CS 542G: Conditioning, BLAS, LU Factorization

Robert Bridson

September 22, 2008

1 Why some RBF Kernel Functions Fail

We derived some sensible RBF kernel functions, like $\phi(r) = r^2 \log r$, from basic principles like minimizing curvature. The fact that we were seeking an interpolating function which minimizes a non-negative quantity (integral of Laplacian squared) gave us some confidence we would at least get a solution even if it might not be unique etc. This is not a proof of course, but a good step on the way to one. However other arbitrary choices of RBF kernel, such as $\phi(r) = r^2$ which doesn't look much different, can fail dramatically.

In the r^2 case, it's because we unwittingly are choosing a basis function which is a quadratic polynomial. (With odd powers like r or r^3, we don't get polynomials since the r is actually the norm of $\|x-x_i\|$ or in 1D the absolute value $|x - x_i|$, not a linear term—with quadratics or other even powers, that norm or absolute value doesn't make a difference: $|x-x_i|^2 = (x-x_i)^2$.) Thus we are accidentally restricting the "interpolant" $f(x)$ to be a quadratic polynomial, i.e. come from a small-dimensional space (the space of quadratic polynomials in 1D is 3-dimensional) independent of n. This means the linear system we have to solve can't have rank greater than this small number (in 1D, rank three), no matter how big n is: the linear system is singular and has no solution.

2 The Condition Number

Last time we arrived at a quantifiable measure of how well-posed a linear system $Ax = b$ is: the condition number $\kappa(A)$ of the matrix. No matter the algorithm chosen, relative errors in computing A and/or b, not to mention floating point rounding errors during solution, may be amplified by this much. Practically speaking, with double-precision floating point numbers, this puts an upper bound of about 10^{12} for $\kappa(A)$, give or take an order of magnitude or two, beyond which the problem is hopeless to solve as stated.

Furthermore, if software reports a condition number of 10^{16} or more (i.e. near the reciprocal of machine epsilon for floating point) it's highly likely the matrix is truly singular, and the notion of solving the linear system must be considered very carefully—more on this later. The above example of using $\phi(r) = r^2$ for RBF's, if you try it out numerically, illustrates this point: in double-precision floating point a package like MATLAB will estimate the condition number as $> 10^{16}$ but not infinite, simply due to round-off error slightly perturbing it back to non-singularity.

Incidentally, you may have noticed already a possible cause for concern with the condition number: from its definition $\kappa(A) = \|A\|\|A^{-1}\|$ it's not clear how easy this is to compute. Finding the inverse of a matrix is at least as hard as solving a linear system with the matrix. If we want to use the 2-norm, potentially difficult eigenvalue problems must be solved—it's a fairly simple exercise to show that it is:

$$\kappa_2(A) = \frac{\sqrt{\lambda_{\max}(A^TA)}}{\sqrt{\lambda_{\min}(A^TA)}}$$

i.e. the ratio of the square roots of the maximum and minimum eigenvalues of the matrix A^TA (we'll discuss this is much greater detail later). In fact, for the most part in practical computation the condition number is never calculated. Instead, it may be calculated exactly for small model problems, or numerically estimated through various means for large matrices, or theoretically bounded through knowledge of how the matrix was produced.

3 Software for Basic Linear Algebra Operations

Before finally getting to algorithms for solving a linear system, now is a great time for an aside about software for doing linear algebra. We'll visit several libraries throughout this course as they become relevant. The first one to know about is the **BLAS**, short for Basic Linear Algebra Subroutines, which is a standardized API for simple matrix and vector operations with many implementations available. See `www.netlib.org/blas` for the official definition, example source code, papers and more.

The BLAS original arose out of readability concerns: FORTRAN, for a long time the standard language of scientific computing, was in earlier incarnations restricted to six characters for identifiers, leading to cryptic function names such as `XERBLA` or `DGEMV`. Different scientific computing codes had their own names for common operations such as taking the dot-product of two vectors, equally incomprehensible, and while the update to FORTRAN to allow readable identifiers would come much later (in Fortran 90) at least the community could standardize on one set of cryptic function names for these operations.

Later, the big advantage of BLAS evolved into high performance more than readability: as architectures grew more complex, and more sophisticated implementation was required to achieve high performance, it became useful to let vendors do the hard work of optimizing BLAS routines that anyone could access through the API. Today's vendors (Intel, AMD, Apple, nVidia, Sun, etc.) do supply such optimized implementations; there are also some free versions such as ATLAS, or the Goto BLAS (currently the fastest on several machines). To varying degrees these exploit all the things necessary for good performance: vectorization, cache prefetching, careful blocking around cache line and page sizes, multithreading, etc. They also are well tested and should handle difficult cases properly.

The BLAS is divided into three levels, which came in that chronological order originally.

- Level 1 deals with vector operations that should run in $O(n)$ time (for vectors of length n): taking the norm of a vector, computing dot-products, adding two vectors, etc.
- Level 2 deals with matrix-vector operations that should run in $O(n^2)$ time (for $n \times n$ matrices): e.g. multiplying a matrix times a vector, solving a lower-triangular or upper-triangular linear system, etc.
- Level 3 deals with matrix-matrix operations that run in $O(n^3)$ time: e.g. multiplying two matrices together

Only the simplest operations are supported—for example, solving a general system of equations is not part of the BLAS—but they do cover many types of matrices, reflected in the naming convention of the routines. For example, the name of the level 2 routine `DGEMV` can be deconstructed as follows:

- The type prefix `D` indicates double-precision floating point real numbers (`S` indicates single-precision reals, `C` single-precision complex numbers, and `Z` double-precision complex numbers).
- The character pair `GE` indicates a general matrix is being used (other possibilities include symmetric matrices, symmetric matrices packed to use just over half the storage, banded matrices, triangular matrices...).
- The final `MV` identifies this as a matrix-vector multiply.

There are similarly functions `SGEMV`, `DHEMV`, `DGETR` to name a few variations. Further options include different "strides" for storing matrices, specifying that the transpose of the matrix be used, and more.

The BLAS are callable from FORTRAN 77, which makes them a little baroque when used from C for example: FORTRAN arguments are always passed by reference (i.e. as a pointer), including the scalars such as the dimension of a vector, and depending on the platform to be called from C the function names may need to be lowercase with an underscore appended as a suffix like `dgemv_`. There is an official C version of the API, the CBLAS, but it's not as widely supported. Personally I write my own user-friendly wrappers in C++ with readable names and some of the options I'm not in the habit of using filled in with default values.

The bottom line about the BLAS is that many scientific computing programs can be structured so most of the time is spent doing these primitive linear algebra operations, so it really pays to use the optimized and correct versions other people have spent their careers implementing. It's easy to write your own code to multiply two matrices, but particularly if done naïvely it's quite possible it will be an order of magnitude slower or worse than a good BLAS implementation for matrices of a decent size.

In fact, there should be a further bias: the higher levels of the BLAS afford more opportunities for optimization. If possible, you can get much higher performance by expressing algorithms in terms of the Level 3 BLAS—even though in principle these could be reduced to calls to level 2 which in turn could call the level 1 routines. Level 2 and especially level 1 are typically memory-bound: most numbers in the input get loaded into a register and used exactly once, meaning that even with cache prefetching and the like the CPU will often be waiting for memory rather than doing useful work. In level 3, on average, a number in the input will be used $O(n)$ times, and it may be possible to "block" the computation so that a number gets used many times per memory access, giving much higher performance.

4 Gaussian Elimination

Assuming a liner system is adequately well-conditioned to permit numerical solution, how do we solve it? There are many, many algorithms and variants on algorithms for solving linear systems, each with their own assumptions, weaknesses and strengths. For example, if the matrix is symmetric, or if it's sparse (i.e. most of its entries are zero, as opposed to a dense matrix where most or all entries could be nonzero), or if you already have a very good guess at the solution of the system, may all lead you to algorithms which can exploit these facts. Our first example of a linear system, derived from RBF interpolation, in fact was symmetric, but we'll hold off on exploiting that. For now, let's just assume that all we know about the matrix is that it's well-conditioned—it might be unsymmetric, dense, etc.—and we have no useful guess at what the solution is.

The simplest algorithm that works in this case is a version of Gaussian Elimination, or row reduction, which you should have seen in earlier linear algebra courses. This works by subtracting multiples of equations (rows in the matrix with their associated entry in the right-hand-side) off other equations, to eventually reduce the system to upper-triangular form which can be solved with back-substitution.

As an example, let's solve the following 3×3 linear system $Ax = b$:

$$\begin{pmatrix} 3 & -2 & 5 \\ -6 & 2 & -9 \\ 12 & -10 & 28 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ -3 \\ -4 \end{pmatrix}$$

We first try to zero out the entries below the diagonal in the first column, starting with the $(2, 1)$ entry by adding twice the first equation to the second:

$$\begin{pmatrix} 3 & -2 & 5 \\ 0 & -2 & 1 \\ 12 & -10 & 28 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ -1 \\ -4 \end{pmatrix}$$

We zero out the $(3, 1)$ entry by subtracting four times the first equation from the third:

$$\begin{pmatrix} 3 & -2 & 5 \\ 0 & -2 & 1 \\ 0 & -2 & 8 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ -1 \\ -8 \end{pmatrix}$$

Finally we zero out the $(3, 2)$ entry below the diagonal in the second column by subtracting the updated second equation from the updated third:

$$\begin{pmatrix} 3 & -2 & 5 \\ 0 & -2 & 1 \\ 0 & 0 & 7 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ -1 \\ -7 \end{pmatrix}$$

This is now an upper triangular system, which can be solve by backwards substitution starting from the last equation, $7x_3 = -7$, which gives $x_3 = -1$. This is substituted into the second equation, $-2x_2 + x_3 = -1$, which gives $x_2 = 0$. Finally both are substituted into the first equation, $3x_1 - 2x_2 + 5x_3 = 1$, to give $x_1 = 2$.

This form of the procedure can be formalized as a general algorithm, but it turns out there are some significant advantages gained by reworking it into a different form. We won't go through the details, but it turns out you can show the action of the elementary matrix operations (subtracting a multiple of one equation from another) can be assembled into the action of the inverse of a lower triangular matrix L with unit diagonal (i.e. all ones along the diagonal). The row reduction is equivalent to:

$$\begin{aligned} L^{-1}Ax &= L^{-1}b \\ Ux &= L^{-1}b \end{aligned}$$

Here $U = L^{-1}A$ is the upper triangular matrix we were left with.

In other words, we have implicitly **factored** the matrix as $A = LU$, with L unit lower triangular and U upper triangular.

5 LU Factorization

Expressing Gaussian elimination as an LU factorization offers several advantages. The first is the solution procedure: to solve $Ax = b$ we break it into three steps:

- Factor $A = LU$
- Find $c = L^{-1}b$, i.e. solve $Lc = b$, which can be done with forward substitution
- Solve $Ux = c$ with backwards substitution

It turns out the first step takes $O(n^3)$ time, but the following two steps are much cheaper at $O(n^2)$ time. If we have several systems to solve that use the same matrix A, we can amortize the cost of factorization over them for huge savings.

More importantly, even when there's only one system to solve, is that the factorization $A = LU$ allows for higher performance versions of the algorithm that exploit BLAS level 3, and it also allows for generally rather good estimates of the condition number of A.

5.1 Block Matrices

The tool we need to get at a good algorithm is block-partitioning: breaking up a matrix into submatrices or blocks. For example:

$$\left(\begin{array}{cc|c} 3 & -2 & 5 \\ -6 & 2 & -9 \\ \hline 12 & -10 & 28 \end{array}\right) = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$$

Here the blocks are:

$$\begin{aligned} A_{11} &= \begin{pmatrix} 3 & -2 \\ -6 & 2 \end{pmatrix} \\ A_{12} &= \begin{pmatrix} 5 \\ -9 \end{pmatrix} \\ A_{21} &= \begin{pmatrix} 12 & -10 \end{pmatrix} \\ A_{22} &= \begin{pmatrix} 28 \end{pmatrix} \end{aligned}$$

For the moment, we'll restrict our blocks to those resulting from partitioning the columns and the rows of the matrix same way, as we did here.

5.2 LU in a 2×2 Block Structure

We can now write down the $A = LU$ factorization in block form:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{pmatrix}$$

Notice the zero blocks in L and U, resulting from the fact they are triangular matrices. Multiplying out the blocks in LU gives:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11}U_{11} & L_{11}U_{12} \\ L_{21}U_{11} & L_{21}U_{12} + L_{22}U_{22} \end{pmatrix}$$

Peeling this apart gives us four matrix equations:

$$\begin{aligned} A_{11} &= L_{11}U_{11} \\ A_{12} &= L_{11}U_{12} \\ A_{21} &= L_{21}U_{11} \\ A_{22} &= L_{21}U_{12} + L_{22}U_{22} \end{aligned}$$

Different algorithms for LU can be easily derived from these equations (or slight variations thereof) just by choosing different block partitions.

For example, a classic divide-and-conquer approach would be to split A roughly in half along the rows and along the columns:

- If A is 1×1, the factorization is trivially $L = 1, U = A$ and you can return.
- Otherwise, split A in half along the rows and along the columns,
- then recursively factor $A_{11} = L_{11}U_{11}$,
- solve the block triangular system $L_{11}U_{12} = A_{12}$ for U_{12},
- solve the block triangular system $L_{21}U_{11} = A_{21}$ for L_{21} (which you may feel more comfortable writing as $U_{11}^T L_{21}^T = A_{21}^T$),
- form $\hat{A} = A_{22} - L_{21}U_{12}$,
- and recursively find the factorization $\hat{A} = L_{22}U_{22}$.

This has some significant performance advantages over classic row reduction since the middle three steps (solving the block triangular systems and forming $\hat{A}$), which is where all the work actually gets done, are BLAS Level 3 operations.

What amounts to classic row reduction can also be formulated this way, by partitioning A so that A_{11} is a 1×1 block and A_{22} is an $(n-1) \times (n-1)$ block. In this case, the algorithm looks like:

- Set $L_{11} = 1$ and $U_{11} = A_{11}$; stop if A is only 1×1
- Copy $U_{12} = A_{12}$ and set $L_{21} = A_{21}/U_{11}$
- Form $\hat{A} = A_{22} - L_{21}U_{12}$, and proceed with the LU factorization of $\hat{A}$

Note that in the last step $\hat{A}$ is formed by subtracting the outer product of the column vector L_{21} and the row vector U_{12}, which can easily be seen to be a rank one matrix (in fact, every rank one matrix can be expressed as an outer product of two vectors). This operation is often termed an **outer product update** or **rank one update** as a result. It can be interpreted in this case as subtracting multiples of the top row of A off the remaining rows to zero out the entries below the diagonal in U, which is classic row reduction. This variation of LU is also termed **right-looking**, since after finishing with a column of L we never deal with anything to the left but instead update all of the matrix to the right. (To be a bit more accurate, maybe this should be called right-and-down-looking, since the same is true for the rows of U, but that term hasn't caught on.)