Linear Algebra using Extensible Database Technology

Timothy Lord
Abstract

Linear Algebra using Extensible Database Technology

Timothy Lord

Computational models of the real world often involve analyzing discrete points of data logically represented as matrices. Such real-world models are produced in a large number of disciplines, including science and engineering. The calculations on these models can be computationally intensive since the data set is usually large. Nowadays most data is stored in database management systems without the matrix support rich computational models require. In this thesis I describe how I adapted an object-relational database management system to store, model and perform calculations on matrices of varying types.
Contents

1 Literature .......................................... 11
   1.1 Scientific Data Today ............................ 11
   1.2 Databases & Scientific Data ...................... 12
   1.3 UDBL Database Research ......................... 13

2 Amos II ............................................. 14
   2.1 Object-oriented Type System ...................... 15
   2.2 Functions .................................... 16
   2.3 Query Language .................................. 18
   2.4 Extending Amos II ............................... 18
      2.4.1 Data Structures and the Storage Manager .... 19
      2.4.2 Multidirectional Functions and Cost Estimates .. 19
      2.4.3 Rewrite Rules .............................. 20

3 Blas and the Gsl .................................. 22
   3.1 Blas ........................................ 22
   3.2 Gsl Overview .................................. 24
   3.3 Gsl Linear Algebra Operations ................. 25

4 Database Model .................................. 28
   4.1 Matrix Types .................................. 28
   4.2 Multi-directional Functions ..................... 32
   4.3 Cost Estimates ................................ 34
List of Figures

2.1 Number type hierarchy in Amos II ......................... 16
2.2 Comparison of table creation functions ..................... 16
2.3 The Amos II data model .................................... 17

4.1 Triangular matrix type model .................................. 29
4.2 Triangular matrix examples ................................... 30
4.3 Symmetric matrix examples ................................... 30
4.4 Model of symmetric matrix types and their relationship with other matrix types .................................................. 31
4.5 General matrix multiplication query tree ..................... 32
4.6 Different argument and result types for matrix multiplication ............ 33

5.1 The interaction between Amos II and the GSL .................. 38
5.2 Examples of matrix queries and displayed result ................ 41

6.1 Comparison of BLAS implementations on gemm (linear scale) ...... 45
6.2 gemm performance ............................................ 46
6.3 trmm and symm performance .................................. 47
6.4 gemm, trmm and symm comparisons ............................ 48
6.5 LU and Cholesky decomposition comparison .................... 48
6.6 gemm and LU/Cholesky decompositions ........................ 49
6.7 Solving using an LU or a Cholesky matrix ..................... 49
6.8 Solving using LU or Cholesky matrices ........................ 50
Introduction

Large and ever expanding amounts of data are being generated as scientists and engineers attempt to map and model the world and solve problems with greater accuracy. The data must be stored, analyzed, queried, cleaned and compared – just the sorts of things a database management system is good for. A rich semantic model to accompany such data is desirable but hard to implement in traditional relational databases. With the emergence of object-oriented databases in the early 90’s came the promise of modelling such data, but a break from the successful relational model limited their functionality to object storage. More recently however the motivation is to extend the relational model to support objects, the combination of which is called an object-relational database management system, or ORDBMS.

The ability to support storing complex data types relationally is helpful for data produced in a number of fields, not least of which include the scientific, engineering and multimedia fields. Common to all these fields and primed for inclusion in an ORDBMS would be the matrix.

Calculations on matrices are often performed using scientific libraries written in FORTRAN such as LINPACK [1] and LAPACK [2]. Solutions are programmed in FORTRAN or C and/or modelled or written using a matrix-based language like MATLAB or OCTAVE [3]. Such solutions miss out on one or more of the benefits of storing data in a database management system, particularly with respect to maintaining the data and scalability.

There is near universal use of a standard set of basic linear algebra subroutines [4] (BLAS) when working with matrices, and the routines have both hardware and optimized software implementations. The purpose of this thesis then is to describe a matrix model and matrix types and implement operations which use BLAS in an object-relational database.

In the next chapter I discuss some research on object-relational database design,
complex data types with potential for storing matrices and research closely related to this thesis in the Uppsala Database Laboratory (UDBL). In section 2 I describe the object-relational database used in the project and an overview of BLAS is described in section 3. The matrix and related database models are shown in section 4 while in section 5 I go through the implementation. In section 6 I compare my implementation with more traditional methods of performing matrix calculations and finally section 7 summarizes the thesis results along with a discussion on potential further related work.
1 Literature

Scientific, engineering and multimedia applications that do calculations on mathematical data stored in a database usually retrieve that raw “flat” data and manipulate it in a custom application. This is because most database management systems don’t provide support for objects and custom data types. On the flip side, mathematical software like Octave [3] or Matlab [5] natively support matrices but lack the benefits of data management and indexing a traditional DBMS has. The merging of the two is the topic of discussion in this chapter where I look at (a) research on how systems have developed to manipulate mathematical data sets (in section 1.1), (b) research that covers different matrix data structures that relate to scientific data stored in databases (in section 1.2) and (c) Uppsala Database Laboratory research that is closely related to this thesis (in section 1.3).

1.1 Scientific Data Today

Plain-text files

Surprisingly, much scientific data is still stored in plain-text files [6] with specialized libraries for access from general-purpose programming languages [7, 8]. Because of the vast amounts of data generated, just loading the scientific data into a database remains one of the major drawbacks. Some work has been done on integrating data stored in files in the network common (netCDF) and hierarchical (HDF) data formats to allow for manipulation of such data from a DBMS [see 9]. Ramachandran et al. [10] outline an XML-based solution that eliminates the need for scientists in the Earth Science community to write their own specialized data readers but still allows them to retain their standard data exchange formats.
1.2 Databases & Scientific Data

Grids, Arrays, Algebras

With relational database systems having proven their resilience and performance at managing data, it makes sense to extend the relational model for use with scientific data sets. An algebra to manipulate gridfields\(^1\) for scientific data sets is described in Howe and Maier [11]. The research emphasizes grid structures that are irregular, or non-rectilinear grids. In other-words they describe how some of the current solutions to multi-dimensional arrays do not work for data sets that are non-rectilinear.

A study on using object-relational technology for scientific data [8] makes use of a database management system to implement two techniques for representing gridded data: a) through the use of a variable array type and b) via a nested table structure [12].

Research exists that has delved into extending current relational algebra to include manipulation of arrays [see 13]. New algebras for scientific data have been created to deal with inherent problems in the relational model [14].

Object, Object-Relational & Scientific Data

There are numerous research databases experimenting with scientific data sets. The widely used Exodus and its successor Shore provide robust persistent object system which could be used in CAD systems or geographical information systems (GIS) for example. The recently open-sourced RasDaMan stands for “raster data of unlimited extent and any dimension” and is developed from research into raster databases for geographical applications [15] where imaging data is in a raster format. RasDaMan is a DBMS centered around the notion of multidimensional arrays of variable length and dimension and provides a set of operations such as trimming (rectangular cutout) for manipulating the data [16].

A popular research field has been in storing and querying geographical information

\(^{1}\)“A gridfield represents the association of a data set with a grid.” [11]
system (GIS) data in relational databases. An open-source example is PostGIS which extends the PostgreSQL object-relational database [see 17, 18] by giving it geographical primitives and objects.

1.3 UDBL Database Research

My research extends upon and to a certain extent overlaps work at the Uppsala Database Laboratory (UDBL). For example Flodin et al. [19] and Orsborn [20] utilize Amos II multi-directional functions for a finite element analysis package that models matrix types and solves linear equations. Åkerlund et al. [21] also use a matrix model for solving finite differences. Åkerlund et al. [21] notes further work would include “improved representations of some financial and mathematical concepts and operations” and this thesis aims to do that with respect to a matrix data type and basic linear algebra operations (BLAS). None of the work above explicitly exploits matrix structure or optimized BLAS at the core of this work.

Research covered in this chapter has described different ways of storing large amounts of data, research on algebras to handle gridded data (or multidimensional arrays) has been done to help manage and work with such data. Research related to my thesis at the UDBL has not tested the use of the fundamental mathematical software BLAS when comparing with a traditional programming solution (which would normally use BLAS). The following chapter describes Amos II and BLAS is introduced in section 3.
2 Amos II

The Uppsala Database Laboratory (UDBL) develops, maintains and conducts database research using their own database system called Amos II, which stands for Active Mediator Object System. Key features include extensibility and an object-relational design; Amos II can be extended with external functions written in a number of programming languages and new complex data types can be integrated to extend the type system. The system may be run stand-alone, with a query language to interact with the database or it may also be embedded in an application.

The database runs in main-memory, eliminating the bottle-neck imposed by disk access [22, 23]. For this project, Amos II will be used in single-user mode, however one could run it as a mediator [24, 25] where a number of Amos II instances interact to integrate (“mediate”) dispersed data sets.

An overview of aspects of Amos II that are relevant to this project follows:

Section 2.1 describes the object-oriented nature of Amos II’s type system,

Amos II functions are explained in section 2.2,

in section 2.3 AmosQL, Amos II’s query language is introduced and finally

an overview of the different ways one can extend Amos II is detailed in section 2.4.

You may be interested in something more in-depth than is covered here, for further reading I refer you to the Amos II manual [26], some technical reports [27, 28, 29] and other Masters’ theses [30, 31] – most of which are found on the UDBL webpage at http://www.it.uu.se/research/group/udbl/.

---

1 http://www.it.uu.se/research/group/udbl/
2.1 Object-oriented Type System

As the Amos II database system’s name suggests, objects are a fundamental part of the system. As in object-oriented programming, objects in Amos II are instances of data types (more commonly referred to as classes in programming). Everything in the database is an object, though there are two quite different types of object in the system:

**surrogate** A surrogate object is usually defined by a user-defined type (UDT) but may also be a “meta-object” – an object defined by a function definition is an example of a meta-object. Each surrogate object has an associated unique numerical object identifier, or OID. Surrogates are not removed from the system automatically.

**literal** Objects of built-in types, such as numbers and strings are literal objects. These objects are system-maintained and self-describing [28]; they don’t carry any object metadata, such as a unique object identifier (OID) and are garbage-collected automatically by the AStorage subsystem. Consequently they are leaner than surrogate objects.

The Integer and Real types in figure 2.1 are examples of literal types.

The type system of Amos II reflects that of the SQL:99 standard [32] through the use of user-defined type (UDT) definitions like matrix:

CREATE TYPE matrix;

Subtypes

Subtypes, or derived types, may be constructed where the new subtype inherits the properties and behaviour of another. Hierarchies of types are formed through inheritance, an example of which can be seen in the type system for numbers shown in figure 2.1.

The keyword UNDER is used when defining a subtype. A square matrix type would then be defined with: CREATE TYPE square−matrix UNDER matrix. Multiple inheritance is possible by adding a comma-separated list after UNDER. Thus a diagonal matrix type may be defined using:
Fig. 2.1: Number type hierarchy in Amos II

CREATE TABLE array_int ( data INT[] );

(a) SQL

CREATE TYPE array_int PROPERTIES ( data vector OF INTEGER );

(b) AmosQL

Fig. 2.2: Comparison of table creation functions

CREATE TYPE diagonal_matrix UNDER upper_triangular, lower_triangular

2.2 Functions

Functions come in the two flavours: a) stored and b) derived which superficially correspond to SQL attributes and user-defined functions (UDFs) respectively.

Stored A stored function is equivalent to a relational table’s column (or in an OODBMS a class’s attribute) and in this respect Amos II does not follow the SQL standard. A comparison of both methods to create table definitions is shown in figure 2.2. As can be seen in the comparison, stored functions are called properties in Amos II when defining a custom type.

By using PROPERTIES one can define stored functions quickly, the full function definition however would look like the following:

CREATE TYPE array_int ;
Derived Functions usually calculate and return a result which is derived from the function parameters. So Amos II’s derived functions are functions in the conventional programming language sense of the word. A derived function for taking the average of two numbers is such an example:

```
CREATE FUNCTION average (number x, number y) → number
AS
  SELECT (x + y)/2;
```

`average` is a function with scalar arguments, but “table”-based derived functions are also possible in Amos II. Here is `average` again but for a bag of numbers:

```
CREATE FUNCTION average (bag OF number x) → number
AS
  SELECT SUM (x) / COUNT (x);
```

One may see how `average` as defined here is similar to aggregate functions in SQL.

The data model of Amos II and the interaction between functions, types and objects is expressed in figure 2.3.
2.3 Query Language

Amos II has an SQL-like query language, AmosQL, which imitates SQL’s SELECT-FROM-WHERE statements:

```
SELECT (result) FROM (TYPE) WHERE (condition);
```

The following would return $A_{3,4}$ from any matrix in matrices() (a function returning a collection of matrices) with 3 or more rows and 4 or more columns.

```
SELECT $A_{3,4}$ FROM matrix $A$
WHERE $A$ IN matrices()
AND rows($A$) ≥ 3
AND cols($A$) ≥ 4;
```

Amos II is not a true relational system, and so no relational tables exist in the system, but rather one can store collections of objects — as Orsborn et al. [33, §2.2] explain

Notice that terms following the FROM clause denote declarations of typed variables universally quantified over type extents. [28]

In this project, the type of linear algebra equation I want to solve is described in AmosQL as:

```
SELECT $B$
FROM matrix $A$, matrix $B$, matrix $C$
WHERE $A = \text{rand}(50)$
AND $C = \text{rand}(50)$
AND $A \times B = C$;
```

In order to implement the previous example efficiently one would want to extend the Amos II type system and define new multi-directional functions. The next section outlines how Amos II can be extended.

2.4 Extending Amos II

Amos II can be extended in a number of different ways:

• Create functions in C or JAVA and connect to Amos II via the call-out interface.
• Creating C structures and registering them through the AMOS II storage manager, AStorage extends the data model.
• Create LISP functions and/or data types. AMOS II contains a built-in LISP interpreter (ALISP) is integrated into the system. Thus, LISP functions and types can be dynamically loaded, making this method of extending AMOS II quicker to write than the other techniques. The downside is poorer performance for larger amounts of data.

2.4.1 Data Structures and the Storage Manager

The AStorage manual covers AMOS II’s storage manager [34]. The AStorage subsystem is written and used in C and is the only way to extend AMOS II with new complex data types. Thus, if one wants to be able to store and save data in a custom data structure (as I do with matrix [see section 5.1.2]) one must use the storage manager interface. To use the new data type, one must define a logical data object in AMOSQL. It is important to note that to ensure data is stored in the database image upon saving, in creating a data type with the storage manager one must allocate space for the data structure and all data contained in it in a constructor function. This is done in the constructor by way of the new_aligned_object function. We elaborate on the use of this in section 5.

2.4.2 Multidirectional Functions and Cost Estimates

Multidirectional functions in AMOS II gives one the freedom to express queries in a declarative manner\(^2\) and are a defining feature of AMOS II. In a multidirectional function any number of variables are known or unknown (bound or free in AMOS II terminology). For example suppose a function solves for the equation \(\alpha A = B\). A multidirectional function may solve for \(\alpha\), \(A\) or \(B\) where the other two variables are known. The AMOSQL function definition for \(\alpha A = B\) might be:

```
CREATE FUNCTION \(x\) (number \(\alpha\), matrix \(A\)) \(\rightarrow\) matrix \(B\)
AS MULTIDIRECTIONAL
   \(solve_{\alpha A B}\) COST \{2, 1\})
```

\(^2\)Declarative queries lets the programmer write what she wants to accomplish, rather than how to go about it.
(solve_{bf} \text{COST} \{2, 1\})
(\text{gem}_{bf} \text{COST} \{1.5, 1\});

**Costs**  Multidirectional foreign functions are given *costs* to let the query optimizer know the possible computational complexity of the function. In addition to the cost, the size of the result can affect query performance – this is called the *fanout*.

The cost and fanout may be scalar values as in the example above or computed using a separate cost function. A cost function uses the values of the known arguments and the order of all the arguments and computes an appropriate cost and fanout. I do not however use cost functions, as mentioned later this is an interesting feature of Amos II and a possibility for further research.

Cost estimates are important for computationally expensive functions such as for solving systems of linear equations and I use them extensively in my model (see section 4.3).

With the function $\times$ defined above one can now write these declarative queries:

```
SELECT B
FROM matrix B
WHERE 2.4 \times B = \text{rand}(5,5);

SELECT A
FROM matrix A
WHERE 1.6 \times A = \text{rand}(3,4);

SELECT \alpha
FROM number \alpha
WHERE \alpha \times \text{rand}(5) = \text{rand}(5);
```

### 2.4.3 Rewrite Rules

In addition to assigning a cost estimate to a multi-directional function in Amos II one may also assign a rewriter function. Such a function will analyze given parameters and determine whether to use another function or series of functions. Similar functionality exists with PostgreSQL’s `CREATE RULE` which allows one to rewrite SQL statements `SELECT`, `UPDATE`, `INSERT` and `DELETE`. In this thesis we did not
implement rewrite functions and see this as a possible avenue for further work.

This chapter has described key features of the Amos II system. I have compared the type system with SQL:99 and shown how new data types can be defined in a similar fashion to SQL. The query language AmosQL was introduced, and with multidirectional functions I showed an example of how declarative AmosQL queries can be written. The next chapter introduces the other fundamental building block of this project, Blas.
3 Blas and the Gsl

Basic Linear Algebra Subprograms (BLAS) are a set of operations that perform various fundamental operations on vectors and matrices [35, 4, 36]. Originally written in Fortran, the GNU Scientific Library (GSL) has a C implementation which I will use (for convenience) along with another open-source implementation called ATLAS which automatically optimizes for your particular architecture [37]. In addition to this, the GSL contains useful prerequisite data structures for vectors and matrices. The GSL is introduced in section 3.2 and BLAS in section 3.1. An introduction to how BLAS solves systems of equations in section 3.3 completes the chapter.

3.1 Blas

BLAS functionality is organized into three “levels”. The levels relate to computational complexity, for example level 3 operations are $O(n^3)$. The types of operations in each level are categorized as follows:

- **Level 1** Vector operations, such as $\alpha x + y$ and $x^T y$.
- **Level 2** Matrix-vector operations, such as $\alpha A x + \beta y$.
- **Level 3** Matrix-matrix operations, such as $\alpha AB + \beta C$.

The naming scheme for the operations in Level 1 and Level 2 outlines a number of characteristics of the arguments given. The types of matrix (or vector) arguments and the precision are encoded in an operation’s name. Precision and numerical type of matrix/vector elements are encoded in the first letter which can be one of:

- **s** for real single precision

---

1The GSL allows one to “plug-in” different BLAS implementations.
The structure of a matrix argument can be one of:

- **ge** for general (full) matrices
- **tr** for triangular matrices
- **sy** for symmetric matrices

In addition to this, additional arguments specify a) whether a triangular matrix is upper or lower or which part of the matrix is to be referenced for a symmetric matrix, b) whether elements on the diagonal are ones (and therefore not referenced) and c) if the transpose of a matrix should be used.

I limit implemented operations to double precision and real type, so only those operations are listed below. The matrices are described in the matrix model in section 4. In section 5 I explain in detail how the BLAS operations are used.

### Blas Level 1

- **ddot** The scalar product $x \cdot y$ which in matrix notation is $x^T y$ for vectors $x$ and $y$.
- **dnrm2** The Euclidean norm $\|x\|_2 = \sqrt{\sum x_i^2}$ of the vector $x$.
- **dasum** The absolute sum $\sum |x_i|$ of the elements of the vector $x$.
- **damax** The index of the largest element of the vector $x$ determined by its absolute magnitude.

### Blas Level 2

- **dgemv** The matrix-vector product $\alpha A'x + \beta y$, where $A'$ is either $A$ or $A^T$.
- **dsymv** The matrix-vector product $\alpha Ax + \beta y$ for the symmetric matrix $A$. Since the matrix $A$ is symmetric only the upper or lower half of $A$ is referenced (and need be stored).
dtrmv The matrix-vector product $A'x$ for the triangular matrix $A$, with $A'$ either $A$ or its transpose. $A$ may be upper or lower triangular. If the diagonal is unitary you may specify the matrix to be unit triangular.

dtrsv Matrix-vector solve $A'^{-1}x$ for $x$ with $A'$ either $A$ or its transpose and $A$ upper/lower triangular or unit triangular.

Blas Level 3

gemm The matrix-matrix product $\alpha A'B' + \beta C$ where $A'$ is either $A$ or $A^T$ and likewise for $B'$.

symm The matrix-matrix product $\alpha AB + \beta C$ with $A$ symmetric.

trmm The matrix-matrix product $\alpha A'B$ for upper/lower triangular or unit triangular matrix $A$. $A'$ may be $A^T$ or $A$.

trsm Matrix-matrix solve $\alpha A'^{-1}B$ for the matrix $B$. The matrix $A$ is upper/lower triangular or unit triangular and $A'^{-1}$ is either $A$ or $A^T$.

3.2 Gsl Overview

The GNU Scientific Library encompasses a wide range of operations for numerical applications. The GSL is a C application programmer interface (API) for which wrappers can be made for higher-level languages. FFTPACK, LAPACK and QUADPACK and of course BLAS contain public-domain FORTRAN routines some of which the GSL has reimplemented in C. Using the GSL is easier than calling FORTRAN from C.

Non-Blas Operations

For matrices $\mathbf{A}_{m \times n}$ and $\mathbf{B}_{m \times n}$ with $1 < i \leq m$ and $1 < j \leq n$ the GSL implements these operations for manipulating matrices (and vectors) element-by-element:

- **add** $a_{ij} + b_{ij}$
- **sub** $a_{ij} - b_{ij}$
- **mul** $a_{ij} \times b_{ij}$
\[
\begin{align*}
\text{div} & \frac{a_{ij}}{b_{ij}} \\
\text{scale} & \alpha a_{ij} \\
\text{add_constant} & \alpha a_{ij}
\end{align*}
\]

3.3 Gsl Linear Algebra Operations

The Gsl contains operations which assist in solving systems of equations, an example being solving \(AB = C\) for \(B\). By exploiting the characteristics of a matrix one can decompose it and make use of specialized BLAS operations for faster execution times. Suppose \(A\) is square, then one may decompose \(A\) into lower and upper triangular matrices and use \text{trsm} to find \(B\). The alternative – inverting \(A\) – needs more operations and takes longer the larger \(A\) is. Some of the following decompositions are to be implemented in order to take advantage of Amos II multidirectional functions. The operation count of these decompositions will be useful for providing cost estimates for the Amos II functions (examined in section 5.2).

**LU Decomposition** The \(\text{LU}\) decomposition of a square matrix \(A_{n \times n}\) is the product of lower and upper triangular matrices

\[PA = LU\]

Gaussian elimination with partial pivoting is used for the decomposition [38, §13.1] and this algorithm requires \(\frac{2n^3}{3}\) operations [see 39, Algorithm 3.4.1] [40].

**Cholesky decomposition** For a square, symmetric and positive-definite matrix \(A\) there is a decomposition

\[A = LL^T\]

with \(L\) lower triangular. This is a special case of \(\text{LU}\) decomposition with \(L^T = U\). This algorithm is twice as efficient as \(\text{LU}\) decomposition, at \(\frac{n^3}{3}\) operations [see 39, Algorithm 4.2.1].

**QR Decomposition** A general rectangular matrix \(A_{m \times n}\) has a decomposition into the product of an orthogonal square matrix \(Q_{m \times m}\) and an upper triangular
matrix $R_{m \times n}$

$$A = QR$$

The algorithm used is called Householder QR [see 39, Algorithm 5.2.1] and has an operation count of $\frac{2n^3}{3}$ [41].

**Singular Value Decomposition** The singular value decomposition of a general rectangular matrix $A_{m \times n}$ is

$$A = USV^T$$

with $U_{m \times n}$ and $V_{n \times n}$ orthogonal and $S_{n \times n}$ diagonal with singular values.

**Hessenberg Decomposition of Real Matrices** The Hessenberg decomposition is

$$A = UHU^T$$

with $U$ a square unitary matrix and $H$ upper Hessenberg (zeroes below the first sub diagonal). The algorithm used is Householder Reduction to Hessenberg [see 39, Algorithm 7.4.2] which requires $\frac{10n^3}{3}$ flops$^2$.

**Hessenberg Triangular Decomposition** The Hessenberg Triangular decomposition of real matrices is

$$A = UHV^T$$

$$B = URV^T$$

where $U$ and $V$ are orthogonal, $H$ is upper Hessenberg and $R$ is upper triangular. The algorithm used is Hessenberg-Triangular Reduction [see 39, Algorithm 7.7.1] with a flop-count of $8n^3$.

**Bidiagonalization**

$$A = UBV^T$$

with $U_{m \times n}$ and $V_{n \times n}$ orthogonal and $B_{n \times n}$ bidiagonal. The algorithm Householder Bidiagonalization [39, Algorithm 5.4.2] is used and it takes $4mn^2 - \frac{4n^3}{3}$ flops.

---

$^2$A *flop* is a “floating-point operation”
Blas operations have been described in this chapter as well as GSL operations for decomposing matrices. The advantage of decomposing matrices in order to solve linear algebra systems has also been shown. The matrix types used in Blas and GSL operations are explained in more detail in the next chapter, and section 5 details how these operations are used in this project.
4 Database Model

This section describes the matrix model used in the database along with operations for such matrices. A matrix type system is defined, as well as a discussion on operation order decisions by way of Amos II cost estimates and algorithm performance.

4.1 Matrix Types

Briefly alluded to earlier in section 3.1 where I described BLAS operations, the matrix type model in this section allows one to make use of the functionality of BLAS and the GSL. In section 5 I will give details on my implementation of the matrix model.

General matrix type

In BLAS one of the most widely used operations is gemm, which stands for “general matrix multiply”. A general matrix is a full matrix—all elements may be used and manipulated. No assumptions are made as to whether the matrix is symmetrical, triangular or diagonal (although it may be any of these).

I show later on in section 5.1 that general is the base type for all other matrix types.

Square

A square matrix is a general matrix with the same number of rows as columns. Most of the following matrix types have this condition and thus inherit square.
**Triangular**

Triangular matrices are square and have zeros above or below the diagonal. Four types of triangular matrix are defined: upper, lower, unit_upper and unit_lower. Unit triangular matrices’ diagonal elements all equal 1. Figure 4.1 shows the relationships between these types and examples of each are shown in figure 4.2.

**Symmetric**

A symmetric matrix $S$ is a square matrix where all elements $a_{ij}$ above the diagonal equal those $a_{ji}$ below the diagonal. Symmetric matrices equal their transposed selves $S = S^T$. Example of symmetric matrices are in figure 4.3.

**Diagonal**

A diagonal matrix has non-zero values on its main diagonal and zero elsewhere. Because of this it is both lower and upper triangular. It is also symmetric since it is equal to its transpose. The identity matrix is a special diagonal matrix where all the main diagonal elements are unitary - this makes the identity matrix symmetric, upper unit triangular and lower unit triangular. These relationships are shown in figure 4.4 and examples of diagonal and identity matrices are shown in figure 4.3.
### Chapter 4. Database Model

#### Fig. 4.2: Triangular matrix examples

$$
\begin{pmatrix}
40 & 0 & 0 & 0 & 0 \\
51 & 84 & 0 & 0 & 0 \\
52 & 49 & 97 & 0 & 0 \\
53 & 77 & 40 & 89 & 0 \\
35 & 81 & 92 & 07 & 95
\end{pmatrix}
$$

(a) Lower triangular

$$
\begin{pmatrix}
1 & 0 & 0 & 0 & 0 \\
35 & 1 & 0 & 0 & 0 \\
24 & 97 & 1 & 0 & 0 \\
54 & 38 & 76 & 1 & 0 \\
53 & 4 & 44 & 93 & 1
\end{pmatrix}
$$

(b) Lower unit triangular

$$
\begin{pmatrix}
40 & 13 & 11 & 1 & 22 \\
0 & 84 & 61 & 30 & 64 \\
0 & 0 & 97 & 29 & 77 \\
0 & 0 & 0 & 89 & 28 \\
0 & 0 & 0 & 0 & 95
\end{pmatrix}
$$

(c) Upper triangular

$$
\begin{pmatrix}
1 & 9 & 19 & 66 & 89 \\
0 & 1 & 2 & 46 & 06 \\
0 & 0 & 1 & 85 & 27 \\
0 & 0 & 0 & 1 & 67 \\
0 & 0 & 0 & 0 & 1
\end{pmatrix}
$$

(d) Upper unit triangular

#### Fig. 4.3: Symmetric matrix examples

$$
\begin{pmatrix}
1 & 2 & 3 \\
2 & 4 & 5 \\
3 & 5 & 6
\end{pmatrix}
$$

(a) Symmetric

$$
\begin{pmatrix}
5 & 0 & 0 \\
0 & 7 & 0 \\
0 & 0 & 14
\end{pmatrix}
$$

(b) Diagonal

$$
\begin{pmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1
\end{pmatrix}
$$

(c) Identity

#### Fig. 4.3: Symmetric matrix examples
Matrix multiplication on diagonal matrices is especially simple, as it is equivalent to multiplying two vectors together. If $A$ and $B$ are (square) diagonal matrices and their diagonal values are $a_1, ..., a_n$ and $b_1, ..., b_n$ then $A \times B$ has the values $a_1 b_1, ..., a_n b_n$ on its diagonal, zero elsewhere. Diagonal matrices are stored as vectors in the implementation.

**Other matrix types**

Two matrix types not modelled or implemented but useful for solving systems of linear equations are Hessenberg and tridiagonal (or more generally, banded). Solving for these types rather than for the general matrix reduces the number of operations required, but was beyond the scope of this thesis, although the same principles would be applied as for triangular and symmetric matrices.
### 4.2 Multi-directional Functions

I wish to solve equations like $AB = C$ where any one of $A$, $B$ or $C$ is unknown. Thus, three separate implementations are required for these different unknowns. In Amos II the known arguments are said to be *bound* and the unknown said to be *free*. Arguments are annotated with $^b$ or $^f$ respectively. So for matrix-matrix multiplication the implementations are as follows:

- $\times_{bbf}$: $A^b \times B^b = C^f$
- $\times_{bfb}$: $A^b \times B^f = C^b$
- $\times_{fbb}$: $A^f \times B^b = C^b$

The query tree for $\times_{bbf}$ is shown in figure 4.5.

By using the matrix model described in section 4.1 I can build a set of operations for the various argument and result types that are possible. For example, multiplying lower triangular matrices together results in another lower triangular matrix (see figure 4.6a). A selection of these characteristics are listed in figure 4.6. Because the operation counts differ for different matrix types, describing these operations to Amos II and applying “costs” orthogonal to their computational complexity (see section 2.4.2), one can give a richer more nuanced model for the Amos II query processor to optimize on.
(a) If $A$ and $B$ are (unit) lower triangular then $A \times B$ is (unit) lower triangular.

\[( \begin{array}{ccc} a_{11} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{array} ) \begin{pmatrix} b_{11} \\ \vdots \\ b_{m1} \cdots b_{mn} \end{pmatrix} = \begin{pmatrix} c_{11} \\ \vdots \\ c_{m1} \cdots c_{mn} \end{pmatrix} \]

(b) If $A$ and $B$ are (unit) upper triangular then $A \times B$ is (unit) upper triangular

\[( \begin{array}{ccc} a_{11} & \cdots & a_{1n} \\ 0 & \ddots & \vdots \\ 0 & \cdots & a_{mn} \end{array} ) \begin{pmatrix} b_{11} & \cdots & b_{1n} \\ 0 & \ddots & \vdots \\ 0 & \cdots & b_{mn} \end{pmatrix} = \begin{pmatrix} c_{11} & \cdots & c_{1n} \\ 0 & \ddots & \vdots \\ 0 & \cdots & c_{mn} \end{pmatrix} \]

(c) If $A$ and $B$ are symmetric then $A \times B$ is symmetric

\[( \begin{array}{ccc} a_{11} & \cdots & a_{m1} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{array} ) \begin{pmatrix} b_{11} & \cdots & b_{m1} \\ \vdots & \ddots & \vdots \\ b_{m1} & \cdots & b_{mn} \end{pmatrix} = \begin{pmatrix} c_{11} & \cdots & c_{m1} \\ \vdots & \ddots & \vdots \\ c_{m1} & \cdots & c_{mn} \end{pmatrix} \]

(d) If $A$ and $B$ are diagonal then $A \times B$ is diagonal

\[( \begin{array}{ccc} a_{11} & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & a_{mn} \end{array} ) \begin{pmatrix} b_{11} & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & b_{mn} \end{pmatrix} = \begin{pmatrix} c_{11} & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & c_{mn} \end{pmatrix} \]

Fig. 4.6: Different argument and result types for matrix multiplication
By modelling the structure of resulting matrix types after multiplication and estimating the cost of such calculations one can make optimal use of the query processor. The cost estimate model for my project is detailed in the next section.

4.3 Cost Estimates

Cost estimates were introduced in section 2.4.2. This cost-estimate model is based on the performance of algorithms used in BLAS and the GSL. Applying costs assists the Amos II query optimizer in choosing the best route to take in processing the query.

Examples of the computational costs of some square matrix $O(n^3)$ algorithms adapted from Stewart [42, section 3] can be seen in table 4.1. Stewart notes an almost 100-fold difference in operation counts between algorithms for Cholesky decomposition and non-symmetric matrix eigenvalues and eigenvectors. He goes on to make the point that $O$-notation is not an accurate representation of algorithm complexity\(^1\). The cost model here will use the number of flops an operation requires as its basis. The flop counts of algorithms used in this project are described below and have been retrieved from the excellent freely downloadable book Matrix Analysis and Applied Linear Algebra [43, chapter 1].

**Gaussian Elimination Operation Counts**

Gaussian elimination with back substitution applied to an $n \times n$ system requires $\frac{n^3}{3} + n^2 - \frac{n}{3}$ multiplications/divisions and $\frac{n^3}{3} + \frac{n^2}{2} - \frac{5n}{6}$ additions/subtractions. For $n = 20$ the operations count would be 5927. As $n$ increases the dominant fraction for both types of operation is $\frac{n^3}{3}$ and so the algorithm is usually said to have $\frac{2n^3}{3}$ flops.

---

\(^1\)Similarly operation counts may not reflect algorithm performance when the algorithm is optimized for and run on a parallel machine.
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Flops</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cholesky decomposition</td>
<td>2780</td>
</tr>
<tr>
<td>LU decomposition (Gaussian Elimination)</td>
<td>5927</td>
</tr>
<tr>
<td>QR decomposition (Householder triangularization)</td>
<td>11875</td>
</tr>
<tr>
<td>Eigenvalues (symmetric matrix)</td>
<td>13502</td>
</tr>
<tr>
<td>The product $A \times A$</td>
<td>16000</td>
</tr>
<tr>
<td>QR decomposition (explicit $Q$)</td>
<td>28975</td>
</tr>
<tr>
<td>Singular values</td>
<td>33321</td>
</tr>
<tr>
<td>Eigenvalues and eigenvectors (symmetric matrix)</td>
<td>71327</td>
</tr>
<tr>
<td>Eigenvalues (non-symmetric matrix)</td>
<td>143797</td>
</tr>
<tr>
<td>Singular values and vectors</td>
<td>146205</td>
</tr>
<tr>
<td>Eigenvalues and eigenvectors (non-symmetric matrix)</td>
<td>264351</td>
</tr>
</tbody>
</table>

Table 4.1: Operation counts for matrix algorithms: $n = 20$

**Gauss-Jordan operation counts**

Gauss-Jordan applied to matrix $A_{n \times n}$ requires $\frac{n^3}{2} + \frac{n^2}{2}$ multiplications/divisions and $\frac{n^3}{3} - \frac{n}{2}$ additions/subtractions. With $n = 20$ the operation count for $A_{20 \times 20}$ would be 8190. As $n$ increases the dominant fraction for both equations is $\frac{n^3}{2}$ and the algorithm is usually said to have $\frac{5n^3}{6}$ flops.

**Inversion operation counts by reducing with Gauss-Jordan**

$A_{n \times n}^{-1}$ by reducing the augmented matrix $(A | I)$ with Gauss-Jordan requires $n^3$ multiplications/divisions and $n^3 - 2n^2 + n$ additions/subtractions.

Solving a non-singular system $Ax = b$ by first computing $A^{-1}$ and then forming the product $x = A^{-1}b$ requires $n^3 + n^2$ multiplications/divisions and $n^3 - n^2$ additions/subtractions [43, §3.7].

So as $n$ increases the dominant fraction is $n^3$ with the number of flops for the algorithm usually said to be $2n^3$. Note that as compared to computing the inverse,
using Gaussian elimination with back substitution (with \(\frac{2n^3}{3}\) flops) requires a third of the floating-point operations for solving such an equation.

**Operation counts for solving \(Ax = b\) using LU factorization**

Given an LU factorization of \(A\), the number of multiplications/divisions for the equation \(Ax = b\) become \(n^2\) and additions/subtractions become \(n^2 - n\), which is an order of magnitude less than without factorizing. LU factorization is therefore better for solving subsequent systems of equations for different \(b\). Similarly, this means the operation count for solving \(AB = C\) for \(B\) is reduced considerably.

This chapter has described the matrix model (in section 4.1) and shown how it is based on types used by BLAS and the GSL. Multidirectional functions that implement operations on such matrices have been outlined in section 4.2, showing the relationship between function arguments and resultant types. Annotating multidirectional functions with cost estimates is a good way to optimize query performance in Amos II. I introduced a costing model in section 4.3 that is based on the number of floating point operations the BLAS and GSL algorithms perform.

The next chapter describes the project implementation, AmosGSL, and its performance is tested in section 6.
5 Implementation

To implement the matrix model and be able to solve linear algebra systems in AMOSQL I must work with the various abstraction layers of Amos II together with the Gsl. In figure 5.1 one can see the layers of Amos II I have used in my implementation and consequently where the Gsl and Blas operations interact with Amos II.

My implementation involves creating AStorage data types like matrix which make use of equivalent GNU Scientific Library (Gsl) data types like gsl_matrix. Amos II foreign functions (see section 2.4.2) bind the Gsl Blas interface with the Amos II interface. Cost-hinting in Amos II ensures the correct Blas functions are executed and in the correct order.

I use the following tools in the implementation:

- **ATLAS** - Automatically Tuned Linear Algebra Software, an optimized Blas implementation.
- **GSL** - GNU Scientific Library, specifically its:
  - Blas support (which can interface with ATLAS).
  - Gsl matrix/vector data structures and manipulation methods.
  - Linear algebra functions for decompositions and solving using those decompositions.
- **Amos II**:
  - AStorage custom data structures written in C.
  - ALisp functions to define the AMOSQL literal types and to bind data structures to the Amos II data model for use in AMOSQL.
  - AMOSQL multidirectional function definitions for specifying costs.
Fig. 5.1: The interaction between Amos II and the Gsl
5.1 Data structures

New data storage types in Amos II are implemented in C with Alisp\textsuperscript{1} “glue” to integrate it with the Amos II system [27] and to be able to work with new data types in AmosQL. How Amos II can be extended with new storage types was described in section 2.4.1. The two AStorage data types implemented for this project are explained in this section and are good examples of how to define new types using AStorage.

5.1.1 Vectors

Creating a data structure for vectors is useful in implementing Blas Level 1 operations, which mainly manipulate vectors and for Level 2 which is mostly matrix-vector operations (see section 3.1). The AStorage “cell” is a C structure containing the underlying data for the GSL structure gsl_vector along with AStorage fields for the subsystem to manage the cell:

\begin{verbatim}
AGslVectorCell
  tags:  objtags
  vector:  *double
\end{verbatim}

5.1.2 Matrices

Because different matrix types are modell data structure, the AStorage matrix cell, AGslMatrixCell, holds more information than the cell for vectors. One example is to store whether a matrix is transposed or not. The Blas Level 3 gemm operation, among other things, allows you to work on the transpose of the given matrices. The advantage of this is that there is no need to physically change the data on disk, so with my implementation a matrix may for example simply be flagged as transposed without manipulating the actual matrix (a real advantage with large matrices). Another example is when converting a matrix to triangular–one does

\textsuperscript{1}Amos II contains a LISP interpreter which is a subset of CommonLisp [44] - more information on this is contained in [27].
not have to change elements of the matrix since BLAS functions do not reference elements assumed as zero. Other properties define the matrix type and whether it is decomposed:

<table>
<thead>
<tr>
<th>AGslMatrixCell</th>
</tr>
</thead>
<tbody>
<tr>
<td>matrix: *gsl_matrix</td>
</tr>
<tr>
<td>upLo: [CblasUpper, CblasLower]</td>
</tr>
<tr>
<td>trans: [CblasTrans, CblasNoTrans]</td>
</tr>
<tr>
<td>diag: [CblasUnit, CblasNonUnit]</td>
</tr>
<tr>
<td>symm: [AgslSymmetric, AgslDiagonal, AgslNonSymmetric]</td>
</tr>
<tr>
<td>decomp: [AgslNotDecomposed, AgslLU, AgslCholesky]</td>
</tr>
<tr>
<td>perm: *gsl_permutation</td>
</tr>
<tr>
<td>tags: objtags</td>
</tr>
</tbody>
</table>

These properties are manipulated via C macros that act as an interface to the cell and make it easier to think of a cell in terms of matrix types. C macros like IsSymmetric, IsLowerTri and IsGeneral check relevant fields in the cell and analyze the logical matrix type. For example IsLowerTri will return true if the matrix is lower triangular, upper triangular and transposed or diagonal. IsSymmetric will return true when the matrix is annotated as symmetric, diagonal or the identity matrix –that is, when the field symm has a value of either AgslSymmetric or AgslDiagonal.

Because an LU decomposition of a matrix can be stored in one gsl_matrix the same ASTORAGE cell is used for decomposed matrices as for non-decomposed. This is the reason for the perm field (since permuted A is equal to LU – that is, PA = LU).

Because the “logical” matrix model is so rich and because one wants to avoid unnecessary data access, operations on matrices are not simple. For example, to print an upper triangular matrix, one should show zeros in the lower triangle (regardless of what is in that triangle), or for a transposed upper triangular matrix it must show as lower triangular. On the other hand, there are half as many data references as for general matrices.

To illustrate this consider what happens to a matrix that is made lower triangular and then transposed and the difference between what is displayed and what really
exists at each element of the matrix. The AmoSQL query for this is:

\[
\text{SELECT } L \\
\text{FROM matrix } A, \text{ matrix } L \\
\text{WHERE } A = \text{rand}(5) \\
\quad \text{AND } L = \text{trans}(\text{tril}(A));
\]

which does the following:

1. Create general matrix $A_{5 \times 5}$, which contains random elements.
2. Create $L$ as tril applied to a copy of $A$. tril sets the matrix type as lower triangular (specifically, sets upLo to CblasLower).
3. Set $L$ as transposed (set trans to CblasTrans).
4. Finally SELECT prints the matrix.

A couple of examples of how matrices are handled throughout such operations are shown in figure 5.2.
CHAPTER 5. IMPLEMENTATION

5.2 Matrix multiplication

Now that the logical matrix types have been implemented, one would like to exploit the type-specific operations possible in BLAS. This is implemented using AmosQL function definitions (introduced in section 2.4.2) by overloading the $\times$ operator:

\[
\begin{align*}
\text{CREATE FUNCTION } &\times(\text{matrix } A, \text{ matrix } B) \rightarrow \text{matrix } C \\
&\text{AS } \text{gemm } (A, B) \\
\text{CREATE FUNCTION } &\times(\text{triangular } T, \text{ matrix } B) \rightarrow \text{matrix } L \\
&\text{AS } \text{trmm } (T, B) \\
\text{CREATE FUNCTION } &\times(\text{symmetric } S, \text{ matrix } B) \rightarrow \text{matrix } C \\
&\text{AS } \text{symm } (S, B)
\end{align*}
\]

Actually that’s not the whole story – since the implemented functions are multidirectional I can include the case for when $B$ is unknown, ending up with the following function definitions:

\[
\begin{align*}
\text{CREATE FUNCTION } &\times(\text{matrix } A, \text{ matrix } B) \rightarrow \text{matrix } C \\
&\text{AS } \text{MULTIDIRECTIONAL} \\
&\{ \text{gemmbbf } \text{COST } \{10, 1\} \\
&\text{gmmsvxfbb } \text{COST } \{50, 1\} \\
&\text{gmmsvx_fbb } \text{COST } \{50, 1\}\};
\end{align*}
\]

\[
\begin{align*}
\text{CREATE FUNCTION } &\times(\text{triangular } T, \text{ matrix } B) \rightarrow \text{matrix } C \\
&\text{AS } \text{MULTIDIRECTIONAL} \\
&\{ \text{trmmmbf } \text{COST } \{5, 1\} \\
&\text{tmmsvxfbb } \text{COST } \{50, 1\} \\
&\text{tmmsvx_fbb } \text{COST } \{35, 1\}\};
\end{align*}
\]

\[
\begin{align*}
\text{CREATE FUNCTION } &\times(\text{symmetric } S, \text{ matrix } B) \rightarrow \text{matrix } C \\
&\text{AS } \text{MULTIDIRECTIONAL} \\
&\{ \text{symmbbf } \text{COST } \{5, 1\} \\
&\text{smmsvxfbb } \text{COST } \{50, 1\} \\
&\text{smmsvx_fbb } \text{COST } \{35, 1\}\};
\end{align*}
\]

Although the type information is stored in the matrix cell, the costs can only be specified in AmosQL, hence the reason behind explicitly defining the functions above\(^2\).

\(^2\)Cost functions which have not been implemented here may provide a cleaner interface for this design.
In addition to the functions for different matrix types above, I can also define functions for when a matrix is decomposed:

```
CREATE FUNCTION \times (ludecomp \text{ LU}, \text{ matrix } B) \rightarrow \text{ matrix } C
\text{ AS MULTIDIRECTIONAL }
(\text{lummbff} \text{ COST } \{15, 1\})
(\text{lumsvx}_bfb \text{ COST } \{50, 1\})
(\text{lumsvx}_fbf \text{ COST } \{25, 1\});
```

```
CREATE FUNCTION \times (choldecomp \text{ CHOL}, \text{ matrix } B) \rightarrow \text{ matrix } C
\text{ AS MULTIDIRECTIONAL }
(\text{chmmmbff} \text{ COST } \{5, 1\})
(\text{chmmsvx}_bfb \text{ COST } \{50, 1\})
(\text{chmmsvx}_fbf \text{ COST } \{10, 1\});
```

The implementation has been explained in this chapter, showing the matrix and cost models as they are defined in Amos II. Multidirectional functions, together with their costs, which are based on flops as shown in section 4.3, were detailed. The following chapter compares the performance of these functions.
6 Performance

Performance of a number of matrix operations are shown in this chapter. CPU clock cycles were timed over a number of iterations and the average was taken. The test machine was an Asus EEE PC 1000H running the Linux kernel 2.6.28-8-netbook-eeepe. An initial comparison between the two BLAS implementations, ATLAS and the C reference implementation\(^1\) is provided, after which ATLAS is used exclusively.

The CPU times for three different execution methods were compared:

- **AmosQL** Clocked and timed in C via the AMOS II call-in interface.
- **AmosGSL** AMOS II foreign functions written in C.
- **Gsl** Direct GSL.

6.1 Matrix multiplication

Tests for the operation \( \mathbf{A}_{n \times n} \times \mathbf{B}_{n \times n} \) for different types of \( \mathbf{A} \) and for different \( n \) ranging from 2 to 1024 \((n = 2^i, i = 1, 2, ..., 10)\) were timed for the BLAS functions\(^2\): `gemm`, `trmm` and `symm`. The given matrix \( \mathbf{A} \) is logically general, lower triangular and symmetric respectively.

**Atlas and cblas**

`gemm` performance between two different BLAS implementations was compared. The GSL comes with the C reference implementation of BLAS which is not optimized. I

---

\(^1\)The reference implementation is distributed with the GSL and is available on the BLAS website [http://www.netlib.org/blas](http://www.netlib.org/blas). ATLAS which stands for *Automatically Tuned Linear Algebra Software*, is available via [http://math-atlas.sourceforge.net](http://math-atlas.sourceforge.net).

\(^2\)These were explained in section 3.
Fig. 6.1: Comparison of BLAS implementations on gemm (linear scale)

tested this with ATLAS and as can be seen in figure 6.1, performance is substantially better. Because of this, the rest of the tests only use ATLAS.

**AmosQL, AmosGSL and Gsl**

Figure 6.2 shows the performance on gemm between AMOSQL, AMOSGSL and GSL. Note how AMOSQL starts relatively high but the difference is made up by when \( n = 1024 \) (the figure shows this more dramatically because of its logarithmic scale). The CPU times are shown in Table 6.1.

AMOSGSL and GSL are too close to differentiate, since AMOSGSL has little overhead (two matrices of type AGslMatrixCell that each contain gsl_matrix pointer will not be affected by the size of the matrices).

Similarly, one can see in figure 6.3a and figure 6.3b how the AMOSQL CPU time converges with AMOSGSL and GSL implementations as the size \( n \) gets larger – the
Table 6.1: CPU times for gemm

<table>
<thead>
<tr>
<th>n</th>
<th>AmosQL</th>
<th>AmosGSL</th>
<th>Gsl</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.00748</td>
<td>0.00001</td>
<td>0.00001</td>
</tr>
<tr>
<td>4</td>
<td>0.00748</td>
<td>0.00001</td>
<td>0.00001</td>
</tr>
<tr>
<td>8</td>
<td>0.00745</td>
<td>0.00003</td>
<td>0.00003</td>
</tr>
<tr>
<td>16</td>
<td>0.0076</td>
<td>0.00011</td>
<td>0.0001</td>
</tr>
<tr>
<td>32</td>
<td>0.00788</td>
<td>0.00044</td>
<td>0.00043</td>
</tr>
<tr>
<td>64</td>
<td>0.01315</td>
<td>0.00373</td>
<td>0.003</td>
</tr>
<tr>
<td>128</td>
<td>0.02181</td>
<td>0.01352</td>
<td>0.01324</td>
</tr>
<tr>
<td>256</td>
<td>0.09909</td>
<td>0.09271</td>
<td>0.08917</td>
</tr>
<tr>
<td>512</td>
<td>0.65375</td>
<td>0.64625</td>
<td>0.6375</td>
</tr>
<tr>
<td>1024</td>
<td>4.795</td>
<td>4.8425</td>
<td>4.8025</td>
</tr>
</tbody>
</table>

Fig. 6.2: gemm performance
inference here being that AmosQL takes some time to setup (around 8ms of CPU

time in these tests). Note that there is no difference between GSL and AmosGSL
(executed directly in C) performance, which makes sense considering that the main
computations uses the same underlying functions (trmm and symm).

6.1.1 Comparing gemm, trmm and symm

Somewhat more interesting is to compare gemm, trmm and symm – this is plotted
in figure 6.4a. gemm is concealed by symm in the figure because their numbers
are so similar. This is because there are twice as many multiplications that must
be performed in the symmetric case as in the triangular case. A linear plot of the
two specialized multiplication operations shown in figure 6.4b clearly shows the
performance advantage of trmm as $n$ increases.

6.2 Matrix decomposition

The two decompositions implemented in this project, LU and Cholesky were tested
with $n$-by-$n$ matrices, also where $n$ is between 2 and 1024. The results are compared
against each other and with gemm in this section.


**Chapter 6. Performance**

---

![Graph showing performance comparisons for gemm, trmm, and symm operations.](image-a)

![Graph showing performance comparisons for LU and Cholesky algorithms.](image-b)

---

**LU and Cholesky Algorithms**

The two algorithms are compared on a logarithmic scale in figure 6.5a, where Cholesky is the faster of the two. This can be seen clearly on a linear scale (see figure 6.5b) as the size of the matrix increases. This makes sense because LU decomposition is more expensive (but covers a broader set of matrix types) and Cholesky is a specialized algorithm since it works only with positive definite matrices. figure 6.6a shows that LU decomposition is as computationally intensive as general matrix multiplication. The benefit of decomposing comes when one wants to solve a system of linear equations with multiple right-hand sides, which is covered next.
6.3 Solving systems of linear equations with multiple right-hand sides

Tests for solving $A \times B = C$ for $B$ were performed for the following situations:

1. $A$ and $C$ are (square) general matrices,
2. $A$ is LU decomposed or
3. $A$ is Cholesky decomposed.

The case in which $A$ is not decomposed is solved naively by inverting $A$ ($B = A^{-1}C$) and can be seen in figure 6.7 taking a long time to compute.
Fig. 6.8: Solving using LU or Cholesky matrices

Finally, a comparison between LU/Cholesky decomposition and solving is shown in figure 6.8 and shows the differences between their CPU times relative to one another.

The performance of implemented algorithms were checked and compared in this section. ATLAS was shown to be superior to the GSL reference implementation and reveals the importance of using BLAS algorithms tuned for the computer hardware it will be run on (see figure 6.1). A small overhead of 8ms of CPU time was seen when using AMOSGSL through AMOSQL, but this is probably start-up time and becomes insignificant as matrix sizes increase (see figure 6.2 for a clear example). When comparing the three BLAS Level 3 routines, no difference between \texttt{gemm} and \texttt{symm} is noticeable but figure 6.4b shows that \texttt{trmm} is markedly faster.
Finally, figure 6.8 displays the CPU times of solving when a matrix is decomposed versus a naive implementation. The benefit of using a decomposed matrix is clearly seen.

Concluding remarks follow, along with a look at possible further work.
7 Conclusions

I have adapted a database system with an interface to BLAS and the GSL and in doing a number of benefits arise:

**Declarative Programming** AmosGSL queries can construct a mathematical operation by declaring what one wants to achieve, as I have shown in section 2.4.2. An AmosQL query such as:

```sql
SELECT B
FROM A, B, C
WHERE A = rand(125)
AND C = rand(125)
AND A \times B = C
```

is arguably more intuitive than in C\(^1\).

**Cost-based processing** By knowing in advance the potential cost of algorithms involved in an ad-hoc query, the database management system can choose the optimal execution path. The Amos II system provides for this through multidirectional function definitions (described in section 2.4.2) and my implementation makes use of them for matrix multiplication as was detailed in section 5.2.

The benefit of having matrix support in a database is good for the science and engineering fields especially, since their data is often represented as such. Using the efficient and widely supported BLAS standard for calculations involving matrices means the performance will hold for very large matrices. The performance tests in section 6 show that using specialized algorithms for different matrix types makes

---

\(^1\)Although certainly an array based language like Octave is simpler: `rand(125) \ rand(125)`.

Actually this could be possible in AmosQL by implementing a backslash operator.
sense from a computational perspective. The utilization of an architecture-optimized BLAS implementation was also shown to improve performance.

7.1 Further Work

Only double-precision real numbers were used in this thesis. Further work could add support for integers, single-precision real numbers and single and double precision complex numbers. The latter would be particularly useful for engineering applications and would involve implementing BLAS functions that operate on complex matrices.

Finite element analysis data normally comes in the form of large matrices and avoiding unnecessary transfers is a top priority. Loading such data into memory generally becomes prohibitive, so integrating stream processing capabilities is essential. Similarly, very large systems would require a general sparse matrix representation. With a sparse matrix representation it follows that sparse BLAS be integrated to cater for it.

Cost-estimates in this thesis only used simple scalar values but to more accurately optimize queries one would make use of cost-estimate functions.

Rewrite rules are a powerful facility of database management systems in general (and Amos II in particular) that were not covered in this thesis. A full implementation would rewrite queries to further optimize them.
Bibliography


