10.4 Linear interpolation method

The next best thing one can do is the linear interpolation method, also known as the double false position method.

This method works similarly to the bisection method by shrinking the interval $[a, b]$, but instead of always dividing it in half, this method makes does a better estimate. First, it calculates the values $f(a)$ and $f(b)$. Then it interpolates the function f between these two values with a linear function. We can easily calculate the root of the linear function, let's call it c. We then use c as the new middle point to create two intervals $[a, c]$ and $[c, b]$. From there on we proceed the same way as in the bisection method. We calculate $f(c)$ and decide which interval is the interesting one. This method is slighlty faster at converging as we make use of the function evaluation $f(c)$ at the point c.

10.5 Newton's method

Newton's method is another iterative way of finding a root of a function $f(x)$. Contrary to the other methods we have talked about before, we have to be able to evaluate the derivate of f as well as the function f itself at any point. This is not always possible, depending on how complicated the function is. If no analytic function can be written down for the derivate, one can try to calculate the derivate numerically (e.g. using finite differences). However, evaluating the derivatives can be very expensive. On the positive side, if it is easy to calculate the derivative, then Newton's method is very fast and converges very quickly.

Let's spend a bit of time discussing how to numerically calculate a derivative. A common method is called *finite difference* which approximate the derivative $f'(x)$ by the expression

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}.$$

Which spacing h to choose is difficult to answer in general. The smaller it is the closer the estimate to the real value of the derivative. However, the h has to remain finite and if it gets too small, we might run into floating point issues. This is an important example of why it it important to understand the limitations of floating point numbers. The term $f(x+h)$ and $f(x)$ are almost identical if h is small. Thus, calculating the difference will result in a large floating point error.

The expression above is a one sided finite difference. One can also make a central difference:

$$f'(x) \approx \frac{f(x+0.5h) - f(x-0.5h)}{h},$$

which has often better properties. But let's go back to our root finding method.

In Newton's method we need one starting point of x, a guess. Let's call this x_0. During every iteration, we improve upon our guess using the formula

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

We continue iterating until we have achieved a precision that is good enough for our purpose. Often, people look at the change from x_n to x_{n+1} as a measure of how close we are to the real root.

There are several problems with Newton's method. We already mentioned that the function needs to be differentiable on the entire interval. Furthermore, it turns out that Newton's method might sometimes not converge at all if the first derivate is not *well* behaved. In such a case we might overshoot and never converge. One such example is the function

$$f(x) = |x|^{1/4}.$$

Another problem are stationary points of the function f (i.e. the derivative is zero). In that case we divide by zero and the method breaks down.

One of the most important uses of Newton's method is in optimization. Optimization refers to minimization or maximization of functions. We already know one example, the least square fit.

10.6 Multiple roots

In many cases, functions have multiple roots. For example

$$f(x) = x^2 - 1$$

has roots at $x_0 = 1$ and $x_1 = -1$. How can we find multiple roots? All the above methods give us only one root. Well, the short answer is that in general we might not be able to find all the root of a function. Especially if you think of a function like $\sin(x)$ which has an infinite number of roots.

However, there is a trick that allows us to find at least multiple roots in simple functions. Suppose the function you are trying to find the root is $f(x)$ and you have already found a root x_0. Then, look at the function

$$h(x) = \frac{f(x)}{x - x_0}$$

and find its roots. The figure below shows the function $f(x) = x^2 - 1$ in red. Suppose our first attempt at root finding resulted in the value $x_0 = 1$. Then, the function h is $h(x) = f(x)/(x - 1)$, which is plotted in blue. You can see that the function h has now only one root, namely the one we missed before at $x_1 = -1$.

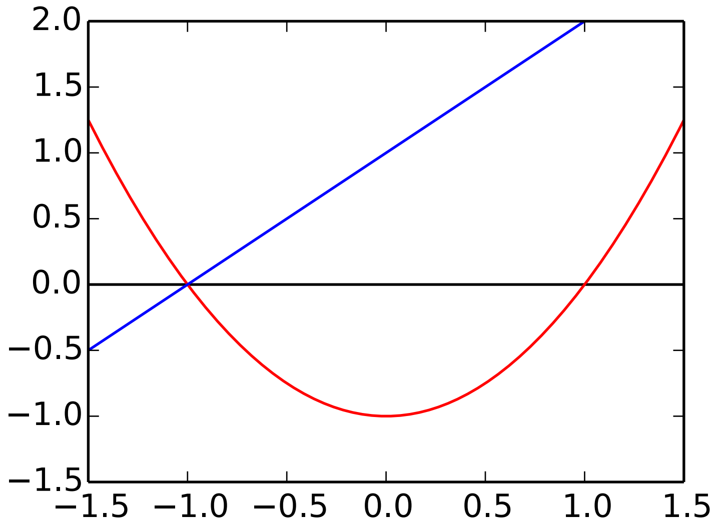

Figure 30: Finding multiple roots.

11 Markov Chain Monte Carlo

11.1 Likelihood function

You already learned how to interpolate data and how to fit a function. The method for fitting that we discussed only worked for linear problems. We discussed a few examples where we were able to rewrite a non-linear problem as a linear problem. But these were special cases and a conversion to an equivalent linear problem might not always be possible.

In such a case we could use a root finding algorithm to fit a function to data. We could define a function that depends on all the model parameters (usually that involves many dimensions) and tells us how good the fit is. This could be some of the squares as in the least square fit. Let's consider a generic case and call this function

$$\chi^2(\theta_1, \theta_2, ...) = \chi^2(\Theta)$$

The function χ^2 returns a scalar. The smaller, the better the fit. If it is exactly zero, then the fit is perfect. The θ_is are the parameters. If we were to fit a line, then one parameter would be the offset, the other parameter would be the gradient. In general there might be many parameters, which is why we just write them as one Θ. You can think of it as a vector of model parameters.

Next up, we'll need to construct a likelihood functions. We have one lecture to cover the this topic and therefore cannot go into great detail. This would need an introduction to a branch of statistics called Bayesian inference. Here, I'll just motivate it in words. Bear in mind tat there are some caveats that I'm not mentioning.

Suppose you are given a dataset D and a set of model parameters Θ. The dataset can just be a collection of points (x_i, y_i) as before. But it can also be more complicated, for example it could involve three measurements per datapoint and include measurement uncertainties: $(x_i, y_i, z_i, \sigma_i)$. Then the χ function has a minimum at Θ_{min} when the error is smallest. It means that this value of Θ is the most likely. This leads us to define a function that describes how likely it is to observe a dataset D under the assumption of model parameters Θ. We call this the likelihood function. It may not be trivial to construct a likelihood function. We'll skip over the details here and simply define it as an exponential of the function χ^2

$$P(D|\Theta) = \exp(-\chi^2)$$

Note that $P(D|\Theta)$ is largest (high likelihood) if χ^2 is smallest (small errors). The name likelihood function makes sense but so far, we haven't really gained much - now we need to maximize a function instead of minimizing one.

However, the simple framework that we have developed let's us now introduce the idea of Markov Chain Monte Carlo (MCMC) and the Metropolis-Hastings algorithm. Before we start, let us reiterate our goal. We want to find the set of parameters Θ that best fits the data D. Ideally, we'd also like to evaluate how good the best model is, not only which model is the best.

11.2 Random numbers

MCMC works by drawing lots of random numbers. So let's spend a second on discussing random numbers on a computer. This may seem like a trivial as throwing a dice, but it turns out it is actually a really hard task for a computer. The problem is that computers are deterministic. They can only do exactly what we tell them to do. And in most cases that's a good thing. But if we ask the computer to give us any number between 0 and 1, it doesn't know what to do.

All modern computer languages have built-in random number generators. They do, what the name suggests, generate random numbers. Most importantly, the generator is supposed to create random numbers that are uncorrelated. That means two things. First, if we run a program which uses random numbers twice, it should give us different results. Random number generators usually use the current time on a computer and the process id to initialize. Second, consecutive random numbers in one program should be independent and uncorrelated.

In python, the most convenient way to draw a random number is using the `numpy` package. There are two kind of random numbers that we need. The first kind is a uniformly distributed floating point number between 0 and 1. The following python code does exactly that:

```python
import numpy
x = numpy.random.rand()
print x
```

Code 56: Drawing a random number with a uniform distribution in the interval $[0:1)$.

The second kind is a random floating point number with a Gaussian or normal distribution. If a random number is drawn from a non-uniform distribution, some numbers are more likely than others. In the case of a Gaussian distribution, the numbers are distributed according to a bell-shaped curve:

$$f(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$$

Thus, it is most likely to get a number close to zero, and it gets less likely to get large numbers. The probability to get a number as large as one million is extremely small but finite. The following python code generates a random number with a Gaussian distribution

```
import numpy
x = numpy.random.randn()
print x
```

Code 57: Drawing a random number with a Gaussian distribution in the interval [0 : 1).

11.3 The Metropolis-Hastings Algorithm

Let's now combine all the ingredients and cook up a formula for finding the best model parameters Θ for our dataset D.

As a test case, let's assume we only have two data-points $(x_0, y_0), (x_1, y_1)$ and we want to fit the model

$$y(x) = m$$

to it with m being a constant. Thus, our Θ vector has one element, m. The function error function χ^2 is then

$$\chi^2(\Theta) = \sum_{i=0}^{1} (m - y_i)^2$$

and thus we have the likelihood function

$$P(D|\Theta) = \exp\left(-\sum_{i=0}^{1} (m - y_i)^2\right)$$

The first step in the Metropolis Hastings algorithm is to make a guess of Θ. The better the guess the faster we will converge onto the real solution. In most cases you can use prior knowledge to come up with a good guess. You can even describe this prior knowledge in a mathematical way, using *a prior probability distribution*. How to choose a good prior would go beyond the scope of this lecture and we will just make a really bad, uninformed guess. For example $\Theta_0 = 0$.

We then calculate χ^2 and P for Θ_0. We now have a number for the likelihood of our parameter. Next up, we propose a new guess for the next Θ, Θ_p. We chose values for Θ_p such that they are correlated with Θ_0. That means that we will most likely only make a small change to all parameters. One of the caveats that we won't discuss is how to best come up with the new guess. We just add a random number to the current values. And we choose normally distributed numbers. Thus

$$\Theta_p = \Theta_0 + X$$

where X is a normally distributed random number with mean 0 and variance 1. For the new Θ_p we can again calculate the χ^2 and the P functions.

We have so far only proposed the new values, not yet accepted them. If we accept them, depends on the ratios of the likelihood

$$r = \frac{P(\Theta_p|D)}{P(\Theta_d|D)}$$

If r is bigger than 1, then the new values are more likely than the old ones. In that case we accept the proposed value Θ_p and add it to our chain.

What if r is less than 1? Well, in that case we draw another random number, Y. This time, we draw the number form a uniform distribution in the interval 0 to 1. We then compare the value of r to the variable Y. If $r > Y$ we accept the new Θ_p despite the fact that the old values Θ_0 were actually better (lower χ^2) and is less likely (smaller P). Only if $r < Y$, we throw away our proposed new value of Θ and start over again by drawing another set of Θ_p parameters.

Let's write this up in a python program and see what happens. The code is listed below. A few things are worth pointing out. In this simple example, we actually don't need the x values of the data because our model $y(x) = m$ is independent of x. Note that we do not take the exponential in the likelihood function, but only after we took the ratio. This is because the likelihoods are often very small numbers, below floating point precision. It is better to work with the log-likelihood than the likelihood itself. This doesn't change the math.

```
import math
import numpy

# Data (hard coded)
Dx=[1.0,2.0]    # We actually don't need the x values
Dy=[30.0,40.0]

# Model, horizontal line, y(x)=m
def y(x,theta):
    return theta[0]

# Chi Squared
def chi2(Dx, Dy, theta):
    s = 0.
    for i in xrange(len(Dx)):
        s += (y(Dx[i],theta)-Dy[i])**2
    return s/len(Dx)

# Likelihood function
def P(Dx,Dy,theta):
    return (-chi2(Dx,Dy,theta))

# Initial guess for model parameters
theta_current = [0.]
P_current = P(Dx,Dy,theta_current)

chain = [] # Array to save the MCMC chain

# Do 5000 generations
for i in xrange(5000):
    # Randomly draw new proposed theta
    theta_proposed = [theta+0.1*numpy.random.randn()
                            for theta in theta_current]
    P_proposed = P(Dx,Dy,theta_proposed)

    # Calculate likelihood ratio
    ratio = math.exp(P_proposed - P_current)

    # Decide if to accept the new theta values
    r = numpy.random.rand()
    if ratio>r:
        theta_current = theta_proposed
        P_current = P_proposed

    # Save current theta value in chain
    chain.append(theta_current[0])

# Plot the result
import matplotlib
matplotlib.use('pdf')
import matplotlib.pyplot as plt
plt.plot(chain)
plt.savefig("chain.pdf")
```

Code 58: Metropolis-Hastings algorithm to fit a horizontal line to two data-points.

The result of the MCMC run can be best illustrated by plotting the parameter(s) Θ as a function of the generation. This is shown in the figure below.

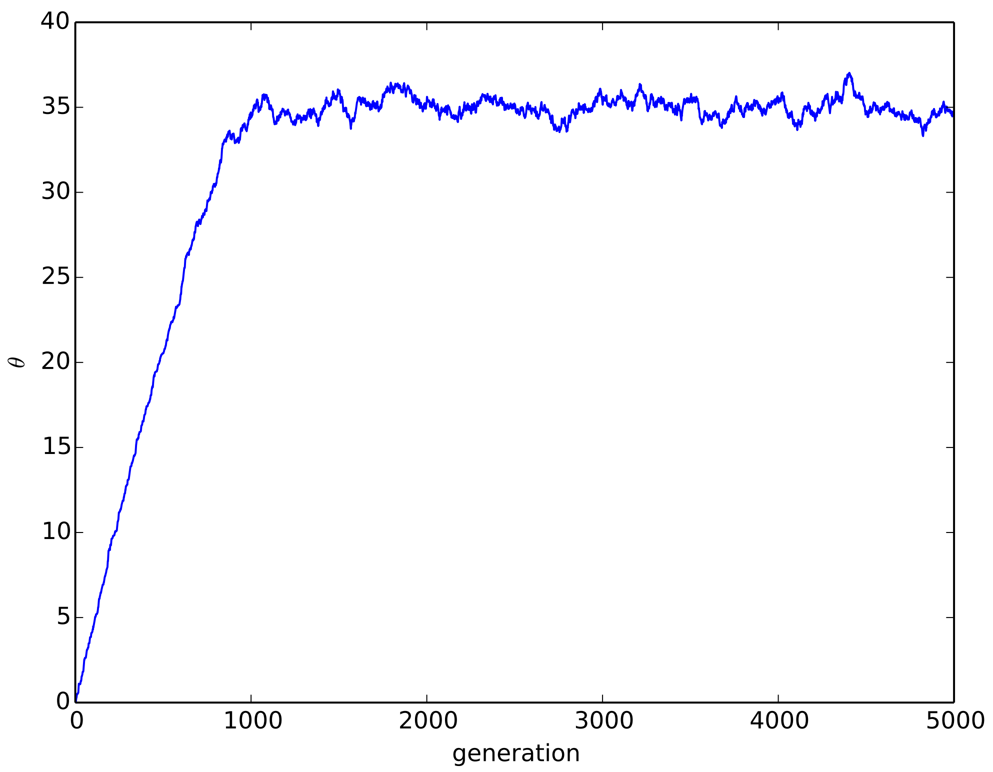

Figure 31: Θ as a function of generation in our MCMC run.

At the beginning the chain moves quickly towards to best fit value, $m = 35$. This phase is called burn-in. Then, rather than converging on the best fit value, it starts to oscillate around it. This is a random walk and it will in fact never converge. However, notice that the chain does not depart from the best fit. If we were to take an average over many generations after the burn-in phase, then we would get a very good estimate of the best fit value.

Not only would we get a best fit value, but we would also get an estimate of how good the fit is. This is given by the spread of the random walk. You can see from the plot that we reach values roughly from 33 to 37. As it turns out this can be used as a confidence interval of the best fit value.

$$m_{\text{best}} = 35 \pm 2$$

We only did the analysis qualitatively, to do it quantitatively, we'd need to write down the proper definition of the prior and make a better choice for our proposed step in the Markov chain. Nevertheless, our method worked. If you've never seen an MCMC before, then this should be at surprising. We never calculate a gradient of a function. We also just chose an arbitrary initial condition. Everything else was done by throwing a dice and comparing two numbers.

Let's do one more example, this time fitting a slightly more complex function:

$$y(x) = a + b \cdot \sin\left(\frac{2\pi}{24}x + c\right)$$

We use the a dataset that we used before, the temperature measured over a day at UTSC as a function of time. This is a fit that we were not able to do with the linear least square fit.

Below is the python code to do this fit using MCMC. It's very similar to the previous code. All we had to change is the model function and the number of parameters. The algorithm is exactly the same.

```
import math
import numpy

# Read in data
Dx=[]
Dy=[]
for line in open("temp.txt"):
        rows = line.split(" ")
        Dx.append(float(rows[0]))
        Dy.append(float(rows[1]))

# Model
def y(x,theta):
    return theta[0]+theta[1]*math.sin(2.*math.pi/24.*x+theta[2])

# Chi Squared
def chi2(Dx, Dy, theta):
    s = 0.
    for i in xrange(len(Dx)):
        s += (y(Dx[i],theta)-Dy[i])**2
    return s/len(Dx)

# Likelihood function
def P(Dx,Dy,theta):
    return -chi2(Dx,Dy,theta)

# Initial guess for model parameters
theta_current = [0.,0.,0.]
P_current = P(Dx,Dy,theta_current)

chain = []
for i in xrange(10000):
    theta_proposed = [theta+0.1*numpy.random.randn()
                                for theta in theta_current]
    P_proposed = P(Dx,Dy,theta_proposed)

    ratio = math.exp(P_proposed-P_current)
    r = numpy.random.rand()

    if ratio>r:
        theta_current = theta_proposed
        P_current = P_proposed
    if i>=5000: # save chain only after burn-in
        chain.append(theta_current)
```

```python
# Calculate average:
theta_avg = [0.,0.,0.]
for theta in chain:
    for i in xrange(3):
        theta_avg[i] += theta[i]
for i in xrange(3):
    theta_avg[i] /= len(chain)


# Calculate model y
My = []
for x in Dx:
    My.append(y(x,theta_avg))

import matplotlib
matplotlib.use("pdf")
import matplotlib.pyplot as plt
plt.plot(Dx, Dy, "ro")
plt.plot(Dx, My, ".")
plt.savefig("fit.pdf")
```

Code 59: Metropolis-Hastings algorithm to fit a sin curve to the UTSC temperature dataset.

The above code also includes the plotting function call to produce this plot

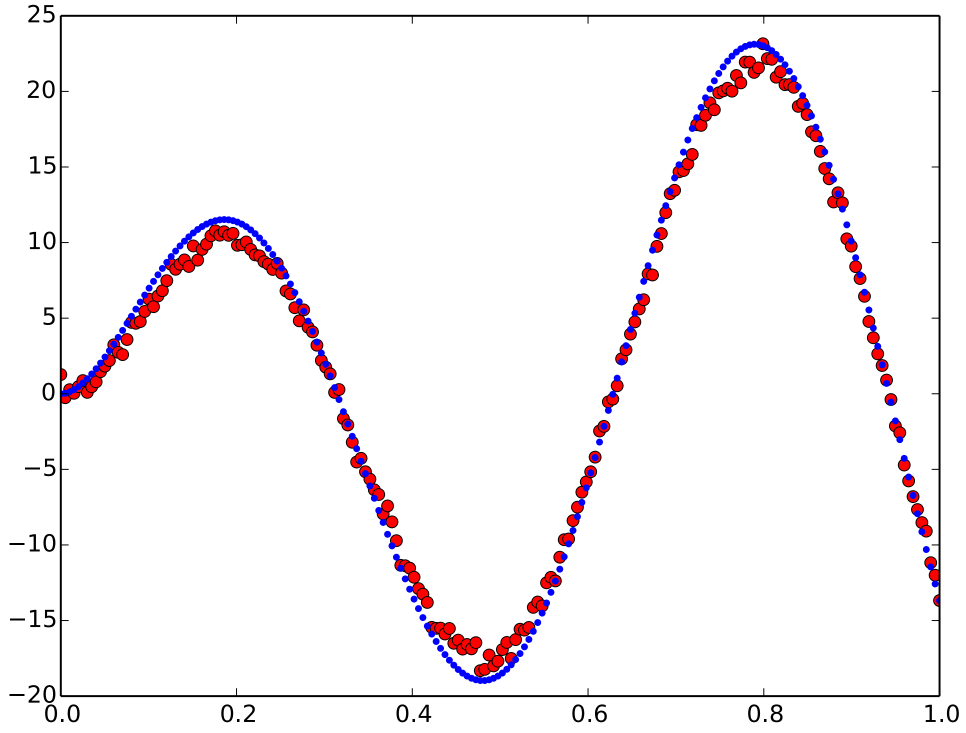

Figure 32: MCMC best fit to the temperature data.

There are a lot of things we couldn't cover in this one lecture on MCMC. It's an interesting subject if you're interested in further reading. And most excitingly, it's a subject still rapidly evolving.

End of lecture 8.