guess, t)
```

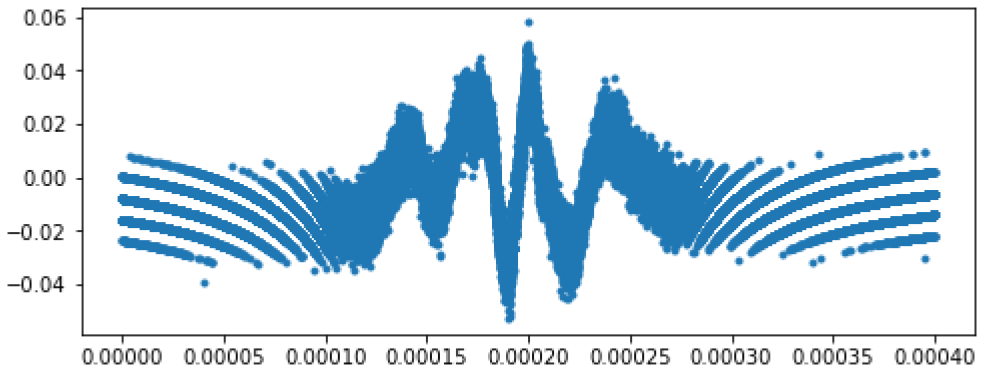
```
plt.figure(figsize=(8,6))
plt.plot(t,d)
plt.plot(t,fit)
plt.show()
```

```
The best fit parameters are:
a=1.44±0.00035
b=0.104±0.00034
c=0.065±0.00033
t0=0.000193±4.2e-09
w=1.61e-05±7.5e-09
dt=4.46e-05±5e-08
```



e) Look at the residuals from subtracting your best-fit model from the data. Do you believe the error bars you got by assuming the data are independent with uniform variance, and that the model is a complete description of the data?

In [66]:
```
plt.figure(figsize=(8,3))
plt.plot(t,d-fit, '.')
plt.show()
```

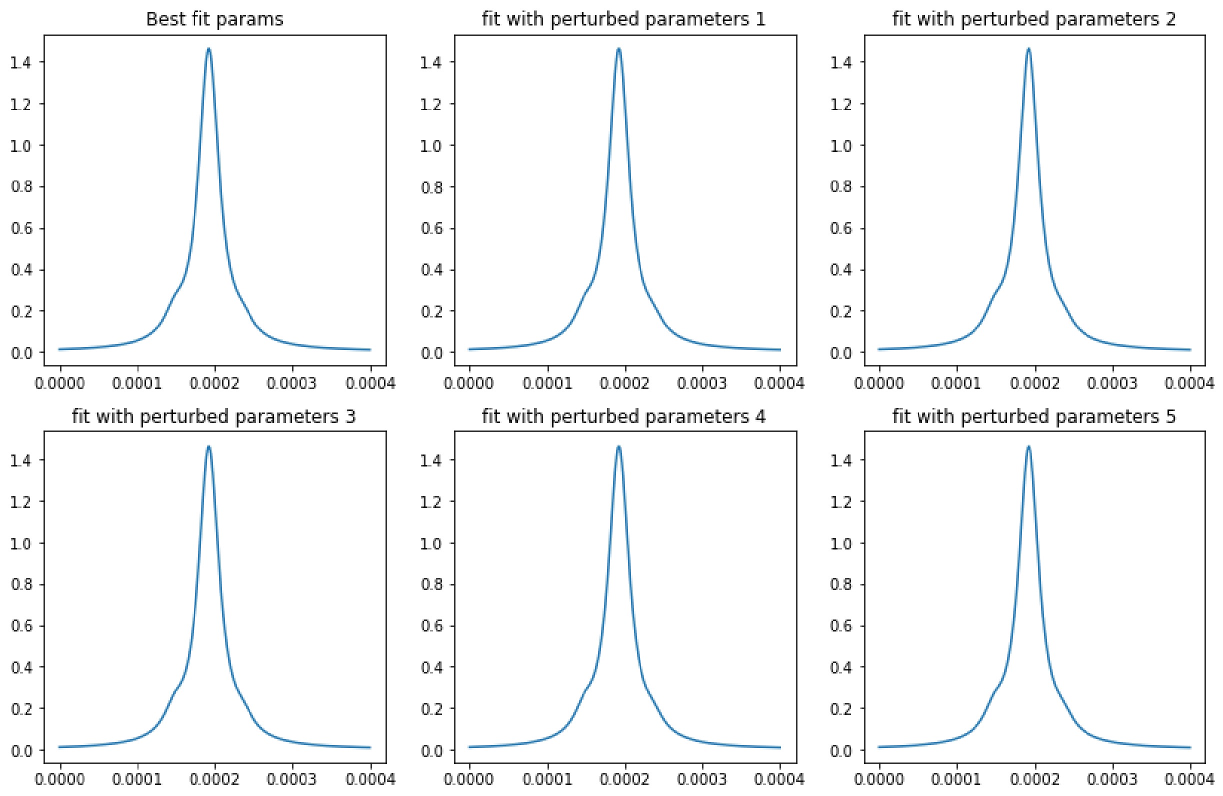


I do not believe the error bars that I got. As explained before there is a clear trend in the residuals, indicating that the equation we used to fit our data is not representative of the data. This means that the model is not a complete description of the data, therefore the error bars for the parameters will not be accurate.

f) Generate some some realizations for the parameter errors using the full covariance matrix $A^TN^{-1}A$ from part d). Plot the models you get from adding these parameters to the parameter errors. What is the typical difference in χ^2 for the perturbed parameters compared to the best-fit χ^2? Is this reasonable?

In [53]:
```
#random variations in the parameters using the covariance matrix
new_params = np.random.multivariate_normal(guess, np.linalg.inv(LHS)*noise**2, 5)

#calculating chisquares to compare later
fit_chsq = np.sum((d-fit)**2)/noise**2
new_chsqs = np.empty(len(new_params))

plt.figure(figsize=(14,9))
plt.subplot(2,3,1)
plt.title('Best fit params')
plt.plot(t, fit)

for i in range(5):
    fit, A = trip_lorentz(new_params[i], t) #plotting our new fits
    new_chsqs[i] = np.sum((d-fit)**2)/noise**2 #calculating chisquare of our new
    plt.subplot(2,3,i+2)
    plt.title('fit with perturbed parameters {}'.format(i+1))
    plt.plot(t, fit)

diff = np.mean(abs(fit_chsq-new_chsqs))
print('The typical difference between the chisquare for the perturbed parameters
print('Below are fits with our new perturbed parameters and we can see they are v
```

```
The typical difference between the chisquare for the perturbed parameters and t
he best fit parameters is  1.2580077093050932
Below are fits with our new perturbed parameters and we can see they are very s
imilar to our fit with our best fit parameters
```

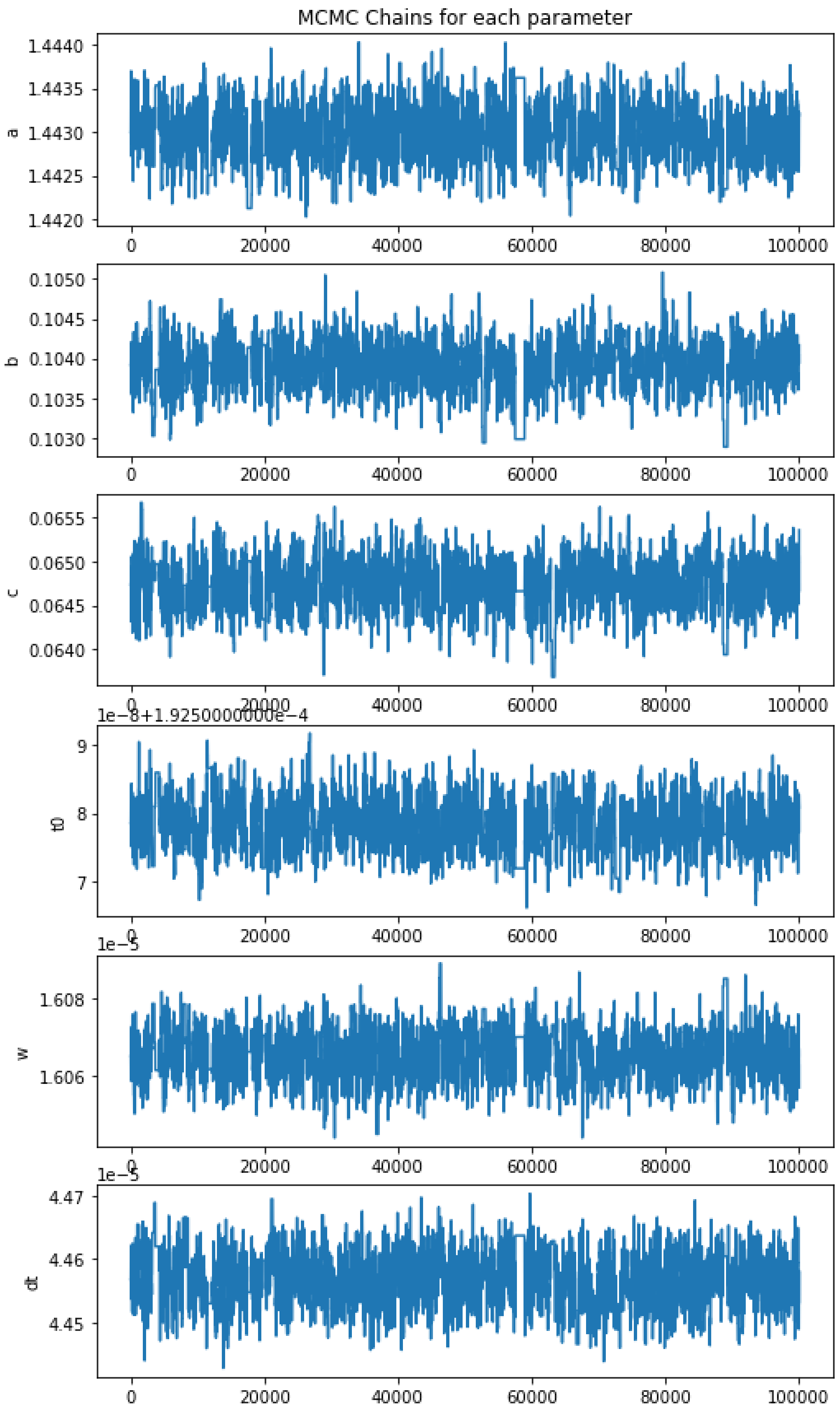


The typical difference in χ^2 for the perturbed parameters and the best fit parameters is reasonable, as we are generating our new parameters based off the covariance matrix we calculated previously, meaning we expect these new parameters to still be decently good fits to the data despite being slightly perturbed. Since we still expect the new parameters to be decent, we wouldn't expect a big difference in χ^2 compared to our best fit parameter, which is what we observe.

g) Redo the fit from part d), using an MCMC. IMPORTANT - please use your parameter covariance estimate from d) to generate your trial MCMC samples, though you can introduce an overall scaling (the overall scaling should be order unity, but you may find faster convergence if it isn't exactly 1). Show at least one plot that explains why you think your chain is converged. Did your error bars change?

In [61]:
```
#mcmc
nstep = 100000 #number of steps
covmat = np.linalg.inv(LHS)*noise**2 #just defining the covariance matrix
                                    #so I dont have to write it out again
def chsq_f(params , t, d, noise): #function to calculate chisquare
    fit, A = trip_lorentz(params, t) #calculates the fit with the new params
    chsq = np.sum((d-fit)**2)/noise**2 #and then the chisquare for the new fit
    return chsq

def MCMC(params, covmat, noise):
    chain=np.zeros([nstep,len(params)]) #initializing the chain
    chain[0,:]=params #start MCMC at our best fit params from newton's method
    chsq = chsq_f(chain[0,:], t, d, noise) #calculating our inital chisquare
    for i in range(1,nstep):
        new_params = chain[i-1,:]+np.random.multivariate_normal(np.zeros(len(para
        #take step in direction according to covariance matrix from initial posit
        new_chsq = chsq_f(new_params, t, d, noise) #calculate chisquare with new
        accept = np.exp(-0.5*(new_chsq-chsq)) #calculating acceptance
        if accept>np.random.rand(1): #choosing whether to accept or decline the s
            chain[i,:] = new_params
        else:
            chain[i,:] = chain[i-1,:]
    return chain

chain = MCMC(guess, covmat, noise)
```

In [70]:
```
param_names = ['a', 'b', 'c', 't0', 'w', 'dt']

#plotting chains
plt.figure(figsize=(8,15))
plt.subplot(6,1,1)
plt.title('MCMC Chains for each parameter')
for i in range(len(guess)):
    plt.subplot(6,1,i+1)
    plt.plot(chain[:,i])
    plt.ylabel(param_names[i])
```



As we can see from the chains above for each parameter, none of them have a clear structure and they all vary about a clear mean, so we can assume that our chain is properly mixed.

In [69]:
```
#comparing the error bars
mcmc_params = np.empty(len(guess))
mcmc_errs = np.empty(len(guess))
for i in range(len(guess)):
    mcmc_params[i] = np.mean(chain[:,i])
    mcmc_errs[i] = np.std(chain[:,i])

print('Best Fit parameters and errors from Newton:\na={}±{}\nb={}±{}\nc={}±{}\nt
print('\nBest Fit parameters and errors from MCMC:\na={}±{}\nb={}±{}\nc={}±{}\nt
```

```
Best Fit parameters and errors from Newton:
a=1.44±0.00035
b=0.104±0.00034
c=0.065±0.00033
t0=0.000193±4.2e-09
w=1.61e-05±7.5e-09
dt=4.46e-05±5e-08

Best Fit parameters and errors from MCMC:
a=1.44±0.00034
b=0.104±0.00035
c=0.065±0.00032
t0=0.000193±4.1e-09
w=1.61e-05±7.1e-09
dt=4.46e-05±4.5e-08
```

Even after performing the MCMC, the error bars have barely changed, which means that our original best fit parameters from Newton's method were quite accurate (or at least as accurate as you could get by fitting our data with an equation that doesn't properly represent it).

h) The laser sidebands are separated from the main peak by 9 GHz (so dx maps to 9 GHz). What is the actual width of the cavity resonance, in GHz?

In [76]:
```
#dt maps to 9GHz, but dt is in time, so 1/dt --> 9Ghz

scale = 9e9/mcmc_params[5] #scaling factor from the mapping
print('The actual width of the cavity resonance in GHz is {}GHz'.format(round(mc
```

```
The actual width of the cavity resonance in GHz is 3.24GHz
```