Numerical linear algebra

Chris Paciorek

2024-10-31

Table of contents

Overview **2**

1. Preliminaries **3**
Context . . . 3
Goals . . . 3
Key principle . . . 3
Computational complexity . . . 3
Notation and dimensions . . . 4
Norms . . . 4
Orthogonality . . . 5
Some vector and matrix properties . . . 6
Trace and determinant of square matrices . . . 6
Transposes and inverses . . . 6
Matrix decompositions . . . 7

2. Statistical interpretations of matrix invertibility, rank, etc. **7**
Linear independence, rank, and basis vectors . . . 7
Invertibility, singularity, rank, and positive definiteness . . . 8
Interpreting an eigendecomposition . . . 9
Generalized inverses (optional) . . . 9
Matrices arising in regression . . . 11

3. Computational issues **11**
Storing matrices . . . 11
Algorithms . . . 11
Ill-conditioned problems . . . 12

4. Matrix factorizations (decompositions) and solving systems of linear equations **16**
Triangular systems . . . 17
Gaussian elimination (LU decomposition) . . . 19
When would we explicitly invert a matrix? . . . 20
Cholesky decomposition . . . 20
QR decomposition . . . 22

Introduction . . . 22
Regression and the QR . . . 23
Regression and the QR in Python and R . . . 23
Computing the QR decomposition . . . 24
The "tall-skinny" QR . . . 26
Determinants . . . 27

5. Eigendecomposition and SVD **28**
Eigendecomposition . . . 28
Singular value decomposition . . . 29

6. Computation **30**
Linear algebra in Python . . . 30
BLAS and LAPACK . . . 30
JAX and PyTorch (and GPUs) . . . 31
Exploiting known structure in matrices . . . 31
Sparse matrix formats . . . 32
Banded matrices . . . 33
Low rank updates (optional) . . . 33

7. Iterative solutions of linear systems (optional) **33**

PDF

Overview

References:

- Gentle: Numerical Linear Algebra for Applications in Statistics (available via UC Library Search) (my notes here are based primarily on this source) [Gentle-NLA]
  - Gentle: Matrix Algebra also has much of this material.
- Gentle: Computational Statistics [Gentle-CS]
- Lange: Numerical Analysis for Statisticians
- Monahan: Numerical Methods of Statistics

Videos (optional):

There are various videos from 2020 in the bCourses Media Gallery that you can use for reference if you want to.

- Video 1. Ill-conditioned problems, part 1
- Video 2. Ill-conditioned problems, part 2
- Video 3. Triangular systems of equations
- Video 4. Solving systems of equations via LU, part 1
- Video 5. Solving systems of equations via LU, part 2
- Video 6. Solving systems of equations via LU, part 3
- Video 7. Cholesky decomposition

In working through how to compute something or understanding an algorithm, it can be very helpful to depict the matrices and vectors graphically. We'll see this on the board in class.

1. Preliminaries

Context

Many statistical and machine learning methods involve linear algebra of some sort - at the very least matrix multiplication and very often some sort of matrix decomposition to fit models and do analysis: linear regression, various more sophisticated forms of regression, deep neural networks, principle components analysis (PCA) and the wide varieties of generalizations and variations on PCA, etc., etc.

Goals

Here's what I'd like you to get out of this unit:

1. How to think about the computational order (number of computations involved) of a problem
2. How to choose a computational approach to a given linear algebra calculation you need to do.
3. An understanding of how issues with computer numbers (Unit 8) affect linear algebra calculations.

Key principle

The form of a mathematical expression and how it should be evaluated on a computer may be very different. Better computational approaches can increase speed and improve the numerical properties of the calculation.

- Example 1 (already seen in Unit 5): If X and Y are matrices and z is a vector, we should compute $X(Yz)$ rather than $(XY)z$; the former is much more computationally efficient.
- Example 2: We do not compute $(X^\top X)^{-1}X^\top Y$ by computing $X^\top X$ and finding its inverse. In fact, perhaps more surprisingly, we may never actually form $X^\top X$ in some implementations.
- Example 3: Suppose I have a matrix A, and I want to permute (switch) two rows. I can do this with a permutation matrix, P, which is mostly zeroes. On a computer, in general I wouldn't need to even change the values of A in memory in some cases (e.g., if I were to calculate PAB). Why not?

Computational complexity

We can assess the computational complexity of a linear algebra calculation by counting the number multiplys/divides and the number of adds/subtracts. Sidenote: addition is a bit faster than multiplication, so some algorithms attempt to trade multiplication for addition.

In general we do not try to count the actual number of calculations, but just their order, though in some cases in this unit we'll actually get a more exact count. In general, we denote this as $O(f(n))$ which means that the number of calculations approaches $cf(n)$ as $n \to \infty$ (i.e., we know the calculation is approximately proportional to $f(n)$). Consider matrix multiplication, AB, with matrices of size $a \times b$ and $b \times c$. Each column of the second matrix is multiplied by all the rows of the first. For any given inner product of a row by a column, we have b multiplies. We repeat these operations for each column and then for each row, so we have abc multiplies so $O(abc)$ operations. We could count the additions

as well, but there's usually an addition for each multiply, so we can usually just count the multiplys and then say there are such and such {multiply and add}s. This is Monahan's approach, but you may see other counting approaches where one counts the multiplys and the adds separately.

For two symmetric, $n \times n$ matrices, this is $O(n^3)$. Similarly, matrix factorization (e.g., the Cholesky decomposition) is $O(n^3)$ unless the matrix has special structure, such as being sparse. As matrices get large, the speed of calculations decreases drastically because of the scaling as n^3 and memory use increases drastically. In terms of memory use, to hold the result of the multiply indicated above, we need to hold $ab + bc + ac$ total elements, which for symmetric matrices sums to $3n^2$. So for a matrix with $n = 10000$, we have $3 \cdot 10000^2 \cdot 8/1e9 = 2.4$Gb.

When we have $O(n^q)$ this is known as polynomial time. Much worse is $O(b^n)$ (exponential time), while much better is $O(\log n)$ (log time). Computer scientists talk about NP-complete problems; these are essentially problems for which there is not a polynomial time algorithm - it turns out all such problems can be rewritten such that they are equivalent to one another.

In real calculations, it's possible to have the actual time ordering of two approaches differ from what the order approximations tell us. For example, something that involves n^2 operations may be faster than one that involves $1000(n \log n + n)$ even though the former is $O(n^2)$ and the latter $O(n \log n)$. The reasons are that the constant, $c = 1000$, can matter (depending on how big n is), as can the extra calculations from the lower order term(s), in this case $1000n$.

A note on terminology: *flops* stands for both floating point operations (the number of operations required) and floating point operations per second, the speed of calculation.

Notation and dimensions

I'll try to use capital letters for matrices, A, and lower-case for vectors, x. Then x_i is the ith element of x, A_{ij} is the ith row, jth column element, and $A_{\cdot j}$ is the jth column and $A_{i\cdot}$ the ith row. By default, we'll consider a vector, x, to be a one-column matrix, and $x^\top$ to be a one-row matrix. Some of the references given at the start of this Unit also use a_{ij} for A_{ij} and a_j for the jth column.

Throughout, we'll need to be careful that the matrices involved in an operation are conformable: for $A + B$ both matrices need to be of the same dimension, while for AB the number of columns of A must match the number of rows of B. Note that this allows for B to be a column vector, with only one column, Ab. Just checking dimensions is a good way to catch many errors. Example: is $\text{Cov}(Ax) = A\text{Cov}(x)A^\top$ or $\text{Cov}(Ax) = A^\top\text{Cov}(x)A$? Well, if A is $m \times n$, it must be the former, as the latter is not conformable.

The **inner product** of two vectors is $\sum_i x_i y_i = x^\top y \equiv \langle x, y \rangle \equiv x \cdot y$.

The **outer product** is $xy^\top$, which comes from all pairwise products of the elements.

When the indices of summation should be obvious, I'll sometimes leave them implicit. Ask me if it's not clear.

Norms

For a vector, $\|x\|_p = (\sum_i |x_i|^p)^{1/p}$ and the standard (Euclidean) norm is $\|x\|_2 = \sqrt{\sum x_i^2} = \sqrt{x^\top x}$, just the length of the vector in Euclidean space, which we'll refer to as $\|x\|$, unless noted otherwise.

One commonly used norm for a matrix is the Frobenius norm, $\|A\|_F = (\sum_{i,j} a_{ij}^2)^{1/2}$.

In this Unit, we'll often make use of the **induced matrix norm**, which is defined relative to a corresponding vector norm, $\|\cdot\|$, as:

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

So we have

$$\|A\|_2 = \sup_{x \neq 0} \frac{\|Ax\|_2}{\|x\|_2} = \sup_{\|x\|_2 = 1} \|Ax\|_2$$

If you're not familiar with the supremum ("sup" above), you can just think of it as taking the maximum. In the case of the 2-norm, the norm turns out to be the largest singular value in the singular value decomposition (SVD) of the matrix.

We can interpret the norm of a matrix as the most that the matrix can stretch a vector when multiplying by the vector (relative to the length of the vector).

A property of any legitimate matrix norm (including the induced norm) is that $\|AB\| \leq \|A\|\|B\|$. Also recall that norms must obey the triangle inequality, $\|A + B\| \leq \|A\| + \|B\|$.

A normalized vector is one with "length", i.e., Euclidean norm, of one. We can easily normalize a vector: $\tilde{x} = x/\|x\|$

The angle between two vectors is

$$\theta = \cos^{-1}\left(\frac{\langle x, y\rangle}{\sqrt{\langle x, x\rangle\langle y, y\rangle}}\right)$$

Orthogonality

Two vectors are orthogonal if $x^\top y = 0$, in which case we say $x \perp y$. An **orthogonal matrix** is a square matrix in which all of the columns are orthogonal to each other and normalized. The same holds for the rows. Orthogonal matrices can be shown to have full rank. Furthermore if A is orthogonal, $A^\top A = I$, so $A^{-1} = A^\top$. Given all this, the determinant of orthogonal A is either 1 or -1. Finally the product of two orthogonal matrices, A and B, is also orthogonal since $(AB)^\top AB = B^\top A^\top AB = B^\top B = I$.

Permutations

Sometimes we make use of matrices that permute two rows (or two columns) of another matrix when multiplied. Such a matrix is known as an elementary permutation matrix and is an orthogonal matrix with a determinant of -1. You can multiply such matrices to get more general permutation matrices that are also orthogonal. If you premultiply by P, you permute rows, and if you postmultiply by P you permute columns. Note that on a computer, you wouldn't need to actually do the multiply (and if you did, you should use a sparse matrix routine), but rather one can often just rework index values that indicate where relevant pieces of the matrix are stored (more in the next section).

Some vector and matrix properties

$AB \neq BA$ but $A + B = B + A$ and $A(BC) = (AB)C$.

In Python, recall the syntax is

```
A + B

# Matrix multiplication
np.matmul(A, B)
A @ B        # alternative
A.dot(B)     # not recommended by the NumPy docs

A * B # Hadamard (direct) product
```

You don't need the spaces, but they're nice for code readability.

Trace and determinant of square matrices

The trace of a matrix is the sum of the diagonal elements. For square matrices, $\text{tr}(A + B) = \text{tr}(A) + \text{tr}(B)$, $\text{tr}(A) = \text{tr}(A^\top)$.

We also have $\text{tr}(ABC) = \text{tr}(CAB) = \text{tr}(BCA)$ - basically you can move a matrix from the beginning to the end or end to beginning, provided they are conformable for this operation. This is helpful for a couple reasons:

1. We can find the ordering that reduces computation the most if the individual matrices are not square.
2. $x^\top Ax = \text{tr}(x^\top Ax)$ since the quadratic form, $x^\top Ax$, is a scalar, and this is equal to $\text{tr}(xx^\top A)$ where $xx^\top A$ is a matrix. It can be helpful to be able to go back and forth between a scalar and a trace in some statistical calculations.

For square matrices, the determinant exists and we have $|AB| = |A||B|$ and therefore, $|A^{-1}| = 1/|A|$ since $|I| = |AA^{-1}| = 1$. Also $|A| = |A^\top|$, which can be seen using the QR decomposition for A and understanding properties of determinants of triangular matrices (in this case R) and orthogonal matrices (in this case Q).

Transposes and inverses

For square, invertible matrices, we have that $(A^{-1})^\top = (A^\top)^{-1}$. Why? Since we have $(AB)^\top = B^\top A^\top$, we have:

$$A^\top (A^{-1})^\top = (A^{-1}A)^\top = I$$

so $(A^\top)^{-1} = (A^{-1})^\top$.

For two invertible matrices, we have that $(AB)^{-1} = B^{-1}A^{-1}$ since $B^{-1}A^{-1}AB = I$.

Other matrix multiplications

The Hadamard or direct product is simply multiplication of the correspoding elements of two matrices by each other. In R this is simply`A * B`.
Challenge: How can I find tr(AB) without using `A %*% B` ?

The Kronecker product is the product of each element of one matrix with the entire other matrix"

$$A \otimes B = \begin{pmatrix} A_{11}B & \cdots & A_{1m}B \\ \vdots & \ddots & \vdots \\ A_{n1}B & \cdots & A_{nm}B \end{pmatrix}$$

The inverse of a Kronecker product is the Kronecker product of the inverses,

$$B^{-1} \otimes A^{-1}$$

which is obviously quite a bit faster because the inverse (i.e., solving a system of equations) in this special case is $O(n^3 + m^3)$ rather than the naive approach being $O((nm)^3)$.

Matrix decompositions

A matrix decomposition is a re-expression of a matrix, A, in terms of a product of two or three other, simpler matrices, where the decomposition reveals structure or relationships present in the original matrix, A. The "simpler" matrices may be simpler in various ways, including

- having fewer rows or columns;
- being diagonal, triangular or sparse in some way,
- being orthogonal matrices.

In addition, once you have a decomposition, computation is generally easier, because of the special structure of the simpler matrices.

We'll see this in great detail in Section 3.

2. Statistical interpretations of matrix invertibility, rank, etc.

Linear independence, rank, and basis vectors

A set of vectors, $v_1, \ldots v_n$, is linearly independent (LIN) when none of the vectors can be represented as a linear combination, $\sum c_i v_i$, of the others for scalars, $c_1, \ldots, c_n$. If we have vectors of length n, we can have at most n linearly independent vectors. The rank of a matrix is the number of linearly independent rows (or columns - it's the same), and is at most the minimum of the number of rows and number of columns. We'll generally think about it in terms of the dimension of the column space - so we can just think about the number of linearly independent columns.

Any set of linearly independent vectors (say $v_1, \ldots, v_n$) span a space made up of all linear combinations of those vectors ($\sum_{i=1}^{n} c_i v_i$). The spanning vectors are known as basis vectors. We can express a vector y that is in the space with respect to (as a linear combination of) basis vectors as $y = \sum_i c_i v_i$, where if the basis vectors are normalized and orthogonal, we can find the weights as $c_i = \langle y, v_i \rangle$.

Consider a regression context. We have p covariates (p columns in the design matrix, X), of which $q \leq p$ are linearly independent covariates. This means that $p-q$ of the vectors can be written as linear combos of the q vectors. The space spanned by the covariate vectors is of dimension q, rather than p, and $X^\top X$ has $p-q$ eigenvalues that are zero. The q LIN vectors are basis vectors for the space - we can represent any point in the space as a linear combination of the basis vectors. You can think of the basis vectors as being like the axes of the space, except that the basis vectors are not orthogonal. So it's like denoting a point in $\Re^q$ as a set of q numbers telling us where on each of the axes we are - this is the same as a linear combination of axis-oriented vectors.

When fitting a regression, if $n = p = q$, a vector of n observations can be represented exactly as a linear combination of the p basis vectors, so there is no residual and we have a single unique (and exact) solution (e.g., with $n = p = 2$, the observations fall exactly on the simple linear regression line). If $n < p$, then we have at most n linearly independent covariates (the rank is at most n). In this case we have multiple possible solutions and the system is ill-determined (under-determined). Similarly, if $q < p$ and $n \geq p$, the rank is again less than p and we have multiple possible solutions. Of course we usually have $n > p$, so the system is overdetermined - there is no exact solution, but regression is all about finding solutions that minimize some criterion about the differences between the observations and linear combinations of the columns of the X matrix (such as least squares or penalized least squares). In standard regression, we project the observation vector onto the space spanned by the columns of the X matrix, so we find the point in the space closest to the observation vector.

Invertibility, singularity, rank, and positive definiteness

For square matrices, let's consider how invertibility, singularity, rank and positive (or non-negative) definiteness relate.

Square matrices that are "regular" have an eigendecomposition, $A = \Gamma\Lambda\Gamma^{-1}$ where Γ is a matrix with the eigenvectors as the columns and Λ is a diagonal matrix of eigenvalues, $\Lambda_{ii} = \lambda_i$. Symmetric matrices and matrices with unique eigenvalues are regular, as are some other matrices. The number of non-zero eigenvalues is the same as the rank of the matrix. Square matrices that have an inverse are also called nonsingular, and this is equivalent to having full rank. If the matrix is symmetric, the eigenvectors and eigenvalues are real and Γ is orthogonal, so we have $A = \Gamma\Lambda\Gamma^\top$. The determinant of the matrix is the product of the eigenvalues (why?), which is zero if it is less than full rank. Note that if none of the eigenvalues are zero then $A^{-1} = \Gamma\Lambda^{-1}\Gamma^\top$.

Let's focus on symmetric matrices. The symmetric matrices that tend to arise in statistics are either positive definite (p.d.) or non-negative definite (n.n.d.). If a matrix is positive definite, then by definition $x^\top Ax > 0$ for any x. Note that if $\text{Cov}(y) = A$ then $x^\top Ax = x^\top \text{Cov}(y)x = \text{Cov}(x^\top y) = \text{Var}(x^\top y)$ if so positive definiteness amounts to having linear combinations of random variables (with the elements of x here being the weights) having positive variance. So we must have that positive definite matrices are equivalent to variance-covariance matrices (I'll just refer to this as a variance matrix or as a covariance matrix). If A is p.d. then it has all positive eigenvalues and it must have an inverse, though as we'll see, from a numerical perspective, we may not be able to compute it if some of the eigenvalues are very close to zero. In Python, `numpy.linalg.eig(A)[1]` is Γ, with each column a vector, and `numpy.linalg.eig(A)[0]` contains the (unordered) eigenvalues.

To summarize, here are some of the various connections between mathematical and statistical properties of **positive definite** matrices:

A positive definite $\Leftrightarrow$ A is a covariance matrix $\Leftrightarrow$ $x^\top Ax > 0 \Leftrightarrow \lambda_i > 0$ (positive eigenvalues) $\Rightarrow |A| > 0$ $\Rightarrow A$ is invertible $\Leftrightarrow$ A is non singular $\Leftrightarrow$ A is full rank.

And here are connections for positive semi-definite matrices:

A positive semi-definite $\Leftrightarrow$ A is a constrained covariance matrix $\Leftrightarrow$ $x^\top Ax \geq 0$ and equal to 0 for some $x \Leftrightarrow \lambda_i \geq 0$ (non-negative eigenvalues), with at least one zero $\Rightarrow |A| = 0 \Leftrightarrow A$ is not invertible $\Leftrightarrow A$ is singular $\Leftrightarrow A$ is not full rank.

Interpreting an eigendecomposition

Let's interpret the eigendecomposition in a generative context as a way of generating random vectors. We can generate y s.t. $\text{Cov}(y) = A$ if we generate $y = \Gamma\Lambda^{1/2}z$ where $\text{Cov}(z) = I$ and $\Lambda^{1/2}$ is formed by taking the square roots of the eigenvalues. So $\sqrt{\lambda_i}$ is the standard deviation associated with the basis vector $\Gamma_{\cdot i}$. That is, the z's provide the weights on the basis vectors, with scaling based on the eigenvalues. So y is produced as a linear combination of eigenvectors as basis vectors, with the variance attributable to the basis vectors determined by the eigenvalues.

To go the other direction, we can project a vector y onto the space spanned by the eigenvectors: $w = (\Gamma^\top\Gamma)^{-1}\Gamma^\top y = \Gamma^\top y = \Lambda^{1/2}z$, where the simplification of course comes from Γ being orthogonal.

If $x^\top Ax \geq 0$ then A is nonnegative definite (also called positive semi-definite). In this case one or more eigenvalues can be zero. Let's interpret this a bit more in the context of generating random vectors based on non-negative definite matrices, $y = \Gamma\Lambda^{1/2}z$ where $\text{Cov}(z) = I$. Questions:

1. What does it mean when one or more eigenvalue (i.e., $\lambda_i = \Lambda_{ii}$) is zero?
2. Suppose I have an eigenvalue that is very small and I set it to zero? What will be the impact upon y and $\text{Cov}(y)$?
3. Now let's consider the inverse of a covariance matrix, known as the precision matrix, $A^{-1} = \Gamma\Lambda^{-1}\Gamma^\top$. What does it mean if a $(\Lambda^{-1})_{ii}$ is very large? What if $(\Lambda^{-1})_{ii}$ is very small?

Consider an arbitrary $n \times p$ matrix, X. Any crossproduct or sum of squares matrix, such as $X^\top X$ is positive definite (non-negative definite if $p > n$). This makes sense as it's just a scaling of an empirical covariance matrix.

Generalized inverses (optional)

Suppose I want to find x such that $Ax = b$. Mathematically the answer (provided A is invertible, i.e. of full rank) is $x = A^{-1}b$.

Generalized inverses arise in solving equations when A is not full rank. A generalized inverse is a matrix, A^- s.t. $AA^-A = A$. The Moore-Penrose inverse (the pseudo-inverse), A^+, is a (unique) generalized inverse that also satisfies some additional properties. $x = A^+b$ is the solution to the linear system, $Ax = b$, that has the shortest length for x.

We can find the pseudo-inverse based on an eigendecomposition (or an SVD) as $\Gamma\Lambda^+\Gamma^\top$. We obtain Λ^+ from Λ as follows. For values $\lambda_i > 0$, compute $1/\lambda_i$. All other values are set to 0. Let's interpret this statistically. Suppose we have a precision matrix with one or more zero eigenvalues and we want to find the covariance matrix. A zero eigenvalue means we have no precision, or infinite variance, for

some linear combination (i.e., for some basis vector). We take the pseudo-inverse and assign that linear combination zero variance.

Let's consider a specific example. Autoregressive models are often used for smoothing (in time, in space, and in covariates). A first order autoregressive model for $y_1, y_2, \ldots, y_T$ has $E(y_i|y_{-i}) = \frac{1}{2}(y_{i-1} + y_{i+1})$. Another way of writing the model is in time-order: $y_i = y_{i-1} + \epsilon_i$. A second order autoregressive model has $E(y_i|y_{-i}) = \frac{1}{6}(4y_{i-1} + 4y_{i+1} - y_{i-2} - y_{i+2})$. These constructions basically state that each value should be a smoothed version of its neighbors. One can figure out that the **precision** matrix for y in the first order model is

$$\begin{pmatrix} \ddots & & \vdots & & \\ -1 & 2 & -1 & 0 & \\ \cdots & -1 & 2 & -1 & \ldots \\ & 0 & -1 & 2 & -1 \\ & & \vdots & & \ddots \end{pmatrix}$$

and in the second order model is

$$\begin{pmatrix} \ddots & & & \vdots & & & \\ 1 & -4 & 6 & -4 & 1 & & \\ \cdots & 1 & -4 & 6 & -4 & 1 & \cdots \\ & & 1 & -4 & 6 & -4 & 1 \\ & & & \vdots & & & \end{pmatrix}.$$

If we look at the eigendecomposition of such matrices, we see that in the first order case, the eigenvalue corresponding to the constant eigenvector is zero.

```
import numpy as np

precMat = np.array([[1,-1,0,0,0],[-1,2,-1,0,0],[0,-1,2,-1,0],[0,0,-1,2,-1],[0,0,0,-1,1]])
e = np.linalg.eig(precMat)
e[0]        # 4th eigenvalue is numerically zero
```

```
array([3.61803399e+00, 2.61803399e+00, 1.38196601e+00, 4.97762256e-17,
       3.81966011e-01])
```

```
e[1][:,3]   # constant eigenvector
```

```
array([0.4472136, 0.4472136, 0.4472136, 0.4472136, 0.4472136])
```

This means we have no information about the overall level of y. So how would we generate sample y vectors? We can't put infinite variance on the constant basis vector and still generate samples. Instead we use the pseudo-inverse and assign ZERO variance to the constant basis vector. This corresponds to generating realizations under the constraint that $\sum y_i$ has no variation, i.e., $\sum y_i = \bar{y} = 0$ - you can see this by seeing that $\text{Var}(\Gamma_{\cdot i}^\top y) = 0$ when $\lambda_i = 0$.

```
# Generate a realization.
evals = e[0]
evals = 1/evals   # variances
evals[3] = 0      # generalized inverse
rng = np.random.default_rng(seed=1)
```

```
y = e[1] @ ((evals ** 0.5) * rng.normal(size = 5))
y.sum()
```

```
-6.661338147750939e-16
```

In the second order case, we have two non-identifiabilities: for the sum and for the linear component of the variation in y (linear in the indices of y).

I could parameterize a statistical model as $\mu + y$ where y has covariance that is the generalized inverse discussed above. Then I allow for both a non-zero mean and for smooth variation governed by the autoregressive structure. In the second-order case, I would need to add a linear component as well, given the second non-identifiability.

Matrices arising in regression

In regression, we work with $X^\top X$. Some properties of this matrix are that it is symmetric and non-negative definite (hence our use of $(X^\top X)^{-1}$ in the OLS estimator). When is it not positive definite?

Fitted values are $X\hat{\beta} = X(X^\top X)^{-1}X^\top Y = HY$. The "hat" matrix, H, projects Y into the column space of X. H is idempotent: $HH = H$, which makes sense - once you've projected into the space, any subsequent projection just gives you the same thing back. H is singular. Why? Also, under what special circumstance would it not be singular?

3. Computational issues

Storing matrices

We've discussed column-major and row-major storage of matrices. First, retrieval of matrix elements from memory is quickest when multiple elements are contiguous in memory. So in a column-major language (e.g., R, Fortran), it is best to work with values in a common column (or entire columns) while in a row-major language (e.g., Python, C) for values in a common row.

In some cases, one can save space (and potentially speed) by overwriting the output from a matrix calculation into the space occupied by an input. This occurs in some clever implementations of matrix factorizations.

Algorithms

Good algorithms can change the efficiency of an algorithm by one or more orders of magnitude, and many of the improvements in computational speed over recent decades have been in algorithms rather than in computer speed.

Most matrix algebra calculations can be done in multiple ways. For example, we could compute $b = Ax$ in either of the following ways, denoted here in pseudocode.

1. Collect the inner products of the rows of A with x.

```
# initialize b[1:n]=0
```

```
for(i=1:n) {
    for(j=1:m){
        b_i = b_i + a_{ij} x_j
    }
}
```

2. Take the linear combination (based on x) of the columns of A

```
# initialize b[1:n]=0
for(j=1:m){
    for(i = 1:n){
        b_i = b_i + a_{ij} x_j
    }
}
```

In this case the two approaches involve the same number of operations. But the first accesses A by row, so it might be better for row-major matrices (so might be how we would implement in C). The second accesses A by column, so it might be better for column-major matrices (so might be how we would implement in Fortran).

> Challenge
>
> Check whether the first approach is faster in Python with numpy's default row-major ordering. (Write the code just doing the outer loop as a for loop and doing the inner loop using vectorized calculation.) Your answer will probably depend on how big the matrices are.

General computational issues

The same caveats we discussed in terms of computer arithmetic hold naturally for linear algebra, since this involves arithmetic with many elements. Good implementations of algorithms are aware of the danger of catastrophic cancellation and of the possibility of dividing by zero or by values that are near zero.

Ill-conditioned problems

Basics

A problem is ill-conditioned if small changes to values in the computation result in large changes in the result. This is quantified by something called the *condition number* of a calculation. For different operations there are different condition numbers.

Ill-conditionedness arises most often in terms of matrix inversion, so the standard condition number is the "condition number with respect to inversion", which when using the L_2 norm is the ratio of the

absolute values of the largest to smallest eigenvalue. Here's an example:

$$A = \begin{pmatrix} 10 & 7 & 8 & 7 \\ 7 & 5 & 6 & 5 \\ 8 & 6 & 10 & 9 \\ 7 & 5 & 9 & 10 \end{pmatrix}.$$

The solution of $Ax = b$ for $b = (32, 23, 33, 31)$ is $x = (1, 1, 1, 1)$, while the solution for $b + \delta b = (32.1, 22.9, 33.1, 30.9)$ is $x + \delta x = (9.2, -12.6, 4.5, -1.1)$, where δ is notation for a perturbation to the vector or matrix.

```
def norm2(x):
    return(np.sum(x**2) ** 0.5)

A = np.array([[10,7,8,7],[7,5,6,5],[8,6,10,9],[7,5,9,10]])
b = np.array([32,23,33,31])
x = np.linalg.solve(A, b)

bPerturbed = np.array([32.1, 22.9, 33.1, 30.9])
xPerturbed = np.linalg.solve(A, bPerturbed)

delta_b = bPerturbed - b
delta_x = xPerturbed - x
```

What's going on? Some manipulations with inequalities involving the induced matrix norm (for any chosen vector norm, but we might as well just think about the Euclidean norm) (see Gentle-CS Sec. 5.1 or the derivation in class) give

$$\frac{\|\delta x\|}{\|x\|} \leq \|A\|\|A^{-1}\|\frac{\|\delta b\|}{\|b\|}$$

where we define the condition number w.r.t. inversion as $\text{cond}(A) \equiv \|A\|\|A^{-1}\|$. We'll generally work with the L_2 norm, and for a nonsingular square matrix the result is that the condition number is the ratio of the absolute values of the largest and smallest magnitude eigenvalues. This makes sense since $\|A\|_2$ is the absolute value of the largest magnitude eigenvalue of A and $\|A^{-1}\|_2$ is the absolute value of the largest magnitude eigenvalue of A^{-1}. But since the eigenvalues of the inverse are the inverses of the eigenvalues of A, the latter is the inverse of the absolute value of the smallest magnitude eigenvalue of A.

We see in the code below that the large disparity in eigenvalues of A leads to an effect predictable from our inequality above, with the condition number helping us find an upper bound.

```
e = np.linalg.eig(A)
evals = e[0]
print(evals)
```

```
[3.02886853e+01 3.85805746e+00 1.01500484e-02 8.43107150e-01]
```

```
## relative perturbation in x much bigger than in b
norm2(delta_x) / norm2(x)
```

```
8.19847546803699
```

```
norm2(delta_b) / norm2(b)
```

```
0.0033319453118976702
```

```
## ratio of relative perturbations
(norm2(delta_x) / norm2(x)) / (norm2(delta_b) / norm2(b))
```

```
2460.567236431514
```

```
## ratio of largest and smallest magnitude eigenvalues
## confusingly evals[2] is the smallest, not evals[3]
(evals[0]/evals[2])
```

```
2984.092701676269
```

The main use of these ideas for our purposes is in thinking about the numerical accuracy of a linear system solution (Gentle-NLA Sec 3.4). On a computer we have the system

$$(A + \delta A)(x + \delta x) = b + \delta b$$

where the 'perturbation' is from the inaccuracy of computer numbers. Our exploration of computer numbers tells us that

$$\frac{\|\delta b\|}{\|b\|} \approx 10^{-p}; \quad \frac{\|\delta A\|}{\|A\|} \approx 10^{-p}$$

where $p = 16$ for standard double precision floating points. Following Gentle, one gets the approximation

$$\frac{\|\delta x\|}{\|x\|} \approx \text{cond}(A)10^{-p},$$

so if $\text{cond}(A) \approx 10^t$, we have accuracy of order 10^{t-p} instead of 10^{-p}. (Gentle cautions that this holds only if $10^{t-p} \ll 1$). So we can think of the condition number as giving us the number of digits of accuracy lost during a computation relative to the precision of numbers on the computer. E.g., a condition number of 10^8 means we lose 8 digits of accuracy relative to our original 16 on standard systems. One issue is that estimating the condition number is itself subject to numerical error and requires computation of A^{-1} (albeit not in the case of L_2 norm with square, nonsingular A) but see Golub and van Loan (1996; p. 76-78) for an algorithm.

Improving conditioning

Ill-conditioned problems in statistics often arise from collinearity of regressors. Often the best solution is not a numerical one, but re-thinking the modeling approach, as this generally indicates statistical issues beyond just the numerical difficulties.

A general comment on improving conditioning is that we want to avoid large differences in the magnitudes of numbers involved in a calculation. In some contexts such as regression, we can center and scale the columns to avoid such differences - this will improve the condition of the problem. E.g., in simple quadratic regression with $x = \{1990, \ldots, 2010\}$ (e.g., regressing on calendar years), we see that centering and scaling the matrix columns makes a huge difference on the condition number

```
import statsmodels.api as sm
rng = np.random.default_rng(seed=1)

t1 = np.arange(1990, 2011)  # naive covariate
n = len(t1)
X1 = np.column_stack((np.ones(n), t1, t1 ** 2))

beta = np.array([5, 0.1, 0.0001])
y = X1 @ beta + rng.normal(size = n)

e1 = np.linalg.eig(X1.T @ X1)
np.sort(e1[0])[::-1]
```

```
array([3.36018564e+14, 7.69949736e+02, 2.24079720e-08])
```

```
np.linalg.cond(X1.T @ X1)  # built-in!
```

```
4.653329826789808e+21
```

```
sm.OLS(y, X1).fit().params
```

```
array([-1.14908044e+03,  1.28438902e+00, -2.03662592e-04])
```

The fitted values are quite different than the true beta values of (5, 0.1, 0.0001), particularly the intercept and linear terms. This is primarily because of the ill-conditioning, rather than that the OLS estimate is different than the true values because we are estimating beta.

Now we'll do a simple transformation, subtracting off the mean of the covariate, so that the covariate values (in particular the quadratic values) don't have such large magnitudes.

```
t2 = t1 - 2000              # centered
X2 = np.column_stack((np.ones(n), t2, t2 ** 2))
e2 = np.linalg.eig(X2.T @ X2)
with np.printoptions(suppress=True):
    print(np.sort(e2[0])[::-1])
```

```
[50677.70427505   770.             9.29572495]
```

```
sm.OLS(y, X2).fit().params
```

```
array([ 6.05047224e+02,  4.69738650e-01, -2.03662592e-04])
```

We can work out that the effect of centering is that the true beta values after centering are (605, 0.5, 0.0001). (Consider $\beta_0 + \beta_1(t - c + c) + \beta_2(t - c + c)^2$ for $c = 2000$.) So for the intercept and linear term, the OLS estimates are now pretty similar to the true values.

Interestingly, the estimate for the quadratic terms has not changed at all. This is quite surprising. There must be something subtle going on that causes that not to be affected by the bad conditioning, but I'm not sure what is happening there.

We could go even further to improve the conditioning.

```
t3 = t2/10                  # centered and scaled
X3 = np.column_stack((np.ones(n), t3, t3 ** 2))
e3 = np.linalg.eig(X3.T @ X3)
with np.printoptions(suppress=True):
    print(np.sort(e3[0])[::-1])
```

```
[24.11293487  7.7         1.95366513]
```

I haven't shown the OLS results for the third version, but with a bit of arithmetic to be done, but you should be able to verify that the third approach also gives reasonable answers.

The basic story is that simple strategies often solve the problem, and that you should be aware of the absolute and relative magnitudes involved in your calculations.

One rule of thumb is to try to work with numbers whose magnitude is around 1. We can often scale the values in our problem in order to do this. I.e., change the units of your variables. Instead of personal income in dollars, use personal income in thousands or hundreds of thousands of dollars.

4. Matrix factorizations (decompositions) and solving systems of linear equations

Suppose we want to solve the following linear system:

$$\begin{aligned} Ax &= b \\ x &= A^{-1}b \end{aligned}$$

Numerically, this is never done by finding the inverse and multiplying. Rather we solve the system using a matrix decomposition (or equivalent set of steps). One approach uses Gaussian elimination (equivalent to the LU decomposition), while another uses the Cholesky decomposition. There are also iterative methods that generate a sequence of approximations to the solution but reduce computation (provided they are stopped before the exact solution is found).

Gentle-CS has a nice table overviewing the various factorizations (Table 5.1, page 219). I've reproduced a variation on it here.

Table 1: Matrix factorizations useful for statistics / data science / machine learning

Name	Representation	Restrictions	Properties	Uses
LU	$A_{nn} = L_{nn}U_{nn}$	A generally square	L lower triangular; U upper triangular	solving equations; inversion
QR	$A_{nm} = Q_{nn}R_{nm}$ or $A_{nm} = Q_{nm}R_{mm}$(skinny)		Q orthogonal; R upper triangular	regression

Name	Representation	Restrictions	Properties	Uses
Cholesky	$A_{nn} = U_{nn}^\top U_{nn}$	A positive (semi-) definite	U upper triangular	multivariate normal; covariance; solving equations; inversion
Eigen decomposition	$A_{nn} = \Gamma_{nn}\Lambda_{nn}\Gamma_{nn}^\top$	A square, symmetric*	Γ orthogonal; Λ (non-negative**) diagonal	principal components analysis and related
SVD	$A_{nm} = U_{nn}D_{nm}V_{mm}^\top$ or $A_{nm} = U_{nk}D_{kk}V_{mk}^\top$		U, V orthogonal; D (non-negative) diagonal	machine learning, topic models

*For the eigen decomposition, I assume A is symmetric, though there is a decomposition for non-symmetric A.

** For positive definite or positive semi-definite A.

Triangular systems

As a preface, let's figure out how to solve $Ax = b$ if A is upper triangular. The basic algorithm proceeds from the bottom up (and therefore is called a 'backsolve'. We solve for x_n trivially, and then move upwards plugging in the known values of x and solving for the remaining unknown in each row (each equation).

1. $x_n = b_n / A_{nn}$
2. Now for $k < n$, use the already computed $\{x_n, x_{n-1}, \ldots, x_{k+1}\}$ to calculate $x_k = \frac{b_k - \sum_{j=k+1}^{n} x_j A_{kj}}{A_{kk}}$.
3. Repeat for all rows.

How many multiplies and adds are done? Solving lower triangular systems is very similar and involves the same number of calculations.

In Scipy, we can use `linalg.solve_triangular` to solve triangular systems. The `trans` argument indicates whether to solve $Ax = b$ or $A^\top x = b$ (so one wouldn't need to transpose before solving).

```
import scipy as sp
rng = np.random.default_rng(seed=1)
n = 20
X = rng.normal(size = (n,n))
## R has the `crossprod` function, which would be more efficient
## than having to transpose, but numpy doesn't seem to have an equivalent.
A = X.T @ X  # produce a positive def. matrix

b = rng.normal(size = n)
L = np.linalg.cholesky(A) # L is lower-triangular
```

```
U = L.T

out1 = sp.linalg.solve_triangular(L, b, lower=True)
out2 = np.linalg.inv(L) @ b
np.allclose(out1, out2)
```

```
True
```

```
out3 =  sp.linalg.solve_triangular(U, b, lower=False)
out4 = np.linalg.inv(U) @ b
np.allclose(out1, out2)
```

```
True
```

To reiterate the distinction between matrix inversion and solving a system of equations, when we write $U^{-1}b$, what we mean on a computer is to carry out the above algorithm, not to find the inverse and then multiply.

Here's a good reason why.

```
import time

rng = np.random.default_rng(seed=1)
n = 5000
X = rng.normal(size = (n,n))

## R has the `crossprod` function, which would be more efficient
## than having to transpose, but numpy doesn't seem to have an equivalent.
A = X.T @ X
b = rng.normal(size = n)
L = np.linalg.cholesky(A) # L is lower-triangular

t0 = time.time()
out1 = sp.linalg.solve_triangular(L, b, lower=True)
time.time() - t0
```

```
0.061060428619384766
```

```
t0 = time.time()
out2 = np.linalg.inv(L) @ b
time.time() - t0
```

```
5.032368183135986
```

That assumes you have L, but we'll see in a bit that even when one accounts for the creation of L, you don't want to invert matrices in order to solve systems of equations.

Gaussian elimination (LU decomposition)

Gaussian elimination is a standard way of directly computing a solution for $Ax = b$. It is equivalent to the LU decomposition. LU is primarily done with square matrices, but not always. Also LU decompositions do exist for some singular matrices.

The idea of Gaussian elimination is to convert the problem to a triangular system. In class, we'll walk through Gaussian elimination in detail and see how it relates to the LU decomposition. I'll describe it more briefly here. Following what we learned in algebra when we have multiple equations, we preserve the solution, x, when we add multiples of rows (i.e., add multiples of equations) together. This amounts to doing $L_1Ax = L_1b$ for a lower-triangular matrix L_1 that produces all zeroes in the first column of L_1A except for the first row. We proceed to zero out values below the diagonal for the other columns of A. The result is $L_{n-1} \cdots L_1Ax \equiv Ux = L_{n-1} \cdots L_1b \equiv b^*$ where U is upper triangular. This is the forward reduction step of Gaussian elimination. Then the backward elimination step solves $Ux = b^*$.

If we're just looking for the solution of the system, we don't need the lower-triangular factor $L = (L_{n-1} \cdots L_1)^{-1}$ in $A = LU$, but it turns out to have a simple form that is computed as we go along, it is unit lower triangular and the values below the diagonal are the negative of the values below the diagonals in $L_1, \ldots, L_{n-1}$ (note that each L_j has non-zeroes below the diagonal only in the jth column). As a side note related to storage, it turns out that as we proceed, we can store the elements of L and U in the original A matrix, except for the implicit 1s on the diagonal of L.

In class, we'll work out the computational complexity of the LU and see that it is $O(n^3)$.

If we look at `help(np.linalg.solve)` in Python, we see that it uses *_gesv*. A Google search indicates that this is a Lapack routine that does the LU decomposition with partial pivoting and row interchanges (see below on what these are), so numpy is using the algorithm we've just discussed.

We can also explicitly get the LU decomposition in Python with `scipy.linalg.lu()`, though for most use cases, what we want to do is solve a system of equations. (In R, one can't easily get the explicit LU decomposition, though `solve()` in R does use the LU.

One additional complexity is that we want to avoid dividing by very small values to avoid introducing numerical inaccuracy (we would get large values that might overwhelm whatever they are being added to, and small errors in the divisor will have large effects on the result). This can be done on the fly by interchanging equations to use the equation (row) that produces the largest value to divide by. For example in the first step, we would switch the first equation (first row) for whichever of the remaining equations has the largest value in the first column. This is called partial pivoting. The divisors are called pivots. Complete pivoting also considers interchanging columns, and while theoretically better, partial pivoting is generally sufficient and requires fewer computations. Partial pivoting can be expressed as multiplying along the way by permutation matrices, $P_1, \ldots P_{n-1}$ that switch rows. One can show with some work that based on pivoting, we have $PA = LU$, where $P = P_{n-1} \cdots P_1$. In the demo code, we'll see a toy example of the impact of pivoting.

Finally $|PA| = |P||A| = |L||U| = |U|$ (why?) so $|A| = |U|/|P|$ and since the determinant of each permutation matrix, P_j is -1 (except when $P_j = I$ because we don't need to switch rows), we just need to multiply by minus one if there is an odd number of permutations. Or if we know the matrix is non-negative definite, we just take the absolute value of $|U|$. So Gaussian elimination provides a fast stable way to find the determinant.

When would we explicitly invert a matrix?

In some cases (as we discussed in class) you actually need the inverse as the output (e.g., for estimating standard errors).

But if you are just needing the inverse to multiply it by a vector or another matrix, you're better off using a decomposition. Basically, one can work out that the cost of computing the inverse and doing subsequent multiplication with it is rather more than computing the decomposition and doing subsequent multiplications with it, regardless of how many columns there are in the matrix you are multiplying by.

With numpy, that may mean computing the decomposition and then carrying out the steps to do the multiplication using the decomposition or using `np.linalg.solve(A, B)` which, as mentioned above, uses the LU behind the scenes. One would not do `np.linalg.inv(A) @ B`.

Cholesky decomposition

When A is p.d., we can use the Cholesky decomposition to solve a system of equations. Positive definite matrices can be decomposed as $U^\top U = A$ where U is upper triangular. U is called a square root matrix and is unique (apart from the sign, which we fix by requiring the diagonals to be positive). One algorithm for computing U is:

1. $U_{11} = \sqrt{A_{11}}$
2. For $j = 2, \ldots, n$, $U_{1j} = A_{1j}/U_{11}$
3. For $i = 2, \ldots, n$,
   - $U_{ii} = \sqrt{A_{ii} - \sum_{k=1}^{i-1} U_{ki}^2}$
   - if $i < n$, then for $j = i+1, \ldots, n$: $U_{ij} = (A_{ij} - \sum_{k=1}^{i-1} U_{ki}U_{kj})/U_{ii}$

We can then solve a system of equations as: $U^{-1}(U^{\top -1}b)$.

Confusingly, while numpy's `cholesky` gives $L = U^\top$, scipy's `cholesky` can return either L or U but defaults to U.

Here are two ways we can use the Cholesky to solve a system of equations:

```
U = sp.linalg.cholesky(A)
sp.linalg.solve_triangular(U,
          sp.linalg.solve_triangular(U, b, lower=False, trans='T'),
          lower=False)

U, lower = sp.linalg.cho_factor(A)
sp.linalg.cho_solve((U, lower), b)
```

The Cholesky has some nice advantages over the LU: (1) while both are $O(n^3)$, the Cholesky involves only half as many computations, $n^3/6 + O(n^2)$ and (2) the Cholesky factorization has only $(n^2+n)/2$ unique values compared to $n^2 + n$ for the LU. Of course the LU is more broadly applicable. The Cholesky does require computation of square roots, but it turns out this is not too intensive. There is also a method for finding the Cholesky without square roots.

Uses of the Cholesky

The standard algorithm for generating $y \sim \mathcal{N}(0, A)$ is:

```
L = sp.linalg.cholesky(A, lower=True)
y = L @ rng.normal(size = n)
```

> Question
>
> Where will most of the time in this two-step calculation be spent?

If a regression design matrix, X, is full rank, then $X^\top X$ is positive definite, so we could find $\hat{\beta} = (X^\top X)^{-1} X^\top Y$ using either the Cholesky or Gaussian elimination.

However, for OLS, it turns out that the standard approach is to work with X using the QR decomposition rather than working with $X^\top X$; working with X is more numerically stable, though in most situations without extreme collinearity, either of the approaches will be fine.

We could also consider the GLS estimator, which accounts for dependence in the errors: $\hat{\beta} = (X^\top \Sigma^{-1} X)^{-1} X^\top \Sigma^{-1} Y$

> Challenge
>
> Write efficient Python code to carry out the GLS solution using the Cholesky factorization.

Numerical issues with eigendecompositions and Cholesky decompositions for positive definite matrices

Monahan comments that in general Gaussian elimination and the Cholesky decomposition are very stable. However, in the Cholesky case, if the matrix is very ill-conditioned we can get $A_{ii} - \sum_k U_{ki}^2$ being negative and then the algorithm stops when we try to take the square root. In this case, the Cholesky decomposition does not exist numerically although it exists mathematically. It's not all that hard to produce such a matrix, particularly when working with high-dimensional covariance matrices with large correlations.

```
rng = np.random.default_rng(seed=1)
locs = rng.uniform(size = 100)
rho = .1
dists = np.abs(locs[:, np.newaxis] - locs)
C = np.exp(-dists**2/rho**2)
e = np.linalg.eig(C)
np.sort(e[0])[::-1][96:100]
```

```
array([-3.73181266e-16-5.84190120e-17j, -4.67035510e-16+1.89571791e-16j,
       -4.67035510e-16-1.89571791e-16j, -9.16675579e-16+0.00000000e+00j])
```

```
try:
    L = np.linalg.cholesky(C)
```

```
except Exception as error:
    print(error)
```

```
Matrix is not positive definite
```

```
vals = np.abs(e[0])
np.max(vals)/np.min(vals)
```

```
4.4428831436221965e+17
```

I don't see a way to use pivoting with the Cholesky in Python, but in R, one can do `chol(C, pivot = TRUE)`.

We can think about the accuracy here as follows. Suppose we have a matrix whose diagonal elements (i.e., the variances) are order of magnitude 1 and that the true value of a U_{ii} is less than 1×10^{-16}. From the given A_{ii} we are subtracting $\sum_k U_{ki}^2$ and trying to calculate this very small number but we know that we can only represent the values A_{ii} and $\sum_k U_{ki}^2$ accurately to 16 places, so the difference is garbage starting in the 17th position and could well be negative. Now realize that $\sum_k U_{ki}^2$ is the result of a potentially large set of arithmetic operations, and is likely represented accurately to fewer than 16 places. Now if the true value of U_{ii} is smaller than the accuracy to which $\sum_k U_{ki}^2$ is represented, we can get a difference that is negative.

Note that when the Cholesky fails, we can still compute an eigendecomposition, but we have negative numeric eigenvalues. Even if all the eigenvalues are numerically positive (or equivalently, we're able to get the Cholesky), errors in small eigenvalues near machine precision could have large effects when we work with the inverse of the matrix. This is what happens when we have columns of the X matrix nearly collinear. We cannot statistically distinguish the effect of two (or more) covariates, and this plays out numerically in terms of unstable results.

A strategy when working with mathematically but not numerically positive definite A is to set eigenvalues or singular values to zero when they get very small, which amounts to using a pseudo-inverse and setting to zero any linear combinations with very small variance. We can also use pivoting with the Cholesky and accumulate zeroes in the last $n - q$ rows (for cases where we try to take the square root of a negative number), corresponding to the columns of A that are numerically linearly dependent.

QR decomposition

Introduction

The QR decomposition is available for any matrix, $X = QR$, with Q orthogonal and R upper triangular. If X is non-square, $n \times p$ with $n > p$ then the leading p rows of R provide an upper triangular matrix (R_1) and the remaining rows are 0. (I'm using p because the QR is generally applied to design matrices in regression). In this case we really only need the first p columns of Q, and we have $X = Q_1 R_1$, the 'skinny' QR (this is what R's QR provides). For uniqueness, we can require the diagonals of R to be nonnegative, and then R will be the same as the upper-triangular Cholesky factor of $X^\top X$:

$$\begin{aligned} X^\top X &= R^\top Q^\top Q R \\ &= R^\top R \end{aligned}$$

There are three standard approaches for computing the QR, using (1) reflections (Householder transformations), (2) rotations (Givens transformations), or (3) Gram-Schmidt orthogonalization (see below for details).

For $n \times n$ X, the QR (for the Householder approach) requires $2n^3/3$ flops, so QR is less efficient than LU or Cholesky.

We can also obtain the pseudo-inverse of X from the QR: $X^+ = [R_1^{-1}\, 0]Q^\top$. In the case that X is not full-rank, there is a version of the QR that will work (involving pivoting) and we end up with some additional zeroes on the diagonal of R_1.

Regression and the QR

Often QR is used to fit linear models, including in R. Consider the linear model in the form $Y = X\beta + \epsilon$, finding $\hat{\beta} = (X^\top X)^{-1}X^\top Y$. Let's consider the skinny QR and note that $R^\top$ is invertible. Therefore, we can express the normal equations as

$$\begin{aligned} X^\top X\beta &= X^\top Y \\ R^\top Q^\top QR\beta &= R^\top Q^\top Y \\ R\beta &= Q^\top Y \end{aligned}$$

and solving for β is just a backsolve since R is upper-triangular. Furthermore the standard regression quantities, such as the hat matrix, the SSE, the residuals, etc. can be easily expressed in terms of Q and R.

Why use the QR instead of the Cholesky on $X^\top X$? The condition number of X is the square root of that of $X^\top X$, and the QR factorizes X. Monahan has a discussion of the condition of the regression problem, but from a larger perspective, the situations where numerical accuracy is a concern are generally cases where the OLS estimators are not particularly helpful anyway (e.g., highly collinear predictors).

What about computational order of the different approaches to least squares? The Cholesky is $np^2 + \frac{1}{3}p^3$, an algorithm called sweeping is $np^2 + p^3$, the Householder method for QR is $2np^2 - \frac{2}{3}p^3$, and the modified Gram-Schmidt approach for QR is $2np^2$. So if $n \gg p$ then Cholesky (and sweeping) are faster than the QR approaches. According to Monahan, modified Gram-Schmidt is most numerically stable and sweeping least. In general, regression is pretty quick unless p is large since it is linear in n, so it may not be worth worrying too much about computational differences of the sort noted here.

Regression and the QR in Python and R

We can get the Q and R matrices easily in Python.

```
Q,R = np.linalg.qr(X)
```

One of the methods used by the `statsmodel` package in Python uses the QR to fit a regression.

Note that by default in Python (and in R), you get the skinny QR, namely only the first p rows of R and the first p columns of Q, where the latter form an orthonormal basis for the column space of X. The remaining columns form an orthonormal basis for the null space of X (the space orthogonal to

the column space of X). The analogy in regression is that we get the basis vectors for the regression, while adding the remaining columns gives us the full n-dimensional space of the observations.

Regression in R uses the QR decomposition via `qr()`, which calls a Fortran function. `qr()` (and the Fortran functions that are called) is specifically designed to output quantities useful in fitting linear models.

In R, `qr()` returns the result as a list meant for use by other tools. R stores the R matrix in the upper triangle of `\$qr`, while the lower triangle of *$qr* and *$aux* store the information for constructing Q (this relates to the Householder-related vectors u below). One can multiply by Q using *qr.qy()* and by $Q^\top$ using *qr.qty()*. If you want to extract R and Q, the following will work:

```
X.qr = qr(X)
Q = qr.Q(X.qr)
R = qr.R(X.qr)
```

As a side note, there are QR-based functions that provide regression-related quantities, such as *qr.resid()*, *qr.fitted()* and *qr.coef()*. These functions (and their Fortran counterparts) exist because one can work through the various regression quantities of interest and find their expressions in terms of Q and R, with nice properties resulting from Q being orthogonal and R triangular.

Computing the QR decomposition

Here we'll see some of the details of the different approaches to the QR, in part because they involve some concepts that may be useful in other contexts. I won't expect you to see all of how this works, but please skim through this to get an idea of how things are done.

One approach involves reflections of vectors and a second rotations of vectors. Reflections and rotations are transformations that are performed by orthogonal matrices. The determinant of a reflection matrix is -1 and the determinant of a rotation matrix is 1. We'll see some of the details in the demo code.

QR Method 1: Reflections

If u and v are orthonormal vectors and x is in the space spanned by u and v, $x = c_1u + c_2v$, then $\tilde{x} = -c_1u + c_2v$ is a reflection (a *Householder* reflection) along the u dimension (since we are using the negative of that basis vector). We can think of this as reflecting across the plane perpendicular to u. This extends simply to higher dimensions with orthonormal vectors, $u, v_1, v_2, ...$

Suppose we want to formulate the reflection in terms of a "Householder" matrix, Q. It turns out that

$$Qx = \tilde{x}$$

if $Q = I - 2uu^\top$. Q has the following properties: (1) $Qu = -u$, (2) $Qv = v$ for $u^\top v = 0$, (3) Q is orthogonal and symmetric.

One way to create the QR decomposition is by a series of Householder transformations that create an upper triangular R from X:

$$\begin{aligned} R = &\quad Q_p \cdots Q_1 X \\ Q = &\quad (Q_p \cdots Q_1)^\top \end{aligned}$$

where we make use of the symmetry in defining Q.

Basically Q_1 reflects the first column of X with respect to a carefully chosen u, so that the result is all zeroes except for the first element. We want $Q_1 x = \tilde{x} = (||x||, 0, \ldots, 0)$. This can be achieved with $u = \frac{x-\tilde{x}}{||x-\tilde{x}||}$. Then Q_2 makes the last $n-2$ rows of the second column equal to zero. We'll work through this a bit in class.

In the regression context, as we work through the individual transformations, $Q_j = I - 2u_j u_j^\top$, we apply them to X and Y to create R (note this would not involve doing the full matrix multiplication - think about what calculations are actually needed) and $QY = Q^\top Y$, and then solve $R\beta = Q^\top Y$. To find $\text{Cov}(\hat{\beta}) \propto (X^\top X)^{-1} = (R^\top R)^{-1} = R^{-1}R^{-\top}$ we do need to invert R, but it's upper-triangular and of dimension $p \times p$. It turns out that $Q^\top Y$ can be partitioned into the first p and the last $n-p$ elements, $z^{(1)}$ and $z^{(2)}$. The SSR is $\|z^{(1)}\|^2$ and SSE is $\|z^{(2)}\|^2$.

Final side note: if X is square (so $n = p$) you might wonder why we need Q_p since after $p-1$ reflections, we don't need to zero anything else out (since the last column of R has n non-zero elements). It turns out that if we go back to thinking about a Householder reflection in general, there is a lack of uniqueness in choosing $\tilde{x}$. It could either be $(||x||, 0, \ldots, 0)$ or $(-||x||, 0, \ldots, 0)$. For better numerical stability, one chooses from the two of those such that x_1 is of the opposite sign to $\tilde{x}_1$, so that one avoids cancellation of numbers that may be of the same magnitude when doing $x - \tilde{x}$. The transformation Q_p is the last step of taking that approach of choosing the sign at each step. Q_p doesn't zero anything out; it just basically just involves potentially setting R_{pp} to be $-R_{pp}$. (To be honest, I'm not clear on why one would bother to do that last step, but that seems to be how it is presented in discussions of the Householder approach.) Of course in the case of $p < n$, we definitely need Q_p so that the last $n-p$ rows of R are zero and we can then discard them when just using the skinny QR.

QR Method 2: Rotations

A *Givens* rotation matrix rotates a vector in a two-dimensional subspace to be axis oriented with respect to one of the two dimensions by changing the value of the other dimension. E.g. we can create $\tilde{x} = (x_1, \ldots, \tilde{x}_p, \ldots, 0, \ldots x_n)$ from $x = (x_1, \ldots, x_p, \ldots, x_q, \ldots, x_n)$ using a matrix multiplication: $\tilde{x} = Qx$. Q is orthogonal but not symmetric.

We can use a series of Givens rotations to do the QR but unless it is done carefully, more computations are needed than with Householder reflections. The basic story is that we apply a series of Givens rotations to X such that we zero out the lower triangular elements.

$$\begin{aligned} R = &\quad Q_{pn} \cdots Q_{23} Q_{1n} \cdots Q_{13} Q_{12} X \\ Q = &\quad (Q_{pn} \cdots Q_{12})^\top. \end{aligned}$$

Note that we create the $n-p$ zero rows in R (because the calculations affect the upper triangle of R), but we can then ignore those rows and the corresponding columns of Q.

QR Method 3: Gram-Schmidt Orthogonalization

Gram-Schmidt involves finding a set of orthonormal vectors to span the same space as a set of LIN vectors, $x_1, \ldots, x_p$. If we take the LIN vectors to be the columns of X, so that we are discussing the column space of X, then G-S yields the QR decomposition. Here's the algorithm:

1. $\tilde{x}_1 = \frac{x_1}{\|x_1\|}$ (normalize the first vector)
2. Orthogonalize the remaining vectors with respect to $\tilde{x}_1$:
    1. $\tilde{x}_2 = \frac{x_2 - \tilde{x}_1^\top x_2 \tilde{x}_1}{\|x_2 - \tilde{x}_1^\top x_2 \tilde{x}_1\|}$, which orthogonalizes with respect to $\tilde{x}_1$ and normalizes. Note that $\tilde{x}_1^\top x_2 \tilde{x}_1 = \langle \tilde{x}_1, x_2 \rangle \tilde{x}_1$. So we are finding a scaling, $c\tilde{x}_1$, where c is based on the inner product, to remove the variation in the x_1 direction from x_2.
    2. For $k > 2$, find interim vectors, $x_k^{(2)}$, by orthogonalizing with respect to $\tilde{x}_1$
3. Proceed for $k = 3, ...,$ in turn orthogonalizing and normalizing the first of the remaining vectors w.r.t. $\tilde{x}_{k-1}$ and orthogonalizing the remaining vectors w.r.t. $\tilde{x}_{k-1}$ to get new interim vectors

Mathematically, we could instead orthogonalize x_2 w.r.t. $\tilde{x}_1$, then orthogonalize x_3 w.r.t. $\{\tilde{x}_1, \tilde{x}_2\}$, etc. The algorithm above is the *modified* G-S, and is known to be more numerically stable if the columns of X are close to collinear, giving vectors that are closer to orthogonal. The resulting $\tilde{x}$ vectors are the columns of Q. The elements of R are obtained as we proceed: the diagonal values are the the normalization values in the denominators, while the off-diagonals are the inner products with the already-computed columns of Q that are computed as part of the numerators.

Another way to think about this is that $R = Q^\top X$, which is the same as regressing the columns of X on Q, since $(Q^\top Q)^{-1} Q^\top X = Q^\top X$. By construction, the first column of X is a scaling of the first column of Q, the second column of X is a linear combination of the first two columns of Q, etc., so R being upper triangular makes sense.

The "tall-skinny" QR

Suppose you have a very large regression problem, with n very large, and $n \gg p$. There is a variant of the QR, called the tall-skinny QR (see http://arxiv.org/pdf/0808.2664v1.pdf for details) that allows us to find the decomposition in a parallel fashion. The basic idea is to do a nested set of QR decompositions on blocks of rows of X:

$$X = \begin{pmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \end{pmatrix} = \begin{pmatrix} Q_0 R_0 \\ Q_1 R_1 \\ Q_2 R_2 \\ Q_3 R_3 \end{pmatrix},$$

followed by 'reduction' steps (this can be done in a map-reduce context) that do the QR of pairs of the R factors:

$$\begin{pmatrix} R_0 \\ R_1 \\ R_2 \\ R_3 \end{pmatrix} = \begin{pmatrix} \begin{pmatrix} R_0 \\ R_1 \end{pmatrix} \\ \begin{pmatrix} R_2 \\ R_3 \end{pmatrix} \end{pmatrix} = \begin{pmatrix} Q_{01} R_{01} \\ Q_{23} R_{23} \end{pmatrix}$$

and

$$\begin{pmatrix} R_{01} \\ R_{23} \end{pmatrix} = Q_{0123} R_{0123}.$$

The full decomposition is then

$$X = \begin{pmatrix} Q_0 & 0 & 0 & 0 \\ 0 & Q_1 & 0 & 0 \\ 0 & 0 & Q_2 & 0 \\ 0 & 0 & 0 & Q_3 \end{pmatrix} \begin{pmatrix} Q_{01} & 0 \\ 0 & Q_{23} \end{pmatrix} Q_{0123} R_{0123} = QR.$$

The computation can be done in parallel (in particular it can be done with map-reduce) and the Q matrix for big problems would generally not be computed explicitly but would be stored in its constituent pieces.

Alternatively, there is a variant on the algorithm that processes the row-blocks of X serially, allowing you to do QR on a large tall-skinny matrix that you can't fit in memory (or possibly even on disk). First you do QR on X_0 to get $Q_0 R_0$. Then you stack R_0 on top of X_1 and do QR to get R_{01}. Then stack R_{01} on top of X_2 to get R_{012}, etc.

Determinants

The absolute value of the determinant of a square matrix can be found from the product of the diagonals of the triangular matrix in any factorization that gives a triangular (including diagonal) matrix times an orthogonal matrix (or matrices) since the determinant of an orthogonal matrix is either one or minus one.

$|A| = |QR| = |Q||R| = \pm|R|$

$|A^\top A| = |(QR)^\top QR| = |R^\top R| = |R_1^\top R_1| = |R_1|^2$

(Note of course that that is for square A, so we have R is \$n $\times n$.

In Python, the following will do it (on the log scale).

```
Q,R = qr(A)
magn = np.sum(np.log(np.abs(np.diag(R))))
```

An alternative is the product of the diagonal elements of D (the singular values) in the SVD factorization, $A = UDV^\top$.

For non-negative definite matrices, we know the determinant is non-negative, so the uncertainty about the sign is not an issue. For positive definite matrices, a good approach is to use the product of the diagonal elements of the Cholesky decomposition.

One can also use the product of the eigenvalues: $|A| = |\Gamma\Lambda\Gamma^{-1}| = |\Gamma||\Gamma^{-1}||\Lambda| = |\Lambda|$

Computation

Computing from any of these diagonal or triangular matrices as the product of the diagonals is prone to overflow and underflow, so we **always** work on the log scale as the sum of the log of the values. When some of these may be negative, we can always keep track of the number of negative values and take the log of the absolute values.

Often we will have the factorization as a result of other parts of the computation, so we get the determinant for free.

We can use `np.linalg.logdet()` or (definitely not recommended) `np.linalg.det()` to calculate the determinant in Python. These functions use the LU decomposition.

5. Eigendecomposition and SVD

Eigendecomposition

The eigendecomposition (spectral decomposition) is useful in considering convergence of algorithms and of course for statistical decompositions such as PCA. We think of decomposing the components of variation into orthogonal patterns (the eigenvectors) with variances (eigenvalues) associated with each pattern.

Square symmetric matrices have real eigenvectors and eigenvalues, with the factorization into orthogonal Γ and diagonal Λ, $A = \Gamma\Lambda\Gamma^\top$, where the eigenvalues on the diagonal of Λ are ordered in decreasing value. Of course this is equivalent to the definition of an eigenvalue/eigenvector pair as a pair such that $Ax = \lambda x$ where x is the eigenvector and λ is a scalar, the eigenvalue. The inverse of the eigendecomposition is simply $\Gamma\Lambda^{-1}\Gamma^\top$. On a similar note, we can create a square root matrix, $\Gamma\Lambda^{1/2}$, by taking the square roots of the eigenvalues.

The spectral radius of A, denoted $\rho(A)$, is the maximum of the absolute values of the eigenvalues. As we saw when talking about ill-conditionedness, for symmetric matrices, this maximum is the induced norm, so we have $\rho(A) = \|A\|_2$. It turns out that $\rho(A) \leq \|A\|$ for any induced matrix norm. The spectral radius comes up in determining the rate of convergence of some iterative algorithms.

Computation

There are several methods for eigenvalues; a common one for doing the full eigendecomposition is the *QR algorithm*. The first step is to reduce A to upper Hessenburg form, which is an upper triangular matrix except that the first subdiagonal in the lower triangular part can be non-zero. For symmetric matrices, the result is actually tridiagonal. We can do the reduction using Householder reflections or Givens rotations. At this point the QR decomposition (using Givens rotations) is applied iteratively (to a version of the matrix in which the diagonals are shifted), and the result converges to a diagonal matrix, which provides the eigenvalues. It's more work to get the eigenvectors, but they are obtained as a product of Householder matrices (required for the initial reduction) multiplied by the product of the Q matrices from the successive QR decompositions.

We won't go into the algorithm in detail, but note that it involves manipulations and ideas we've seen already.

If only the largest (or the first few largest) eigenvalues and their eigenvectors are needed, which can come up in time series and Markov chain contexts, the problem is easier and can be solved by the *power method*. E.g., in a Markov chain context, steady state is reached through $x_t = A^t x_0$. One can find the largest eigenvector by multiplying by A many times, normalizing at each step. $v^{(k)} = Az^{(k-1)}$ and $z^{(k)} = v^{(k)}/\|v^{(k)}\|$. There is an extension to find the p largest eigenvalues and their vectors. See the demo code in the qmd source file for an implementation (in R).

Singular value decomposition

Let's consider an $n \times m$ matrix, A, with $n \geq m$ (if $m > n$, we can always work with $A^\top$). This often is a matrix representing m features of n observations. We could have n documents and m words, or n gene expression levels and m experimental conditions, etc. A can always be decomposed as

$$A = UDV^\top$$

where U and V are matrices with orthonormal columns (left and right eigenvectors) and D is diagonal with non-negative values (which correspond to eigenvalues in the case of square A and to squared eigenvalues of $A^\top A$).

The SVD can be represented in more than one way. One representation is

$$A_{n\times m} = U_{n\times k} D_{k\times k} V^\top_{k\times m} = \sum_{j=1}^{k} D_{jj} u_j v_j^\top$$

where u_j and v_j are the columns of U and V and where k is the rank of A (which is at most the minimum of n and m of course). The diagonal elements of D are the singular values.

That representation is as the sum of rank-one matrices (since each term is the scaled outer product of two vectors).

If A is positive semi-definite, the eigendecomposition is an SVD. Furthermore, $A^\top A = VD^2V^\top$ and $AA^\top = UD^2U^\top$, so we can find the eigendecomposition of such matrices using the SVD of A (for $AA^\top$ we need to fill out U to have n columns). Note that the squares of the singular values of A are the eigenvalues of $A^\top A$ and $AA^\top$.

We can also fill out the matrices to get

$$A = U_{n\times n} D_{n\times m} V^\top_{m\times m}$$

where the added rows and columns of D are zero with the upper left block the $D_{k\times k}$ from above.

Uses

The SVD is an excellent way to determine a matrix rank and to construct a pseudo-inverse ($A^+ = VD^+U^\top$).

We can use the SVD to approximate A by taking $A \approx \tilde{A} = \sum_{j=1}^{p} D_{jj} u_j v_j^\top$ for $p < m$. The Eckart-Minsky-Young theorem shows that the truncated SVD minimizes the Frobenius norm of $A - \tilde{A}$ over all possible rank-p approximations. As an example if we have a large image of dimension $n \times m$, we could hold a compressed version by a rank-p approximation using the SVD. The SVD is used a lot in clustering problems. For example, the Netflix prize was won based on a variant of SVD (in fact all of the top methods used variants on SVD, I believe).

Here's another way to think about the SVD in terms of transformations and bases. Applying the SVD to a vector, $ UDV^{ }x$, carries out the following steps:

- $V^\top x$ expresses x in terms of weights for the columns of V.
- Multiplying the result by D scales/stretches the weights.

- Multiplying by U produces the result, which is a weighted combination of columns of U, spanning the column-space of U.

So applying the SVD transforms x from the column space of V to the column space of U.

Computation

The basic algorithm (Golub-Reinsch) is similar to the QR method for the eigendecomposition. We use a series of Householder transformations on the left and right to reduce A to an upper bidiagonal matrix, $A^{(0)}$. The post-multiplications (the transformations on the right) generate the zeros in the upper triangle. (An upper bidiagonal matrix is one with non-zeroes only on the diagonal and first subdiagonal above the diagonal). Then the algorithm produces a series of upper bidiagonal matrices, $A^{(0)}$, $A^{(1)}$, etc. that converge to a diagonal matrix, D . Each step is carried out by a sequence of Givens transformations:

$$\begin{aligned} A^{(j+1)} &= R_{m-2}^{\top} R_{m-3}^{\top} \cdots R_0^{\top} A^{(j)} T_0 T_1 \cdots T_{m-2} \\ &= R A^{(j)} T \end{aligned}.$$

This eventually gives $A^{(\cdots)} = D$ and by construction, U (the product of the pre-multiplied Householder matrices and the R matrices) and V (the product of the post-multiplied Householder matrices and the T matrices) are orthogonal. The result is then transformed by a diagonal matrix to make the elements of D non-negative and by permutation matrices to order the elements of D in nonincreasing order.

Computation for large tall-skinny matrices

The SVD can also be generated from a QR decomposition. Take $X = QR$ and then do an SVD on the R matrix to get $X = QUDV^{\top} = U^{*}DV^{\top}$. This is particularly helpful for the case when X is tall and skinny (suppose X is $n \times p$ with $n \gg p$), because we can do the tall-skinny QR, and the resulting SVD on R is easy computationally if p is manageable.

6. Computation

Linear algebra in Python

Note that for many matrix decompositions, you can change whether all of the aspects of the decomposition are returned, or just some, which may speed calculations.

Given the importance of linear algebra to many (most) statistical and machine learning algorithms, to improve efficiency of the algorithms, for large problems, it's important to see whether the linear algebra is being done using a fast linear algebra package in parallel, potentially on the GPU. How the linear algebra is done on the back end can make a huge difference in speed.

BLAS and LAPACK

In general, matrix operations in Python and R go to compiled C or Fortran code without much intermediate Python or R code, so they can actually be pretty efficient and are based on the best algorithms developed by numerical experts. The core libraries that are used are LAPACK and BLAS

(the Linear Algebra PACKage and the Basic Linear Algebra Subroutines). As we've discussed in the parallelization unit, one way to speed up code that relies heavily on linear algebra is to make sure you have a BLAS library tuned to your machine. These include OpenBLAS (open source), Intel's MKL, AMD's ACML, and Apple's vecLib. On newer Macs (Apple Silicon-based, namely using the M1, M2, M3, M4 chips), vecLib uses the Mac's AMX co-processor.

If you use Conda, numpy will generally be linked against MKL or OpenBLAS (this will depend on the locations online of the packages being installed, i.e., the *channel(s)* used). With pip, numpy will generally be linked against OpenBLAS. it's possible to install numpy so that it uses OpenBLAS. R can be linked to the shared object library file (*.so* file or *.dylib* on a Mac) for a fast BLAS. These BLAS libraries are also available in threaded versions that farm out the calculations across multiple cores or processors that share memory. More details are available in this SCF documentation.

BLAS routines do vector operations (level 1), matrix-vector operations (level 2), and dense matrix-matrix operations (level 3). Often the name of the routine has as its first letter "d", "s", "c" to indicate the routine is double precision, single precision, or complex. LAPACK builds on BLAS to implement standard linear algebra routines such as eigendecomposition, solutions of linear systems, a variety of factorizations, etc.

JAX and PyTorch (and GPUs)

Python packages such as JAX and PyTorch provide alternative linear algebra implementations that can exploit parallelization on CPUs and GPUs, automatically using the GPU if it is available. As discussed in Unit 5, JAX can also do just-in-time compilation.

The SCF parallelization tutorial has this example of doing matrix multiplication using PyTorch either on the CPU or a GPU.

The same tutorial also has an example of doing matrix multiplication using JAX, either on the CPU or on the GPU. Use of the CPU doesn't give much speed up compared to numpy since as noted above numpy linked to a parallel BLAS will use multiple threads.

Exploiting known structure in matrices

Speedups and storage savings can be obtained by working with matrices stored in special formats when the matrices have special structure. E.g., we might store a symmetric matrix as a full matrix but only use the upper or lower triangle. Banded matrices and block diagonal matrices are other common formats. Banded matrices are all zero except for $A_{i,i+c_k}$ for some small number of integers, c_k. Viewed as an image, these have bands. The bands are known as co-diagonals.

Scipy provides functionality for working with matrices in various ways, including the `scipy.sparse` module, which provides support for structured sparse matrices such as triangular and diagonal matrices as well as unstructured sparse matrices using various standard representations.

Some useful packages in R for matrices are *Matrix*, *spam*, and *bdsmatrix*. *Matrix* can represent a variety of rectangular matrices, including triangular, orthogonal, diagonal, etc. and provides methods for various matrix calculations that are specific to the matrix type. *spam* handles general sparse matrices with fast matrix calculations, in particular a fast Cholesky decomposition. *bdsmatrix* focuses on block-diagonal matrices, which arise frequently in contexts where there is clustering that induces within-cluster correlation and cross-cluster independence.

Sparse matrix formats

As an example of exploiting sparsity, we can use a standard format (CSR = compressed sparse row) in the Scipy `sparse` module:

Consider the matrix to be row-major and store the non-zero elements in order in an array called `data`. Then create a array called *indptr* that stores the position of the first element of each row. Finally, have a array, *indices* that tells the column identity of each element.

```
import scipy.sparse as sparse
mat = np.array([[0,0,1,0,10],[0,0,0,100,0],[0,0,0,0,0],[1000,0,0,0,0]])
mat = sparse.csr_array(mat)
mat.data
```

```
array([   1,   10,  100, 1000])
```

```
mat.indices  # column indices
```

```
array([2, 4, 3, 0], dtype=int32)
```

```
mat.indptr   # row pointers
```

```
array([0, 2, 3, 3, 4], dtype=int32)
```

```
## Ideally don't first construct the dense matrix if it is large.
mat2 = sparse.csr_array((mat.data, mat.indices, mat.indptr))
mat2.toarray()
```

```
array([[   0,    0,    1,    0,   10],
       [   0,    0,    0,  100,    0],
       [   0,    0,    0,    0,    0],
       [1000,    0,    0,    0,    0]])
```

That's also how things are done in the `spam` package in R.

We can do a fast matrix multiply, $x = Ab$, as follows in pseudo-code:

```
for(i in 1:nrows(A)){
    x[i] = 0
    # should also check that row is not empty...
    for(j in (rowpointers[i]:(rowpointers[i+1]-1)) {
        x[i] = x[i] + entries[j] * b[colindices[j]]
    }
}
```

How many computations have we done? Only k multiplies and $O(k)$ additions where k is the number of non-zero elements of A. Compare this to the usual $O(n^2)$ for dense multiplication.

Note that for the Cholesky of a sparse matrix, if the sparsity pattern is fixed, but the entries change, one can precompute an optimal re-ordering that retains as much sparsity in U as possible. Then multiple Cholesky decompositions can be done more quickly as the entries change.

Banded matrices

Suppose we have a banded matrix A where the lower bandwidth is p, namely $A_{ij} = 0$ for $i > j + p$ and the upper bandwidth is q ($A_{ij} = 0$ for $j > i + q$). An alternative to reducing to $Ux = b^*$ is to compute $A = LU$ and then do two solutions, $U^{-1}(L^{-1}b)$. One can show that the computational complexity of the LU factorization is $O(npq)$ for banded matrices, while solving the two triangular systems is $O(np + nq)$, so for small p and q, the speedup can be dramatic.

Banded matrices come up in time series analysis. E.g., moving average (MA) models produce banded covariance structures because the covariance is zero after a certain number of lags.

Low rank updates (optional)

A transformation of the form $A - uv^\top$ is a rank-one update because $uv^\top$ is of rank one.

More generally a low rank update of A is $\tilde{A} = A - UV^\top$ where U and V are $n \times m$ with $n \geq m$. The Sherman-Morrison-Woodbury formula tells us that

$$\tilde{A}^{-1} = A^{-1} + A^{-1}U(I_m - V^\top A^{-1}U)^{-1}V^\top A^{-1}$$

so if we know $x_0 = A^{-1}b$, then the solution to $\tilde{A}x = b$ is $x + A^{-1}U(I_m - V^\top A^{-1}U)^{-1}V^\top x$. Provided m is not too large, and particularly if we already have a factorization of A, then $A^{-1}U$ is not too bad computationally, and $I_m - V^\top A^{-1}U$ is $m \times m$. As a result $A^{-1}(U(\cdots)^{-1}V^\top x)$ isn't too bad.

This also comes up in working with precision matrices in Bayesian problems where we may have A^{-1} but not A (we often add precision matrices to find conditional normal distributions). An alternative expression for the formula is $\tilde{A} = A + UCV^\top$, and the identity tells us

$$\tilde{A}^{-1} = A^{-1} - A^{-1}U(C^{-1} + V^\top A^{-1}U)^{-1}V^\top A^{-1}$$

Basically Sherman-Morrison-Woodbury gives us matrix identities that we can use in combination with our knowledge of smart ways of solving systems of equations.

7. Iterative solutions of linear systems (optional)

Gauss-Seidel

Suppose we want to iteratively solve $Ax = b$. Here's the algorithm, which sequentially updates each element of x in turn.

- Start with an initial approximation, $x^{(0)}$.
- Hold all but $x_1^{(0)}$ constant and solve to find $x_1^{(1)} = \frac{1}{a_{11}}(b_1 - \sum_{j=2}^n a_{1j}x_j^{(0)})$.
- Repeat for the other rows of A (i.e., the other elements of x), finding $x^{(1)}$.
- Now iterate to get $x^{(2)}$, $x^{(3)}$, etc. until a convergence criterion is achieved, such as $\|x^{(k)} - x^{(k-1)}\| \leq \epsilon$ or $\|r^{(k)} - r^{(k-1)}\| \leq \epsilon$ for $r^{(k)} = b - Ax^{(k)}$.

Let's consider how many operations are involved in a single update: $O(n)$ for each element, so $O(n^2)$ for each update. Thus if we can stop well before n iterations, we've saved computation relative to exact methods.

If we decompose $A = L + D + U$ where L is strictly lower triangular, U is strictly upper triangular, then Gauss-Seidel is equivalent to solving

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

and we know that solving the lower triangular system is $O(n^2)$.

It turns out that the rate of convergence depends on the spectral radius of $(L + D)^{-1}U$.

Gauss-Seidel amounts to optimizing by moving in axis-oriented directions, so it can be slow in some cases.

Conjugate gradient

For positive definite A, conjugate gradient (CG) reexpresses the solution to $Ax = b$ as an optimization problem, minimizing

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

since the derivative of $f(x)$ is $Ax - b$ and at the minimum this gives $Ax - b = 0$.

Instead of finding the minimum by following the gradient at each step (so-called steepest descent, which can give slow convergence - we'll see a demonstration of this in the optimization unit), CG chooses directions that are mutually conjugate w.r.t. A, $d_i^\top Ad_j = 0$ for $i \neq j$. The method successively chooses vectors giving the direction, d_k, in which to move down towards the minimum and a scaling of how much to move, α_k. If we start at $x_{(0)}$, the kth point we move to is $x_{(k)} = x_{(k-1)} + \alpha_k d_k$ so we have

$$x_{(k)} = x_{(0)} + \sum_{j \leq k} \alpha_j d_j$$

and we use a convergence criterion such as given above for Gauss-Seidel. The directions are chosen to be the residuals, $b - Ax_{(k)}$. Here's the basic algorithm:

- Choose $x_{(0)}$ and define the residual, $r_{(0)} = b - Ax_{(0)}$ (the error on the scale of b) and the direction, $d_0 = r_{(0)}$ and set $k = 0$.
- Then iterate:
  - $\alpha_k = \frac{r_{(k)}^\top r_{(k)}}{d_k^\top Ad_k}$ (choose step size so next error will be orthogonal to current direction - which we can express in terms of the residual, which is easily computable)
  - $x_{(k+1)} = x_{(k)} + \alpha_k d_k$ (update current value)
  - $r_{(k+1)} = r_{(k)} - \alpha_k Ad_k$ (update current residual)
  - $d_{k+1} = r_{(k+1)} + \frac{r_{(k+1)}^\top r_{(k+1)}}{r_{(k)}^\top r_{(k)}} d_k$ (choose next direction by conjugate Gram-Schmidt, starting with $r_{(k+1)}$ and removing components that are not A-orthogonal to previous directions, but it turns out that $r_{(k+1)}$ is already A-orthogonal to all but d_k).
- Stop when $\|r^{(k+1)}\|$ is sufficiently small.

The convergence of the algorithm depends in a complicated way on the eigenvalues, but in general convergence is faster when the condition number is smaller (the eigenvalues are not too spread out). CG will in principle give the exact answer in n steps (where A is $n \times n$). However, computationally we lose accuracy and interest in the algorithm is really as an iterative approximation where we stop before n steps. The approach basically amounts to moving in axis-oriented directions in a space stretched by A.

In general, CG is used for large sparse systems.

See the extensive description from Shewchuk for more details, as well as the use of CG when A is not positive definite.

Updating a solution

Sometimes we have solved a system, $Ax = b$ and then need to solve $Ax = c$. If we have solved the initial system using a factorization, we can reuse that factorization and solve the new system in $O(n^2)$. Iterative approaches can do a nice job if $c = b + \delta b$. Start with the solution x for $Ax = b$ as $x^{(0)}$ and use one of the methods above.