Vectorised 5×5 Matrix Inversion

Taras Kolomatski

Choosing an Algorithm

I needed an inversion algorithm that was vectorisable. The instructions that I would run were to be the same for each matrix in my collection, and in particular had to avoid branching (for as long as possible, at least, there being a separate case of singular matrices).

First, I went to the Wikipedia page for matrix inversion, and having read that, I considered the following algorithms:

(1) `Gaussian Elimination`: What you're taught in a first linear algebra course, unless the instructor uses an approach in the style of Axler. Unless you re-order the rows, this is not numerically stable (indeed without branching matrices with many zero entries would give a divide by zero error). So neither is this vectorisable, nor does it lend itself to detection of degeneracy.

(2) `Factorisation`: LU or others. What you're probably taught in a second linear algebra course. This is the way to go for large matrices. I think that branching is necessary to implement this and I suspected that there was a way to save overhead apart from this in the 5×5 case.

(3) `Series approximation`: Won't give exact results in nice situations and it could be the case that I write this and subsequently realise that I would want to go an order that makes this inefficient.

(4) `Cayley-Hamilton`: Most theoretically rich approach, but I would need to find the sums products of the eigenvalues taken k at a time... *Apparently Bell functions do this. Initial reaction: Sorcery!* Then I read about and worked on this combinatorial problem for personal interest. In the 5×5 case you would only need the traces of the first four powers of each matrix, although taking the fifth power gives an immediate way to find the determinant, which we would otherwise do manually. This approach quickly becomes less desirable for larger dimensions, and hence taking it is a bold move. Obviously this.

The Cayley-Hamilton Approach

Consider a $n \times n$ matrix A. Let $\lambda_1, \ldots, \lambda_n \in \mathbb{C}$ denote its eigenvalues. Its characteristic polynomial is:

$$p = \prod_{j=1}^{n} (x - \lambda_j).$$

This is a monic polynomial. If we expand the product, then we get:

$$p = x^n + c_1 x^{n-1} + \ldots c_{n-1} x + c_n,$$

where, by Vieta,

$$c_k = (-1)^k \sum_{\substack{S \subset [n] \\ |S| = k}} \left(\prod_{j \in S} \lambda_j \right),$$

that is, the sum of products of the eigenvalues taken k at a time.

The Cayley-Hamilton theorem states that an operator satisfies its characteristic polynomial. In this case,

$$p(A) = A^n + c_1 A^{n-1} + \ldots + c_n I = 0.$$

Knowing this, we can rearrange:

$$-c_n I = A \left(A^{n-1} + c_1 A^{n-2} + \ldots + c_{n-1} I \right).$$

Note that $c_n = (-1)^n \det A$. Hence, exactly when A is non-singular, we obtain:

$$A^{-1} = \frac{-1}{c_n} \left(A^{n-1} + c_1 A^{n-2} + \ldots + c_{n-1} I \right).$$

Note that $c_1 = -\mathrm{Tr}A$. Let

$$s_k = \mathrm{Tr}A^k = \sum_{j=1}^{n} \lambda_j^k.$$

It is an amazing combinatorial fact that we can derive c_k given s_j for $j \leq k$. This lets us compute the characteristic polynomial of A without computing its eigenvalues! The identity is:

$$c_k = \frac{(-1)^k}{k!} B_k \left(s_1, -1!\ s_2, \ldots, (-1)^{k-1} k!\ s_k\right),$$

where B_k it the k-th complete Bell polynomial. These can be defined recursively with the following identities:

$$B_0 = 1$$

$$B_{k+1}(x_1 \ldots, x_{k+1}) = \sum_{j=0}^{k} \binom{k}{j} B_{k-j}(x_1, \ldots, x_{k-j}) x_{j+1}.$$

For specific cases, here 5×5, we can just hard code this computation up to the pertinent B_k.

Our algorithm is as follows. Always let $1 \leq j \leq 5$.

i *Compute the* A^j. (`mmult`)
ii *Take the traces to yield the* s_j. (`mtrace`)
iii *Multiply by constants to get the arguments of the Bell polynomials:* $s_1, -s_2, 2s_3, -6s_4, 24s_5$.
iv *Use these to compute the values of* B_j *for* $0 \leq j \leq 5$ *via the recursive definition.* (`Bell5`)
v *Multiply by constants to get the* c_n.
vi *Compute:*

$$\left(A^4 + c_1 A^3 + c_2 A^2 + c_3 A + c_4 I\right).$$

vii *If* $c_5 \nsim 0$, *multiply by* $-\frac{1}{c_5}$. *Otherwise multiply by 0.*
viii *Return the result.*

Note that, apart from choosing which multiple to use at the end, this algorithm involves no branching and is thus suitable to vectorisation.

Implementation

I skimmed the `xtensor` docs prior to starting. However, I first implemented a non-polished library-free implementation. (This can be found in the `cinvert.cpp` file.) This meant that I would have to implement functionality that would be otherwise provided, such as a pretty-print function for my matrices, which helped me debug the code. This was a correct choice as I went into struggling with `xtensor` while being sure of my algorithm.

I found `xtensor` to be poorly documented and struggled with learning the library. There were several typos when listing namespaces, for example, on the *Arrays and Tensors* tutorial page, the following appears: `xt::layout::row_major`. The type `xtensorf`, which was recently introduced, is essentially undocumented (it's only mentioned in a tutorial page). While this was stated to be non-realisable, it was not reasonable to conclude from the description that it did not support a dynamic layout, and hence that it was incompatible with, for example taking a transpose, which I tried to do with other features in `view`. Further, there is no explanation of vectorising arbitrary scalar functions. Eventually, I read through the most of the docs and came to an understanding of most of the library.

Roughly, I found *purely vectorisable* functionality to work very well, but could not get accumulation with user functions or imperative expression building to work in the way that I'm used to in functional languages (a large part of this difficulty arose from type matters and runtime errors). As a result, I have an absolutely beautiful implementation of naive matrix multiplication using the broadcasting rules (see

my pdf with a drawing explaining how this works), but an inelegant inversion algorithm body (I am well aware of the way this function ought to be structured). I was not able to successfully vectorise my code, but I expect that somebody with better knowledge of `xtensor` would be able to adapt my code with little alteration (plus it's really not something to which I should be allocating time at the point of this application).

My implementation is primarily contained in the `xinvert.cpp` sourse file. It's header provides the following typedefs:

- `reals`: the scalars on which our tensors are built, here `long double`.
- `x_lst`: a 1-D `xtensor` of `reals`.
- `x_mat`: a 2-D `xtensor` of `reals`.

And the following functions:

- `xinvert5`: Takes a `x_mat` by const reference, assumed be 5×5, and returns a `x_mat` which is either the inverse or zero if the argument is a singular matrix.
- `mprod`: Returns the product (a `x_mat`) of two arbitrary `x_mat` arguemnts passed by const reference, provided that they have compatible sizes.
- `mtrace`: Returns a `reals` that is the trace of a square `x_mat` arguemnt passed by const reference.
- `clearepsilons`: Takes a reference to a `x_mat` and mutates it by setting near zero entries to zero.

If the stated size assumptions are not met, then this should raise a runtime error.

Testing

The body of `main.cpp` uses `xtensor`'s random capabilities to a generate a random 5×5 matrix A with entries randomly sampled between a specified range. We compute A^{-1} and AA^{-1}. I wrote `clearepsilons` in order to make the final output to not be filled with entries absolutely smaller than 10^{-15}. Thus this final output should be either the identity matrix, or zero if A is singular. Apart from this, I tested my code at each point in the development process, but am not providing a test suite with this submission, although I have several commented off cases.

The inversion gives results accurate to our specified epsilon when entries are randomly selected between $\pm 10^{10}$, which is very good was not initially expected as I compute the fifth power of A. The size of `long double` is system dependant, so milage may vary.

Running the Project

I have provided a Makefile that produces the executable `invert` in the parent directory of the project. Run `make` to build and `make clean` to remove the built files.