1 GSW… Counting the Costs

I've mentioned a couple of times that a lot of linear algebra is about solving sets of simultaneous equations of the form $\mathbf{y} = \mathbf{A}\mathbf{x}$, where $\mathbf{y}$ and $\mathbf{A}$ are known and $\mathbf{x}$ is unknown; and the most obvious way to do this (inverting the matrix $\mathbf{A}$) is one we try and avoid. Why? What's so bad about inverting matrices? It certainly makes writing the answer very easy: $\mathbf{x} = \mathbf{A}^{-1}\mathbf{y}$.

The problem with working out matrix inverses is that it takes a lot of time (and energy). Any technique that promises us a solution without having to work out the matrix inverse would be worth knowing about. We've already seen one such technique: using Gaussian elimination on an extended matrix of the form:

$$\left[\begin{array}{ccc|c} A_{1,1} & A_{1,2} & A_{1,3} & y1 \\ A_{2,1} & A_{2,2} & A_{2,3} & y2 \\ A_{3,1} & A_{3,2} & A_{3,3} & y3 \end{array}\right] \tag{0.1}$$

How much better is this? How bad is working out the inverse? Is it worth trying to find any more efficient methods? Before we can answer these questions, we need a method of assessing how many calculations are required to do certain matrix operations.

Estimating the cost[1] of doing calculations on matrices can get very involved, as the actual amount of energy and number of clock cycles required to do some operations, and indeed the most efficient way to do the calculations, depends on the architecture of the hardware being used to implement the designs[2]. However, we can make some useful estimates of the work required to solve sets of simultaneous equations by dividing up the operations into two types: inversions, and multiply-and-accumulate operations.

These 'multiply-and-accumulate' instructions (usually called MACs for short[3]) are the basic building blocks of digital signal processing, and most integrated circuits (DSP processors and FPGAs) contain dedicated hardware to do these instructions very efficiently. The same is not true for divisions: divisions take much longer and more energy. For the purposes of the comparisons here, I'll assume I have a dedicated circuit capable of doing inversions[4], and divisions are done by first inverting the denominator, and then performing a multiplication.

[1] That's 'cost' in terms of time (usually counted in terms of clock cycles of the processor being used), or in terms of energy (since many devices have limited battery life). Quite often minimising one also minimises the other.

[2] For example, some digital signal processing chips (DSPs) may have a single processor with specialised instructions that can do certain sequences of operations very efficiently; whereas FPGA implementations may allow many operations to happen in parallel. This is one of those fun subjects where intelligence and good, original ideas really can make a difference, and clever tricks abound.

[3] MAC = Multiply-and-Accumulate Cycle.

[4] Perhaps a look-up table to get started, followed by a series of iterative steps to approach a more accurate inverse, based on multiplying the current approximate inverse by the original number, and seeing if the result is greater than one or not. Alternatively, the inversion could be done step-by-step by long division.

1.1 The Cost of Simple Multiplications

I'll start with the costs for some simple operations: multiplying two vectors, a vector by a matrix, and two matrices.

1.1.1 The Cost of a Multiplying Two Vectors

I'll assume I want the inner (scalar) product here. Then, the two vectors must have the same number of elements, say n, and to multiply the two vectors, we need to do:

$$\mathbf{x}^H\mathbf{y} = \begin{bmatrix} x_1^* & x_2^* & \dots & x_n^* \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \left(x_1^* y_1 + x_2^* y_2 + \dots + x_n^* y_n \right) \tag{0.2}$$

or in summation format:

$$s = \sum_{i=1}^{n} x^*_i y_i \tag{0.3}$$

and that means doing n multiplications, and adding up all the answers. This is exactly what a multiply-and-accumulate instruction does: take the next product term $x_i^* y_i$ and add it to the total so far. For example here, we'd get:

$$\begin{aligned} y_k &= x_1^* y_1 && \text{1 MAC operation} \\ &+ x_2^* y_2 && \text{2 MAC operations} \\ &+ x_3^* y_3 && \text{3 MAC operations} \\ &+ \dots \\ &+ x_n^* y_n && n \text{ MAC operations} \end{aligned} \tag{0.4}$$

So, a total of n MAC operations are required[5].

1.1.2 The Cost of a Multiplying a Vector by a Matrix

Now something slightly more difficult: pre-multiplying a vector (with n elements) by an m-by-n matrix.

$$\begin{bmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,n} \\ A_{2,1} & A_{2,2} & \cdots & A_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{m,1} & A_{m,2} & \cdots & A_{m,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \tag{0.5}$$

To determine each of the m elements in this product, we need to do the sum:

[5] Note that I'm ignoring taking the complex conjugate of the elements of **x**: usually that's easy and doesn't take much (if any) additional time. All that is required is inverting one bit: the sign bit of the imaginary part of the complex number.

$$y_k = \sum_{i=1}^{n} A_{k,i} x_i \tag{0.6}$$

and that means doing n multiplications, and adding up all the answers. That's n MACs. Since it takes a total of n MAC operations for each element of **y**, to complete the product of an m-by-n matrix and an n-element vector, we'll need to do mn MAC operations.

There's a variation on this that turns out to be important later: multiplying a vector by a square unit lower-triangular matrix.

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ L_{2,1} & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ L_{n,1} & L_{n,2} & \cdots & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \tag{0.7}$$

this requires no MACs at all for y_1 (it's just equal to x_1), one MAC for $y_2 = L_{2,1}x_1 + x_2$, two MACs for $y_3 = L_{3,1}x_1 + L_{3,2}x_2 + x_3$, and so on. The total number of MACs required for an n-by-n matrix and an n-element vector in this case is[6]:

$$0 + 1 + 2 + 3 + \ldots + (n-1) = \frac{n(n-1)}{2} \tag{0.8}$$

1.1.3 The Cost of Multiplying Two Matrices

If both matrices are square with n rows and n columns, and the elements can have any value, then the number of MACs required to find the matrix project is easy to calculate: each element in the product is the sum of n product terms; and there are n^2 elements in the matrix product, so that means we need to do a total of n^3 MAC cycles. That can be a lot when the matrices are large.

In the more general case of an n-by-m matrix multiplied by an m-by-p matrix, we'd need nmp MAC cycles.

Again, there's an important special case. If the first matrix happens to be upper-triangular (i.e. all terms below the main diagonal are zero), and the second matrix is unit lower-triangular (i.e. all terms above the main diagonal are zero and all terms on the main diagonal are one) then we can get away with doing rather less. Consider multiplying two such matrices, with six rows and six columns:

$$\begin{bmatrix} U_{1,1} & U_{1,2} & U_{1,3} & U_{1,4} & U_{1,5} & U_{1,6} \\ 0 & U_{2,2} & U_{2,3} & U_{2,4} & U_{2,5} & U_{2,6} \\ 0 & 0 & U_{3,3} & U_{3,4} & U_{3,5} & U_{3,6} \\ 0 & 0 & 0 & U_{4,4} & U_{4,5} & U_{4,6} \\ 0 & 0 & 0 & 0 & U_{5,5} & U_{5,6} \\ 0 & 0 & 0 & 0 & 0 & U_{6,6} \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ L_{2,1} & 1 & 0 & 0 & 0 & 0 \\ L_{3,1} & L_{3,2} & 1 & 0 & 0 & 0 \\ L_{4,1} & L_{4,2} & L_{4,3} & 1 & 0 & 0 \\ L_{5,1} & L_{5,2} & L_{5,3} & L_{5,4} & 1 & 0 \\ L_{6,1} & L_{6,2} & L_{6,3} & L_{6,4} & L_{6,5} & 1 \end{bmatrix} \tag{0.9}$$

[6] This is an arithmetic series. See the chapter 'WYNTKA Series' if unsure how to add up all these terms.

If I construct another matrix with elements that are the number of MAC operations required to calculate the corresponding term in the product, I'll get:

$$\begin{bmatrix} 5 & 4 & 3 & 2 & 1 & 0 \\ 5 & 4 & 3 & 2 & 1 & 0 \\ 4 & 4 & 3 & 2 & 1 & 0 \\ 3 & 3 & 3 & 2 & 1 & 0 \\ 2 & 2 & 2 & 2 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 \end{bmatrix} \tag{0.10}$$

which is a total of 70 MACs. (Notice it doesn't take any MACs to calculate the last column of the product, since this will be identical to the last column of the upper-triangular matrix **U**.)

In general, for a product of an *n*-by-*n* upper-triangular and an *n*-by-*n* unit lower triangular matrix, it takes at least $(n^3 - n) / 3$ MAC operations[7,8].

1.2 The Cost of Forward and Back Substitutions

Usually, after the Gaussian-elimination stage when solving systems of simultaneous equations, we're left with a set of equations of the form:

$$\begin{bmatrix} y_1 \\ \vdots \\ y_{n-1} \\ y_n \end{bmatrix} = \begin{bmatrix} U_{1,1} & \cdots & U_{1,n-1} & U_{1,n} \\ 0 & \ddots & \vdots & \vdots \\ \vdots & \vdots & U_{n-1,n-1} & U_{n-1,n} \\ 0 & \cdots & 0 & U_{n,n} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} \tag{0.11}$$

where **U** is upper-triangular, and we need to solve these equations using back-substitution. The first equation to solve is:

$$y_n = U_{n,n} x_n \tag{0.12}$$

so that requires just one division to find x_n (which with the hardware I'm assuming here is done with one inversion and one multiplication (MAC)), so in total this needs one MAC and one inversion. The second equation has the form:

$$\begin{aligned} y_{n-1} - U_{n-1,n} x_n &= U_{n-1,n-1} x_{n-1} \\ x_{n-1} &= \frac{1}{U_{n-1,n-1}} \left(y_{n-1} - U_{n-1,n} x_n \right) \end{aligned} \tag{0.13}$$

[7] Don't believe me? Count them. See the problems and solutions for how to work out things like this.

[8] In some cases, the complexity of the code required to reduce the number of MAC cycles to this minimum is just not worth it, and simpler code that actually does more MACs may be faster: but for the purposes of the comparisons in this chapter, I'll stick to using the minimum. Using this result turns out to be a pretty awful way to solve systems of simultaneous equations anyway, so it doesn't really matter, no-one ever does things this way.

so that's one MAC operation (subtract $U_{n,n-1}x_n$ from y_{n-1}) and one division, for a total of two MACs and an inversion. The third requires three MAC operations and one inversion, and so on, with the general term requiring i MACs and one inversion:

$$x_i = \frac{1}{U_{i,i}}\left(y_i - \sum_{j=i+1}^{n} U_{i,j}x_j \right) \tag{0.14}$$

The entire set of back-substitution equations then needs:

$$1+2+3+...+n = \frac{n(n+1)}{2} \tag{0.15}$$

MAC operations, and n divisions. Exactly the same is true for forward-substitution, you just do the sums in a different order.

Once again, there's another interesting case to consider: this time, when the matrix is a unit-upper triangular matrix (i.e. one with all terms on the main diagonal equal to one):

$$\begin{bmatrix} y_1 \\ \vdots \\ y_{n-1} \\ y_n \end{bmatrix} = \begin{bmatrix} 1 & \cdots & U_{1,n-1} & U_{1,n} \\ 0 & \ddots & \vdots & \vdots \\ \vdots & \vdots & 1 & U_{n,n-1} \\ 0 & \cdots & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} \tag{0.16}$$

The first equation to solve is now:

$$y_n = x_n \tag{0.17}$$

which doesn't require any MACs at all. The second equation is:

$$y_{n-1} - U_{n,n-1}x_n = x_{n-1} \tag{0.18}$$

which requires one MAC, but no divisions. In fact, all the equations in this case require one less MAC than the equivalent with a general upper-triangular matrix, since there is no need to do the division. The entire set of back-substitution equations in this case needs only:

$$0+1+2+3+...+(n-1) = \frac{n(n-1)}{2} \tag{0.19}$$

MAC operations, and no divisions. Again, the same is true for forward-substitution using a unit lower-triangular matrix.

1.3 The Cost of Gaussian Elimination

How many operations Gaussian elimination requires depends on what sort of elimination you're doing. The simplest case just takes a matrix **A** and converts it to upper-triangular form **U**; the most complex computes the inverse of the matrix **A**.

1.3.1 The Cost of Gaussian Elimination to U

This is the simplest possible form of Gaussian elimination. We don't use an extended matrix at all, all we do is convert the original matrix to upper triangular form. Starting with a matrix such as:

$$\begin{bmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,n} \\ A_{2,1} & A_{2,2} & \cdots & A_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n,1} & A_{n,2} & \cdots & A_{n,n} \end{bmatrix} \tag{0.20}$$

the first stage in the Gaussian elimination removes the term in $A_{2,1}$, and this requires one inversion ($1 / A_{1,1}$) and then ($n - 1$) MAC operations to compute the new values for the n elements on the second row (only $n - 1$ MACs are required, since there's no point in calculating the new value for $A_{2,1}$ we already know the answer will be zero). The same number of operations is required to remove each other term below the main diagonal, but all of the divisions are by the same factor: $A_{1,1}$. This is where our technique of calculating $1/A_{1,1}$ first helps: we just do this inversion once, and then we can work out the factors $A_{3,1}/A_{1,1}$ then $A_{4,1}/A_{1,1}$ up to $A_{n,1}/A_{1,1}$ with multiplications, rather than having to do a lot of more difficult divisions.

Doing this, we can reduce all the terms below $A_{1,1}$ in the first column to zero doing one inversion, ($n - 1$) multiplications to provide the factors, and then $(n - 1)^2$ MAC operations along the rows. Counting the multiplication as another MAC operation, this gives a total of one inversion and $(n - 1) + (n - 1)^2 = n(n - 1)$ MAC operations so far.

Moving on to the second column in **A**, we need to invert $A_{2,2}$, then do ($n - 2$) multiplications to get the factors required to multiply the second row before subtracting from the rows beneath, and then $(n - 2)^2$ MAC operations to subtract the multiples of the second row from the rows beneath. That's a total of one more inversion, and $(n - 1)(n - 2)$ MACs.

Keep on going through the whole matrix like this, and we'll find that reducing the entire matrix **A** to upper triangular form requires a total of ($n - 1$) inversions, and:

$$n(n-1)+(n-1)(n-2)+(n-2)(n-3)+\ldots+(2)(1)=\frac{n^3-n}{3} \tag{0.21}$$

MAC operations[9].

1.3.2 The Cost of Gaussian Elimination to z = Ux

Next, there's the slightly less-simple case that converts a system of simultaneous equations from the form:

$$\mathbf{y} = \mathbf{A}\mathbf{x} \tag{0.22}$$

into the form:

[9] This isn't an arithmetic series, or a geometric series, or anything as simple as that. For more about how to work out the sums of series like this one, see the problems and solutions.

$$\mathbf{z} = \mathbf{U}\mathbf{x} \tag{0.23}$$

where **U** is an upper-triangular matrix. This works by operations on the extended matrix:

$$\left[\begin{array}{cccc|c} A_{1,1} & A_{1,2} & \cdots & A_{1,n} & y_1 \\ A_{2,1} & A_{2,2} & \cdots & A_{2,n} & y_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ A_{n,1} & A_{n,2} & \cdots & A_{n,n} & y_n \end{array}\right] \tag{0.24}$$

The Gaussian elimination proceeds almost exactly as before, except we've got to extend each of the elimination stages along the rows by one element to include the components of the vector **y**. Counting up in a similar way to before, gives a total of $(n-1)$ inversions, and

$$(n+1)(n-1)+(n)(n-2)+(n-1)(n-3)+\ldots+(3)(1)=\frac{2n^3+3n^2-5n}{6} \tag{0.25}$$

MAC operations to reduce **A** to upper-triangular form and then calculate the vector **z** from **y**.

1.3.3 The Cost of Gaussian Elimination to Ly = Ux

Slightly more complex than this is the form of Gaussian elimination that preserves the value of **y**, and reduces the equation:

$$\mathbf{y} = \mathbf{A}\mathbf{x} \tag{0.26}$$

into the form:

$$\mathbf{L}\mathbf{y} = \mathbf{U}\mathbf{x} \tag{0.27}$$

where **U** is an upper-triangular matrix and **L** is a unit lower-triangular matrix. This one starts by creating the extended matrix:

$$\left[\begin{array}{cccc|cccc} A_{11} & A_{12} & \cdots & A_{1n} & 1 & 0 & \cdots & 0 \\ A_{21} & A_{22} & \cdots & A_{2n} & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ A_{m1} & A_{m2} & \cdots & A_{mn} & 0 & 0 & \cdots & 1 \end{array}\right] \tag{0.28}$$

Removing all the terms in the matrix **A** below the main diagonal takes more calculations than previously, since for most of the terms reduced to zero, we now have to do n MAC cycles. For example, the first one required produces the extended matrix:

$$\left[\begin{array}{cccc|cccc} A_{11} & A_{12} & \cdots & A_{1n} & 1 & 0 & \cdots & 0 \\ 0 & A_{22}-A_{12}A_{21}/A_{11} & \cdots & A_{2n}-A_{1n}A_{21}/A_{11} & -A_{21}/A_{11} & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nn} & 0 & 0 & \cdots & 1 \end{array}\right] \tag{0.29}$$

and as the number of MACs required for each elimination stage in the first n columns of the extended matrix decreases as you move across the matrix, the number required in the second n columns increases. The total is always n. So, you might think that to get the first matrix in

upper-triangular form (and the second matrix in lower-triangular form) now requires $(n-1)$ inversions and

$$(n+1)(n-1)+(n+1)(n-2)+(n+1)(n-3)+\ldots+(n+1)(1)=\frac{n^3-n}{2} \tag{0.30}$$

MAC operations. Except: look at this a bit more carefully, and you might notice that to calculate the terms immediately underneath the '1's on the main diagonal of the extended matrix requires a multiplication by one. That's easy: that doesn't require a MAC. So, that's $(n-1)$ MACs we've counted that in fact we don't really have to do. The minimum number of MACs required is actually:

$$\frac{n^3-n}{2}-(n-1)=\frac{n^3-3n+2}{2} \tag{0.31}$$

1.3.4 The Cost of Almost Inverting a Matrix

Finally, there is the cost of extending the elimination stages in the last case to reduce the first n-columns of the extended matrix to a diagonal matrix **D**, so we're left with:

$$\left[\begin{array}{cccc|cccc} D_{1,1} & 0 & \cdots & 0 & B_{1,1} & B_{1,2} & \cdots & B_{1,n} \\ 0 & D_{2,2} & \cdots & 0 & B_{2,1} & B_{2,2} & \cdots & B_{2,n} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & D_{n,n} & B_{n,1} & B_{n,2} & \cdots & B_{n,n} \end{array}\right] \tag{0.32}$$

The first step requires $(n^3-3n+2)/2$ MACs to get the first matrix into upper-triangular form, at which point we'd be left with:

$$\left[\begin{array}{cccc|cccc} U_{1,1} & U_{1,2} & \cdots & U_{1,n} & 1 & 0 & \cdots & 0 \\ 0 & U_{2,2} & \cdots & U_{2,n} & L_{2,1} & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & U_{n,n} & L_{n,1} & L_{n,2} & \cdots & 1 \end{array}\right] \tag{0.33}$$

The next step is to eliminate the term $U_{1,n}$, which can be done by subtracting $U_{1,n} / U_{n,n}$ times the last row from the first row. This requires one MAC to calculate the factor $U_{1,n} / U_{n,n}$ and then $(n-1)$ MACs to calculate the new terms in on the top row of the matrix L (just $n-1$ and not n, since we don't need to do the last one: that's just equal to $-U_{1,n} / U_{n,n}$ and we've already calculated that factor). The same is true for the rest of the terms above the term $\underline{U_{n,n}}$, so that requires $n(n-1)$ MACs to finish off that column. Oh, and we need to calculate $1 / U_{n,n}$ as well, we haven't required that inversion before, although all the other inversions we need (of the elements on the main diagonal of the first matrix) have already been done.

Continue through the matrix like this, and we'll find we need to do another:

$$n(n-1)+(n+1)(n-2)+(n+1)(n-3)+\ldots+(n+1)=\frac{n^3-3n+2}{2} \tag{0.34}$$

MACs (exactly the same number as we required to do the first stage). That makes a total of n inversions, n^3-3n+2 MACs to get the matrix this far.

Having done this, the matrix **B** is almost the inverse of the original matrix **A**. All we need to do is divide the elements in each row of **B** by the term on the main diagonal of the same row in **D**, and we'd have calculated the inverse. That, however, would take another n^2 MACs.

1.4 The Cost of Inverting a Matrix

We've just seen one method of calculating the matrix inverse: use Gaussian elimination to reduce to convert the equation **y** = **Ax** to the form **By** = **Dx**, where D is a diagonal matrix, at a total cost of n inversions and $n^3 - 3n + 2$ MACs.

All we need to do to complete the calculation of the matrix inverse is to divide each row of the extended matrix by the value of the term on the main diagonal of **B**, to give:

$$\left[\begin{array}{cccc|cccc} 1 & 0 & \cdots & 0 & B_{1,1}/D_{1,1} & B_{1,2}/D_{1,1} & \cdots & B_{1,n}/D_{1,1} \\ 0 & 1 & \cdots & 0 & B_{2,1}/D_{2,2} & B_{2,2}/D_{2,2} & \cdots & B_{2,n}/D_{2,2} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 & B_{n,1}/D_{n,n} & B_{n,2}/D_{n,n} & \cdots & B_{n,n}/D_{n,n} \end{array}\right] \tag{0.35}$$

and we've successfully reduced the original set of equations to the form **Cy** = **Ix** = **x**, where **C** is the matrix formed from the rightmost n columns of the extended matrix. Comparing this with the original equation **y** = **Ax**, shows that $\mathbf{A}^{-1} = \mathbf{C}$. This takes a further n^2 MACs, but no more inversions, since we've already calculated the inverse of all the terms on the main diagonal of **D**.

So, that's a complete matrix inverse calculated for a total of:

$$\frac{n^3 - 3n + 2}{2} + \frac{n^3 - 3n + 2}{2} + n^2 = n^3 + n^2 - 3n + 2 \tag{0.36}$$

MACs, and n inversions.

1.4.1 The Cost of Inverting an Upper-Triangular Matrix

There's another interesting case here: what if the original matrix was upper-triangular (or lower-triangular, it doesn't affect the number of operations required). This matrix inverse can conveniently be done by calculating the co-efficients in the inverse directly, using the useful fact that the inverse of an upper-triangular matrix is another upper-triangular matrix. For example, consider the matrix product:

$$\begin{bmatrix} U_{1,1} & U_{1,2} & \cdots & U_{1,n} \\ 0 & U_{2,2} & \cdots & U_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & U_{n,n} \end{bmatrix} \begin{bmatrix} V_{1,1} & V_{1,2} & \cdots & V_{1,n} \\ 0 & V_{2,2} & \cdots & V_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & V_{n,n} \end{bmatrix} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix} \tag{0.37}$$

where $\mathbf{V} = \mathbf{U}^{-1}$. The first thing to do is work out the inverses of all the terms on the main diagonal of the upper-triangular matrix **U** (that's $U_{1,1}$, $U_{2,2}$, $U_{3,3}$ and so on). That requires n inversions, and the results are the terms $U_{1,1}$, $U_{2,2}$, $U_{3,3}$, since for all the terms on the main diagonal of **V**, we get:

$$V_{i,i} = \frac{1}{U_{i,i}} \tag{0.38}$$

(try working out the product terms that give the '1' terms in the unit matrix to see where this comes from).

Next, consider the terms one along from the main diagonal in **V**: $V_{1,2}$, $V_{2,3}$, $V_{3,4}$ and so on up to $V_{n-1,n}$. All of these can be calculated using expressions requiring two multiplications (which I'll count as two MACs) for example:

$$\begin{aligned} U_{1,1}V_{1,2} + U_{1,2}V_{2,2} &= 0 \\ V_{1,2} &= -\frac{U_{1,2}V_{2,2}}{U_{1,1}} = -U_{1,2}V_{2,2}V_{1,1} \end{aligned} \tag{0.39}$$

Then, consider the terms two along from the main diagonal in **V**: $V_{1,3}$, $V_{2,4}$, $V_{3,5}$ and so on up to $V_{n-2,n}$. All of these can be calculated using expressions requiring two MACs and a further multiplication (which I'll count as three MACs) for example:

$$\begin{aligned} U_{1,1}V_{1,3} + U_{1,2}V_{2,3} + U_{1,3}V_{3,3} &= 0 \\ V_{1,3} &= -\frac{U_{1,2}V_{2,3} + U_{1,3}V_{3,3}}{U_{1,1}} = -\left(U_{1,2}V_{2,3} + U_{1,3}V_{3,3}\right)V_{1,1} \end{aligned} \tag{0.40}$$

and so on. The general terms is:

$$V_{i,j} = -V_{i,i}\sum_{k=i+1}^{n} U_{i,k}V_{k,j} \tag{0.41}$$

Again, I can write out a matrix with elements equal to the number of MAC operations required to calculated each element of **V**, and I'll get (for a six-by-six matrix):

$$\begin{bmatrix} 0 & 2 & 3 & 4 & 5 & 6 \\ 0 & 0 & 2 & 3 & 4 & 5 \\ 0 & 0 & 0 & 2 & 3 & 4 \\ 0 & 0 & 0 & 0 & 2 & 3 \\ 0 & 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{0.42}$$

so inverting this matrix requires a total of 50 MACs and 6 inversions. In the general case, for an *n*-by-*n* upper-triangular matrix, we'd need:

$$n + 2(n-1) + 3(n-2) + 4(n-3) + \ldots + n(1)2 = \frac{n^3 + 3n^2 - 4n}{6} \tag{0.43}$$

MACs and *n* inversions. For large *n*, this is six times fewer MACs than if the matrix was full.

1.4.2 Another Cost of Inverting a Matrix

Just for interest, there is another way to invert matrices that requires slightly fewer MACs than the method described above. Consider the equation:

$$\mathbf{Iy} = \mathbf{Ax} \tag{0.44}$$

then use Gaussian elimination to manipulate the matrix **A** into upper triangular format and the matrix **I** into a unit lower-triangular matrix, giving:

$$\mathbf{Ly} = \mathbf{Ux} \tag{0.45}$$

Then, invert the upper-triangular matrix, and multiply both sides of the equation by the matrix $\mathbf{U}^{-1}$:

$$\left(\mathbf{U}^{-1}\mathbf{L}\right)\mathbf{y} = \mathbf{U}^{-1}\mathbf{Ux} = \mathbf{x} \tag{0.46}$$

Clearly, comparing this to equation (0.44), the matrix ($\mathbf{U}^{-1}\mathbf{L}$) is the inverse of matrix **A**.

To get this far we need to:

1. Gaussian elimination to find **L** and **U**: $(n^3 - n) / 2$ MACs and $(n - 1)$ inversions.
2. Inversion of upper-triangular matrix **U**: $(n^3 + 3n^2 - 4n) / 6$ MACs and n inversions.
3. Multiplication of **L** and $\mathbf{U}^{-1}$ to find $\mathbf{A}^{-1}$: $(n^3 - n) / 3$ MACs

but things are not this bad, since the inversions required in steps 1 and 2 are mostly the same: they are the inverses of the terms on the main diagonal of the upper-triangular matrix. The grand total is $(2n^3 + n^2 - 3n) / 2$ MACs and n inversions. That's slightly fewer (e.g. for a 10-by-10 matrix this would require 1026 MACs rather than 1072), the method is more complex, and probably isn't worth the additional length of program required.

Note that for large, or even medium size matrices, the number of MACs is *much* larger than the number of inversions, so in many cases we can neglect the number of inversions required and just count the MACs.

1.5 Summary of the Costs So Far

Where **A** and **B** are n-by-n square matrices, **L** is an n-by-n unit lower triangular matrix, **U** is an n-by-n upper-triangular matrix and **x** is an n-element vector, the total number of MACs and other operations required by the operations discussed so far is:

	MACs	Inversions
Multiply **Ax**	n^2	0
Multiply **AB**	n^3	0
Multiply **UL**	$(n^3 - n) / 3$	0
Back-substitution to solve **y** = **Ux**	$(n^2 + n) / 2$	n
Forward substitution[10] to solve **y** = **Lx**	$(n^2 - n) / 2$	0

[10] Remember the only reason the back-substitution and forward-substitution have different numbers of MACs is that **L** is assumed to be a unit lower-triangular matrix (one with a main diagonal with all '1's on it). If **L** was any other lower-triangular matrix, then it would need $(n^2 + n) / 2$ MACs, just like back-substitution with **U**.

Gaussian elimination of **A** to upper-triangular **U**	$(n^3 - n) / 3$	$n - 1$
Gaussian elimination of **y** = **Ax** to **z** = **Ux**	$(2n^3 + 3n^2 - 5n) / 6$	$n - 1$
Gaussian elimination of **y** = **Ax** to **Ly** = **Ux**	$(n^3 - 3n + 2) / 2$	$n - 1$
Inverting matrix **A**	$n^3 + n^2 - 3n + 2$	n
Inverting upper-triangular matrix **U**	$(n^3 + 3n^2 - 4n) / 6$	n

1.6 Tutorial Questions

1) Suppose I wanted to calculate $\mathbf{x}^H\mathbf{A}\mathbf{x}$ where **A** is a square n-by-n matrix and **x** is an n-element vector. How many MACs does this require?

2) Prove that:

$$n(n-1)+(n-1)(n-2)+(n-2)(n-3)+\ldots+(2)(1)=\frac{n^3-n}{3}$$

(Hint: try a proof by induction. Show that it is true for $n = 1$, and then that if it is true for $n = k$, it is also true for $n = k + 1$.)

3) How many operations does it take to invert a unit upper-triangular matrix?