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

Anders *Sandholm*[♮,♯,*] Peter *Bunus*[♮,†] Peter *Fritzson*[♮,‡]

[♮] PELAB - Programming Environment Lab
Dept. Computer Science, Linköping University
S-581 83 Linköping, Sweden

[♯] 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 [9], [10]. 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 introduces a new wrapper library called Numeric intended for Modelica users who want to use standard common numeric libraries as well as methods and routines for saving and loading matrixes 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 returns the size of the matrix and the other one returns the matrix data, both taking 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.

Below 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 eigenvalues Lapack [2] containsa function dgeev

[*]andsa@ida.liu.se
[†]petbu@ida.liu.se
[‡]petfr@ida.liu.se

that calculates the eigenvalues along with the left and right eigenvectors of a general matrix. The dgeev routine uses double precision but the Lapack library also contains a corresponding function for single precision calculations, named sgeev.
In the library outlined in this paper all Modelica wrapper functions for Lapack are stored in subpackages. The wrapper for the dgeev function is located in **Numeric.Lapack.SimpleDriver**, for further detail se the section dealing with the structure of the library.
Below 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.

```
Real eigenvReal[size(A, 1)];
Real eigenvImag[size(A, 1)];
Real eigenVectors[n,n];
(eigenvReal, eigenvImag,
eigenVectors) =
calcEigenValGeneralMatrix_dgeev(A1);
```

The Modelica wrapper function **calcEigenValGeneralMatrix_dgeev** allows the user to specify more input data and receive more information from Lapack than is shown here, which 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
- The package structure should 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 act as wrappers to C and FORTRAN 77 functions [9],[1]. The top level structure of the Numeric library can be seen in Figure 1 with the subpackages Lapack, SuperLU, MatrixIO, and Interpolation

2.1 The Structure of the Numeric Package

The subpackages Lapack and SuperLU contain Modelica wrapper functions that call corresponding external functions in each external library. The MatrixIO subpackage is further 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 library routines.

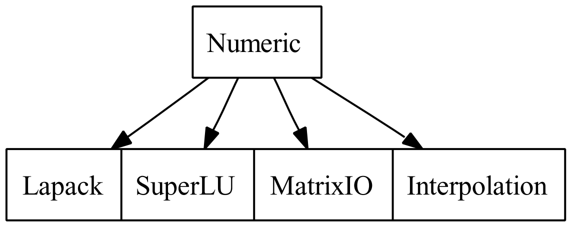

Figure 1: Structure of numeric package

2.2 Structure of the Lapack Subpackage

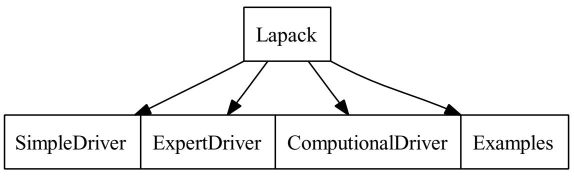

Figure 2: Structure of the Lapack subpackage and it subpackages

The Lapack subpackage can be seen in Figure 2. This package contains four subpackages, SimpleDriver, ExpertDriver, ComputionalDriver and Examples. For more information about SimpleDriver, ExpertDriver and ComputianalDriver se the Lapack section. In the Examples library different examples have been implemented which explain how the Lapack subpackage can be used in Modelica code. These examples are mostly constructed for users who know the Modelica language but are new to the Lapack library.

2.3 Structure of SuperLU package

The SuperLU subpackage has been divided into library subpackages, Driver, Computation, Utility as well as a section called Examples that has been added. The packaged structure can be view in Figure 3. For detailed information about the Driver, Computation and Utility subpackages se the SuperLU subpackage. In the Example subpackage to SuperLU different Modelica examples have been implemented that show how the SuperLU library can be used

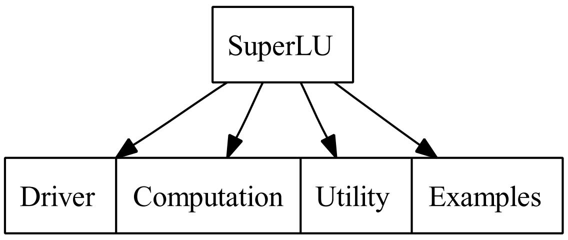

Figure 3: Structure of Lapack package with its subpackages

2.4 Structure of MatrixIO package

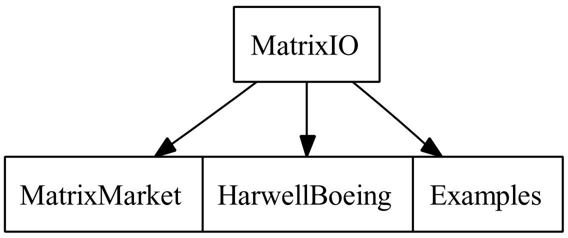

Figure 4: Structure of Lapack package

The MatrixIO packages implement support for different matrix file formats. Currently the Matrix-Market and the Harwelll-Boeing subpackages 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 detailed information about the Matrix Market and the Harwell-Boeing se corresponding sections. Examples that show how matrix data can be loaded and saved are implemented in the Examples subpackage.

2.5 Structure of the Interpolation subpackage

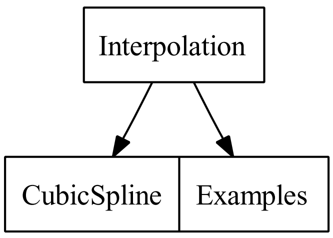

Figure 5: Structure of the Interpolation subpackage

The Interpolation subpackage is designed with the same idea as the other packages. Currently the subpackage is divided into two subpackages, CubicSpline and Examples, se Figure 5. The CubicSpline subpackage contains both native Modelica function implementations and Modelica wrapper functions for use of external cubic spline function implemented in C code. The Examples subpackage contains easily understandable examples that show both how the Modelica implemented versions and the external version can be called from Modelica code.
For further details about cubic spline se the Interpolation subpackage section.

3 Library Design Issues

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

3.1 Naming Conventions

The Modelica Numeric library uses function and variable names from the original package as a postfix part of the name along with a more explanatory Java-style name comprising the beginning of the name. This will give new users more understanding of functions and variables, without reading the detailed documentation for each variable. Users who are familiar with the corresponding non-Modelica libraries will recognize functions and variables due to the postfix part of the name.
An example is the Modelica wrapper function **calcEigenValGeneralMatrix_dgeev** which is introduced in the Introduction part of this paper. The first part of the function name tells the user that it calculates the eigenvalues for a general matrix and the postfix part specifies that the dgeev function is used. The same naming convention is used for variables. The dgeev function has a variable named JOBVL that specifies it the left eigenvalues should be calculated or not. In the Modelica wrapper function this variable is named `calcLeftEigenV_JOBVL` which are a more self explanatory Java-style name along with the Lapack

variable name as a postfix part of the name.

3.2 Documentation

The issue about documentation has been addressed by including the external function documentation into the Modelica wrapper function documentation node. Below the first part of the documentation for the Modelica wrapper function **calcEigenValGeneralMatrix_dgeev** is shown.
First in the documentation comes a specification of the difference 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 have 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. In this example 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 numerical problems in linear algebra. The library includes routines for solving systems 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. Examples 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 normalizing matrices to improve accuracy.
- Computational Routines shall more be seen as routines designed to perform a specific task, such as a LU factorization or reduction of a real system matrix to tridiagonal form. Usually these functions are used to construct more advanced functions in the Basic and Advanced routines libraries. The routines are categorized in systems of linear equations, eigenvalue problems, orthogonal factorization, and singular value decomposition.

4.1 Example

An example of the simple driver routines is the dgeev function that calculates right and left eigenvalues and eigenvectors for an N-by-N real nonsymmetric matrix. This calculation can be described as finding the eigenvalues λ and corresponding eigenvectors $z \neq 0$ as equattion (1) and (2) describe.

$$Az = \lambda z \tag{1}$$

$$A = A^T \text{ where A is real} \tag{2}$$

When all eigenvalues and eigenvectors have been calculated equation (3) is solved.

$$A = Z \Lambda Z^T \tag{3}$$

Where Λ is a diagonal matrix whose diagonal elements are the eigenvalues, Z is an orthogonal matrix whose columns are the eigenvectors [3].
As described previously the Modelica wrapper function for dgeev is called **calcEigenValGeneralMatrix_dgeev** and is shown below, where the documentation part has been removed in this example.

```
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";
output Integer INFO
 "=0 successful computation";

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 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 the 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 are needed for the Lapack implementation have been placed in the protected section. For the function outlined above the working variables LWORK and WORK have been placed here, along with the variable N that specifies the order of the matrix.

5 SuperLU

For solving large, sparse, nonsymmetric systems of linear equations the SuperLU library is commonly used [11]. The SuperLU library is available either in C or in Fortran code. Here our Modelica implementation uses the Fortran interface for maximum performance. The SuperLU library starts by performing an LU decomposition [15] with partial pivoting and triangular systems solved through forward and backward substitution.
The LU decomposition can handle non-square matrices, but it is only for square matrices the triangular solver is used. For improving backward stability interactive 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 re-fined solutions.
The SuperLU subpackage is divided into three parts: Driver, Computation, and Utility. In the Driver subpackage functions for solving systems of linear equation are provided. In the Computation subpackage specified computational routines are provided instead of a complete driver as in the Driver package. Using this pack-age the user can develop a new computation driver in the Modelica environments. The last package is the Utility subpackage that supplies 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 a LU factorization of a general sparse m-by-n matrix, A, using partial pivoting with row interchanges. Factorization has the form of equation (4)

$$Pr*A = L*U \tag{4}$$

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 [11], [12].

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 intervals. Interpolation is a technique which uses the sequence of known values to estimate the value of an unknown point [14]. Given a sequence of known sample points, x_k, and the corresponding values, y_k, the interpolation tries to fit a function, f, that which when given an value in x_k, returns the corresponding value in y_k, shown in equation (5).

$$f(x_k) = y_k \quad where \;\; k = 1,2,3,.....n \tag{5}$$

This method of trying to find f is commonly known as curve fitting and the function f is then called the interpolant.
When calculating a value for an unknown data point, α, a control has to be made that ,α, lies inside the sequence of known values, se equation (6).

$$min(x_k) \leq \alpha \leq max(x_k) \tag{6}$$

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 placed on the second derivate of each polynomial end point. If the boundary condition is that the second derivative 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 conditions can be used for creating other spline interpolation scheme [4] [7]. Suppose that the function f is to be interpolated, given by the data (x_i, f_i), $i = 0,....,N$ where $f_i = f(z_i)$ and z_i form an order of sequence such as $a = x_0 < x_1 < ... < x_N = b$. 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 the fact that the polynomials are smoothly adjusted (10) and that the interpolation condition (13) is satisfied [13].

$$S(x) \equiv S_i(x) \tag{7}$$

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

$$\text{for } x \in [x_i, xi + 1] \text{ , } i = 0, \ldots\ldots, N - 1 \tag{9}$$

$$S^r_{i-1}(x_i - 0) = S^r_i(x_i + 0) \tag{10}$$

$$i = 1, \ldots, N - 1 \tag{11}$$

$$r = 0, 1, 2 \tag{12}$$

$$S^r_{i-1}(x_i - 0) = S^r_i(x_i + 0) \tag{13}$$

$$i = 1, \ldots, N - 1 \tag{14}$$

$$r = 0, 1, 2 \tag{15}$$

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 have chosen to support the Matrix Market [6] [5] and Harwell-Boeing [8] formats.

7.1 Harwell-Boeing Matrix Format

The Harwell-Boeing format is today one of the most popular text-file exchange formats 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 column 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 storage can be allocated before the actual matrix data is accessed [8].

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 specifies the matrix entries, i.e, real, complex, integer, pattern. The format also specifies the symmetry structure such as general, symmetric, skew-symmetric or Hermitian [6].

7.3 Examples

The easiest way to read a Matrix Market file is using the functions **getMatrixSize** and **getMatrixFile**. **getMatrixSize** takes the file name as argument and reads the size of the matrix so that 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. A Modelica pseudo code example can be seen below where a matrix is loaded from a file called matrix.mtx.

```
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

[1] The Modelica Language specification version 2.2. The Modelica Association, March 2005. http://www.modelica.org.

[2] E Anderson, Z Bai, C Bischof, J Demmel, J Dongarra, J Du Croz, A Greenbaum, S Hammarling, A McKenney, S Ostrouchov, and D Sorensen. *LAPACK Users' Guide*. 1995.

[3] George B. Arfken, Hans J. Weber, and Hans-Jurgen Weber. *Mathematical Methods for Physicists*. Academic Press, 1985.

[4] Richard H. Bartels, John C. Beatty, and Brian A. Barsky. *An Introduction to Splines for Use in Computer Graphics and Geometric Modeling*. Morgan Kaufmann, 1995.

[5] Ronald F. Boisvert, editor. *The Matrix Market: A Web Resource for Test Matrix Collections*, London, 1997. Chapman & Hall.

[6] Ronald F. Boisvert, Roldan Pozo, and Karin Remington. The matrix market exchange formats: Initial design. Technical Report NISTIR 5935, National Institute of Standards and Technology, Gaithersburg, MD, USA, December 1996.

[7] Carl De Boor. *A Practical Guide to Splines*. Springer, 2002.

[8] Iain Duff, Roger G. Grimes, and John G. Lewis. Users' guide for the harwell-boeing sparse matrix collection (Release I). Technical Report TR/PA/92/86, CERFACS, October 1992.

[9] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley-IEEE Press, 2004. ISBN 0-471-471631.

[10] Peter Fritzson, Peter Aronsson, Håkan Lundvall, Kaj Nyström, Adrian Pop, Levon Saldami, and David Broman. The Openmodelica modeling, simulation, and software development environment. *Simulation News Europe*, 44/45, 2005.

[11] John Gilbert James W. Demmel and Xiaoye S. Li. Superlu users' guide. Technical Report UCB/CSD-97-944, EECS Department, University of California, Berkeley, 1997.

[12] John R. Gilbert Xiaoye S. Li James W. Demmel, Stanley C. Eisenstat and Joseph W.H. Liu. A supernodal approach to sparse partial pivoting. Technical Report UCB/CSD-95-883, EECS Department, University of California, Berkeley, 1995.

[13] Boris I Kvasov. *Methods of Shape-Preserving Spline Approximation*. World Scientific, 2000.

[14] Erik Meijering. Chronology of interpolation: From ancient astronomy to modern signal and image processing. volume 90, pages 319–342. IEEE, March 2002.

[15] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical recipes in FORTRAN (2nd ed.): the art of scientific computing*. Cambridge University Press, New York, NY, USA, 1992.