A Numeric Library for Use in Modelica Simulations with Lapack, SuperLU, Interpolation and MatrixIO

Anders Sandholm\(^\natural,\sharp,\ast\), Peter Bunus\(^\natural,\dagger\), Peter Fritzson\(^\natural,\ddagger\)

\(\natural\) PELAB - Programming Environment Lab
Dept. Computer Science, Linköping University
S-581 83 Linköping, Sweden

\(\sharp\) eHealth Institute
Dept. Health and Behavioural Sciences, University of Kalmar
Kalmar, Sweden

Abstract

This paper introduces a numerical Modelica library that provides access to some of the most well-known powerful libraries for numerical methods. Our approach has been to develop wrappers that allow Modelica users easy access as functions both from textual and graphical Modelica environments [8], [9]. This library also includes additional external functions with corresponding Modelica wrappers to interpolate data and to read/write matrix data from/to files.

Keywords: Matrix, Lapack, SuperLU, Matrix Market File Format, Harwell-Boeing Matrix Format, Interpolation

1 Introduction

One important area of research is developing and implementing fast numerical methods that can be used to simulate physical phenomena. Researchers who are working with simulation usually do not want to spend time and resources implementing, debugging, and maintaining new numerical libraries. Instead they want to use existing libraries that are recognized as stable and efficient.

Numerical methods can be divided into different areas such as: optimization, solution of ordinary and partial differential equations, mesh generation, numerical integration, solution of nonlinear equations, solution of linear equations, eigenvalue problems, curve and surface fitting, interpolation, etc. Finite element methods is a well-known group of methods for solving PDE problems, which typically are rather computation intensive.

This paper introduced a new Numeric wrapper library for Modelica users that want to use standard common numeric libraries as well as methods and routines for saving and loading matrices to/from files.

1.1 Small Example of Using the Library

Assume that the user wants to calculate the eigenvalues for an N-by-N real nonsymmetric matrix stored in the Matrix Market file format. The first task would be to load the matrix file, here called matrix.mtx. This is done by using the functions getMatrixSize and getMatrixFile where the first one return the size of the matrix and the other one return the matrix data, both takes the file name as a string argument. Functions for loading and saving matrices in Matrix Market is located in package Numeric.MatrixIO.MatrixMarket along with other Matrix Market functions.

Bellow Modelica pseudo code is shown for loading the matrix.

```
Integer n = getMatrixSize("matrix.mtx");
Real A[n,n];
A = getMatrix("matrix.mtx");
```

More information about loading and saving data can be found in the MatrixIO section. For the calculation of the eigenvalues Lapack [2] contain a function dgeev that calculates the eigenvalues along with the left and right eigenvectors of a general matrix. The dgeev uses...
double precision but the Lapack library also contains function for single precision calculation, named sgeev. In the library outlined in this paper all Modelica wrapper functions for Lapack is stored in subpackages, the wrapper for the dgeev function is located in Numeric.Lapack.SimpleDriver, for further detail see the section dealing with the structure of the library. Bellow Modelica pseudo code is shown that outlines the call to the calcEigenValGeneralMatrix_dgeev which uses the lapack dgeev function for the calculations of the eigenvalues.

\[
\begin{align*}
\text{Real eigenvReal}[\text{size}(A, 1)]; \\
\text{Real eigenvImag}[\text{size}(A, 1)]; \\
\text{Real eigenVectors}[n,n]; \\
\text{(eigenvReal, eigenvImag, eigenVectors)} = \text{calcEigenValGeneralMatrix_dgeev}(A_1);
\end{align*}
\]

The Modelica wrapper function calcEigenValGeneralMatrix_dgeev allows the user to specify more indata and receive more information from LAPACK then is shown here, this is further outlined in the Lapack section.

2 Structure of the Numeric Library

The design of this library focuses on two major issues:

- It should be easy to locate libraries and functions
- The package should be easy to maintain with all the external library dependencies
- Package structure shall allow easy addition of new external libraries and native Modelica functions

This library contains both functions that are implemented natively in Modelica and functions that acts as wrappers to C and FORTRAN 77 functions [8],[1]. The top structure can be seen in figure 1 with the subpackages Lapack, SuperLU, MatrixIO, and Interpolation

2.1 Structure of Main Package

The subpackages Lapack and SuperLU contain Modelica wrapper functions that calls the external function in each corresponding external library. The MatrixIO subpackage is divided into subpackages that implement different matrix file formats for saving and loading matrix data. The Interpolation subpackage contains subpackages with methods both developed natively in Modelica code but also Modelica wrapper functions to interpolation libraries.

2.2 Structure of Lapack Subpackage

The Lapack subpackage can be seen in figure 2. This package contains four subpackages, SimpleDriver, ExpertDriver, ComputationalDriver and Examples. For more information about SimpleDriver, ExpertDriver and ComputationalDriver se the Lapack section. In the examples library different examples has been implemented that explain how the Lapack package can be used in Modelica code. Theses examples is mostly constructed for users that are common with the Modelica language but is new to the Lapack library.

2.3 Structure of SuperLU package

The SuperLU package has been divided up in its library sections, Driver, Computation, Utility and a section Examples has been added. The packaged structure can be view in figure 3. For detail information...
about the Driver, Computation and Utility packages see the SuperLU section. In the Example section different Modelica examples has been implemented that show how the SuperLU library can be used.

2.4 Structure of MatrixIO package

The MatrixIO packages implement support for different matrix file formats. Currently the Matrix-Market and the Harwell-Boeing packages are supported with functions for saving and loading dense and sparse matrix data. An overview of the MatrixIO package can be viewed in figure 4. For more detail information about the Matrix Market and the Harwell-Boeing see each sections. Examples that show how matrix data can be loaded and saved are implemented in the Examples subpackage.

2.5 Structure of Interpolation package

The Interpolation package is designed with the same idea as the other packages. Currently the package are divided into two subpackages, CubicSpline and Examples, see figure 5. The CubicSpline subpackage contains both native Modelica implementation and Modelica wrapper function for use of external cubic spline function implemented in C code. The Examples subpackage contains easy understandable examples that show both how the Modelica implemented version and the external version can be called from Modelica code.

3 Library Design Issues

As already mentioned, the main idea is to create a package where different numerical methods, format handling functions, and solvers can be readily available for use from Modelica. Several design issues has been addressed on how to handle documentation from the external libraries, variable name in the external functions. Without the library documentation the package would be hard to use and a user that are familiar with the packaged will be confused if the input/output variable has changed name in the Modelica wrapper.

3.1 Name convention

The Modelica Numeric library uses function and variable name from the original package as a postfix notation along with a more explanatory Java-style name. This will give new user more understanding of functions and variable, without reading the documentation for each variable. Users that is familiar with the libraries will recognize functions and variable due to the postfix notation.

An example is the Modelica wrapper function calcEigenValGeneralMatrix_dgeev which is introduced in the Introduction part of this paper. The first part of the functions name tells the user that it calculates the eigenvalues for a general matrix and the postfix notation specify that the dgeev function is used. The same naming convention are used for variables. The dgeev function has a variable named JOBVL that specify it the left eigenvalues should be calculation or not. In the Modelica wrapper function this variable is named calcLeftEigenV_JOBVL which are a more self emplained java based name along with the Lapack variable name as a postfix.

3.2 Documentation

The issue about documentation was addressed by including the external functions documentation to the Modelica wrapper functions documentation node. Bellow the first part of the documentation for the Modelica wrapper function calcEigenValGeneralMatrix_dgeev is shown.

For further details about cubic spline see the Interpolation section.
different between the native function call and the Modelica wrapper function call. In the Modelica wrapper function the LDA, LDVL and LDVR variables are not needed, and therefore has been removed from the Modelica interface. After the library annotation the fortran function declaration follows along with version and argument documentation. Further down comes the purpose and argument documentation, here only four arguments are shown.

```
annotation( Documentation(info="Lapack

#### Numerical Library annotation ###

Variables that has been excluded in Numerical Library

LDA = size(A,1);
LDVL = size(A,1);
LDVR = size(A,1);

SUBROUTINE DGEEV( JOBVL, JOBVR, N, A, LDA, WR, WI, VL, LDVL, VR, LDVR, WORK, LWORK, INFO )

-- LAPACK driver routine (version 3.0)
Univ. of Tennessee, Univ.
 of California Berkeley, NAG Ltd.,
 Courant Institute, Argonne National
 Lab,and Rice University
 December 8, 1999

.. Scalar Arguments ..
CHARACTER JOBVL, JOBVR
INTEGER INFO, LDA, LDVL, LDVR,
LWORK, N

.. Array Arguments ..
DOUBLE PRECISION A( LDA, * ),
VL( LDVL, * ), VR( LDVR, * ),
WI( * ), WORK( * ), WR( * )

Purpose
=======

DGEEV computes for an N-by-N real
nonsymmetric matrix A, the
eigenvalues and, optionally,
the left and/or right eigenvectors.

The right eigenvector v(j) of A satisfies
A * v(j) = lambda(j) * v(j)
where lambda(j) is its eigenvalue.
The left eigenvector u(j) of A satisfies
u(j)**H * A = lambda(j) * u(j)**H
where u(j)**H denotes the conjugate of u(j).

The computed eigenvectors are
normalized to have Euclidean norm
equal to 1 and largest component real.

Arguments
========

JOBVL (input) CHARACTER*1
= ’N’: left eigenvectors of
 A are not computed;
= ’V’: left eigenvectors of
 A are computed.

JOBVR (input) CHARACTER*1
= ’N’: right eigenvectors of
 A are not computed;
= ’V’: right eigenvectors of
 A are computed.

N (input) INTEGER
The order of the matrix A. N >= 0.

A (input/output) DOUBLE PRECISION
array, dimension (LDA,N)
On entry, the N-by-N matrix A.
On exit, A has been overwritten.

4 Lapack

Lapack is one of the most widely used libraries for
solving many common problems in numerical linear
algebra. The library includes routines for solving
system of simultaneous linear equations, finding
least square solutions of overdetermined systems of equations, solving eigenvalue problems and solving singular value problems [2]. The Modelica Numeric.Lapack sublibrary is divided into three different parts: Basic Routines, Advanced Routines and Computational Routines.

- Basic Routines solves a specified problem with a few options, example of functionality in basic routines are finding the eigenvalues of a matrix or solving a set of linear equations.
- Advanced Routines allows the user to control the calculations more by taking more options and returning more information than the simple driver routines. An example can be calculation of error bounds or equilibrating matrices to improve accuracy.
Computational Routines shall more be seen as routines design to perform a specific task, such as a LU factorization or reduction of a real system matrix to tridiagonal form. Usually these function is used to construct more advanced functions in the Basic and Advanced routines libraries. The routines are categorized in system of linear equations, eigenvalue problems, orthogonal factorization and singular value decomposition.

4.1 Example
An example for the simple driver routines is the dgeev function that calculates right and left eigenvalues and eigenvectors for a N-by-N real nonsymmetric matrix. This calculation can be described as finding the eigenvalues λ and corresponding eigenvectors \( \mathbf{z} \neq \mathbf{0} \) as equation (1) and (2) describe.

\[
\begin{align*}
\mathbf{A}\mathbf{z} &= \lambda \mathbf{z} \\
\mathbf{A} &= \mathbf{A}^T \quad \text{where A is real}
\end{align*}
\]

When all eigenvalues and eigenvectors has been calculated equation (3) are determined.

\[
\mathbf{A} = \mathbf{Z} \mathbf{\Lambda} \mathbf{Z}^T
\]

Where \( \mathbf{A} \) is a diagonal matrix whose diagonal elements are the eigenvalues, \( \mathbf{Z} \) is an orthogonal matrix whose columns are the eigenvectors. As described previously the Modelica wrapper function for the dgeev is called calcEigenValGeneralMatrix_dgeev and is shown bellow, the documentation node has been removed in this example.

```modelica
function calcEigenValGeneralMatrix_dgeev

input Real A[:, size(A, 1)];
input String calcLeftEigenV_JOBVL = "N" "Left eigenvectors of A are not computed";
input String calcRighEigenV_JOBVR = "V" "Right eigenvectors of A are computed";
output Real eigenReal_WR[size(A, 1)] "Real part of eigenvalues";
output Real eigenImag_WI[size(A, 1)] "Imaginary part of eigenvalues";
output Real leftEigenVectors_VL [size(A, 1), size(A, 1)] "Left Eigenvectors";
output Real reightEigenVectors_VR [size(A,1), size(A,1)] "Right Eigenvectors";
protected
Integer N=size(A, 1) "The order of the matrix";
Integer LWORK=10*N "MAX size if JOBVL = V or JOBVR = V LWORK >= 4*N";
Real WORK[LWORK];

external "Fortran 77" dgeev( calcLeftEigenV_JOBVL, calcRighEigenV_JOBVR, N, A, N, eigenReal_WR, eigenImag_WI, leftEigenVectors_VL, N, reightEigenVectors_VR, N, WORK, LWORK, INFO)
annotation (Library="lapack");
end calcEigenValGeneralMatrix_dgeev;
```

The first argument is the Matrix \( \mathbf{A} \) which the eigenvalues and eigenvectors are to be calculated for. The following two arguments, calcLeftEigenV_JOBVL and calcRighEigenV_JOBVR, determine if the right or/and left eigenvalues/eigenvectors are to be calculated. In the default setting only the right eigenvalues are calculated.

In the output section the eigenvalues variable comes first then left and right eigenvectors and last an information flag that tells if the calculation could be performed.

Variables that don’t add to the functionality of the Modelica wrapper function but is need for the Lapack implementation has been placed in the protection section. For the function outlined above the working variables LWORK and WORK has been placed here, along with the variable N that specify the order of the matrix.

5 SuperLU
For solving large, sparse, nonsymmetrical systems of linear equations the SuperLU library is commonly used [10]. The SuperLU library can be called either from C or from Fortran code where our Modelica implementation uses the Fortran interface for maximum performance. The SuperLU library starts by performing an LU decomposition [14] with partial pivoting and triangular system solves through forward and backward substitution.

The LU decomposition can handle non-square matrices, but it is only for square matrices the triangular
solutions are performed. For improving backward stability iterative refinement subroutines are used, the library also contains routines provided to equilibrate the system, estimate the condition number, calculate the relative backward error and estimate error bounds for refined solutions.

The SuperLU package can be divided up into three parts: Driver, Computation, and Utility. In the Driver package functions for solving system of linear equation is provided. In Computation package specified computational routines is provided instead of a complete driver as in the Driver package. Using this package the user can develop new computation driver in the Modelica environments. The last package is the Utility package that supply the user with routines for creating and destroy SuperLU matrices.

5.1 Examples

Take the function dgstrf as an example in the Numeric.SuperLU.Computational sublibrary. It performs an LU factorization of a general sparse m-by-n matrix, A, using partial pivoting with row interchanges. Factorization has the form of equation (4)

\[ Pr \ast A = L \ast U \]  (4)

where Pr is a row permutation matrix, L is lower triangular with unit diagonal elements and U is upper triangular. The documentation for the function call dgstrf can be found in the SuperLU documentation [10], [11].

6 Interpolation

In many engineering and science areas data is gathered either from sampling real observations or by simulations where data is created at certain time interval. Interpolation is techniques which uses the sequence of known values to estimate the value of an unknown point [13]. Given an sequence of known sample points, \( x_k \), and the corresponding values, \( y_k \), the interpolation try to fit an function, \( f \), that when given an value in \( x_k \), returns the corresponding value in \( y_k \), shown in equation (5).

\[ f(x_k) = y_k \quad \text{where} \quad k = 1, 2, 3, \ldots, n \]  (5)

This method of trying to finding \( f \) is common known as curve fitting and the function \( f \) is then called the interpolant.

When calculating a value for an unknown data point, \( \alpha \), an control has to be made that \( \alpha \) lies inside the sequence of known values, se equation (6).

\[ \min(x_k) \leq \alpha \leq \max(x_k) \]  (6)

No interpolation can be performed if the data point is lying outside the sequence \( x_k \). To calculate the interpolated value the point is inserted in the interpolation function, \( f(\alpha) \) and the function is evaluated. In the numeric package a cubic spline interpolation scheme has been implemented both in native Modelica code and by using external library. The external library can be reached through a Modelica function that acts as a wrapper.

6.1 Cubic Spline

A cubic spline is a function that is defined as a piecewise third-order polynomial function which passes through a set of points. To create a solvable system a boundary condition is commonly place on the second derivate of each polynomial endpoint. If the boundary condition is that the second derivate is equal to zero the spline is commonly called a natural cubic spline which gives a tridiagonal system that easily can be solved. Different boundary condition can be used for creating other spline interpolation scheme [3] [6]. Suppose that the function \( f \) is to be interpolated, given by the data \((x_i, f_i), \, i = 0, \ldots, N\) where \( f_i = f(z_i) \) and \( z_i \) form an order of sequence such as \( a = x_0 < x_1 < \ldots < x_N = b \). From this the cubic interpolation function \( S \in C^2[a, b] \) can be described for each interval \([x_i, x_{i+1}]\) as equations (7) and (8) along with that the polynomials are smoothly adjusted (10) and that the interpolation condition (13) is satisfied [12].

\[ S(x) = S_i(x) \]  (7)

\[ S_i(x) = a_{i,0} + a_{i,1}(x-x_i) + a_{i,2}(x-x_i)^2 + a_{i,3}(x-x_i)^3 \]  (8)

for \( x \in [x_i, x_{i+1}] \), \( i = 0, \ldots, N-1 \)  (9)

\[ S'_{i-1}(x_i - 0) = S'_i(x_i + 0) \]  (10)

\( i = 1, \ldots, N-1 \)  (11)

\[ r = 0, 1, 2 \]  (12)

\[ S'_{i-1}(x_i - 0) = S'_i(x_i + 0) \]  (13)

\( i = 1, \ldots, N-1 \)  (14)

\[ r = 0, 1, 2 \]  (15)
7 MatrixIO

While working with numerical applications the ability to save and load matrix data in an efficient file format is often needed. Here, we decided not to create our own file format but rather to build in support for the most common formats. This gives the user the ability to work with existing data and to easier exchange data with other users. We choose to support the Matrix Market [5] [4] and Harwell-Boeing [7] formats.

7.1 Harwell-Boeing Matrix Format

The Harwell-Boeing format is today one of the most popular text-file exchange format for sparse matrixes. The file format starts with a header block where the first line contains the title and an identifier. The second line contain the number of lines for each of the data blocks and the total number of lines in the file, excluding the header. The third line contains a three character string denoting the matrix type and the number of rows and cols entries. The fourth line contains the variable Fortran format for the following data block and the fifth line is only present if there is a right hand side of the matrix. The data is stored in an 80-column, fixed length format where each matrix begins with a multiple line header block, which is followed by two, three or four data blocks.

Using this information the correct amount of space requirements can be allocated before the actual matrix data is been accessed [7].

7.2 Matrix Market Format

The Matrix Market format provides a powerful and simple file format for storing and exchanging matrix data. The format is based on an ASCII file format that is based on a collection of affiliated formats which share certain design elements. So far, we have focused on supplying routines for accessing two of these design elements, general sparse matrices and general dense matrices.

In the general sparse matrices version only the non-zero entries are stored, and for each value the corresponding matrix coordinates is stored. For general dense matrices the array format is the most efficient, and the data is provided in a column-oriented order. In both of the formats an arithmetic field is defined that specify the matrix entries, i.e. real, complex, integer, pattern, the format also specify the symmetry structure such as general, symmetric, skew-symmetric or Hermitian [5].

7.3 Examples

The easiest way to read a Matrix Market file is using the functions `getMatrixSize` and `getMatrixFile`. `getMatrixSize` takes the filename as argument and reads the size of the matrix so the a matrix with the correct size can be allocated. The function `getMatrixFile` also takes the filename as argument and reads the matrix data and store it in the corresponding data structure. An Modelica pseudo code example can be seen bellow where a matrix is loaded from a file called matrix.mtx.

```modelica
Integer n = getMatrixSize("matrix.mtx");
Real A[n,n];
A = getMatrix("matrix.mtx");
```

During the process of reading the file and storing it in the MatrixMarket format messages are provided through the `ModelicaMessage()` function.

Acknowledgements

This work was supported by Kalmar eHälsoinstitut, by Vinnova in the GRIDModelica project, and SSF in the VISIMOD project.

References


