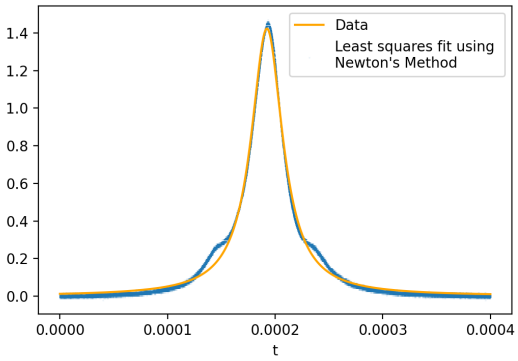
1 Question 1

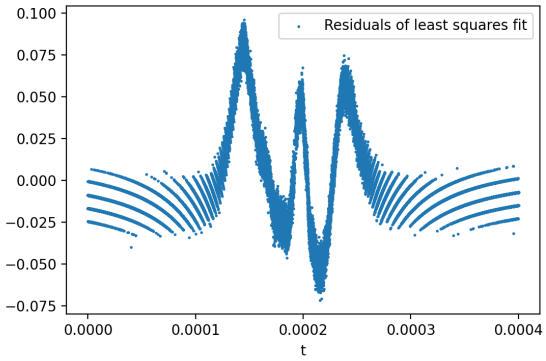
a) & b)

To fit the single Lorentzian, the same code from class was adapted for the new parameters. The analytic derivatives calculated for the parameter gradient of the Lorentzian fit function had to be changed for the new parameterization. The two functions below were used to fit the data :

```
def calc_lorentz(m,t):  # Same function from class, different derivatives
    a,t_0,w=m
    y=a/(1+(t-t_0)**2/w**2)
    grad=np.zeros([len(t),len(m)])
    grad[:,0]=1/(1+(t-t_0)**2/w**2)
    grad[:,1]=a/(1+(t-t_0)**2/w**2)**2*2*(t-t_0)/w**2      # Analytic derivatives
    grad[:,2]=a/(1+(t-t_0)**2/w**2)**2*2*(t-t_0)**2/w**3
    return y,grad

def newton_lorentz(t,y,m0,n):  # Same code for Newtons method from class
    m=m0.copy()
    for i in range(n):
        pred,grad=calc_lorentz(m,t)
        r=y-pred
        r=np.matrix(r).T
        grad=np.matrix(grad)
        lhs=grad.T@grad
        rhs=grad.T@r
        dm=np.linalg.inv(lhs)*rhs
        for j in range(len(m)):
            m[j]=m[j]+float(dm[j])  # Keep m as list type to pass through calc_lorentz
    return m
```

Using an initial guess of `m_0= [1,0.0002,0.00002]` with 20 steps, the following fit was produced:





To estimate the noise in the data, the beginning of the data (first 1000 data points) without any signal where it is approximately flat was used. The standard deviation of this part of the data was taken as the noise estimate. To estimate the error from this noise estimation, the following equation was used:

$$\langle \delta_m \delta_m^T \rangle = (\mathbf{A}^T \mathbf{N}^{-1} \mathbf{A})^{-1}$$

Where $\langle \delta_m \delta_m^T \rangle$ is the covariance matrix with each parameter variance on the diagonals. Here is the code used to calculate this matrix and the parameter errors:

```
noise_est=np.std(y[:1000])  # Noise estimate from residuals
N_inv=noise_est**(-2)*np.identity(len(m))  # Match shape of A.T@A, can put N to front of
cov=np.linalg.inv(N_inv@A.T@A)             # equation since N=const*identity (avoid memory
m_err=np.sqrt(np.diag(cov))                # error of large matrix)
```

Where the matrix `A` is the gradient calculated from `param_grad`.

The following are the parameter values and their errors:

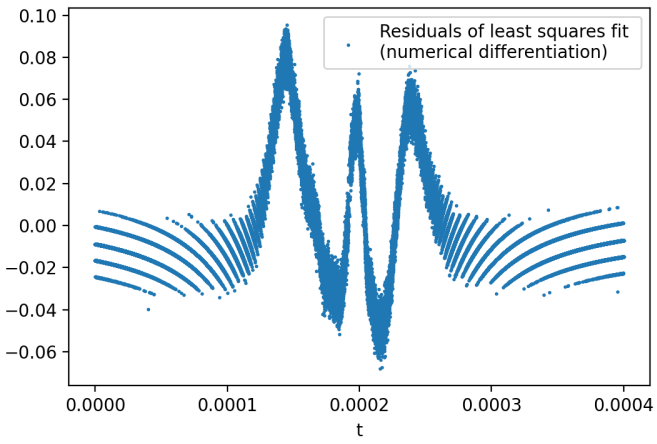
$$\begin{gathered} \text{a} = 1.4228106806322425 \text{ +/- } 8.997120280772151\text{e-}05 \\ t_0 = 0.0001923586493756531 \text{ +/- } 1.1330682440632378\text{e-}09 \\ \text{w} = 1.7923690794012603\text{e-}05 \text{ +/- } 1.6045684100177063\text{e-}09 \\ \chi^2 = 2236399.0562241757 \end{gathered}$$

c)

To calculate parameter gradient of the Lorentzian numerically, a function called `param_grad(fun,m,t)` was written which took the numerical derivative with respect to each parameter in `m`, for each time t. The function `calc_lorentz` was also modified to used this new function. Here is its code:

```
def param_grad(fun,m,t):  # Get parameter gradient of function numerically
    dx=10**-5
    derivs=[]
    for i in range(len(m)):
        m_c=m.copy()
        m_c[i]+=dx    # m_i -> m_i + dm_i
        y1=fun(m_c,t)
        m_c[i]-=2*dx  # m_i -> m_i - dm_i
        y_1=fun(m_c,t)
        d=(y1-y_1)/(2*dx)  # Central difference
        derivs.append(d)
    return np.matrix(derivs).T  # Transpose fit with Newton method function

def calc_lorentz(m,t):
    y=lorentz(m,t)
    grad=param_grad(lorentz,m,t)
    return y,grad
```

The new `calc_lorentz` function was then used by the same Newton method function written in a) to get the optimal parameters. Their associated errors were calculated the same as in b). Here is the plot of the residuals and the fit parameters with their errors.

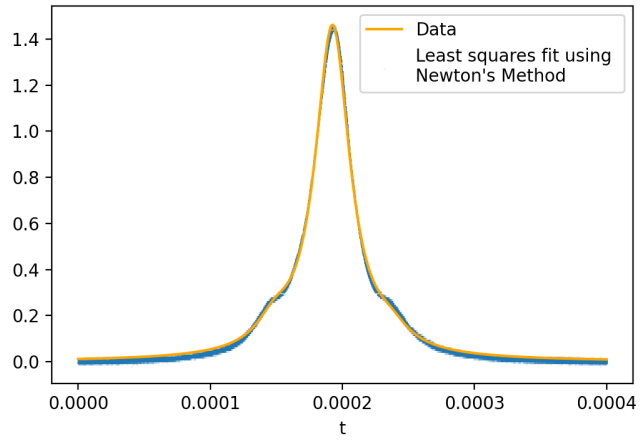


$$\begin{gathered} \text{a} = 1.4234152245072382 \text{ +/- } 9.655182693286411\text{e-}05 \\ t_0 = 0.00019225158096665602 \text{ +/- } 1.2621360962099775\text{e-}09 \\ \text{w} = 1.7908121960468934\text{e-}05 \text{ +/- } 1.719685914281451\text{e-}09 \\ \chi^2 = 2370215.8683269 \end{gathered}$$

Comparing residual plots and parameter values/errors, the results are not statistically significant from parts a) & b).

d)

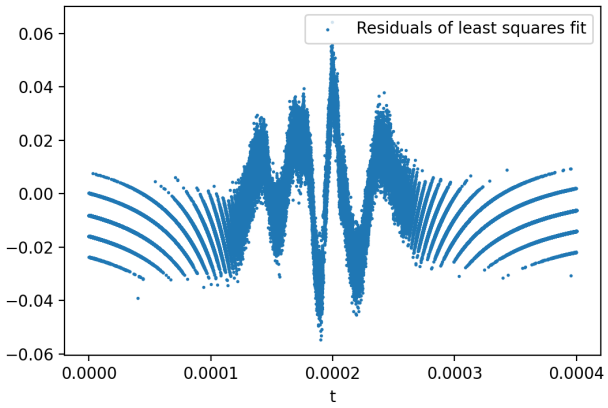
The function `param_grad` written in the previous section can be used for any function with any number of parameters, therefore was easily used for this new model function. The same Newton method function was used as in the previous sections, with the same method of finding parameter errors. The following best fit and residual plots were produced:



The parameters and their errors are:

$$\begin{aligned}
\text{a} &= 1.4412685748812604 \text{ +/- } 0.00011234235603351328 \\
t_0 &= 0.0001924775450222628 \text{ +/- } 1.4337598857982661\text{e-}09 \\
\text{w} &= 1.6091160302899222\text{e-}05 \text{ +/- } 2.3259838752449917\text{e-}09 \\
\text{b} &= 0.10185476176384073 \text{ +/- } 9.542638649378547\text{e-}05 \\
\text{c} &= 0.06567239227957164 \text{ +/- } 9.257742991117536\text{e-}05 \\
\text{dt} &= 4.406368858236271\text{e-}05 \text{ +/- } 1.694269516632876\text{e-}08 \\
\chi^2 &= 756420.5204737067
\end{aligned}$$

e)



This plot is the residuals from d). As can be seen on ends where there is little structure, there is little correlation in the noise as there is much variation from point to point. Therefore the choice of the $\mathbf{N}$ matrix

as as the identity matrix multiplied by the noise estimate squared is justified. However the model does not seem to be a complete description of the data as there is still noticeable structure in the residuals.

f)

To generate realizations of the parameter errors, the following equation was used:

$$m - m_t = (\mathbf{A}^T\mathbf{N}^{-1}\mathbf{A})^{-1}\mathbf{A}^T\mathbf{N}^{-1}n$$

Where n is a random noise vector with entries for each number of data points. Its values were generated from a normal distribution with standard deviation equal to that of the noise estimate. This random error $m - m_t$ was calculated 25 times then added to the model parameters calculated in d). Below is the code for generating the realizations:

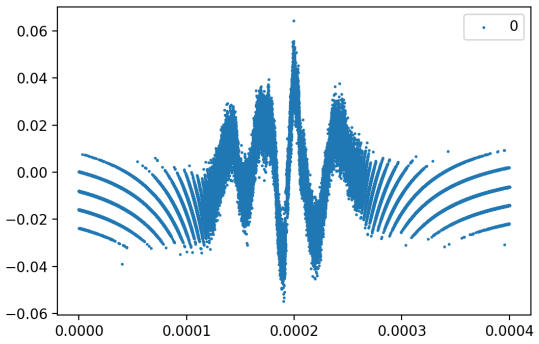
```
num=25  # Generate 25 model realizations
list_chi_sq=[]
for i in range(num):
    n_rand=np.random.normal(loc=0.0,scale=noise_est,size=len(t))  # Random noise
    n_rand=np.matrix(n_rand).T
    m_err_rlz=cov@N_inv@A.T@n_rand  # Can pull N_inv out in front
    m_rlz=np.empty(len(m))
    for j in range(len(m)):
        m_rlz[j]=m[j]+float(m_err_rlz[j])  # realized random m
    list_chi_sq.append(chi_sq(y,m_rlz,t,noise_est))
    pred_rlz=calc_lorentz(m_rlz,t)[0]
```

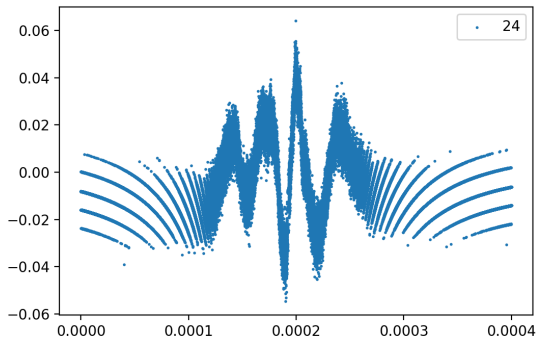
The χ^2 was calculated for each of the 25 times, then the mean and standard deviation of the samples were taken. Here are the results compared to χ^2 of the actual fit:

Mean and standard deviation of χ^2 for 25 model realizations:
756382.9087659069 +/- 266.96897717621073

χ^2 for Newton method fit:
756420.5204737067

Therefore the error of 266 in the χ^2 is reasonable relative to its large value, therefore this is indicative of a good fit, since there is very little variation in the χ^2 given a small perturbation of the parameters from random noise. Below are two graphs of the residuals of the perturbed parameters, one from the first realization in the loop and one from the last:





g)

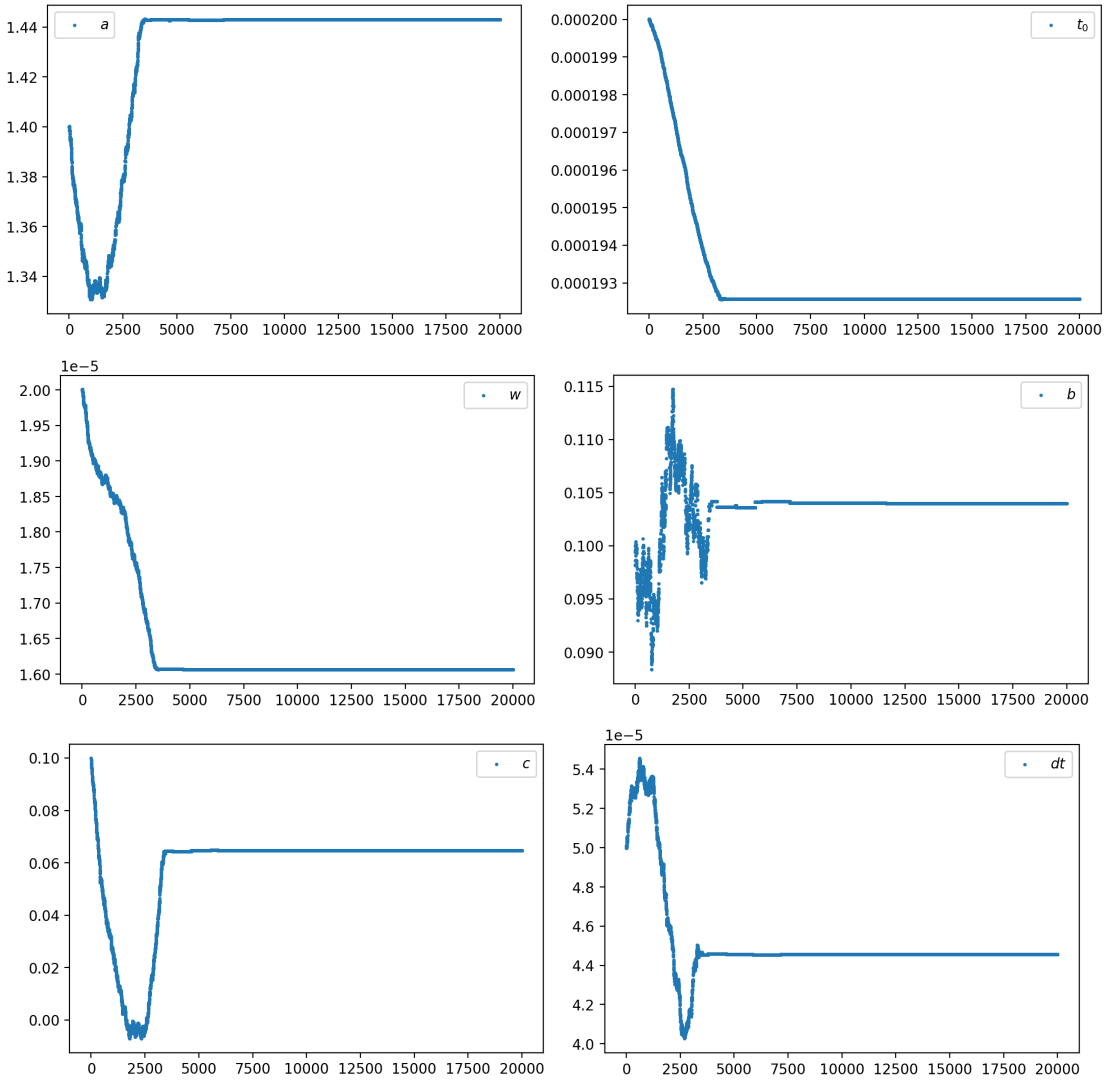
The same MCMC functions from class were used for this fit, along with the parameter errors from d) used for the scale. Below is the code used for the fit:

```
def lorentz_chisq(m,data,noise):
    t=data['time']
    y=data['signal']
    errs=[noise]*len(t)

    pred=calc_lorentz(m,t)[0]
    chisq=np.sum(np.power(pred-y,2)/np.power(errs,2))
    return chisq

def MCMC_chain(fun_chisq,data,start_params,noise_est,scale,nstep=10000,T=1):
    nparam=len(start_params)
    chain=np.zeros([nstep,nparam])
    chisq=np.zeros(nstep)
    chain[0,:]=start_params
    cur_chisq=fun_chisq(start_params,data,noise_est)
    chisq[0]=cur_chisq
    params=start_params
    for i in tqdm(range(1,nstep)):
        trial_params=params+np.random.randn(len(params))*scale
        new_chisq=fun_chisq(trial_params,data,noise_est)
        accept_prob=np.exp(-0.5*(new_chisq-cur_chisq)/T)
        if np.random.rand(1)<accept_prob:
            params=trial_params
            cur_chisq=new_chisq
        chain[i,:]=params
        chisq[i]=cur_chisq
    return chain,chisq

def chain_eval(chain,chisq,T=1):
    dchi=chisq-np.min(chisq)
    wt=np.exp(-0.5*dchi*(1-1/T))
    npar=chain.shape[1]
    tot=np.zeros(npar)
    totsqr=np.zeros(npar)
    for i in range(npar):
        tot[i]=np.sum(wt*chain[:,i])
        totsqr[i]=np.sum(wt*chain[:,i]**2)
    mean=tot/np.sum(wt)
    meansqr=totsqr/np.sum(wt)
    var=meansqr-mean**2
    return mean,np.sqrt(var),wt

data=np.load('sidebands.npz')
t=data['time']
y=data['signal']
m0=[1.4,0.0002,0.00002,0.1,0.1,0.00005]
noise_est=np.std(y[:10000])
scale=5*m_err

nstep=20001
chain,chisq=MCMC_chain(lorentz_chisq,data,m0,noise_est,scale,nstep=nstep)
```

After plotting, the chain was evaluated excluding the portion that is non converged. In this case, this corresponded to starting from the 4000^{th} iteration:

```
m,m_err=chain_eval(chain[1-000:],chisq[10000:])[:2] # Start from where converged
```

On the next page are the graphs showing the convergence of the chain for each variable, and then the residuals from the evaluation of the chain evaluation and the data:

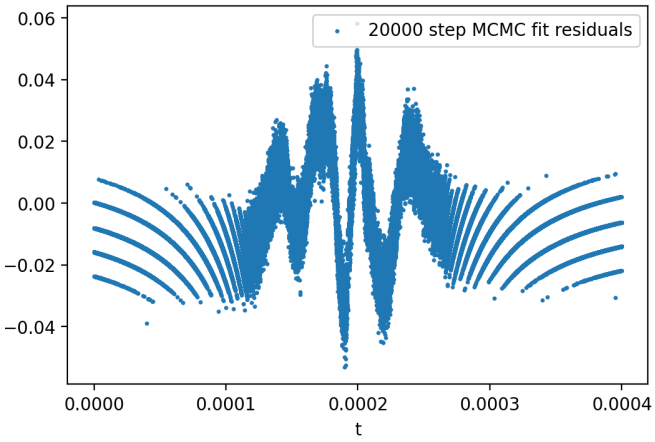


The parameters of the MCMC fit and their errors are:

$$\begin{gathered}
\text{a} = 1.4430841373340746 \text{ +/- } 6.997011565712201\text{e-}06 \\
t_0 = 0.00019257670892308614 \text{ +/- } 7.791816860234336\text{e-}10 \\
\text{w} = 1.6063420698585657\text{e-}05 \text{ +/- } 2.2678173335937904\text{e-}11 \\
\text{b} = 0.10398277551991834 \text{ +/- } 1.58404144044039\text{e-}05 \\
\text{c} = 0.06473784061792497 \text{ +/- } 4.002652740564577\text{e-}05 \\
\text{dt} = 4.457032917384558\text{e-}05 \text{ +/- } 5.943201673490474\text{e-}10 \\
\chi^2 = 746281.3321022362
\end{gathered}$$

In this case the error bars are almost identical to Newton's method.

Here is the plot for the residuals of the MCMC chain evaluation:



h)

If dx maps to 9GHZ, then we can calculate w in GHZ by:

$$w_{GHZ} = w\frac{9\,GHZ}{dx}$$

Using this, the width was calculate to be 3.244 GHZ.