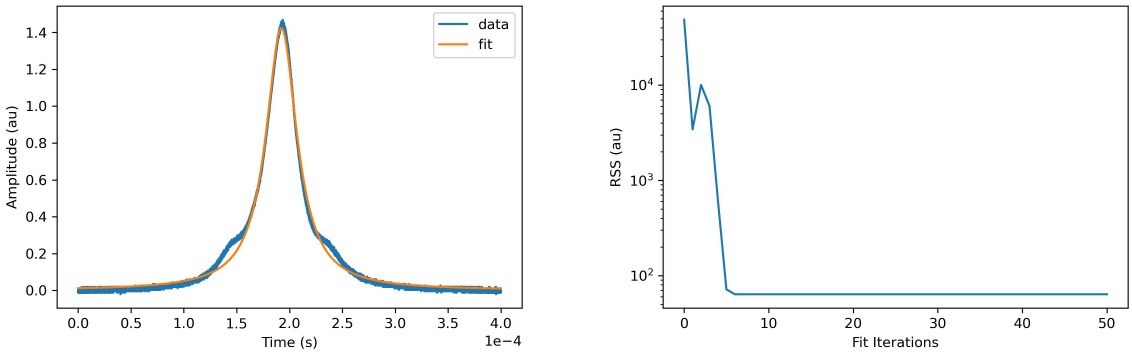
PHYS-512: Problem Set 4

Simon Briesenick

October 15, 2022

(a) Newton's method is implemented similarly to the in-class example. The resulting fit for `50` iterations is shown in Fig. 1a. To implement Newton's method, one needs the gradient of the fit function



(a) Fit

(b) Residual Squared Sum (RSS)

Figure 1: Single Lorentzian Fit to the signal and associated error in the number of fitting iterations.

w.r.t. the model parameters `m`:

```
def lorentzian(t, m):
 a, t_0, w = m
 d = a/(1 + ((t - t_0)**2)/(w**2))
 # make empty array to store gradient values
 grad = np.zeros([t.size, 3])
 # derivative w.r.t. a
 grad[:,0] = (w**2)/((w**2) + (t - t_0)**2)
 # deriv t_0
 grad[:,1] = 2*a*w**2*(t - t_0)/(((w**2) + (t - t_0)**2)**2)
 # deriv w
 grad[:,2] = 2*a*w*(t - t_0)**2/(((w**2) + (t - t_0)**2)**2)
 return d, grad
```

A `for` loop then solves the algebraic system

$$A_m^T N^{-1} A_m \delta m = A_m^T N^{-1} r \tag{1}$$

for δm - which serves as a correction to each guess, i.e.

$$m \rightarrow m + \delta m. \tag{2}$$

The implementation is very similar to how it was done in class:

```
model_params[0] = (a, t_0, w)
error = []
for i in range(iterator):
    pred, grad = lorentzian(time, model_params[i])
    # calculate residual (serves as noise covariance matrix)
    r = d - pred
    err = np.mean(np.abs(r))
    error.append(err**2)
    # cast matrices in readily available forms
    r = np.matrix(r).transpose()
    grad = np.matrix(grad)
    # compute linear system components
    lhs = grad.transpose()*grad
    rhs = grad.transpose()*r
    # solve it for dm
    dm = np.linalg.inv(lhs)@rhs
    # output is matrix, reshape for next step
    dm = dm.reshape(3,)
    if i == iterator - 1:
        break
    else:
        model_params[i + 1] = model_params[i] + dm
```

As initial guess for the model parameters, values could easily be read of from a preliminary plot of the signal as (units dropped)

$$t_0 = 1.8 \times 10^{-4}, a = 1.5, w = 1 \times 10^{-4}.$$

The final values for the parameters are

$$t_0 = 1.92 \times 10^{-4}, a = 1.42, w = 1.79 \times 10^{-5}.$$

(b) To estimate the uncertainties on the calculated parameters, one needs to compute

$$\left(A^T N^{-1} A\right)^{-1} \tag{3}$$

and take the square root of the diagonal entries. The Code is very similar to how I have done it in the previous P-set:

```
par_errs = np.sqrt(np.diag(np.linalg.inv(lhs)*err**2))
```

For the parameter errors, I get

$$\delta t_0 = 4.1 \times 10^{-9}, \delta a = 3.3 \times 10^{-4}, \delta w = 5.8 \times 10^{-9}.$$

(c) This exercise follows from (d), upon forcing the model parameters for the two additional Lorentzian functions to vanish (Originally, I just took a `single_lorentzian(t, m)` for modeling the data. The Code is only marginally different and is not repeated here). The parameters are the approximately as before:

$$t_0 = 1.92 \times 10^{-4}, a = 1.42, w = 1.80 \times 10^{-5}$$

to two significant figures.

The absolute difference in a parameter p, $\Delta p = |p_{\text{analytic}} - p_{\text{numeric}}|$ with the two methods is calculated as

$$\Delta t_0 = 1.6 \times 10^{-8}, \Delta a = 3.0 \times 10^{-3}, \Delta w = 2.9 \times 10^{-8}.$$

However, it should be noted that the discrepancies in those differences depend heavily on the chosen step size. The above results are obtained with `step = 1e-6` (as used in (d)), while `step = 1e-10` leads to much better agreement

$$\Delta t_0 = 1.0 \times 10^{-12}, \Delta a = 1.1 \times 10^{-7}, \Delta w = 2.9 \times 10^{-12}.$$

But in either case, the values lie within each others bounds and the deviations are not statistically significant. It was shown in P-set 1 (or rather in class leading up to PS1), that an optimal step-size for the forward difference method is roughly proportional to $\sqrt{\epsilon_f}$ under the assumption that f and f'' are approximately equal to each other. Under this assumption an ideal step-size is calculated to be on the order of `step = 1e-8`. Clearly, there is no single `step` to minimize all truncation errors at the same time, since each second-order partial derivative behaves differently. In principle, one could minimize the truncation errors further by using separate `step`s for the individual parameters. It seems that the relative differences are therefore statistically significant, only when a sub-ideal choice for `step` is used.

(d) At the heart of this implementation lies the forward difference method for each model parameter m_i:

```
[TRUNCATED]
# numerical derivs
 step = 1e-6
 for i, v in enumerate(m):
     dm = np.zeros(len(m))
     dm[i] = step
     grad[:,i] = (f(m + dm) - f(m))/step
```

where `grad` has the same functionality as above. `f(m)` represents the given model that calls three instances of `single_lorentzian(t, m)`

```
 def single_lorentzian(t, m):
     return m[0]/(1 + ((t - m[1])**2)/(m[2]**2))
 def grad_lorentz(t, m):
     [TRUNCATED]
     fun = single_lorentzian
     # define individual instance of Lorentzian
     f_single = lambda amp, center, width : (
         fun(t, np.array([amp, center, width]))
```

```
    )
# triple Lorentzian model
f = lambda m : (
    f_single(m[0], m[3], m[5]) +
    f_single(m[1], m[3] + m[4], m[5]) +
    f_single(m[2], m[3] - m[4], m[5]))
```

`lambda` Notation is useful here, since calling the functions is easier and the code is overall more readable. Using this method, the model parameters are found to be

$$a = 1.44 \qquad b = 6.53 \times 10^{-2} \qquad c = 1.05 \times 10^{-1}$$
$$t_0 = 1.93 \times 10^{-4} \qquad dt = 4.44 \times 10^{-5} \qquad w = 1.60 \times 10^{-5}$$

with corresponding uncertainties (calculated similarly as described in (b))

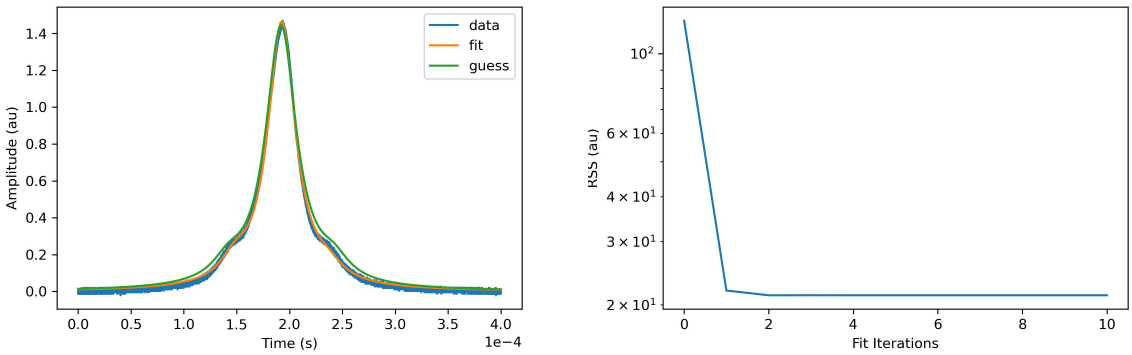
$$\delta a = 2.3 \times 10^{-4} \qquad \delta b = 2.2 \times 10^{-4} \qquad \delta c = 2.2 \times 10^{-4}$$
$$\delta t_0 = 2.7 \times 10^{-9} \qquad \delta dt = 3.2 \times 10^{-8} \qquad \delta w = 4.9 \times 10^{-9}$$

Complete Code:

```
def single_lorentzian(t, m):
    return m[0]/(1 + ((t - m[1])**2)/(m[2]**2))

def grad_lorentz(t, m):
    # model params m
    # m[0] = a      m[3] = t_0
    # m[1] = b      m[4] = dt
    # m[2] = c      m[5] = w
    fun = single_lorentzian
    f_single = lambda amp, center, width : (
        fun(t, np.array([amp, center, width]))
        )
    f = lambda m : (
        f_single(m[0], m[3], m[5]) +
        f_single(m[1], m[3] + m[4], m[5]) +
        f_single(m[2], m[3] - m[4], m[5]))
    pred = f(m)
    # make empty array to store gradient values
    grad = np.empty([t.size, 6])
    # numerical derivs
    step = 1e-6
    for i, v in enumerate(m):
        dm = np.zeros(len(m))
        dm[i] = step
        grad[:,i] = (f(m + dm) - f(m))/step
    return pred, grad
```

Fig. 2 shows fitting the data to a sum of three Lorentzian functions. This model compares better with those used in (a)/(c). One can compare the RSS values after some number of fit iterations to see that the initial implementation led to $RSS \approxeq 60$ while this one leads to $RSS \approxeq 20$.



(a) Fit (b) Residual Squared Sum (RSS)

Figure 2: Single Lorentzian Fit to the signal and associated error in the number of fitting iterations.

(e) If the noise was uncorrelated, we would expect the residual to randomly scatter around the center line. As seen in Fig. 3, this is clearly not the case.

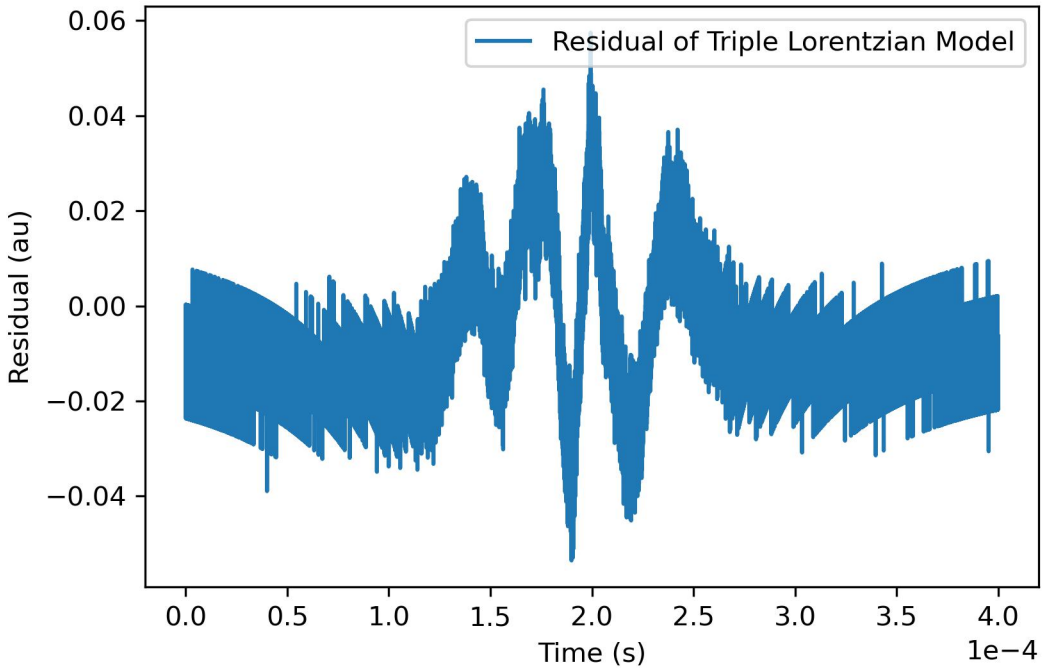


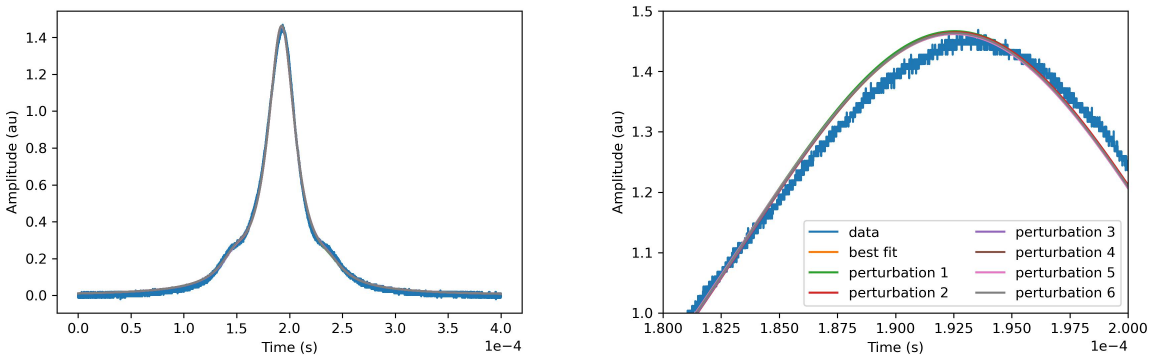
Figure 3: Residual in triple Lorentz model from (d).

(f) The following code generates realizations of models with parameters offset from their best-fit values as found in (d) from the full covariance matrix.

```
# calculate covariance matrix
cov = np.linalg.inv(lhs)*err
sampler = 6
```

```
for j in range(sampler):
    alt_params = np.random.multivariate_normal(model_params[-1], cov)
    plt.plot(time, grad_lorentz(time, alt_params)[0],
        label=f"perturbation {j + 1}")
```

The offset is determined from a 6D normal distribution of the parameters around their best fit values. The diagonal entries are the variances of the parameters "on their own", while the off-diagonals are the co-variances and determine how two parameters are correlated, i.e. if one were to take a slice of this multi-dimensional normal distribution parallel to two parameter axis, the amount by which a sphere (in the uncorrelated case) is distorted into an ellipse is given by the off-diagonal entries. The results are shown in Fig. 4. The best-fit, as well as all perturbations fail to "correctly" reproduce



(a) Best fit and perturbed fits.

(b) Enlarged view.

Figure 4: Perturbing the best fit parameters by offsets set through the covariance matrix gives separate implementations. 4b shows a magnified cut-out near the central maximum.

the data. This is especially apparent around the central maximum. The relative χ^2 values, $\Delta\chi^2$, are shown in Fig. 5. On average, $\Delta\chi^2 \approxeq 0.08$.

(g) To run an MCMC chain for this fitting problem, I first imported the results from the previous exercises - most importantly the best-fit parameters, as well as the associated covariances:

```
import p_set4d_e as prev
import numpy as np
from matplotlib import pyplot as plt

# from previous answer, generate step size array and initial values
par_best= prev.model_params[-1]
par_errs = np.sqrt(np.diag(np.linalg.inv(prev.lhs)))
```

Since it will cancel out in later computation anyway, I dropped the calculation of σ^2. To evaluate the quality of a specific run in the MCMC chain, a helper function for the computation of χ^2 is implemented

```
def get_chi_sq(data, params):
    pred, grad = grad_lorentz(time, params)
```

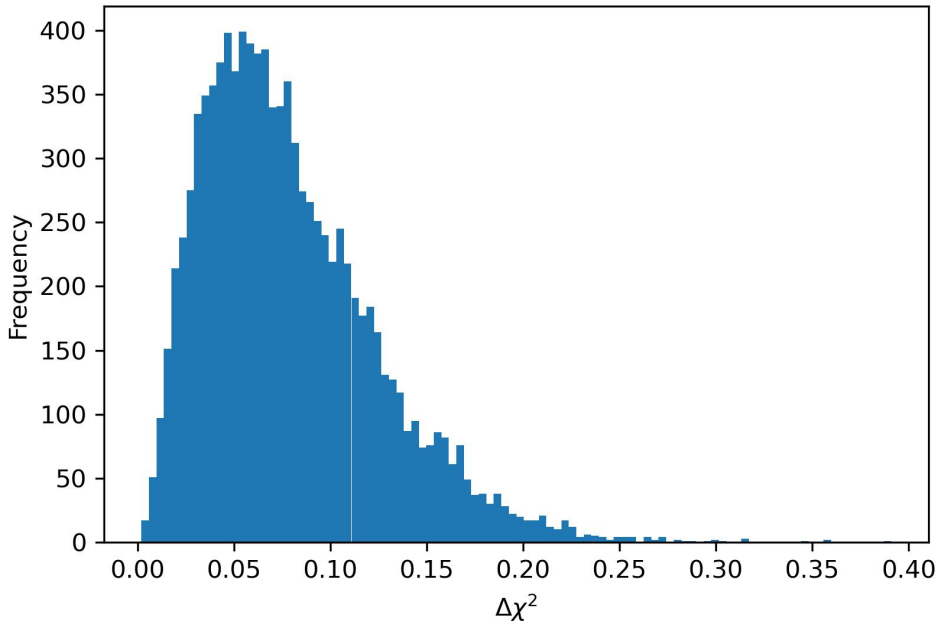


Figure 5: $\Delta\chi^2$ for 10,000 realizations.

```
    chisq = np.sum((data - pred)**2)
    return chisq
```

The initial parameters are taken from the best-fit values and their covariances:

```
n_steps = 100000
steps_taken = 0
chi_sq = []
par = []
# initialize parameter list by using best fit +/- a little something
init_par = par_best + 3.0*np.random.randn(len(par_best))*par_errs
chi_sq.append(get_chi_sq(d, init_par))
par.append(init_par)
```

The MCMC chain is then started for `n_steps` runs. The result is shown below. The typical run-in is followed by a fast convergence to an optimum. To visualize the parameter distribution in the chain, corner plots are shown in Fig. 7 with corresponding central values for each of the parameters, as well as their errors. While the error bars stayed approximately the same, the parameter values did change - even significantly for b which jumped from $b_{\text{Newton}} = 6.47 \times 10^{-2}$ to $b_{\text{MCMC}} = 6.27 \times 10^{-2}$. Complete Code:

```
import p_set4d_e as prev
import numpy as np
from matplotlib import pyplot as plt
# from previous answer, generate step size array and initial values
par_best= prev.model_params[-1]
par_errs = np.sqrt(np.diag(np.linalg.inv(prev.lhs)))
```

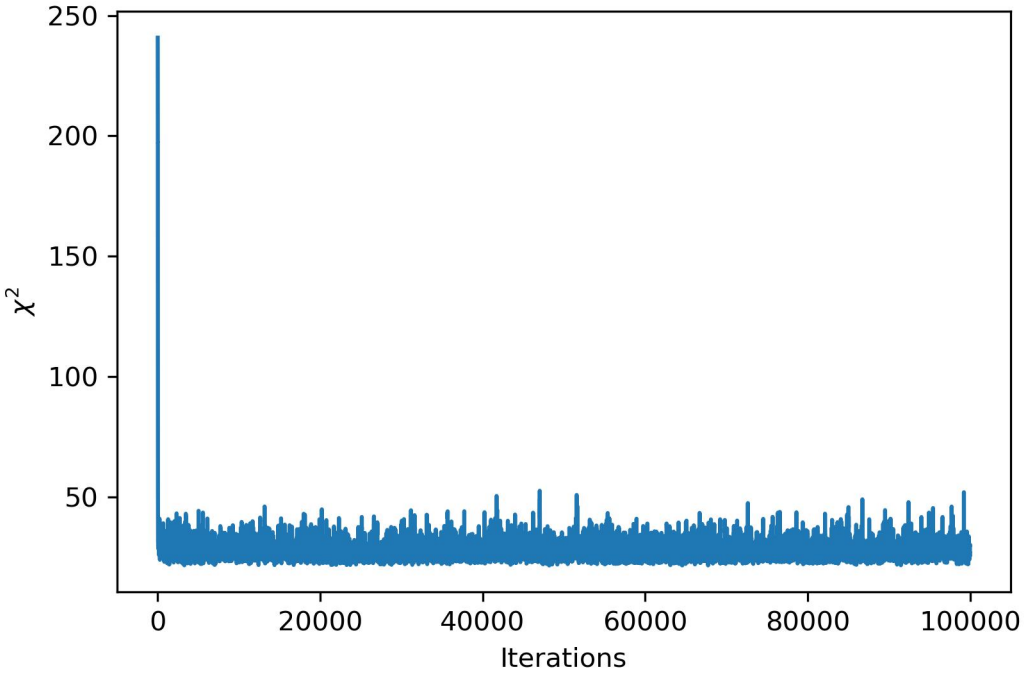


Figure 6: χ^2 (unscaled) for MCMC with 100,000 cycles.

```
def get_chi_sq(data, params):
    pred, grad = grad_lorentz(time, params)
    chisq = np.sum((data - pred)**2)
    return chisq

n_steps = 100
steps_taken = 0
chi_sq = []
par = []
# initialize parameter list by using best fit +/- a little something
init_par = par_best + 3.0*np.random.randn(len(par_best))*par_errs
chi_sq.append(get_chi_sq(d, init_par))
par.append(init_par)


while steps_taken < n_steps:
    # generate new parameters
    par_new = par[-1] + 1.0*np.random.randn(len(par_best))*par_errs
    # calculate new chi squared
    new_chi = get_chi_sq(d, par_new)
    delta_chi = new_chi - chi_sq[-1]
    prob_step = np.exp(-0.5*delta_chi)
```

```
            # if prob_step is greater then one, chi will have increased
            # and we want to discard that step, otherwise accept with a certain
            # probability:
            if prob_step > np.random.rand(1):
                par.append(par_new)
                chi_sq.append(new_chi)
            else:
                # if step is not taken, copy last entries for params and chi_sq
                par.append(par[-1])
                chi_sq.append(chi_sq[-1])
            steps_taken += 1
```

(h) I calculate the cavity width with the equation

$$x_{\text{cavity}} = \frac{dt}{w}(9\text{GHz}) \tag{4}$$

to be $x_{\text{cavity}} = 250\text{GHz}$ with a fractional uncertainty of 0.001, i.e. an absolute uncertainty of 0.25GHz. I haven't actually derived this equation, but taken it from a class colleague with my own parameters.

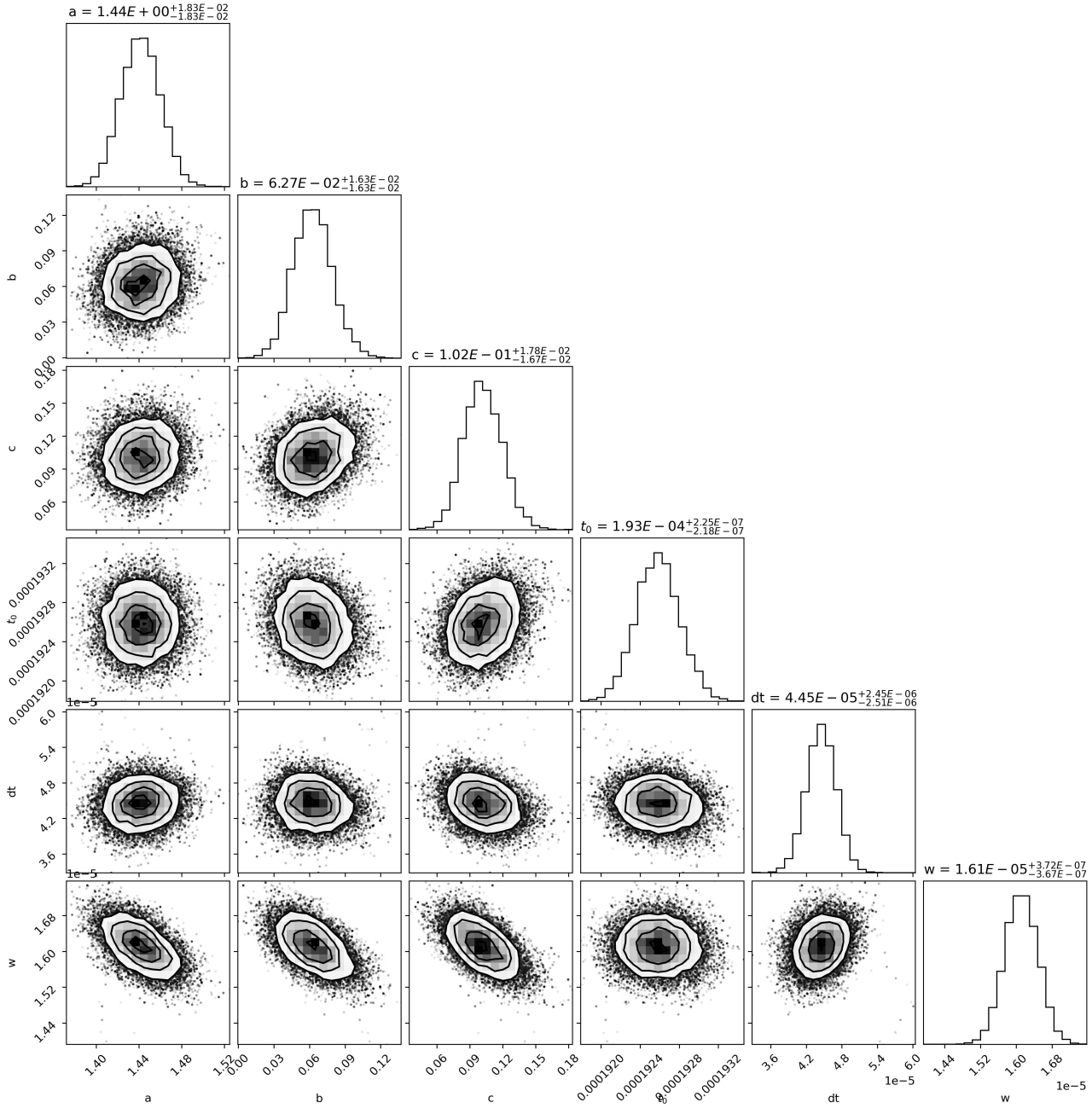


Figure 7: Corner Plots