Introduction to Scientific Computing
Lecture 3

Professor Hanno Rein

Last updated: September 25, 2017

Linear equation systems

In the following sections, we will spend some time to solve linear systems of equations. This is a tool that will come in handy in many different places during this course. For some problems, there are specialized algorithms available. For example, matrices can be diagonal, sparse or tri-diagonal. In these cases, optimized algorithms can speed up the calculation dramatically. In our course, we discuss a general purpose algorithm that can be used to do many tasks related to matrices. The most important task is to solve the system of linear equations

$$A \cdot x = b$$

for x. Here, A is a $N \times N$ matrix and x and b are vectors of lengths N. You might have heard of at least one way to solve this system in a linear algebra course: Gaussian elimination. This would also work on a computer. However, we can do even better. We'll implement what is called the LU-decomposition with the Crout algorithm. But before we go into the algorithm itself, let's start with an real world problem where linear equation systems appear: fitting.

0.1 Linear least square fit

One task where one needs to solve a linear set of equations is a least square fit. There is in general no closed solution for a non-linear least square fit and it requires iteration. In this section we will keep it simple and only discuss the linear version of a least square fit.

Suppose we are given a set of 500 temperature measurements. The data was taken here at the UTSC weather station over several months. The figure below shows the temperature as a function of time of day. One can clearly see a trend, temperatures are warmest around 3pm on an average day. The above data is given to us as a set of x and y values with N pairs.

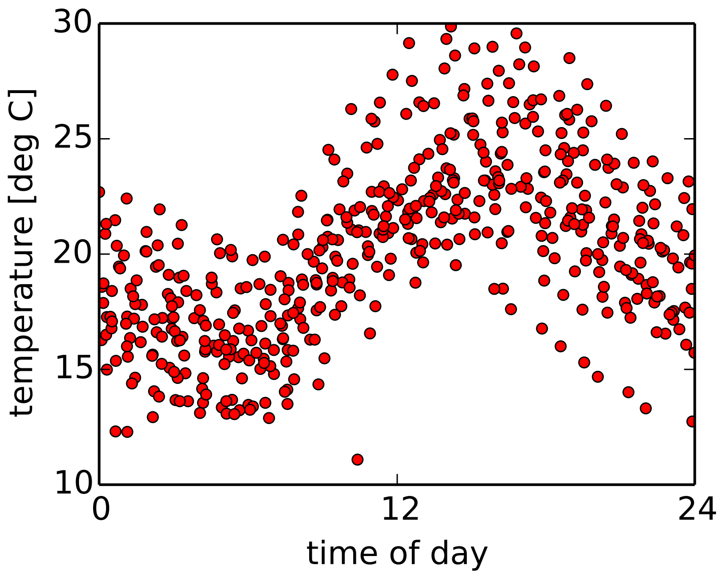

Figure 1: Temperature measurements over several month as a function of time of day. Source: UTSC weather station, `http://weather.utsc.utoronto.ca`.

Let us assume the data is described by a function of the following form

$$f(t) = a_0 + a_1 \sin\left(\frac{t}{24} 2\pi\right) + a_2 \cos\left(\frac{t}{24} 2\pi\right)$$

We use one hour as the unit of time. Also note that the sin and cos functions expect an argument in radians, not degrees. This is a convention that we'll use for the remainder of the course. Finally, note that we use two sin functions instead of one. We really only want one sin function, but we don't know the phase. We can add a phase argument to a sine function, but then the problem would not be linear anymore. Instead, we use two sine functions and the identity

$$\sin(\alpha+\beta) = \sin\alpha\cos\beta + \cos\alpha\sin\beta,$$

which allows us to convert one sin function with a phase to two sin and cos functions without a phase.

If we had as many unknowns as equations (we have 3 and 500 respectively) we could just write down the equation system as

$$\begin{aligned}
y_0 &= f(x_0) = & a_0 + a_1 \sin\left(\frac{x_0}{24} 2\pi\right) + a_2 \cos\left(\frac{x_0}{24} 2\pi\right) \\
y_1 &= f(x_1) = & a_0 + a_1 \sin\left(\frac{x_1}{24} 2\pi\right) + a_2 \cos\left(\frac{x_1}{24} 2\pi\right) \\
y_2 &= f(x_2) = & a_0 + a_1 \sin\left(\frac{x_2}{24} 2\pi\right) + a_2 \cos\left(\frac{x_2}{24} 2\pi\right)
\end{aligned}$$

or in matrix form

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} 1 & \sin\left(\frac{x_0}{24}2\pi\right) & \cos\left(\frac{x_0}{24}2\pi\right) \\ 1 & \sin\left(\frac{x_1}{24}2\pi\right) & \cos\left(\frac{x_1}{24}2\pi\right) \\ 1 & \sin\left(\frac{x_2}{24}2\pi\right) & \cos\left(\frac{x_2}{24}2\pi\right) \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \end{pmatrix}.$$

These equations have exactly one solution, but only if we are given exactly 3 data value pairs for 3 unkowns. Most of the times, we have a few unknown parameters (in our case a_0, a_1 and a_2) but are given many more, let's say N, data value pairs. Then, we have an over-defined system. We solve this by performing a *fit*. We want to minimize the average (vertical) distance of any point (x_i, y_i) from our function f. One way to do this is a least square fit.

In matrix and vector notation, this relates to

$$\begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{N-1} \end{pmatrix} = \underbrace{\begin{pmatrix} 1 & \sin\left(\frac{x_0}{24}2\pi\right) & \cos\left(\frac{x_0}{24}2\pi\right) \\ 1 & \sin\left(\frac{x_1}{24}2\pi\right) & \cos\left(\frac{x_1}{24}2\pi\right) \\ & \vdots & \\ 1 & \sin\left(\frac{x_{N-1}}{24}2\pi\right) & \cos\left(\frac{x_{N-1}}{24}2\pi\right) \end{pmatrix}}_{\equiv C} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \end{pmatrix} + \begin{pmatrix} e_0 \\ e_1 \\ \vdots \\ e_{N-1} \end{pmatrix}$$

The values e are errors or residuals. This is what we want to minimize. If we have a perfect fit, $e_i = 0$ for all i. Let us define the total residual as

$$S = \sum_{i=0}^{N-1} e_i^2$$

and switch to a slightly more general case: instead of assuming exactly 3 free parameters, we now assume we have m free parameters. When S is minimized, its gradient vector is zero. The derivatives of S can be easily calculated (we replace e_i by the values from the above matrix equation) to get

$$\begin{aligned} 0 &= \frac{\partial S}{\partial a_j} = 2\sum_{i=0}^{N-1} e_i \frac{\partial e_i}{\partial a_j} & \forall j \\ &= 2\sum_{i=0}^{N-1}\left[y_i - \sum_{k=0}^{m-1}(a_k C_{ik})\right] C_{ij} \end{aligned}$$

Rearranging gives

$$\sum_{i=0}^{N-1} C_{ij} y_i = \sum_{i=0}^{N-1}\sum_{k=0}^{m-1} C_{ik} C_{ij} a_k$$

In matrix notation this is

$$\underbrace{C^T \cdot y}_{b} = \underbrace{\left(C^T C\right)}_{A} \cdot a$$

We have now a linear equation system. We want to solve it for a, our coefficient a_0, a_1 and a_2 in the above example. Note that calculating the left side is trivial (it is just a matrix-vector multiplication). But we need to invert $C^T C$ in order to get a. This is what the LU decomposition will do for us.

Before me move on, let's have a look at the dimensions of the vectors and matrices. The matrix C has dimensions of $N \times m$, the vector y is N elements long. Thus the left hand side (and therefore the right hand side) is a vector of length m. The matrix $C^T C$ that we need to invert is only a $m \times m$ matrix. So its size is only determined by the number of free parameters, not the number of data-points we want to fit. Usually $m \ll N$. This is important because the LU decomposition will be slow for large matricies as it scales as $O(m^3)$.

0.2 The LU decomposition algorithm

In this section, we derive the LU decomposition algorithm to solve a general matrix equation of the form

$$A \cdot x = b$$

Suppose we could decompose the matrix A into two triangular matrices L and U such that

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & \\ a_{10} & a_{11} & a_{12} & \dots \\ a_{20} & a_{21} & a_{22} & \\ & \vdots & & \ddots \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & \\ l_{10} & 1 & 0 & \dots \\ l_{20} & l_{21} & 1 & \\ & \vdots & & \ddots \end{pmatrix} \cdot \begin{pmatrix} u_{00} & u_{01} & u_{02} & \\ 0 & u_{11} & u_{12} & \dots \\ 0 & 0 & u_{22} & \\ & \vdots & & \ddots \end{pmatrix}.$$

(Note that we have chosen $l_{ii} = 1$ for $i = 0, \dots N-1$. This is arbitrary, but works nicely in practice and leads us to the Crout algorithm.) Then we could easily solve

$$L \cdot y = b$$

by a simple back-substitution. Then, we can solve

$$U \cdot x = y$$

again, with a simple back-substitution.

So how can we get the matrix elements u_{ij} and l_{ij} from the matrix elements a_{ij}? We can write the above matrix equation in terms of the elements

$$a_{ij} = l_{i0} \cdot u_{0j} + l_{i1} \cdot u_{1j} + \dots$$

This looks like we didn't really gain anything. We now have to solve a new set of N^2 equations to solve for u_{ij} and l_{ij}. However, it turns out it's easier than you might think. Let's look at the equation for different cases. If $i < j$, then we have

$$a_{ij} = u_{ij} + \sum_{k=0}^{i-1} l_{ik} \cdot u_{kj}.$$

Or, bringing the sum on the other side, this is equivalent to

$$u_{ij} = a_{ij} - \sum_{k=0}^{i-1} l_{ik} \cdot u_{kj}.$$

For $i > j$, we have

$$a_{ij} = l_{ij} u_{jj} + \sum_{k=0}^{j-1} l_{ik} \cdot u_{kj}.$$

Again, we can rewrite this slightly to get

$$l_{ij} = \frac{a_{ij} - \sum_{k=0}^{j-1} l_{ik} \cdot u_{kj}}{u_{jj}}.$$

In our algorithm to compute L and U from A we'll loop over the rows. This means the outer loop will be j. The equations from above are:

$$\begin{aligned} u_{ij} &= a_{ij} - \sum_{k=0}^{i-1} l_{ik} \cdot u_{kj}. \\ l_{ij} &= \frac{a_{ij} - \sum_{k=0}^{j-1} l_{ik} \cdot u_{kj}}{u_{jj}}. \end{aligned}$$

When you go through the first few step, you can convince yourself that the left hand side is completely determined by previously calculated values of l_{ij} and u_{ik} (a_{ij} is known anyway). This procedure is called Crout's algorithm.

There is one further nice feature of this algorithm. We only need the elements a_{ij} once. That means we can override the element afterwards and save some memory. This is also called an *in place* LU decomposition. However, note that this destroys the original matrix.

So, are we done? Well, there is one very obvious issue with the algorithm. What happens if u_{jj} is 0? The division by zero would break our scheme. But we can fix it. The way we're fixing it is called *pivoting*. This works the same way as in a Gaussian elimination scheme: We simply swap two rows and always choose the one we're dividing by to be the element that is the largest. Code 1 implements the in-place partial pivoting LU decomposition

```python
def swap_rows(a,i1,i2):
    for k in xrange(len(a)):
        temp = a[i1][k]
        a[i1][k] = a[i2][k]
        a[i2][k] = temp

def lu_decomposition(a, indx):
    n = len(a)
    for j in xrange(n):
        for i in xrange(j):
            s = a[i][j]
            for k in xrange(i):
                s -= a[i][k]*a[k][j]
            a[i][j] = s
        big = 0.
        for i in xrange(j,n):
            s = a[i][j]
            for k in xrange(j):
                s -= a[i][k]*a[k][j]
            a[i][j] = s
            temp = math.fabs(s)
            if temp >= big:
                big = temp
                imax = i
        if j != imax:
            swap_rows(a,j,imax)
        indx[j] = imax
        d = 1./a[j][j]
        for i in xrange(j+1,n):
            a[i][j] *= d
```

Code 1: LU decomposition with partial pivoting and in-place storage. The helper function `swap_rows` simply swaps, as the name suggests, two rows in a matrix. Here, `a` is a square $n \times n$matrix and `indx` is a vector of length n where we store the permutations needed for pivoting.

From here on, we only need to solve the triangular matrices using a simple substitution. Note that we can do that as often as we like, we only need to do the LU decomposition once. This is one reason that makes this method so interesting.

As mentioned before, we first solve

$$L \cdot y = b$$

then

$$U \cdot x = y.$$

The only thing worth mentioning is that we need to make sure we undo the row swapping that we did in the LU decomposition for pivoting. That's why we saved every swap in the `indx` array. The code to do solve the substitutions is shown in Code 2. Note that the original vector b will be overwritten with the result (x).

```
def lu_substitute(a, indx, b):
    n = len(a)
    for i in xrange(n):
        imax = indx[i]
        s = b[imax]
        b[imax] = b[i]
        for j in xrange(i):
            s -= a[i][j]*b[j]
        b[i] = s
    for i in reversed(xrange(n)):
        for j in xrange(i+1,n):
            b[i] -= a[i][j]*b[j]
        b[i] = b[i]/a[i][i]
```

Code 2: Substitution for the LU decomposition. Expects the L and U matrices in the form produced by the `lu_decomposition routing`. The array `indx` must contain any row permutations performed in the LU decomposition.