Numerical linear algebra

Chris Paciorek

2024-10-31

Table of contents

Overview

PDF

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

```
array([ 3.61803399e+00,  2.61803399e+00,  1.38196601e+00, -4.84577457e-17,
        3.81966011e-01])
```

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

```
-4.440892098500626e-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. Stack the inner products of the rows of A with x.

```
for(i=1:n){
    b_i = 0
    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

```
for(i=1:n){
    b_i = 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 might be better for row-major matrices (so might be how we would implement in C) and the second for column-major (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 $\|A^{-1}\|_2$ that of 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)

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

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

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

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

```
8.198475468037087
```

```
0.0033319453118976702
```

```
2460.5672364315433
```

```
2984.092701676514
```

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 $x = \{1990, \dots, 2010\}$ (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

t1 = np.arange(1990, 2011)  # naive covariate
X1 = np.column_stack((np.ones(21), t1, t1 ** 2))

beta = np.array([5, .1, .0001])
y = X1 @ beta + np.random.normal(size = len(t1))

e1 = np.linalg.eig(np.dot(X1.T, X1))
np.sort(e1[0])[::-1]
np.linalg.cond(np.dot(X1.T, X1))  # built-in!
sm.OLS(y, X1).fit().params

t2 = t1 - 2000              # centered
X2 = np.column_stack((np.ones(21), t2, t2 ** 2))
e2 = np.linalg.eig(np.dot(X2.T, X2))
with np.printoptions(suppress=True):
    np.sort(e2[0])[::-1]

t3 = t2/10                  # centered and scaled
X3 = np.column_stack((np.ones(21), t3, t3 ** 2))
e3 = np.linalg.eig(np.dot(X3.T, X3))
with np.printoptions(suppress=True):
    np.sort(e3[0])[::-1]
```

```
array([3.36018564e+14, 7.69949718e+02, 2.01189323e-08])
```

```
5.667974329837199e+21
```

```
array([ 3.08551780e+04, -3.07510720e+01,  7.81281289e-03])
```

```
array([50677.70427505,   770.        ,     9.29572495])
```

```
array([24.11293487,  7.7       ,  1.95366513])
```

I haven't shown the OLS results for the transformed version of the problem because the transformations modify the true parameter values, so there is a bit of arithmetic to be done, but you should be able to verify that the second and third approaches give reasonable answers.