Notes on Calculation of the Transition Probability Matrix P(*t*) = exp(*Qt*)

by Ziheng Yang, 20 June 2001

Last modified: April 2003

The test program testPMat.c tests two algorithms in paml for calculating the transition probability matrix $P(t)$ given the rate matrix Q and time (distance) t. The first uses repeated matrix squaring and works on general real rate matrices. It may not be terribly reliable on large distances. It is used for the unrestricted (UNREST) model of nucleotide substitution (Yang, 1994) implemented in baseml. The second algorithm calculates the eigenvalues and eigenvectors of the rate matrix Q and works for the rate matrix for a time-reversible process only. This is used in for the REV (GTR) model in baseml and all amino acid and codon substitution models in codeml. Earlier versions of paml include an algorithm (in the file eigen.c) for calculating the eigenvalues and eigenvectors of a general real matrix, but this is now decommissioned.

Algorithm 1 (routine matexp)

The transition probability matrix $P(t) = e^{Qt}$. Because of the confounding effect of time and rate, we usually rescale Q so that the average rate of change is 1 and then time t is measured by the distance, the expected number of changes per character. Anyway, in the discussion here, time t is the distance. The Taylor expansion gives

$$P(t) = e^{Qt} = I + Qt + \tfrac{1}{2!}(Qt)^2 + \tfrac{1}{3!}(Qt)^3 + \tfrac{1}{4!}(Qt)^4 + \dots . \quad (1)$$

This formula is good to use when t is very small. When t is large, direct use of the formula produces an inefficient algorithm because many terms are needed to achieve reliable accuracy. Since the off-diagonals of Q are positive and the diagonals are positive, with $Q_{ii} = -\sum_{j \neq i} Q_{ij}$, the formula involves rounding errors; in effect we get the probabilities in the range (0, 1) because the positive and negative numbers cancel each other. Algorithm 1 in paml relies on the observation that $e^{Qt} = \left(e^{Qt/m}\right)^m$. We can choose $m = 2^k$, and use two or three terms in equation (1) to approximate $e^{Qt/m}$, and then obtain e^{Qt} by squaring the matrix k times; that is,

$$e^{Qt/m} \approx I + Qt/m + \tfrac{1}{2!}(Qt/m)^2, \quad (2)$$

and

$$e^{Qt} = \left(e^{Qt/m}\right)^m = \left(e^{Qt/2^k}\right)^{2^{2\dots^2}} . \quad (3)$$

The matrix calculated in equation (2) is squared k times. Based on my limited test, k = 5 to 31appears sufficient for distance t up to 10 or 50 changes per character.

Algorithm 2: P(t) for a time-reversible Markov process

Calculation of $P(t)$ according to this algorithm involves two steps. First we get

$$Q = U \Lambda U^{-1}, \tag{4}$$

where matrix $\Lambda = \text{diag}\{\lambda_1, \lambda_{2,}, ..., \lambda_n\}$ has the eigenvalues of Q on the diagonal and 0 elsewhere, while columns of matrix U are the corresponding right eigenvectors and rows of $V = U^{-1}$ are the left eigenvectors. Then

$$P(t) = \exp\{Qt\} = U \exp\{\Lambda t\} U^{-1}, \tag{5}$$

Equation 5 is implemented in paml as the routine PMatUVRoot(). The above two equations apply to general rate matrices and not just reversible matrices. The rate matrix for a reversible process can be written in the form

$$Q = S \Pi \tag{6}$$

where S is symmetrical and $\Pi = \text{diag}\{\pi_1, \pi_2, \ldots, \pi_n\}$ is a diagonal matrix with the nucleotide, amino acid or codon frequencies on the diagonal. Construct a symmetrical matrix A

$$A = \Pi^{1/2} Q \Pi^{1/2} \tag{7}$$

where $\Pi^{1/2} = \text{diag}\left\{\sqrt{\pi_1}, \sqrt{\pi_2}, ..., \sqrt{\pi_n}\right\}$ and $\Pi^{1/2}$ is its inverse. We can use an algorithm for a real symmetrical matrix to get the eigensolution of matrix A and obtain, say,

$$A = R \Lambda R^{-1}, \tag{8}$$

where $\Lambda = \text{diag}\{\lambda_1, \lambda_{2,}, ..., \lambda_n\}$ has the eigenvalues of A on the diagonal, while columns of R are the right eigenvectors of A and rows of $R^{-1} = R'$ are the left eigenvectors. Then the eigenvalues and vectors of Q are given by

$$Q = U \Lambda U^{-1} = (\Pi^{-1/2} R) \Lambda (R^{-1} \Pi^{1/2}). \tag{9}$$

Any function of Q, say, $f(Q) = \exp(Qt)$, can be calculated as

$$f(Q) = U f(\Lambda) U^{-1}. \tag{10}$$

As $U = (\Pi^{1/2} R)$ and its inverse is $V = U^{-1} = (R^{-1} \Pi^{1/2}) = (R' \Pi^{1/2})$, there is no need for matrix inversion.

The above discussion assumes that all character frequencies are nonzero, that is, $\pi_j > 0$ for every j. When some $\pi_j = 0$, the Markov chain really have fewer states. Take the case of codon substitutions as an example and suppose c of the $n = 61$ sense codons have frequency 0. The Markov chain really has $n - c$ states. It seems awkward to have to deal with matrices of size $(n - c) \times (n - c)$ in the code. In codeml I modify the algorithm for calculating the eigenvalues and eigenvectors for the reduced Markov chain (with $n - c$ states) to construct the eigenvalues and eigenvectors for the Markov chain with n states. This involves some copying and moving but seems fine. The rate matrix for the n-state Markov chain is

$$Q = \begin{bmatrix} Q_0 & 0 \\ Q_1 & 0 \end{bmatrix} \tag{11}$$

where Q_0 is $(n - c) \times (n - c)$, and Q_1, of size $c \times (n - c)$, can be constructed as simple as possible. Tim Massingham suggested $Q_1 = 0$. Let the spectral decomposition of Q_0 be

$$Q_0 = U_0 \Lambda_0 V_0 \tag{12}$$

Then

$$Q = \begin{bmatrix} \mathcal{Q}_0 & 0 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} U_0 & 0 \\ 0 & I \end{bmatrix} \begin{bmatrix} \Lambda_0 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} U_0^{-1} & 0 \\ 0 & I \end{bmatrix} \tag{13}$$

$$P(t) = \begin{bmatrix} U_0 & 0 \\ 0 & I \end{bmatrix} \begin{bmatrix} \exp\{\Lambda_0 t\} & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} U_0^{-1} & 0 \\ 0 & I \end{bmatrix} = \begin{bmatrix} U_0 \exp\{\Lambda_0 t\} U_0^{-1} & 0 \\ 0 & I \end{bmatrix} \tag{14}$$

Comparison

Algorithm 1, of squaring matrices, is easy to implement. It seems to work fine especially if you don't have monstrously large distances (branch lengths). Algorithm 2, which calculates the eigensolution of a real symmetric matrix, is fast and stable. Another advantage of this algorithm is that if we need $P(t)$ for different values of t but the same rate matrix Q, a lot of computation can be saved, whereas algorithm 1 does not have such savings.

Table 1 below compares algorithm 2 for calculating the eigensolutions with the old routine for general real matrices (which is not included in paml anymore). This was done a long time ago, and I did not keep a record of the machine configuration. One thing to note is that the general algorithm is not entirely reliable whereas the algorithm for a real symmetrical matrix is mature and stable.

Table 1. Running time (seconds) to calculate the eigenvalues and vectors of 1M random REV rate matrices (results for big numbers in the table are extrapolated as I did not have that much patience)

Matrix size n	Sym algorithm (algorithm 2)	General algorithm (decommissioned eigen.c)
4	10	19
10	74	150
20	350	750
40	1925	4400
61	5800	13800

I also did some comparison between algorithms 1 and 2 described in this document. I think I used $k = 10$ for algorithm 1, but I am not very sure. For matrices smaller than 200 × 200, I didn't see any difference in speed. For matrices of size 500 × 500, algorithm 1 took 17s while algorithm 2 took 5s. Algorithm 2 is preferred as it is faster, but perhaps more importantly, it is more stable.

Notes about code

You can use my code in paml, as long as you acknowledge appropriately. The tools.c file has all the routines involved, but I did not spend the time to extract only the relevant routines. (Is there a program for doing this automatically?) You can look at the small program testPMat.c. The variable TimeSquare is the number of squaring in algorithm 1 (k) above. One thing to note is that I use a 1-D vector to store a matrix so the ijth element of a matrix of size $m \times n$ is referred as A[i*n+j] rather than A[i][j]. If you declare your matrix statically like A[200][200], you can pass &A[0][0] as the address of the matrix to call routines in paml, and the

program should still work. However, if your matrix is allocated dynamically using many alloc or malloc statements, you cannot pass A as an argument to my routines. You should then copy the rate matrix into a vector using my format and then call my routine. Afterwards you can copy the returned P(t) into your matrix.

Reference

Yang, Z. 1994. Estimating the pattern of nucleotide substitution. J. Mol. Evol. 39:105-111.