GAUSS-JORDAN ELIMINATION: A METHOD TO SOLVE LINEAR SYSTEMS

MATH 196, SECTION 57 (VIPUL NAIK)

Corresponding material in the book: Section 1.2.

Executive summary

(1) A system of linear equations can be stored using an *augmented matrix*. The part of the matrix that does not involve the constant terms is termed the *coefficient matrix*.
(2) Variables correspond to columns and equations correspond to rows in the coefficient matrix. The augmented matrix has an extra column corresponding to the constant terms.
(3) In the paradigm where the system of linear equations arises from an attempte to determine the parameters of a model (that is linear in the parameters) using (input,output) pairs, the parameters of the model are the new variables, and therefore correspond to the columns. The (input,output) pairs correspond to the equations, and therefore to the rows. The input part controls the part of the row that is in the coeficient matrix, and the output part controls the augmenting entry.
(4) If the coefficient matrix of a system of simultaneous linear equations is in reduced row-echelon form, it is easy to read the solutions from the coefficient matrix. Specifically, the non-leading variables are the parameters, and the leading variables are expressible in terms of those.
(5) In reduced row-echelon form, the system is inconsistent if and only if there is a row of the coefficient matrix that is all zeros with the corresponding augmented entry nonzero. Note that if the system is *not* in reduced row-echelon form, it is *still* true that a zero row of the coefficient matrix with a nonzero augmenting entry implies that the system is inconsistent, but the converse does not hold: the system may well be inconsistent despite the absence of such rows.
(6) In reduced row-echelon form, if the system is consistent, the dimension of the solution space is the number of non-leading variables, which equals (number of variables) - (number of nontrivial equations). Note that the all zero rows give no information.
(7) Through an appropriately chosen sequence of row operations (all reversible), we can transform any linear system into a linear system where the coefficient matrix is in reduced row-echelon form. The process is called Gauss-Jordan elimination.
(8) The process of Gauss-Jordan elimination happens entirely based on the coefficient matrix. The final column of the augmented matrix is affected by the operations, but does not control any of the operations.
(9) There is a quicker version of Gauss-Jordan elimination called Gaussian elimination that converts to row-echelon form (which is triangular) but not reduced row-echelon form. It is quicker to arrive at, but there is more work getting from there to the actual solution. Gaussian elimination is, however, sufficient for determining which of the variables are leading variables and which are non-leading variables, and therefore for computing the dimension of the solution space and other related quantities.
(10) The arithmetic complexity of Gauss-Jordan elimination, in both space and time terms, is polynomial in n. To get a nicely bounded bit complexity (i.e., taking into account the sizes of the numbers blowing up) we need to modify the algorithm somewhat, and we then get a complexity that is jointly poynomial in n and the maximum bit length.
(11) Depending on the tradeoff between arithmetic operations and using multiple processors, it is possible to reduce the arithmetic complexity of Gauss-Jordan elimination considerably.
(12) Gauss-Jordan elimination is not conceptually different from iterative substitution. Its main utility lies in the fact that it is easier to code since it involves only numerical operations and does not require the machine to understand equation manipulation. On the other hand, because the operations are

purely mechanical, it may be easier to make careless errors because of the lack of "sense" regarding the operations. To get the best of both worlds, use the Gauss-Jordan elimination procedure, but keep the symbolic algebra at the back of your mind while doing the manipulations.

(13) The Gauss-Jordan elimination process is only one of many ways to get to reduced row-echelon form. For particular structures of coefficient matrices, it may be beneficial to tweak the algorithm a little (for instance, swapping in an easier row to the top) and save on computational steps. That said, the existence of the Gauss-Jordan elimination process gives us a guarantee that we can reach our goal by providing one clear path to it. If we can find a better path, that's great.

(14) We can use the Euclidean algorithm/gcd algorithm/Bareiss algorithm. This is a variant of the Gauss-Jordan algorithm that aims for the same end result (a reduced row-echelon form) but chooses a different sequencing and choice of row operations. The idea is to keep subtracting multiples of rows from one another to get the entries down to small values (hopefully to 1), rather than having to divide by the leading entry. It's not necessary to master this algorithm, but you might find some of its ideas helpful for simplifying your calculations when solving linear systems.

1. Notational setup

1.1. **Augmented matrix and coefficient matrix.** Storing equations requires storing letters, operations, and equality symbols. For linear equations, however, there is a predictable manner in which the operations are arranged. Hence, linear equations can be stored and manipulated more compactly simply by storing certain *numbers*, namely the coefficients of the variables and the constant term, in a structured form.

1.1.1. *Example 1.* For instance, the system:

$$\begin{aligned} 2x + 3y + z &= 5 \\ x - 4y + 5z &= 7 \\ 11x + 3y + 6z &= 13 \end{aligned}$$

can be described using this box of numbers, called the *augmented matrix*:

$$\left[\begin{array}{ccc|c} 2 & 3 & 1 & 5 \\ 1 & -4 & 5 & 7 \\ 11 & 3 & 6 & 13 \end{array}\right]$$

Note that this makes sense *only* if we agree beforehand what it's being used for. Explicitly, each row represents one equation, with the first entry representing the coefficient of x, the second entry representing the coefficient of y, the third entry representing the coefficient of z, and the final entry representing the constant term *placed on the opposite side of the equality symbol*.

1.1.2. *Example 2.* Note that, unlike the example used above, some linear systems may be written in a way where we cannot just "read off" the augmented matrix from the system, because the variables appear in different orders. The first step there is to rearrange each equation so that we can read off the coefficients. For instance, consider the system:

$$\begin{aligned} x - 3y + 5z &= 17 \\ y - 2(z - x) &= 5 \\ x + y + 2(z - 3y + 4) &= 66 \end{aligned}$$

Each equation is linear in each of the variables x, y, and z. However, the variables appear in different orders in the equations. Moreover, the third equation contains multiple occurrences of y and also contains constant terms on both sides. We need to simplify and rearrange the second and third equation before being able to read off the coefficients to construct the augmented matrix. Here's what the system becomes:

$$\begin{aligned} x - 3y + 5z &= 17 \\ 2x + y - 2z &= 5 \\ x - 5y + 2z &= 58 \end{aligned}$$

The augmented matrix can now be read off:

$$\left[\begin{array}{ccc|c} 1 & -3 & 5 & 17 \\ 2 & 1 & -2 & 5 \\ 1 & -5 & 2 & 58 \end{array}\right]$$

1.1.3. *Return to Example 1 for the coefficient matrix.* There is a related concept to the augmented matrix, namely the *coefficient matrix*. The coefficient matrix ignores the constant terms (the last column in the augmented matrix). Recall our first example:

$$\begin{aligned} 2x + 3y + z &= 5 \\ x - 4y + 5z &= 7 \\ 11x + 3y + 6z &= 13 \end{aligned}$$

The coefficient matrix for this is:

$$\begin{bmatrix} 2 & 3 & 1 \\ 1 & -4 & 5 \\ 11 & 3 & 6 \end{bmatrix}$$

1.2. **Dimensions of the coefficient matrix and augmented matrix.** If there are n variables and m equations, then the coefficient matrix is a $m \times n$ matrix: m rows, one for each equation, and n columns, one for each variable. The augmented matrix is a $m \times (n+1)$ matrix. The additional column is for the constant terms in the system.

For a typical precisely determined linear system, we expect that the number of equations equals the number of variables. Thus, if there are n variables, we expect n equations. The coefficient matrix is thus expected to be a $n \times n$ matrix. A matrix where the number of rows equals the number of columns is termed a *square matrix*, so another way of putting it is that the coefficient matrix is a square matrix. The augmented matrix is expected to be a $n \times (n+1)$ matrix, with the extra column needed for the constant terms.

Explicitly:

- Variables correspond to columns, so (number of variables) = (number of columns in coefficient matrix) = (number of columns in augmented matrix) - 1.
- Rows correspond to equations, so (number of equations) = (number of rows in coefficient matrix) = (number of rows in augmented matrix).
- (Number of variables) - (Number of equations) = (number of columns in coefficient matrix) - (number of rows in coefficient matrix). In an ideal world, this number should be the dimension of the solution space. The world is not always ideal, something we shall turn to in a while.

1.3. **The application to parameter estimation (you won't understand this completely just by reading, so pay attention in class and/or return to this later again).** Recall that if we have a functional model that is linear in the parameters (though not necessarily in the variables) we can use (input,output) pairs to construct a system of linear equations that we can then solve for the parameters.

We now note that:

- The coefficient matrix is completely determined by the *input* part of the input-output pairs.
- The additional column in the augmented matrix that stores the constant terms of the linear system is completely determined by the *output* part of the input-output pairs.

This is significant in a number of ways. As we will see in matrix theory, a lot of the basic nature of a linear system is controlled by the coefficient matrix. Roughly, if the coefficient matrix satisfies certain conditions, then the system has no redundancy and no inconsistency. If the coefficient matrix does not satisfy these conditions, then the system has potential for either redundancy or inconsistency. Which of these cases occurs depends on the outputs obtained.

We will also study techniques to solve linear systems that do the bulk of their processing on the coefficient matrix alone. An advantage of using such techniques is that much of the solution process can be run even without knowing the outputs, and the part of the algorithm that relies on the outputs can be done quickly. This is useful for two reasons: *amortization in repeated application*, and *faster processing once the outputs are known*.

To be more explicit: In some cases, the same model is being used for many different situations with different parameter values. For instance, a particular model of the economy may be used for different historic eras and different countries, where we expect the parameter values to be different. We can thus standardize a choice of inputs in input-output pairs that is common to all the uses of the model, then preprocess the solution procedure, and thus apply this to each of the model situations quickly on receiving the outputs. Similar remarks apply to models such as weather and traffic systems that have a fixed pattern of input collection at periodic intervals and need to rapidly process the outputs obtained at a particular interval to estimate the parameters involved.

2. Row reduction

Row reduction is a process for manipulating a system of linear equations to obtain an equivalent system of linear equations that is easier to solve. In this context, *equivalent* means that the solutions to the old system are *precisely* the same as the solutions to the old systems. Another way of thinking of this is that we can deduce the new system from the old system *and* we can deduce the old system from the new system. In other words, all the transformations that we do on the system of equations need to be *reversible transformations*.

We saw in a previous lecture that given a system of equations:

$$\begin{aligned} F(x_1, x_2, \ldots, x_n) &= 0 \\ G(x_1, x_2, \ldots, x_n) &= 0 \end{aligned}$$

and a real number λ, we can transform this to a new system:

$$\begin{aligned} F(x_1, x_2, \ldots, x_n) + \lambda G(x_1, x_2, \ldots, x_n) &= 0 \\ G(x_1, x_2, \ldots, x_n) &= 0 \end{aligned}$$

This type of transformation is sometimes called a *shear operation* based on a geometric interpretation. The shear operation by λ is reversible because its inverse is the shear operation by $-\lambda$.

We also saw that shear operations (and more generally, adding and subtracting equations) are typically used in order to eliminate certain variables or certain expressions involving variables.

The shear will be the most important and nontrivial of the three types of *elementary operations* that we use in order to simplify linear systems. The three operations are:

(1) Multiply or divide an equation by a nonzero scalar.
(2) Add or subtract a multiple of an equation to another equation (the shear operation).
(3) Swap two equations.

Interestingly, all these operations can be done purely as operations on the augmented matrix, without having to (tediously) write down the variables and operational symbols. The operations on the augmented matrix work in the same way. Explicitly, the three operations are:

(1) Multiply or divide a row by a nonzero scalar.
(2) Add or subtract a multiple of a row to another row (the shear operation).
(3) Swap two rows.

2.1. **Reduced row-echelon form.** The reduced row-echelon form can be thought of as trying to capture the idea of a triangular system where some equations are missing. It further restricts attention to particularly nice triangular forms that cannot be made to look nicer.

A matrix is said to be in reduced row-echelon form if it satisfies the following conditions:

(1) For every row, the left-most nonzero entry, if any, is a 1. We will call this the *pivotal* entry for the row and the column where this appears the pivotal column. It is possible for a row to be all zeros.
(2) The pivotal column for a later (lower) row is a later (more to the right) column, if it exists. If a given row is all zeros, all later rows are also all zeros.
(3) Any column containing a pivotal 1 must have all other entries in it equal to 0. This has two subparts:
   (a) All entries in the column below the pivotal 1 are equal to 0 (note that this is redundant if we assume both 1 and 2).
   (b) All entries in the column above the pivotal 1 are equal to 0.

Of the above, 3(b), while part of the definition of reduced row-echelon form, is not necessary for the system to be triangular. Condition 1 is also not strictly necessary, i.e., we can relax that condition and use the term *pivotal entry* for the left-most nonzero-valued entry in the row. The main advantage of 1 and 3(b) is mostly in terms of computational simplicity, both as far as the process for obtaining the echelon form is concerned (something we discuss in a subsequent section), and as far as using it to find the actual solution is concerned.

Conditions 2 and 3(a) are the conditions that really enforce the triangular nature of the system.

2.2. **Solving a linear system whose coefficient matrix is in reduced row-echelon form.** If the coefficient matrix for a system of linear equations is in reduced row-echelon form, then we say that the *leading variable* (also called the pivotal variable) for a particular equation is the variable corresponding to the pivotal column for that row.

Consider the following linear system in four variables x_1, x_2, x_3, and x_4:

$$\begin{aligned} x_1 - 2x_2 + x_4 &= 5 \\ x_3 + 5x_4 &= 31 \end{aligned}$$

The coefficient matrix for this system is as follows:

$$\begin{bmatrix} 1 & -2 & 0 & 1 \\ 0 & 0 & 1 & 5 \end{bmatrix}$$

The leading 1 for the first row occurs in the first column. The leading 1 for the second row occurs in the third column. Note that the first nonzero entry in each row is a 1, and columns for later rows are later columns. This confirms conditions 1 and 2. Condition 3 is also clearly true here.

To solve a system of this sort, we start from the variables corresponding to the later columns and move backward to the variables corresponding to the earlier columns. First, consider the variable x_4, corresponding to the last column. This column is not pivotal for any row. So, x_4 is free to vary over all reals. It is helpful to name it using a letter customarily used to describe a parameter, such as the letter t.

The column for x_3 is pivotal for the second equation. Thus, the second equation can be used to deduce the value of x_3 based on the values of all the later variables. In this case, the equation is:

$$x_3 + 5x_4 = 31$$

Note that $x_4 = t$, so we get:

$$x_3 = 31 - 5t$$

x_2 does not appear as a leading variable for any row. Thus, it is again free to vary over all reals. Let s be the freely varying value of x_2.

This leaves x_1, which is pivotal for the first equation. We have the equation:

$$x_1 - 2x_2 + x_4 = 5$$

This gives:

$$x_1 - 2s + t = 5$$

This simplifies to:

$$x_1 = 5 + 2s - t$$

Thus, our solution set is:

$$\{(5 + 2s - t, s, 31 - 5t, t) : s, t \in \mathbb{R}\}$$

Note that if we had a third equation in the system which read:

$$0x_4 = 1$$

the coefficient matrix would be:

$$\begin{bmatrix} 1 & -2 & 0 & 1 \\ 0 & 0 & 1 & 5 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

and the system would be inconsistent. On the other hand, if the third equation reads:

$$0x_4 = 0$$

the coefficient matrix would be the same as above, but the system is consistent, albeit the third equation conveys no information and might as well be dropped.

The following are clear from these examples for a *system in reduced row-echelon form*:

- The number of leading variables for a system in reduced row-echelon form can be given by the formula:
  Number of leading variables = (Number of rows that are not identically zero) = (Total number of rows) - (Number of rows that are identically zero)
  And therefore:
  Number of non-leading variables = (Number of columns) - (Number of rows that are not identically zero)
- For the system to be consistent, all rows with zeros for the coefficient matrix should also have zero as the augmented entry. Note that if there is no row of all zeros in the coefficient matrix, the system is automatically consistent. We can think of the rows that are identically zero as redundant and uninformative, and they can be discarded with no loss of information.
- Assuming the system is consistent, the "dimension" of the solution space is the number of freely varying parameters, which is equal to the number of variables that are *not* pivotal. Thus, we have:
  Dimension of solution space = (Number of columns) - (Number of rows that are not identically zero)
  Another way of putting it is that:
  Dimension of solution space = (Number of variables) - (Number of equations that actually give us new information)
- In the particular case that there are no rows that are identically zero in the coefficient matrix, we simply get:
  Dimension of solution space = (Number of variables) - (Number of equations)
  Note that this agrees with our general theoretical understanding.

2.3. **Converting a system into reduced row-echelon form.** Recall our three types of reversible operations on systems of linear equations:

(1) Multiply or divide an equation by a nonzero scalar.
(2) Add or subtract a multiple of an equation to another equation.
(3) Swap two equations.

These correspond to similar operations on the augmented matrix:

(1) Multiply or divide a row by a nonzero scalar.
(2) Add or subtract a multiple of a row to another row.
(3) Swap two rows.

Our *end goal* is to convert the system to reduced row-echelon form. Our *permissible moves* are the above three types of reversible operations. We now need to figure out a *general procedure* that, starting with any system, can identify the appropriate sequence of permissible moves to reach our end goal. The choice of moves will obviously depend on the system we start with. We want to make the process as algorithmically straightforward as possible, so that we ultimately do not have to think much about it.

The operations that we perform are done on the augmented matrix, but they are *controlled* by the coefficient matrix, in the sense that our decision of what operation to do at each stage is completely governned by the coefficient matrix. We do not ever actually need to look at the right-most column in the decision process for the operation.

Here is the procedure.

- We scan the first column, then the second column and so on till we find a column with a nonzero entry. As soon as we have discovered such a column, look for the top-most nonzero entry in that column. If that is not in the first row, then *swap* the first row with the row where it occurs. Note that we perform the swap in the augmented matrix, so we also move the entries of the right-most column. Recall that swapping equations (i.e., swapping rows) is a reversible transformation that preserves the set of solutions.
- We now have a situation of (possibly) some all-zero columns on the left edge, then a column where the top entry (first row) is nonzero. Now, if this top entry is not 1, *divide* that equation by the top entry. In the augmented matrix, this involves dividing the row by a nonzero number.
- What's left now is to clear all the nonzero stuff that may occur *below* the first row in that particular column. We can clear this stuff out by subtracting suitable multiples of the first row from each of the later rows.

Once we have done this clearing out, we ignore the first row, and look for the earliest column that has a nonzero entry in some row other than the first row. Note that that earliest column must be strictly later than the pivotal column for the first row. Having done so, swap it with the second row, then divide to make the pivotal entry 1. Now, clear all the nonzero entries in its colum below and above it by subtracting a suitable multiple of the second row from each of the other rows. Keep repeating a similar process. Explicitly:

- If the first i rows have been dealt with, and the pivotal column for the i^{th} row is the j^{th} column, we look for the first column with nonzero stuff below the i^{th} row (this column must have a number bigger than j). We swap the first row where there is a nonzero entry in that column (among those below the i^{th} row) with the $(i+1)^{th}$ row).
- We now divide out by the leading entry of that row to make it 1.
- Finally, we clear out all entries in the column of that row to 0s by subtracting a suitable multiple of the row.

This process, completed to the end, gives a reduced row-echelon form. If at any stage you get to the situation where a particular row of the coefficient matrix is all zeros, all later rows are all zeros. Reduced row-echelon form has been reached. If any of the corresponding augmented matrix column values is nonzero, then the system is inconsistent. If they are all zero, then the system is consistent, and these last few equations can be discarded.

Note that the column-scanning happens completely in the coefficient matrix, and the judgment of whether we have reached reduced row-echelon form is based completely on the coefficient matrix. However, the operations are also done to the last column of the augmented matrix, which is not part of the coefficient matrix. Note that the last column does not *control* any of the operations and does not affect how the operations are decided. In fact, we could determine the entire sequence of operations, and the final reduced row-echelon form of the coefficient matrix, even without knowledge of the last column. Then, when we get access to the last column (the outputs) we can quickly apply the sequence of operations to this to get the new system, that we then proceed to solve.

We will review in class the example from the book.

2.4. **Cleaning your room.** A full proof that this process works "as advertised" would be too tedious to write down, though it should be conceptually clear. We need to check that the strategy we have outlined above is a *fully general strategy* that is guaranteed to give us a reduced row-echelon form regardless of the system that we start with.

Imagine that you are cleaning your room. You clean one corner of the room. There is a lot of junk there, but you mostly just dump all the junk in another corner which you haven't yet gotten around to cleaning. This doesn't really make the rest of the room dirtier, because the rest of the room was pretty dirty to begin with.

Now, you gradually proceed to more corners of the room. At each stage, as you are cleaning that corner, you can afford to do anything to the parts of the room you have not yet gotten around to cleaning. But you should not dump any stuff in the part you have already cleaned. If you do so, then you will end up having to return and re-clean the part of the room you already cleaned. This is something that you want to avoid. If you can demonstrate that each part of the room you clean makes that part clean and preserves the cleanliness of earlier parts, then you are doing well.

The appropriate analogues to parts of the room in this case are columns of the matrix. In each iteration of the process, one or more columns (traversing from left to right) get "cleaned up." In the process of cleaning up these columns, a lot of changes are made to later columns. But we didn't have any guarantee of cleanliness for those columns. They were a mess anyways. The key point is that while cleaning up a column, *earlier columns* (the ones we already cleaned up) should not get sullied. This indeed is something you should check carefully (in order to get an understanding of why the process works; it's not necessary to check this while executing the process). It's easy to see that these earlier columns are unaffected by swapping and by scalar multiplication. They are also unaffected by the shears because the value being added or subtracted is zero for these columns.

2.5. **Gaussian elimination and row-echelon form.** Recall the three conditions that we used to declare a matrix to be in reduced row-echelon form:

(1) For every row, the left-most nonzero entry, if any, is a 1. We will call this the *pivotal* entry for the row and the column where this appears the pivotal column.
(2) The pivotal column for a later (lower) row is a later (more to the right) column, if it exists. If a given row is all zeros, all later rows are also all zeros.
(3) Any column containing a pivotal 1 must have all other entries in it equal to 0. This has two subparts:
    (a) All entries in the column below the pivotal 1 are equal to 0.
    (b) All entries in the column above the pivotal 1 are equal to 0.

We had mentioned a while back that condition 3(b) is less important. If we do not insist on condition 3(b), we will still end up with a "triangular" system from which the solutions can be constructed with relative ease. A matrix that satisfies conditions 1, 2, and 3(a) is said to be in *row-echelon form* (without the qualifier "reduced").

We can solve systems of linear equations by simply targeting to convert them to row-echelon form, rather than reduced row-echelon form. Some authors use the term *Gaussian elimination* for this process (as opposed to Gauss-Jordan elimination for converting to reduced row-echelon form). Gaussian elimination saves somewhat on the number of operations that needs to be done in row reduction, but a little extra effort needs to be spent constructing the solution. The main saving in row reduction happens at the stage where we are clearing the nonzero entries in the same column as a pivotal 1. With Gauss-Jordan elimination (working towards the *reduced* row-echelon form), we need to clear entries both below and above the 1. With Gaussian elimination (working towards the row-echelon form), we only need to clear entries below the 1.

3. Computational considerations

Recall from an earlier discussion that we evaluate algorithms using five broad criteria:

- *Time* is a major constraint. There's a lot more to life than solving equations. The faster, the better.
- *Memory* is another constraint. Manipulating large systems of equations can take a lot of space. Lots of memory usage can also affect time indirectly. Ideally, our equation-solving process should use as little "working memory" as possible.

- *Parallelizability* is great. Parallelizability means that if multiple processors are available, the task can be split across them in a meaningful manner that cuts down on the time required. Some kinds of procedures are parallelizable, or partially so. Others are not parallelizable.
- *Numerical precision* is another issue, particularly in cases where the numbers involved are not rational numbers. The degree of precision with which we measure solutions is important. Lots of major errors occur because of inappropriate truncation. Hypersensitivity to small errors is a bad idea.
- *Elegance and code simplicity*: Another criterion that is arguably important is the complexity of the code that powers the algorithm. Given two algorithms that take about the same amount of time to run, the algorithm that is based on easier, simpler code is preferable, since the code is easier to maintain, tweak, and debug. Elegance is also valuable for aesthetic reasons, which you may or may not care about.

We will now discuss Gaussian elimination and Gauss-Jordan elimination from the perspective of each of these.

3.1. **Time.** If you've tried doing a few problems using Gauss-Jordan elimination, time feels like a major constraint. Indeed, the process feels deliberately long and tedious. You might wonder whether there exist other *ad hoc* approaches, such as substituting from one equation into the other, that work better.

In fact, there is no known method that works better than Gaussian elimination in general (though it can be tweaked at the margins, and there are special matrix structures where shorter methods apply). The "substitute from one equation into another" approach ends up being computationally the same as Gaussian elimination. The main reason why Gaussian elimination is preferred is that it is programmatically easier to code row operations rather than actual manipulation rules for equations (as substitution rules would do).

We first try to compute the *number of row operations* involved in Gaussian elimination and Gauss-Jordan elimination. For simplicity, we will assume that we are working with an equal number of variables and equations n. We note that:

- There are n rows that we need to fix.
- For each row, the process for fixing that row involves a scanning (finding nonzero entries), an interchange operation (that is one row operation), a scalar multiplication (that is another row operation) and up to $n-1$ row operations that involve clearing out the nonzero entries in the pivotal column. This suggests an upper bound of $n+1$ row operations. In fact, the upper bound is n, because if there is an interchange operation, then one of the entries is already zero, so we do not need to clear that out.

Overall, then, we could require n^2 row operations. It is easy to construct an example matrix for every n where this upper bound is achieved.

Note that if we are doing Gaussian elimination rather than Gauss-Jordan elimination, the worst case for the number of row operations would be somewhat lower: $n + (n-1) + \cdots + 1 = n(n+1)/2$. This is about half the original number. However, the saving that we achieve does not change the number of row operations in the "big O notation" terms. The worst-case number of row operations remains $\Theta(n^2)$ in that notation.

Let us now proceed to the *arithmetic complexity* of Gauss-Jordan elimination. Arithmetic complexity is time complexity measured in terms of the total number of arithmetic operations involved, where an arithmetic operation includes addition, subtraction, multiplication, or division of two numbers at a time. Each of the three types of row operations involves at most $2n$ arithmetic operations, so since the row operation complexity is $\Theta(n^2)$, this gives an upper bound of $O(n^3)$ on the arithmetic complexity. It is also relatively easy to see that this is the best we can achieve order-wise, although the constant can be somewhat improved once we look at the actual details of the operations (namely, because the nature of our past operations guarantees some entries to already be zero, we can spare ourselves from doing those operations). Thus, the arithmetic complexity is $\Theta(n^3)$. Note that the complexity incurred from other bookkeeping and tracking operations does not affect the complexity estimate.

Similar complexity estimates are valid for systems where the number of rows differs from the number of columns.

Arithmetic complexity is, however, potentially misleading because, it is not true that "an addition is an addition." Addition of large, unwieldy integers is more time-consuming than addition of small integers.

Integer multiplication similarly becomes more complicated the larger the integers are. If dealing with non-integers, we have to worry about precision: the more decimal places we have, the more time-consuming the operations are (this also affects the precision/accuracy evaluation, which we will turn to later).

One of the problems is that even if the original matrix we start with has small integer entries, row reduction can lead to larger integer entries in magnitude, and repeatedly doing so can get us out of control. Consider, for instance, the matrix:

$$\begin{bmatrix} 1 & 3 \\ 3 & -1 \end{bmatrix}$$

The first row is already good, so we clear out the 3 in the first column that occurs below the 1. To do this, we subtract 3 times the first row from the second row. We get:

$$\begin{bmatrix} 1 & 3 \\ 0 & -10 \end{bmatrix}$$

Note that we ended up with a large magnitude integer -10, much bigger than the entries in the matrix we started with. The problem could keep getting compounded in bigger matrices. Further, note that converting leading entries to 1 by division also brings up the possibility of having to deal with rational numbers that are not integers, even if we started completely with integers.

There is a variation of Gauss-Jordan elimination called the *Bareiss algorithm* that does a good job of controlling the size of entries at all intermediate steps of the algorithm. The algorithm bounds the bit complexity of the algorithm to a polynomial in n and the bit length of the entries. Discussion of the algorithm in full is beyond the scope of this course. Nonetheless, we will explore some ideas related to this algorithm to help keep the coefficient matrix as close to integers as possible and make your life easier, even though we'll avoid the algorithm in full (see Section 4.4 for a few details).

3.2. **Space requirements.** As was the case with time, space requirements can be interpreted in two ways: the *arithmetic space requirement* and the *bit space requirement*. Arithmetically, at every stage, we need to store a matrix of the same size as the original augmented matrix, plus a much smaller amount of tracking information that helps us remember how far in the algorithm we have gotten. In other words, the space requirement for the algorithm for a situation with n variables and n equations is $\Theta(n^2)$.

Note that this assumes that space is *reusable*. If space is not reusable, then the space requirement would be more complicated.

The computation also becomes more complicated if we consider the issue of the numbers in the matrix becoming larger in magnitude or getting to involve fractions. The *bit complexity* estimate will therefore be somewhat higher, though it will still be polynomial in n and L, where L is an upper bound on the bit length of all the augmented matrix entries.

3.3. **Parallelizability.** The algorithm is partly parallelizable. Specifically, the process of clearing the nonzero entries in a column with a pivotal entry can be done in parallel: each row can be done separately. Moreover, within each row, the different entry subtractions can be done in parallel. In other words, we could be doing all the $O(n^2)$ operations corresponding to clearing out a pivotal column more or less simultaneously.

This parallelization is most useful in situations where read/write access to an always-updated augmented matrix is a lot cheaper than actually performing arithmetic operations on the entries. In other words, if the cost of having many different processors have read/write access to the matrix is much less than the time cost of doing an arithmetic operation, then this parallelization makes sense. With such parallelization, the total arithmetic complexity could become as low as $O(n)$. However, these assumptions are not typically realistic.

There are other ways of parallelizing which include the use of block matrices, but these involve modifications to the algorithm and are beyond the present scope.

3.4. **Precision and accuracy.** If we are working with rational entries we can maintain perfect precision and accuracy. This is because all entries in a process starting with rational entries will remain rational. If we wish, we could first convert the entries to integers and then use the Bareiss algorithm to avoid entries that become too complicated.

Precision becomes an issue when the entries are not rationals, but observed real numbers without closed-form expressions, and we are truncating them to a manageable level. The problem arises most for matrices where the leading entry in a row is very small in magnitude, so dividing out by it means multiplying by a huge number, thus amplifying any measurement error present in that and remaining entries. There are certain kinds of matrices for which this problem does not arise, and there are modifications of Gauss-Jordan elimination that (a) keep the problem in check, and (b) failing that, warn us in case it cannot be kept in check.

3.5. **Elegance and code simplicity.** Gauss-Jordan elimination is conceptually not too hard, albeit it is still somewhat tricky to code. The main advantage of coding Gauss-Jordan elimination as opposed to "substitute and solve" approaches, even though they are computationally equivalent, is that the latter approaches require coding symbolic manipulations of equations, which is trickier. Gauss-Jordan elimination, on the other hand, converts those symbolic manipulations of equations into addition and subtraction of numbers, and the algorithm does not require the computer to "understand" how to manipulate an equation.

3.6. **Amortization when multiple systems share the same coefficient matrix.** We discussed earlier that when the same coefficient matrix appears in multiple systems, with the final column (the outputs) of the systems not known in advance, it makes sense to "preprocess" the system by converting the coefficient matrix to reduced row-echelon form and recording all the row operation steps performed. Then, when given access to the output column, we can simply apply the operations step-by-step to that output column alone (the coefficient matrix having already been dealt with) and read off the solution.

The arithmetic complexity of the operations done on the output column alone is the number of row operations done in Gauss-Jordan elimination, which is $\Theta(n^2)$ assuming a worst-case scenario. Thus, the fixed pre-processing cost is $\Theta(n^3)$ but the marginal time cost for each new output is $\Theta(n^2)$.

4. Human computer tips

You're a human computer. Yes, you! `https://en.wikipedia.org/wiki/Human_computer`

In other words, you need to figure out how to do computations effectively and correctly.

The peril of being a human computer is that you are more liable to make careless errors. The advantage of being a human computer is that you can often rely on visual heuristics that mechanical computers cannot really understand. Let's use our advantages to help guard against the perils.

That said, to a large extent, the tips for smart computation and clear solution-writing (including how much detail to show) are closely linked with the tips for programming an appropriate algorithm for a computer to execute. Some of the ideas we discussed in the previous section will reappear in a somewhat different form here.

4.1. **Writing the operation clearly.** For every operation or sequence of operations that you do, write the operation or sequence clearly. In case of a sequence, write the operations in sequence. You may write the operation above a transition arrow, or on the right of the matrix where you are adding and subtracting. We'll see soon how much we can combine operations, and your approach to writing operations should take that into account.

One advantage of clearly writing the row operation steps is in terms of *pre-processing*: if we ever need to solve a linear system with the same coefficient matrix but a different augmenting column, we can easily see all the row operation steps and simply apply them to the new augmenting column, getting to the answer much more quickly than if we had not written down the steps.

4.2. **Should the steps be combined? Yes, as long as they are parallelizable.** One can consider two extremes in terms of how extensively you write your steps. At one extreme, you rewrite the entire augmented matrix *after every row operation*, stating the row operation used at the transformation at each stage. The problem with this is that you have to write a lot of stuff. Since there are $\Theta(n^2)$ row operations, you end up writing $\Theta(n^2)$ matrices. In other words, a lot of space gets used up.

At the other extreme, you just write one matrix with an erasable pencil, and keep erasing and updating entries each time you do a row operation. The problem with this is that it can make you feel giddy.

Is there a middle ground? Yes. The idea is to essentially follow the first approach, *except that we do not separately write steps for operations that are done in parallel*. Explicitly, this means that we can combine all

the clearing steps for a single pivotal column together. Let us say that at some stage, our augmented matrix reads as:

$$\left[\begin{array}{ccc|c} 1 & 2 & 3 & 4 \\ 2 & 5 & 1 & 1 \\ 5 & 7 & 8 & -2 \end{array}\right]$$

We need to clear out the entries in the first column below the pivotal 1. There are two operations we need to perform: $R2 \to R2 - 2R1$ (i.e., we sutract twice the first row from the second row), and $R3 \to R3 - 5R1$ (i.e., we subtract 5 times the first row from the third row). We can directly write down the augmented matrix obtained after *both* these operations are performed, instead of writing down the full new augmented matrix after one operation and then again the one obtained after the second. Note that you should also write down clearly *all the row operations that are being done simultaneously*, so in this case, you should write down $R2 \to R2 - 2R1$ and $R3 \to R3 - 5R1$ (some people write these expressions on the immediate right of the matrix, some write then above the transtion arrow).

The reason why parallel steps can be combined is that doing so does not involve any erasing and rewriting, since the steps do not influence one another. On the other hand, a sequence such that $R2 \to R2 - 2R1$ followed by $R1 \to R1 - R2$ should be done step-by-step because the second step is dependent on the outcome of the first one. Note that with this approach, you only end up writing $\Theta(n)$ matrices, and about $\Theta(n^3)$ numbers, which makes sense because $\Theta(n^3)$ is the arithmetic complexity of the procedure.

In conceptual terms, parallelizable operations can be done together because they commute with each other. It does not matter in what order we do the operations. For instance, in the above example, the operations $R2 \to R2 - 2R1$ and $R3 \to R3 - 3R1$ yield the same final result regardless of the order in which they are performed.

4.3. **Noting excellent rows to swap to the top.** A literal reading of the Gauss-Jordan algorithm would suggest that we look for the *earliest* row with a nonzero entry in the relevant column. However, it typically makes more sense to look at all the rows and pick the one that's most likely to minimize further computation. A good row is one where the leading entry is as close to 1 as possible already, and as many of the other entries are 0 (and if not 0, at least small) as possible. If you find that type of row, swap that to the top.

For instance, consider the augmented matrix:

$$\left[\begin{array}{ccc|c} 4 & 7 & 5 & 12 \\ 5 & 4 & 3 & 10 \\ 1 & 0 & 2 & 3 \end{array}\right]$$

One can start blindly here: divide the first row by 4, and proceed to clear the remaining entries in the first column. Alternatively, one can proceed smartly: interchange the first and third row. Note how much easier the remaining row clearing operations become if we use the latter approach.

Of course, sometimes, all the good stuff doesn't come together, and in that case, one has to prioritize (for instance, one row might be mostly zeros but its leading entry may be large, another row might lead with a 1 but have messy later entries). The goal here isn't to come up with a foolproof way to avoid further work. Rather, the goal is to identify a few methods that might on occasion save some precious time.

4.4. **Keeping the entries as close to integers as possible.** Yes, the official version of Gauss-Jordan elimination tells you to divide and make the pivotal entry 1. In practice, this can cause a headache. It's generally advisable to use other techniques. Did you know that, in practice, one can dispense of division almost entirely in the quest to get a pivotal 1? That does entail doing more subtraction steps. But it's often worth it, if it means we can keep everything integral. This is the secret of the Bareiss algorithm, but we'll just catch a glimpse here.

For instance, consider the system:

$$\begin{aligned} 3x + y &= 4 \\ 5x + 7y &= 44 \end{aligned}$$

$$\left[\begin{array}{cc|c} 3 & 1 & 4 \\ 5 & 7 & 44 \end{array}\right]$$

The standard Gauss-Jordan method would involve dividing the first row by 3. Let's try to do it another way.

We first do $R2 \to R2 - R1$, and obtain:

$$\left[\begin{array}{cc|c} 3 & 1 & 4 \\ 2 & 6 & 40 \end{array}\right]$$

We now do $R1 \to R1 - R2$, and obtain:

$$\left[\begin{array}{cc|c} 1 & -5 & -36 \\ 2 & 6 & 40 \end{array}\right]$$

Note that we have managed to make the column entry 1 without doing any division. We now do $R2 \to R2 - 2R1$ and obtain:

$$\left[\begin{array}{cc|c} 1 & -5 & -36 \\ 0 & 16 & 112 \end{array}\right]$$

Now this R2 can be divided by 16 (we have no real choice) but luckily everything remains integral, and we obtain:

$$\left[\begin{array}{cc|c} 1 & -5 & -36 \\ 0 & 1 & 7 \end{array}\right]$$

We finally do $R1 \to R1 + 5R2$ and obtain:

$$\left[\begin{array}{cc|c} 1 & 0 & -1 \\ 0 & 1 & 7 \end{array}\right]$$

Thus, $x = -1$ and $y = 7$.

The puzzling part of the process is the first two subtractions. How did we think of them? The secret sauce is something called the *gcd algorithm* (this is the algorithm used to compute the gcd of two numbers): keep applying the Euclidean algorithm to find quotients and remainders. If you wish to learn how that works, feel free to read it up. Otherwise, just try to get an intuitive sense of how to do it in easy cases.

4.5. **Keeping a number sense.** We mentioned above that dealing with numbers makes things a lot easier to code than handling full-scale algebraic manipulations. Note, however, that what's simple for computers need not be simple for humans, particularly humans who understand algebra. I've often observed that people can sometimes make fewer careless errors when solving systems of linear equations using symbolic manipulation of the equation (i.e., writing down the full equations and adding, subtracting, and substituting) rather than performing row operations on the augmented matrix, even though the row operations on the matrix are tantamount to equation manipulation. This may have something to do with the degree of natural structural understanding and care that people exhibit: when manipulating equations, you have more of a sense of whether what you're doing is right or wrong. When adding and subtracting numbers, the process feels more mechanical and you may therefore be less attentive. Ideally, you'd want the best of both worlds: just do row operations on the matrices, but still have a "number sense" that alerts you to when things are going wrong.

Perhaps the best way to achieve this is to periodically "make sense" of the numerical operations that we performend in terms of the symbolic algebra. In addition, the tips for good presentation and avoiding careless errors given above should help.