Math 4610 Lecture Notes

Vector and Matrix Operations in Computational Mathematics [*]

Joe Koebbe

December 14, 2019

[*]These notes are part of an Open Resource Educational project sponsored by Utah State University

Vector/Matrix Operations: Basic Operations on Vectors

The central algorithms used in most computer simulations involve computer codes that perform basic vector and matrix operations. Operations like matrix-vector multiplications are central to methods for solving linear systems of equations and determination of eigenvalues and eigenvectors. In this section of the class notes, we will cover a number of basic vector and matrix operations.
Suppose that we want to compute the solution of a linear system of equations in matrix form

$$A\,\mathbf{x} = \mathbf{b}$$

where A is the coefficient matrix, $\mathbf{b}$ is an input or right hand side vector, and we want to determine $\mathbf{x}$, a vector of unknowns. Due to issues related to roundoff error and machine precision representation of numbers, any vector we obtain will most likely be at best an approximation of the solution.
Since we know there will be errors, we will need some way to measure or compute the error associated with any approximation obtained. As we will see, there are infinitely many ways to measure errors. However, we will settle for three standard meausres of error introduced in the next section of the notes. To implement the error definitions, we will need to write codes that perform basic vector operations one would learn in a standard linear algebra course in a college curriculum. We will begin with basic algebraic operations involvinv vectors such as addition and scalar multiplication and then will proceed with the development codes for computing the magnitude/length of a vector and will end up with matrix algebra and operations.

Vector/Matrix Operations: Basic Operations on Vectors

For a set of objects to form a vector space a long list of properties must be satisfied. In these note, we will be working primarily with vectors in $\Re^n$ or possibly the space of vectors with complex entries. As such, we will assume that we are working with standard vector spaces. Any vector space or subspace must satisfy closure of addition and closure of scalar multiplication. This means that if V is a vector space and $\mathbf{u}$ and $\mathbf{v}$ are in the space, then the sum of the vectors is in V. That is, $\mathbf{w}$, defined by

$$\mathbf{w} = \mathbf{u} + \mathbf{v}$$

is also in the vector space. Also, if $\mathbf{v} \in V$ and a is any number/scalar then $\mathbf{u}$ defined by

$$\mathbf{v} = a\ \mathbf{u}$$

is also in V. Based on this idea, we should at least implement a pair of codes. One that computes the sum of two vectors and one that computes a scalar multiple of a vector.
Almost all of the work we will do will involve real vectors of length n. A vector will be of the form $(v_1, v_2, \ldots, v_n)^T$ where each component, $v_i, i = 1, \ldots, n$ is a real number. The superscript, T, denotes the transpose. It should be noted that there is a differnce between row and column vectors in linear algebra. In our vector notation, we will by default assume that a given vector is a column vector. Transposing a column vector results in a row vector and vice versa. This convention lends itself to operations like matrix-vector multiplication.
So, let's start with the sum of two vectors. The conventional definition of addition of vectors is a componentwise definition. If $\mathbf{u}$ and $\mathbf{v}$ are two vectors of length n. Then

$$\mathbf{u} + \mathbf{v} = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{bmatrix} + \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} u_1 + v_1 \\ u_2 + v_2 \\ \vdots \\ u_n + v_n \end{bmatrix}$$

For completeness, we can also write down a relationship between row and column vectors. That is,

$$\mathbf{v}^T = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}^T = [v_1, v_2, \cdots v_n]$$

and

$$[v_1, v_2, \cdots v_n]^T = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} = \mathbf{v}$$

This is one of the easiest operations that we can code. There are a few things that might need checking. For example, we might need to check the dimension of the two vectors. Without any checks on compabilitiy of dimensions, the code, written in Java might look like the following.

```
    public double[] vecadd(double[] u, double[] v) {
      //
      // get the dimension of the first vector and instantiate some storage for
      // the output vector
      // ----------------
      //
      int n = u.length;
      double [] output = new double[n];
      //
      // compute the componentwise sum for the vector
      // --------------------------------------------
      //
      for(int i=0; i<n; i++) output[i] = u[i] + v[i];
      //
      // return the output
      // -----------------
      //
      return output;
      //
    }
```

Before coding a scalar multiple method for our library, it would be best to write a code for computing the difference in a pair of vectors. Given two vectors **u** and **v**, the formula for the difference is the following.

$$\mathbf{u} - \mathbf{v} = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{bmatrix} - \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} u_1 - v_1 \\ u_2 - v_2 \\ \vdots \\ u_n - v_n \end{bmatrix}$$

It is an easy exercise to write a code that will compute the difference of two vectors. All that needs to happen is to give the routine a unique name and change the addition to a subtraction. For exaxmple,

---

```
  public double[] vecsub(double[] u, double[] v) {
    //
    // get the dimension of the first vector and instantiate some storage for
    // the output vector
    // -----------------
    //
    int n = u.length;
    double [] output = new double[n];
    //
    // compute the componentwise sum for the vector
    // --------------------------------------------
    //
    for(int i=0; i<n; i++) output[i] = u[i] - v[i];
    //
    // return the output
    // -----------------
    //
    return output;
    //
  }
```

This particular routine can be used as a means to compute the difference between two vectors as a part of measuring errors in vector approximations. Note that the dimensions of the two vectors in the subtraction of vectors has not been checked. This means it is up to the user to ensuere that the two vectors have the same length.

The last code to write in this section is the scalar multiplication code. The vector operation is defined as follows. For any vector, $\mathbf{v}$, and number, a, scalar multiplication is defined by

$$a\,\mathbf{v} = a \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} a\ v_1 \\ a\ v_2 \\ \vdots \\ a\ v_n \end{bmatrix}$$

and the code to implement this operation is given by

```
public double[] scalvec(double a, double[] v) {
  //
  // get the dimension of the first vector and instantiate some storage for
  // the output vector
  // -----------------
  //
  int n = u.length;
  double [] output = new double[n];
  //
  // compute the componentwise sum for the vector
  // --------------------------------------------
  //
  for(int i=0; i<n; i++) output[i] = a * v[i];
  //
  // return the output
  // -----------------
  //
  return output;
  //
}
```

In the scalar multiplication, the length of the vector can be any integer and as long as the vector is not null, the method will work. However, it might be necessary to test to see if the vector actually exists. If a null vector is passed to any method, there is likely to be a problem. So, in a software manual you would need to warn the user of this and any other assumptions that might cause problems.
The methods developed in this section of the notes provides a starting point for a number of other methods. It will be easy to cut and paste and rename methods to do most of the basic matrix operations outlined in these notes.

Vector/Matrix Operations: The Euclidean Length of a Vector

In many problems it will be necessary to measure the length or magnitude of a vector (at least so say the villian in Despicable Me). So, we need some way to determine the length. Actually, there are infinitely many ways to do this. The approach most students see first involves using the Euclidean distance between two points used to define a vector. From any standard linear algebra course, the Euclidean length can be defined as follows.

$$||v|| = (v_1^2 + v_2^2 + \cdots + v_n^2)^{1/2}$$

where the notation, $|| * ||$, provides a mathematical notation for the magnitude of the vector. The notation emphasizes the difference between the absolute value of the difference of two real numbers and the length of a vector. This can be implemented easily into a method or routine that will return this length of a vector.

```
public double l2norm(double[] v) {
  double sum = 0.0;
  //
  // extract the length of the vector
  // -------------------------------
  //
  int n = v.length;
  //
  // compute the sum of squares
  // -------------------------
  //
  for(int i=0; i<n; i++) sum = sum + v[i] * v[i];
  //
  // return the value
  // ----------------
  //
  return Math.sqrt(sum);
  //
}
```

Note that the Java package including the square root function defines the last step in the code. It is best to use an intrinsic for the square root since implementing our own version of a square root method would take some time.

Vector/Matrix Operations: Definition of the Norm of a Vector

There are many ways to compute the length of a vector. In this section we will consider several ways to do this. However, we need to have a general definition of what is meant by the length of a vector. To start, the definition of a norm is given.

Definition 1 *Suppose that V is a vector space and $\mathbf{u}$ and $\mathbf{v}$ are any two vectors in V. Also, assume a is an arbitrary number/scalar. The norm of a vector is a function*

$$|| * || : V \to \Re$$

such that

1. $||v|| = 0$ *if and only if* $\mathbf{v} = \mathbf{0}$,
2. $||a\mathbf{v}|| = |a| \; ||\mathbf{v}||$, *and*
3. $||\mathbf{u} + \mathbf{v}|| \leq ||\mathbf{u}|| + ||\mathbf{v}||$.

The Euclidean magnitude/length of a vector can be shown to satisfy this definition of a norm. This means we can use the terms length, magnitude, and norm interchangeably. The term norm is used in the name, **l2norm**, chosen for the method in the code above.
There are an infinite number of ways to compute norms on vector spaces. A general definition for a norm on n-dimensional real space is the following.

$$||v||_p = (v_1^p + v_2^p + \cdots + v_n^p)^{1/p} = \left(\sum_{i=1}^{n} \; |v_i|^p \right)^{1/p}$$

for any positive integer p. The Euclidean norm is obtained when $p = 2$. The reality in computational mathematics is that there are three norms of interest including the 2-norm. The other two norms are the 1-norm and the other is the infinity-norm. The definitions for these two norms are given below.

$$||v||_1 = |v_1| + |v_2| + \cdots + |v_n| = \sum_{i=1}^{n} \; |v_i|$$

and

$$||v||_\infty = \max_{1 \leq i \leq n} |v_i|$$

are the definitions we will use in our work. The gold standard for measuring length is typically the 2-norm. However, in some problems the 1-norm and infinity-norm are easier to compute in many problems.
The 1-norm and infinity-norm can easily be coded into methods as was done above to the 2-norm. Using the 2-norm code, the other two methods just need a couple of minor changes to implement the other two norms into their own methods.

Vector/Matrix Operations: Errors in Vector Approximations

Just as the errors in approximating roots of functions of a single variable, having a way to compute the magnitude of the error in approximating one vector $\mathbf{v}$ by another vector $\mathbf{u}$. We can use the definitions of norms earlier in this section to define a consistent formula for the vector approximation error. If we use a generic definition of the norm of a vector, we can define

$$\text{absolute error} = ||\mathbf{v} - \mathbf{u}||$$

and

$$\text{relative error} = \frac{||\mathbf{v} - \mathbf{u}||}{||\mathbf{u}||}$$

These formulas should look familiar when compared to error measurement in root finding problems.
A code that will implement the absolute error with the 2-norm might look like

```
public double absl2err(double[] u, double[] v) {
  double sum = 0.0;
  //
  // extract the length of the vector
  // --------------------------------
  //
  int n = v.length;
  double diff = 0.0;
  for(int i=0; i<n; i++) {
    diff = u[i] - v[i];
    sum = sum + diff * diff;
  }
  //
  // return the norm of the difference
  // ---------------------------------
  //
  return Math.sqrt(sum);
  //
}
```

Of course, we could reuse code that we have already written as follows. Provided that the methods have been created and test and inserted in some sort of archive that is available for our use. A first simpler version is the following.

```
public double absl2err(double[] u, double[] v) {
  double [] diff = null;
  int n = v.length;
  diff = vecsub(u, v);
  return l2norm(diff);
}
```

An even more concise version of the method might like the following.

```
public double absl2err(double[] u, double[] v) {
  return l2norm(vecsub(u, v));
}
```

Note that in most cases, calling another method or routine requires a bit of overhead. The first version of the code will be more efficient than the last version. However, it is a lot easier to maintain one version of a code than to modify multiple codes all of which have in-lined versions of any reusable code. In native languages like C or Fortran, you can always compile the code into a static executable that in-lines all of these types of codes just as if the code was all there.

Vector/Matrix Operations: Dot Products

There are several vector products that should also be included in any discussion of computational linear algebra. The first of these is the usual dot product or inner product on $\Re^n$. To start, suppose we have two vectors out of the same vector space. Then the dot product of the two vectors is given by

$$\mathbf{u} \cdot \mathbf{v} = u_1\ v_1 + u_2\ v_2 + u_3\ v_3 + \cdots + u_1\ v_1$$

This product is about as easy to code as anything. In C, the code would look like

```
//
// dot product routine written in C
// -------------------------------
//
double dotprd(double u[], double v[], int n)
{
  //
  // some variable declarations
  // -------------------------
  //
  double sum = 0.0;
  //
  // compute the dot product
  // ----------------------
  //
  for(int i=0; i<n; i++) sum  = sum + u[i] * v[i];
  //
  // return the value
  // ---------------
  //
  return sum;
  //
}
```

The code given above passes in the length of the arrays that represent the two vectors. Note that you can write code that will determine the size of the arrays internally using the *sizeof* operator. In other languages like Java there are operators for getting the size of the array. In the case above, the user would need to make sure things match up.

There are weighted inner products that can be defined as modifications to the standard dot product. For example, the energy norm with respect to a given symmetric positive definite matrix, A. This type of inner product starts with the definition of an energy norem. So, suppose that the matrix, A, is symmetric and positive definite. Then the energy norm is defined by

$$||\mathbf{x}||_A = \sqrt{\mathbf{x}^T\ A\ \mathbf{x}}$$

The energy norm is used in some applications in the approximation of solutions of differential equations and in the development of the conjugate gradient method for iteratively solving linear systems of equations. These topics will be addressed later in these lectures.

Vector/Matrix Operations: The Cross Product

The second product that is used in some computational problems is the cross product of two vectors. If the cross product of two three dimensional vectors is computed, the result is another three dimensional vector. The formulas for the components of the output vector are the following. If **u** and **v** are two three dimensional vectors, then

$$\mathbf{u} \times \mathbf{v} = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} \times \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} (u_2\ v_3 - u_3\ v_2) \\ -(u_1\ v_3 - u_3\ v_1) \\ (u_1\ v_2 - u_2\ v_1) \end{bmatrix}$$

Again, this is a relatively easy to program, but it should be noted that this product is valid only in three dimensions. One version of a code might look like the following.

```
//
// the cross product of two vectors
// --------------------------------
//
double [] crsprd(double u[], double v[])
{
  //
  // some variable declarations
  // --------------------------
  //
  double cross[3];
  cross[0] = u[1] * v[2] - u[2] * v[1];
  cross[1] = u[2] * v[0] - u[0] * v[2];
  cross[2] = u[0] * v[1] - u[1] * v[0];
  //
  // return the value
  // ----------------
  //
  return cross;
  //
}
```

Note that the second component is written to absorb the negative in front of the second component. This will be slightly more efficient.

Vector/Matrix Operations: The Vector Outer Product

As we start into this type of vector product, the standard inner product can be easily written as

$$\mathbf{u} \cdot \mathbf{v} = \mathbf{u}^T \mathbf{v} = \mathbf{v}^T \mathbf{u}$$

That is, the dot product can be couched in terms of matrix/vector multiplication. This is just an interpretation of the dot product in terms of matrix multiplication - a $1 \times n$ times a $n \times 1$ giving a one by one result or number. So, what happens if we reverse the product. That is,

$$A = \mathbf{u} \, \mathbf{v}^T$$

We use a matrix, A, as the output from the product. Note that this product is well defined. That is, $n \times 1$ times a $1 \times n$ giving a matrix that is $n \times n$. This type of product is called an outer product and is a useful addition to a software library. Some forms of matrix multiplication can be written in terms of an outer product. In addition, whereas the dot product requires two vectors of the same length, the outer product is well defined for any pair of vectors.

```
//
// routine to compute the outer product of two vectors
// ---------------------------------------------------
//
double [] otrprd(double u[], double v[], int m, int n)
{
  double A[m][n];
  //
  // a double loop is needed to compute the components
  // -------------------------------------------------
  //
  for(int i=0; i<m; i++) {
    for(int j=0; j<n; j++) {
      a[i][j] = u[i] * v[j];
    }
  }
  //
  // return the value
  // ----------------
  //
  return A;
  //
}
```

The result of this product is a matrix. It should be noted that the resulting matrix is a rank-1 matrix. This means that the output is a matrix with only one independdent vector. The rest of the vectors are a constant multiple of the first nonzero vector. If any of the components of either of the vectors is zero, then at least one row or column will be the zero vector.

Vector/Matrix Operations: Matrix Addition

The next step is to work through matrix algebra that help define sets of matrices that form linear vector spaces. The first operation is matrix addition. So, given two matrices, A and B, that are the same size - same number of rows and columns in both matrices, matrix multiplication is defined using the following.

$$A+B=\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix}+\begin{bmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,n} \\ b_{2,1} & b_{2,2} & \cdots & b_{2,n} \\ \vdots & \vdots & & \vdots \\ b_{m,1} & b_{m,2} & \cdots & b_{m,n} \end{bmatrix}$$

$$=\begin{bmatrix} (a_{1,1}+b_{1,1}) & (a_{1,2}+b_{1,2}) & \cdots & (a_{1,n}+b_{1,n}) \\ (a_{2,1}+b_{2,1}) & (a_{2,2}+b_{2,2}) & \cdots & (a_{2,n}+b_{2,n}) \\ \vdots & \vdots & & \vdots \\ (a_{m,1}+b_{m,1}) & (a_{m,2}+b_{m,2}) & \cdots & (a_{m,n}+b_{m,n}) \end{bmatrix}$$

The formula is just an extension of vector addition. The only restriction on the operation of matrix addition is that the two matrices, A and B, must be the same size. We can rewrite the formula for addition as follows: If C is a matrix that is the result of adding matrices, A and B, then the components of C satisfy

$$(C)_{i,j}=c_{i,j}=a_{i,j}+b_{i,j}=(A)_{i,j}+(B)_{i,j}$$

In the previous equation, there are a couple of things going on. First, the notation using $(A)_{i,j}$ refers to the component in the i^{th} row and j^{th} column of A. We will also use lower cased letters to indicate the same thing. The second thing to notice is that the component-wise formula just given provides the formula we need in a routine that will add two matrices. A code that provides the sum of two matrices is the following.

```
//
// matrix addition code in C
// -------------------------
//
double [][] matadd(double A[][], double B[][], int m, int n)
{
  double C[m][n];
  //
  // a double loop is needed to compute the components of the sum of two
  // matrices
  // --------
  //
  for(int i=0; i<m; i++) {
    for(int j=0; j<n; j++) {
      C[i][j] = A[i][j] + B[i][j];
    }
  }
  //
  // return the value
  // ----------------
  //
  return C;
  //
}
```

Keep in mind that C/C++ and linux and Unix are in general case-sensittive. In many applications, capitalized letters are assumed to be environment or global variables. So, it might be a good idea to rewrite the code with different variable names. For example,

```
//
// an alternate version of doing a matrix addition
// ----------------------------------------------
//
double [][] matadd(double amat[][], double bmat[][], int m, int n)
{
  double cmat[m][n];
  //
  // a double loop is needed to compute the components of the sum of two
  // matrices
  // --------
  //
  for(int i=0; i<m; i++) {
    for(int j=0; j<n; j++) {
      cmat[i][j] = amat[i][j] + bmat[i][j];
    }
  }
  //
  // return the value
  // ----------------
  //
  return cmat;
  //
}
```

There are also some things that can be done to speed these types of operations up through parallelism or using the ordering of values in the rows and columns of a matrix stored in a specific programming language like C or Fortran.

Vector/Matrix Operations: Matrix Subtraction

Matrix subtraction involves a trivial modification of the matrix addition method given in the previous section of these notes. The shorthand notation is the following. A matrix, C, is the difference of two matrices A and B if we write $c_{i,j}$ is the entry in i^{th} row and j^{th} column of C, then

$$c_{i,j} = a_{i,j} - b_{i,j}$$

A code that will work using lower case variable names is the following.

```
//
// code to compute the difference of two matrices of the same dimension
// --------------------------------------------------------------------
//
double [][] matsub(double amat[][], double bmat[][], int m, int n)
{
  double cmat[m][n];
  //
  // a double loop is needed to compute the components of the sum of two
  // matrices
  // --------
  //
  for(int i=0; i<m; i++) {
    for(int j=0; j<n; j++) {
      cmat[i][j] = amat[i][j] - bmat[i][j];
    }
  }
  //
  // return the value
  // ----------------
  //
  return cmat;
  //
}
```

The modifications to matrix addition involve a name change and a single change in the code.

Vector/Matrix Operations: Scalar Matrix Multiplication

This operation multiplies a scalar into a matrix by multiplying the number into each entry in the matrix as follows.

$$\alpha\ A = \alpha \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix} = \begin{bmatrix} \alpha\ a_{1,1} & \alpha\ a_{1,2} & \cdots & \alpha\ a_{1,n} \\ \alpha\ a_{2,1} & \alpha\ a_{2,2} & \cdots & \alpha\ a_{2,n} \\ \vdots & \vdots & & \vdots \\ \alpha\ a_{m,1} & \alpha\ a_{m,2} & \cdots & \alpha\ a_{m,n} \end{bmatrix}$$

In terms of the components of the output, assume that a matrix, B, is the result of multipling a matrix A by a number. This implies

$$b_{i,j} = \alpha\ a_{i,j}$$

and a code to complete this task might look like the following.

```
//
// routine to compute a constant/scalar multiple of a matrix
// ------------------------------------------------------
//
double [][] scalmat(double alpha, double amat[][], int m, int n)
{
  double bmat[m][n];
  //
  // a double loop is needed to compute the components of the sum of two
  // matrices
  // --------
  //
  for(int i=0; i<m; i++) {
    for(int j=0; j<n; j++) {
      bmat[i][j] = alpha * amat[i][j];
    }
  }
  //
  // return the value
  // ----------------
  //
  return bmat;
  //
}
```

The mathematical importance of matrix addition and scalar multiplication imply that the set of all $m \times n$ real matrices is closed under these operations which gives consistency of these operations.

Vector/Matrix Operations: The Determinant of a Matrix

The determinant of a matrix is presented in most linear algebra courses and is an indispensible tool in analyzing the properties of matrices. The standard way of computing the determinant of a matrix is to use cofactor expansion. We will review a couple of examples of computing determinants, but determinants are computationally intensive and should not be computed unless it is absolutely necessary. In some cases, the matrix has a special structure which lends itself to an easy computation of the determinant. For example, the determinant of any upper or lower triangular or is a diagonal matrix, the determinant is the product of the diagonal values of the matrix. In general, the work needed depends on the size of the matrix. If a matrix, A, is square with the number of rows and columns is some integer, n. Then, the number of flops needed to complete the computation of a determinant is $n!$. This is a large number of flops.
For example, for a 2×2 matrix, say

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{bmatrix}$$

the determinant is computed as

$$det(A) = |A| = a_{1,1}\ a_{2,2} - a_{1,2}a_{2,1}$$

and for the 3×3 case,

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix}$$

the result is

$$det(A) = a_{1,1}\ \ (a_{2,2}\ a_{3,3} - a_{3,2}\ a_{2,3}) - a_{2,1}\ \ (a_{1,2}\ a_{3,3} - a_{3,2}\ a_{2,3}) + a_{3,1}\ \ (a_{1,2}\ a_{2,3} - a_{2,2}\ a_{1,3})$$

The result can be extended to any size matrix, but that is the topic for a linear algebra course. It is rarely the case that we will need to evaluate or compute a determinant beyond a 3×3 matrix. The main issue that we are faced in computing determinants is the sheer amount of work needed to complete the computation. The number of flops need is $O(n!)$ and we already know that $n!$ gets large relatively quickly.

Vector/Matrix Operations: The Transpose of a Matrix

For any matrix, we can compoute the transpose a matrix, $A \in \Re^{n \times n}$, by interchanging rows and columns of the matrix. The shorthand definition of the transpose is if $A \in \Re^{n \times n}$ is any matrix, then $B = A^T$ using the definition

$$b_{i,j} = a_{j,i}$$

So, a routine for computing the transpose of a matrix will look like the following.

```
//
// computing the transpose of a matrix
// -----------------------------------
//
double [][] mattrns(double amat[][], int m, int n)
{
  double bmat[n][m];
  //
  // a double loop is needed to compute the components of the sum of two
  // matrices
  // --------
  //
  for(int i=0; i<m; i++)
    for(int j=0; j<n; j++)
      bmat[j][i] = amat[i][j];
  //
  // return the value
  // ----------------
  //
  return bmat;
  //
}
```

The code clearly switches the roles of the rows and columns in the output matrix by interchanging the indices in the loop. Note also that the dimensions of the temporary storage, **bmat**, are switched when used to allocate enough space to do the operation.
Note that this routine returns a matrix with the same number of entries as the original. This may cause some issues when a code to solve problems with large matrices is considered. As an example, suppose that we need to compute the action of the tranpose on a vector. That is, something like

$$\mathbf{y} = A^T \ \mathbf{x}$$

Even though it might be tempting to do the transpose first, physically exchanging the rows and columns is not necessary. A code to do complete the multiplication operation is the following.

```
double [] atrnx(double A[][], double x[], int m, int n)
{
  double output[n];
  //
  // loop over the rows in the matrix A - columns of the transpose of A
  // ------------------------------------------------------------------
  //
```

```
    for(int i=0; i<n; i++) {
      //
      // initialize the output value
      // --------------------------
      //
      output[i] = 0.0;
      //
      // loop over the columns of the matrix A.....
      // ------------------------------------------
      //
      for(int j=0; j<m; j++)
      {
        //
        // add the appropriate contribution
        // --------------------------------
        //
        output[i] = output[i] + a[j][i] * x[j];
      }
    }
    //
    return output;
  }
```

Note the change in the order of the indices used from the input matrix, **amat**, to the output matrix, **bmat**. The main advantages are that we do not need to double the storage in the process of explicitly storing the transpose of the original matrix. In addition, the original matrix is retained and can be resused for any other computations. Note that we will revisit matrix multiplication later in this set of notes. You may need to review this after visiting the section on matrix vector multiplication.

Vector/Matrix Operations: The Trace of a Matrix

The trace of a matrix is just the sum of the entries on the main diagonal of the matrix. This is useful in a limited number of computational codes, but is presented for completeness of the presentation.

```
//
// this is a simple routine to add up the diagonal entries in a square
// matrix
// ------
//
double [][] mattrc(double amat[][], int n)
{
  double output;
  //
  // a single loop is enough to grab and add the diagonal values
  // -----------------------------------------------------------
  //
  for(int i=0; i<m; i++) output = output + amat[i][i];
  //
  return output;
  //
}
```

One property of square matrices is that the trace of the matrix is the same as the sum of the eigenvalues of the matrix. However, the trace, besides being trivial to code up, is not used very often in computational mathematics.

Vector/Matrix Operations: Matrix-Norms

An important concept in computational linear algebra is that of matrix norms. Most error estimates in computational linear algebra are stated in terms of inequalities involving the norm of a matrix. There are an infinite number of measures of the norm or size of a matrix. One example of a norm is the Frobenius norm. Given a square matrix, $A \in \Re^{n\times n}$, the definition of the norm is

$$||A|| = \left(\sum_{i=1}^{n} \sum_{j=1}^{n} |a_{i,j}|^2 \right)^{1/2}$$

The Frobenius norm looks a bit like the 2-norm for a vector. The similarity is that you can think of concatenating the rows (or columns) of the matrix and then apply the vector 2-norm to the $n \times n$ length vector that results. The Frobenius norm is used sparingly in computational linear algebra.
The main norms that are used are called induced norms and are defined in terms of vector norms. Recall that the p-norm of a vector of length n is defined by

$$||\mathbf{v}||_p = \left(\sum_{i=1}^{n} |v_i|^p \right)^{1/p}$$

for $p = 1, 2, \ldots$ and any vector, $\mathbf{v} \in \Re^n$. Note that the definition changes slightly if we allow $p = \infty$. In this case

$$||\mathbf{v}||_\infty = \max_{1 \le n} |v_i|$$

The p-norms can be used to define norms of matrices. The induced p-norm of a matrix, $A \in \Re^{m\times n}$, is defined by

$$||A||_p = \max_{||\mathbf{x}||_p \neq 0} \frac{||A\ \mathbf{x}||_p}{||\mathbf{x}||_p}$$

This definition is based on vector p-norms. Note that the p-norm in the numerator of the expression is applied to vectors of length m, while the norm in the denominator is applied to vectors of length n.
With a bit of work, we can prove that the induced matrix norms can be written as

$$||A||_p = \max_{||\mathbf{x}||_p = 1} ||A\ \mathbf{x}||_p$$

This means that matrix norm can be determined on vectors of length one in the corresponding p-norm. The set of vectors of length one identifies the " unit ball" for the norm.
Although any $p \in (0, \infty)$ can be used, the most commonly used norms in computational work are $p = 1$, $p = 2$, and $p = \infty$. These will be referred to as the 1-norm, the 2 or Euclidean norm, and the ∞ or " sup"-norm, respectively. For completeness, the 1-norm defined by

$$||A||_1 = \max_{||\mathbf{x}||_1 = 1} ||A\ \mathbf{x}||_1$$

the 2-norm or Euclidean norm is defined by

$$||A||_2 = \max_{||\mathbf{x}||_2 = 1} ||A\ \mathbf{x}||_2$$

and the ∞-norm or sup-norm is defined by

$$||A||_\infty = \max_{||\mathbf{x}||_\infty = 1} ||A\ \mathbf{x}||_\infty$$

The definitions of these norms are not necessarily very helpful. However, the 1-norm and the ∞-norm can be computed for any matrix. The 2-norm requires the computation of eigenvalues or singular values of the matrix. This computation will be taken up in a later section in the lectures.

Vector/Matrix Operations: Two Important Computable Matrix Norms

A formula that allows us to compute the 1-norm of a matrix, A, involves the determination of the maximum absolute column sum. That is,

$$||A||_1 = \max_{1 \le j \le n} \sum_{i=1}^{m} |a_{i,j}|$$

and the formula for the ∞-norm of the same matrix involves the determination of the maximum absolute row sum. That is,

$$||A||_\infty = \max_{1 \le i \le m} \sum_{j=1}^{n} |a_{i,j}|$$

Both of these norms can be computed with nothing more than the entries of the matrix.
A code for computing the 1-norm of a matrix might look like the following.

```
double l1matnrm(double a[][], int m, int n)
{
  double output = 0.0;
  //
  // loop over the columns of the matrix
  // ---------------------------------
  //
  for(int j=0; j<n; j++) {
    //
    // provide a temporary storage value
    // --------------------------------
    //
    double tmpval = 0.0;
    //
    // loop over the entries in the column adding in the absolute value
    // ----------------------------------------------------------------
    //
    for(int i=0; i<m; i++)
       tmpval = tmpval + abs( a[i][j];
    //
    // test this column sum for the maximum value so far
    // ----------------------------------------------
    //
    if(tmpval > output) output = tmpval;
  }
  //
  return output;
  //
}
```

The algorithm for the ∞-norm is just as easy to write and might look like the following.

```
double linfmatnrm(double a[][], int m, int n)
{
  double output = 0.0;
```

```
    //
    // loop over the rows of the matrix
    // --------------------------------
    //
    for(int i=0; i<m; i++) {
      //
      // provide a temporary storage value
      // ---------------------------------
      //
      double tmpval = 0.0;
      //
      // loop over the entries in the row adding in the absolute value
      // -------------------------------------------------------------
      //
      for(int j=0; j<m; j++)
         tmpval = tmpval + abs( a[i][j];
      //
      // test this column sum for the maximum value so far
      // -------------------------------------------------
      //
      if(tmpval > output) output = tmpval;
    }
    //
    return output;
    //
  }
```

All we needed to do is switch the order of the two loops to move from the matrix 1-norm to the matrix ∞-norm.

Vector/Matrix Operations: Formal Matrix Norm Definition, Consistent Norms

In the case of vector norms, we have definition of a norm that requires three properties must be satisfied. Matrix norms also satisfy these requirements.

Definition 2 *Suppose that V is a vector space of matrices and* $\mathbf{A}$ *and* $\mathbf{B}$ *are any two matrices in V. Also, assume c is an arbitrary number/scalar. The norm of a matrix is a function*

$$||*||: V \to \Re$$

such that

1. $||A|| = 0$ *if and only if* $A = 0$,
2. $||c\ A|| = |c|\ ||A||$, *and*
3. $||A + B|| \leq ||A|| + ||B||$.

There are a lot of ways to define a norm for matrices. In many results in linear algebra an additional property for a matrix norm is helpful. The following definition adds a fourth property to the norm.

Definition 3 *Suppose that V is a vector space of matrices and* $\mathbf{A}$ *and* $\mathbf{B}$ *are any two matrices in V. Also, assume that the function $||*||: V \to \Re$ is a norm. The norm is a consistent norm if the following additional property is satisfied*

$$||A\ B|| \leq ||A||\ ||B||$$

Note that all matrix p-norms as have already been defined are consistent norms.

Vector/Matrix Operations: Comparison/Equivalence of Norms in Finite Dimensions

In previous sections of these lectures, we learned that the norm of a vector or matrix is not a unique function. For example, the p-norms of vectors all give a value that represents the length of a vector. In fact, there are infinitely many norms that we can choose from. A valid question is which one should we use or maybe, which norm is the best norm to use for a specific problem? In computational mathematics, we will work in finite dimensional vector spaces. The finite precision of number representation and finite precision of computer arithmetic guarantee that this will always be the case. It turns out that the answer to the question of which norm should be used is that it does not matter due to the concept of equivalence of norms.

Definition 4 *Two norms $||\cdot||_A$ and $||\cdot||_B$ are equivalent if there exist positive constants, C_1 and C_2, such that*

$$C_1\ ||\mathbf{x}||_A \leq ||\mathbf{x}||_B \leq C_2||\mathbf{x}||_A$$

for all $\mathbf{x}$ in the underlying vector space. Note that this means that we can bound one of the norms in terms of the other. It can be shown that if the above inequalities are true, one can write.

$$C_3\ ||\mathbf{x}||_B \leq ||\mathbf{x}||_A \leq C_4||\mathbf{x}||_B$$

for positive constants, C_3 and C_4.

An important result for our work is contained in the following theorem.

Theorem 1 *Suppose V is a finite-dimensional matrix and $||\cdot||_\alpha$ and $||\cdot||_\beta$ are any two norms defined on V. Then the two norms are equivalent. That is, there exist two positive constants, C_1 and C_2, such that*

$$C_1||\mathbf{u}||_\alpha \leq ||\mathbf{u}||_\beta \leq C_2||\mathbf{u}||_\alpha$$

for all $\mathbf{u} \in V$.

It is important to have some estimate of the relationahips between the norms in terms of the bounding coefficients, C_1 and C_2.

Vector/Matrix Operations: Bounds Relating $p = 1$, $p = \infty$, and $p = 2$ Norms

As mentioned previously, the norms we use are basically the same up to constant bounds. The 1-norm, 2-norm, and ∞-norm can be used to bound each other as follows. In general, for the vector norms and any vector, $\mathbf{x}$.

$$||\mathbf{x}||_\infty \leq ||\mathbf{x}||_2 \leq ||\mathbf{x}||_1$$

and equivalence relationships

$$||\mathbf{x}||_\infty \leq ||\mathbf{x}||_2 \leq \sqrt{n}\ ||\mathbf{x}||_\infty$$

and

$$||\mathbf{x}||_\infty \leq ||\mathbf{x}||_1 \leq n\ ||\mathbf{x}||_\infty$$

Since the matrix p-norms are constructed from the vector p-norms, analogous inequalities can be written down for matrices. For a matrix $A \in \Re^{m \times n}$ some important inequalties are the following.

1. $$||A||_2 \leq ||A||_F \leq \sqrt{n}\ ||A||_2$$

2. $$\max_{i,j} |a_{i,j}| \leq ||A||_2 \leq \sqrt{mn}\ |\max_{i,j} |a_{i,j}|$$

3. $$\frac{1}{\sqrt{n}}\ ||A||_\infty\ \leq ||A||_2 \leq \sqrt{m}\ ||A||_\infty$$

4. $$\frac{1}{\sqrt{m}}\ ||A||_1\ \leq ||A||_2 \leq \sqrt{n}\ ||A||_1$$

where the subscript F denotes the Frobenius norm. We should keep in mind that the matrix 2-norm is the norm that is being bounded by the matrix 1-norm and matrix ∞-norm. Since we can compute the other two norms this makes sense. In addition, the matrix 2-norm requires computing the eigenvalues or singular values of the matrix. This take a lot of computational effort.

Vector/Matrix Operations: Matrix Condition Number

A great many problems in computational linear algebra involve the solution of a linear system of equations

$$A\,\mathbf{x} = \mathbf{b}$$

where $A \in \Re^{n\times n}$ and $\mathbf{b} \in \Re^{n}$ are a given matrix and right hand side for the linear system. From a theoretical point of view, if the coefficient matrix, A, has an inverse, then a solution exists for the linear system. Normally, we will use A^{-1} to denote the inverse of A and we can write the equation

$$\mathbf{x} = A^{-1}\,\mathbf{b}$$

Computationally, we (almost) never compute the inverse of the coefficient matrix explicitly. Instead, we compute the action of the inverse of the matrix without actually computing A^{-1}.
We would still like to have a measure how well an algorithm will work in the solution of some linear algebra problem. Suppose that $\mathbf{x}^*$ is the exact solution of the linear system above. That is,

$$A\,\mathbf{x}^* = \mathbf{b} \quad\rightarrow\quad \mathbf{x}^* = A^{-1}\,\mathbf{b}$$

and suppose that $\mathbf{x}$ is an approximate solution for the linear system. The residual vector for the approximation is given by

$$\mathbf{r} = \mathbf{b} - A\,\mathbf{x}$$

Subtracting the original linear system equation from the equation for the residual vector equation just defined gives

$$\mathbf{r} - \mathbf{0} = (\mathbf{b} - A\,\mathbf{x}) - (\mathbf{b} - A\,\mathbf{x}^*) = A\;(\mathbf{x}^* - A\,\mathbf{x})$$

or rewriting gives the equation

$$A\,(\mathbf{x}^* - \mathbf{x}) = \mathbf{r} \quad\rightarrow\quad \mathbf{x}^* - \mathbf{x} = A^{-1}\,\mathbf{r}$$

This gives a measure of the absolute error when we apply a vector norm to both sides of the equation. That is,

$$\text{absolute error}\quad ||\mathbf{x}^* - \mathbf{x}|| = ||A^{-1}\,\mathbf{r}||$$

If we want to use the relative error measure, we will write

$$\text{relative error}\quad \frac{||\mathbf{x}^* - \mathbf{x}||}{||\mathbf{x}^*||}$$

We need an estimate of the denominator.
To do this, consider the original system of equations

$$A\,\mathbf{x}^* = \mathbf{b}$$

Taking vector norms of both sides of the equation using a consistent matrix norm gives

$$||\mathbf{b}|| = ||A\,\mathbf{x}|| \le ||A||\;||\mathbf{x}||$$

Using some arithmetic and the fact the norm is consistnt, we can write

$$\frac{1}{||\mathbf{x}||} \le \frac{||A||}{||\mathbf{b}||}$$

So, the relative error can be bounded as follows.

$$\frac{||\mathbf{x}^* - \mathbf{x}||}{||\mathbf{x}^*||} \le \frac{||A^{-1}||\;||\mathbf{r}||\;||A||}{||\mathbf{b}||} = ||A^{-1}||\;||A||\;\frac{||\mathbf{r}||}{||\mathbf{b}||}$$

The product of the matrix norms is called the condition number of the matrix, A.

Definition 5 *Suppose that A is an invertible matrix. The condition number, κ, is defined by*

$$\kappa_\alpha(A) = ||A^{-1}||_\alpha \; ||A||_\alpha$$

The value of the condition number is dependent on the norm chosen.

It should be noted that $\kappa_\alpha(A)$ is always greater than or equal to one. Values of $\kappa\alpha(A)$ closer to one indicate that the matrix is well-conditioned. The larger $\kappa_\alpha(A)$ is, the more poorly conditioned the matrix is. One can take a look at the Hilbert matrix as an example of a poorly conditioned matrix.
As an example, let's compute the condition number of a matrix using the matrix ∞-norm. ..... need a matrix for the example

---

Vector/Matrix Operations: Matrix-Vector Multiplication

Suppose that we are asked to solve a system of three linear equations that can be written in the fowm

$$\begin{aligned} a_{1,1}\ x_1 &+ a_{1,2}\ x_2 + a_{1,3}\ x_3 &= b_1 \\ a_{2,1}\ x_1 &+ a_{2,2}\ x_2 + a_{2,3}\ x_3 &= b_2 \\ a_{3,1}\ x_1 &+ a_{3,2}\ x_2 + a_{3,3}\ x_3 &= b_3 \end{aligned}$$

where the coefficients, $a_{i,j}$, are given real numbers, b_i, are real numbers, and the unknown variables are x_i all of these for $i = 1, 2, 3$. The system of equations can be translated into a matrix-vector equation as follows.

$$A\ \mathbf{x} = \mathbf{b}$$

where

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

This is a place where a matrix-vector multiplication $A\ \mathbf{x}$ is part of the expression.
In this section of the notes, we will see how to compute a matrix-vector product. The operation basically, computes the (dot) product of each row and the column vector $\mathbf{x}$. We can define this operation in terms of the output we expect. For a general matrix-vector product

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

where $A \in \Re^{m\times n}$, $x \in \Re^n$, and $y \in \Re^m$. Thus the output is a vector of length m where

$$y_i = a_{i,1}\ x_1 + a_{i,2}\ x_2 + \cdots + a_{i,n}\ x_n = b_i$$

for $i = 1, 2, \ldots, m$. This dot product definition can be translated into a computer code as follows.

```
double matvecmlt(double a[][], double x[], int m, int n)
{
  double output[m];
  //
  // loop over the rows of the matrix
  // --------------------------------
  //
  for(int i=0; i<m; i++) {
    //
    // iniitalize the component of the output vector, y
    // ------------------------------------------------
    //
    output[i] = 0.0;
    //
    // loop over the entries in the row each new contribution to the output
    // --------------------------------------------------------------------
    //
    for(int j=0; j<m; j++) {
      output[i] = output[i] + a[i][j] * x[j];
    }
    //
  }
  //
  return output;
  //
}
```

There are a couple of things to notice in this definition. The inner loop computes the dot product of the i^{th} row in the coefficient matrix with the column vector, $\mathbf{x}$. One thing that can be done would be to initialize the first value of $\mathbf{y}$ to the initial product. That is, write a code that looks like the following.

```
double matvecmlt(double a[][], double x[], int m, int n)
{
  double output[m];
  //
  // loop over the rows of the matrix
  // -------------------------------
  //
  for(int i=0 i<m; i++) {
    //
    // iniitalize the component of the output vector, y
    // ------------------------------------------------
    //
    output[i] = a[i][0] * x[0];
    //
    // loop over the entries in the row each new contribution to the output
    // --------------------------------------------------------------------
    //
    for(int j=1; j<m; j++) {
      output[i] = output[i] + a[i][j] * x[j];
    }
    //
  }
  //
  return output;
  //
}
```

All that this new routine does is to eliminate one extra computation by using the first product in the dot product computing each new component. It might save a little computation time to do this. Now that we have a code written for computing a matrix vector product, you should go back to the section on the transpose of a matrix, A, and review the example where the matrix of a transpose was needed to compute a matrix vector product.

Vector/Matrix Operations: Matrix-Matrix Multiplication

The ideas in the matrix-vector multiplication can be used easily to define matrix-matrix multiplication. Suppose $A \in \Re^{m\times n}$ and $B \in \Re^{n\times r}$ are two matrices, then the product

$$C = A\ B$$

where $C \in \Re^{m\times r}$ is determined by the following formula.

$$C_{i,j} = \sum_{k=1}^{n} a_{i,k}\ b_{k,j}$$

This is a formula that is covered in most linear algebra courses. The product formula involves the computation of $m \times r$ dot products. As in the matrix-vector multiplication the formula for $C_{i,j}$ can be used to build a computer code to do this work.

```
double matmatmlt(double a[][], double b[][], int m, int n, int r)
{
  double output[m][r];
  //
  // loop over the rows of the matrix
  // --------------------------------
  //
  for(int i=0 i<m; i++) {
    for(int j=0 j<r; j++) {
      //
      // initial this component of the output
      // ------------------------------------
      //
      output[i][j] = 0.0;
      //
      // loop over the entries in the output matrix
      // ------------------------------------------
      //
      for(int k=1; k<n; k++) {
        output[i][j] = output[i][j] + a[i][k] * b[k][j];
      }
    }
    //
  }
  //
  return output;
  //
}
```

There are a number of ways to structure this code. In fact, there are 6 ways to write this code. For this section, the code above will provide the product of two matrices. The next section will reorder the loops in the original, *matmatmlt*, to come up with alternate codes for the same operation that may or may not improve performance.

Vector/Matrix Operations: Some Building Blocks for Computational Linear Algebra - SAXPY

In the world of computational mathematics, one approaches the construction of complicated algorithms by starting out with simple building block algorithms and codes to match. We have already covered a number of these in previous sections. However, there are some algorithms that come standard with packages designed to solve computational linear algebra problems. The examples in this and the next section can be found in just about any document that covers Linpack or LAPACK. In this section, we will consider the SAXPY operation for updating a vector within a loop structure.
The moniker “ SAXPY” stands for **S**calar **A** **X** **P**lus **Y** and is defined by the assignment operation

$$\mathbf{y} \leftarrow a\,\mathbf{x} + \mathbf{y}$$

that can be written component-wise as

$$y_i \leftarrow a\ x_i + y_i$$

for $i = 1, 2, \ldots, n$. This last form gives the inner most statement of a loop for performing the update. So, a code can be constructed using the last formula and might look like the following.

```
double * saxpy(double y[], double x[], double a, int n)
{
  //
  // loop over the rows of the matrix
  // -------------------------------
  //
  for(int i=0; i<n; i++)
    y[i] = a * x[i] + y[i];
  //
  return y;
  //
}
```

This may seem very simple. However, this operation can be used in all kinds of computational linear algorithms and as such this code provides a building block for other computational algorithms to use. Note that this is an update operation for the vector **y**. This means **y** will be overwritten each and every time the routine is called. This is useful in algorithms that involve iterating on a matrix form.

Vector/Matrix Operations: The GAXPY Operation

Another standard operation is a generalization of the SAXPY operation that was discussed in the last section of these notes. A formula that defines this operation is

$$\mathbf{y} = A\,\mathbf{x} + \mathbf{y}$$

So, instead of updating a vector $\mathbf{y}$ with a scalar multiple of another vector, $a\mathbf{x}$, the formula will update a vector with $A\mathbf{x}$, output from a matrix-vector multiplication operation. A component form of the GAXPY operation is the following.

$$y_i = \sum_{j=1}^{n} a_{i,j}\; x_j + y_i$$

Note that this version of the update does the work row by row in the column vector, $\mathbf{y}$. A code to perform a GAXPY updata operation might look like the following.

```
double * gaxpy(double y[], double a[][], double x[], int m, int n)
{
  //
  // loop over the rows of the matrix
  // --------------------------------
  //
  for(int i=0; i<n; i++) {
    for(int j=0; j<n; j++) y[i] = a[i][j] * x[j] + y[i];
  }
  //
  return y;
  //
}
```

Using today's computational hardware, a great deal of emphasis involves the use of parallel computer hardware. This includes multi-core programming, GPU programming, and other parallel and vector computer resources. The code listing above may need some work in order to create a parallel version of the algorithm. An alternative code mught look like

```
double * gaxpy(double y[], double a[][], double x[], int m, int n)
{
  //
  // loop over the rows of the matrix
  // --------------------------------
  //
  for(int i=0; i<n; i++) {
    #pragma omp parallel for
    for(int j=0; j<n; j++) y[i] += a[i][j] * x[j];
  }
  //
  return y;
  //
}
```

There is a minor difference in how the code is written. The increment operator is used and when combined with the OpenMp construct, the code can be compiled with multi-core coding features. The extra line in the code,

```
        #pragma omp parallel for
```

is a compiler directive for OpenMP. The content on how to implement the code using OpenMP (and other parallel programming constructs) will be covered in other places in the notes.

Vector/Matrix Operations: Matrix Outer-Product Update

An outer product update for a matrix involves two vectors, $\mathbf{u} \in \Re^m$, and $\mathbf{v} \in \Re^n$ and computes the update

$$A \leftarrow A + \mathbf{u}\,\mathbf{v}^T$$

The second term in the update creates a matrix of the same size as the matrix A. The result of this operation is a matrix that is stored back into A. Of course, this means that A is overwritten in the update process. Component-wise, we can write

$$A_{i,j} = A_{i,j} + u_i\, v_j$$

A C code for implementing this algorithm is the following.

```
double * matupdate(double a[][], double u[], double v[], int m, int n)
{
  double output[m,n];
  //
  // loop over the rows of the matrix
  // --------------------------------
  //
  for(int i=0 i<m; i++) {
    for(int j=0; j<n; j++) output[i][j] = a[i][k] + u[i] * v[j];
  }
  //
  return output;
  //
}
```

We will use this type of outer-product update for implementing a matrix-matrix update in the next section of these notes.

Vector/Matrix Operations: Matrix Multiplication Update Code

As a last example of how to write codes that are helpful in computational linear algebra problems, we consider a matrix update formula involving the computation of a matrix-matrix product. Given matrices $A \in \Re^{m\times p}$, $B \in \Re^{p\times n}$, and $C \in \Re^{m\times n}$ a useful update formula in computational mathematics is

$$C \;\leftarrow\; A\,B + C$$

This is a recursive formula for updating the matrix C. The number of flops needed to update C is order n^3 for square matrices.
A component-wise formula for the update can be written as

$$C_{i,j} \;= \sum i = 1^k \; A_{i,k}\; B_{k,j} + C_{i,j}$$

for $i = 1, 2, \ldots, m$ and $j = 1, 2, \ldots, n$. A code to perform this operation might look like the following.

```
double * matupdate(double c[][], double a[][], double b[][],
                   int m, int p, int n)
{
  double output[m,n];
  //
  // loop over the rows of the matrix
  // -------------------------------
  //
  for(int i=0 i<m; i++) {
    //
    // loop through the columns in the output matrix
    // ---------------------------------------------
    //
    for(int j=0; j<n; j++) {
      //
      // iniitalize the component of the output vector, y
      // ------------------------------------------------
      //
      for(int k=0; k<p; k++)
                  output[i][j] = a[i][k] * b[k][j] + c[i][j];
    }
  }
  //
  return output;
  //
}
```

This version of the code is called a dot-product implementation of the algorithm. This is due to the inner most of the triple-nested loop in the code is a dot product of a row in A and a column of B. There are actually a total of six different versions of this code. The different versions of the algorithm are created by interchanging the three loops. In exact arithmetic, all six of the code will produce the same result. If the computations are rearranged the output will change slightly due to round-off error. In addition, due to the way computer languages set up storage of arrays, most notably C and Fortran, one of the codes will run faster than another version.
One rearrangement of the loops just interchanges the outer most loops and looks like

```
double * matupdate(double c[][], double a[][], double b[][],
                   int m, int p, int n)
{
  double output[m,n];
  //
  // loop over the rows of the matrix
  // ------------------------------
  //
  for(int i=0 i<m; i++) {
    //
    // loop through the columns in the output matrix
    // ---------------------------------------------
    //
    for(int j=0; j<n; j++) {
      //
      // loop over the inner dimension of the matrix-matrix multipliction
      // ----------------------------------------------------------------
      //
      for(int k=0; k<p; k++)
                  output[i][j] = a[i][k] * b[k][j] + c[i][j];
    }
  }
  //
  return output;
  //
}
```

As we can see, the inner most loop does not change. This gives a second dot-product version of the algorithm due to the fact that the inner most loop is still just a dot product operation. The difference is in the order in which the components of the output matrix are compouted.
Another version of the same code is the following changes the loop structure so that the inner most loop is a matrix outer-product update operation. versions

---

```
double * matupdate(double c[][], double a[][], double b[][],
                   int m, int p, int n)
{
  double output[m,n];
  //
  // loop over the inner dimension of the matrix-matrix multiplication
  // -----------------------------------------------------------------
  //
  for(int k=0 k<p; k++) {
    //
    // loop through the columns in the output matrix
    // ---------------------------------------------
    //
    for(int j=0; j<n; j++) {
      //
      // loop over the row entries
      // ------------------------
      //
```

```
        for(int i=0; i<m; i++)
                    output[i][j] = a[i][k] * b[k][j] + c[i][j];
      }
    }
    //
    return output;
    //
  }
```

We can see that the only change in the code is the order of the loops. As mentioned in Matrix Computations by Golub and Van Loan the difference is in the data flow within the code. How the entries are accessed in the code is the only feature that can change the code. The same number of operations are performed, just ordered differently. There are differences in how different hardware and software access data and thus the different versions may provide different levels of performance.
A final SAXPY version of the matrix update in this section looks like the following.

```
  double * matupdate(double c[][], double a[][], double b[][],
                     int m, int p, int n)
  {
    double output[m,n];
    //
    // loop over the columns of the matrix
    // ----------------------------------
    //
    for(int j=0; j<n; j++) {
      //
      // loop over the inside dimension in the matrix-matrix multiplication
      // ------------------------------------------------------------------
      //
      for(int k=0 k<p; k++) {
        //
        // loop through the rows in the output matrix
        // ------------------------------------------
        //
        for(int i=0; i<m; i++)
                    output[i][j] = a[i][k] * b[k][j] + c[i][j];
      }
    }
    //
    return output;
    //
  }
```

Note that there are two dot-product versions, two SAXPY versions, and two outer-product versions of this algorithm. Any of these six codes can be written to implement the given matrix-matrix multiplication update. However, if performance increases with one of the forms over the others, it should make sense to take advantage of that difference. The coding difference is just swapping three loops in the code.