Computer Solution Of Simultaneous Equations: A Byproduct Of The First Programming Course

Prof. W. D. Maurer
Department of Computer Science
The George Washington University
Washington, DC 20052
Phone: (202)994-5921
Email: maurer@seas.gwu.edu

ABSTRACT

In our first programming course, which teaches C with a small amount of C++, we have a semester-long case study on the computer solution of simultaneous equations, which it is our purpose here to describe. Real and integer-fractional coefficients, error cases, status codes, and the contrast between the determinant approach (using recursion) and the elimination approach (using pivoting when the coefficients are real numbers) are considered in detail. This case study is being made at the request of our curriculum committee, which has observed that computer solution of simultaneous equations should be required for all engineering students, but is not taught anywhere else in a course that would be required for them.

1. INTRODUCTION

At our university, the Computer Science Department is in the engineering school. Our numerical methods course is not required for all engineers; yet all engineers deal with matrices and need to know the special requirements of computer solution of simultaneous equations. A programming exercise on this topic has, therefore, become mandatory in our first programming course in C, as we found out upon assuming faculty oversight responsibility for that course.

What started as a routine effort in support of a necessary requirement at our university has become much more than that. Essentially, we are now advocating such a requirement in any presentation of C, with some C++, as the first programming language. This is partly because all scientists and engineers learn how to solve simultaneous equations by hand, but solving them by computer is very easy to get wrong, and indeed was gotten wrong, often and publicly, in the early days of computing.

Also, solving simultaneous equations by computer involves many features of C and elementary C++: two-dimensional arrays (for coefficients); enumerated types (for reporting the number of

solutions); a fraction class (for exact solution of equations with integer coefficients); and even, as we shall show, a challenge to conventional wisdom regarding the go-to statement.

2. STATUS CODES

Any simultaneous equation solver must be concerned with the error case in which the matrix is singular. A thorough approach to the subject will distinguish two situations with singular matrices. As an example, the equations

2x+3y = 18
2x+3y = 19

have no solutions, because 2x+3y cannot be equal to both 18 and 19 at the same time; while the equations

2x+3y = 11
6x+9y = 33

have more than one solution. These look like two different equations, but they are really not, because the second equation is the first one multiplied through by 3. Any values for `x` and `y` which satisfy the first equation will automatically satisfy the second one — such as `x` = 4 and `y` = 1, or `x` = 1 and `y` = 3, or, more generally, `y` = (11–2`x`)/3 for any value of `x`.

This is a simple example of a situation which arises very often in programming, namely a single normal case and more than one error case. Conventionally, this is handled by means of a status code, which is zero in the normal case and has other values, starting from 1, in the various error cases. In C and C++, the usual way to declare status codes involves an enumeration, using the keyword `enum`. This then becomes our way, in the first course, of teaching `enum` and the issues which arise from it.

3. ONE EQUATION IN ONE UNKNOWN

Perhaps surprisingly, we start with one equation in one unknown, $ax = b$. Even in this case, there are three possible outcomes: one solution ($a \neq 0$); no solutions ($a = 0$ and $b \neq 0$); or many solutions ($a = b = 0$). We can therefore use this case as a way of introducing enumeration issues in a simple context.

Enumerations have to be motivated. Why not just use the codes 0, 1, and 2 directly for one solution, no solutions, and many solutions? In order to see why not, we start our presentation by

doing just that; setting a status variable equal to 0, 1, or 2, and then using that status variable later.

3.1 Global Status Variables

There are three possible approaches to status variables. One is to make them global, which is clearly unadvisable because of name conflicts. We do not even bring up this possibility until much later, when we are looking at separate compilation issues.

3.2 Status Variables As Parameters

The second approach to a status variable is to make it into a parameter of the simultaneous equation solver. However, this requires a parameter which can be changed, and this cannot be done in straightforward C.

Our approach to this was set forth in an earlier paper (Maurer, 2000). In the first course, we are concerned, first of all, with teaching all of elementary C. Too many introductory C++ courses, in our opinion, spend so much time teaching classes and object-oriented concepts that large parts of elementary C are never learned at all. Nevertheless, there are a few C++ concepts which ought to be taught from the beginning, because they make life easier for the programmer. One of these is reference parameters. You can fake up a reference parameter in straight C, using the unary `&` and `*` operators, but this is awkward; it is not necessary in any other language; and, most important to us in an engineering school, it requires knowledge of pointers, which is inappropriate for those who are concerned merely with scientific and engineering formulas and not with the internals of a computer.

In fact, we use the status variable situation as our way of teaching reference parameters. At this point, our status variable still has values 0, 1, and 2, and it therefore becomes a parameter that is declared by `int& status;` instead of simply by `int status;` .

3.3 Returning A Status Variable

The third approach to a status variable, which is the one most commonly found in practice, is to return it as the value of the simultaneous equation solver. If this is done, then there is still a reference parameter when we are solving $ax = b$, because now a, x, and b are all parameters, and x is changed by the function. Therefore x becomes a parameter that is declared by `double& x;` instead of simply by `double x;` .

Returning a status variable causes a subtle problem in understandability. The usual status variable convention (0 = no error; nonzero = error) allows us to write code like

`if (solve1(i,j,n) != 0)` { perform some error action }

where `solve1` is the simultaneous equation solver. Students have to be taught that such an expression always actually calls `solve1(i,j,n)`; in other words, that when a function is part of an condition, *that function is always actually called*. This might seem strange to a beginner; for example, consider `f(k)`, defined as

```
int f(int k)
  {cout << "f(" << k << ")" << endl; return k; }
```

The value of `f(k)` is always the same as the value of `k`. However, this does not mean that `if (f(k) != 0)` means the same as `if (k != 0)`. If it were, then `f(k)` would not have to be called, when `if (f(k) != 0)` is done. Students have to know, explicitly, that here `f(k)` is always called, *and the output is always produced*.

3.4 Enumerations

We take up enumerations only now, after introducing all the above issues, which are easier to understand if we use codes for status variables (0, 1, 2, and so on). In a practical situation, we don't use codes because they are easy to get wrong, and because, if we get them wrong, we have a bug which it is unusually difficult to find. In setting up an enumeration for simultaneous equations, we might define

```
enum sim_eqn_stat {one_sol, no_sol, many_sol} status;
```

The basic concepts to be introduced here are the enumeration (in this case, `sim_eqn_stat`) and the enumerators, or enumeration constants (in this case, `one_sol`, `no_sol`, and `many_sol`). The strong similarity between enuumeration declarations and enumeration definitions must be emphasized here; thus, in addition to `enum name {defn} vars;` we also have the declaration form `enum name {defn};` and the definition form `enum name vars;`. It must also be emphasized that C and C++ give no protection (as Pascal and Ada do) against misuse of enumerations. In C and C++, it is merely custom, and not the enforced rules of the language, which dictates that, once enumeration constants have been defined, you always, in writing your programs, use these constants rather than their associated integer codes.

3.5 Using A Status Code

In our course, we introduce the `switch` statement in C and C++ by using a character variable as the switch expression; but we immediately go on to a treatment of the very common use of

enumerations as switch expressions. Thus our `sim_eqn_stat` variable above, called `status`, could be used as follows (and we use this example):

```
switch (solve1(a1, b1, x1)) {
  case one_sol:
    cout << "One solution: " << x1 << endl; break;
  case no_sol: cout << "No solutions" << endl; break;
  case many_sol:
    cout << "Many solutions" << endl; break;
  default: error("Error in sim_eqn_stat");
}
```

4. FRACTION CLASSES

The main difficulty in working with simultaneous equations has to do with control of floating point error. This, however, is unnecessarily hard to understand for those who do not yet sufficiently understand the basic algorithm. It is easier to do this if we start with integer coefficients, because here all calculations are exact. However, simultaneous equations with integer coefficients can easily have solutions which are fractional. How do we deal with these?

In straight C, this would be difficult, much more so than working with real numbers. In C++, it is easy, if we use a `fraction` class. However, how do we justify introducing this C++ concept in a course which is basically on C? We do it by noting that, even in a course on C, we have to introduce a few class concepts, because the alternative would be to teach `struct`, which is very seldom used any more except for bit fields.

Our choice has been to set aside one week for an introduction to classes, using a fraction class as an example. Note that, with a fraction class, we can now treat simultaneous equations with fractional, as well as integer, coefficients. All this material, of course, will be covered in much greater detail in the second course (on C++ and data structures). In that one week, we cover: class and object declarations; design of a fraction class; constructors; lowest terms; overloaded operators; and safe constructors.

5. MEMBER FUNCTIONS

Member functions are mentioned only in passing, in this elementary course, and then only where they illustrate points that are necessary for our treatment of simultaneous equations or for input, such as `cin.get` and `cin.getline`. Briefly stated are:

- The two ways of defining a member function (inside and outside its class);
- The rules for overloaded operators as member functions;
- Declaration of private data;
- Friend declarations;

• The rule that private data may be accessed only by members and friends.

The `lowest_terms` function, introduced before, is now reintroduced as a member function, which is private because it is intended for use only by constructors, which are member functions, and by overloaded operators expressed as member functions.

6. MIXING FRACTIONS WITH INTEGERS

In working with simultaneous equations, we will need to be able to combine fractions and integers. This is analogous to the usual ways in which we combine integers and real numbers in C. Given an integer and a real number, we can add them; the integer is converted to real and the two real numbers are added. In the same way, we can have an `int` argument `k` to a function in which the corresponding formal parameter is of type `double`, which is then set to `k` as converted from an `int` to a `double`.

In the same way, given an `int` and a `fraction`, we can add them; the `int` is converted to a fraction and the two fractions are added. Similarly, in a function, we can have a formal `fraction` parameter and a corresponding `int` argument, which is then converted to a `fraction`. Neither of these conversions, however, are automatic; they depend on a special property of one of our constructors. The rule here is that when a constructor has exactly one argument, it also serves as a coercion. We have a constructor with one argument for fractions, which produces a fraction equal to its argument. This is then used in converting int to fraction in both of the above cases.

A `fraction` plus an `int` could also be calculated by an overloaded operator having `fraction` and `int` as parameters. This would be more efficient, because the calculation formulas are simpler, and also because we do not have to call `lowest_terms` (it can be shown that the result is already in lowest terms, in this case).

7. TWO-DIMENSIONAL ARRAYS

Many treatments of elementary C (or Pascal or Fortran, for that matter) bring up arrays only near the end of the course. We have never done that, because arrays, to us, are of fundamental importance and come up in a wide variety of programming situations. Two-dimensional arrays, however, are another matter. They are subject to a number of errors (as when their indices start from 1, or when their names are used as parameters), and a good understanding of pointers, which comes earlier in the course, is necessary to see how to circumvent these errors. Of course, two-dimensional arrays are necessary for simultaneous equations, and we now present an entire week devoted to these two topics.

We motivate two-dimensional arrays by means of spreadsheets. The declaration of such an array, and the use of its elements, are straightforward. Further issues introduced here include searching a two-dimensional array; further examples of multidimensional arrays; array access formulas; arrays of pointers to arrays; two-dimensional array names as parameters; general lower bounds (other than 0); and initializing a two-dimensional array.

8. MATRICES AND SIMULTANEOUS EQUATIONS

Now, finally, the students have all the basic material they need in order to understand how to solve simultaneous equations by computer.

8.1 Specifications Of A General Simultaneous Equation Solver

There are three parameters to a function which solves *n* simultaneous equations:
• an *n*-dimensional vector for the right sides of the equations;
• an *n*-dimensional vector for the variables being solved for; and
• an *n*-by-*n* matrix of the coefficients. By what we have just learned (see section 7 above), *n* may not vary, from one call of the function to the next.

8.2 Two Equations In Two Unknowns

In order to have a better understanding of the *n*-dimensional case, we now take up the two-dimensional case in more detail. In this case, we do not use a matrix, but instead we deal with eight quantities separately; that is, *a*, *b*, *c*, *d*, *e*, *f*, *x*, and *y* in the equations

```
ax+by = c
dx+ey = f
```

Our function is declared as follows:

```
sim_eqn_stat solve2(double a, double b, double c,
   double d, double e, double f, double &x, double &y);
```

If there is one solution, it is easily obtained. We can multiply the first equation above through by *e*, and the second one by *b*, then subtracting the second one from the first, obtaining an equation for *x* which does not involve *y*. We can also multiply the first equation through by *d*, and the second one by *a*; this time, when we subtract, we get an equation for *y* which does not involve *x*. This is all done as follows:

```
aex+bey = ceadx+bdy = cd
bdx+bey = bfadx+aey = af
aex–bdx = ce–bfbdy–aey = cd–af
```

```
or(ae–bd)x = ce–bf(ae–bd)y = af–cd
```

where we have multiplied the last equation through by –1, so that the factor, *ae–bd*, comes out the same in both equations. If this factor is not zero, our solutions are

```
x = (ce-bf)/(ae-bd)y = (af-cd)/(ae-bd)
```

If the factor is zero, we now ask whether *ce-bf* and *af-cd* are zero. If either of these is not zero, then the equations have no solutions, because either *x* or *y* (or both) is expressed as a non-zero quantity, divided by zero. If *ce-bf* and *af-cd* are both zero, however, the situation is not as simple as in the case of one equation in one unknown. Our equations might have many solutions, as was the case then; but they also might have no solution, as we will now see.

8.3 Dependent Equations

We first consider the case in which both *ce-bf* and *af-cd* (in addition to *ae-bd*) are zero. This implies that $ae = bd$, $ce = bf$, and $af = cd$. Usually, in this case, we have dependent equations. In general, one equation depends on another if the first equation is true only when the second one is true. Our two equations here are $ax+by = c$ and $dx+ey = f$, and the second of these is dependent on the first only if it is of the form $g(ax+by) = gc$, or $gax+gby = gc$, for some g. Note that in this case $d = ga$, $e = gb$, and $f = gc$, so that indeed $ae = agb = bd$; $ce = cgb = bf$; and $af = agc = cd$.

Suppose first that $a \neq 0$. Then d/a is defined, and $d = ga$ where $g = d/a$. Also, e = gb because $ae = bd$, and dividing through by a ($\neq 0$) gives $e = bd/a = (d/a)b = gb$. Similarly, $f = gc$ because $af = cd$, and dividing through by a gives $f = cd/a = (d/a)c = gc$. Our second equation, $dx+ey = f$, is now $gax+gby = gc$, as above, so the two equations are dependent.

If $a = 0$, and g, as above, exists at all, then $d = ga = 0$. However, in general, if $d \neq 0$, then a/d is defined, and $a = hd$ where $h = a/d$. In this case it is the first of our equations that is dependent on the second, rather than the other way around. That is, we now have $a = hd$; $b = he$ (because $bd = ae$ and thus $b = ae/d = (a/d)e = he$ after dividing through by $d \neq 0$); and $c = hf$ (because $cd = af$ and thus $c = af/d = (a/d)f = hf$).

There remains the case in which $a = d = 0$; and here the two equations can still be dependent. Of course, in this case, $d = ga$ and $a = hd$ for any g and h whatsoever. Suppose first that $b \neq 0$. Then e/b is defined, and $e = gb$ where $g = e/b$. Also, $f = gc$ because $ce = bf$, and dividing through by b ($\neq 0$) gives $f = ce/b = (e/b)c = gc$. On the other hand, if $e \neq 0$, then b/e is defined, and $b = he$ where $h = b/e$. In this case it is the first equation that is dependent on the second, as before. Note that $ce = bf$, and dividing through by e ($\neq 0$) gives $c = bf/e = (b/e)f = hf$.

The only case in which the equations are not dependent is that in which $a = d = b = e = 0$. In this case there are no solutions unless $c = f = 0$, in which case every value of x and y is a solution (and again the equations are dependent).

If one of our equations is dependent on the other, it can be removed from consideration, and we can simply solve the other equation. In this case there are usually many solutions. We consider only the equation $ax+by = c$; the other equation is handled similarly. If $b \neq 0$, then $y = (c–ax)/b$. This is a straight line, which is horizontal if $a = 0$. If $b = 0$ but $a \neq 0$, then $x = a/c$ (and y can be anything); this is a vertical straight line. If $a = b = 0$, then $a = d = b = e = 0$, which is a case we have already treated above.

8.4 The Program for Two Equations

We may now summarize the mathematics above, as follows:
• If ae–bd ≠ 0, there is one and only one solution; and it is x = (ce–bf)/(ae–bd) and y = (*af*–*cd*)/(*ae*–*bd*).
• If *ae*–*bd* = 0, and either *ce*–*bf* ≠ 0 or *af*–*cd* ≠ 0, there are no solutions.
• If $a = b = c = d = e = f = 0$, then every value of x and y is a solution.
• If $a = b = d = e = 0$, but either $c \neq 0$ or $f \neq 0$ (or both), there are no solutions.
• In all other cases, there are many solutions, lying along a straight line.

An initial version of our program is therefore as follows, using the enumerated type sim_eqn_stat as in section 3.4, and with an important error to be explained in section 8.5 below:

```
sim_eqn_stat solve2(double a, double b, double c,
   double d, double e, double f, double &x, double &y) {
    // solves ax+by = c and dx+ey = f for x and y
    // returns a status code
  double u = a*e-b*d;
  double v = c*e-b*f;
  double w = a*f-c*d;
  if (u != 0) {x = v/u; y = w/u; return one_sol; }
  if (v != 0 || w != 0) return no_sol;
  if (a == 0 && b == 0 && d == 0 && e == 0)
    return (c == 0 && f == 0 ? many_sol : no_sol);
  return many_sol;
}
```

Note that, when we are testing whether $a = b = d = e = 0$, we cannot write it that way, in our program; that is, `if (a == b == d == e == 0)`. This is not a syntax error; it will compile, but its meaning is `if ((((a == b) == d) == e) == 0)`. That is, `(a == b)` is either 1 (if $a = b$) or 0 (if $a \neq b$); this 1 or 0 is then compared with d; the result of that (either 1 or 0) is compared with e, and so on.

This time, we are using many_sol for two different cases, one in which the solutions lie along a straight line, and the other in which every solution is possible. If we wanted to, we could expand the type sim_eqn_stat to include a new value, `every_sol`, which could be returned if *a*, *b*, *c*, *d*, *e*, and *f* were all zero.

8.5 Fudge Factors

The real problem with solve2, as given in section 8.4 above, is in the statement which starts with `if (u != 0)` . This does not always work properly, due to floating point error. It might seem that we can ask whether the answer, when we subtract *bd* from *ae*, is very small. However, this depends on how you define "very small." In our example, the real question is whether *ae–bd* is small, compared to *ae* and *bd* themselves. We will test this while we subtract *bd* from *ae*, using a function `fsub` ("floating subtract" or "fuzzy subtract"). The value of `fsub(x,y)` is *x–y* unless *x–y* is "too small" compared to *x* (or to *y*), in which case it is zero. We may define `fsub` like this:

```
inline double fsub(double x, double y) {
    // returns x-y or 0 if x-y is "too close" to 0
  double u = x-y;
  if (abs(u/x) <1.0e-6) u = 0;
  return u;
}
```

making it inline because it is so short. Here `1.0e-6` (that is, 1/1,000,000) is called the fudge factor. If `fsub(x,y)` is calculated as zero, then statements like `if (u != 0)` , above, will work properly, and there is no need to adjust them. Notice the need for `abs` here; it is the absolute value of *u/x* that must be smaller than the fudge factor (since *u/x* might be –1, for example).

8.6 Calculating the Fudge Factor

We can now proceed in either of two directions. One is to modify our program by using `fsub` as defined above, like this:

```
sim_eqn_stat solve2(double a, double b, double c,
   double d, double e, double f, double &x, double &y) {
    // solves ax+by = c and dx+ey = f for x and y
    // returns a status code
  double u = fsub(a*e,b*d);
  double v = fsub(c*e,b*f);
  double w = fsub(a*f,c*d);
  if (u != 0) {x = v/u; y = w/u; return one_sol; }
  if (v != 0 || w != 0) return no_sol;
  if (a == 0 && b == 0 && d == 0 && e == 0)
    return (c == 0 && f == 0 ? many_sol : no_sol);
  return many_sol;
}
```

This works, almost all of the time. One problem with it is that every so often we really mean for *ae–bd* to be very small, compared to *ae*, but still not zero. This is a fundamental problem with all fudge factors, and there is no solution that works in all cases.

8.7 Two Equations With Fractional Coefficients

The other approach to solving our equations is not to use type double at all, but rather type fraction. Since the numerator and denominator of a fraction are integers, there is now no floating point error to contend with. Then we would write

```
sim_eqn_stat solve2(fraction a, fraction b,
    fraction c, fraction d, fraction e,
    fraction f, fraction &x, fraction &y) {
    // solves ax+by = c and dx+ey = f for x and y
    // returns a status code
  fraction u = a*e-b*d;
  fraction v = c*e-b*f;
  fraction w = a*f-c*d;
  if (u != 0) {x = v/u; y = w/u; return one_sol; }
  if (v != 0 || w != 0) return no_sol;
  if (a == 0 && b == 0 && d == 0 && e == 0)
    return (c == 0 && f == 0 ? many_sol : no_sol);
  return many_sol;
}
```

Note that *a*, *b*, *c*, *d*, *e*, and *f* are now allowed to be fractions. When we introduced the `fraction` type, these were supposed to be integers. However, we can still specify them as integers, because they will be converted to fractions when `solve2` is called.

8.8 Three Equations in Three Unknowns

How does the case of two unknowns generalize to the case of n unknowns? This is not obvious, and, to make it clearer, we work out in detail the case of three unknowns (although we do not write the program). If our three equations are

```
ax+by+cz = d
ex+fy+gz = h
ix+jy+kz = l
```

then we show that

```
afk–agj+bgi–bek+cej–cfi
```

must be nonzero, if the equations have one solution.

8.9 Determinants for Two and Three Equations

The quantity *ae–bd* in section 8.2, and the quantity *afk–agj+bgi–bek+cej–cfi* in section 8.8, are known as determinants. The general rule is that every n-by-n matrix has a determinant. When we solve *n* equations in *n* unknowns, we set up a matrix of the coefficients, as we saw in section 8.1 above. If the determinant of this matrix is non-zero, then the equations have one solution. We now introduce the usual method of calculating a determinant in the general case, using minors.

Looking at the formulas of section 8.2 to solve two equations in two unknowns, we see that, not only is the denominator a determinant in each case, but the numerators are also determinants of new matrices, each of which is obtained by substituting the right-hand side of each of our equations for the corresponding term in column 1 (or 2). Much the same thing happens for three equations in three unknowns, as we now show in detail.

8.10 Error Cases for Three Equations

We have seen that the various error cases for two equations are more complex than they are for one equation. In particular, there are now four cases: one solution, no solutions, many solutions, and every solution. However, for two equations it is still not too difficult to keep track of all possible cases. Let us now see what happens when we have more than two equations.

The first two cases are always the same. If the determinant of the matrix of coefficients is non-zero, then there is always a unique solution. If this determinant is zero, then we look at the numerators in the solution formula, which are also determinants. If any one of these is non-zero, we have the second case, in which there are no solutions. If the denominator and all the numerators are zero, in the formulas, you can always have either no solutions or many solutions, but not a unique solution, in every case except n = 1 (where you must have “every solution”).

When do you have no solutions, in this case, and when do you have many solutions? This is harder to keep track of, in three dimensions. The first problem is that there are now three forms for the solutions, rather than two. The solutions can lie along a line, in three-dimensional space; or along a plane; or they can be the entire three-dimensional space (if all the aij and the bj are zero). More importantly, there are many more ways to draw a straight line in three-dimensional space than in the plane; and the same is true of planes in three-dimensional space.

All these error cases become even harder to keep track of, when there are *n* equations. Because of this, the functions which we write below will make no attempt to distinguish among the error cases. They will use a simplified enumeration type of the form

```
enum sim_status {one_solution, singular);
```

The word *singular* is jargon for the case in which a determinant is zero.

8.11 Using Determinants to Solve the Equations

We will now look at two ways of solving n simultaneous equations in n unknowns. The first will involve direct calculation of determinants. In this paper we omit the details, but merely mention that minors are calculated by recursion, and that this becomes another example of how recursion is used in programming.

8.12 The Elimination Method for Fractions

Whenever we have a recursive method of solving a problem, we try to look for another method, which is not recursive, because explicit recursion is slow. We now introduce a method of solving simultaneous equations which is faster than using determinants. It is, indeed, the method generally used to solve such equations; and we can obtain it directly from the idea of eliminating one of a set of n variables. If we can eliminate the nth variable, then we can eliminate the $(n-1)$st, and so on all the way back to the second variable. This leaves only one variable, whose value we can get. Then we can substitute its value into the second equation, giving us the second variable; substitute the first two variables into the third equation, giving us the third variable, and so on until we are done.

We illustrate the process for fractions first, since here there is no floating point error. Note that in section 8.2 above, where we solved two equations in two unknowns, we multiplied the first equation through by e, and the second one by b. If $e \neq 0$, we can improve this process by multiplying the first equation by b/e, and leaving the second one as it stands. This will be our general approach here; if $a_{nn} \neq 0$, then, for $1 \leq i \leq n-1$, we multiply the ith equation by a_{nn}/a_{in}, obtaining $(a_{nn}/a_{in})a_{in}x_n = a_{nn}x_n$ as the new nth term. Then we subtract the last equation, which also has $a_{nn}x_n$ as its nth term, giving a new equation with the nth term zero. There are $n-1$ of these equations, and they are in the $n-1$ unknowns x_1 through x_{n-1}, because the term involving x_n is eliminated.

What if $a_{nn} = 0$? In that case, we first try to find an earlier equation with $a_{in} \neq 0$. If we cannot find such an equation, then the entire last column of the matrix is zero. In that case the determinant is zero, and we report an error, just as we did before. However, if $a_{in} \neq 0$ for some value of i, then we interchange equation i and equation n. In other words, the old ith equation becomes the new nth equation, and the old nth equation becomes the new ith equation. After this exchange, we proceed as before.

8.13 The Elimination Method for Real Numbers

When we interchange the *n*th equation with some other equation, it does not matter, when we work with fractions, which equation we exchange it with, so long as the *n*th coefficient is non-zero. When we work with real numbers instead of fractions, however, it does matter. Specifically, we need to exchange, with the *n*th equation, the equation having the smallest non-zero coefficient (in absolute value) of the *n*th variable. Furthermore, we need to do this, in the real-number case, whether the *n*th coefficient in the *n*th equation is zero or not.

The reason for this has to do with what we multiply the other equations by, before subtracting. If, after any exchanging of equations, we have a_{nn} a_{in}, then ann/ain 1. Thus, when we multiply an equation through by a_{nn}/a_{in}, as in section 8.12, we are multiplying it through by something with absolute value not greater than 1. This tends to minimize the error in the floating point subtraction that follows, as is taken up in a course on numerical analysis.
Our program to find the equation with which to exchange, and then performing the actual exchange, is now

```
for (i = 1; i < n; i++)
  if (a[i][n] != 0) goto found;
error("determinant is zero");
found:
k = i; absmin = abs(a[i][n]);
for (j = i+1; j < n; j++) {
  newmin = abs(a[j][n]);
  if (newmin < absmin){k = j; absmin = newmin; }
}
if (newmin != absmin) {
  z = b[k]; b[k] = b[n]; b[n] = z;
  for (j = 1; j <= n; j++)
    {z = a[k][j]; a[k][j] = a[n][j]; a[n][j] = z; }
}
```

First we look for an element of the last column, `a[i][n]`, which is not zero. If all of these are zero, we have an error. Otherwise, we proceed from *j* equal to that specific value of *i*, through *n*–1, looking for the smallest absolute value of `a[j][n]` over this range. If that happens to be `a[n][n]`, we are done. Otherwise, we need to exchange equation *n* with equation *k*, just as we exchanged it with equation *i* earlier.

8.14 A Function For The Elimination Method

We now proceed to our function for the elimination method, using real numbers:

```
const int n = 10; // this constant can be changed,
                  // and the program recompiled
typedef double matrix_type[n][n];
sim_status solve_n
    (double *x0, matrix_type a0, double *b0) {
    // solves n simultaneous equations in n unknowns
  matrix_type a = (matrix_type)((double *)a-n-1);
  double *x = x0-1; double *b = b0-1;
```

```
          double z,lower_right; int i,j,k,m;
        for (m = n; m > 1; m--) {
        // this is derived from the program of §11.25
        //  (replacing n by m)
          for (i = 1; i < m; i++)
            if (a[i][m] != 0) goto found;
          return singular;
found:
          k = i; absmin = abs(a[i][m]);
          for (j = i+1; j < m; j++) {
            newmin = abs(a[j][m]);
            if (newmin < absmin){k = j; absmin = newmin; }
          }
          if (newmin != absmin) {
            z = b[k]; b[k] = b[m]; b[m] = z;
            for (j = 1; j <= m; j++)
              {z = a[k][j]; a[k][j] = a[m][j]; a[m][j] = z; }
          }
          // now we multiply equations and subtract
          lower_right = a[m][m];
          for (i = 1; i < m; i++) {
            term = a[i][m];
            if (term != 0) {
              factor = lower_right/term;
              b[i] = fsub(b[i]*factor, b[m]);
              for (j = 1; j <= m; j++)
                a[i][j] = fsub(a[i][j]*factor, a[m][j]);
            }
          }
        }
        z = a[1][1];
        if (z == 0) return singular;
        x[1] = b[1]/z;
        for (i = 2; i <= n; i++) {
          z = b[i];
          for (j = 1; j < i; j++) z = fsub(z, a[i][j]*x[j]);
          x[i] = z/a[i][i];
        }
        return one_solution;
      }
```

First we interchange an equation with the last equation, if this is needed, as in section 8.13. Then we perform the elimination of x_m, from $m = n$ down to $m = 2$. If at any time during this process the last column of the matrix becomes zero, the program stops, and we return `singular`. The same thing happens if a_1 is now zero, when we solve for x_1. Otherwise, we enter the final loop, in which the values of a_2 through a_n are calculated in order and placed in the result vector. Note that equation i, for $2 \quad i \quad n$, is of the form

$$a_{i1}x_1 + a_{i2}x_2 + \ldots + a_{ii}x_i = b_i$$

at this point, since $a_{ij} = 0$ for all $j > i$. Therefore, solving for x_i, we obtain

$$x_i = (b_i - a_{i1}x_1 + a_{i2}x_2 + \ldots + a_{i-1,i-1}x_{i-1})/a_{ii}$$

and this is calculated at the end of the function. Note that the original matrix is changed by `solven`; if we want to re-use this matrix later on, we should copy it into another matrix, which is then used as a parameter to solven, instead of the original matrix.

If we were using fractions here instead of real numbers, the following changes would be made:

- we would replace `double` by `fraction` throughout;
- we would simplify the function of section 8.13 above;
- we would eliminate the uses of `fsub`, and just use ordinary subtraction.

9. PROGRAMMING EXERCISE

As part of our course, the students are expected to rewrite the `solven` function of section 8.14 above, using fractions instead of integers, and making use of the changes suggested at the end of that section. They then run it on several cases and check the results.

REFERENCES

Maurer, W. (2000). The first course: C is not Pascal. *Journal of Computing in Small Colleges* , 15, 3, 91-100.