Appendix A

Mathematical Background

A.1 Normed Linear Spaces

A *linear space* or *vector space* is a nonempty set X endowed with two operations, vector addition $+$ and scalar multiplication $\cdot$, that satisfy

- $x + y = y + x$ for all $x, y \in X$
- $(x + y) + z = x + (y + z)$ for all $x, y, z \in X$
- there is a $\theta \in X$ such that $x + \theta = x$ for all $x \in X$
- for each $x \in X$ there is a $y \in X$ such that $x + y = \theta$
- $(\alpha\beta) \cdot x = \alpha \cdot (\beta \cdot x)$ for all $\alpha, \beta \in \Re$ and $x \in X$
- $\alpha \cdot (x + y) = \alpha \cdot x + \alpha \cdot y$ for all $\alpha \in \Re$ and $x, y \in X$
- $(\alpha + \beta) \cdot x = \alpha \cdot x + \beta \cdot y$ for all $\alpha, \beta \in \Re$ and $x \in X$
- $1 \cdot x = x$ for all $x \in X$.

The elements of X are called vectors.

A normed linear space is a linear space endowed with a real-valued function $|| \cdot ||$ on X, called a norm, which measures the size of vectors. By definition, a norm must satisfy

- $||x|| \geq 0$ for all $x \in X$;
- $||x|| = 0$ if and only if $x = \theta$;

- $||\alpha \cdot x|| = |\alpha| \ ||x||$ for all $\alpha \in \Re$ and $x \in X$;
- $||x + y|| \leq ||x|| + ||y||$ for all $x, y \in X$.

Every norm on a linear space induces a metric that measures the distance $d(x, y)$ between arbitrary vectors x and y. The induced metric is defined via the relation $d(x, y) = ||x - y||$. It meets all the conditions we normally expect a distance function to satisfy:

- $d(x, y) = d(y, x) \geq 0$ for all $x, y \in X$;
- $d(x, y) = 0$ if and only if $x = y \in X$;
- $d(x, y) \leq d(x, z) + d(z, y)$ for all $x, y, z \in X$.

Norms and metrics play a critical role in numerical analysis. In many numerical applications, we do not solve a model exactly, but rather compute an approximation via some iterative scheme. The iterative scheme is usually terminated when the change in successive iterates becomes acceptably small, as measured by the norm of the change. The accuracy of the approximation or approximation error is measured by the metric distance between the final approximant and the true solution. Of course, in all meaningful applications, the distance between the approximant and true solution is unknown because the true solution is unknown. However, in many theoretical and practical applications, it is possible to compute upper bounds on the approximation error, thus giving a level of confidence in the approximation.

In this book we will work almost exclusively with three classes of normed linear spaces. The first normed linear space is the familiar $\Re^n$, the space of all real n-vectors. The second normed linear space is $\Re^{m \times n}$, the space of all real m-by-n matrices. We will use a variety of norms for real vector and matrix spaces, all of which are discussed in greater detail in the following section.

A second class of normed linear space is $C(S)$, the space of all bounded continuous real-valued functions defined on $S \subset \Re^m$. Addition and scalar multiplication in this space are defined pointwise. Specifically, if $f, g \in C(S)$ and $\alpha \in \Re$, then $f + g$ is the function whose value at $x \in S$ is $f(x) + g(x)$ and αf is the function whose value at $x \in S$ is $\alpha f(x)$. We will use only one norm, called the sup or supremum norm, on the function space $C(S)$:

$$||f|| = sup\{|f(x)| \mid x \in S\}.$$

In most applications, S will be a bounded interval of $\Re^n$.

A subset Y of a normed linear space X is called a subspace if it is closed under addition and scalar multiplication, and thus is a normed linear space in its own right. More specifically, Y is a subspace of X if $x + y \in Y$ and $\alpha x \in Y$ whenever $x, y \in Y$ and $\alpha \in \Re$. A subspace Y is said to be dense in X if for any $x \in X$ and $\epsilon > 0$, we can always find a $y \in Y$ such that $||x - y|| < \epsilon$. Dense linear subspaces play an important role in numerical analysis. When constructing approximants for elements in a normed linear space X, drawing our approximants from a dense linear subspace guarantees that an arbitrarily accurate approximation can always be found, at least in theory.

Given a nonempty subset S of X, $span(S)$ is the set of all finite linear combinations of elements of S:

$$span(S) = \{\sum_{i=1}^{n} \alpha_i x_i | \alpha_i \in \Re, x_i \in X, n \text{ an integer}\}.$$

We say that a subset B is a basis for a subspace Y if $Y = span(B)$ and if no proper subset of B has this property. A basis has the property that no element of the basis can be written as a linear combination of the other elements in the basis. That is, the elements of the basis are linearly independent.

Except for the trivial subspace $\{\theta\}$, a subspace Y will generally have many distinct bases. However, if Y has a basis with a finite number of elements, then all bases have the same number of nonzero elements and this number is called the dimension of the subspace. If the subspace has no finite basis, it is said to be infinite dimensional.

Consider some examples. Every normed linear space X, has two trivial subspaces: $\{\theta\}$, whose dimension is zero, and X. The sets $\{(0,1)', (1,0)'\}$ and $\{(2,1)', (3,4)'\}$ both are bases for $\Re^2$, which is a two-dimensional space; the set $\{(\alpha, 0.5 \cdot \alpha)' | \alpha \in \Re\}$ is a one-dimensional subspace of $\Re^2$. In general, $\Re^n$ is an n-dimensional space with many possible bases; moreover, the span of any $k < n$ linearly independent n-vectors constitutes a proper k-dimensional subspace of $\Re^n$.

The function space $C(S)$ of all real-valued bounded continuous functions on an interval $S \subset \Re$ is an infinite-dimensional space. This space has a number of subspaces that are important in numerical analysis. The set of all polynomials on S of degree at most n forms an $n + 1$ dimensional subspace of $C(S)$ with one basis being $\{1, x, x^2, \ldots, x^n\}$. The set of all polynomials, regardless of degree, is also a subspace of $C(S)$. It is infinite-dimensional.

Other subspaces of $C(S)$ interest include the space of piecewise polynomials splines of a given order. These subspaces are finite-dimensional and are discussed further in the text.

A sequence $\{x_k\}$ in a normed linear space X converges to a limit x^* in X if $\lim_{k\longrightarrow\infty} ||x_k - x^*|| = 0$. We write $\lim_{k\longrightarrow\infty} x_k = x^*$ to indicate that the sequence $\{x_k\}$ converges to x^*. If a sequence converges, its limit is necessarily unique.

An open ball centered at $x \in X$ is a set of the form $\{y \in X \mid ||x-y|| < \epsilon\}$, where $\epsilon > 0$. A set S in X is open if every element of S is the center of some open ball contained entirely in S. A set S in X is closed if its complement, that is, the set of elements of X not contained in S, is an open set. Equivalently, a set S is closed if it contains the limit of every convergent sequence in S.

The Contraction Mapping Theorem has many uses in computational economics, particularly in existence and convergence theorems: Suppose that X is a complete normed linear space, that T maps a nonempty set $S \subset X$ into itself, and that, for some $\delta < 1$,

$$||T(x) - T(y)|| \leq \delta||x - y||, \text{ for all } x, y \in S.$$

Then, there is an unique $x^* \in S$ such that $T(x^*) = x^*$. Moreover, if $x_0 \in S$ and $x_{k+1} = T(x_k)$, then $\{x_k\}$ necessarily converges to x^* and

$$||x_k - x^*|| \leq \frac{\delta}{1-\delta}||x_k - x_{k-1}||.$$

When the above conditions hold, T is said to be a strong contraction on S and x^* is said to be a fixed-point of T in S.

We shall not define what we mean by a complete normed linear space, save to note that $\Re^n$, $C(S)$, and all their subspaces are complete.

A.2 Matrix Algebra

We write $x \in \Re^n$ to denote that x is an n-vector whose i^{th} entry is x_i. A vector is understood to be in column form unless otherwise noted.

If x and y are n-vectors, then their sum $z = x + y$ is the n-vector whose i^{th} entry is $z_i = x_i + y_i$. Their inner product or dot product, $x \star y$, is the real number $\sum_i x_i \cdot y_i$. And their array product, $z = x. \star y$, is the n-vector whose i^{th} entry is $z_i = x_i \cdot y_i$.

If α is a scalar, that is, a real number, and x is an n-vector, then their scalar sum $z = \alpha + x = x + \alpha$ is the n vector whose i^{th} entry is $z_i = \alpha + x_i$. Their scalar product, $z = \alpha \star x = x \star \alpha$, is the n-vector whose i^{th} entry is $z_i = \alpha \cdot x_i$.

The most useful vector norms are, respectively, the 1-norm or sum norm, the 2-norm or Euclidean norm, and the infinity or max norm:

$$\begin{aligned} ||x||_1 &= \textstyle\sum_i |x_i|, \\ ||x||_2 &= \sqrt{\textstyle\sum_i |x_i|^2}, \\ ||x||_\infty &= \max\{|x_1|, |x_2|, \ldots, |x_n|\}. \end{aligned}$$

In Matlab, the norms may be computed for any vector x, respectively, by writing: `norm(x,1)`, `norm(x,2)`, and `norm(x,inf)`. If we simply write `norm(x)`, the 2-norm or Euclidean norm is computed.

All norms on $\Re^n$ are equivalent in the sense that a sequence converges in one vector norm, if and only if it converges in all other vector norms. This is not true of generally of all normed linear spaces.

A sequence of vectors $\{x_k\}$ converges to x^* at a rate of order $p \geq 1$ if for some $c \geq 0$ and for sufficiently large n,

$$||x_{k+1} - x^*|| \leq c||x_k - x^*||^p.$$

If $p = 1$ and $c < 1$ we say the convergence is linear; if $p > 1$ we say the convergence is superlinear; and if $p = 2$ we say the convergence is quadratic.

We write $A \in \Re^{m \times n}$ to denote that A is an m-row by n-column matrix whose row i, column j entry, or, more succinctly, ij^{th} entry, is A_{ij}.

If A is an m by n matrix and B is an m by n matrix, then their sum $C = A + B$ is the m by n matrix whose ij^{th} entry is $C_{ij} = A_{ij} + B_{ij}$. If A is an m by p matrix and B is a p by n matrix, then their product $C = A \star B$ is the m by n matrix whose ij^{th} entry is $C_{ij} = \sum_{k=1}^{p} A_{ik}B_{kj}$. If A and B are both m by n matrices, then their array product $C = A. \star B$ is the m by n matrix whose ij^{th} entry is $C_{ij} = A_{ij}B_{ij}$.

A matrix A is square if it has an equal number of rows and columns. A square matrix is upper triangular if $A_{ij} = 0$ for $i > j$; it is lower triangular if $A_{ij} = 0$ for $i < j$; it is diagonal if $A_{ij} = 0$ for $i \neq j$; and it is tridiagonal if $A_{ij} = 0$ for $|i - j| > 1$. The identity matrix, denoted I, is a diagonal matrix whose diagonal entries are all 1. In Matlab, the identity matrix of order n may is generated by the statement `eye(n)`.

The transpose of an m by n matrix A, denoted A', is the n by m matrix whose ij^{th} entry is the ji^{th} entry of A. A square matrix is symmetric if

$A = A'$, that is, if $A_{ij} = A_{ji}$ for all i and j. A square matrix A is orthogonal if $A' \star A = A \star A'$ is diagonal, and orthonormal if $A' \star A = A \star A' = I$. In Matlab, the transpose of a matrix A is generated by the statement A'.

A square matrix A is invertible if there exists a matrix A^{-1}, called the inverse of A, such that $A \star A^{-1} = A^{-1} \star A = I$. If the inverse exists, it is unique. In Matlab, the inverse of a square matrix A can be generated by the statement `inv(A)`.

The most useful matrix norms, and the only ones used in this book, are constructed from vector norms. A given n-vector norm $|| \cdot ||$ induces a corresponding matrix norm for n by n matrices via the relation

$$||A|| = max_{||x||=1}||A \star x||$$

or, equivalently,

$$||A|| = max_{||x|| \neq 0} \frac{||A \star x||}{||x||}.$$

Given corresponding vector and matrix norms,

$$||A \star x|| \leq ||A|| \cdot ||x||.$$

Moreover, if A and B are square matrices,

$$||A \star B|| \leq ||A|| \cdot ||B||.$$

Common matrix norms include the matrix norms induced by the sup, sum, and Euclidean vector norms:

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

for $p = 1, 2, \infty$. In Matlab, these norms may be computed for any matrix A, respectively, by writing: `norm(A,1)`, `norm(A,2)`, and `norm(A,inf)`. The Euclidean matrix norm is relatively expensive to compute. The sum and max norms, on the other hand, take a relatively simple form:

$$\begin{aligned} ||A||_1 &= max_{1 \leq j \leq n} \sum_{i=1}^{n} |A_{ij}| \\ ||A||_\infty &= max_{1 \leq i \leq n} \sum_{j=1}^{n} |A_{ij}|. \end{aligned}$$

The spectral radius of a square matrix A, denoted $\rho(A)$, is the infimum of all the matrix norms of A. We have $\lim_{k=1}^{\infty} A^k = 0$ if and only if $\rho(A) < 1$, in which case $(I - A)^{-1} = \sum_{k=1}^{\infty} A^k$. Thus, if $||A|| < 1$ in any vector norm, A^k converges to zero.

A square symmetric matrix A is negative semidefinite if $x' \star A \star x \leq 0$ for all x; it is negative definite if $x' \star A \star x < 0$ for all $x \neq 0$; it is positive semidefinite if $x' \star A \star x \geq 0$ for all x; and it is positive definite if $x' \star A \star x > 0$ for all $x \neq 0$.

A.3 Real Analysis

The gradient or Jacobian of a vector-valued function $f : \Re^n \mapsto \Re^m$ is the m by n matrix-valued function of first partial derivatives of f. More specifically, the gradient of f at x, denoted by either $f'(x)$ or $f_x(x)$, is the m by n matrix whose ij^{th} entry is the partial derivative $\frac{\partial f_i}{\partial x_j}(x)$. More generally, if $f(x_1, x_2)$ is an n-vector-valued function defined for $x_1 \in \Re^{n_1}$ and $x_2 \in \Re^{n_2}$, then $f_{x_1}(x)$ is the m by n_1 matrix of partial derivatives of f with respect to x_1 and $f_{x_2}(x)$ is the m by n_2 matrix of partial derivatives of f with respect to x_2.

The Hessian of the real-valued function $f : \Re^n \mapsto \Re$ is the n by n matrix-valued function of second partial derivatives of f. More specifically, the Hessian of f at x, denoted by either $f''(x)$ or $f_{xx}(x)$, is the symmetric n by n matrix whose ij^{th} entry is $\frac{\partial^2 f}{\partial x_i \partial x_j}(x)$. More generally, if $f(x_1, x_2)$ is a real-valued function defined for $x_1 \in \Re^{n_1}$ and $x_2 \in \Re^{n_2}$, where $n_1 + n_2 = n$, then $f_{x_i x_j}(x)$ is the n_i by n_j submatrix of $f''(x)$ obtained by extracting the rows corresponding to the elements of x_i and the columns corresponding to the columns of x_j.

A real-valued function $f : \Re^n \mapsto \Re$ is smooth on a convex open set S if its gradient and Hessian are defined and continuous on S. By Taylor's theorem, a smooth function may be approximated locally by either a linear or quadratic function. More specifically, for all x in S,

$$f(x) = f(x_0) + f_x(x_0) \star (x - x_0) + o(||x - x_0||)$$

and

$$\begin{aligned} f(x) &= f(x_0) + f_x(x_0) \star (x - x_0) \\ &\quad + \tfrac{1}{2}(x - x_0)' \star f_{xx}(x_0) \star (x - x_0) + o(||x - x_0||^2) \end{aligned}$$

where $o(t)$ denotes a term with the property that $lim_{t \longrightarrow 0}(o(t)/t) = 0$.

The Intermediate Value Theorem asserts that if a continuous real-valued function attains two values, then it must attain all values in between. More precisely, if f continuous on a convex set $S \in \Re^n$ and $f(x_1) \leq y \leq f(x_2)$ for some $x_1 \in S$, $x_2 \in S$, and $y \in \Re$, then $f(x) = y$ for some $x \in S$.

The Implicit Function Theorem gives conditions under which a system of nonlinear equations will have a locally unique solution that will vary continuously with some parameter: Suppose $F : \Re^{m+n} \mapsto \Re^n$ is continuously differentiable in a neighborhood of (x_0, y_0), $x_0 \in \Re^m$ and $y_0 \in \Re^n$, and that $F(x_0, y_0) = 0$. If $F_y(x_0, y_0)$ is nonsingular, then there is an unique function $f : \Re^m \mapsto \Re^n$ defined on a neighborhood N of x_0 such that for all $x \in N$, $F(x, f(x)) = 0$. Furthermore, the function f is continuously differentiable on N and $f'(x) = -F_y^{-1}(x, f(x)) \star F_x(x, f(x))$.

A subset S is bounded if it is contained entirely inside some ball centered at zero. A subset S is compact if it is both closed and bounded. A continuous real-valued function defined on a compact set has well-defined maximum and minimum values; moreover, there will be points in S at which the function attains its maximum and minimum values.

A real-valued function $f : \Re^n \mapsto \Re$ is concave on a convex set S if $\alpha_1 f(x_1) + \alpha_2 f(x_2) \leq f(\alpha_1 x_1 + \alpha_2 x_2)$ for all $x_1, x_2 \in S$ and $\alpha_1, \alpha_2 \geq 0$ with $\alpha_1 + \alpha_2 = 1$. It is strictly concave if the inequality is always strict. A smooth function is concave (strictly concave) if and only if $f''(x)$ is negative semidefinite (negative definite) for all $x \in S$. A smooth function f is convex if and only $-f$ is concave. If a function is concave (convex) on an convex set, then its maximum (minimum), if it exists, is unique.

A.4 Markov Chains

A Markov process is a sequence of random variables $\{X_t \mid t = 0, 1, 2, \ldots\}$ with common state space S whose distributions satisfy

$$\Pr\{X_{t+1} \in A \mid X_t, X_{t-1}, X_{t-2}, \ldots\} = \Pr\{X_{t+1} \in A \mid X_t\} \quad A \subset S.$$

A Markov process is often said to be memoryless because the distribution X_{t+1} conditional on the history of the process through time t is completely determined by X_t and is independent of the realizations of the process prior to time t.

A Markov chain is a Markov process with a finite state-space $S = \{1, 2, 3, \ldots, n\}$. A Markov chain is completely characterized by its transition probabilities

$$P_{tij} = \Pr\{X_{t+1} = j \mid X_t = i\}, \qquad i, j \in S.$$

A Markov chain is stationary if its transition probabilities

$$P_{ij} = \Pr\{X_{t+1} = j \mid X_t = i\}, \qquad i, j \in S$$

are independent of t. The matrix P, called the transition probability matrix.

The steady-state distribution of a stationary Markov chain is a probability distribution $\{\pi_i | i = 1, 2, \ldots, n\}$ on S, such that

$$\pi_j = \lim_{\tau \to \infty} \Pr\{X_\tau = j \mid X_t = i\} \qquad i, j \in S.$$

The steady-state distribution π, if it exists, completely characterizes the longrun behavior of a stationary Markov chain.

A stationary Markov chain is irreducible if for any $i, j \in S$ there is some $k \geq 1$ such that $\Pr\{X_{t+k} = j \mid X_t = i\} > 0$, that is, if starting from any state there is positive probability of eventually visiting every other state. Given an irreducible Markov chain with transition probability matrix P, if there is an n-vector $\pi \geq 0$ such that

$$P' \star \pi = \pi$$
$$\textstyle\sum_i \pi_i = 1,$$

then the Markov chain has a steady-state distribution π.

In computational economic applications, one often encounters irreducible Markov chains. To compute the steady-state distribution of the Markov chain, one solves the $n + 1$ by n linear equation system

$$\begin{bmatrix} I - P' \\ i' \end{bmatrix} \star \pi = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

where P is the probability transition matrix and i is the vector consisting of all ones. Due to linear dependency among the probabilities, any one of the first n linear equations is redundant and may be dropped to obtain an uniquely soluble matrix linear equation.

Consider a stationary Markov chain with transition probability matrix

$$P = \begin{bmatrix} 0.5 & 0.2 & 0.3 \\ 0.0 & 0.4 & 0.6 \\ 0.5 & 0.5 & 0.0 \end{bmatrix}$$

Although one cannot reach state 1 from state 2 in one step, one can reach it with positive probability in two steps. Similarly, although one cannot return to state 3 in one step, one can return in two steps. The steady-state distribution π of the Markov chain may be computed by solving the linear equation

$$\begin{bmatrix} 0.5 & 0.0 & -0.5 \\ -0.2 & 0.6 & -0.5 \\ 1.0 & 1.0 & 1.0 \end{bmatrix} \star \pi = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}.$$

The solution is

$$\pi = \begin{bmatrix} 0.316 \\ 0.368 \\ 0.316 \end{bmatrix}.$$

Thus, over the long run, the Markov process will spend about 32.6 percent of its time in state 1, 36.8 percent of its time in state 2, and 31.6 percent of its time in state 3.

Appendix B

Computer Programming

B.1 Computer Arithmetic

Some knowledge of how computers perform numerical computations and how programming languages work is useful in applied numerical work, especially if one is to write efficient programs. It often comes as an unpleasant surprise to many people to learn that exact arithmetic and computer arithmetic do not always give the same answers, even in programs without programming errors.

For example, consider the following two statements

$$x = (1e - 20 + 1) - 1$$

and

$$x = 1e - 20 + (1 - 1).$$

Here, 1e-20 is computer shorthand for 10^{-20}. Mathematically the two statements are equivalent because addition and subtraction are associative. A computer, however, would evaluate these statements differently. The first statement would likely result in $x = 0$, whereas the second would result in $x = 1e - 20$. The reason has to do with how computers represent numbers.

Typically, computer languages such as Fortran and C allow several ways of representing a number. Matlab makes things simple by only have one representation for a number. Matlab uses what is often called a double precision floating point number. The exact details of the representation depends on the hardware but there are several features in common. First, the representation has three parts, a sign bit, an exponent, a mantissa. Consider the

number -3210.4. This can be equivalently written as -3.2104×10^3. The mantissa is 3.2104, the exponent is 3, and the sign bit is 1.

The computer has only a predefined set of storage elements (bytes) for a number. On most personal computers a number has 8 bytes. If the mantissa is very long it gets truncated by rounding or chopping, depending on the hardware. For example, suppose only 5 places are allocated for the mantissa. A number like -3210.48 might be represented as -3.2104×10^3, that is, the lowest digit may be chopped off.

In our original example, when the computer processes $x = (1e-20+1)-1$ it first adds 1 to 1e-20, which is the number 1.00000000000000000001. Unfortunately, most computers cannot handle this long a mantissa and truncate the result to 1. The computer then subtracts 1 from the first sum, which results in 0. On the other hand, with the statement $x = 1e-20+(1-1)$, the subtraction in parenthesis occurs first, resulting in 0, which is then added to $1e-20$.

To understand more fully how numbers are stored in a computer, let us examine a few numbers in their so-called hexadecimal form. Hexadecimal numbers are numbers expressed in base 16; this is a useful base for computer arithmetic because it is a power of base 2, which is the form in which numbers are ultimately stored in a computer. Hexadecimal numbers use the usual digits 0 through 9 and supplement them with the letters a through f; a=10, b=11,..., f=15. An 8-byte floating point number (i.e., "double-precision") looks something like:

$$3ff1000000000000.$$

The hexadecimal representation makes clear some of the problems that arise in floating point arithmetic. Suppose one compared the values derived by the following expressions

$$1/3 + 1/2$$

and

$$5/6.$$

The first operation results in

$$3feaaaaaaaaaaaaa$$

whereas the second results in

$$3feaaaaaaaaaaab.$$

We know that these operations should result in the same number but the computer represents them in a way that differs by a single bit in the lowest order byte. Although this may not seem like a big deal, if one were to test the expression

$$1/3 + 1/2 = 5/6,$$

the expression would be deemed false.

Similar problems arise in other case as well. For example,

$$7^{-20} = 3c6ce5e856164656$$

whereas

$$7^{-19}/7 = 3c6ce5e856164655,$$

even though in exact arithmetic these two quantities are theoretically the same.

Reversing a mathematical operation sometimes does not work either. In general, one should be careful when a number is raised to a large power and then to a very small power or vice versa. For example,

$$(1.1\hat{}(10e - 12))\hat{}(10e + 12)$$

should result in 1.1. However, on many computers the operation will result in 1.09941652517756.

Roundoff error is only one of the pitfalls of computer programming. In numerical computations, error is also introduced by the computer's inherent inability to evaluate certain mathematical expressions exactly. For all its power, a computer can only perform a limited set of arithmetic operations directly. Essentially this list includes the four arithmetic operations of addition, subtraction, multiplication and division, as well as logical operations of comparison. Other common functions, such as exponential, logarithmic, and trigonometric functions cannot be evaluated directly using computer arithmetic. They can only be evaluated approximately using algorithms based on the four basic arithmetic operations.

For the common functions very efficient algorithms typically exist and these are sometimes "hardwired" into the computer's processor or coprocessor. An important area of numerical analysis involves determining efficient approximations that can be computed using basic arithmetic operations. For example, the exponential function has the series representation

$$\exp(x) = \sum_{i=0}^{\infty} x^n/n!.$$

Obviously one cannot compute the infinite sum, but one could compute a finite number of these terms, with the hope that one will obtain sufficient accuracy for the purpose at hand. The result, however, will always be inexact.

B.2 Data Storage

Matlab's basic data type is the matrix, with a scalar just a 1 by 1 matrix and an n-vector an n by 1 or 1 by n matrix. Actually, the basic data type in Matlab also contains additional information that is stored along with the matrix itself. In particular, Matlab attaches the row and column information about the matrix. This is a significant advantage over writing in low level language like Fortran or C because it relieves one of the necessity of keeping track of array size and memory allocation.

When one wants to represent an m by n matrix of numbers in a computer there are a number of ways to do this. The most simple way is to store all the elements sequentially in memory, starting with the one indexed (1,1) and working down successive columns or across successive rows until the (m,n)th element is stored. Different languages make different choices about how to store a matrix. Fortran stores matrices in column order, whereas C stores in row order. Matlab, although written in C, stores in column order, thereby conforming with the Fortran standard.

Many matrices encountered in practice are sparse, meaning that they consist mostly of zero entries. Clearly, it is a waste of memory to store all of the zeros, and it is time consuming to process the zeros in arithmetic matrix operations. Matlab allows one to store a sparse matrix efficiently by keeping track of only the non-zero elements of the original matrix and their location. In this storage scheme, the row indices and non-zero entries are stored in a two-column vector. A separate vector is used to keep track of where the first element in each column is located. If one wants to access element (i, j),

Matlab check the jth element of the column indicator vector to find where the jth column starts and then searches the row column for the ith element (if one is not found then the element must be zero).

Although sparse matrix representations are useful, their use incurs a cost. To access element (i, j) of a full matrix, one simply goes to element (i-1)*m+j storage location the. To access an element in a sparse matrix involves a search over row indices and hence can take longer. This additional overhead can add up significantly and actually slow down a computational procedure.

A further consideration in using sparse matrices concerns memory allocation. If a procedure repeatedly alters the contents of a sparse matrix, the memory needed to store the matrix may change, even if its dimension does not. This means that more memory may needed each time the number of non-zero elements increases. This memory allocation is both time consuming and may eventually exhaust computer memory. This problem does not arise with full matrices because mn elements are stored in fixed locations from the beginning.

The decision whether to use a sparse or full matrix representation depends on a balance between a number of factors. Clearly for very sparse matrices (less than 10% non-zero) one is better off using sparse matrices and anything over 67% non-zeros one is better off with full matrices (which actually require less storage space at that point). In between, some experimentation may be required to determine which is better for a given application.

B.3 Programming Style

In general there are different ways to write a program that produce the same end results. Algorithmic efficiency refers to the execution time and memory used to get the job done. In many cases, especially in a matrix processing language like Matlab, there are important trade-offs between execution time and memory use. Often, however, the trade-offs are trivial and there so one way of writing the code may be unambiguously better than another.

In Matlab, the rule of thumb is to avoid loops where possible. Matlab is a hybrid language that is both interpreted and complied. A loop executed by the interpreter is generally slower than direct vector operations that are implemented in compiler code. For example, suppose one had a scalar x that one wanted to multiply by the integers from 1 to n to create a vector y whose i^{th} entry is $y_i = x^i$. Both of the following code segments produce the desired

result:

```
for k = 1 : n
    y(i) = x^i;
end
```

and

```
y = x.^(1 : n);
```

The second way avoids the looping of the first and hence executes substantially faster.

Programmer development effort is another critical resource required in program construction that is sometimes ignored in discussions of efficiency. One reason for using high level language such as Matlab, rather than a low level language such as Fortran, is that programming time is often greatly reduced. Matlab carries out many of the housekeeping tasks that the programmer must deal with in lower level languages. Even in Matlab, however, one should consider carefully how important it is to write very efficient code. If the code will be used infrequently, less effort should be devoted to making the code computationally efficient than if the code will be used often or repeatedly.

Furthermore, computationally efficient code can sometimes be fairly difficult to read. If one plans to revise the code at a later date or if someone else is going to use it, it may be better to approach the problem in a simpler way that is more transparent, though possibly slower. The proper balance of computational efficiency versus clarity and development effort is a judgment call. A good idea, however, is embodied in the saying "Get it to run right, then get it to run fast." In other words, get one's code to do what one what it to do first, then look for ways to improve its efficiency.

It is especially important to document one's code. It does not take long for even an experienced programmer to forget what a piece of code does if it is undocumented. We suggest that one get in the habit of writing headers that explain clearly what the code in a file does. If it is a function, the header should contain details on the input and output arguments and on the algorithm used (as appropriate), including references. Within the code it is a good idea to sprinkle reminders about what the code is doing at that point.

Another good programming practice is modularity. Functions that perform a simple well defined task that is to be repeated often should be written

separately and called from other functions as needed. The simple functions can be debugged and then depended on to perform their job in a variety of applications. This not only saves program development time, but makes the resulting code far easier to understand. Also, if one decides that there is a better way to write such a function, one need only make the changes in one place. An example of this principle is a function that computes the derivatives of a function numerically. Such a function will be used extensively in this book.