

November 16, 2004

Version 3.0, July 2003
(c) E.T.H. Zurich

```
> Set(screenwidth=74):  
> Set(quiet=false):
```

ignore off Dayhoff matrices and mutation matrices

A mutation matrix, denoted by M , describes the probabilities of amino acid mutations for a given period of evolution.

$$Pr\{\text{amino acid } i \longrightarrow \text{amino acid } j\} = M_{ji}$$

This corresponds to a model of evolution in which amino acids mutate randomly and independently from one another but according to some predefined probabilities. While simple, this is one of the best methods for modelling evolution.

A 1-PAM mutation matrix describes an amount of evolution which will change, on the average, 1% of the amino acids. In mathematical terms this is expressed as a matrix M such that

$$\sum_{i=1}^{20} f_i(1 - M_{ii}) = 0.01$$

where f_i is the frequency of the i^{th} amino acid. The diagonal elements of M are the probabilities that a given amino acid does not change.

If we have a probability or frequency vector p , the product Mp gives the probability vector or the expected frequency of p after a random evolution equivalent to 1-PAM unit. Or, if we start with a given amino acid (a probability vector which contains a 1 in position i and 0s in all others) M_{*i} (the i^{th} column of M) is the corresponding probability vector after one unit of random evolution. Similarly, after k units of evolution (what is sometimes called k -PAM evolution) a frequency vector p will be changed into the frequency vector $M^k p$.

Dayhoff, Schwartz and Orcutt, in their paper "A Model of Evolutionary Change in Proteins" (1978) presented a method for estimating the matrix M from the observation of 1572 accepted mutations between 34 superfamilies of closely related sequences. Their method was pioneering in the field. Nowadays we are able to estimate M by more accurate and better founded methods.

This model of evolution is symmetric, i.e. we cannot distinguish the probability of amino acid i evolving to j from the probability of j evolving to i . This implies a simple relation for the entries in any M^k :

$$f_i(M^k)_{ji} = f_j(M^k)_{ij}$$

A Dayhoff matrix (in honour of Margaret O. Dayhoff) is a matrix, computed from a 250-PAM mutation matrix, used for the standard dynamic programming method of sequence alignment. The Dayhoff matrix entries are related to M^{250} by

$$D_{ij} = 10 \log_{10} \frac{(M^{250})_{ij}}{f_i}$$

Aligning sequences by dynamic programming using Dayhoff matrices is equivalent to finding the alignment which maximizes the probability that the two sequences evolved from an ancestral sequence as opposed to being random sequences. More precisely, we are comparing two events

- a) that the two sequences are independent of each other, and hence an arbitrary position with amino acid i aligned to another arbitrary position with amino acid j has the probability equal to the product of the individual frequencies

$$Pr\{\text{independent alignment of } i \text{ and } j\} = f_i f_j$$

- b) that the two sequences have evolved from some common ancestral sequence after some amount, t , of evolution.

$$\begin{aligned} Pr\{i \text{ and } j \text{ from a common ancestor } x\} &= \sum_x f_x Pr\{x \rightarrow i\} Pr\{x \rightarrow j\} \\ &= \sum_x f_x (M^t)_{ix} (M^t)_{jx} \\ &= \sum_x f_j (M^t)_{ix} (M^t)_{xj} \\ &= f_j (M^{2t})_{ij} = f_i (M^{2t})_{ji} \end{aligned}$$

The entries of the Dayhoff matrix are the logarithm of the quotient of these two probabilities. Since dynamic programming maximizes the sum of the similarity measure, dynamic programming maximizes the sum of the logarithms or maximizes the product of these quotients of probabilities. As a conclusion, dynamic programming finds the alignment which maximizes the probability of having evolved from a common ancestor (a maximum likelihood alignment) against the null hypothesis of being independent. This makes aligning sequences using Dayhoff matrices a soundly based algorithm (better founded than any other alignment algorithm), a fact little known among many people in the area.

The resulting measure of similarity after a matching is a sum of Dayhoff entries, and hence it is 10 times the logarithm of this probability. For example, the matching described by

```
> ReadDb('~darwin/DB/SwissProt');
```

```
Peptide file(/pub/home/darwin/DB/SP45.0/SwissProt45.0(169638448), 163235
  entries, 59631787 aminoacids)
```

```
> m := Match(238.8,9457216,9457223,101,101,250):
```

with a similarity (or cost or score) of 238.8 means that the probability of both sequences coming from a common ancestor, as opposed to being a random alignment, is $10^{24.000000}$ times more likely. Although crude, this gives a rule of thumb for estimating the quality of a matching.

1 Gap probabilities and gap penalties

Modelling amino acid alignments is based on the evolution from an ancestral sequence. This model allows us to link the mutation matrix to the scores used by dynamic programming.

In this section we describe how to compute the gap penalty costs from the probabilities of such gaps happening. In other words, the same derivation we have done from mutation probabilities to scores, done for gap probabilities to gap penalties.

The gap model we will use here is the one which is implicit in the vast majority of the present literature, which assumes that gaps are accidents at the DNA level which produce a random insertion or deletion of k amino acids with probability $q(k)$. This probability is independent of the amino acids being inserted or deleted and of any neighbouring amino acids. It is a natural assumption that this probability also depends on the PAM distance between the two sequences, i.e. gaps in similar sequences are less frequent than gaps in distant ones. Then

$$Pr\{k\text{-gap at distance } t\} = q(k, t)$$

Gap penalties used with dynamic programming are typically of the form $a + bk$. In some cases the values of a and b depend on the PAM distance t . This gap penalty function used for dynamic programming will be denoted by $d(k, t)$. For example

$$d(k, t) = a + bk$$

The linear gap penalty function is very popular, in part because it is possible to compute dynamic programming alignments very efficiently using Gotoh's algorithm.

The relation between $d(k, t)$ and $q(k, t)$ is simply

$$d(k, t) = 10 \log_{10} q(k, t)$$

This relation is not well known, so we will prove it here. Readers not interested in the mathematical theory of alignments should skip the rest of this section.

The purpose of aligning sequences with dynamic programming is to find the alignment which maximizes the probability of the two sequences having evolved from a common ancestor as opposed to being just random sequences. Let A and B be two sequences with lengths n_A and n_B . Let A_i be the i^{th} amino acid in A and similarly for B . Let f_i be the probability of amino acid i as before. The probability that the two sequences are random is

$$P_R = Pr\{\text{random}\} = \prod_{i=1}^{n_A} f_{A_i} \times \prod_{i=1}^{n_B} f_{B_i}$$

Given an alignment, for example

$$\begin{array}{cccccc} A_1 & A_2 & - & - & - & A_3 \\ X_1 & X_2 & X_3 & X_4 & X_5 & X_6 \\ B_1 & B_2 & B_3 & B_4 & B_5 & B_6 \end{array}$$

where X denotes the unknown ancestral parent sequence. The probability of this alignment being the consequence of having a common ancestor is

$$P_A = Pr\{\text{having evolved from a common, unknown, ancestor X}\} =$$

$$\sum_{X_1=1}^{20} f_{X_1}(M^{t/2})_{A_1 X_1} (M^{t/2})_{B_1 X_1} \times \sum_{X_2=1}^{20} f_{X_2}(M^{t/2})_{A_2 X_2} (M^{t/2})_{B_2 X_2} \times$$

for all the possible X_1 which evolved into A_1 on one side and into B_1 on the other, and similarly for X_2 ;

$$\times q(3, t) f_{B_3} f_{B_4} f_{B_5} \times$$

where this term denotes the probability of a gap of length 3 on one sequence times the probability of the sequence $B_3 B_4 B_5$ on the other and

$$\times \sum_{X_6=1}^{20} f_{X_6}(M^{t/2})_{A_3 X_6} (M^{t/2})_{B_6 X_6}$$

for the alignment of A_3 against B_6 . For this alignment the quotient of the probabilities is

$$\frac{P_A}{P_R} = \frac{(M^t)_{B_1 A_1}}{f_{B_1}} \times \frac{(M^t)_{B_2 A_2}}{f_{B_2}} \times q(3, t) \times \frac{(M^t)_{B_6 A_3}}{f_{B_6}}$$

once properly grouped and simplified (see beginning of this chapter). To find the alignment which maximizes this probability we transform the maximization of a product (of all positive values) into the maximization of a sum by computing the logarithm of each term. To find the alignment which maximizes a sum, we use the dynamic programming algorithm. For historical reasons and to work with simpler numbers, these logarithms are multiplied by 10, which has no effect on the maximization process. Notice that the terms like

$$D(A_1, B_1) = 10 \log_{10} \left(\frac{(M^t)_{B_1 A_1}}{f_{B_1}} \right)$$

are exactly the values of the standard Dayhoff matrices. The deletion cost for the above example, under dynamic programming, must be

$$d(3, t) = 10 \log_{10} q(3, t)$$

It is obvious how to generalize this result for any number or length of gaps.

The relationship found can be applied in two directions, either we can estimate the probabilities from a sample and compute the appropriate gap penalties (see chapter 16 for an example of such a derivation) or we can compute the probabilities that are implied by a gap penalty function.

A linear gap penalty function implies an exponential distribution of probabilities.

$$d(k, t) = a + bk = 10 \log_{10} q(k, t)$$

implies

$$q(k, t) = 10^{a/10} (10^{b/10})^k$$

For example, if $a = -10$ and $b = -3$, then

$$q(k, t) = 0.1000 \times (0.5012)^k$$

This further implies that the probability of a gap, of any length, is

$$\sum_{k=1}^{\infty} q(k, t) = \sum_{k=1}^{\infty} 10^{a/10} (10^{b/10})^k = \frac{10^{a/10}}{10^{-b/10} - 1}$$

in our example, the probability of having a gap would be 10.05%. Finally, the expected length of a gap is

$$\frac{\sum_{k=1}^{\infty} kq(k, t)}{\sum_{k=1}^{\infty} q(k, t)} = \frac{1}{1 - 10^{b/10}}$$

in our example, the expected length of a gap would be 2.00.

2 Computation of matrices

In Darwin, a Dayhoff matrix can be computed following basically the method described by Dayhoff et al. with the command `CreateOrigDayMatrix`

```
> print(CreateOrigDayMatrix);
```

```
CreateOrigDayMatrix: Usage: CreateOrigDayMatrix( Mutations:array(numeric
,
    20,20), AaCounts:array(numeric,20), PamNumber:{numeric,1..posint} )
```

This command takes three arguments, an observed mutations matrix, this matrix is lower triangular, only the entries below the diagonal are taken into account (the entry `Mutations[i,j]` corresponds to the number of observed mutations between amino acids `i` and `j`). The second argument is a frequency vector (or probability vector) of the amino acids. This vector is necessary in the computation, and it is not clear whether the correct one to use is the one originally cited in Dayhoff's paper, or the one arising from the data (the entire database) where the computations will be done. In all our work we have been using the frequencies of our entire database, which differ from those reported in the original paper. The last argument to `Dayhoff` is the value of the PAM distance (amount of evolutionary distance) that we want to model with this matrix.

The observed mutation matrix reported by Dayhoff et al. is available in Darwin as a constant, `Mutations1978` (here we show the first 18 columns).

```
> print(Mutations1978);
```

```

  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 30  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
109 17  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
154  0 532  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 33 10  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 93 120 50 76  0  0  0  0  0  0  0  0  0  0  0  0  0  0
266  0 94 831  0 422  0  0  0  0  0  0  0  0  0  0  0  0
579 10 156 162 10 30 112  0  0  0  0  0  0  0  0  0  0  0
 21 103 226 43 10 243 23 10  0  0  0  0  0  0  0  0  0  0
```

```

66 30 36 13 17 8 35 0 3 0 0 0 0 0 0 0 0 0
95 17 37 0 0 75 15 17 40 253 0 0 0 0 0 0 0 0
57 477 322 85 0 147 104 60 23 43 39 0 0 0 0 0 0 0
29 17 0 0 0 20 7 7 0 57 207 90 0 0 0 0 0 0
20 7 7 0 0 0 0 17 20 90 167 0 17 0 0 0 0 0
345 67 27 10 10 93 40 49 50 7 43 43 4 7 0 0 0 0
772 137 432 98 117 47 86 450 26 20 32 168 20 40 269 0 0 0
590 20 169 57 10 37 31 50 14 129 52 200 28 10 73 696 0 0
0 27 3 0 0 0 0 0 3 0 13 0 0 10 0 17 0 0
20 3 36 0 30 0 10 0 40 13 23 10 0 260 0 22 23 6
365 20 13 17 33 27 37 97 30 661 303 17 77 10 50 43 186 0

```

The originally reported frequencies are proportional to

```
> OrigFreq := [87,41,40,47,33,38,50,89,34,37,85,81,15,40,51,70,58,10,30,65]:
```

and the frequencies of the SwissProt version 64 database are

```
ReadDb('~darwin/DB/SwissProt');
```

```
> SPFreq := GetAaCount(DB):
```

Normalizing and printing the frequencies we can see that some values are notoriously different:

```
> OrigFreq := OrigFreq/sum(OrigFreq):
```

```
> SPFreq := SPFreq/sum(SPFreq):
```

```
> for i to 20 do
```

```
>   printf('%15.15s%7.2f%%%7.2f%%\n', IntToAmino(i),
```

```
>     100*OrigFreq[i], 100*SPFreq[i]) od;
```

Alanine	8.69%	7.83%
Arginine	4.10%	5.32%
Asparagine	4.00%	4.20%
Aspartic acid	4.70%	5.31%
Cysteine	3.30%	1.56%
Glutamine	3.80%	3.94%
Glutamic acid	5.00%	6.60%
Glycine	8.89%	6.94%
Histidine	3.40%	2.28%
Isoleucine	3.70%	5.91%
Leucine	8.49%	9.63%
Lysine	8.09%	5.93%
Methionine	1.50%	2.38%
Phenylalanine	4.00%	4.01%
Proline	5.09%	4.85%
Serine	6.99%	6.88%
Threonine	5.79%	5.46%
Tryptophan	1.00%	1.16%
Tyrosine	3.00%	3.08%
Valine	6.49%	6.72%

The original Dayhoff matrix can be recomputed with the command

```
> OrigDM := CreateOrigDayMatrix(Mutations1978,OrigFreq,250):  
> print(OrigDM);
```

```
DayMatrix(Peptide, pam=250, Sim: max=17.302, min=-7.510,  
max offdiag=6.951, del=-19.814-1.396*(k-1))
```

```
C 12.0  
S -0.0 1.6  
T -2.2 1.3 2.6  
P -2.7 0.9 0.3 5.9  
A -2.0 1.1 1.2 1.1 1.8  
G -3.3 1.1 -0.0 -0.5 1.3 4.8  
N -3.6 0.7 0.4 -0.5 0.2 0.4 2.0  
D -5.1 0.3 -0.1 -1.0 0.3 0.6 2.1 3.9  
E -5.3 -0.0 -0.4 -0.6 0.3 0.2 1.4 3.4 3.9  
Q -5.3 -0.5 -0.8 0.2 -0.4 -1.2 0.8 1.6 2.5 4.1  
H -3.4 -0.8 -1.3 -0.3 -1.4 -2.1 1.6 0.7 0.6 2.9 6.6  
R -3.6 -0.3 -0.9 -0.2 -1.6 -2.6 -0.0 -1.3 -1.1 1.2 1.5 6.1  
K -5.4 -0.2 -0.0 -1.2 -1.2 -1.7 1.0 0.1 -0.1 0.7 -0.1 3.4 4.7  
M -5.2 -1.6 -0.6 -2.1 -1.2 -2.8 -1.8 -2.6 -2.2 -1.0 -2.2 -0.5 0.4 6.6  
I -2.3 -1.4 0.1 -2.0 -0.5 -2.6 -1.8 -2.4 -2.0 -2.0 -2.5 -2.0 -1.9 2.2  
L -6.0 -2.8 -1.7 -2.6 -1.9 -4.0 -2.9 -4.0 -3.3 -1.8 -2.1 -3.0 -2.9 3.7  
V -1.9 -1.0 0.3 -1.2 0.2 -1.4 -1.8 -2.2 -1.8 -1.9 -2.3 -2.5 -2.5 1.8  
F -4.3 -3.2 -3.1 -4.6 -3.5 -4.8 -3.5 -5.6 -5.4 -4.7 -1.8 -4.5 -5.3 0.2  
Y 0.4 -2.8 -2.8 -5.0 -3.5 -5.3 -2.1 -4.3 -4.3 -4.0 -0.1 -4.2 -4.5 -2.5  
W -7.5 -2.3 -5.0 -5.5 -5.6 -6.8 -3.9 -6.6 -6.8 -4.6 -2.5 2.3 -3.3 -4.1
```

From a Dayhoff matrix we can extract the values of the entries,

```
> OrigDM[Sim,1,5], OrigDM[Sim,AToInt(W),AToInt(W)];
```

```
-1.9896, 17.3021
```

the PAM distance for which this matrix was computed,

```
> OrigDM[PamNumber];
```

```
250
```

maximum and minimum values

```
> OrigDM[MaxSim], OrigDM[MinSim], OrigDM[MaxOffDiag];
```

```
17.3021, -7.5098, 6.9511
```

the adjusted deletion penalty and the deletion penalty increment.

```
> OrigDM[FixedDel], OrigDM[IncDel];
```

-19.8137, -1.3961

The matchings using dynamic programming or Needleman & Wunsch's algorithm will apply penalties for deletions of length k which are $\text{FixedDel} + (k-1)*\text{IncDel}$. This gap penalty implies that a gap of length k happens with probability $0.0076 \times (0.7251)^k$.

From the all-against-all matching of an entire database containing more than 27,000 sequences we have derived new similarity matrices and new deletion penalty models. These are available in Darwin with the command `CreateDayMatrices()`.

```
> print( CreateDayMatrices );
```

```
CreateDayMatrices: Usage: CreateDayMatrices( count:array(numeric,20,20) )
```

This command computes a similarity matrix at PAM distance 250, left in the variable `DM`, 1266 similarity matrices for various PAM values between 0.049 and 1000, which are suitable to estimate PAM distances and `logPAM1`, the logarithm of a 1-PAM mutation matrix.

The `CreateDayMatrix` command can be used to compute a similarity matrix based on the logarithm of a 1-PAM matrix. This is done like

```
> Sim123 := CreateDayMatrix( logPAM1, 123 );
```

```
Sim123 := DayMatrix(Peptide, pam=123, Sim: max=16.500, min=-9.462,  
max offdiag=5.480, del=-22.104-1.396*(k-1))
```

For various purposes, in particular for computing an estimate of the PAM distance of a match, it may be necessary to use an array of Dayhoff matrices, for various PAM numbers. If the third argument to the `Dayhoff` (or the second argument of `CreateDayMatrix`) function is a range of integers starting at 1, then the command will compute an array of Dayhoff matrices for all PAM numbers between 1 and the upper limit of the range. For example

```
> DMS := CreateDayMatrix(logPAM1,1..500):
```

computes an array with 500 Dayhoff matrices, with PAM values from 1 to 500. It is interesting to see how these matrices change, from a very low PAM number where the matrix is positive on the diagonal and very negative in the off-diagonal to a higher PAM number where the values smooth out.

```
> print(DMS[1]);
```

```
DayMatrix(Peptide, pam=1, Sim: max=18.820, min=-37.533,  
max offdiag=-11.365, del=-37.640-1.396*(k-1))
```

```
C 17.2  
S -18.5 12.1  
T -21.6-12.7 12.0  
P -33.2-18.6-19.5 13.4  
A -18.1-14.3-17.5-18.8 11.0  
G -25.2-18.7-25.3-24.9-18.2 11.3  
N -24.1-15.5-17.6-24.0-22.3-19.2 13.4  
D -32.1-18.7-20.1-22.7-21.2-20.5-14.0 12.7
```

```

E -35.4-19.4-20.8-21.6-18.6-23.8-19.5-12.8 12.3
Q -28.7-18.4-18.9-19.7-19.4-22.8-17.4-18.7-13.2 14.2
H -22.1-20.2-19.7-22.8-22.1-24.1-15.3-19.4-19.4-14.3 16.2
R -24.2-20.4-20.6-23.1-21.7-22.5-20.3-24.4-21.0-15.2-18.2 12.7
K -33.3-19.6-18.3-21.8-20.9-23.9-16.7-20.5-16.0-14.2-18.6-12.5 12.3
M -21.6-21.7-19.3-31.0-19.4-27.6-25.4-32.7-23.7-17.8-21.2-24.3-20.7 16.4
I -25.5-26.3-20.4-27.5-25.0-34.6-25.5-33.8-27.2-25.8-25.4-26.9-24.2-13.4
L -24.5-25.7-24.0-23.8-22.3-31.1-27.7-35.3-27.4-21.0-24.4-24.1-24.3-12.9
V -19.4-24.3-17.1-24.6-16.6-29.5-27.2-33.8-22.7-23.9-26.5-24.7-24.1-17.5
F -22.0-28.0-24.8-29.6-25.1-32.2-26.5-33.1-32.5-26.3-20.6-31.1-30.3-16.5
Y -21.0-22.0-24.8-26.7-25.7-29.6-21.5-25.8-27.2-24.1-14.2-23.0-24.7-22.7
W -22.4-25.4-28.7-32.1-28.5-26.5-28.3-37.5-29.5-24.1-22.9-21.5-30.6-22.8

```

```
> print(DMS[500]);
```

```
DayMatrix(Peptide, pam=500, Sim: max=9.801, min=-2.049, max offdiag=3.11
del=-17.576-1.396*(k-1))
```

```

C 6.4
S 0.1 0.4
T -0.1 0.3 0.3
P -0.9 0.3 0.2 3.2
A 0.2 0.3 0.2 0.2 0.3
G -0.6 0.4 -0.1 -0.2 0.4 3.2
N -0.5 0.3 0.2 0.0 0.1 0.5 0.8
D -0.9 0.4 0.2 0.1 0.1 0.5 0.9 1.4
E -0.8 0.3 0.2 0.1 0.1 0.1 0.6 1.0 0.9
Q -0.7 0.2 0.1 0.1 0.0 -0.0 0.4 0.5 0.5 0.5
H -0.4 0.1 0.0 -0.2 -0.1 -0.2 0.4 0.3 0.3 0.3 1.1
R -0.7 0.2 0.1 -0.0 -0.0 -0.0 0.4 0.3 0.4 0.5 0.3 1.2
K -0.7 0.2 0.2 0.1 0.0 -0.0 0.4 0.5 0.6 0.5 0.3 0.9 0.8
M -0.1 -0.5 -0.2 -0.7 -0.2 -1.3 -0.7 -1.0 -0.7 -0.4 -0.4 -0.6 -0.5 1.1
I -0.1 -0.6 -0.2 -0.8 -0.2 -1.6 -0.9 -1.2 -0.9 -0.6 -0.6 -0.8 -0.7 1.2
L -0.2 -0.7 -0.3 -0.8 -0.3 -1.7 -1.0 -1.3 -1.0 -0.6 -0.5 -0.8 -0.7 1.2
V 0.2 -0.3 -0.1 -0.5 -0.1 -1.2 -0.7 -0.9 -0.6 -0.4 -0.5 -0.6 -0.5 0.9
F 0.0 -1.0 -0.7 -1.4 -0.7 -2.0 -1.1 -1.7 -1.4 -0.9 0.1 -1.1 -1.1 1.0
Y 0.1 -0.7 -0.6 -1.1 -0.7 -1.6 -0.7 -1.1 -0.9 -0.6 0.7 -0.6 -0.7 0.4
W -0.1 -1.3 -1.3 -2.0 -1.3 -1.9 -1.4 -2.0 -1.7 -1.1 -0.0 -0.8 -1.3 0.1

```

When we have an array of similarity matrices, we can select a particular one just by indexing the array or with the function `SearchDayMatrix`. `SearchDayMatrix` takes a PAM number (not necessarily an integer) and an array of matrices and returns the matrix which has a PAM number closest to the given one. This function is most useful when we have an array `DMS` as the one generated by `CreateDayMatrices`, for example

```
> SearchDayMatrix( 27.7, DMS );
```

```
DayMatrix(Peptide, pam=28, Sim: max=18.302, min=-19.521,
max offdiag=1.939, del=-26.882-1.396*(k-1))
```

Every time that we use the Dayhoff function, Darwin also assigns the natural logarithm of the corresponding mutation matrix M to the global variable `logPAM1`. From the previous computations, the first 18 columns of `logPAM1`, scaled and rounded are:

```
> PrintMatrix( 10000*logPAM1, '%4f' );

-110.4189645.2902984.5994485.90872912.2346539.10509310.80900811.8839894.
3.595612-93.0500484.9700491.9269452.02120316.2044514.2269273.0299638.088
2.6669904.240175-112.28388218.3173171.7659038.3571005.0684775.55476513.4
4.0399221.93845421.598571-95.3720380.3213397.24838228.3035194.8128926.14
2.9360690.7136610.7308460.112787-54.5097630.2515680.0507730.5694031.1586
4.37104511.4457066.9189595.0893590.503248-144.70263418.1663621.96778314.
8.0342724.6226696.49714230.7696240.15725928.127280-111.1381252.4507946.7
11.1241574.1730118.9671586.5891912.2210093.8369063.086391-47.9522382.870
1.4395323.5642036.9423242.6935531.4459878.7910452.7253270.918354-105.542
1.7965791.1727691.6350820.2369731.6033351.5116831.1025440.1948221.691006
5.5130253.7187661.5840270.2682793.3268867.5942811.7064580.7337643.419877
4.83408033.03913512.7284565.2914500.26156522.69195714.8955712.3930348.16
2.6616990.8510540.6633270.1169001.5890463.8171270.9811310.3941651.754447
1.2578180.3112240.9157850.1976452.6029870.9509330.2244420.2436283.582735
6.0390012.2252471.7941652.4513690.2075794.9416123.1518501.4815672.406558
22.8046265.49038117.0249348.1560118.5651278.7817766.9691078.1116975.7460
11.0596215.35556410.8886546.0985814.2776607.9074755.1500081.7905786.5776
0.1819100.9275630.1908030.0218740.7494670.5103360.1450930.2906930.677269
0.8821031.6358132.3164380.8558122.5890051.2634510.6264580.35714712.62460
15.1809032.3343541.3177130.2696408.0665052.8101773.7486770.7731991.53320
```

This matrix is convenient, as if we want to compute the mutation matrix for any PAM distance we just use the function `exp` (which can also work with matrices) using the standard algebraic rules for exponentials and logarithms, i.e.

$$M^k = e^{k \times \log(M)}$$

```
> M55 := exp( 55*logPAM1 );
> PrintMatrix( 1000*M55, '%4f' );

559.69787725.16021625.79195828.36852149.04022536.98330241.91709947.56960
17.100428612.27949523.71788612.8650799.75613157.01239022.71116214.429746
14.95547020.234808549.64327062.6453969.04898531.64432125.11677223.382749
19.39615312.94192073.867316606.0463103.73481833.07548994.17608121.897893
11.7686623.4447653.7450601.310887742.1215602.0802791.2248863.2061905.205
17.75442340.26961726.19877123.2235314.161480463.38221255.9878869.7724314
31.15673024.83747132.196506102.3817083.79386486.686975559.53314113.86262
44.52812819.87334337.74719529.97976912.50603019.05489617.457810772.19161
6.96862314.61867724.15374511.9819806.49597428.11736112.1419574.935754564
14.3790137.5994487.9916102.92308411.2246119.9425037.1727742.4248099.1738
25.96364717.90636810.0687724.34778918.34856129.79425811.1248155.40917117
21.955010108.56648846.11390926.6041924.50317674.01692653.49264713.097347
```

10.0617834.7706203.8188391.7961587.03009712.3653554.8428862.3132677.2542
6.5809513.0943515.0441241.87783312.2469115.5969982.3266321.81334416.8421
24.77233311.55736010.41793812.4272713.12426120.26573415.0827108.37682611
71.87986024.46333557.03896133.85390833.67043833.91249129.92360632.933874
43.74047924.57548641.87795027.06756920.85630631.83684124.64249012.384009
1.1585573.9066341.1939720.4195263.5799292.3091070.9100141.4496713.501238
5.0869407.57532410.0448874.65281611.7222476.8542743.9733192.43029144.591
51.09493412.3242739.3273325.22667833.03439615.06828916.2413146.11877610.

Using logPAM1 it is very economical to compute various mutation matrices or Dayhoff matrices, even for non-integer PAM distances.

A 1-PAM mutation matrix measures a unit of evolution where the expected change is exactly 1% of the amino acids. What about a 2-PAM evolution? It is clear that a 2-PAM evolution will not produce a 2% change (a 2-PAM evolution is absolutely equivalent to two consecutive 1-PAM evolutions) since some changes may mutate back to the original amino acid. The percentage identity (pi) is related to the PAM distance p by

$$pi(p) = \sum_{i=1}^{20} f_i (M^p)_{ii}$$

In the above formula we need both the mutation matrix and the frequency vector. As it turns out to be, the frequency vector can be derived from any mutation matrix. Recall the symmetry relation:

$$f_i M_{ji} = f_j M_{ij}$$

Setting $j = 1$ and rearranging

$$f_i = \frac{f_1 M_{i1}}{M_{1i}}$$

On the other hand we know that $\sum_{i=1}^{20} f_i = 1$, consequently

$$f_i = \frac{M_{i1}}{M_{1i} \sum_{j=1}^{20} \frac{M_{j1}}{M_{1j}}}$$

The formula for $pi(p)$ can be made independent of the frequencies f_i as:

$$pi(p) = \frac{\sum_{i=1}^{20} \frac{M_{i1}}{M_{1i}} (M^p)_{ii}}{\sum_{i=1}^{20} \frac{M_{i1}}{M_{1i}}}$$

for any mutation matrix M , in particular for M^p which is also a mutation matrix. The next two functions compute the percent identity for a given PAM distance and its inverse, the PAM number p which will result in the given pi . The first function is a straightforward computation using the above formula:

```
> PamtoPI := proc( pam:numeric, logPAM1:array(numeric,20,20) )
> description
>   'compute the percentage identity that a pam distance will leave';
> if pam < 0 then ERROR('invalid percentage range') fi;
```

```

>
> num := den := 0;
> M := exp(pam*logPAM1);
> for i to 20 do
>   num := num + M[i,1]*M[i,i]/M[1,i];
>   den := den + M[i,1]/M[1,i]
>   od;
> 100*num/den;
> end:

```

By definition, the following should be 99%

```
> PamtoPI(1,logPAM1);
```

99.0000

```
> PamtoPI(250,logPAM1);
```

16.9051

the classical PAM value gives about 16.905077% identity.

The inverse function, called `PItoPam` requires the solution of a nontrivial equation. Since it is very unlikely that a closed form solution exists, it is best to use a numerical method. In this case we use Newton's iteration. Newton's method for solving equations numerically starts from an initial guess x_0 and computes x_1, x_2, \dots using

$$x_{i+1} = x_i - \frac{g(x_i)}{g'_x(x_i)}$$

where $g(x)$ is the equation to be solved in the unknown x and $g'_x(x)$ is the derivative of $g(x)$ with respect to x . In our case, the equation is

$$g(p) = \sum_{i=1}^{20} f_i(M^p)_{ii} - pi = 0$$

and our unknown is p , the PAM distance. Computing the derivative of M^p with respect to p is not so complicated, recall that M is a square matrix, $\ln(M)$ is also a square matrix and using the series definition we can compute

$$\begin{aligned}
(M^p)'_p = (e^{p \ln(M)})'_p &= (I + p \ln(M) + \frac{p^2 \ln(M)^2}{2} + \dots)'_p \\
&= \ln(M) + p \ln(M)^2 + \frac{p^2 \ln(M)^3}{2} + \dots \\
&= \ln(M) e^{p \ln(M)} = \ln(M) M^p \\
&= e^{p \ln(M)} \ln(M) = M^p \ln(M)
\end{aligned}$$

(M and $\ln(M)$ necessarily commute under matrix multiplication). As expected the derivative of powers of matrices has the same expression as for scalars.

Consequently the derivative of the equation is

$$g'_x(p) = \sum_{i=1}^{20} f_i(\ln(M)M^p)_{ii}$$

This exercise also shows how to protect a Newton iteration from going astray by using binary search techniques. Newton's iteration, when it converges, it does quadratically, so it is very fast, but often does not converge at all. The rest of the details are explained with the code.

```
> PIttoPam := proc( pi:numeric, logPAM1:array(numeric,20,20) )
> description 'compute the PAM distance which results in the given
>   percentage identity';
```

check the argument range and trivial case

```
> if pi <= 0 or pi > 100 then ERROR('invalid percentage range') fi;
> if pi=100 then RETURN(0) fi;
```

Compute the amino acid frequencies as described before

```
> AF := CreateArray(1..20);
> M := exp(logPAM1);
> for i to 20 do AF[i] := M[i,1]/M[1,i] od;
> AF := AF/sum(AF);
```

Now check that the percentage identity is not less than its asymptotic value. I.e. for PAM $\rightarrow \infty$, the percentage identity does not decrease to zero but to the percentage identity of two random sequences. This value is

$$\lim_{p \rightarrow \infty} pi(p) = \sum_{i=1}^{20} f_i^2$$

```
> asy := AF*AF;
> if pi/100 <= asy then
>   ERROR('pi cannot be less than the asymptotic value',100*asy) fi;
>
```

For reasons which escape the scope of this tutorial, the following is a good initial guess for the PAM value.

```
> pam := -100*ln(pi/100-asy);
```

Next, the PAM value will be bound below by lo and above by hi. Setting the above bound to 5 times the initial guess is sufficient.

```
> lo := 0; hi := 5*pam;
```

We will now iterate until we reach the desired accuracy, i.e. the difference between the bounds is close to the machine epsilon.

```

> while hi-lo > hi*DBL_EPSILON*10 do
>   mp := exp(pam*logPAM1);
>   m1p := logPAM1*mp;
>   num := -pi/100;
>   den := 0;
>   for i to 20 do
>     num := num + AF[i]*mp[i,i];
>     den := den + AF[i]*m1p[i,i]
>   od;

```

Reset lo or hi as appropriate

```

>   if num >= 0 then lo := pam else hi := pam fi;
>   incr := -num/den;
>   pam := pam + incr;

```

If the increment squared is relatively less than the machine epsilon we can stop the iteration. Notice that when doing Newton's iteration, the relative error is proportional to the square of the previous relative error.

```

>   if abs(incr)^2 < abs(pam)*DBL_EPSILON then break fi;

```

If the value falls outside the bounds, then the Newton's value is not converging, so we just compute the midpoint of the bounds as the next guess.

```

>   if pam <= lo or pam >= hi then pam := (lo+hi)/2 fi;
>   od;
> pam
> end:

```

Again, by definition, the following should evaluate to 1.

```

> PItoPam(99,logPAM1);

```

1.0000

Several people consider that 15% identity is the limit of our ability to align sequences, this corresponds to PAM:

```

> PItoPam(15,logPAM1);

```

278.5236

3 Estimating mutation matrices

To close this section we will describe a method for computing mutation matrices, and hence Dayhoff matrices, which is more accurate and based on better theoretical grounds. This method will derive the information from a sample of matches of sequences (see "Computing similarity matrices based on the results of an All-against-all database matching", G. Gonnet,

to appear). We will assume that we have a large enough sample which has been inspected and approved by an expert in the area. Although we could construct sample sets automatically, it is important that a person analyses this sample and weeds out unsuitable sequences which do not represent mutations by evolution.

We will further require that this sample be selected so that all its matches have approximately the same PAM distance. For example, let's say that all the matches have been selected so that their distance is between 35 and 45 PAM. Section 8 explains how to compute PAM distances of matches.

Suppose that we could run the following experiment:

- i) Generate a random sequence S_0 .
- ii) Randomly mutate S_0 with a 40-PAM mutation matrix generating a new sequence S_1 .
- iii) Compare S_0 and S_1 and build a comparison matrix C , such that every time that we have an amino acid i in S_0 mutating into an amino acid j in S_1 we add 1 to C_{ij} (and if it does not mutate, i.e. stays the same, we add 1 to C_{ii}).

It is easy to see that the expected value of C is given by

$$C \approx M^{40} \times N$$

where N is a diagonal matrix where N_{ii} is the number of amino acids i in S_0 . This approximation, by the law of large numbers, will be more accurate as we increase the number of sample points, i.e. increase the length of S_0 and hence increase N .

The above equation is crucial to estimate M when we can tabulate a matrix C . Doing some trivial algebra, we find that

$$M \approx (C \times N^{-1})^{1/40}$$

Since we do not know, from our real data, which sequence evolved from which, i.e. our model of evolution is symmetric. Instead of adding 1 to C_{ij} we will add 1/2 to C_{ij} and 1/2 to C_{ji} .

The matrix N is obtained from counting all the amino acids analysed, and in particular, multiplying by N^{-1} is the same as dividing each of the columns of the matrix C by their sum. This coincides nicely with the fact that the columns of a mutation matrix add up to 1.

Thirdly, if we were certain about the PAM distance of the sample, we could compute $(C \times N^{-1})^{1/40}$. However, we may not know the PAM distances accurately. This is not a problem at all, as we can determine when a mutation matrix is 1-PAM by definition:

$$\sum_i f_i(1 - M_{ii}) = 0.01$$

So computationally we just need to find α such that $(C \times N^{-1})^{1/\alpha}$ is a 1-PAM matrix. This value α will be an estimate of the average PAM distance of the sample.

The following Darwin function computes an estimate of a 1-PAM matrix based on a sample of matches stored in a file. This function also receives as parameters a minimum value for the similarity, a minimum value for the length of the match and the minimum and maximum value for the PAM distance that we want to select.

```

> EstMutMat := proc( filename:string, MinSim:numeric, MinLength:numeric,
>     MinPam:numeric, MaxPam:numeric )
> description 'estimate a mutation matrix from matches read from a file';
> C := CreateArray(1..20,1..20);
> M := CreateArray(1..20,1..20);

```

Print some appropriate message to describe the present run.

```

> printf('Sample matches from file %s, MinSim=%g, MinLength=%g\n',
>     filename,MinSim,MinLength);
> printf('PAM bounded between %3g and %3g\n', MinPam, MaxPam );

```

The alignments are in a variable called SampleAl that is read in from the file data.drw. line by line we have to first issue

```

> ReadDb('~darwin/DB/SwissProt');
> ReadProgram(filename);

```

Initialize various counters to zero.

```

> totm := totma := totlm := toteq := totmut := 0;

```

This is now the main reading loop.

```

> for m in SampleAl do

```

It is worthwhile to check that we read a match from the file.

```

> if not type(m,Alignment) then ERROR('invalid input in', filename) fi;
> totm := totm+1;

```

If the values of the alignments do not fall within the acceptable ranges, then skip this match.

```

> if m[Sim] < MinSim or max(m[Length1],m[Length2]) < MinLength or
>     m[PamNumber] > MaxPam or m[PamNumber] < MinPam then next fi;
> totma := totma+1;

```

The function DynProgStrings prepares an alignment for printing. In this case we are interested in the strings of the aligned sequences as they would be printed. By scanning these two strings we can count the identities and the mutations.

```

> sm := DynProgStrings( Match(m), SearchDayMatrix(m[PamNumber],DMS) );
> lm := length(sm[2]);
> totlm := totlm + lm;

```

For each character in the matched strings add in the C matrix.

```

> for i to lm do
>     i1 := AToInt(sm[2,i]); i2 := AToInt(sm[3,i]);
>     if i1>0 and i1<21 and i2>0 and i2<21 then
>         if i1=i2 then toteq := toteq+1;
>             C[i1,i1] := C[i1,i1]+1

```

```

>         else totmut := totmut+1;
>         C[i1,i2] := C[i1,i2]+1/2;
>         C[i2,i1] := C[i2,i1]+1/2
>         fi
>     fi
> od
> od;

```

Print the collected counts.

```

> printf('%d matches read, %d selected with a total of %d positions,\n',
>     totm, totma, totlm );
> printf('%d exact matches, %d mutations and %d deletions\n\n',
>     toteq, totmut, totlm-toteq-totmut );

```

Compute the sum of the columns, but since the matrix C is symmetric, it is easier to compute the sum of the rows.

```

> Cols := CreateArray(1..20);
> for i to 20 do Cols[i] := sum(C[i]) od;

```

Now compute the frequencies of amino acids into F and normalize C into M.

```

> tot := sum(Cols);
> F := Cols/tot;
> for i to 20 do for j to 20 do M[i,j] := C[i,j]/Cols[j] od od;

```

To compute the 1-PAM matrix we need to find the exponent $1/\alpha$ that will make $M^{1/\alpha}$ a 1-PAM matrix. The following iteration converges quite rapidly to the desired solution. It is based on estimating $1/\alpha$ as if it were linear on the probability of no change. This is not true, but both values are highly correlated. Hence by estimating and further correcting we converge to the right exponent.

```

> lnM := ln(M);
> alpha := 1;

```

Now iterate until a break is executed.

```

> do nochange := 0;

```

Compute the probability of no change and if it is sufficiently close to 1%, exit the loop.

```

>     for i to 20 do nochange := nochange + F[i]*(1-M[i,i]) od;
>     if abs(nochange-0.01) < DBL_EPSILON then break fi;

```

Not accurate enough, needs to be corrected.

```

>     corr := 0.01/nochange;
>     alpha := alpha/corr;
>     lnM := lnM*corr;
>     M := exp(lnM);
>     od;
> printf('The PAM of the aggregated sample was %5g\n\n',alpha);

```

Return the logarithm of a 1-PAM matrix and the frequency vector as a list with two elements in it.

```
> [lnM,F]
> end:
```

Now run this function with the file SamplePAM40.

```
> res := EstMutMat( '/home/darwin/v2/source/bio-recipes/SampleAlignments/data.drw', 80, 100, 35,
Sample matches from file /home/darwin/v2/source/bio-recipes/SampleAlignments/data.drw, MinSim=80
PAM bounded between 35 and 45
241 matches read, 21 selected with a total of 6848 positions,
4455 exact matches, 2214 mutations and 179 deletions
```

The PAM of the aggregated sample was 42.6908

The PAM estimate of the sample is slightly lower than expected. We have found that the original Dayhoff matrix tends to overestimate PAM distances.

Now we will compute a new Dayhoff matrix, with PAM 250, so that we can compare it with the original one. Since we have the logarithm of a 1-PAM mutation matrix in `res[1]`, we can use the `CreateDayMatrix` command:

```
> NewD := CreateDayMatrix(res[1],250):
> print( NewD );
```

```
DayMatrix(Peptide, pam=250, Sim: max=14.174, min=-7.500,
max offdiag=5.460, del=-19.814-1.396*(k-1))
C 11.2
S 0.9 2.8
T 0.4 1.2 2.3
P -1.2 0.8 0.6 7.5
A 0.7 1.0 1.1 0.5 2.4
G -1.7 1.0 -0.4 -1.3 0.9 6.5
N -0.9 1.5 0.1 -1.4 -0.5 0.6 3.9
D -3.6 0.4 -0.9 -1.3 -0.6 0.5 2.2 5.4
E -3.1 0.1 -0.8 -0.9 -0.5 -0.2 1.1 4.1 4.4
Q -3.5 0.1 -0.3 0.0 -0.9 -0.8 1.0 0.8 1.8 4.6
H -3.8 -0.1 -1.1 0.3 -1.6 -1.4 2.2 0.3 -0.8 1.8 7.3
R -3.1 -0.1 -0.5 -1.7 -1.1 -1.2 0.2 -0.8 0.4 2.1 0.2 6.1
K -2.7 0.3 0.1 -1.3 -0.7 -1.1 0.8 0.2 1.3 2.3 -0.2 4.2 4.1
M -1.4 -2.4 -0.2 -3.3 -0.7 -3.1 -2.5 -3.6 -2.3 -2.2 -2.9 -2.6 -1.8 4.8
I -0.9 -2.5 0.3 -2.3 -0.4 -4.1 -3.5 -4.5 -3.5 -3.2 -3.1 -3.3 -3.1 2.3
L -2.1 -2.8 -0.7 -2.7 -1.1 -4.9 -3.8 -4.9 -3.7 -2.2 -2.2 -3.0 -2.9 3.2
V 0.6 -1.7 0.5 -1.5 0.4 -2.9 -2.8 -3.2 -2.4 -2.8 -3.1 -3.1 -2.6 1.9
F -2.0 -2.5 -2.7 -4.4 -2.0 -5.5 -1.9 -5.3 -4.8 -4.1 -0.5 -5.5 -4.8 1.2
Y -2.7 -2.0 -3.0 -3.3 -2.6 -5.1 0.0 -3.1 -3.5 -1.9 3.0 -3.3 -3.0 -0.5
W -1.0 -5.5 -4.7 -6.8 -3.8 -3.2 -5.4 -7.5 -6.1 -1.7 -2.0 -5.3 -3.1 0.7
```