

Numerical Linear Algebra

Preliminaries

Conditioning and Stability

- Some problems are inherently difficult: no algorithm involving rounding of inputs can be expected to work well. Such problems are called *ill-conditioned*.
- A numerical measure of conditioning, called a *condition number*, can sometimes be defined:
 - Suppose the objective is to compute $y = f(x)$.
 - If x is perturbed by Δx then the result is changed by

$$\Delta y = f(x + \Delta x) - f(x).$$

- If

$$\frac{|\Delta y|}{|y|} \approx \kappa \frac{|\Delta x|}{|x|}$$

for small perturbations Δx then κ is the *condition number* for the problem of computing $f(x)$.

- A particular algorithm for computing an approximation $\tilde{f}(x)$ to $f(x)$ is *numerically stable* if for small perturbations Δx of the input the result is close to $f(x)$.

Error Analysis

- Analyzing how errors accumulate and propagate through a computation, called *forward error analysis*, is sometimes possible but often very difficult.
- *Backward error analysis* tries to show that the computed result

$$\tilde{y} = \tilde{f}(x)$$

is the exact solution to a slightly perturbed problem, i.e.

$$\tilde{y} = f(\tilde{x})$$

for some $\tilde{x} \approx x$.

- If
 - the problem of computing $f(x)$ is well conditioned, and
 - the algorithm \tilde{f} is stable,

then

$\tilde{y} = \tilde{f}(x)$	computed result
$= f(\tilde{x})$	exact result for some $\tilde{x} \approx x$
$\approx f(x)$	since f is well-conditioned

- Backward error analysis is used heavily in numerical linear algebra.

Solving Linear Systems

Many problems involve solving linear systems of the form

$$Ax = b$$

- least squares normal equations:

$$X^T X \beta = X^T y$$

- stationary distribution of a Markov chain:

$$\begin{aligned}\pi P &= \pi \\ \sum \pi_i &= 1\end{aligned}$$

If A is $n \times n$ and non-singular then in principle the solution is

$$x = A^{-1}b$$

This is not usually a good numerical approach because

- it can be numerically inaccurate;
- it is inefficient except for very small n .

Triangular Systems

- Triangular systems are easy to solve.
- The upper triangular system

$$\begin{bmatrix} 5 & 3 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 16 \\ 4 \end{bmatrix}$$

has solution

$$\begin{aligned} x_2 &= 4/2 = 2 \\ x_1 &= (16 - 3x_2)/5 = 10/5 = 2 \end{aligned}$$

- This is called *back substitution*
- Lower triangular systems are solved by *forward substitution*.
- If one of the diagonal elements in a triangular matrix is zero, then the matrix is singular.
- If one of the diagonal elements in a triangular matrix is close to zero, then the solution is very sensitive to other inputs:

$$\begin{bmatrix} 1 & a \\ 0 & \varepsilon \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

has solution

$$\begin{aligned} x_2 &= \frac{b_2}{\varepsilon} \\ x_1 &= b_1 - a \frac{b_2}{\varepsilon} \end{aligned}$$

- This sensitivity for small ε is inherent in the problem: For small values of ε the problem of finding the solution x is ill-conditioned.

Gaussian Elimination

- The system

$$\begin{bmatrix} 5 & 3 \\ 10 & 8 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 16 \\ 36 \end{bmatrix}$$

can be reduced to triangular form by subtracting two times the first equation from the second.

- In matrix form:

$$\begin{bmatrix} 1 & 0 \\ -2 & 1 \end{bmatrix} \begin{bmatrix} 5 & 3 \\ 10 & 8 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ -2 & 1 \end{bmatrix} \begin{bmatrix} 16 \\ 36 \end{bmatrix}$$

or

$$\begin{bmatrix} 5 & 3 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 16 \\ 4 \end{bmatrix}$$

which is the previous triangular system.

- For a general 2×2 matrix A the lower triangular matrix used for the reduction is

$$\begin{bmatrix} 1 & 0 \\ -\frac{a_{21}}{a_{11}} & 1 \end{bmatrix}$$

- The ratio $\frac{a_{21}}{a_{11}}$ is called a *multiplier*.
- This strategy works as long as $a_{11} \neq 0$.
- If $a_{11} \approx 0$, say

$$A = \begin{bmatrix} \varepsilon & 1 \\ 1 & 1 \end{bmatrix}$$

for small ε , then the multiplier $1/\varepsilon$ is large and this does not work very well, even though A is very well behaved.

- Using this approach would result in a numerically unstable algorithm for a well-conditioned problem.

Partial Pivoting

- We can ensure that the multiplier is less than or equal to one in magnitude by switching rows before eliminating:

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 5 & 3 \\ 10 & 8 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 16 \\ 36 \end{bmatrix}$$

or

$$\begin{bmatrix} 10 & 8 \\ 5 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 36 \\ 16 \end{bmatrix}$$

- The matrix to reduce this system to triangular form is now

$$\begin{bmatrix} 1 & 0 \\ -0.5 & 1 \end{bmatrix}$$

- So the final triangular system is constructed as

$$\begin{bmatrix} 1 & 0 \\ -0.5 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 5 & 3 \\ 10 & 8 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ -0.5 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 16 \\ 36 \end{bmatrix}$$

or

$$\begin{bmatrix} 10 & 8 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 36 \\ -2 \end{bmatrix}$$

- Equivalently, we can think of our original system as

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0.5 & 1 \end{bmatrix} \begin{bmatrix} 10 & 8 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 16 \\ 36 \end{bmatrix}$$

- The decomposition of A as

$$A = PLU$$

with P a permutation matrix, L lower triangular with ones on the diagonal, and U upper triangular is called a *PLU decomposition*.

PLU Decomposition

- In general, we can write a square matrix A as

$$A = PLU$$

where

- P is a *permutation matrix*, i.e.
 - * it is an identity matrix with some rows switched
 - * it satisfies $PP^T = P^T P = I$, i.e. it is an *orthogonal matrix*
 - L is a *unit lower triangular matrix*, i.e.
 - * it is lower triangular
 - * it has ones on the diagonal
 - U is upper triangular
- The permutation matrix P can be chosen so that the multipliers used in forming L all have magnitude at most one.
 - A is non-singular if and only if the diagonal entries in U are all non-zero.
 - If A is non-singular, then we can solve

$$Ax = b$$

in three steps:

1. Solve $Pz = b$ for $z = P^T b$ (permute the right hand side)
 2. Solve $Ly = z$ for y (forward solve lower triangular system)
 3. Solve $Ux = y$ for x (back solve upper triangular system)
- Computational complexity:
 - Computing the PLU decomposition takes $O(n^3)$ operations.
 - Computing a solution from a PLU decomposition takes $O(n^2)$ operations.

Condition Number

- Linear systems $Ax = b$ have unique solutions if A is non-singular.
- Solutions are sensitive to small perturbations if A is close to singular.
- We need a useful measure of closeness to singularity
- The *condition number* is a useful measure:

$$\begin{aligned}\kappa(A) &= \frac{\max_{x \neq 0} \frac{\|Ax\|}{\|x\|}}{\min_{x \neq 0} \frac{\|Ax\|}{\|x\|}} \\ &= \left(\max_{x \neq 0} \frac{\|Ax\|}{\|x\|} \right) \left(\max_{x \neq 0} \frac{\|A^{-1}x\|}{\|x\|} \right) \\ &= \|A\| \|A^{-1}\|\end{aligned}$$

where $\|y\|$ is a *vector norm* (i.e. a measure of length) of y and

$$\|B\| = \max_{x \neq 0} \frac{\|Bx\|}{\|x\|}$$

is the corresponding *matrix norm* of B .

- Some common vector norms:

$$\|x\|_2 = \sqrt{\sum_{i=1}^n x_i^2} \quad \text{Euclidean norm}$$

$$\|x\|_1 = \sum_{i=1}^n |x_i| \quad L_1 \text{ norm, Manhattan norm}$$

$$\|x\|_\infty = \max_i |x_i| \quad L_\infty \text{ norm}$$

Some Properties of Condition Numbers

- $\kappa(A) \geq 1$ for all A .
- $\kappa(A) = \infty$ if A is singular
- If A is diagonal, then

$$\kappa(A) = \frac{\max |a_{ii}|}{\min |a_{ii}|}$$

- Different norms produce different values; the values are usually qualitatively similar

Sensitivity of Linear Systems

Suppose x solves the original system and x^* solves a slightly perturbed system,

$$(A + \Delta A)x^* = b + \Delta b$$

and suppose that

$$\begin{aligned} \delta \kappa(A) &\leq \frac{1}{2} \\ \frac{\|\Delta A\|}{\|A\|} &\leq \delta \\ \frac{\|\Delta b\|}{\|b\|} &\leq \delta \end{aligned}$$

Then

$$\frac{\|x - x^*\|}{\|x\|} \leq 4\delta \kappa(A)$$

Stability of Gaussian Elimination with Partial Pivoting

Backward error analysis: The numerical solution \hat{x} to the system

$$Ax = b$$

produced by Gaussian elimination with partial pivoting is the exact solution for a perturbed system

$$(A + \Delta A)\hat{x} = b$$

with

$$\frac{\|\Delta A\|_\infty}{\|A\|_\infty} \leq 8n^3 \rho \mathbf{u} + O(\mathbf{u}^2)$$

- The value of ρ is not *guaranteed* to be small, but is rarely larger than 10
- The algorithm would be considered numerically stable if ρ were guaranteed to be bounded.
- *Complete pivoting* is a bit more stable, but much more work.
- The algorithm is considered very good for practical purposes.

General Linear Systems in R

R provides

- `solve` for general systems, based on LAPACK's DGESV.
- DGESV uses the *PLU* decomposition.
- `forwardsolve`, `backsolve` for triangular systems.
- `kappa` computes an estimate of the condition number or the exact condition number based on the Euclidean norm.

Cholesky Factorization

Suppose A is symmetric and (strictly) positive definite, i.e.

$$x^T A x > 0$$

for all $x \neq 0$. Examples:

- If X is the $n \times p$ design matrix for a linear model and X is of rank p , then $A = X^T X$ is strictly positive definite.

If X is not of full rank then $A = X^T X$ is non-negative definite or positive semi-definite, i.e. $x^T A x \geq 0$ for all x .

- If A is the covariance matrix of a random vector X then A is positive semidefinite:

$$\begin{aligned} c^T A c &= c^T E[(X - \mu)(X - \mu)^T] c \\ &= E[((X - \mu)^T c)^T (X - \mu)^T c] \\ &= \text{Var}((X - \mu)^T c) \geq 0 \end{aligned}$$

The covariance matrix is strictly positive definite unless $P(c^T X = c^T \mu) = 1$ for some $c \neq 0$, i.e. unless there is a perfect linear relation between some of the components of X .

Theorem

If A is strictly positive definite, then there exists a unique lower triangular matrix L with positive diagonal entries such that

$$A = LL^T$$

This is called the *Cholesky factorization*.

Properties of the Cholesky Factorization Algorithm

- It uses the symmetry to produce an efficient algorithm.
- The algorithm needs to take square roots to find the diagonal entries.
- An alternative that avoids square roots factors A as

$$A = LDL^T$$

with D diagonal and L unit lower triangular.

- The algorithm is numerically stable, and is guaranteed not to attempt square roots of negative numbers if

$$q_n \mathbf{u}\kappa_2(A) \leq 1$$

where q_n is a small constant depending on the dimension n .

- The algorithm will fail if the matrix is not (numerically) strictly positive definite.
- Modifications using pivoting are available that can be used for nonnegative definite matrices.
- Another option is to factor $A_\lambda = A + \lambda I$ with $\lambda > 0$ chosen large enough to make A_λ numerically strictly positive definite. This is often used in optimization.

Some Applications of the Cholesky Factorization

- Solving the normal equations in least squares. This requires that the predictors be linearly independent
- Generating multivariate normal random vectors.
- Parameterizing strictly positive definite matrices: Any lower triangular matrix L with arbitrary values below the diagonal and positive diagonal entries determines and is uniquely determined by the positive definite matrix $A = LL^T$

Cholesky Factorization in R

- The function `chol` computes the Cholesky factorization.
- The returned value is the upper triangular matrix $R = L^T$.
- LAPACK is used.

QR Factorization

An $m \times n$ matrix A with $m \geq n$ can be written as

$$A = QR$$

where

- Q is $m \times n$ with orthonormal columns, i.e. $Q^T Q = I_n$
- R is upper triangular
- Several algorithms are available for computing the QR decomposition:
 - Modified Gram-Schmidt
 - Householder transformations (reflections)
 - Givens transformations (rotations)

Each has advantages and disadvantages.

- LINPACK `dqrdc` and LAPACK `DGEQP3` use Householder transformations.
- The QR decomposition exists regardless of the rank of A .
- The rank of A is n if and only if the diagonal elements of R are all non-zero.

Householder Transformations

- A Householder transformation is a matrix of the form

$$P = I - 2vv^T / v^T v$$

where v is a nonzero vector.

- Px is the reflection of x in the hyperplane orthogonal to v .
- Given a vector $x \neq 0$, choosing $v = x + \alpha e_1$ with

$$\alpha = \pm \|x\|_2$$

and e_1 the first unit vector (first column of the identity) produces

$$Px = \mp \|x\|_2 e_1$$

This can be used to zero all but the first element of the first column of a matrix:

$$P \begin{bmatrix} \times & \times & \times \\ \times & \times & \times \end{bmatrix} = \begin{bmatrix} \times & \times & \times \\ 0 & \times & \times \end{bmatrix}$$

This is the first step in computing the QR factorization.

- The denominator $v^T v$ can be written as

$$v^T v = x^T x + 2\alpha x_1 + \alpha^2$$

- Choosing $\alpha = \text{sign}(x_1) \|x\|_2$ ensures that all terms are non-negative and avoids cancellation.
- With the right choice of sign Householder transformations are very stable.

Givens Rotations

- A Givens rotation is a matrix G that is equal to the identity except for elements $G_{ii}, G_{ij}, G_{ji}, G_{jj}$, which are

$$\begin{bmatrix} G_{ii} & G_{ij} \\ G_{ji} & G_{jj} \end{bmatrix} = \begin{bmatrix} c & s \\ -s & c \end{bmatrix}$$

with $c = \cos(\theta)$ and $s = \sin(\theta)$ for some θ .

- Premultiplication by G^T is a clockwise rotation by θ radians in the (i, j) coordinate plane.
- Given scalars a, b one can compute c, s so that

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix}^T \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}$$

This allows G to zero one element while changing only one other element.

- A stable way to choose c, s :

```

if  $b = 0$ 
   $c = 1; s = 0$ 
else
  if  $|b| > |a|$ 
     $\tau = -a/b; s = 1/\sqrt{1 + \tau^2}; c = s\tau$ 
  else
     $\tau = -b/a; c = 1/\sqrt{1 + \tau^2}; s = c\tau$ 
  end
end

```

- A sequence of Givens rotations can be used to compute the QR factorization.
 - The zeroing can be done working down columns or across rows.
 - Working across rows is useful for incrementally adding more observations.

Applications

- The QR decomposition can be used for solving $n \times n$ systems of equations

$$Ax = b$$

since $Q^{-1} = Q^T$ and so

$$Ax = QRx = b$$

is equivalent to the upper triangular system

$$Rx = Q^T b$$

- The QR decomposition can also be used to solve the normal equations in linear regression: If X is the $n \times p$ design matrix then the normal equations are

$$X^T X b = X^T y$$

If $X = QR$ is the QR decomposition of X , then

$$X^T X = R^T Q^T QR = R^T R$$

$$X^T y = R^T Q^T y$$

If X is of full rank then R^T is invertible, and the normal equations are equivalent to the upper triangular system

$$Rb = Q^T y$$

This approach avoids computing $X^T X$.

- If X is of full rank then $R^T R$ is the Cholesky factorization of $X^T X$ (up to multiplications of rows of R by ± 1).

QR with Column Pivoting

Sometimes the columns of X are linearly dependent or nearly so.

By permuting columns we can produce a factorization

$$A = QRP$$

where

- P is a permutation matrix
- R is upper triangular and the diagonal elements of R have non-increasing magnitudes, i.e.

$$|r_{ii}| \geq |r_{jj}|$$

if $i \leq j$

- If some of the diagonal entries of R are zero, then R will be of the form

$$R = \begin{bmatrix} R_{11} & R_{12} \\ 0 & 0 \end{bmatrix}$$

where R_{11} is upper triangular with non-zero diagonal elements non-increasing in magnitude.

- The rank of the matrix is the number of non-zero rows in R .
- The *numerical rank* of a matrix can be determined by
 - computing its QR factorization with column pivoting
 - specifying a tolerance level ϵ such that all diagonal entries $|r_{ii}| < \epsilon$ are considered numerically zero.
 - Modifying the computed QR factorization to zero all rows corresponding to numerically zero r_{ii} values.

Some Regression Diagnostics

The projection matrix, or hat matrix, is

$$H = X(X^T X)^{-1} X^T = QR(R^T R)^{-1} R^T Q^T = QQ^T$$

The diagonal elements of the hat matrix are therefore

$$h_i = \sum_{j=1}^p q_{ij}^2$$

If $\hat{e}_i = y_i - \hat{y}_i$ is the residual, then

$$s_{-i}^2 = \frac{\text{SSE} - \hat{e}_i^2 / (1 - h_i)}{n - p - 1} = \text{estimate of variance without obs. } i$$

$$t_i = \frac{\hat{e}_i}{s_{-i} \sqrt{1 - h_i}} = \text{externally studentized residual}$$

$$D_i = \frac{\hat{e}_i^2 h_i}{(1 - h_i)^2 s^2 p} = \text{Cook's distance}$$

QR Decomposition and Least Squares in R

- The R function `qr` uses either LINPACK or LAPACK to compute QR factorizations.
- LINPACK is the default.
- The core linear model fitting function `lm.fit` uses QR factorization with column pivoting.

Singular Value Decomposition

An $m \times n$ matrix A with $m \geq n$ can be factored as

$$A = UDV^T$$

where

- U is $m \times n$ with orthonormal columns, i.e. $U^T U = I_n$.
- V is $n \times n$ orthogonal, i.e. $VV^T = V^T V = I_n$.
- $D = \text{diag}(d_1, \dots, d_n)$ is $n \times n$ diagonal with $d_1 \geq d_2 \geq \dots \geq d_n \geq 0$.

This is the *singular value decomposition*, or *SVD* of A .

- The values d_1, \dots, d_n are the *singular values* of A .
- The columns of U are the *right singular vectors* of A .
- The columns of V are the *left singular vectors* of A .
- If the columns of A have been centered so the column sums of A are zero, then the columns of UD are the *principal components* of A .
- Excellent algorithms are available for computing the SVD.
- These algorithms are usually several times slower than the QR algorithms.

Some Properties of the SVD

- The Euclidean matrix norm of A is defined as

$$\|A\|_2 = \max_{x \neq 0} \frac{\|Ax\|_2}{\|x\|_2}$$

with $\|x\|_2 = \sqrt{x^T x}$ the Euclidean vector norm.

- If A has SVD $A = UDV^T$, then

$$\|A\|_2 = d_1$$

- If $k < \text{rank}(A)$ and

$$A_k = \sum_{i=1}^k d_i u_i v_i^T$$

then

$$\min_{B: \text{rank}(B) \leq k} \|A - B\|_2 = \|A - A_k\|_2 = d_{k+1}$$

In particular,

- $d_1 u_1 v_1^T$ is the best rank one approximation to A (in the Euclidean matrix norm).
 - A_k is the best rank k approximation to A .
 - If $m = n$ then $d_n = \min\{d_1, \dots, d_n\}$ is the distance between A and the set of singular matrices.
- If A is square then the condition number based on the Euclidean norm is

$$\kappa_2(A) = \|A\|_2 \|A^{-1}\|_2 = \frac{d_1}{d_n}$$

- For an $n \times p$ matrix with $n > p$ we also have

$$\kappa_2(A) = \frac{\max_{x \neq 0} \frac{\|Ax\|_2}{\|x\|_2}}{\min_{x \neq 0} \frac{\|Ax\|_2}{\|x\|_2}} = \frac{d_1}{d_n}$$

- This can be used to relate $\kappa_2(A^T A)$ to $\kappa_2(A)$.
 - This has implications for regression computations.
- The singular values are the non-negative square roots of the eigenvalues of $A^T A$ and the columns of V are the corresponding eigenvectors.

Moore-Penrose Generalized Inverse

Suppose A has rank $r \leq n$ and SVD $A = UDV^T$. Then

$$d_{r+1} = \cdots = d_n = 0$$

Let

$$D^+ = \text{diag} \left(\frac{1}{d_1}, \dots, \frac{1}{d_r}, 0, \dots, 0 \right)$$

and

$$A^+ = VD^+U^T$$

Then A^+ satisfies

$$AA^+A = A$$

$$A^+AA^+ = A^+$$

$$(AA^+)^T = AA^+$$

$$(A^+A)^T = A^+A$$

A^+ is the unique matrix with these properties and is called the Moore-Penrose *generalized inverse* or *pseudo-inverse*.

SVD and Least Squares

If X is an $n \times p$ design matrix of less than full rank, then there are infinitely many values of b that minimize

$$\|y - Xb\|_2^2$$

Among these solutions,

$$b = (X^T X)^+ X^T y$$

minimizes $\|b\|_2$.

This is related to *penalized regression* where one might choose b to minimize

$$\|y - Xb\|_2^2 + \lambda \|b\|_2^2$$

for some choice of $\lambda > 0$.

SVD and Principal Components Analysis

- Let X be an $n \times p$ matrix of n observations on p variables.
- Principal components analysis involves estimating the eigenvectors and eigenvalues of the covariance matrix.
- Let \tilde{X} be the data matrix with columns centered at zero by subtracting the column means.
- The sample covariance matrix is

$$S = \frac{1}{n-1} \tilde{X}^T \tilde{X}$$

- Let $\tilde{X} = UDV^T$ be the SVD of the centered data matrix \tilde{X} .

- Then

$$S = \frac{1}{n-1} VDU^TUDV^T = \frac{1}{n-1} VD^2V^T$$

- So

- The diagonal elements of $\frac{1}{n-1}D^2$ are the eigenvalues of S .
- The columns of V are the eigenvectors of S .

- Using the SVD of \tilde{X} is more numerically stable than
 - forming $\tilde{X}^T \tilde{X}$
 - computing the eigenvalues and eigenvectors.

SVD and Numerical Rank

- The rank of a matrix A is equal to the number of non-zero singular values.
- Exact zeros may not occur in the SVD due to rounding.
- Numerical rank determination can be based on the SVD. All $d_i \leq \delta$ can be set to zero for some choice of δ . Golub and van Loan recommend using

$$\delta = \mathbf{u} \|A\|_\infty$$

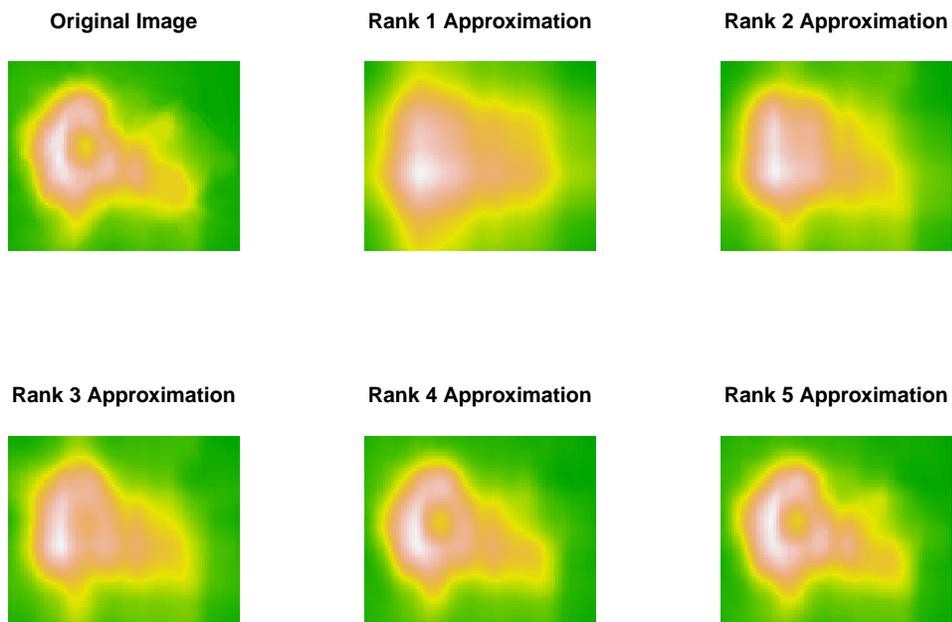
- If the entries of A are only accurate to d decimal digits, then Golub and van Loan recommend

$$\delta = 10^{-d} \|A\|_\infty$$

- If the numerical rank of A is \hat{r} and $d_{\hat{r}} \gg \delta$ then \hat{r} can be used with some confidence; otherwise caution is needed.

Other Applications

- The SVD is used in many areas of numerical analysis.
- It is also often useful as a theoretical tool.
- Some approaches to compressing $m \times n$ images are based on the SVD.
- A simple example using the `volcano` data:



```
head(s$d, 10)

## [1] 9644.28782 488.60992 341.18358 298.76602 141.83363
## [6] 72.12443 43.55698 33.52319 27.38376 19.97622

tail(s$d, 2)

## [1] 1.0526941 0.9545092
```

SVD in R

- R provides the function `svd` to compute the SVD.
- Implementation used to use LINPACK but now can use LINPACK or LAPACK, with LAPACK the default.
- You can ask for the singular values only—this is will be faster for larger problems.

Eigenvalues and Eigenvectors

Let A be an $n \times n$ matrix. λ is an *eigenvalue* of A if

$$Av = \lambda v$$

for some $v \neq 0$; v is an *eigenvector* of A .

- If A is a real $n \times n$ matrix then it has n eigenvalues.
 - Several eigenvalues may be identical
 - Some eigenvalues may be complex; if so, then they come in conjugate pairs.
 - The set of eigenvalues is called the *spectrum*
- If A is symmetric then the eigenvalues are real
- If A is symmetric then
 - A is strictly positive definite if and only if all eigenvalues are positive.
 - A is positive semi-definite if and only if all eigenvalues are non-negative.
 - There exists an orthogonal matrix V such that

$$A = V\Lambda V^T$$
 with $\Lambda = \text{diag}(\lambda_1, \dots, \lambda_n)$; the columns of V are the corresponding normalized eigenvectors.
 - This is called the *spectral decomposition* of A .
- Some problems require only the largest eigenvalue or the largest few, sometimes the corresponding eigenvectors are also needed.
 - The stationary distribution of an irreducible finite state-space Markov chain is the unique eigenvector, normalized to sum to one, corresponding to the largest eigenvalue $\lambda = 1$.
 - The speed of convergence to the stationary distribution depends on the magnitude of the second largest eigenvalue.
- The R function `eigen` can be used to compute eigenvalues and, optionally, eigenvectors.

Determinants

- Theoretically the determinant can be computed as the product of
 - the diagonals of U in the PLU decomposition
 - the squares of the diagonals of L in the Cholesky factorization
 - the diagonals of R in the QR decomposition
 - the eigenvalues
- Numerically these are almost always bad ideas.
- It is almost always better to work out the sign and compute the sum of the logarithms of the magnitudes of the factors.
- The R functions `det` and `determinant` compute the determinant.
 - `determinant` is more complicated to use, but has a `logarithm` option.
- Likelihood and Bayesian analyses often involve a determinant;
 - usually the log likelihood and log determinant should be used.
 - usually the log determinant can be computed from a decomposition needed elsewhere in the log likelihood calculation, e.g. a Cholesky factorization

Non-Negative Matrix Factorization

A number of problems lead to the desire to approximate a non-negative matrix X by a product

$$X \approx WH$$

where W, H are non-negative matrices of low rank, i.e. with few columns.

There are a number of algorithms available, most of the form

$$\min_{W,H} [D(X, WH) + R(W, H)]$$

where D is a loss function and R is a possible penalty for encouraging desirable characteristics of W, H , such as smoothness or sparseness.

The R package NMF provides one approach, and a vignette in the package provides some background and references.

As an example, using default settings in the NMF package the `volcano` image can be approximated with factorizations of rank $1, \dots, 5$ by

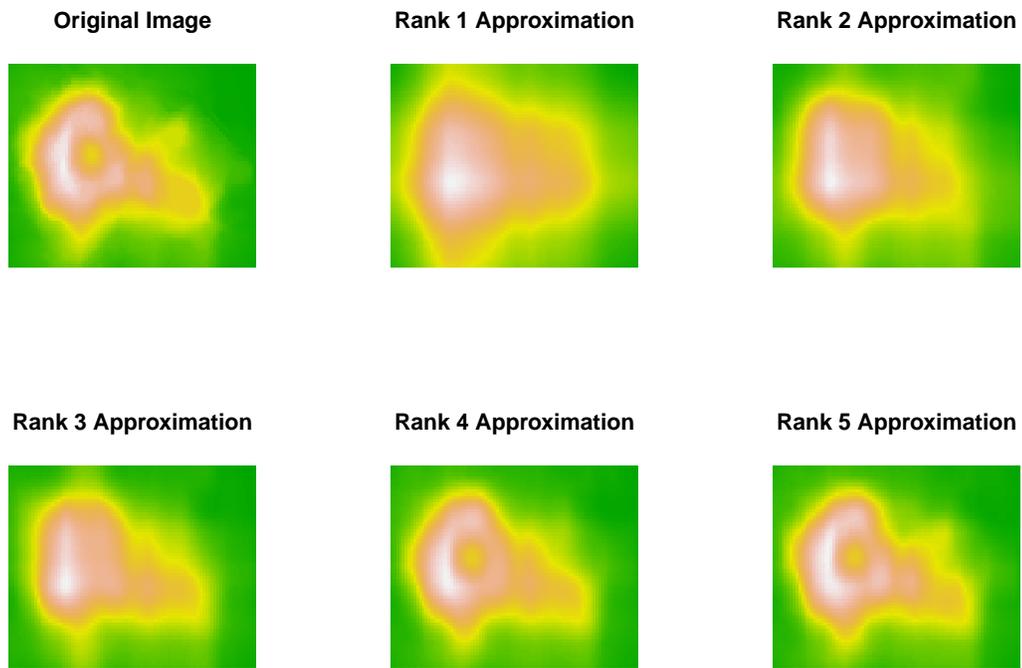
```
library(NMF)
nmf1 = nmf(volcano, 1); V1 <- nmf1@fit@W %*% nmf1@fit@H
nmf2 = nmf(volcano, 2); V2 <- nmf2@fit@W %*% nmf2@fit@H
nmf3 = nmf(volcano, 3); V3 <- nmf3@fit@W %*% nmf3@fit@H
nmf4 = nmf(volcano, 4); V4 <- nmf4@fit@W %*% nmf4@fit@H
nmf5 = nmf(volcano, 5); V5 <- nmf5@fit@W %*% nmf5@fit@H
```

The relative error for the final image is

```
max(abs(volcano - V5)) / max(abs(volcano))

## [1] 0.03096702
```

The images:



Another application is *recommender systems*.

- For example, X might be ratings of movies (columns) by viewers (rows).
- The set of actual values would be very sparse as each viewer will typically rate only a small subset of all movies.
- W would be a user preference matrix, H a corresponding movie feature matrix.
- The product WH would provide predicted ratings for movies the users have not yet seen.

Other Factorizations

Many other factorizations of matrices are available and being developed. Some examples are

- Robust variants of the SVD
- Sparse variants, e.g. Dan Yang, Zongming Ma, and Andreas Buja (2014), “A Sparse Singular Value Decomposition Method for High-Dimensional Data,” *Journal of Computational and Graphical Statistics* 23(4), 923–942.
- Constrained factorizations, e.g. C. Ding, T. Li, and M. I. Jordan (2010), “Convex and Semi-Nonnegative Matrix Factorizations,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(1), 45–55.

Exploiting Special Structure

Specialized algorithms can sometimes be used for matrices with special structure.

Toeplitz Systems

- Stationary time series have covariance matrices that look like

$$\begin{bmatrix} \sigma_0 & \sigma_1 & \sigma_2 & \sigma_3 & \dots \\ \sigma_1 & \sigma_0 & \sigma_1 & \sigma_2 & \dots \\ \sigma_2 & \sigma_1 & \sigma_0 & \sigma_1 & \dots \\ \sigma_3 & \sigma_2 & \sigma_1 & \sigma_0 & \ddots \\ \ddots & \ddots & \ddots & \ddots & \ddots \end{bmatrix}$$

- This is a *Toeplitz* matrix.
- This matrix is also symmetric — this is not required for a Toeplitz matrix.
- Special algorithms requiring $O(n^2)$ operations are available for Toeplitz systems.
- General Cholesky factorization requires $O(n^3)$ operations.
- The R function `toeplitz` creates Toeplitz matrices.

Circulant Systems

- Some problems give rise to matrices that look like

$$C_n = \begin{bmatrix} a_1 & a_2 & a_3 & \dots & a_n \\ a_n & a_1 & a_2 & \dots & a_{n-1} \\ a_{n-1} & a_n & a_1 & \dots & a_{n-2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_2 & a_3 & a_4 & \dots & a_1 \end{bmatrix}$$

- This is a *circulant* matrix, a subclass of Toeplitz matrices.
- Circulant matrices satisfy

$$C_n = F_n^* \text{diag}(\sqrt{n}F_n a) F_n$$

where F_n is the *Fourier matrix* with

$$F_n(j, k) = \frac{1}{\sqrt{n}} e^{-i(j-1)(k-1)2\pi/n}$$

and F_n^* is the *conjugate transpose*, *Hermitian transpose*, or *adjoint matrix* of F_n .

- The eigen values are the elements of $\sqrt{n}F_n a$.
- Products $F_n x$ and $F_n^* x$ can be computed with the *fast Fourier transform (FFT)*.
- In R

$$\begin{aligned} \sqrt{n}F_n x &= \text{fft}(x) \\ \sqrt{n}F_n^* x &= \text{fft}(x, \text{inverse} = \text{TRUE}) \end{aligned}$$

- These computations are generally $O(n \log n)$ in complexity.
- Circulant systems can be used to approximate other systems.
- Multi-dimensional analogs exist as well.
- A simple example is available on line.

Sparse Systems

- Many problems lead to large systems in which only a small fraction of coefficients are non-zero.
- Some methods are available for general sparse systems.
- Specialized methods are available for structured sparse systems such as
 - tri-diagonal systems
 - block diagonal systems
 - banded systems
- Careful choice of row and column permutations can often turn general sparse systems into banded ones.

Sparse and Structured Systems in R

- Sparse matrix support in R is improving.
- Some packages, like `nlme`, provide utilities they need.
- One basic package available on CRAN is `sparseM`
- A more extensive package is `Matrix`
- `Matrix` is the engine for mixed effects/multi-level model fitting in `lme4`

Update Formulas

- Update formulas are available for most decompositions that allow for efficient adding or dropping of rows or columns.
- These can be useful for example in cross-validation and variable selection computations.
- They can also be useful for fitting linear models to very large data sets; the package `biglm` uses this approach.
- I am not aware of any convenient implementations in R at this point but they may exist.

Iterative Methods

- Iterative methods can be useful in large, sparse problems.
- Iterative methods for sparse problems can also often be parallelized effectively.
- Iterative methods are also useful when
 - Ax can be computed efficiently for any given x
 - It is expensive or impossible to compute A explicitly

Gauss-Seidel Iteration

Choose an initial solution $x^{(0)}$ to

$$Ax = b$$

and then update from $x^{(k)}$ to $x^{(k+1)}$ by

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^n a_{ij}x_j^{(k)} \right)$$

for $i = 1, \dots, n$.

This is similar in spirit to Gibbs sampling.

This can be written in matrix form as

$$x^{(k+1)} = (D + L)^{-1}(-Ux^{(k)} + b)$$

with

$$L = \begin{bmatrix} 0 & 0 & \dots & \dots & 0 \\ a_{21} & 0 & \dots & & \vdots \\ a_{31} & a_{32} & \ddots & & 0 \\ \vdots & & & 0 & 0 \\ a_{n1} & a_{n2} & \dots & a_{n,n-1} & 0 \end{bmatrix}$$

$$D = \text{diag}(a_{11}, \dots, a_{nn})$$

$$U = \begin{bmatrix} 0 & a_{12} & \dots & \dots & a_{1n} \\ 0 & 0 & \dots & & \vdots \\ 0 & 0 & \ddots & & a_{n-2,n} \\ \vdots & & & & a_{n-1,n} \\ 0 & 0 & \dots & 0 & 0 \end{bmatrix}$$

Splitting Methods

The Gauss-Seidel method is a member of a class of *splitting methods* where

$$Mx^{(k+1)} = Nx^{(k)} + b$$

with $A = M - N$.

For the Gauss-Seidel method

$$M = D + L$$

$$N = -U.$$

Other members include Jacobi iterations with

$$M_J = D$$

$$N_J = -(L + U)$$

Splitting methods are practical if solving linear systems with matrix M is easy.

Convergence

A splitting method for a non-singular matrix A will converge to the unique solution of $Ax = b$ if

$$\rho(M^{-1}N) < 1$$

where

$$\rho(G) = \max\{|\lambda| : \lambda \text{ is an eigenvalue of } G\}$$

is the *spectral radius* of G .

This is true, for example, for the Gauss-Seidel method if A is strictly positive definite.

Convergence can be very slow if $\rho(M^{-1}N)$ is close to one.

Successive Over-Relaxation

A variation is to define

$$x_i^{(k+1)} = \frac{\omega}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^n a_{ij}x_j^{(k)} \right) + (1 - \omega)x_i^{(k)}$$

or, in matrix form,

$$M_\omega x^{(k+1)} = N_\omega x^{(k)} + \omega b$$

with

$$M_\omega = D + \omega L$$

$$N_\omega = (1 - \omega)D - \omega U$$

for some ω , usually with $0 < \omega < 1$.

- This is called *successive over-relaxation (SOR)*, from its first application in a structural engineering problem.
- For some choices of ω we can have

$$\rho(M_\omega^{-1}N_\omega) \ll \rho(M^{-1}N)$$

and thus faster convergence.

- For some special but important problems the value of ω that minimizes $\rho(M_\omega^{-1}N_\omega)$ is known or can be computed.

Conjugate Gradient Method

If A is symmetric and strictly positive definite then the unique solution to $Ax = b$ is the unique minimizer of the quadratic function

$$f(x) = \frac{1}{2}x^T Ax - x^T b$$

Any nonlinear or quadratic optimization method can be used to find the minimum; the most common one used in this context is the conjugate gradient method.

Choose an initial x_0 , set $d_0 = -g_0 = b - Ax_0$, and then, while $g_k \neq 0$, for $k = 0, 1, \dots$ compute

$$\begin{aligned}\alpha_k &= -\frac{g_k^T d_k}{d_k^T A d_k} \\ x_{k+1} &= x_k + \alpha_k d_k \\ g_{k+1} &= Ax_{k+1} - b \\ \beta_{k+1} &= \frac{g_{k+1}^T A d_k}{d_k^T A d_k} \\ d_{k+1} &= -g_{k+1} + \beta_{k+1} d_k\end{aligned}$$

Some properties:

- An alternate form of g_{k+1} is

$$g_{k+1} = g_k + \alpha_k A d_k$$

This means only one matrix-vector multiplication is needed per iteration.

- The vector g_k is the gradient of f at x_k .
- The initial direction $d_0 = -g_0$ is the *direction of steepest descent* from x_0
- The directions d_0, d_1, \dots are *A-conjugate*, i.e. $d_i^T A d_j = 0$ for $i \neq j$.
- The directions d_0, d_1, \dots, d_{n-1} are linearly independent.

Convergence

- With exact arithmetic,

$$Ax_n = b$$

That is, the conjugate gradient algorithm terminates with the exact solution in n steps.

- Numerically this does not happen.
- Numerically, the directions will not be exactly A -conjugate.
- A convergence tolerance is used for termination; this can be based on the relative change in the solution

$$\frac{\|x_{k+1} - x_k\|}{\|x_k\|}$$

or the residual or gradient

$$g_k = Ax_k - b$$

or some combination; an iteration count limit is also a good idea.

- If the algorithm does not terminate within n steps it is a good idea to restart it with a steepest descent step from the current x_k .
- In many sparse and structured problems the algorithm will terminate in far fewer than n steps for reasonable tolerances.
- Convergence is faster if the condition number of A is closer to one. The error can be bounded as

$$\|x - x_k\|_A \leq 2\|x - x_0\|_A \left(\frac{\sqrt{\kappa_2(A)} - 1}{\sqrt{\kappa_2(A)} + 1} \right)^k$$

with $\|x\|_A = \sqrt{x^T A x}$.

- *Preconditioning* strategies can improve convergence; these transform the original problem to one with $\tilde{A} = C^{-1}AC^{-1}$ for some symmetric strictly positive definite C , and then use the conjugate gradient method for \tilde{A}
- Simple choices of C are most useful; sometimes a diagonal matrix will do.
- Good preconditioners can sometimes be designed for specific problems.

A Simple Implementation

```
cg <- function(A, b, x, done) {  
  dot <- function(x, y) crossprod(x, y)[1]  
  
  n <- 0  
  g <- A(x) - b  
  d <- -g  
  
  repeat {  
    h <- A(d)  
    u <- dot(d, h)  
    a <- -dot(g, d) / u  
  
    n <- n + 1  
    x.old <- x  
    x <- x + a * d  
    g <- g + a * h  
  
    b <- dot(h, g) / u  
    d <- -g + b * d  
    if (done(g, x, x.old, n))  
      return(list(x = as.vector(x),  
                  g = as.vector(g),  
                  n = n))  
  }  
}
```

- The linear transformation and the termination condition are specified as functions.
- The termination condition can use a combination of the gradient, current solution, previous solution, or iteration count.

A simple example:

```
X <- crossprod(matrix(rnorm(25), 5))
y <- rnorm(5)
cg(A = function(x) X %*% x,
  b = y,
  x = rep(0, 5),
  done = function(g, x, x.old, n) n >= 5)

## $x
## [1] -1.0829111 -0.0410215  0.1092576 -1.1596751 -0.4051707
##
## $g
## [1]  1.276756e-15 -2.151057e-15 -1.276756e-15 -3.178013e-15
## [5]  1.221245e-15
##
## $n
## [1] 5

solve(X, y)

## [1] -1.0829111 -0.0410215  0.1092576 -1.1596751 -0.4051707
```

A simple conjugate gradient based solve function might be defined as

```
cgsolve <- function(X, y, tol =sqrt(.Machine$double.eps)) {
  A <- function(x)
    X %*% x
  done <- function(g, x, x.old, n)
    n >= length(x) ||
    (max(abs(x - x.old) / (tol + abs(x.old))) < tol &&
     max(abs(g)) < tol)
  x <- rep(0, length(y))
  v <- cg(A, y, x, done = done)
  if (v$n >= length(y))
    message("iteration limit reached")
  else
    message(v$n, " iterations")
  v$x
}
```

With a positive definite matrix with a large diagonal component `cgsolve` can be very effective:

```
N <- 1000
X <- crossprod(matrix(rnorm(N * N), N))
XX <- X / N + 5 * diag(N)
y <- rnorm(N)

system.time(v0 <- solve(XX, y))

##      user  system elapsed
##    0.167    0.000    0.167

system.time(v <- cgsolve(XX, y))

## 14 iterations

##      user  system elapsed
##    0.03    0.00    0.03

max(abs((v - v0) / v0))

## [1] 2.215307e-09
```

Linear Algebra Software

Some Standard Packages

Open source packages developed at national laboratories:

- LINPACK for linear equations and least squares
- EISPACK for eigenvalue problems
- LAPACK newer package for linear equations and eigenvalues

Designed for high performance. Available from Netlib at

`http://www.netlib.org/`

Commercial packages:

- IMSL used more in US
- NAG used more in UK
- ...

BLAS: Basic Linear Algebra Subroutines

Modern BLAS has three levels:

Level 1: Vector and vector/vector operations such as

- dot product $x^T y$
- scalar multiply and add (axpy): $\alpha x + y$
- Givens rotations

Level 2: Matrix/vector operations, such as Ax

Level 3: Matrix/matrix operations, such as AB

- LINPACK uses only Level 1; LAPACK uses all three levels.
- BLAS defines the interface.
- Standard reference implementations are available from Netlib.
- Highly optimized versions are available from hardware vendors and research organizations.

Cholesky Factorization in LAPACK

The core of the DPOTRF routine:

```

*
*      Compute the Cholesky factorization A = L*L'.
*
*      DO 20 J = 1, N
*
*          Compute L(J,J) and test for non-positive-definiteness.
*
*          AJJ = A( J, J ) - DDOT( J-1, A( J, 1 ), LDA, A( J, 1 ),
$              LDA )
*          IF( AJJ.LE.ZERO ) THEN
*              A( J, J ) = AJJ
*              GO TO 30
*          END IF
*          AJJ = SQRT( AJJ )
*          A( J, J ) = AJJ
*
*          Compute elements J+1:N of column J.
*
*          IF( J.LT.N ) THEN
*              CALL DGEMV( 'No transpose', N-J, J-1, -ONE, A( J+1, 1 ),
$                  LDA, A( J, 1 ), LDA, ONE, A( J+1, J ), 1 )
*              CALL DSCAL( N-J, ONE / AJJ, A( J+1, J ), 1 )
*          END IF
20      CONTINUE

```

- **DDOT** and **DSCAL** are Level 1 BLAS routines
- **DGEMV** is a Level 2 BLAS routine

ATLAS: Automatically Tuned Linear Algebra Software

Available at

`http://math-atlas.sourceforge.net/`

- Analyzes machine for properties such as cache characteristics.
- Runs extensive tests to determine performance trade-offs.
- Creates Fortran and C versions of BLAS and some LAPACK routines tailored to the particular machine.
- Provides some routines that take advantage of multiple processors using *worker threads*.

OpenBLAS

- Another high-performance BLAS library was developed and maintained by Kazushige Goto.
- This is now being developed and maintained as the OpenBLAS project, available from

`http://xianyi.github.com/OpenBLAS/`

- Also provides versions that take advantage of multiple processors.

Vendor Libraries

- Intel provides the Math Kernel Libraries (MKL)
- AMD has a similar library.

Using a High-Performance BLAS with R

- R comes with a basic default BLAS.
- R can be built to use a specified BLAS.
- Once built one can change the BLAS R uses by replacing the shared library R uses.
- Some simple computations using the default and MKL vendor BLAS for the data

```
N <- 1000
X <- matrix(rnorm(N^2), N)
XX <- crossprod(X)
```

Results:

Timing Expression	Default/ Reference	MKL SEQ	MKL THR
<code>system.time(for (i in 1:5) crossprod(X))</code>	2.107	0.405	0.145
<code>system.time(for (i in 1:5) X %*% X)</code>	3.401	0.742	0.237
<code>system.time(svd(X))</code>	3.273	0.990	0.542
<code>system.time(for (i in 1:5) qr(X))</code>	2.290	1.094	1.107
<code>system.time(for (i in 1:5) qr(X, LAPACK=TRUE))</code>	2.629	0.834	0.689
<code>system.time(for (i in 1:20) chol(XX))</code>	2.824	0.556	0.186

- These results are based on the non-threaded and threaded Intel Math Kernel Library (MKL) using the development version of R.
- Versions of the current R using MKL for BLAS are available as

```
/group/statsoft/R-patched/build-MKL-seq/bin/R
/group/statsoft/R-patched/build-MKL-thr/bin/R
```

- Currently the standard version of R on our Linux systems seems to be using OpenBLAS with multi-threading disabled.

Final Notes

- Most reasonable approaches will be accurate for reasonable problems.
- Choosing good scaling and centering can make problems more reasonable (both numerically and statistically)
- Most methods are efficient enough for our purposes.
- In some problems worrying about efficiency is important if reasonable problem sizes are to be handled.
- Making sure you are using the right approach to the right problem is much more important than efficiency.
- Some quotes:
 - D. E. Knuth, restating a comment by C. A. R. Hoare:

We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.
 - W. A. Wulf:

More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason — including blind stupidity.