Chapter 10 Case Study: Generative Matrix Computation Library (GMCL) [155]

10.1 Domain Analysis

10.1.1 Domain Definition

10.1.1.1 Goals and Stakeholders

Our goal is to develop a *matrix computation library*. Thus, our domain is the *domain of matrix computation libraries*. The most important group of stakeholders are users solving linear algebra problems. We want the library to be highly reusable, adaptable, and very efficient (in terms of execution speed and memory consumption) and provide a highly intentional interface to application programmers. For our case study, we will always prefer technologically better solutions and ignore organizational issues. In a real world setting of a software developing organization, the analysis of stakeholders and their goals, strategic project goals, and other organizational issues may involve a significant effort.

10.1.1.2 Domain Scoping and Context Analysis

10.1.1.2.1 Characterization of the Domain of Matrix Computation Libraries

Our general domain of interest is referred to as *matrix computations*, which is a synonym for applied, algorithmic linear algebra. Matrix computations is a mature domain with a history of more than 30 years (e.g. [Wil61]). The domain includes both the well-defined mathematical theory of linear algebra as well as the knowledge about efficient implementations of algorithms and data structures for solving linear algebra problems on existing computer architectures. This implementation knowledge is well documented in the literature, e.g. [GL96, JK93].

In particular, we are interested in the *domain of matrix computation libraries*. A matrix computation library contains ADTs and algorithm families for matrix computations and is intended to be used as a part of a larger application. Thus, it is an example of a *horizontal* domain. Examples of *vertical* domains involving matrix computations would be matrix computation environments (e.g. Matlab [Pra95]) or specialized scientific workbenches (e.g. for electromagnetics or quantum chemistry). They are vertical domains since they contain entire applications including GUIs, graphical visualization, persistent storage for matrices, etc.

The main concepts in matrix computations are vectors, matrices, and computational methods, e.g. methods for solving a system of linear equations or computing the eigenvalues. A glossary of some of the terms used in matrix computations is given in 10.4.

10.1.1.2.2 Sources of Domain Knowledge

The following sources of domain knowledge were used in the analysis of the matrix computation libraries domain:

- literature on matrix computations: [GL96, JK93];
- documentation, source code, and articles describing the design of existing matrix computation libraries: [LHKK79, DDHH88, DDDH90, CHL+96] and those listed in Table 15 and Table 16;
- online repository of matrices: [MM].

10.1.1.2.3 Application Areas of Matrix Computation Libraries

In this section, we will identify features characterizing matrix computation libraries by analyzing different application areas of matrix computations.

Table 13 and Table 14 list some typical application areas of matrix computations and the types of matrices and computations which are required for solving the problems in the listed areas. The application areas were grouped into two categories: one requiring dense matrices (Table 13) and the other one requiring sparse matrices (Table 14). In general, large matrix problems usually involve sparse matrices and large dense matrix problems are much less common.

Application area	**Dense matrix types**	**Computational problems**
electromagnetics (Helmholtz equation), e.g. radar technology, stealth (i.e. “radar-invisible”) airplane technology	complex, Hermitian (rarely also non-Hermitian), e.g. 55296 by 55296	boundary integral solution (specifically the method of moments)
flow analysis (Laplace or Poisson equation), e.g. airflow past an airplane wing, flow around ships	symmetric, e.g. 12088 by 12088	boundary integral solution (specifically the panel method)
diffusion of solid bodies in liquids	block Toeplitz	i. n. a.[156]
diffusion of light through small particles	block Toeplitz	i. n. a.
noise reduction	block Toeplitz	i. n. a.
quantum mechanical scattering (computing the scattering of elementary particles from other particles and atoms; involves Schrödinger wave function)	i. n. a.	dense linear systems
quantum chemistry (Schrödinger wave function)	real symmetric, occasionally Hermitian, small and dense (large systems are usually sparse)	symmetric eigenvalue problems
material science	i. n. a.	unsymmetric eigenvalue problems
real-time signal processing applications	i. n. a.	rank-revealing factorizations and the updating of factorizations after low rank changes

Table 13 *Examples of application areas for dense matrix computations (based on examples found in [Ede91, Ede93, Ede94,Hig96])*

Application area	**Sparse matrix types**	**Computational problems**
static analyses in structural engineering[157], e.g. static analysis of buildings, roofs, bridges, airplanes, etc.	real symmetric positive definite, pattern symmetric indefinite, e.g. 3948 by 3948 with 60882 entries	generalized symmetric eigenvalue problem, finite-element modeling, linear systems
dynamic analysis in structural engineering, e.g. dynamic analysis of fluids, suspension bridges, transmission towers, robotic control	real symmetric and positive definite or positive semi-definite or indefinite	symmetric eigenvalue problems, linear systems
hydrodynamics	real unsymmetric, e.g. 100 by 100 with 396 entries	eigenvalues of the Jacobi matrix
oceanic modeling, e.g. models of the shallow waves for the Atlantic and Indian Oceans	real symmetric indefinite, real skew symmetric, e.g. 1919 by 1919 with 4831 entries	finite-difference model
acoustic scattering	complex symmetric	i. n. a.
fluid flow modeling, fluid dynamics, flow in networks	real unsymmetric, symmetric structure, e.g. 511 by 511, 2796 entries and 23560 by 23560 with 484256 entries	iterative and direct methods, eigenvalue and eigenvector problems (in perturbation analysis), Lanczos method
petroleum engineering, e.g. oil recovery, oil reservoir simulation	real unsymmetric, symmetric structure, e.g. 2205 by 2205 with 14133 entries	i. n. a.
electromagnetic field modeling, e.g. integrated circuit applications, power lines	real pattern symmetric indefinite, real pattern symmetric positive definite, real unsymmetric, e.g. 1074 by 1074 with 5760 entries	finite-element modeling, symmetric and unsymmetric eigenvalue problem
power systems simulations, power system networks	real unsymmetric, real symmetric indefinite, real symmetric positive definite, e.g. 4929 by 10595 with 47369 entries	symmetric and unsymmetric eigenvalue problems
circuit simulation	real unsymmetric, 58 by 59 with 340 entries	i. n. a.
astrophysics, e.g. nonlinear radiative transfer and statistical equilibrium in astrophysics	real unsymmetric, e.g. 765 by 765 with 24382 entries	i. n. a.
nuclear physics, plasma physics	real unsymmetric, e.g. 1700 by 1700 with 21313 entries	Large unsymmetric generalized eigenvalue problems
quantum chemistry	complex symmetric indefinite, e.g. 2534 by 2534 with 463360 entries	symmetric eigenvalue problems

chemical engineering, e.g. simple chemical plant model, hydrocarbon separation problem	real unsymmetric, e.g. 225 by 225 with 1308 entries	conjugate gradient eigenvalue computation, initial Jacobian approximation for sparse nonlinear equations
probability theory and its applications, e.g. simulation studies in computer systems involving Markov modeling techniques	real unsymmetric, e.g. 163 by 163 with 935 entries	unsymmetric eigenvalues and eigenvectors
economic modeling e.g. economic models of countries, models of economic transactions	real unsymmetric, e.g. 2529 by 2529 with 90158 entries	i. n. a.
demography, e.g. model of inter-country migration	real unsymmetric, often relatively large fill-in with no pattern, e.g. 3140 by 3140 with 543162 entries	i. n. a.
surveying	real unsymmetric, e.g. 480 by 480 with 17088 entries	least squares problem
air traffic control	sparse real symmetric indefinite, e.g. 2873 by 2873 with 15032 entries	conjugate gradient algorithms
ordinary and partial differential equations	real symmetric positive definite, real symmetric indefinite, real unsymmetric, e.g. 900 by 900 with 4322 entries	symmetric and unsymmetric eigenvalue problems

Table 14 *Examples of application areas for sparse matrix computations (based on examples found in [MM])*

10.1.1.2.4 Existing Matrix Computation Libraries

As of writing, the most comprehensive matrix computation libraries available are written in Fortran. However, several object-oriented matrix computation libraries (for performance reasons, they are written mostly in C++) are currently under development. Table 15 and Table 16 list some of the publicly and commercially available matrix computation libraries in Fortran and in C++ (also see [OONP]).

Matrix computations library	**Features**
LINPACK a matrix computation library for solving dense linear systems; superseded by LAPACK see [DBMS79] and http://www.netlib.org/linpack	*language*: Fortran *matrix types*: dense, real, complex, rectangular, band, symmetric, triangular, and tridiagonal *computations*: factorizations (Cholesky, QR), systems of linear equations (Gaussian elimination, various factorizations), linear

	least squares problems, and singular value problems
EISPACK a matrix computation library for solving dense eigenvalue problems; superseded by LAPACK see [SBD+76] and http://www.netlib.org/eispack	*language*: Fortran *matrix types*: dense, real, complex, rectangular, symmetric, band, and tridiagonal *computations*: eigenvalues and eigenvectors, linear least squares problems
LAPACK a matrix computation library for dense linear problems; supersedes both LINPACK and LAPACK see [ABB+94] and http://www.netlib.org/lapack	*language*: Fortran *matrix types*: dense, real, complex, rectangular, band, symmetric, triangular, and tridiagonal *computations*: systems of linear equations, linear least squares problems, eigenvalue problems, and singular value problems
ARPACK a comprehensive library for solving real or complex and symmetric or unsymmetric eigenvalue problems; uses LAPACK and BLAS (see text below Table 16) see [LSY98] and http://www.caam.rice.edu/software/ARPACK	*language*: Fortran *matrix types:* provided by BLAS and LAPACK *computations*: Implicitly Restarted Arnoldi Method (IRAM), Implicitly Restarted Lanczos Method (IRLM), and supporting methods for solving real or complex and symmetric or unsymmetric eigenvalue problems
LAPACK++ a matrix computation library for general dense linear problems; provides a subset of LAPACK functionality in C++ see [DPW93] and http://math.nist.gov/lapack++	*language*: C++ *matrix types*: dense, real, complex, rectangular, symmetric, symmetric positive definite, band, triangular, and tridiagonal *computations*: factorizations (LU, Cholesky, QR), systems of linear equations and eigenvalue problems, and singular value problems
ARPACK++ subset of ARPACK functionality in C++ (using templates) see [FS97] and http://www.caam.rice.edu/software/ARPACK /arpack++.html	*language*: C++ *matrix types*: dense, sparse (CSC), real, complex, rectangular, symmetric, band *computations*: Implicitly Restarted Arnoldi Method (IRAM)
SparseLib++ library with sparse matrices; intended to be used with IML++ see [DLPRJ94] and http://math.nist.gov/sparselib++	*language*: C++ *matrix types*: sparse

IML++ (Iterative Methods Library) library with iterative methods; requires a library implementing matrices, e.g. SparseLib++ see [DLPR96] and http://math.nist.gov/iml++	*language*: C++ *matrix types*: library implementing matrices *computations*: iterative methods for solving both symmetric and unsymmetric linear systems of equations (Richardson Iteration, Chebyshev Iteration, Conjugate Gradient, Conjugate Gradient Squared, BiConjugate Gradient, BiConjugate Gradient Stabilized, Generalized Minimum Residual, Quasi-Minimal Residual Without Lookahead)
Newmat, version 9 a matrix computation library for dense linear problems; does not use C++ templates see http://nz.com/webnz/robert/nzc_nm09.html	*language*: C++ *matrix types*: dense, real, rectangular, diagonal, symmetric, triangular, band *computations*: factorizations (Cholesky, QR, singular value decomposition), eigenvalues of a symmetric matrix, Fast Fourier
TNT (Template Numerical Toolkit) a C++ matrix computation library for linear problems; it has a template-based design; eventually to supersede LAPACK++, SparseLib++, and IMC++; as of writing, with rudimentary functionality see [Poz96] and http://math.nist.gov/tnt	*language*: C++, extensive use of templates *matrix types*: dense, sparse, real, complex, rectangular, symmetric, triangular *computations*: factorizations (LU, Cholesky, QR), systems of linear equations contains an interface to LAPACK
MTL (Matrix Template Library) a C++ matrix library; it has an STL-like template-based design; its goal is to provide only one version of any algorithm and adapt it for various matrices using parameterization and iterators; it implements register blocking using template metaprogramming; it exhibits an excellent performance comparable to tuned Fortran90 code see [SL98a, SL98b] and http://www.lsc.nd.edu/	*language*: C++, extensive use of templates *matrix types*: dense, sparse, real, complex, rectangular, symmetric, band

Table 15 *Some of the publicly available matrix computation libraries and their features*

Matrix computations library	**Features**
Math.h++ C++ vector, matrix, and an array library in one	*language*: C++ *matrix types*: dense, real, complex, rectangular

Rogue Wave Software, Inc., see http://www.roguewave.com/products/math	*computations*: LU factorization, FFT
LAPACK.h++ works on top of Math.h++; offers functionality of the Fortran LAPACK library in C++ Rogue Wave Software, Inc., see http://www.roguewave.com/products/lapack	*language*: C++ *matrix types*: dense (some from Math.h++), sparse, real, complex, symmetric, hermitian, skew-symmetric, band, symmetric band, hermitian band, lower triangular, and upper triangular matrices *computations*: factorizations (LU, QR, SVD, Cholesky, Schur, Hessenberg, complete orthogonal, tridiagonal), real/complex and symmetric/unsymmetric eigenvalue problems
Matrix<LIB> LINPACK and EISPACK functionality in C++ Matlab-like syntax MathTools Ltd, see http://www.mathtools.com	*language*: C++ *matrix types*: dense, real, complex *computations*: factorizations (Cholesky, Hessenberg, LU, QR, QZ, Schur and SVD), solving linear systems, linear least squares problems, eigenvalue/eigenvector problems
ObjectSuite™ C++: IMSL Math Module for C++ a matrix computation library for dense linear problems Visual Numerics, Inc., see http://www.vni.com/products/osuite	*language*: C++ *matrix types*: dense, real, complex, rectangular, symmetric/Hermitian and symmetric/Hermitian positive definite *computations*: factorizations (LU, Cholesky, QR, and Singular Value Decomposition), linear systems, linear least squares problems, eigenvalue and eigenvector problems, two-dimensional FFTs

Table 16 *Some of the commercially available matrix computation libraries and their features*

A set of basic matrix operations and formats for high-performance architectures has been standardized in the form of the *Basic Linear Algebra Subprograms (BLAS).* The operations are organized according to their complexity into three levels: Level-1 BLAS contain operations requiring O(n) of storage for input data and O(n) time of work, e.g. vector/vector operations (see [LHKK79]), Level-2 BLAS contain operations requiring O(n^2) of input and O(n^2) of work, e.g. matrix-vector multiplication (see [DDHH88]), Level-3 BLAS contain operations requiring O(n^2) of input and O(n^3) of work, e.g. matrix-matrix multiplication (see [DDDH90, BLAS97]). There are also Sparse BLAS [CHL+96], which are special BLAS for sparse matrices. The Sparse BLAS standard also defines various sparse storage formats. Different implementations of BLAS are available from http://www.netlib.org/blas/.

10.1.1.2.5 Features of the Domain of Matrix Computation Libraries

From the analysis of application areas and existing matrix computation libraries (Table 13, Table 14, Table 15, and Table 16), we can derive a number of major matrix types and computational method types which are common in the matrix computations practice. They are listed in Table 17 and Table 18, respectively. The types of matrices and computations represent the main features of the domain of matrix computation libraries and can be used to describe the scope of a matrix computation library. The rationale for including each of these features in a concrete matrix computation library implementation are also given in Table 17 and Table 18. Some features such as dense matrices and factorizations are basic features required by many other features and they should be included in any matrix computation library implementation. Some other features such as complex matrices and methods for computing eigenvalues are — unless directly required by some stakeholders — optional and their implementation may be deferred.

Matrix type	**Rationale for inclusion**
dense matrices	Dense matrices are ubiquitous in linear algebra computations and are mandatory for any matrix computation library.
sparse matrices	In practice, large linear systems are usually sparse.
real matrices	Real matrices are very common in linear algebra problems.
complex matrices	Complex matrices are less common than real matrices but still very important for a large class of problems.
rectangular, symmetric, diagonal, and triangular matrices	Rectangular, symmetric, diagonal, and triangular matrices are very common in linear algebra problems and are mandatory for any matrix computation library.
band matrices	Band matrices are common in many practical problems, e.g. a large percentage of the matrices found in [MM] are band matrices.
other matrix shapes (e.g. Toeplitz, tridiagonal, symmetric band)	There is a large number of other matrix shapes which are specialized for various problems. In general, providing all possible shapes in a general purpose matrix computation library is not possible since new applications may require new specialized shapes.

Table 17 *Major matrix types and the rationale for their inclusion in the implemented feature set*

Computational methods	Rationale for inclusion
factorizations (decompositions)	Factorizations are needed for direct methods and matrix analysis and are mandatory for any matrix computation library.
direct methods for solving linear systems	Direct methods (e.g. using the LU factorization) are standard methods for solving linear systems.
least squares methods	The least squares approach is concerned with the solution of overdetermined systems of equations. It represents the standard scientific method to reduce the influence of errors when fitting models to given observations.
symmetric and unsymmetric eigenvalue and eigenvector methods	Eigenvalue methods have numerous applications in science and engineering.
iterative methods for linear systems	Iterative methods for linear systems are the methods of choice for some large sparse systems. There are iterative methods for solving linear systems and for computing eigenvalues.

Table 18 *Major matrix computational methods types and the rationale for their inclusion in the implemented feature set*

Figure 137 summarizes the features of a matrix computation library in a feature diagram. The priorities express the typicality rate of the variable features. These typicality rates are informal and are intuitively based on the analysis of application areas and existing matrix computation libraries.

Another important feature of a matrix computation library, which will not be considered here, is its target computer architecture, e.g. hierarchical memory, multiple processors with distributed memory or shared memory, etc.

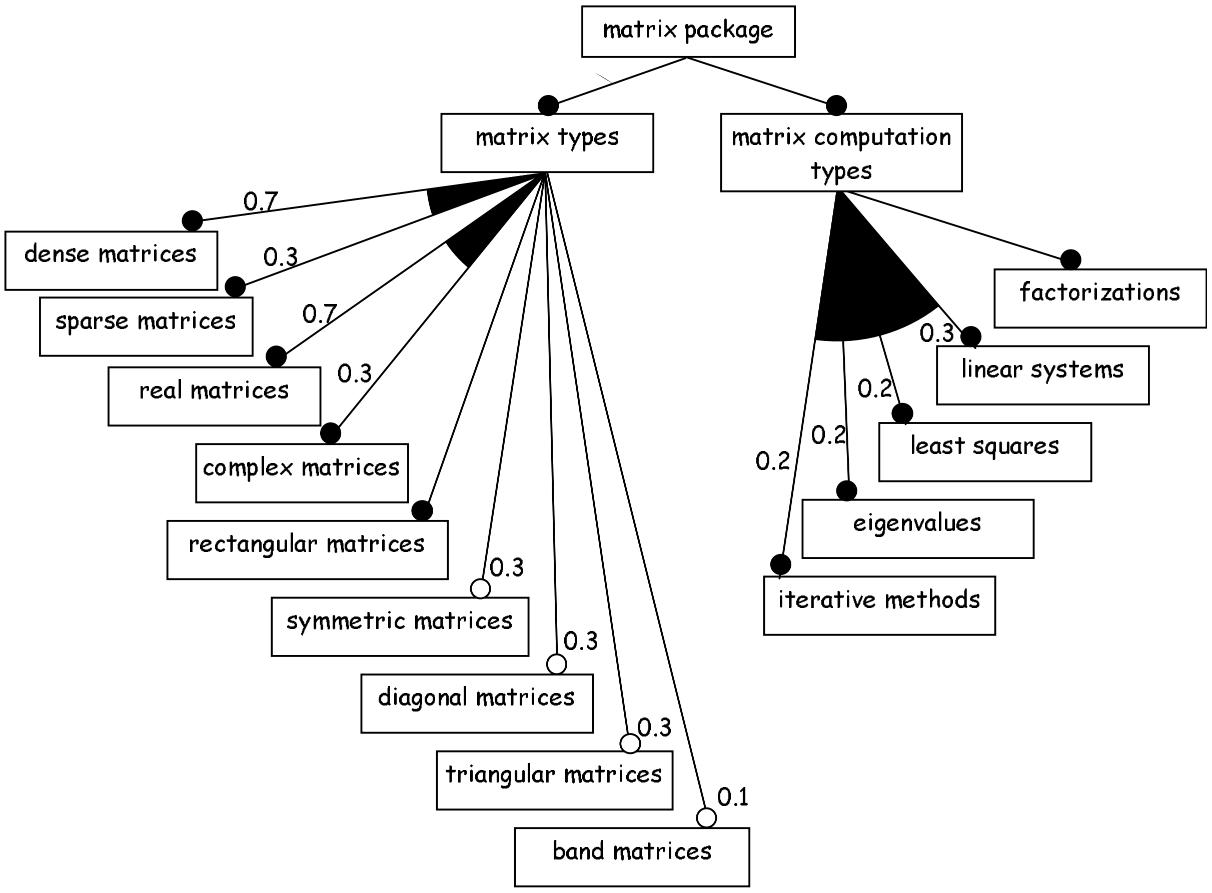

Figure 137 *Feature diagram of a matrix computation library*

10.1.1.2.6 Relationship to Other Domains

The *domain of array libraries* is an example of an *analogy domain* (see Section 3.6.3) of the domain of matrix computation libraries. Array libraries implement arrays (including two-dimensional arrays) and numerical computations on arrays. Thus, there are significant similarities between array libraries and matrix computation libraries. But there are also several differences:

- Array libraries, in contrast to matrix computation libraries, also cover arrays with more than two dimensions.
- Array operations are primarily elementwise operations. For example, in an array library * means elementwise multiply, whereas in a matrix computation library * designates matrix multiplication.
- Arrays usually support a wide range of element types, e.g. int, float, char, bool, and user defined types, whereas the type of matrix elements is either real or complex numbers.
- Array libraries usually do not provide a comprehensive set of algorithms for solving complicated linear problems. They rather focus on other areas, e.g. signal processing.

- Array libraries usually do not provide any special support for different shapes and densities.

Blitz++ [Vel97] is an example of an array library. Math.h++ (see Table 16) combines aspects of both an array and a matrix computation library in one.

An example of another analogy domain is the domain of *image processing libraries*. Images are somewhat similar to matrices. However, there are also several differences:

- The elements of an image are binary, gray scale, or color pixel values. Binary and color pixel values require a different representation than matrix elements. For example, color pixels may be represented using three values and a collection of binary pixels is usually represented by one number.

- The operations and algorithms required in image processing are different than those used for solving linear problems.

An example of a *support domain* is the *domain of container libraries*. A container library, e.g. the Standard Template Library (STL; [MS96, Bre98]) could be used to implement storage for matrix elements in a matrix computation library.

10.1.2 Domain Modeling

10.1.2.1 Key Concepts of the Domain of Matrix Computation Libraries

The key concepts of the domain of matrix computation libraries are

- abstract data types: *vectors* and *matrices*;

- algorithm families: *factorizations*, *solving systems of linear equations*, *solving least squares problems*, *solving eigenvalue and eigenvector problems*, and *iterative methods*.

10.1.2.2 Feature Modeling of the Key Concepts

10.1.2.2.1 Features of Vectors and Matrices

This section describes the features of vectors and matrices. Since the vector features represent a subset of the matrix features, we only list the matrix features and indicate if a feature does not apply to vectors. Please note that vectors can be adequately represented as matrices with number of rows equal one or number of columns equal one.

We have the following matrix features:

- *element type*: type of the matrix elements;

- *subscripts*: subscripts of the matrix elements;

- *structure*: the arrangement and the storage of matrix elements:

- *entry type*: whether an entry is a scalar or a matrix;

- *density*: whether the matrix is sparse or dense;

- *shape*: the arrangement pattern of the nonzero matrix elements (this feature does not apply to vectors);

- *representation*: the data structures used to store the elements;

- *format*: the layout of the elements in the data structures;
- *memory management*: allocating and relinquishing memory;
- *operations*: operations on matrices (including their implementations);
- *attributes*: matrix attributes, e.g. number of rows and columns;
- *concurrency and synchronization*: concurrent execution of algorithms and operations and synchronization of memory access;
- *persistency*: persistent storage of a matrix instance;
- *error handling*: error detection and notification, e.g. bounds checking, compatibility checking for vector-vector, matrix-vector, and matrix-matrix operations.

10.1.2.2.1.1 Element Type

The only element types occurring in linear algebra (and also in the application areas listed in Table 13 and Table 14) are real and complex numbers. Existing libraries (Table 15 and Table 16) typically support single and double precision real and complex element types. Other element types (e.g. bool, user defined types, etc.) are covered by array libraries (see Section 10.1.1.2.6).

10.1.2.2.1.2 Subscripts (Indices)

The following are the subfeatures concerning subscripts:

- *index type*: The type of subscripts is an integral type, e.g. char, short, int, long, unsigned short, unsigned int, or unsigned long.
- *maximum index value*: The choice of index type, e.g. char or unsigned long, determines the maximal size of a matrix or vector.
- *index base*: There are two relevant choices for the start value of indices: *C-style indexing* (or *0-base indexing*), which starts at 0, and the *Fortran-style indexing* (or *1-base indexing*), which starts at 1. Some libraries, e.g. TNT (see Table 15), provide both styles at the same time (TNT provides the operator "[]" for 0-base indexing and the operator "()" for 1-base indexing).
- *subscript ranges*: Additionally, we could also have a subscript type representing subscript ranges. An example of range indexing is the Matlab indexing style [Pra95], e.g. 1:4 denotes a range from 1 to 4, 0:9:3 denotes a range from 0 to 9 with stride 3. An example of a library supporting subscript ranges is Matrix<LIB> (see Table 16).

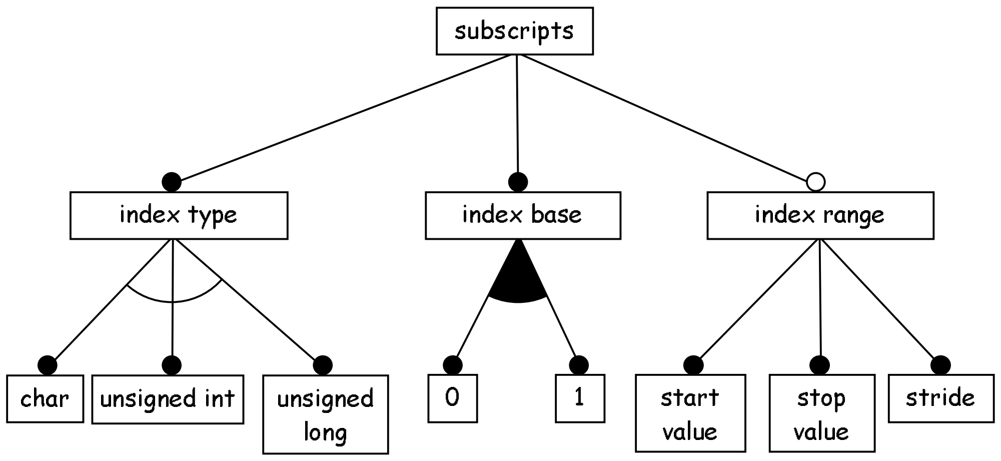

Figure 138 *Subscripts*

10.1.2.2.1.3 Structure

Structure is concerned with the arrangement and the storage of matrix or vector elements. We can exploit the arrangement of the elements in order to reduce storage requirements and to provide specialized and faster variants of basic operations and more complex algorithms. The subfeatures of the *structure* of matrices are shown in Figure 139.

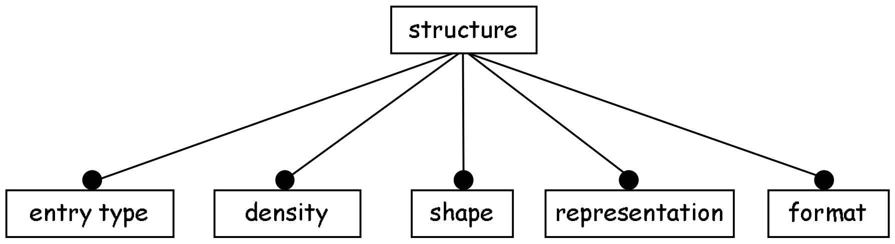

Figure 139 *Structure*

10.1.2.2.1.3.1 Entry Type

The entries in a matrix are usually scalars (e.g. real or complex). These types of matrices are referred to as *point-entry matrices* [CHL+96]. There are also matrices whose entries are matrices and they are referred to as *block matrices* [CHL+96, GL96]. Block matrices are common in high-performance computing since they allow us to express operations on large matrices in terms of operations on small matrices. This formulation enables us to take advantage of the hierarchical memory organization on modern computer architectures.

The memory of modern computer architectures is usually organized into a hierarchy: The higher levels in the hierarchy feature memory fast in access but of limited capacity (e.g. processor cache). As we move down the hierarchy, the memory speed decreases but its capacity increases (e.g. main memory, disk).

When performing a matrix operation, it is advantageous to keep all the operands in cache in order to eliminate excessive data movements between the cache and the main memory during the operation. If the operands are matrices which entirely fit into the cache, we can use the point-entry format. But if a matrix size exceeds the cache size, the block format should be preferred. Operations on block matrices are performed in terms of operations on their blocks, e.g. matrix multiplication is performed in terms of multiplications of the block submatrices. By properly adjusting the block size, we are able to fit the arguments of the submatrix operation into the cache.

Furthermore, we distinguish between *constant blocks* (i.e. blocks have equal sizes) and *variable blocks* (i.e. blocks have variable sizes). The subfeatures of *entry type* are summarized in Figure 140.

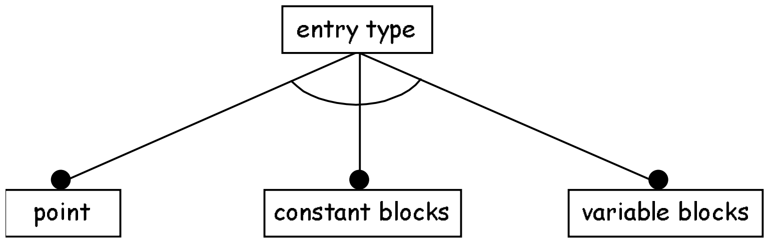

Figure 140 *Entry Type*

Blocking is used in high performance linear algebra libraries such as LAPACK (see Table 15).

10.1.2.2.1.3.2 Density

One of the major distinctions between matrices is whether a matrix is *dense* or *sparse*. A dense matrix is a matrix with a large percentage of *nonzero elements* (i.e. elements not equal zero). A sparse matrix, on the other hand, contains a large percentage of *zero elements* (usually more than 90%). In [Sch89], Schendel gives an example of a sparse matrix in the context of the frequency analysis of linear networks which involves solving a system of linear equations of the form $A(\omega_i)x = b$. In this example, A is a 3304-by-3304 Hermitian matrix with 60685 nonzero elements. Thus, the nonzero elements make up only 0.6% of all elements in A (i.e. the fill-in is 0.6%). Furthermore, the LU-factorization of A yields a new matrix with an even smaller fill-in of 0.4% (see Table 14 for more examples of sparse matrices). The representation of A as a dense matrix would require several megabytes of memory. However, it is necessary to store only the nonzero elements, which dramatically reduces the storage requirements for sparse matrices. The knowledge of the density of a matrix allows us not only to optimize the storage consumption, but also the processing speed since we can provide specialized variants of operations which take advantage of sparseness.

Most matrix computation libraries provide dense matrices and some matrix computation libraries also implement sparse matrices (e.g. LAPACK.h++; see Table 16). Since most of the large matrix problems are sparse (see Table 14), a general-purpose matrix computation library is much more attractive if it implements both dense and sparse matrices.

10.1.2.2.1.3.3 Shape

Matrix computations involve matrices with different arrangement patterns of the nonzero elements. Such arrangement patterns are referred to as *shapes*. Some of the more common shapes include the following (see Figure 141):

- *Rectangular* and *square matrices*: A rectangular matrix has a different number of rows than the number of columns. The number of rows and the number of columns in a square matrix are equal.

- *Null matrix*: A null matrix consists of only zero elements. No elements have to be stored for a null matrix but only the number of rows and columns.

- *Diagonal matrix*: A diagonal matrix is a square matrix with all zero elements except the diagonal elements (i.e. elements whose row index and column index are equal). Only the (main) diagonal elements have to be stored. If they are all equal, the matrix is referred to as a *scalar matrix* and only the scalar has to be stored.

- *Identity matrix*: An identity matrix is a diagonal matrix whose diagonal entries are all equal 1. No elements have to be stored for an identity matrix but only the number of rows and columns.

- *Symmetric*, *skew-symmetric* or *anti-symmetric*, *Hermitian*, and *skew-Hermitian matrices*: For all elements of a symmetric matrix the following equation holds $a_{ij} = a_{ji}$. For a skew-symmetric matrix, we have a slightly different equation: $a_{ij} = - a_{ji}$. A complex-valued matrix with symmetric real part and skew-symmetric imaginary part is referred to as a Hermitian. If, on the other hand, the real part is skew-symmetric and the imaginary part is symmetric, we have a skew-Hermitian matrix. For all these four matrix types we only need to store one half of the matrix. One possible storage format is to consecutively store all the rows (or columns or diagonals) of one half of the matrix in a vector and use an indexing formula to access the matrix elements.

- *Upper* or *lower triangular* or *unit triangular* or *Hessenberg matrices*: An upper triangular matrix is a square matrix which has nonzero elements only on and above the main diagonal. If the diagonal elements are only ones, the matrix is referred to as *unit upper triangular*. If the diagonal elements are only zeros, the matrix is referred to as *strictly upper triangular*. If, on the other hand, the main diagonal and also the diagonal below contains nonzeros, the matrix is referred to as an upper Hessenberg. The lower triangular, lower unit triangular, and lower Hessenberg matrices are defined analogously. Similarly as in the case of symmetric matrices, only one half of the elements of a triangular matrix has to be stored.

- *Upper* or *lower bidiagonal*, and *tridiagonal matrices*: These matrices are diagonal matrices with an extra nonzero diagonal above, or below, or both above and below the main diagonal.

- *Band matrices*: Band matrices have nonzero fill-in in one or more adjacent diagonals (see Figure 141. Diagonal and triangular matrices can be regarded as a special case of band matrices. An example of a general storage schema for band matrices would be storing the nonzero diagonals in a smaller matrix, with one diagonal per row and accessing the elements using an indexing formula. Special types of band matrices are *upper* and *lower band triangular matrices*, *band diagonal matrices*, and *symmetric band matrices*.

- *Toeplitz matrices*: A Toeplitz matrix is a square matrix, where all elements within each of its diagonals are equal. Thus, a Toeplitz matrix requires the same amount of storage as a diagonal matrix.

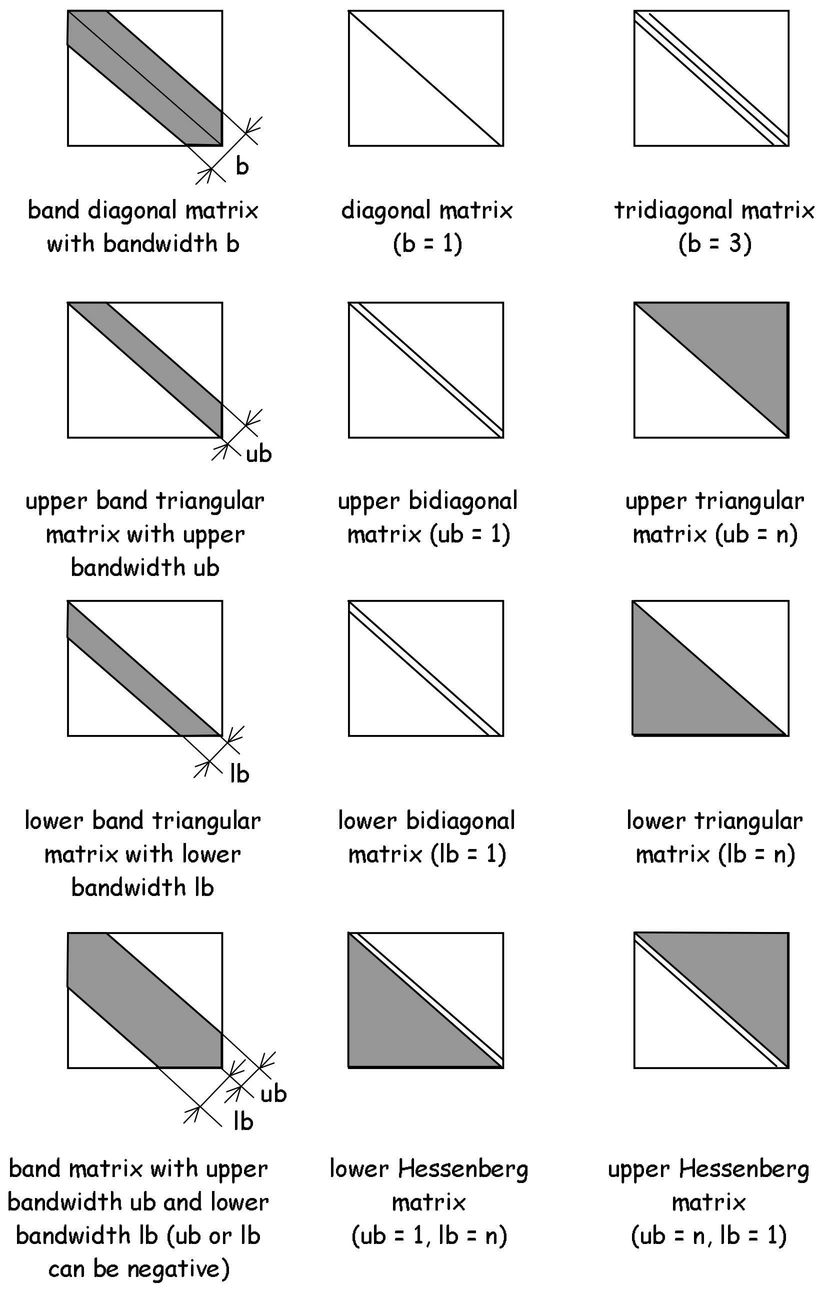
b
band diagonal matrix with bandwidth b
diagonal matrix (b = 1)
tridiagonal matrix (b = 3)
ub
upper band triangular matrix with upper bandwidth ub
upper bidiagonal matrix (ub = 1)
upper triangular matrix (ub = n)
lb
lower band triangular matrix with lower bandwidth lb
lower bidiagonal matrix (lb = 1)
lower triangular matrix (lb = n)
ub
lb
band matrix with upper bandwidth ub and lower bandwidth lb (ub or lb can be negative)
lower Hessenberg matrix (ub = 1, lb = n)
upper Hessenberg matrix (ub = n, lb = 1)

Figure 141 *Examples of n×n band matrices (only the gray region and the shown diagonals may contain nonzeros)*

Some of the above-listed shapes also apply to block matrices, e.g. a block matrix with null matrix entries except for the diagonal entries is referred to as a *block diagonal matrix*. There are also numerous examples of “more exotic”, usually sparse matrix types in the literature, e.g. in [Sch89]: *strip matrix, band matrix with margin (also referred to as a bordered matrix), block diagonal matrix with margin, band matrix with step* (see Figure 142).

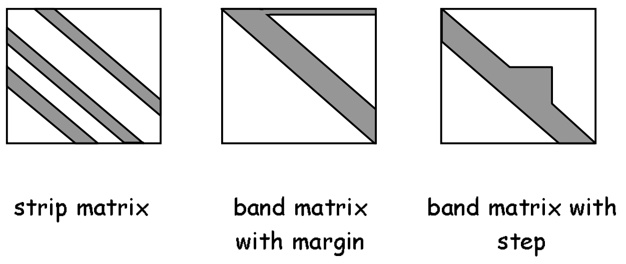

Figure 142 *Some more exotic matrix shapes*

Most matrix computation libraries provide rectangular, symmetric, diagonal, and triangular matrices. Some libraries also provide band matrices (e.g. LAPACK, LAPACK++, ARPACK++, Newmat, LAPACK.h++; see Table 15 and Table 16). Other shapes are less commonly supported.

10.1.2.2.1.3.4 Representation

The elements of a matrix or a vector can be stored in a variety of data structures, e.g. arrays, lists, binary trees, dictionaries (i.e. maps). Each data structure exhibits different performance regarding adding, removing, enumerating, and randomly accessing the elements.

10.1.2.2.1.3.5 Memory Management in Data Structures

The data structures for storing matrix elements may use different memory allocation strategies. We discussed different strategies in Section 9.3.2.2.1. Here, we require at least static and dynamic memory allocation. We extend the representation feature with the *memory allocation* subfeature. The resulting diagram is shown in Figure 143.

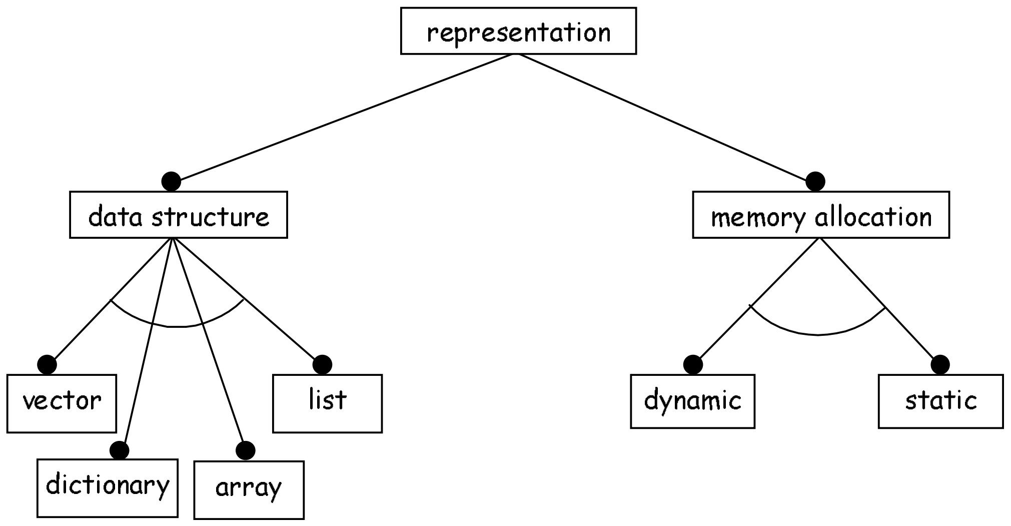

Figure 143 *Representation*

10.1.2.2.1.3.6 Format

Format describes how the elements of a matrix of certain entry type, shape, and density are stored in concrete data structures. For the sake of simplicity, we will further investigate only dense or sparse, point-entry matrices with the most common shapes: rectangular, symmetric, triangular, diagonal, and band. We first describe the common formats for rectangular dense matrices and general sparse matrices and then discuss the special storage and access requirements of other shapes.

10.1.2.2.1.3.6.1 Rectangular Dense Matrices

There are two major formats for storing the elements of a rectangular dense matrix:

1. *row-major* or *C-style format*: In the row-major format, the matrix elements are stored row-wise, i.e. the neighboring elements of one row are also adjacent in memory. This format corresponds to the way arrays are stored in C.

2. *column-major* or *Fortran-style format*: In the column-major format, the matrix elements are stored column-wise, i.e. the neighboring elements of one column are also adjacent in memory. This format corresponds to the array storage convention of Fortran.

Newer matrix computation libraries usually provide both formats (e.g. LINPACK++, TNT). The column-major format is especially useful for interfacing to Fortran libraries.

10.1.2.2.1.3.6.2 General Sparse Matrices

There are several common storage formats for general sparse matrices, i.e. formats that do not assume any specific shape. However, they are also used to represent shaped sparse matrices. The general sparse storage formats include the following:

- *coordinate format* (COO): Only the nonzero matrix elements along with their coordinates are stored. This format is usually implemented using three vectors, one containing the nonzeros and the other two containing their row and the column indices, respectively. Another possibility is to use one array or list with objects, where each of the objects encapsulates a matrix element and its coordinates. Yet another possibility is to use a hash dictionary data structure, where the keys are the coordinates and the values are the nonzeros.

- *compressed sparse column format* (CSC): The nonzeros are stored column-wise, i.e. the nonzeros of a column are stored in the order of their occurrence within the columns. One possibility is to store the columns containing nonzeros in sparse vectors.

- *compressed sparse row format* (CSR): The nonzeros are stored row-wise, i.e. the nonzeros of a row are stored in the order of their occurrence within the rows. One possibility is to store the rows containing nonzeros in sparse vectors.

There are also several other sparse formats including *sparse diagonal* (DIA), *ellpack/itpack* (ELL), *jagged diagonal* (JAD), and *skyline formats* (SKY) and several *block matrix formats* (see [CHL+96]). Table 19 summarizes when to use which sparse format.

Sparse Format	When to Use?
coordinate (COO)	Most flexible data structure when constructing or modifying a sparse matrix.
compressed sparse column (CSC)	Natural data structure for many common matrix operations including matrix multiplication and constructing or solving sparse triangular factors.
compressed sparse row (CSR)	Natural data structure for many common matrix operations including matrix multiplication and constructing or solving sparse triangular factors.
sparse diagonal (DIA)	Particularly useful for matrices coming from finite difference approximations to partial differential equations on uniform grids.
ellpack/itpack (ELL)	Appropriate for finite element or finite volume approximations to partial differential equations where elements are of the same type, but the gridding is irregular.
jagged diagonal (JAD)	Appropriate for matrices which are highly irregular or for a general-purpose matrix multiplication where the properties of the matrix are not known a priori.
skyline (SKY)	Appropriate for band triangular matrices. Particularly well suited for Cholesky or LU decomposition when no pivoting is required. In this case, all fill will occur within the existing nonzero structure.

Table 19 *Choice of sparse format (adapted from [CHL+96])*

10.1.2.2.1.3.6.3 Dependency Between Density, Shape, and Format

The storage and access requirements of dense or sparse, point-entry matrices with the shapes rectangular, symmetric, triangular, diagonal, and band are set out in Table 20.

cki

Structure Type	**Storage and access requirements**
dense rectangular	We store the full matrix in the row- or the column-major dense format.
dense symmetric	We store one half of the matrix, e.g. row-, column-, or diagonal-wise in a dense vector, and use an indexing formula to access the elements. Alternatively, we can store the elements in a full-size two-dimensional array using only half of it. The latter approach needs double as much memory as the first one, but it is faster in access since we do not have to transform the indices. Assigning a value to the element a_{ij} with $i \neq j$, automatically assigns the same value to a_{ji}.
dense triangular	We store the nonzero half of the matrix, e.g. row-, column-, or diagonal-wise in a dense vector. Alternatively, we can store the elements in a two-dimensional array, which requires more space but is faster in access. Reading an element from the other half returns 0 and setting such an element to a value other than 0 results in an error.
dense diagonal	We store only the diagonal in a dense vector. Reading an element off the diagonal returns 0 and setting such an element to a value other than 0 results in an error.
dense band	We store the band only, e.g. diagonal-wise in a dense vector or a smaller two-dimensional array. Alternatively, we can store the elements in a full-size two-dimensional array, which requires more space but is faster in access. Reading an element off the band returns 0 and setting such an element to a value other than 0 results in an error.
sparse rectangular	We store only the nonzero elements in one of the sparse formats, e.g. CSR, CSC, COO, ELL, JAD;
sparse symmetric	We store only one half of the matrix and only the nonzero elements using one of the sparse formats (esp. SKY or DIA). Assigning a value to the element a_{ij} with $i \neq j$, automatically assigns the same value to a_{ji}.
sparse triangular	We use one of the sparse formats (esp. SKY or DIA) to store the nonzero elements. Setting an element in the zero-element half of the matrix to a value other than 0 results in an error.
sparse diagonal	We use one of the sparse matrix formats (esp. DIA) to store the nonzero elements or we store them in a sparse vector. Reading elements off the diagonal returns 0 and assigning a value other than 0 to them causes an error.
sparse band	We use one of the sparse formats (esp. DIA for band diagonal and DIA or SKY for band triangular) to store the nonzero elements. Setting an element off band to a value other than 0 results in an error.

Table 20 *Storage and access requirements of matrices of different structures*

The particular shape of a matrix — especially of a sparse matrix — is application dependent and not all of the possible shapes and formats can be provided by a matrix computation library. Thus, it is important to allow a client program to supply specialized formats.

10.1.2.2.1.3.7 Error Checking: Checking Assignment Validity

Checking the validity of an assignment, e.g. checking whether the value assigned to an element within the zero-element half of a triangular matrix is actually a zero, should be parameterized.

10.1.2.2.1.4 Matrix and Vector Operations

We will consider the following operations on matrices and vectors as parts of a matrix component:

- access operations, i.e. set element and get element and
- basic mathematical operations directly based on access operations, e.g. matrix addition and multiplication.

More complex operations, such as computing the inverse of a matrix or solving triangular systems, will be analyzed together with the algorithm families (e.g. solving linear systems).

The basic operations can be clustered according to their arity and argument types. The unary operations are listed in Table 21. The operation type indicates the input argument type and result type. They are separated by an arrow.

Operation type	**Operations**
vector → scalar	Vector norms, e.g. p-norms (1-norm, 2-norm, etc.)
vector → vector	transposition
matrix → scalar	matrix norms, e.g. Frobenius norm, p-norms determinant
matrix → matrix	transposition

Table 21 *Unary operations*

The binary operations are set out in Table 22. An *update* operation stores the result in one of its input arguments. The definitions of the operations listed in Table 21 and Table 22 can be found in [GL96].

Operation type	Operations
(scalar, vector) $\rightarrow$ vector	scalar-vector multiplication
(scalar, matrix) $\rightarrow$ matrix	scalar-matrix multiplication
(scalar, vector) $\rightarrow$ update vector	saxpy, which is defined as follows $y := ax + y$. where $x, y \in R^n$ and $a \in R$
(vector, vector) $\rightarrow$ vector	vector addition, vector difference, vector multiply (or the Hadamard product)
(vector, vector) $\rightarrow$ scalar	dot product
(vector, vector) $\rightarrow$ matrix	outer product
(vector, vector) $\rightarrow$ update matrix	outer product update, which is defined as follows $A := A + xy^T$, where $x \in R^m$, $y \in R^n$ $A \in R^{m \times n}$
(matrix, vector) $\rightarrow$ vector	matrix-vector multiplication
(matrix, vector) $\rightarrow$ update vector	gaxpy (i.e. generalized saxpy), which is defined as follows $y := Ax + y$, where $x, y \in R^n$ and $A \in R^{m \times n}$
(matrix, matrix) $\rightarrow$ matrix	matrix addition, matrix difference, matrix multiplication

Table 22 *Binary operations*

A standard set of vector and matrix operations is defined in the form of the BLAS (see Section 10.1.1.2.4). The BLAS are a superset of the operations listed in Table 21 and Table 22. Matrix algorithms can be expressed at different levels, e.g. at the level of operations on matrix elements or at the Level-2 or Level-3 BLAS. Level-3 BLAS formulation of an algorithm contains matrix-matrix operations as their smallest operations. This formulation is especially suited for block matrices.

10.1.2.2.1.4.1 Error Checking: Bounds Checking

Invoking the get or the set operation on a matrix or vector with subscripts which are outside the matrix dimensions or vector dimension is an error. Checking for this condition should be parameterized.

10.1.2.2.1.4.2 Error Checking: Checking Argument Compatibility

The vectors and matrices supplied as input arguments to one of the binary operations must have compatible dimensions. For addition and subtraction of vectors and matrices and dot and outer product, the corresponding dimensions of both arguments must be equal. For the matrix-matrix multiplication, the number of columns of the first matrix must be equal to the number of rows in the second matrix. Similarly, the dimension of the vector in a matrix-vector product must be equal to the number of columns of the matrix. Moreover, a determinant can be computed only for square matrices. Checking argument compatibility should be parameterized. If the numbers of rows and columns are available at compile time, the checking should be performed at compile time.

10.1.2.2.1.5 Interaction Between Operations and Structure

The operations on matrices and vectors interact with their structures in various ways:

- There are dependencies between the shape of the arguments and the shape the result of operations.
- There are dependencies between the density of the arguments and the density of the result of operations.
- The implementation algorithms of the matrix operations can be specialized based on the shape to save floating point operations.
- The implementation of an operation's algorithm depends on the underlying representation and format of the arguments, e.g. dense storage provides fast random access. This is not the case with most sparse storage formats.

The following is true about the shape of the result of an operation:

- the result of multiplying a matrix by a scalar is a matrix of the same shape;
- adding, subtracting, or multiplying two lower triangular matrices results in a lower triangular matrix;
- adding, subtracting, or multiplying two upper triangular matrices results in an upper triangular matrix;
- adding or subtracting two symmetric matrices results in a symmetric matrix;
- adding, subtracting, or multiplying two diagonal matrices results in a diagonal matrix.

When we consider rectangular, triangular, and diagonal matrices, the addition, subtraction, or multiplication of two such matrices can potentially produce a matrix whose shape is equal to the shape resulting from superimposing the shapes of the arguments, e.g. rectangular and diagonal matrices yield rectangular matrices and lower diagonal and upper diagonal matrices also yield rectangular matrices, but diagonal and lower triangular matrices yield lower triangular matrices.

Adding, subtracting, or multiplying two dense matrices results — in most cases — in a dense matrix. Adding or subtracting two sparse matrices results in a sparse matrix. Multiplying two sparse matrices can result in a sparse or a dense matrix.

The algorithms of the matrix operations can be specialized based on the shape of the arguments. For example, the multiplication of two lower triangular matrices requires about half the floating point operations needed to multiply two rectangular matrices. Some of the special cases are adding, subtracting, and multiplying two diagonal matrices, two lower or upper matrices, or a diagonal and a triangular matrix, or multiplying a matrix by a null or identity matrix.

10.1.2.2.1.6 Optimizations

In addition to specializing algorithms for different shapes of the argument matrices, we can also optimize whole expressions. For example, more than one adjacent matrix addition operations in an expression should be all performed using one pair of nested loops adding the matrices elementwise without any intermediate results. Thus, this optimization involves the elimination of temporaries and loop fusing. We already described it in Section 9.4.1.

10.1.2.2.1.7 Attributes

An important attribute of a vector is its *dimension* (or *length*), which is the number of elements the vector contains. Since matrices are two dimensional, they have two attributes describing their size: *number of rows* and *number of columns*. For a square matrix, the number of rows and the number of columns are equal. Thus, we need to specify only one number, which is referred to as the *order*. For band matrices, we have to specify the *bandwidth* (i.e. the number of nonzero diagonals; see Figure 141). It should be possible to specify all these attributes statically or dynamically.

10.1.2.2.1.8 Concurrency and Synchronization

Matrix operations are well suited for parallelization (see [GL96,p. 256]). However, parallelization of matrix algorithms constitutes a complex area on its own and we will not further investigate this topic. A simple form of concurrent execution, however, can be achieved using threads, i.e. lightweight processes provided by the operating system. In this case, we have to synchronize the concurrent access to shared data structures, e.g. matrix element containers, matrix attributes. This can be achieved through various locking mechanisms, e.g. semaphores or monitors. We use the locking mechanisms to make all operations of a data structure mutually exclusive and to make the writing operations self exclusive (see Section 7.4.3). In the simplest case, we could provide a matrix synchronization wrapper, which makes get and set methods mutually exclusive and the set method self exclusive.

10.1.2.2.1.9 Persistency

We need to provide methods for storing a matrix instance on a disk in some appropriate format and for restoring it back to main memory.

10.1.2.2.2 Matrix Computation Algorithm Families

During Domain Definition in Section 10.1.1.2.5, we identified the main areas of matrix computations:

- factorizations,
- solving linear systems,
- computing least squares solutions,
- eigenvalue computations, and
- iterative methods.

Each of these areas contain large families of matrix computation algorithms.

As an example, we will discuss the family of factorization algorithms. The discussion focuses on the structure of this family rather than on explaining all the mathematical concepts behind the algorithms. The interested reader will find detailed explanations of these concepts in [GL96].

In general, factorizations decompose matrices into factor matrices with some desired properties by applying a number of transformations. Factorizations are used in nearly all the major areas of matrix computations: solving linear systems of equations, computing least squares solutions, and eigenvalue computations. For example, the LU factorization of a matrix A computes the lower triangular matrix L and the upper triangular matrix U, such that A = L*U. The LU factorization can be used to solve a linear system of the form A*x=b, where A is the coefficient matrix, b is the right-hand side vector, and x is the sought-after solution vector. After factoring A into L and U, solving the system involves solving two triangular systems: L*y=b and U*x=y, which is very simple to do using *forward* or *back substitution*.

In general, we can solve a linear system using either factorizations such as the LU, which are also referred to as *direct methods*, or we can use so-called *iterative methods*. Iterative methods generate series of approximate solutions, which hopefully converge on a single solution. Examples of iterative methods for solving linear systems are Jacobi iterations, Gauss-Seidel iterations, SOR iterations, and the Chebyshev semi-iterative method (see [GL96]).

There are two important categories of factorizations: the *LU family* and *orthogonal factorizations*. Examples of the latter are Singular Value Decomposition (SVD) and the QR factorizations (e.g. Hausholder QR, Givens QR, Fast Givens QR, Gram-Schmidt QR).

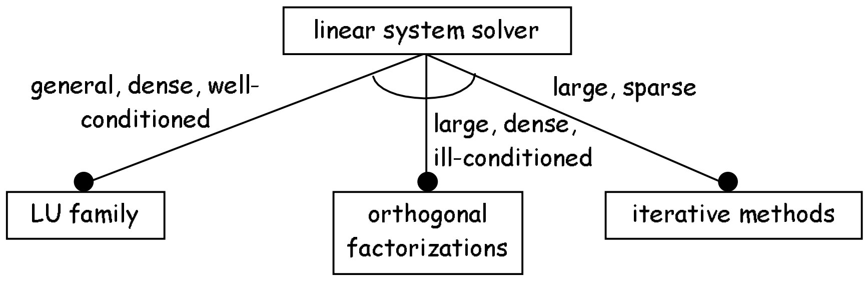
linear system solver
general, dense, well-conditioned
large, dense, ill-conditioned
large, sparse
LU family
orthogonal factorizations
iterative methods

Figure 144 *Approaches to solving linear systems*

Figure 144 explains when to use LU factorizations and when orthogonal factorizations or iterative methods to solve a linear system. Each alternative method is annotated with the properties of the coefficient matrix A as preconditions. These preconditions indicate when a given method is most appropriate. For example, if A is ill-conditioned, LU factorizations should not be used. A is ill-conditioned if it is nearly singular. Whether A is ill-conditioned or not is determined using *condition estimators* (see [GL96] for details).

In the rest of this section, we will concentrate on LU factorizations. The LU factorization in its general form, i.e. *general LU*, corresponds to the Gaussian elimination method. The general LU can be specialized in order to handle systems with special properties more efficiently. For example, if A is square and positive definite, we use the Cholesky factorization, which is a specialization of the general LU.

There are specialized versions of LU factorizations for different matrix shapes, e.g. band matrices or Hessenberg matrices, and for different entry types, i.e. point-entry and block-entry variants (see [GL96]).

An important issue in factorization algorithms is *pivoting*. Conceptually, pivoting involves data movements such as the interchange of two matrix rows (and columns, in some approaches). Gaussian elimination without pivoting fails for a certain class of well-conditioned systems. In this case, we have to use pivoting. However, if pivoting is not necessary, it should be avoided since it degrades performance. We have various pivoting strategies, e.g. no pivoting, partial pivoting, or complete pivoting. Some factorization algorithms have special kinds of pivoting, e.g. symmetric pivoting or diagonal pivoting. In certain cases, e.g. when using the band version of LU factorizations, pivoting destroys the shape of the matrix. This is problematic if we want to factor dense matrices *in place*, i.e. by storing the resulting matrices in the argument matrix. In-place computation is an important optimization technique in matrix computations allowing us to avoid the movement of large amounts of data.

The pivoting code is usually scattered over the base algorithm causing the code tangling problem we discussed in Chapter 7. Thus, pivoting is an example of an aspect in the AOP sense and we need to develop mechanisms for separating the pivoting code from the base algorithm (see e.g. [ILG+97]).

```
ERROR: syntaxerror
OFFENDING COMMAND: --nostringval--

STACK:

0
-0.648
```