

There are several ways to define a matrix. Formally, a matrix represents any linear function whose domain and range are subsets of (finite) vector spaces. More practically, a matrix is an $m \times n$ array:

$$\mathbf{A} = \begin{bmatrix} \mathbf{a}_{1,1} & \mathbf{a}_{1,2} & \cdots & \mathbf{a}_{1,n} \\ \mathbf{a}_{2,1} & \mathbf{a}_{2,2} & \cdots & \mathbf{a}_{1,n} \\ \cdots & \cdots & \cdots & \cdots \\ \mathbf{a}_{m,1} & \mathbf{a}_{m,2} & \cdots & \mathbf{a}_{m,n} \end{bmatrix}.$$

(A.1)

Of course, it is possible to think of matrices as objects in their own right. On the other hand, it is useful to keep in mind that they act as linear functions. To see how a matrix can represent a linear function, it helps to understand what a linear function is. By definition, a linear function, \mathbf{A} , is a function that preserves addition and multiplication by scalars; that is:

$$\mathbf{A}(a\mathbf{x} + b\mathbf{y}) = a\mathbf{A}(\mathbf{x}) + b\mathbf{A}(\mathbf{y})$$

for all scalars a and b , and vectors \mathbf{x} and \mathbf{y} . Linear functions include the identity operator, mul-

#1

p.5

tiplication by a fixed scalar, and inner product with a fixed element.

Note that if \mathbf{V} and \mathbf{W} are vector spaces and \mathbf{V} is finite dimensional we can always construct a linear function $\mathbf{V} \rightarrow \mathbf{W}$ with prescribed values at the basis elements of \mathbf{V} . Further, if $\mathbf{v}_1 \dots \mathbf{v}_n$ is a basis set for the n -dimensional \mathbf{V} and $\mathbf{y}_1 \dots \mathbf{y}_n$ are n arbitrary elements in \mathbf{W} , then there is one and only one function, \mathbf{A} , such that $\mathbf{A}(\mathbf{v}_k) = \mathbf{y}_k$. In particular, \mathbf{A} maps an arbitrary element, $\mathbf{x} = \sum x_i \mathbf{v}_i$, to \mathbf{W} by $\mathbf{A}(\mathbf{x}) = \sum x_i \mathbf{y}_i$.²

#2

p.5

A.5 Eigenvectors and Eigenvalues

Let \mathbf{V} be a vector space, \mathbf{S} a subspace of \mathbf{V} , and \mathbf{A} a linear function (i.e. matrix) of \mathbf{S} onto \mathbf{V} . A scalar, λ , is called an *eigenvalue* of \mathbf{A} if there is a nonzero element \mathbf{x} in \mathbf{S} such that:

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

(A.5)

Further, \mathbf{x} is considered to be the *eigenvector* of \mathbf{A} corresponding to λ . There is exactly one eigenvalue corresponding to a given eigenvector (note that this does not mean that each eigenvector has an eigenvalue that is different from every other eigenvector).

Eigenvalues are not necessarily real numbers; they may be complex. When all the

#3

p.8

tive semi-definite.

Eigenvalues and eigenvectors are useful objects; however, they are not always easily interpretable beyond their mathematical definitions. There are some cases where they are. To illustrate such a case, let us introduce the *singular value decomposition* (SVD) of a matrix. Briefly, the SVD of a matrix, \mathbf{A} , is:

$$\mathbf{A} = \mathbf{U}\mathbf{S}\mathbf{V}^T$$

where \mathbf{U} contains orthonormal vectors, \mathbf{V} contains orthonormal vectors and \mathbf{S} is diagonal. All matrices have a singular value decomposition.

#4

p.8

the purposes of this discussion, n is the number of total elements in a matrix.

Assuming a normal model with elements stored in the obvious way (as a contiguous block of memory), element access is a constant operation, $O(1)$. Operations like addition and subtraction take linear time, $O(n)$. Multiplication by a scalar is also linear. Calculating the transpose of a matrix is also linear.

Matrix-Matrix Multiplication is polynomial, roughly $O(n^3)$, assuming that the two matrices involved are of roughly the same size³. Inversion is no harder than multiplication and requires the same time. Numerically stable techniques for finding inverses are often preferred to more straightforward algorithms, but techniques like SVD still take roughly $O(n^3)$ time.

Important to an understanding of computational issues in linear algebra is realizing that such issues arise as much because of system and engineering realities as they do because of theoretical performance bounds on algorithms. For example, there are several reasonable

#5

p.9

work is the sparse matrix, so we will begin there.

Recall that sparse matrices have many zero entries. Therefore, it seems reasonable to create a data structure that represents only the non-zero elements. A common representation—and one used in this work—is to represent a sparse matrix as a pair, $\langle \mathbf{I}, \mathbf{D} \rangle$ where \mathbf{I} contains an ordered list of the indices of all the non-empty entries and \mathbf{D} contains the corresponding values. If the number of non-zero elements is m and $m \ll n$, then this representation may require many fewer orders of magnitude than a standard representation to store. For example, the AP collection used in this work is roughly described by a 280,000 x 280,000 matrix. This translates into roughly 292 gigabytes of storage (assuming four byte numbers). The sparse representation requires only about 334 megabytes.

This savings does have a cost, of course. Arbitrary element access, for example, now

#6

p.10

sparse-sparse operations should probably yield sparse objects. Therefore, to maintain a performance edge, it is often necessary to be very careful in how operations are ordered. For example, a series of additions like $\mathbf{A} + \mathbf{B} + \mathbf{C} + \mathbf{D}$ where \mathbf{A} is dense but \mathbf{B} , \mathbf{C} , and \mathbf{D} are sparse should be executed from right to left, rather than left to right. Another common case in this work involves subtracting the mean column (or row) from a matrix and then multiplying that result by another matrix; that is, (with appropriate abuse of notation) $\mathbf{A}(\mathbf{B} - \boldsymbol{\mu})$. The vector $\boldsymbol{\mu}$ is dense, so the obvious order of operations would require a dense-sparse subtraction and then a dense-dense multiplication. In many cases, it would be far faster to perform the *three* operations of a dense-sparse matrix multiplication, a dense-dense matrix-vector multiplication, and a dense-dense matrix subtraction, as in $\mathbf{AB} - \mathbf{A}\boldsymbol{\mu}$.

Sparse matrices are especially important in this work. Still, other special-purpose representations and algorithms for matrices such as diagonal matrices and symmetric matrices

#7

p.11

umn, this requires the cooperation of (and thus communication overhead for) all processors.

Consider the matrix multiplication operation, \mathbf{AB} . Depending upon the algorithm chosen to compute the product, communication cost is minimized when \mathbf{A} is distributed by rows and \mathbf{B} by columns. Is it worth the initial cost of redistributing the matrices before performing the computation? Although it can require shuffling hundreds of megabytes across processors and machines, it is often worth it to avoid the communication cost of computing a misdistributed \mathbf{AB} (not to mention the savings in programming effort).

Note that this only hints at the added complexity of normally simple operations, like

#8

p.12

Appendix A

A Linear Algebra and Eigenproblems

A working knowledge of linear algebra is key to understanding many of the issues raised in this work. In particular, many of the discussions of the details latent semantic indexing and singular value decomposition assume an understanding of eigenproblems. While a comprehensive discussion of linear algebra is well beyond the scope of this paper, it seems worthwhile to provide a short review of the relevant concepts. Readers interested in a deeper discussion are referred to [Apostol, 1969] and [Golub and Van Loan, 1989], where most of this material is derived.

A.1 Gratuitous Mathematics: Definitions of Vector Spaces

A *vector space* is a set of elements upon which certain operations—namely addition and multiplication by numbers—can be performed. This includes real numbers, real-valued functions, n -dimensional vectors, vector-valued functions, complex numbers and a host of others.

Formally, if \mathbf{V} denotes a non-empty set of elements; \mathbf{x} , \mathbf{y} and \mathbf{z} represent elements from \mathbf{V} ; and a and b are real numbers, then \mathbf{V} is a linear space if it satisfies ten axioms:

1. Closure under addition.

For all \mathbf{x} and \mathbf{y} in \mathbf{V} , there exists a unique element in \mathbf{V} called their sum, denoted by $\mathbf{x} + \mathbf{y}$.

2. Commutative law for addition.

For all \mathbf{x} and \mathbf{y} in \mathbf{V} , it is the case that $\mathbf{x} + \mathbf{y} = \mathbf{y} + \mathbf{x}$.

3. Associative law for addition.

For all \mathbf{x} , \mathbf{y} and \mathbf{z} in \mathbf{V} , it is the case that $(\mathbf{x} + \mathbf{y}) + \mathbf{z} = \mathbf{x} + (\mathbf{y} + \mathbf{z})$.

4. Existence of a zero element.

There exists an element in \mathbf{V} , denoted by $\mathbf{0}$ such that $\mathbf{x} + \mathbf{0} = \mathbf{x}$ for all \mathbf{x} in \mathbf{V} .

5. Existence of negative elements.

For each \mathbf{x} in \mathbf{V} , the element $(-1)\mathbf{x}$ has the property that $\mathbf{x} + (-1)\mathbf{x} = \mathbf{0}$.

6. Closure under multiplication by numbers.

For each \mathbf{x} in \mathbf{V} and real number a , there exists an element in \mathbf{V} called their product, denoted by $a\mathbf{x}$.

7. Associative law for multiplication.

For each \mathbf{x} in \mathbf{V} and real numbers a and b , it is the case that $a(b\mathbf{x}) = (ab)\mathbf{x}$.

8. Distributive law for addition in \mathbf{V} .

For each \mathbf{x} in \mathbf{V} and real numbers a and b , it is the case that $(a + b)\mathbf{x} = a\mathbf{x} + b\mathbf{x}$.

9. Distributive law for addition of numbers.

For each \mathbf{x} in \mathbf{V} and real numbers a and b , it is the case that $(a + b)\mathbf{x} = a\mathbf{x} + b\mathbf{x}$.

10. Existence of an identity element.

For each \mathbf{x} in \mathbf{V} , it is the case that $1\mathbf{x} = \mathbf{x}$.

where subtraction between two vectors from \mathbf{V} and division of an element from \mathbf{V} by a real number are defined in the obvious way.

Although we specified that a and b are real numbers, it is worth noting that the theorems that apply to real vector spaces apply to complex vector spaces as well. When the real numbers in the axioms are replaced with complex numbers, we have a *complex vector space*. Generally, whatever the “numbers” are, we refer to them as *scalars*.

A.2 Bases and Components

Imagine a finite set of elements in a vector space, S . These elements are *dependent* if there exists a set of distinct elements in S , $\mathbf{x}_1 \dots \mathbf{x}_n$, and a corresponding set of scalars, $c_1 \dots c_n$ —that are all not zero—such that $\sum c_i \mathbf{x}_i = \mathbf{0}$. The elements are *independent* otherwise. If a set is independent, it then follows that for all choices of distinct $\mathbf{x}_1 \dots \mathbf{x}_n$, and corresponding scalars, $c_1 \dots c_n$, $\sum c_i \mathbf{x}_i = \mathbf{0}$ implies that $c_1 = \dots = c_n = 0$.

A *basis* is a finite set of elements from a vector space, \mathbf{V} , that form an independent set and *span* \mathbf{V} . In order for a set, $\mathbf{x}_1 \dots \mathbf{x}_n$, to span a vector space, \mathbf{V} , it must be the case that $\mathbf{y} = \sum c_i \mathbf{x}_i$ yields an element of \mathbf{V} , for all choices of $c_1 \dots c_n$.

A *subspace* of \mathbf{V} is defined as any non-empty subset of \mathbf{V} , S , that satisfies the closure axioms defined in § A.1. If S is a subspace of \mathbf{V} , we say that the elements of S span that subspace. Note that it is trivially true that if S spans \mathbf{V} that it is a subspace of \mathbf{V} .

As should be clear from this discussion, if we have a basis for \mathbf{V} , we can represent an element from that space as $\mathbf{y} = \sum c_i \mathbf{x}_i$. Further, $c_1 \dots c_n$ are uniquely determined by the basis, $\mathbf{x}_1 \dots \mathbf{x}_n$, and \mathbf{y} . The ordered n -tuple $(c_1 \dots c_n)$ is called the *components of \mathbf{y} with respect to the basis, $\mathbf{x}_1 \dots \mathbf{x}_n$* . Generally, vector space elements (or vectors for short) are represented by their components, which are delimited by brackets, $[]$. An element, \mathbf{v} , is written as $[v_1 \dots v_n]$, either horizontally (a row vector) or vertically (a column vector).

A.3 Inner Products and Norms

A real vector space, \mathbf{V} , has an *inner product* if for all \mathbf{x} , \mathbf{y} and \mathbf{z} from \mathbf{V} , and scalar a , there exists a unique real number (\mathbf{x}, \mathbf{y}) , that satisfies four axioms¹:

1. Symmetry or Commutativity.

$$(\mathbf{x}, \mathbf{y}) = (\mathbf{y}, \mathbf{x}).$$

2. Linearity or Distributivity.

$$(\mathbf{x}, \mathbf{y} + \mathbf{z}) = (\mathbf{x}, \mathbf{y}) + (\mathbf{x}, \mathbf{z}).$$

3. Associativity.

$$a(\mathbf{x}, \mathbf{y}) = (a\mathbf{x}, \mathbf{y}).$$

4. Positivity.

$$(\mathbf{x}, \mathbf{x}) > 0 \text{ if } \mathbf{x} \neq \mathbf{0}.$$

In a Euclidean space, the *norm* of a vector, $\|\mathbf{x}\|$ is defined to be the non-negative number $(\mathbf{x}, \mathbf{x})^{1/2}$. Norms satisfy three properties:

1. Positivity.

$$\|\mathbf{x}\| \geq 0, \text{ with equality only when } \mathbf{x} = \mathbf{0}.$$

2. Homogeneity.

$$\|a\mathbf{x}\| = |a| \|\mathbf{x}\|.$$

3. Triangle Inequality.

$$\|\mathbf{x} + \mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\|.$$

A real vector, \mathbf{x} , is *normalized* when $\|\mathbf{x}\| = 1$. It follows from homogeneity that a normalized vector can be constructed from a vector by dividing that vector by its norm.

Notice that the value of the norm of an element depends on the choice of the inner product. Here, we will usually be concerned with the *dot product*, the most commonly used inner product. It is usually denoted $\mathbf{x} \bullet \mathbf{y}$. It is defined as: $\mathbf{x} \bullet \mathbf{y} = \sum x_i y_i$, so the norm would

1. This is one of those cases where the axioms for complex vectors are slightly different than for their real counterparts; however, we will not discuss this here. The curious reader is invited to skim the references.

be: $\|\mathbf{x}\| = \sqrt{\sum (\mathbf{x}_i)^2}$. Two real vectors, \mathbf{x} and \mathbf{y} , are *orthogonal* when $\mathbf{x} \bullet \mathbf{y} = 0$ and *orthonormal* when $\mathbf{x} \bullet \mathbf{y} = 0$ and $\|\mathbf{x}\| = \|\mathbf{y}\| = 1$. A set is orthogonal (or orthonormal) when its elements are all orthogonal (or orthonormal) to each other.

A.4 The Matrix

There are several ways to define a matrix. Formally, a matrix represents any linear function whose domain and range are subsets of (finite) vector spaces. More practically, a matrix is an $m \times n$ array:

$$\mathbf{A} = \begin{bmatrix} \mathbf{a}_{1,1} & \mathbf{a}_{1,2} & \cdots & \mathbf{a}_{1,n} \\ \mathbf{a}_{2,1} & \mathbf{a}_{2,2} & \cdots & \mathbf{a}_{1,n} \\ \cdots & \cdots & \cdots & \cdots \\ \mathbf{a}_{m,1} & \mathbf{a}_{m,2} & \cdots & \mathbf{a}_{m,n} \end{bmatrix}. \quad (\text{A.1})$$

Of course, it is possible to think of matrices as objects in their own right. On the other hand, it is useful to keep in mind that they act as linear functions. To see how a matrix can represent a linear function, it helps to understand what a linear function is. By definition, a linear function, \mathbf{A} , is a function that preserves addition and multiplication by scalars; that is:

$$\mathbf{A}(a\mathbf{x} + b\mathbf{y}) = a\mathbf{A}(\mathbf{x}) + b\mathbf{A}(\mathbf{y})$$

for all scalars a and b , and vectors \mathbf{x} and \mathbf{y} . Linear functions include the identity operator, multiplication by a fixed scalar, and inner product with a fixed element.

Note that if \mathbf{V} and \mathbf{W} are vector spaces and \mathbf{V} is finite dimensional we can always construct a linear function $\mathbf{V} \rightarrow \mathbf{W}$ with prescribed values at the basis elements of \mathbf{V} . Further, if $\mathbf{v}_1 \dots \mathbf{v}_n$ is a basis set for the n -dimensional \mathbf{V} and $\mathbf{y}_1 \dots \mathbf{y}_n$ are n arbitrary elements in \mathbf{W} , then there is one and only one function, \mathbf{A} , such that $\mathbf{A}(\mathbf{v}_k) = \mathbf{y}_k$. In particular, \mathbf{A} maps an arbitrary element, $\mathbf{x} = \sum x_i \mathbf{v}_i$, to \mathbf{W} by $\mathbf{A}(\mathbf{x}) = \sum x_i \mathbf{y}_i$.²

2. This is actually a theorem, but we will state it without proof.

This says that \mathbf{A} is completely determined by its action on the given set of basis elements from \mathbf{V} . If \mathbf{W} is also finite-dimensional, say m -dimensional, with basis vectors $\mathbf{w}_1 \dots \mathbf{w}_m$, then each element of \mathbf{W} , $\mathbf{A}(\mathbf{v}_k)$, can be expressed uniquely as:

$$\mathbf{A}(\mathbf{v}_k) = \sum_{i=1}^m a_{ik} \mathbf{w}_i \quad (\text{A.2})$$

where $a_{1k} \dots a_{mk}$ are the components of $\mathbf{A}(\mathbf{v}_k)$ relative to the basis $\mathbf{w}_1 \dots \mathbf{w}_m$. If we think of these components as a column vector and collect them side by side, we reconstruct our picture of a matrix given in Equation (A.1). An element in the i^{th} row and j^{th} column will generally be written interchangeably as either \mathbf{A}_{ij} or a_{ij} , depending upon the context.

It is now useful to introduce the notion of matrix multiplication. If \mathbf{A} is an $m \times p$ matrix and \mathbf{B} is an $p \times n$ matrix, and \mathbf{C} is the product of \mathbf{A} and \mathbf{B} , denoted \mathbf{AB} , then \mathbf{C} is an $m \times n$ matrix whose ij^{th} element is:

$$\mathbf{C}_{ij} = \sum_{k=1}^p \mathbf{A}_{ik} \mathbf{B}_{kj} \quad (\text{A.3})$$

\mathbf{AB} is not defined when the number of columns of \mathbf{A} is not equal to the number of rows of \mathbf{B} . Note that the product is defined in such a way that it corresponds to the composition of the linear functions that \mathbf{A} and \mathbf{B} represent.

Vectors can also be thought of as matrices. A k -dimensional row vector is a $1 \times k$ matrix while its column vector counterpart is a $k \times 1$ matrix. Equation (A.2) is therefore a particular case of matrix-matrix multiplication.

There are many more operations that can be defined on matrices. We will define some common ones here. Before doing that, it is worthwhile to introduce names for several matrices with special forms. A *square* matrix is a matrix with the same number of rows as columns. The *zero* matrix, $\mathbf{0}$, is any matrix all of whose elements are 0. Generally, its size is obvious from context. A *diagonal* matrix is a matrix, \mathbf{D} , where $\mathbf{D}_{ij}=0$ for all $i \neq j$. The elements \mathbf{D}_{ii} may take on any value, including zero. Note that $\mathbf{0}$ is a diagonal matrix. The *identity* matrix is the square diagonal matrix, denoted \mathbf{I} , where $\mathbf{I}_{ii}=1$ for all i . It is called the identity matrix because it is the function whose output is always the same as its input. A *symmetric* matrix is one

where $\mathbf{A}_{ij}=\mathbf{A}_{ji}$ for all i,j . Note that a symmetric matrix must be square and that any square diagonal matrix is also symmetric.

Matrix addition, denoted $\mathbf{A}+\mathbf{B}$, results in a matrix \mathbf{C} , such that $\mathbf{C}_{ij}=\mathbf{A}_{ij}+\mathbf{B}_{ij}$. Addition is not defined when \mathbf{A} and \mathbf{B} are not of the same size. Matrix subtraction is also defined in the obvious way. Matrix multiplication by a scalar, denoted $a\mathbf{A}$, results in a matrix \mathbf{C} such that $\mathbf{C}_{ij}=a\mathbf{A}_{ij}$. Note that this is equivalent to $\mathbf{A}\mathbf{B}$ where \mathbf{B} is a square diagonal matrix of the correct size, such that $\mathbf{B}_{ii}=a$.

In general, it is possible to scale each column of a matrix \mathbf{A} with n columns by a corresponding value, $c_1 \dots c_n$, simply by constructing a square diagonal matrix \mathbf{B} such that $\mathbf{B}_{ii}=c_i$ and computing $\mathbf{A}\mathbf{B}$. In order to scale \mathbf{A} 's m rows by $c_1 \dots c_m$, construct \mathbf{B} such that $\mathbf{B}_{ii}=c_i$ and compute $\mathbf{B}\mathbf{A}$.

The *transpose* of an $m \times n$ matrix, denoted \mathbf{A}^T , is the $n \times m$ matrix, \mathbf{C} , such that $\mathbf{C}_{ij}=\mathbf{A}_{ji}$. Note that the transpose of the diagonal matrix \mathbf{D} is \mathbf{D} ; that is, $\mathbf{D}^T=\mathbf{D}$. For a product of matrices, $\mathbf{A}\mathbf{B}$, $(\mathbf{A}\mathbf{B})^T$ is $\mathbf{B}^T\mathbf{A}^T$.

If \mathbf{A} is a square matrix and there exists another matrix \mathbf{A}^{-1} such that $\mathbf{A}^{-1}\mathbf{A}=\mathbf{A}\mathbf{A}^{-1}=\mathbf{I}$ then \mathbf{A} is *nonsingular* and \mathbf{A}^{-1} is its *inverse*. Note that for a diagonal square matrix, \mathbf{D} , \mathbf{D}^{-1} is the matrix such that $\mathbf{D}^{-1}_{ii}=1/\mathbf{D}_{ii}$. From this, it should be clear that not all matrices, such as $\mathbf{0}$, have an inverse. They are called *singular* matrices. For a matrix, \mathbf{B} , whose columns (or rows) are pairwise orthonormal, $\mathbf{B}^{-1}=\mathbf{B}^T$. There is also a notion of a *pseudo-inverse*, that can be computed for non-square matrices as well as square matrices. We will discuss this in § A.5.

Just as vectors have norms, we can define one for matrices. There are several possibilities; however, we will mention only the *Frobenius norm*:

$$\|\mathbf{A}\|_{\mathbf{F}} = \sqrt{\sum_{i,j} (\mathbf{A}_{ij})^2}. \quad (\text{A.4})$$

Assuming the dot product as the inner product, this is equivalent to treating the matrix as if it were simply a vector and computing its norm.

A.5 Eigenvectors and Eigenvalues

Let \mathbf{V} be a vector space, \mathbf{S} a subspace of \mathbf{V} , and \mathbf{A} a linear function (i.e. matrix) of \mathbf{S} onto \mathbf{V} . A scalar, λ , is called an *eigenvalue* of \mathbf{A} if there is a nonzero element \mathbf{x} in \mathbf{S} such that:

$$\mathbf{Ax} = \lambda \mathbf{x} \quad (\text{A.5})$$

Further, \mathbf{x} is considered to be the *eigenvector* of \mathbf{A} corresponding to λ . There is exactly one eigenvalue corresponding to a given eigenvector (note that this does not mean that each eigenvector has an eigenvalue that is different from every other eigenvector).

Eigenvalues are not necessarily real numbers; they may be complex. When all the eigenvalues of a matrix are positive real numbers, the matrix is *positive definite*. When all the eigenvalues are simply non-negative real numbers (i.e. may include zero), the matrix is *positive semi-definite*.

Eigenvalues and eigenvectors are useful objects; however, they are not always easily interpretable beyond their mathematical definitions. There are some cases where they are. To illustrate such a case, let us introduce the *singular value decomposition* (SVD) of a matrix. Briefly, the SVD of a matrix, \mathbf{A} , is:

$$\mathbf{A} = \mathbf{U}\mathbf{S}\mathbf{V}^T$$

where \mathbf{U} contains orthonormal vectors, \mathbf{V} contains orthonormal vectors and \mathbf{S} is diagonal. All matrices have a singular value decomposition.

Decompositions like the SVD are useful because they often allow us to represent a matrix in some other form that makes it easier to understand and/or manipulate its structure. Here, the special form of \mathbf{U} , \mathbf{S} and \mathbf{V} make it is easy to manipulate \mathbf{A} . For example, it is easy to see that $\mathbf{A}^{-1} = \mathbf{V}\mathbf{S}^{-1}\mathbf{U}^T$ if \mathbf{A} is a square matrix. If \mathbf{A} is not square, this is known as the *pseudo-inverse* of \mathbf{A} , denoted \mathbf{A}^+ . As another example, note that the SVD of the matrix $\mathbf{A}\mathbf{A}^T$ is:

$$\mathbf{A}\mathbf{A}^T = \mathbf{U}\mathbf{S}\mathbf{V}^T\mathbf{V}\mathbf{S}\mathbf{U}^T = \mathbf{U}\mathbf{S}^2\mathbf{U}^T, \quad (\text{A.6})$$

where \mathbf{U} contains the eigenvectors of $\mathbf{A}\mathbf{A}^T$ and each diagonal element of \mathbf{S} is the square root of the corresponding eigenvalue. $\mathbf{A}\mathbf{A}^T$ has several interesting properties. In particular, it is positive semi-definite. Further, each eigenvalue represents the contribution of its eigenvector to the

variance of the data contained in \mathbf{A} . This particular computation is one way to perform *principal components analysis*, a popular linear dimensionality reduction technique.

A.6 Practical Issues in Linear Algebra

In practice there are several algorithmic and numeric issues that arise in the practical study of linear algebra. This is a rich and active research field, and as impossible to summarize in an appendix as it is to summarize the theories of linear algebra. Therefore, we will only introduce a few issues that affect this work directly in the hope that the reader gains an appreciation of some of the computational issues that arise in designing many of the algorithms encountered in this work.

A.6.1 Algorithmic Complexity

To begin with, let us state the complexity of several basic vector and matrix operations. For the purposes of this discussion, n is the number of total elements in a matrix.

Assuming a normal model with elements stored in the obvious way (as a contiguous block of memory), element access is a constant operation, $O(1)$. Operations like addition and subtraction take linear time, $O(n)$. Multiplication by a scalar is also linear. Calculating the transpose of a matrix is also linear.

Matrix-Matrix Multiplication is polynomial, roughly $O(n^3)$, assuming that the two matrices involved are of roughly the same size³. Inversion is no harder than multiplication and requires the same time. Numerically stable techniques for finding inverses are often preferred to more straightforward algorithms, but techniques like SVD still take roughly $O(n^3)$ time.

Important to an understanding of computational issues in linear algebra is realizing that such issues arise as much because of system and engineering realities as they do because of theoretical performance bounds on algorithms. For example, there are several reasonable algorithms that can be used to implement matrix multiplication. They are all equivalent mathematically; however, performance can differ greatly depending upon the underlying machine architecture and the way in which these procedures process memory. In general matrices can

3. It is possible to lower that time to around $O(n^{\log(7)})$ by cleverly storing some common subexpressions.

be very large objects. Therefore, it often matters a great deal whether data is stored in column or row order, for example, and whether implemented algorithms can exploit that knowledge.

A.6.2 Exploiting Matrix Structure

Rather than delve into a detailed discussion of specific algorithms, we will discuss the general notion of exploiting matrix structure to make certain operations not only faster, but practically tractable. Of the many kinds of matrices that we have noted, perhaps the most relevant to this work is the sparse matrix, so we will begin there.

Recall that sparse matrices have many zero entries. Therefore, it seems reasonable to create a data structure that represents only the non-zero elements. A common representation—and one used in this work—is to represent a sparse matrix as a pair, $\langle \mathbf{I}, \mathbf{D} \rangle$ where \mathbf{I} contains an ordered list of the indices of all the non-empty entries and \mathbf{D} contains the corresponding values. If the number of non-zero elements is m and $m \ll n$, then this representation may require many fewer orders of magnitude than a standard representation to store. For example, the AP collection used in this work is roughly described by a 280,000 x 280,000 matrix. This translates into roughly 292 gigabytes of storage (assuming four byte numbers). The sparse representation requires only about 334 megabytes.

This savings does have a cost, of course. Arbitrary element access, for example, now requires a binary search, which is $O(\lg k)$, in general. Inserting a new non-empty element also requires a search and—depending upon the structure used— $O(k)$ for moving/copying old elements in order to maintain sorted order. Iteration through all the elements of a matrix is still linear, however, and many of the other operations do not change in complexity. Further, if $m \ll n$, the savings in cost in space and time are extremely large. For example, sparse matrix-matrix addition will be $O(m)$ (the $\lg(m)$ cost of insertions is not necessary because we are iterating in order; this is essentially a merge operation). Matrix multiplication has similar behavior. Even SVD calculations (which often depend upon matrix-vector multiplication at their core) will benefit a great deal.

As the number of non-zeros increases, the benefit of this representation decreases greatly. It is therefore necessary to recognize when to abandon it. For example, dense-sparse operations should almost always yield dense objects, not sparse ones. On the other hand,

sparse-sparse operations should probably yield sparse objects. Therefore, to maintain a performance edge, it is often necessary to be very careful in how operations are ordered. For example, a series of additions like $\mathbf{A}+\mathbf{B}+\mathbf{C}+\mathbf{D}$ where \mathbf{A} is dense but \mathbf{B} , \mathbf{C} , and \mathbf{D} are sparse should be executed from right to left, rather than left to right. Another common case in this work involves subtracting the mean column (or row) from a matrix and then multiplying that result by another matrix; that is, (with appropriate abuse of notation) $\mathbf{A}(\mathbf{B}-\mu)$. The vector μ is dense, so the obvious order of operations would require a dense-sparse subtraction and then a dense-dense multiplication. In many cases, it would be far faster to perform the *three* operations of a dense-sparse matrix multiplication, a dense-dense matrix-vector multiplication, and a dense-dense matrix subtraction, as in $\mathbf{AB}-\mathbf{A}\mu$.

Sparse matrices are especially important in this work. Still, other special-purpose representations and algorithms for matrices such as diagonal matrices and symmetric matrices can also lead to significant improvements in speed and storage.

It is also worth mentioning that many of the useful intermediate matrices that result from algorithms used in this work are too large to store. For example, computing a small SVD of a very large matrix, \mathbf{A} , is commonly performed in this work; however, the matrix itself is well beyond the capacity of our machines. To address this problem, it is important to note that it is often the case that \mathbf{A} is the result of a matrix multiply, like $\mathbf{A}=\mathbf{BCD}$ where \mathbf{B} , \mathbf{C} , and \mathbf{D} are relatively small (recall that an $n \times 1$ vector times a $1 \times n$ vector will result in a quadratically larger $n \times n$ matrix). In this case, if we store only the multiplicands, \mathbf{B} , \mathbf{C} , and \mathbf{D} it is still possible to calculate the SVD. This works because at the center of the method that we use for computing $\text{SVD}(\mathbf{A})$ is the repeated computation of $\mathbf{AA}^T\mathbf{x}$ for various \mathbf{x} . Computing $\mathbf{BCDD}^T\mathbf{C}^T\mathbf{B}^T\mathbf{x}$ from right to left yields the same result, but each multiplication yields a relative small vector instead of a prohibitively large matrix.

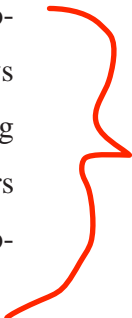
A.6.3 Parallel Algorithms

Most of the algorithms used in this work are implemented in parallel. BRAZIL, the primary system used for our experiments, is implemented as a part of the Parallel Problems Server, a “linear algebra server” that operates across many processors and machines (see Chapter 7 for

a complete discussion). When implementing linear algebra systems in parallel, a number of issues arise. We will very briefly mention a few.

To begin with, data must be *distributed* across processors. The way in which this is done has a large impact on algorithm performance. For example, imagine that we allow two kinds of distributions: a processor “owns” either a subset of the rows of a matrix or a subset of the columns of a matrix. If \mathbf{A} is distributed by rows, but we want to retrieve, say, a single column, this requires the cooperation of (and thus communication overhead for) all processors.

Consider the matrix multiplication operation, \mathbf{AB} . Depending upon the algorithm chosen to compute the product, communication cost is minimized when \mathbf{A} is distributed by rows and \mathbf{B} by columns. Is it worth the initial cost of redistributing the matrices before performing the computation? Although it can require shuffling hundreds of megabytes across processors and machines, it is often worth it to avoid the communication cost of computing a misdistributed \mathbf{AB} (not to mention the savings in programming effort).



Note that this only hints at the added complexity of normally simple operations, like element access. Retrieving a particular element of a matrix now requires determining which processor “owns” an element followed by communication across processors (and possibly machines) to retrieve the element.

As with sparse representations, one hopes that the benefit of using a parallel approach far outweighs costs. For the large data sets explored here, this is certainly the case. Many operations require minimum communication and so enjoy a nearly linear speed-up. Even when a great deal of communication is needed, the practical increase in speed and in-core memory is significant.