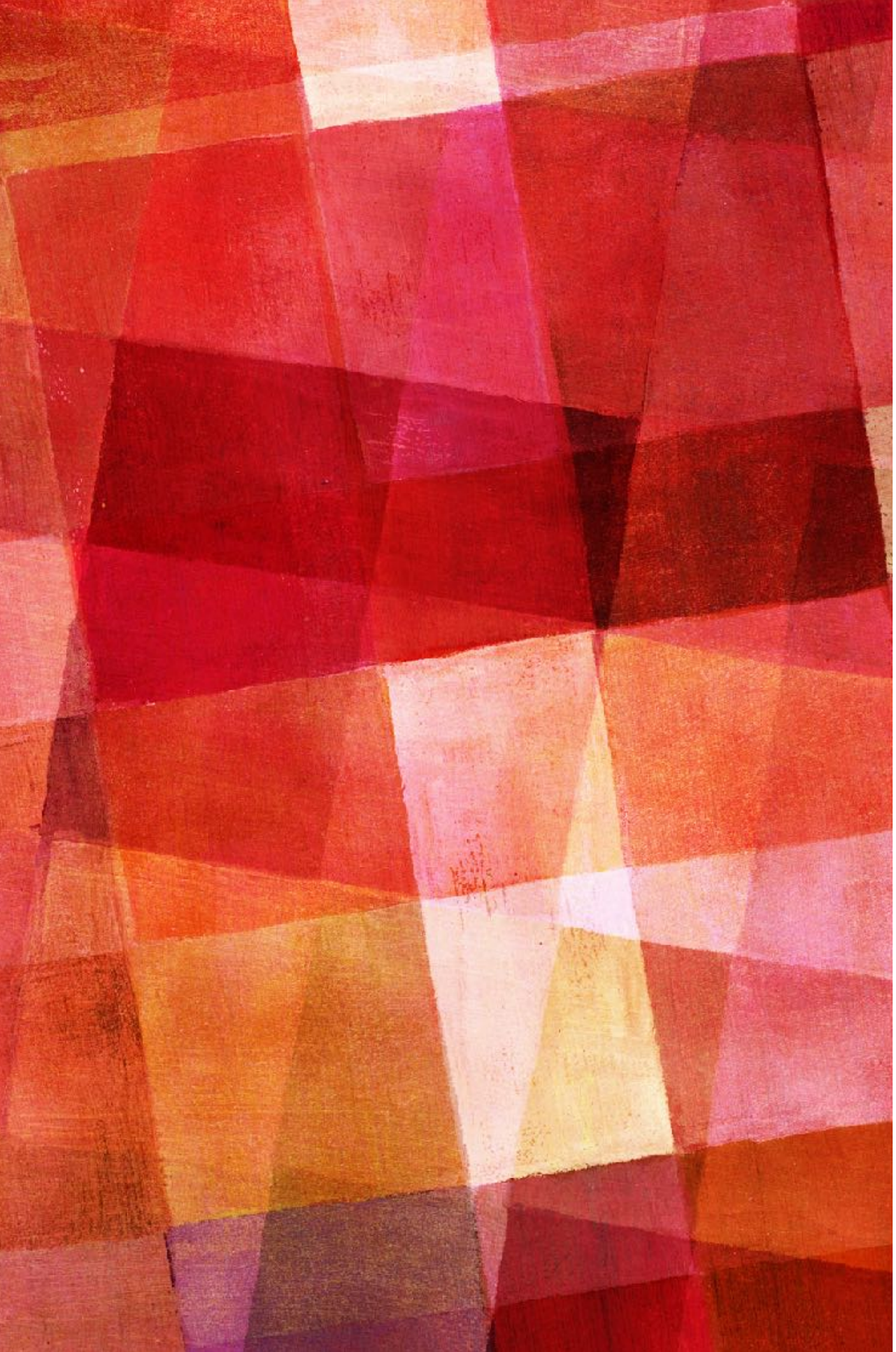
PARAMETER ESTIMATION

Ari Maller

OPTIMIZATION

- One of the major uses of computers is for finding optimum solutions to problems. That can range from our traveling salesman, to designing airplane wings or recognizing speech.
- Optimization is usually the same as finding the global maximum or minimum of a function. If our problem is one dimensional then the methods we discussed earlier can be used. Unfortunately, many problems of interest are of higher dimensionality.
- While optimization can be about almost anything, we will focus on *parameter estimation*, that is finding the 'best' choice of parameters for a model given some data. While the methods used to solve this type of problem are the same as any other optimization problem, parameter estimation is something that occurs in all branches of physics (science).



METHODS

- Linear Least Squares
- Nonlinear Least Squares
- Amoeba
- Gradient Descent
- MCMC
- Simulated Annealing
- Genetic Algorithm

FITTING A LINE TO DATA

- Let's start with one of the most common practices in the sciences. Fitting a line to data. There is a rather standard approach to this, but we will explore it in detail to understand exactly what one is doing and enable discussion of the issues associated with that.

- First off, it is often unclear that fitting a line to data is a good idea. If one has a theoretical model that suggest the y-values of the data should depend linearly on the x-values, then it is a sensible thing to do. But in practice people often fit a line based not on a theory, but simply because a line represents a simple function.

LEAST-SQUARE FITTING

- If you have a set of two-dimensional points (x,y) that depart from a perfect, narrow, straight line y=mx+b only by the addition of Gaussian-distributed noise of known amplitudes in the y direction only, then the maximum-likelihood line for the points has a slope m and intercept b that can be obtained by linear matrix-algebra operation known as 'weighted linear least-square fitting'.
- Let us consider this situation first and then discuss its validity. We want to find the function f(x) = mx+b. We have some number N of sample points for x and y and errors. We can construct the following matrices, **Y, A, C** from those points and errors

LEAST-SQUARE FITTING

$$Y = \begin{bmatrix} y_1 \\ y_2 \\ ... \\ y_N \end{bmatrix} \qquad A = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ ... & ... \\ 1 & x_N \end{bmatrix} \qquad C = \begin{bmatrix} \sigma_{y1}^2 & 0 & ... & 0 \\ 0 & \sigma_{y2}^2 & ... & 0 \\ & & ... & \\ 0 & 0 & ... & \sigma_{yN}^2 \end{bmatrix}$$

- The matrix C can have off diagonal elements in which case it would be a covariance matrix.

- The best fit values for m and b are the components of the vector X

$$\begin{bmatrix} b \\ m \end{bmatrix} = X = [A^T C^{-1} A]^{-1} [A^T C^{-1} Y]$$

- We can see where this comes from by starting with Y=AX, but this is overdetermined if N>2. So we weigh the points by the inverse covariance matrix and then multiple by A^T to reduce the dimensionality.

$$A^T C^{-1} Y = A^T C^{-1} A X$$

LEAST-SQUARE FITTING

- This procedure is not arbitrary. It is minimizing an objective function χ^2 (chi-squared) which is the total squared error normalized by the errors.

$$\chi^2 = \sum_{i=1}^{N} \frac{[y_i - f(x_i)]^2}{\sigma_{yi}^2} = [Y - AX]^T C^{-1} [Y - AX]$$

- The previous equation simply solves for the minimal value of this function. But now we get to the question, in what way is minimizing the 'best fit' line?

- In order to address this we will need to create a *generative model* of our data. That is a parameterized model that can produce the data in question in a statistical sense.

OBJECTIVE FUNCTION

- Our generative model shall be what we already described. A linear relationship between f(x)=mx+b plus Gaussian distributed errors. Then the probability of getting a value y_i is

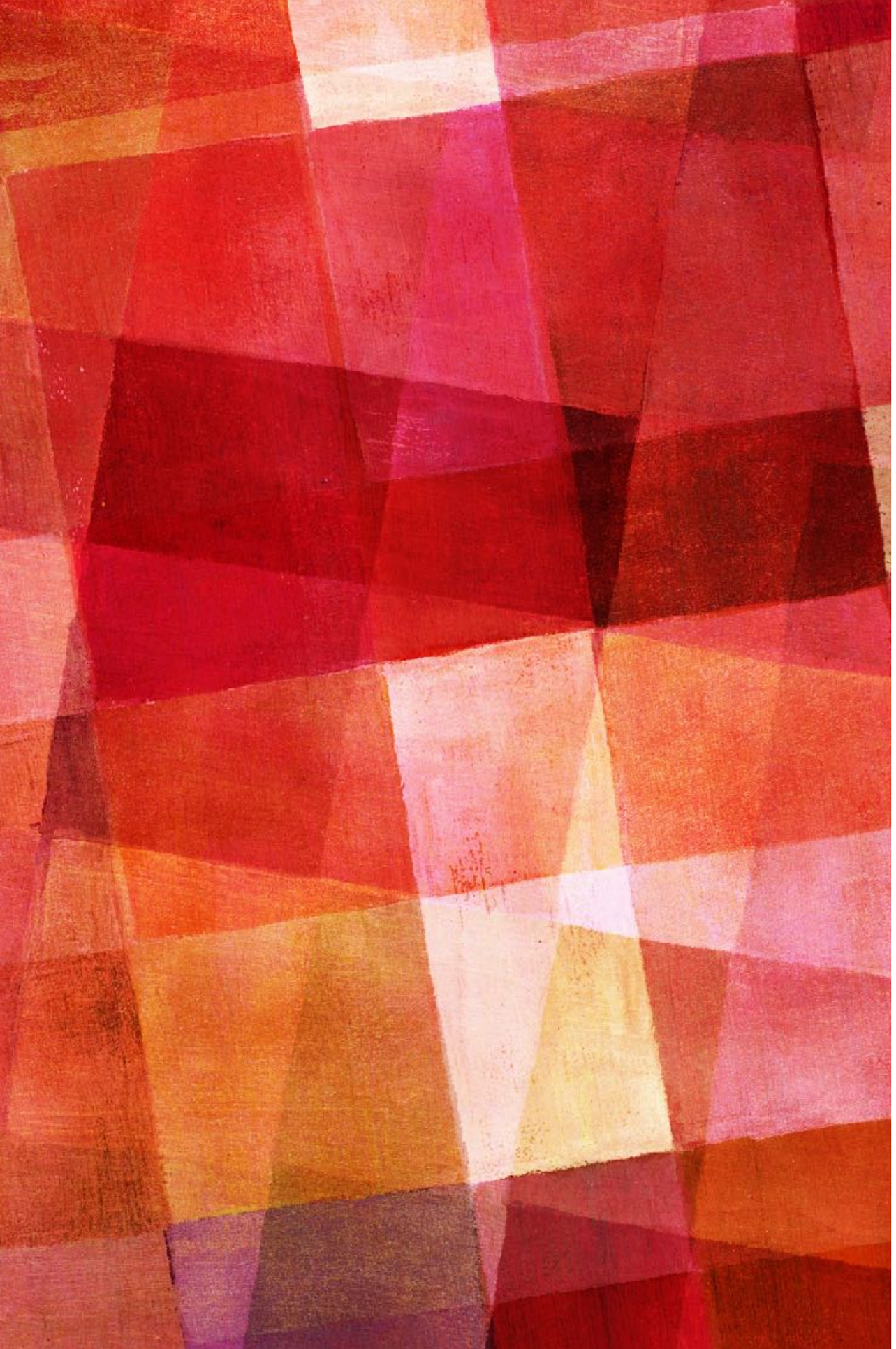
$$p(y_i|x_i,\sigma_{yi},m,b) = \frac{1}{\sqrt{2\pi\sigma_{yi}^2}}\exp\left(-\frac{[y_i-mx_i-b]^2}{\sigma_{yi}^2}\right)$$

- Thus the likelihood, $\mathscr{L}$, of getting the y-values we have assuming the points are independent is

$$\mathscr{L} = \prod_{i=1}^{N} p(y_i|x_i,\sigma_{yi},m,b)$$

- taking the log and letting K equal the value outside the sum

$$\log\mathscr{L} = K - \sum_{i=1}^{N}\frac{[y_i-mx_i-b]^2}{2\sigma_{yi}^2} = K - \frac{1}{2}\chi^2$$



EXERCISE 12.1

- Use the generative model to generate some data and plot it. Let the function be a line f(x) = mx + b, with m=1 and b=0. Let the standard deviation of the y data points be $\sigma_{yi} = 0.2$. Generate 50 random points between 0 and 1 and plot them.

- Generative models are a very powerful computational tool. They allow you to test your analysis on data which by construction you know everything about.

NUMPY POLYFIT

- The python numpy package includes a function for performing linear least square fitting called polyfit.
- Note that while we have been focusing on fitting a line, higher order polynomials like $f(x) = qx^2 + mx + b$ are still linear in the parameters and thus can be solved with just linear algebra.
- The polyfit function takes x and y values, the degree of the polynomial and weights. It returns the best fit and errors.

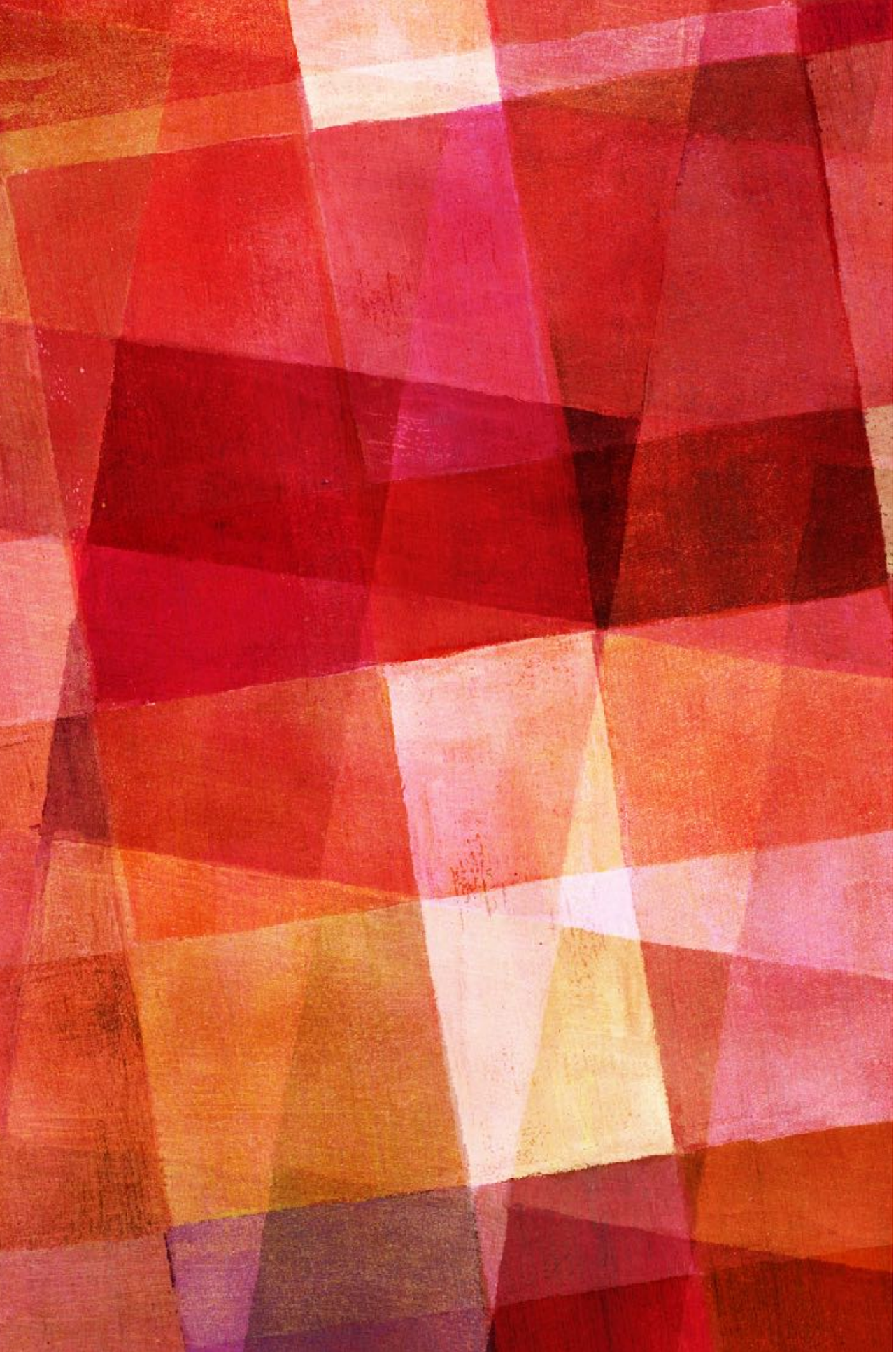
```
params, error_matrix=polyfit(x, y, 2, w=1/y_sigma, cov=True)
```

UNCERTAINTIES IN THE BEST FIT PARAMETERS

- Let's look at the uncertainties in our best fit parameters. The standard output from linear-least squares is just a covariance matrix given by

$$\begin{bmatrix} \sigma_b^2 & \sigma_{mb} \\ \sigma_{mb} & \sigma_m^2 \end{bmatrix} = [A^T C^{-1} A]^{-1}$$

- Often people only quote the diagonal element, though it is very likely that the off-diagonal values are large (i.e. the fit parameter are likely to be highly correlated).
- These uncertainties will only be correct if our generative model holds exactly. Since this is rarely the case it is useful to use uncertainty estimates derived from the data by either the *bootstrapping* or *jackknife* techniques.



EXERCISE 12.2

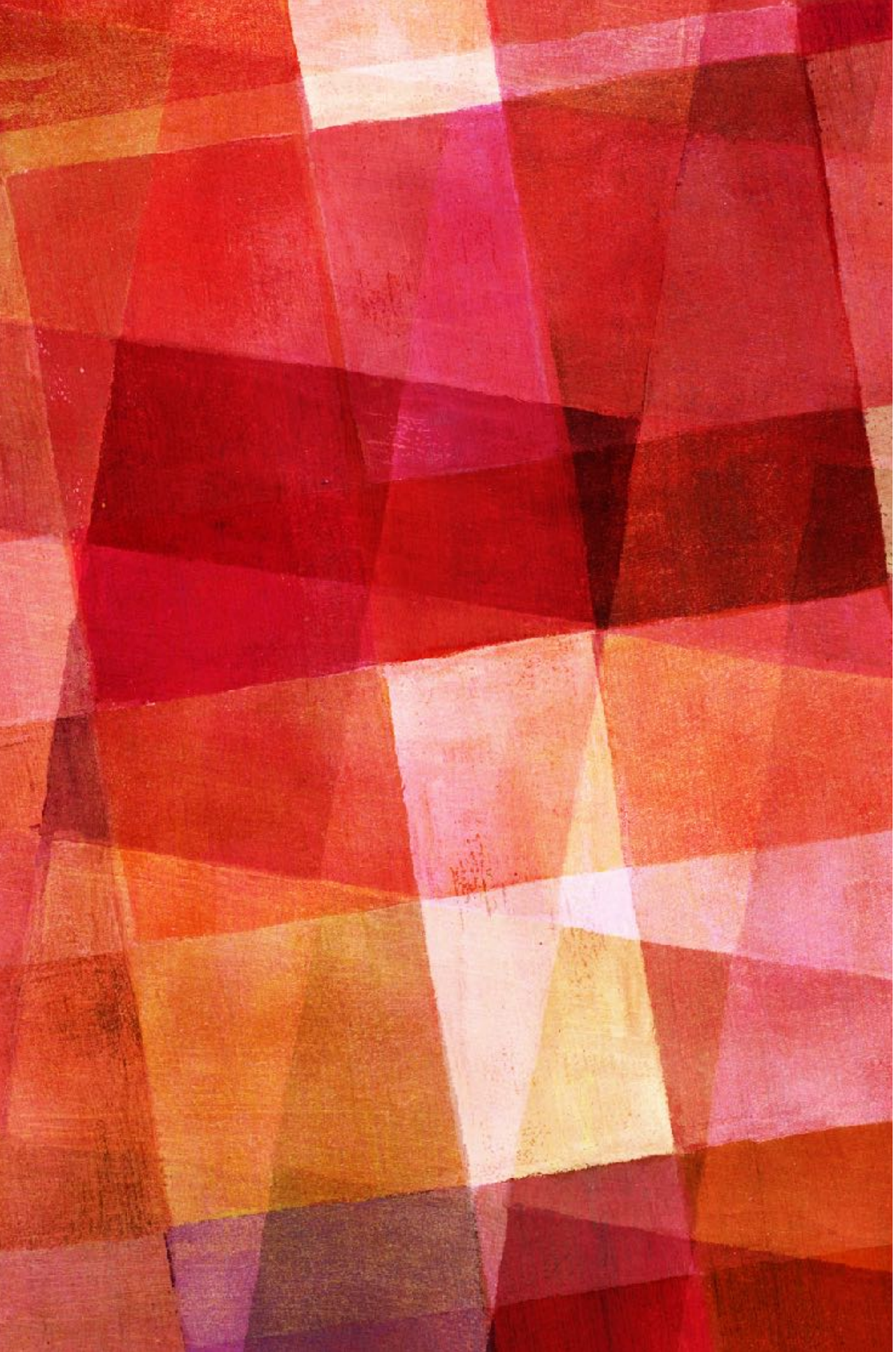
- Use polyfit to fit your generated data. Determine the difference between your fit parameters and their true values. Compare this to the diagonal elements of the error matrix (note the matrix is σ^2.)
- Now loop over this 30 times, generating new data and then fitting it. Make a plot of the distribution of differences between the fit parameters and their true values (2 distributions).

BOOTSTRAPING RESAMPLING

- Bootstraping is drawing N random data points from the N data points that you have, with replacement. That is some data points get dropped and some get sampled 2 or more times. Each j sample one creates this way you then use to estimate your parameters, m_j and b_j. With M trails an estimate on the variance in your determination of m is then

$$\sigma_m^2 = \frac{1}{M}\sum_{j=1}^{M}(m_j - m)^2$$

- The variance of b and the covariance can be calculated in a similar way (i.e., sum over $(b_j - b)^2$ and $(m_j-m)(b_j-b)$).



EXERCISE 12.3

- Now create bootstrap resamples of your data and use polyfit to fit a line to each resampling.
- Plot the distribution of your fit parameters and measure their standard deviation. How do they compare to what you found in Ex. 12.2?

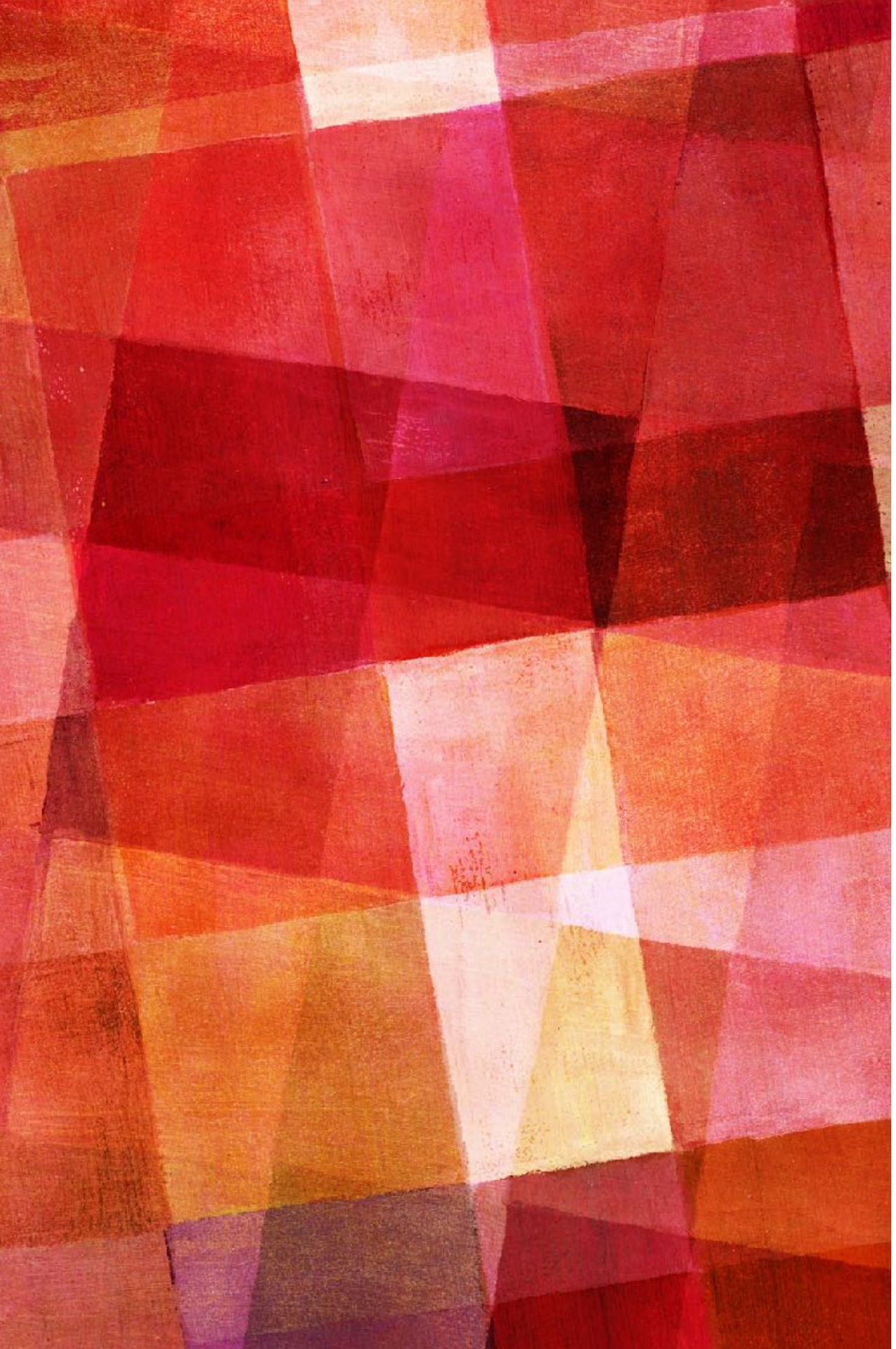
JACKKNIFE RESAMPLING

- In jackknife resampling one removes the one data point and then recalculates the estimated parameters. This is done for each data point in the sample, so for N data points you get N resamples data sets. The variance is then given by

$$\sigma_m^2 = \frac{N-1}{N}\sum_{i=1}^{N}(m_i - m)^2$$

- with the obvious extension for b and the off-diagonal elements.

- Note that both jackknife and bootstrapping will not give good results if the model (here a linear function) is not a good fit to the data.



EXERCISE 12.3

- Now use jackknife resampling to estimate the variance in your data set. Remove one of the data points and run polyfit. Numpy delete may be useful here. Note that the formula for the variance is different when using the jackknife and bootstrapping methods.
- Plot the distribution of your fit parameters in this case and compare to the bootstrapping method.

NONGAUSSIAN ERRORS

- What if our errors are not Gaussian? The best approach is to include the non-Gaussianity in our generative model. For example let's suppose our errors are the sum or mixture of k Gaussian distributions. This is a good approach because most reasonable error distributions can be approximated as the sum of Gaussian distributions. In this case our probability of getting the value y_i becomes

$$p(y_i|x_i, \sigma_{yi}, m, b) = \sum_{j=1}^{k} \frac{a_{ij}}{\sqrt{2\pi\sigma_{yij}^2}} \exp\left(- \frac{[y_i + \Delta y_{ij} - mx_i - b]^2}{\sigma_{yij}^2} \right)$$

- where a_{ij} are the amplitudes and Δy_{ij} are offsets of the means of the Gaussians. With this equation we can then construct a likelihood. Minimizing it is much harder, but still leads to the correct 'best fit' parameters.

GOODNESS OF FIT

- If the assumption of our simplest generative model are correct (independent data points, Gaussian errors, linear relationship and no error in the x values), then the expectation is that the data points will contribute a mean square error comparable to σ^2_{yi}. The distribution of χ^2 can be written analytically and is called the chi-square distribution. The expected value of χ^2 is

$$\chi^2 = N - 2 \pm \sqrt{2(N-2)}$$

- In practice values of χ^2 smaller and larger than this will occur. In both cases they suggest the model is incorrect, either because it is the wrong function or because the errors are under or over estimated. One common problem is errors with covariance being treated as independent.

GOODNESS OF FIT

- Correlated measurements are not uncommon and can be accounted for if χ^2 is defined in terms of a covariance matrix

$$\chi^2 = [Y - AX]^T C^{-1} [Y - AX] = \sum_{i=1}^{N} \sum_{j=1}^{N} w_{ij} [y_i - f(x_i)][y_j - f(x_j)]$$

- w_{ij} are the elements of the inverse covariance matrix C^{-1}.

- So far we have only considered uncertainties in the y-variable, but most real world data has uncertainties in the x-variables too. If we have x and y errors and possible covariance we can define a matrix

$$S_i \equiv \begin{bmatrix} \sigma_{xi}^2 & \sigma_{xyi} \\ \sigma_{xyi} & \sigma_{yi}^2 \end{bmatrix}$$

2D UNCERTAINTIES

- In this case if the distribution of the errors are Gaussian we can generalize our expression for the probability of getting a value y_i to a point x_i,y_i

$$p(x_i, y_i|S_i, x, y) = \frac{1}{2\pi \det(S_i)} \exp\left(-\frac{1}{2}[Z_i - Z]^T S_i^{-1}[Z_i - Z] \right)$$

- where Z=[x,y] and Z_i = [x_i,y_i]. In order to fit a line to this data we need to construct a likelihood from the above probability. However, given a data point, we don't know what x,y it is displaced from. Errors that move points along the line appear like points with no errors.
- One way to treat this is to look at how far the data points are from the line measured as a perpendicular distance.

2D UNCERTAINTIES

- That is we define a unit vector perpendicular to the slope of our line

$$\hat{v} = \frac{1}{\sqrt{1+m^2}} \begin{bmatrix} -m \\ 1 \end{bmatrix} = \begin{bmatrix} -\sin\theta \\ \cos\theta \end{bmatrix}$$

- where $\theta = \arctan(m)$. We can define the orthogonal displacement Δ_i of each point by

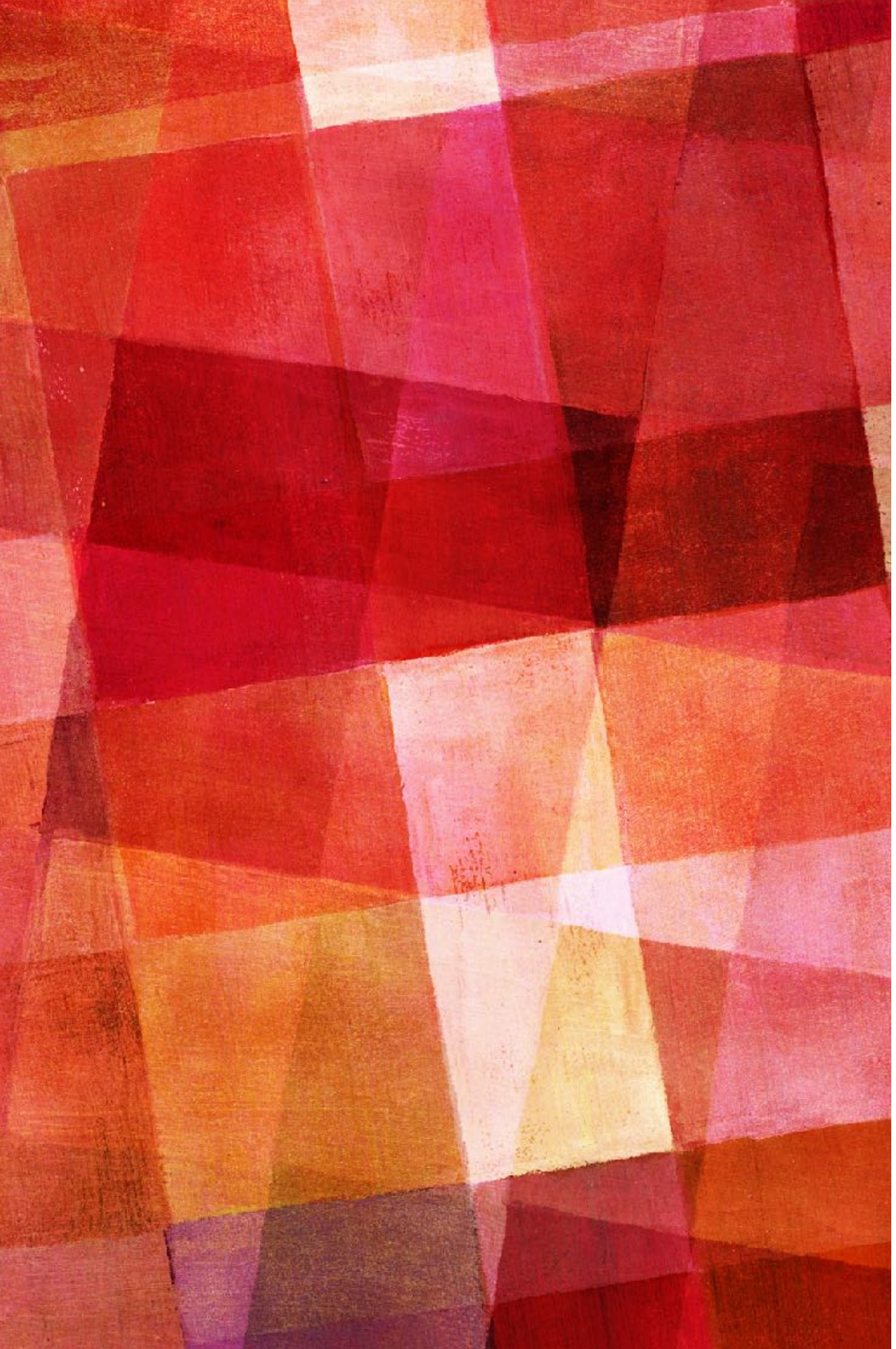
$$\Delta_i = \hat{v}^T Z_i - b\cos\theta$$

- each points covariance matrix can be projected into a variance

$$\Sigma_i^2 = \hat{v}^T S_i \hat{v}$$

- and then the log likelihood can be written down as

$$\ln \mathscr{L} = K - \sum_{i=1}^{N} \frac{\Delta_i^2}{2\Sigma_i^2}$$



EXERCISE 12.4

- Now generate new data, but adding Gaussian x error of $\sigma_{xi} = 0.2$. Find the best parameters and their errors in this case.
- Do this 30 times and look at the distribution of your best fit parameters, how does this compare to what you got in Ex. 12.2

NONLINEAR LEAST-SQUARES

- So far we have focussed on fitting a line to data, but what if the underlying relationship between x and y is not linear, but some other function f(x).
- As far as the analysis goes, almost nothing changes. χ^2 can be expressed in a similar manner, the only difference is that we can no longer solve the problem with linear algebra. If we still assume the error are Gaussian this gives us nonlinear least squares. Our objective function is still the sum of the weighted distances of our points from our model function, the only difference is the numerical technique we use to find the minimum.
- However, if the objective function is not χ^2, then we can still maximize a likelihood, but we can't do it with least-squares. We will have to use some other method of optimization.

GAUSS-NEWTON ALGORITHM

- The Gauss-Newton algorithm is an extension of Newton's method for finding the minimum of a function. Remember Newton's method was to use the derivative to step our guess towards the minimum

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)}$$

- The Gauss-Newton applied to N variables and N equations minimizes

$$\chi^2 = \sum_{i=1}^{N} r_i^2(x_i)$$

- by making steps

$$\vec{x}_{n+1} = \vec{x}_n - J_r^{-1}\vec{r}(\vec{x}_n)$$

- where J is the Jacobian of the functions r, $(J_r)_{ij} = \frac{\partial r_i}{\partial x_j}$

GRADIENT DESCENT

- Another first-order method for finding minimizes of many variable functions is called *gradient descent* or *steepest descent*.
- In this method we use the gradient of our function to estimate the next step. So we have

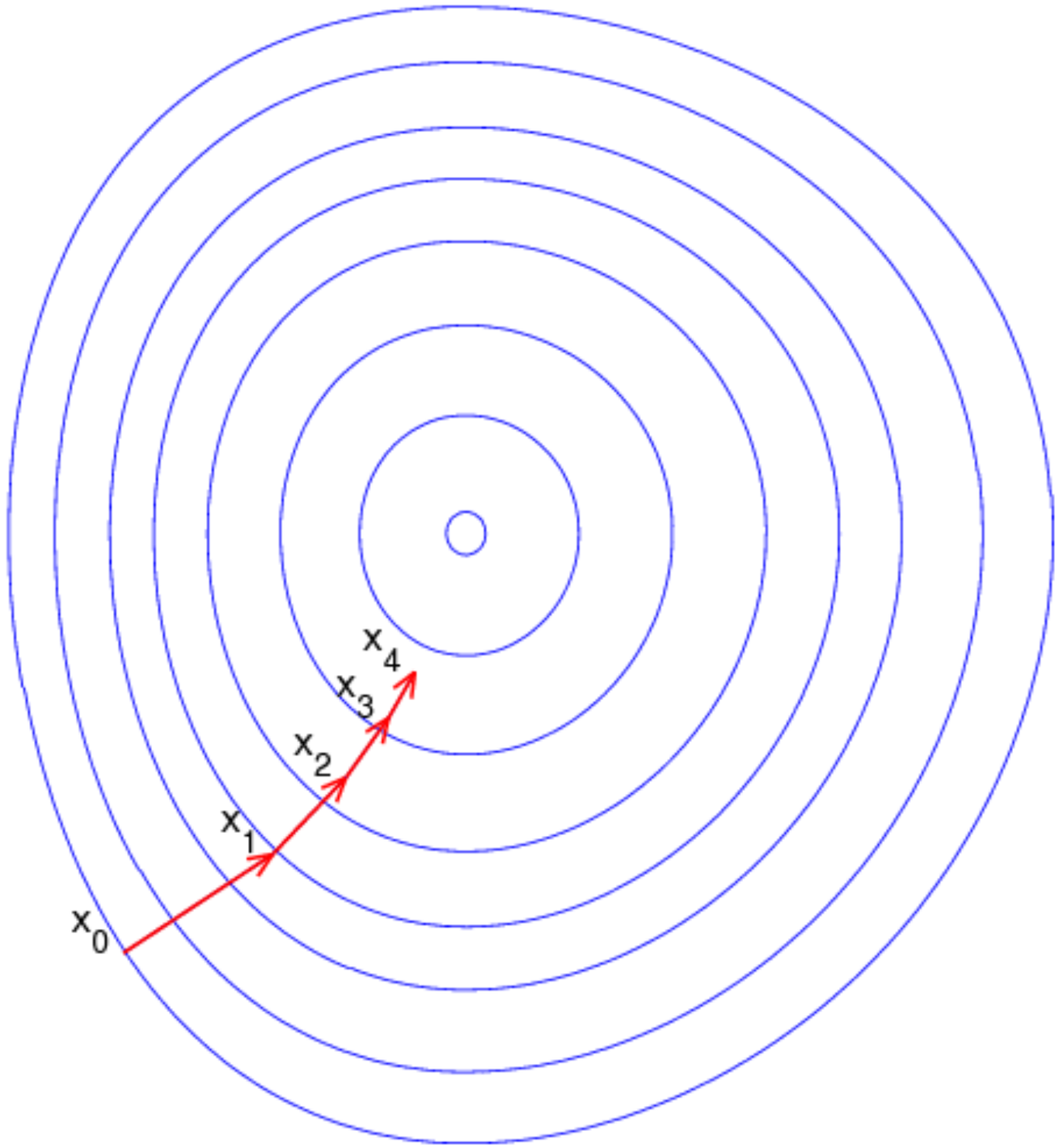
$$\vec{x}_{n+1} = \vec{x}_n - h\nabla F(\vec{x}_n)$$

- Notice that unlike Gauss-Newton this work on any function, not just a function which is the sum of squares of functions.

GRADIENT DESCENT

- Gradient descent looks like this. Each step you calculate the gradient and then advance in that direction.

- Note that both gradient descent and Gauss-Newton will only give you the local minimum. The minimum you get will depend entirely on your starting guess.



LEVENBERG–MARQUARDT ALGORITHM

- The standard algorithm for solving nonlinear least square problems is the Levenberg-Marquardt algorithm.
- This algorithm combines the Gauss-Newton method and the gradient descent method by introducing a damping parameter.
- If the value of the parameter is adjusted for each iteration (like adaptive step sizes). If the parameter is small the method becomes Gauss-Newton, if it is big the algorithm is like gradient descent.
- Thus Levenberg-Marquardt uses both algorithms trying to take advantage of each strengths. It thus performs very well. Note it is only applicable to sum of squares problems.

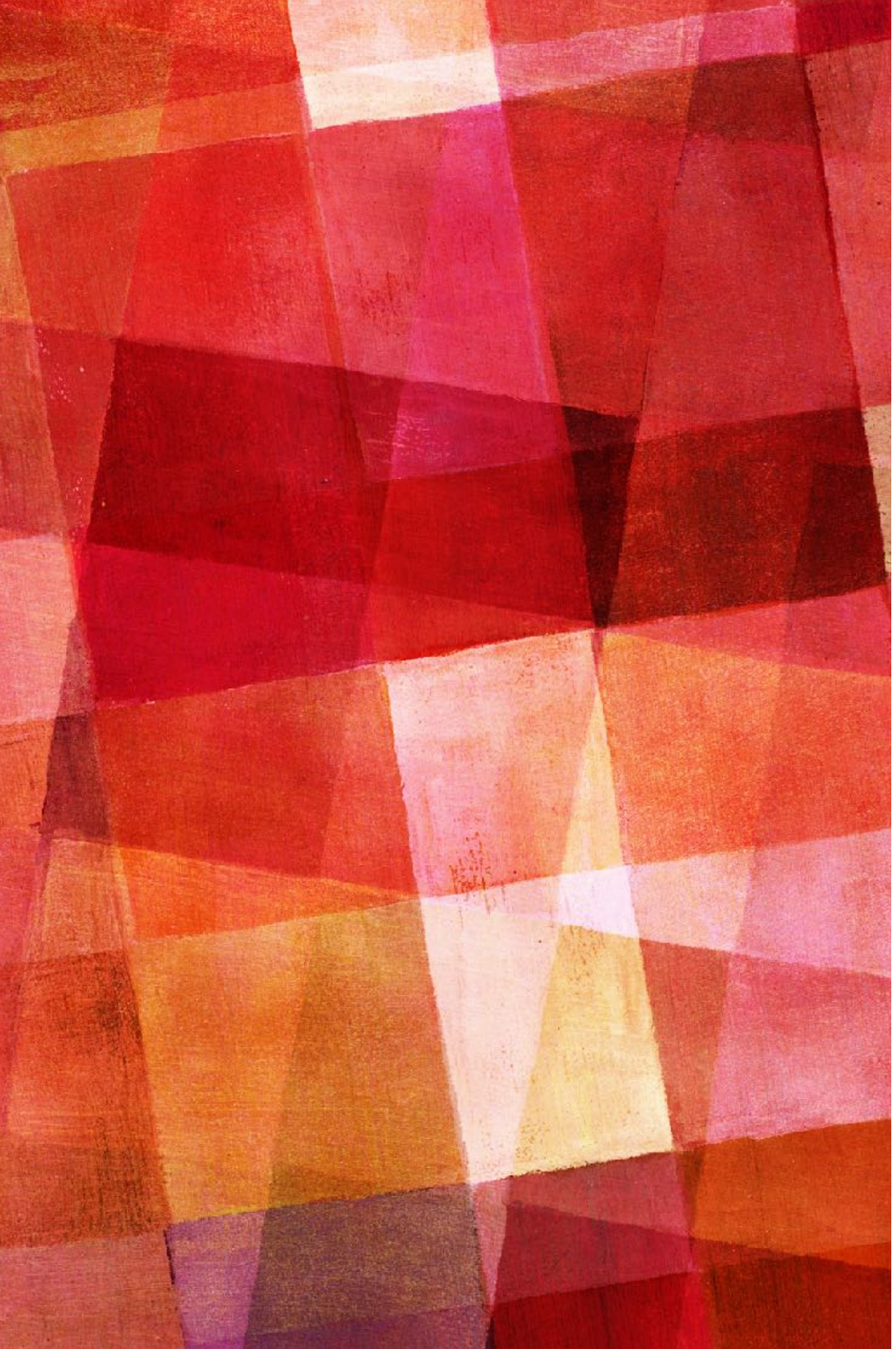
SCIPY CURVE_FIT

- The python scipy package subpackage optimize contains the function curve_fit that performs nonlinear least squares curve fitting using the Levenberg–Marquardt algorithm (though there are other options). The function call looks like

```
params, p_covariance=curve_fit(function, x, y,
p0=starting_guess, sigma=1/sigma or 1/covariance)
```

- The most important thing about using this routine is that you must supply your own function that takes x and the params and returns the y values. Something like

```
def my_func(x, m ,b)
    return m*x+b
```



EXERCISE 12.5

- Now let us generate non linear data. Take $f(x) = \sin(x/2\pi)$ and generate 50 data points with error $\sigma_{yi} = 0.2$ for x in the range [0,1]
- Fit a line to the data using polyfit. Then fit a quadratic to the data.
- Then using curve_fit to fit a sin function to the data.
- One of the weaknesses of these functions is they do not return a χ^2 value for your fit. However, it is easy enough to calculate it once we have the best fit parameters. Determine χ^2 for each of your three fits.

SUMMARY

- Least-square fitting is often done to fit a function to data. This is justified if the errors on the data are Gaussian which then implies that $\log \mathscr{L} = \chi^2$. In that case, and in that case only minimizing chi-squared is the same as finding the maximum of the likelihood.
- If the errors are not Gaussian, and you can model them, then the correct way to fit data is to maximize the likelihood as given by whatever model you have for the likelihood. In this case the minimization is not least-squares and the algorithms we have discusses so far are not appropriate.
- Instead one must use general optimization algorithms that make no assumption about the functional form over which you are minimizing.

GENERAL MINIMIZATION

- There are many algorithms that can be used to minimize arbitrary functions. The most important differences are whether they make use of derivatives and if they do only first or also second derivatives and how much effort they spend looking for a global minimum instead of a local minimum.
- Much like we saw when we first learned about numeric integration, there is generally a trade off between higher order methods that can converge much faster, but are more likely to have problems with pathological functions.
- There is also a trade off between how hard one wants to search for a global minimum and how much more time that search will take.

SCIPY OPTIMIZE

- The scipy sub-package optimize contains implementations of most of the commonly used minimization algorithms.
- These functions can all be called with a single interface scipy.optimize.minimize. The function call is something like

```
OptimizeResult=minimize(function, guess, method=None)
```

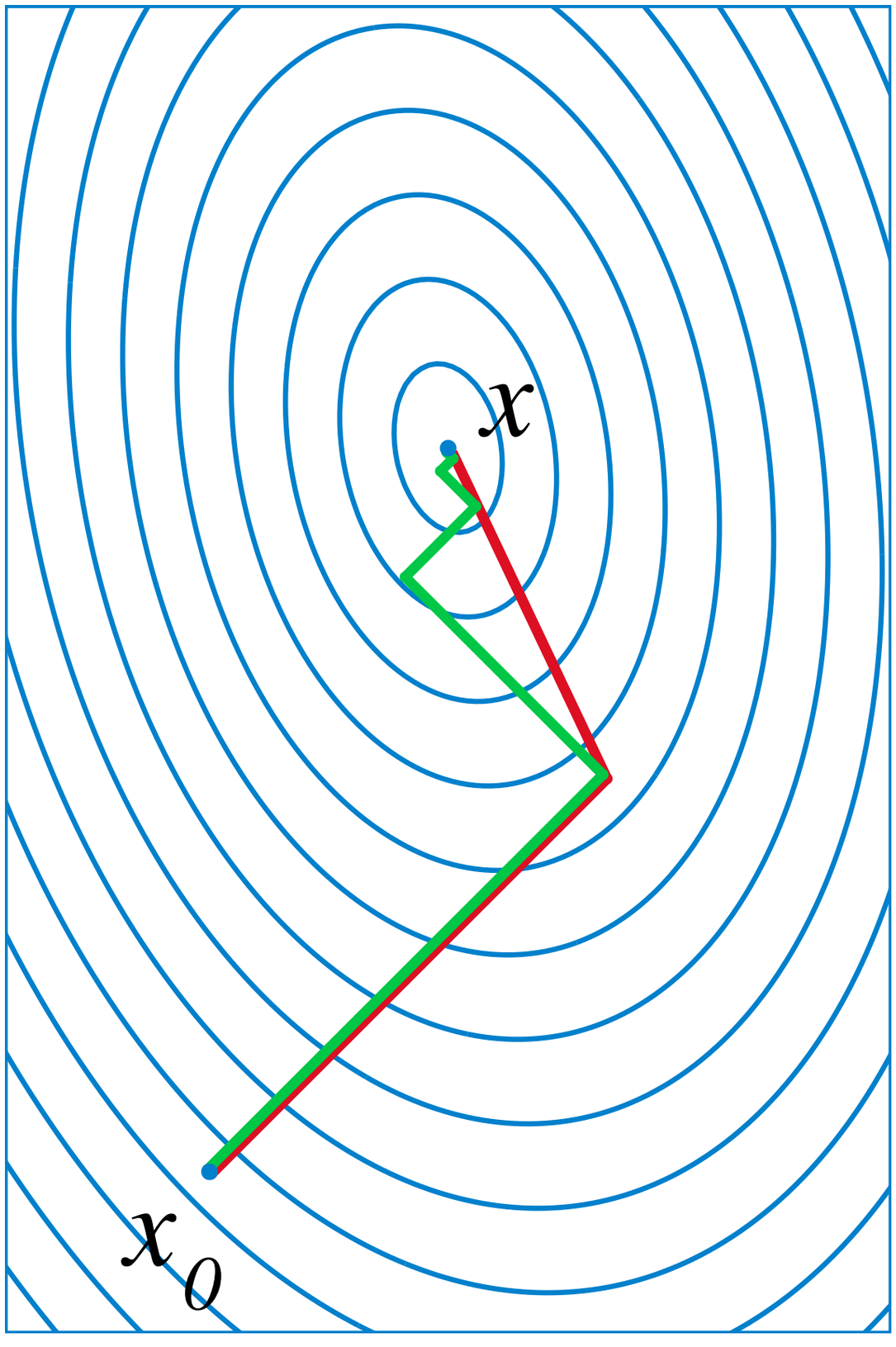
- where function is the objective function you want to minimize, guess is your starting guess and method can be set to different choices if you want to use that algorithm.
- This function returns OptimizeResult which is a class that has data and methods associated with it. So for example OptimizeResult.x is the minimum values you are looking for while OptimizeResults.nit is the number of iterations performed before reaching that result.

POWELL'S METHOD

- Powell's method or more correctly *Powell's conjugate direction method* is a straightforward extension of our 1D methods for finding local minimum in higher dimensional spaces.
- It this method you simply use bi-section to find the minimum in one of the dimensions. Then you do the same thing stepping through the rest of the N-1 dimensions. Then you keep repeating until going through all the N dimensions improves your result by less than a set tolerance.
- This method requires no derivatives and finds a local minimum. Modifications to this method can use other means of finding the minimum in one dimension.

CONJUGATE GRADIENT METHOD

- Powell's method is a conjugate method it that each step since you performing your minimization in a direction orthogonal to the previous step.
- Conjugate methods generalize this by not requiring that you use the basis set of your function (x,y,z…) but finding a new basis set of vectors that performs the minimization much quicker.
- Conjugate gradient methods combine this with the gradient descent method so that you take gradients, but with respect to the optimized basis vectors.
- It thus in general preforms better than Powell's method or gradient descent, but it requires first derivatives and only finds the local minimum.



- An illustration showing the advantage of the conjugate method.
- The green shows normal gradient descent. The line tends to wiggle back and forth as it goes to the minimum because that is the direction of x and y.
- The conjugate method reorients the vector so that it goes straight to the minimum.

BFGS

- The Broyden–Fletcher–Goldfarb–Shanno algorithm is one of the most popular nonlinear minimization algorithms and the default when using scipy minimize.
- This is a quasi-Newton method which makes it analogous to the secant method but in higher dimensional spaces. Newton's method requires the first and second derivatives of the function. The secant method approximates the second derivative numerically.
- In higher dimensions Newton's method for root finding is

$$x_{n+1} = x_n - [J_g(x_n)]^{-1} g(x_n)$$

- where the first derivative has been replaced by the inverse of the Jacobian matrix.

BFGS

- Minimization is just root finding on the gradient of the function so

$$x_{n+1} = x_n - [J_g(x_n)]^{-1} \nabla g(x_n)$$

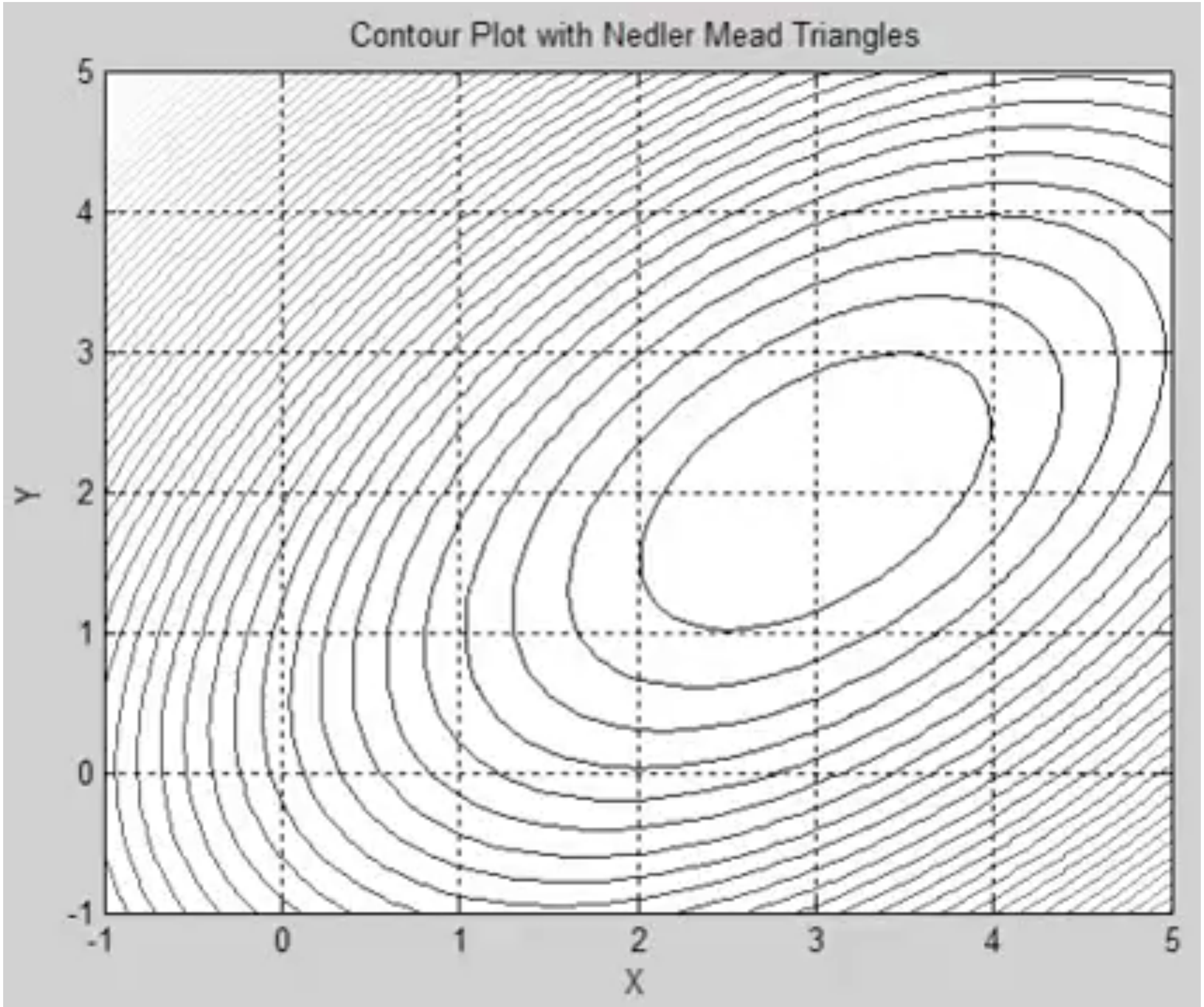
- The Jacobian of the gradient is the Hessian matrix, the matrix of second derivatives. The BFGS method just like the secant method approximates this from the first derivatives.
- Thus is basically works like the secant method with the added complication that you have matrices and not just a single value.
- Similar to how Newton's method could find roots much faster than methods that did not use the derivatives the BFGS method will find a minimum in a small number of steps if the function is well behaved.

AMEOBA

- The Nelder-Mead method or downhill simplex method or amoeba method is a commonly applied numerical method for finding the minimum of an objective function in multidimensional space.
- Its greatest strength is that it doesn't require derivatives, unlike many other methods.
- The method makes use of the concept of a simplex which is a polytope of n+1 vertices in n dimensions. That is a shape that has one more vertex than the dimension of a space. So a triangle in 2D, a tetrahedron in 3D, etc.

AMEOBA

- Let's consider 2D for simplicity. We start with a triangle and one evaluates the function at the vertices.
- Then one performs a reflection, mirroring the triangle and evaluating the function at the new point. If the new value is lower then that becomes are new triangle. Then one rotates thought the points this way.
- This causes the polytope to crawl towards the minimum, hence the amoeba name.
- The polytope is also shrunk are expanded, this is like adaptive step sizes, so that you move a great distance when far from the minimum, but become more refined as you get closer.


Contour Plot with Nedler Mead Triangles
5
4
3
2
1
0
-1
Y
-1
0
1
2
3
4
5
X

GLOBAL SEARCH

- All the methods we have discussed find local minima. The easiest way to extend them to global searches is simply to start the process in many different locations.
- That is to perform a global search you should never start from only one location. You can use a grid of initial guesses or one can choose initial guesses randomly.
- One can also use methods like simulated annealing, where you push out of local minimum based on some criteria. When you are in a true global minima you will keep coming back to it regardless of how you are ejected from it.
- Another approach is to map the space to find the minimum. This has the advantage that you also get a map of the space.

GRID SEARCH

- One of the safest ways to look for a global minimum and also very inefficient is to simply evaluate the function on a grid. The minimum is then the lowest value on your grid.
- One can keep refining the grid until the minima changes by less than some value. This is guaranteed to give you a global minimum as long as the function doesn't vary wildly on scales less than your grid spacing.
- While this method is very inefficient it does have the advantage the you not only find the minimum but also map the space which may be useful if you are for example going to draw contour lines.

MCMC

- However, as we have already learned, a more efficient way to sample a space is to use a Monte-Carlo Markov-Chain. This way you sample the space, but choose the points proportionally to how informative they are. MCMC maps the region of interest with many fewer function calls then a uniform grid.
- MCMC is a very powerful method for sampling likelihoods because once the chain has adjusted the density of the sampling is the same as the likelihood of those parameters.
- So if you want to know the 90% confidence levels for you parameters, it is just the 90% of your Markov-Chain with the lowest values.

SIMULATED ANNEALING

- Simulated annealing uses something like the metropolis algorithm in looking for a global minimum.
- The important difference is that the ‘temperature’ - the probability that you will accept a less optimized solution - does not stay fixed, but gradually decreases as the search continues.
- Having a higher temperature in the beginning hopefully keeps one from finding local minima, but then cooling later on allows one to find a precise minima.
- The ‘cooling function’ determines the rate at which the search becomes more localized.

GENETIC OR EVOLUTIONARY ALGORITHMS

- The genetic algorithm is one type of evolutionary algorithm that tries to approach problems like living systems do.
- The genetic algorithm starts with a population, a group of solutions to the problem you are trying to solve.
- These solutions can be characterized by parameters like living creatures are characterized by their genes.
- In the genetic algorithm one judges the solutions by some fitness criteria and then only lets some number of fittest solutions survive.
- Then these solutions have their genes exchanged so form a new population and the process is repeated.
- In this way one evolves the parameters to get the optimized (fittest) solution.