Lecture 9

- Metropolis algorithm (continued)
- Linear systems
- Gauss-Jordan elimination, pivoting.

The Metropolis algorithm (Metropolis, Rosembluth[2], Teller[2] (1953))

If the dimensionality involved is high, all previous methods become quite impractical.

Supose you wish to generate points **X** with probability density $w(\mathbf{X})$.

This algorithm proposes to generate such points by taking a random walk through the space of **X**'s in such a way that when the walk is a long one the points visited are sprinkled with the desired probability.

The random walk proceeds in the following way. Suppose you have taken n steps and are sitting at X_n. To go to $X_{n+1}$the walker takes a random trial step (for instance by choosing points randomly in a cube of small size). The trial step is then accepted or rejected according to the ratio $r=w(X_t)/w(X_n)$.

If the ratio is larger than one the step is accepted. Otherwise it is accepted with probability r.

This latter step is easily accomplished by generating a random number η in [0,1] and accepting the step if η<r.

To prove that the algorithm does indeed generate a sequence of points distributed according to w, let us consider a number of walkers starting from different initial points and moving independently.

If $N_n(\mathbf{X})$ is the density of walkers at **X** after n steps then the net number of walkers moving from **X** to **Y** in the next step will be,

$$\Delta N(X) = N_n(X)P(X \to Y) - N_n(Y)P(Y \to X)$$

The equilibrium densities are therefore related by,

$$\frac{N_e(X)}{N_e(Y)} = \frac{N_n(X)}{N_n(Y)} = \frac{P(Y \to X)}{P(X \to Y)}$$

And the system is **stable** (meaning it returns to equilibrium if perturbed).

It remains to be proved that the distribution one achieves in equilibrium is the desired one.

The probability of taking a step from X to Y was,

$$P(X \to Y) = T(X \to Y)A(X \to Y)$$

Where T is the probability of taking a trial step from X to Y and A is the probability of accepting it.

If Y can be reached from X in a single step, and the walker is moving isotropically then,

$$T(X \to Y) = T(Y \to X)$$

So the equilibrium distribution satisfies,

$$\frac{N_e(X)}{N_e(Y)} = \frac{A(Y \to X)}{A(X \to Y)}$$

Now, if $w(X) > w(Y)$ then $\mathrm{A}(\mathrm{Y} \to X) = 1$ and $A(X \to Y) = \dfrac{w(Y)}{w(X)}$

And, if $w(Y) > w(X)$ then $\mathrm{A}(\mathrm{X} \to Y) = 1$ and $A(Y \to X) = \dfrac{w(X)}{w(Y)}$

Therefore in either case,

$$\frac{N_e(X)}{N_e(Y)} = \frac{A(Y \to X)}{A(X \to Y)} = \frac{w(X)}{w(Y)}$$

So indeed the equilibrium densities are distributed with the desired probability weights.

-Any probability rule for the stepsize will do. For instance T(X->Y)=w(Y) would be optimal (but of course, hard to do).

-The art of choosing the rule consists in minimizing the number of lost steps.

A problem of this algorithm is that although the points generated are sprinkled with the desired probability if the walk is a long one, each point is statistically correlated with the next one. So for instance, if one wished to use this method to compute an integral using the Monte Carlo method, the variance estimate of the error we introduced would be incorrect, since it assumes points are not correlated with each other.

A way around this problem is to sample the integral using points generated in the random walk sampled at a fixed interval of steps large enough that the resulting points have very little correlation. Of course, this is where things become more an art than science. A similar remark corresponds to the choice of starting point of the random walk. A lousy choice will yield a distribution that is only vaguely related to the desired one.

Example: the Ising model

A system of spins living in a lattice, each spin can be up or down and the interaction Hamiltonian is given by,

$$H = -J \sum_{<i,j>} S_i S_j - B \sum_i S_i$$

Nearest neighbor

One wishes to consider the canonical ensemble

$$w(S) = \frac{e^{-H(S)}}{Z} \qquad Z(J,B) = \sum_S e^{-H(S)}$$

And compute quantities like the energy, $E = \sum_S w(S) H(S)$

The problem lies in the sums. For a 16x16 lattice, one has 2^{256} or about 10^{77} configurations.

So one way to handle the problem is to evaluate the sums using Monte Carlo. To generate the configurations, one takes "steps" by flipping one spin in the configuration and "sweeping" across it until the step is accepted.

The acceptance depends on the probability

$$r = \frac{w(S_t)}{w(S)} = e^{-H(S_t)+H(S)}$$

Because of the nearest-neighbor form of the Hamiltonian this can be explicitly written as ,

$$r = e^{-2S_a(Jf+B)} \qquad f = S_{i+1,j} + S_{i-1,j} + S_{i,j+1} + S_{i,j-1}$$

And since f can only take the values $0, \pm 2, \pm 4$,
only 10 different values of r can arise (there are two values of S_a)
so one can tabulate them and repeatedly use them in a very efficient manner.

Linear systems:

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$$

$$\begin{aligned}
a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1N}x_N &= b_1 \\
a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \cdots + a_{2N}x_N &= b_2 \\
a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + \cdots + a_{3N}x_N &= b_3 \\
\cdots & \quad \cdots \\
a_{M1}x_1 + a_{M2}x_2 + a_{M3}x_3 + \cdots + a_{MN}x_N &= b_M
\end{aligned}$$

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1N} \\ a_{21} & a_{22} & \cdots & a_{2N} \\ & \cdots & & \\ a_{M1} & a_{M2} & \cdots & a_{MN} \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \cdots \\ b_M \end{bmatrix}$$

If M=N and the matrix A is non-singular, the system has a solution. Most solution methods are algebraic in nature. They can only fail due to roundoff error. For instance, if your system is close to singular. Another instance is if the system is large (N=100 or more) since the number of operations involved is large and so is the accumulation of roundoff error. The two issues compound each other!

Rules of thumb:

For “normal” systems, N up to 50 with single precision, N up to a few thousand with double precision (mostly limited by computer time). For more than a few thousand equations, something special has to be happening (sparse systems).

For systems that are close to singular, one may need a lot of work for N=5!

Detail: computers store your arrays as vectors, either in successive columns (Fortran) or rows (C, Pascal). In this representation, there is a huge difference between representing, say, a 4x4 array as such or as a 10x10 array with many zeros in it. Whenever you pass your array to a subroutine, this has to be properly communicated. A rather large number of programming bugs are generated by such a step!

Packages: there are out there highly developed packages for linear algebra. LINPACK, LAPACK, IMSL, NAG are some of them.

Column augmented matrices:

Many times one wishes to solve a system of equations for different right hand-sides and also one wishes the explicit inverse of the matrix of coefficients. Then it is good to consider the augmented matrix:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \cdot \left[\begin{pmatrix} x_{11} \\ x_{21} \\ x_{31} \\ x_{41} \end{pmatrix} \sqcup \begin{pmatrix} x_{12} \\ x_{22} \\ x_{32} \\ x_{42} \end{pmatrix} \sqcup \begin{pmatrix} x_{13} \\ x_{23} \\ x_{33} \\ x_{43} \end{pmatrix} \sqcup \begin{pmatrix} y_{11} & y_{12} & y_{13} & y_{14} \\ y_{21} & y_{22} & y_{23} & y_{24} \\ y_{31} & y_{32} & y_{33} & y_{34} \\ y_{41} & y_{42} & y_{43} & y_{44} \end{pmatrix} \right]$$

$$= \left[\begin{pmatrix} b_{11} \\ b_{21} \\ b_{31} \\ b_{41} \end{pmatrix} \sqcup \begin{pmatrix} b_{12} \\ b_{22} \\ b_{32} \\ b_{42} \end{pmatrix} \sqcup \begin{pmatrix} b_{13} \\ b_{23} \\ b_{33} \\ b_{43} \end{pmatrix} \sqcup \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \right] \qquad (2.1.1)$$

The solution of the system is the same if one substitutes rows by linear combinations of them or exchanges rows.

Interchanging columns of a only can be done if one interchanges the appropriate rows of x and y.

Gauss-Jordan elimination:

- Divide the first row by a_{11}
- Add the first row (times an appropriate factor) to all the other rows in such a way that you make $a_{i1}=0$.
- Divide the second row by a_{22}
- Add the second row (times an appropriate factor) to all rows in such a way that you make $a_{i2}=0$
- At the end of the day, you have converted a to the identity matrix and you will have the solution of the system at your disposal in the b vectors.

The method is guaranteed to work. The only problem that can arise is if one of the a_{ii}'s we are dividing by vanishes (or, numerically, if it is very small). To fix this problem a technique named **pivoting** is used.

Pivoting:

Consists in interchanging rows and columns in such a way that the a_{ii} one needs ("pivot") is as large as possible. The exchanges should follow the rules we discussed, and should only involve the elements that are to the bottom and right of a_{ii}.

Since exchanging rows is harder, sometimes it is avoided. This is called "partial pivoting".

Another issue is that pivoting depends on scaling. If we multiply the fourth equation times a million, it is very likely to contribute a pivot, yet the system is unchanged. Some routines therefore choose pivots by rescaling the equations in such a way that the largest coefficient in each is unity.

```
      subroutine gaussj(a,n,np,b,m,mp)
      parameter (nmax=50)
      dimension a(np,np),b(np,mp),ipiv(nmax)
      dimension indxr(nmax),indxc(nmax)
      do 11 j=1,n
        ipiv(j)=0
11    continue
      do 22 i=1,n
        big=0.
        do 13 j=1,n
          if(ipiv(j).ne.1)then
            do 12 k=1,n
              if (ipiv(k).eq.0) then
                if (abs(a(j,k)).ge.big)then
                  big=abs(a(j,k))
                  irow=j
                  icol=k
                endif
              else if (ipiv(k).gt.1) then
                pause 'singular matrix'
              endif
12          continue
          endif
13      continue
        ipiv(icol)=ipiv(icol)+1
        if (irow.ne.icol) then
          do 14 l=1,n
            dum=a(irow,l)
            a(irow,l)=a(icol,l)
            a(icol,l)=dum
14        continue
          do 15 l=1,m
            dum=b(irow,l)
            b(irow,l)=b(icol,l)
            b(icol,l)=dum
15        continue
        endif
        indxr(i)=irow
        indxc(i)=icol
        if (a(icol,icol).eq.0.) pause 'singular matrix.'
        pivinv=1./a(icol,icol)
        a(icol,icol)=1.
        do 16 l=1,n
          a(icol,l)=a(icol,l)*pivinv
16      continue
        do 17 l=1,m
          b(icol,l)=b(icol,l)*pivinv
17      continue
        do 21 ll=1,n
          if(ll.ne.icol)then
            dum=a(ll,icol)
            a(ll,icol)=0.
            do 18 l=1,n
              a(ll,l)=a(ll,l)-a(icol,l)*dum
18          continue
            do 19 l=1,m
              b(ll,l)=b(ll,l)-b(icol,l)*dum
19          continue
          endif
21      continue
22    continue
      do 24 l=n,1,-1
        if(indxr(l).ne.indxc(l))then
          do 23 k=1,n
            dum=a(k,indxr(l))
            a(k,indxr(l))=a(k,indxc(l))
            a(k,indxc(l))=dum
23        continue
        endif
24    continue
      return
      end
```

Find Pivot

Do pivoting

Divide by pivot

Gauss Jordan Elim

Gaussj from Numerical Recipes

LU Decomposition:

Suppose we can decompose the coefficients matrix as $\mathbf{L} \cdot \mathbf{U} = \mathbf{A}$

Where L is a lower triangular and U an upper triangular matrix,

$$\begin{bmatrix} \alpha_{11} & 0 & 0 & 0 \\ \alpha_{21} & \alpha_{22} & 0 & 0 \\ \alpha_{31} & \alpha_{32} & \alpha_{33} & 0 \\ \alpha_{41} & \alpha_{42} & \alpha_{43} & \alpha_{44} \end{bmatrix} \cdot \begin{bmatrix} \beta_{11} & \beta_{12} & \beta_{13} & \beta_{14} \\ 0 & \beta_{22} & \beta_{23} & \beta_{24} \\ 0 & 0 & \beta_{33} & \beta_{34} \\ 0 & 0 & 0 & \beta_{44} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

We can rearrange the system as,

$$\mathbf{A} \cdot \mathbf{x} = (\mathbf{L} \cdot \mathbf{U}) \cdot \mathbf{x} = \mathbf{L} \cdot (\mathbf{U} \cdot \mathbf{x}) = \mathbf{b}$$

And solve it by defining a vector y such that,

$$\mathbf{L} \cdot \mathbf{y} = \mathbf{b}$$

$$\mathbf{U} \cdot \mathbf{x} = \mathbf{y}$$

Why do this? Because solving systems of equations with triangular matrices of coefficients is very easy. One uses “back-substitution”.

Back-substitution:

$$\begin{bmatrix} a'_{11} & a'_{12} & a'_{13} & a'_{14} \\ 0 & a'_{22} & a'_{23} & a'_{24} \\ 0 & 0 & a'_{33} & a'_{34} \\ 0 & 0 & 0 & a'_{44} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \end{bmatrix}$$

The last equation is trivial to solve,

$$x_4 = b'_4 / a'_{44}$$

But then substituting in the next-to-last also yields a trivial equation,

$$x_3 = \frac{1}{a'_{33}} [b'_3 - x_4 a'_{34}]$$

And so on. The typical solution will be,

$$x_i = \frac{1}{a'_{ii}} \left[b'_i - \sum_{j=i+1}^{N} a'_{ij} x_j \right]$$

(if the matrix is lower triangular the method is similar but called forward substitution)

Gaussian elimination with back-substitution:

Let us briefly go back to the Gauss-Jordan elimination. Suppose we choose a pivot, divide the row by it, but in the next step only combine linearly the row with those subsequent to it in such a way as to produce an upper triangular matrix. The resulting system of equations could be solved by back-substitution.

This method is (a factor less than ten) faster than ordinary Gauss-Jordan elimination.

Both Gauss and Gauss-Jordan require to know explicitly the right hand side of the equations, so the whole method has to be re-done if you wish to change right hand sides. The LU decomposition is independent of the right-hand-side. It is therefore the method of choice if you have to solve for various right-hand-sides. Judiciously implemented it is as fast as Gaussian elimination with back-substitution.

How to perform the LU decomposition?

One has N^2 equations and N^2+N unknowns (the diagonal appears twice, once in U and once in L.

So we just pick the diagonal elements of the L matrix, $\alpha_{ii}=1$

If we now examine carefully the equations,

$$
\begin{aligned}
i<j: &\qquad \alpha_{i1}\beta_{1j}+\alpha_{i2}\beta_{2j}+\cdots+\alpha_{ii}\beta_{ij}=a_{ij}\\
i=j: &\qquad \alpha_{i1}\beta_{1j}+\alpha_{i2}\beta_{2j}+\cdots+\alpha_{ii}\beta_{jj}=a_{ij}\\
i>j: &\qquad \alpha_{i1}\beta_{1j}+\alpha_{i2}\beta_{2j}+\cdots+\alpha_{ij}\beta_{jj}=a_{ij}
\end{aligned}
$$

We see that for i<j, the first two equations determine the beta's,

$$\beta_{ij}=a_{ij}-\sum_{k=1}^{i-1}\alpha_{ik}\beta_{kj}.$$

And substituting in the last equation, one solves for the alpha's,

$$\alpha_{ij}=\frac{1}{\beta_{jj}}\left(a_{ij}-\sum_{k=1}^{j-1}\alpha_{ik}\beta_{kj}\right)$$

All this is for a given j.

Careful examination of the equations shows that every a_{ij} is only used once (Crout). Therefore one can store the computed elements in the same array that was used to enter a,

$$\begin{bmatrix} \beta_{11} & \beta_{12} & \beta_{13} & \beta_{14} \\ \alpha_{21} & \beta_{22} & \beta_{23} & \beta_{24} \\ \alpha_{31} & \alpha_{32} & \beta_{33} & \beta_{34} \\ \alpha_{41} & \alpha_{42} & \alpha_{43} & \beta_{44} \end{bmatrix}$$

The diagonal elements of alpha are all one.

The equation $$\alpha_{ij} = \frac{1}{\beta_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} \alpha_{ik} \beta_{kj} \right)$$

Indicates that the method will require pivoting to be stable. Only partial pivoting can be easily implemented. One will then be LU decomposing not the original matrix a, but a row rearrangement of it, which is equivalent as long as one keeps good track of the rearrangements.

```
 subroutine ludcmp(a,n,np,indx,d)
parameter (nmax=100,tiny=1.0e-20)
      dimension a(np,np),indx(n),vv(nmax)
      d=1.
      do 12 i=1,n
        aamax=0.
        do 11 j=1,n
          if (abs(a(i,j)).gt.aamax) aamax=abs(a(i,j))
11      continue
        if (aamax.eq.0.) pause 'singular matrix.'
        vv(i)=1./aamax
12    continue
      do 19 j=1,n
        if (j.gt.1) then
          do 14 i=1,j-1
            sum=a(i,j)
            if (i.gt.1)then
              do 13 k=1,i-1
                sum=sum-a(i,k)*a(k,j)
13            continue
              a(i,j)=sum
            endif
14        continue
        endif
aamax=0.
        do 16 i=j,n
          sum=a(i,j)
          if (j.gt.1)then
            do 15 k=1,j-1
              sum=sum-a(i,k)*a(k,j)
15          continue
            a(i,j)=sum
          endif
          dum=vv(i)*abs(sum)
          if (dum.ge.aamax) then
            imax=i
            aamax=dum
          endif
16      continue
        if (j.ne.imax)then
          do 17 k=1,n
            dum=a(imax,k)
            a(imax,k)=a(j,k)
            a(j,k)=dum
17        continue
          d=-d
          vv(imax)=vv(j)
        endif
        indx(j)=imax
        if(j.ne.n)then
          if(a(j,j).eq.0.)a(j,j)=tiny
          dum=1./a(j,j)
          do 18 i=j+1,n
            a(i,j)=a(i,j)*dum
18        continue
        endif
19    continue
      if(a(n,n).eq.0.)a(n,n)=tiny
      return
      end
```

First and second Crout equation

Find Pivot

Do pivoting

Keep parity

If singular matrix (zero pivot) replace tiny

```
 subroutine lubksb(a,n,np,indx,b)
      dimension a(np,np),indx(n),b(n)
      ii=0
      do 12 i=1,n
        ll=indx(i)
        sum=b(ll)
        b(ll)=b(i)
        if (ii.ne.0)then
          do 11 j=ii,i-1
            sum=sum-a(i,j)*b(j)
11        continue
        else if (sum.ne.0.) then
          ii=i
        endif
        b(i)=sum
12    continue
      do 14 i=n,1,-1
        sum=b(i)
        if(i.lt.n)then
          do 13 j=i+1,n
            sum=sum-a(i,j)*b(j)
13        continue
        endif
        b(i)=sum/a(i,i)
14    continue
      return
      end
```

Assumes that a is already in LU form

Unscrambles pivoting

Is prepared to handle a b with many zeros at the beginning (as in matrix inversion)

Do backsubstitution

Summary

- Metropolis algorithm generates efficiently distributions in higher dimensions.
- Gauss-Jordan elimination works well, particularly with pivoting.
- LU decomposition is better and may handle more robustly problematic situations.