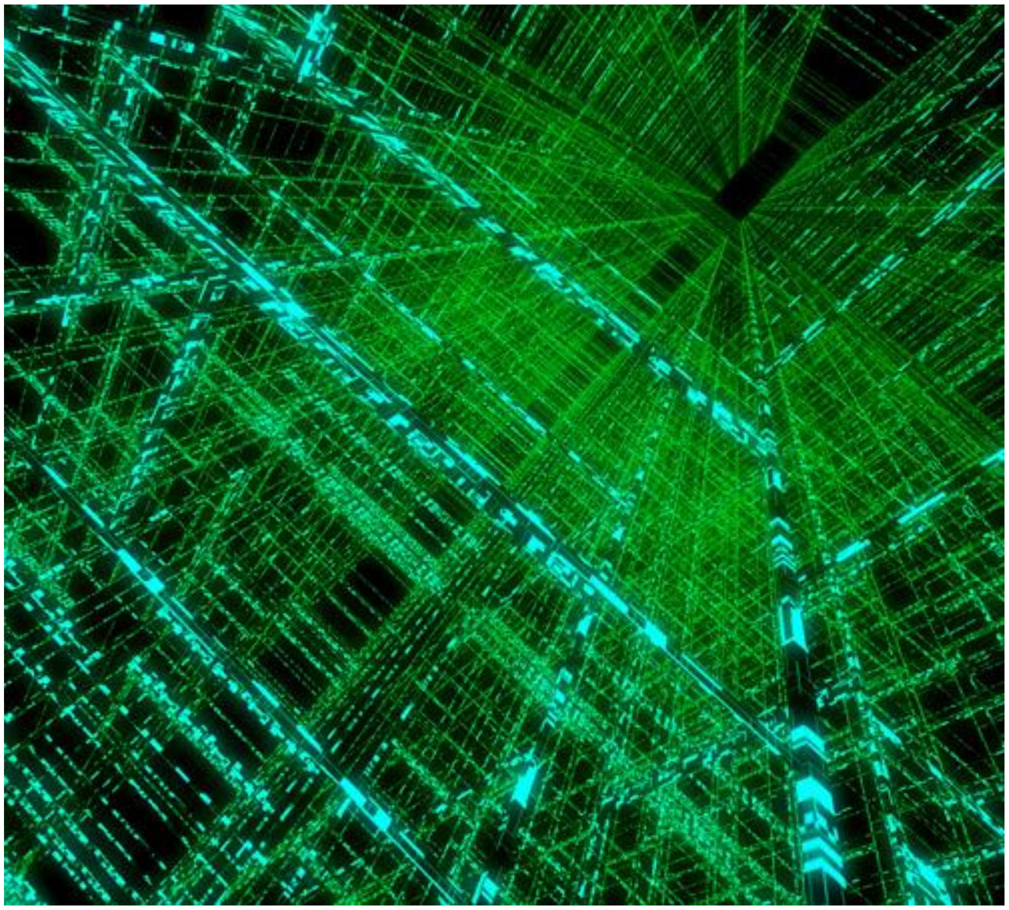
Ponderosa Computing

Linear Algebra .NET Class Library



Paul J. McClellan, Ph.D.

7 August 2019

Table of Contents

1 Ponderosa Computing Linear Algebra .NET Class Library

The C# programming language and .NET Framework provide one- and two-dimensional array data types that are well suited to representing linear algebra vectors and matrices. The Ponderosa Computing Linear Algebra .NET class library, **PonderosaComputing.LinearAlgebra.dll**, provides linear algebra vector/matrix metrics and operations using one-dimensional and two-dimensional, double-precision .NET array objects to represent column vectors and matrices, respectively. These metrics and operations are implemented as **LinearAlgebra** class methods using **LAPACK** algorithms.

LAPACK [1,2] is a freely-available, peer-reviewed computational linear algebra software library that provides routines for solving systems of simultaneous linear equations, computing least-squares solutions of linear systems of equations, and computing eigenvalue and singular value decompositions. The associated matrix factorizations (LU, Cholesky, LQ/QR, SVD, Schur, generalized Schur) are also provided, as are related computations such as estimating condition numbers. LAPACK has been used in or as a starting point for implementation of linear algebra computing environments and is a standard by which other libraries and computing environments are often compared.

The Ponderosa Computing Linear Algebra .NET class library provides selected LAPACK linear algebra computations using the public-domain CLAPACK library from the Netlib Repository [3,4]. The CLAPACK library is a machine-translation of the LAPACK Fortran library to C code. The **LinearAlgebra** class methods are implemented using C++ Interop in C++/CLI to wrap the CLAPACK functions so they can be accessed by code that is authored in C# or another .NET Framework language [5].

This document describes version 1.2 of the Ponderosa Computing Linear Algebra .NET class library. It was built using the Microsoft .NET Framework 4.5.2.

2 How to Use this Class Library

We illustrate here how to use the Ponderosa Computing Linear Algebra .NET class library in a C# project using Microsoft Visual Studio 2012.

The Ponderosa Computing Linear Algebra .NET class library installer installs by default the class library DLL, **PonderosaComputing.LinearAlgebra.dll**, in the folder C:\Program Files (x86)\Ponderosa Computing\LinearAlgebra.NET. If needed, the installer will also attempt to install the .NET Framework 4.5.2 from the web.

To use the class methods of this class library first add a reference to the library in your Visual Studio C# project:

1. In Visual Studio click the menu item “Project” and “Add Reference…”.
2. Click the “Browse…” button on the lower right and locate the file **PonderosaComputing.LinearAlgebra.dll.** By default the installer installs this file in C:\Program Files (x86)\Ponderosa Computing\LinearAlgebra.NET.
3. Click on the file name and click the “Add” button on the lower right of the file dialog.
4. Click the “OK” button.

Then create a **PonderosaComputing.LinearAlgebra** class instance and call the desired class method through this instance. **LinearAlgebraExceptions** can also be caught.

Here is example C# code using this library:

```
using System;
using PonderosaComputing;

namespace PcLinAlgCLTest
{
    class Program
    {
        static void WriteVector(double[] Vec)
        {
            for (int i = 0; i < Vec.GetLength(0); ++i)
            {
                Console.Write("{0}  ", Vec[i]);
            }
            Console.WriteLine();
            Console.WriteLine();
        }

        static void WriteMatrix(double[,] Mat)
        {
            for (int i = 0; i < Mat.GetLength(0); ++i)
            {
                for (int j = 0; j < Mat.GetLength(1); ++j)
                {
                    Console.Write("{0}  ", Mat[i, j]);
                }
                Console.WriteLine();
            }
            Console.WriteLine();
        }

        static void Main(string[] args)
        {
            LinearAlgebra la = new LinearAlgebra();

            // Exception thrown
            double[,] MS = {{1,2,4},{1,2,4},{1,2,4}};
            double[] v = {1,2,3};
            try
            {
                Console.WriteLine("*** Test error handling ***");
                Console.WriteLine("");

                Console.WriteLine("Singular matrix M");
                WriteMatrix(MS);
                Console.WriteLine("rank = {0}", la.Rank(MS));
                Console.WriteLine("determinant = {0}", la.Determinant(MS));
                Console.WriteLine();

                Console.WriteLine("vector v");
                WriteVector(v);

                Console.WriteLine("Matrix-vector product u = M * v");
                double[] u = la.Multiply(MS, v);
                WriteVector(u);

                Console.WriteLine("Try to solve M * x = u for x");
                Console.WriteLine("(Expect exception)");
                Console.WriteLine();
                double[] x = la.SolveFullRankLinearSystem(MS, u);
                Console.WriteLine("Expected not to reach here!");
                WriteVector(x);
            }
            catch (LinearAlgebraException e)
            {
                Console.WriteLine("Exception caught: {0}", e.Message);
            }
            finally
            {
                Console.WriteLine();
            }

            // metrics
            double[,] M33 = {{1,2,3},{1,4,9},{1,8,27}};
            double[,] M43 = {{1,2,3},{1,4,9},{1,8,27},{1,16,81}};
            try
            {
                Console.WriteLine("*** Solve full rank square system  ***");
                Console.WriteLine("");

                Console.WriteLine("Full rank matrix M:");
                WriteMatrix(M33);
                Console.WriteLine("rank = {0}", la.Rank(M33));
                Console.WriteLine("determinant = {0}", la.Determinant(M33));
                Console.WriteLine();

                Console.WriteLine("vector v");
                WriteVector(v);

                Console.WriteLine("Matrix-vector product u = M * v");
                double[] u = la.Multiply(M33, v);
                WriteVector(u);

                Console.WriteLine("Solve M * x = u for x");
                double[] x = la.SolveFullRankLinearSystem(M33, u);
                WriteVector(x);

                Console.WriteLine("*** Solve overdetermined system  ***");
                Console.WriteLine("");

                Console.WriteLine("matrix H:");
                WriteMatrix(M43);
                Console.WriteLine("rank = {0}", la.Rank(M43));
                Console.WriteLine("");

                Console.WriteLine("Matrix-vector product w = H * v");
                double[] w = la.Multiply(M43, v);
                WriteVector(w);

                Console.WriteLine("Solve H * y = w for y with error bound:");
                double[] ye = la.SolveLsqLinearSystemEB(M43, w);
                WriteVector(ye);
            }
            catch (LinearAlgebraException e)
            {
                Console.WriteLine("Exception caught: {0}", e.Message);
            }
            finally
            {
                Console.WriteLine();
            }

            Console.ReadLine();
        }
    }
}
```

3 Notation, Implementation, and Numerical Precision

3.1 Linear Algebra Notation

A linear algebra **column vector** is a one-dimensional array of elements consisting of numbers, symbols, or expressions arranged in a column. A column vector of m elements consists of **elements** x_i, where *i* is the row location of the element:

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} .$$

The Ponderosa Computing Linear Algebra .NET class library uses double-precision one-dimensional .NET arrays to represent column vectors. A column vector can also be represented by a double-precision two-dimensional .NET array object having one column.

A linear algebra **row vector** is a one-dimensional array of elements consisting of numbers, symbols, or expressions arranged in a row. A row vector of n elements consists of **elements** y_i, where *i* is the column location of the element:

$$\boldsymbol{y} = \begin{bmatrix} y_1 & y_2 & \cdots & y_n \end{bmatrix} .$$

This class library uses double-precision two-dimensional .NET arrays with one row to represent row vectors.

A linear algebra **matrix** is a two-dimensional rectangular array of elements consisting of numbers, symbols, or expressions arranged in rows and columns. A matrix of m rows and n columns consists of **elements** x_{ij}, where *i* is the row location and *j* is the column location of the element:

$$\boldsymbol{X} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & \cdots & x_{mn} \end{bmatrix} .$$

This class library uses double-precision two-dimensional .NET arrays to represent matrices.

The **transpose** operation $\boldsymbol{Y} = \boldsymbol{X^T}$ on an m x n matrix ***X*** creates an n x m matrix ***Y*** with rows and columns interchanged:

$$\boldsymbol{Y} = \boldsymbol{X^T} = \begin{bmatrix} y_{11} & y_{12} & \cdots & y_{1m} \\ y_{21} & y_{22} & \cdots & y_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ y_{n1} & y_{n2} & \cdots & y_{nm} \end{bmatrix}, \ where\ y_{ij} = x_{ji} .$$

The transpose operation $\boldsymbol{y} = \boldsymbol{x}^{\boldsymbol{T}}$ on an m-element column vector $\boldsymbol{x}$ creates an m-element row vector:

$$\boldsymbol{x}^T = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}^T = [x_1 \quad x_2 \quad \cdots \quad x_m] \, .$$

This class library will return the transpose of a column vector as a row vector represented by a double-precision two-dimensional .NET array with one row.

Numerical linear algebra operations are defined for matrices and vectors with all elements defined on the domain of complex numbers ℂ. The Ponderosa Computing Linear Algebra .NET class library provides linear algebra vector/matrix metrics and operations using double-precision one-dimensional and two-dimensional .NET array objects to represent vector and matrix arguments and return values. Since these array objects are declared as double-precision the elements of these vectors and matrices are defined on the domain of real numbers ℝ = (-∞, +∞).

3.2 Library Uses the CLAPACK Implementation of LAPACK

LAPACK [1] is a freely-available, peer-reviewed numerical linear algebra software package that provides routines for solving systems of simultaneous linear equations, computing least-squares solutions of linear systems of equations, and computing eigenvalue and singular value decompositions. The associated matrix factorizations (LU, Cholesky, LQ/QR, SVD, Schur, generalized Schur) are also provided, as are related computations such as estimating condition numbers.

The Netlib Repository provides a cross-platform Fortran source distribution of LAPACK [2] and a C source distributions of CLAPACK [3]. Instructions and tools for building CLAPACK on the Windows platform are available from the University of Tennessee Innovative Computing Laboratory [4]. The Ponderosa Computing Linear Algebra .NET class library uses the CLAPACK implementation of the LAPACK software package.

3.3 Library Computations use IEEE 754 Double Precision Format

The numerical linear algebra computations are all defined for matrix and vector elements defined on the domain of real numbers ℝ = (-∞, +∞). To describe limiting behavior of linear algebra computations it can be useful to define them for matrix and vector elements defined on the domain of the **affinely extended real numbers** [-∞, +∞], which adds the elements +∞ (positive infinity) and -∞ (negative infinity) to the real numbers.

The Ponderosa Computing Linear Algebra .NET class library implements its functions using the double precision format defined by the IEEE Standard 754 for Binary Floating-Point Arithmetic [6]. This format includes representations of signed zeros and normalized, denormalized, and nonfinite (infinite and indeterminate) floating point numbers.

3.3.1 Machine Constants

We define here some double precision constants that appear later in this document.

eps : The relative machine precision, the distance from 1.0 to the next largest double-precision number. This number is $eps = \text{dlamch\_(P)} = 2^{-52} = 2.2204460492503131e-016$.

$safemin$: The minimum positive floating point value such that $1/safemin$ does not overflow. $safemin = \text{dlamch\_(S)} = 2.2250738585072014e-308$.

3.3.2 Normalized and Denormalized Numbers

The Ponderosa Computing Linear Algebra .NET class library supports normalized IEEE 754 double precision numbers with 53 binary digits of precision (equivalent to nearly 16 decimal digits of precision).

The library also supports IEEE 754 denormalized numbers. As an example, for the matrix

$$\boldsymbol{A} = \begin{bmatrix} 1E-160 & 0 \\ 0 & 1E-160 \end{bmatrix},$$

the class method **`LinearAlgebra.Determinant()`** returns the denormalized value 9.99988867182683E-321.

3.3.3 Nonfinite Numbers

The CLAPACK implementation does not reliably handle nonfinite (infinite and indeterminate) floating point matrix or vector elements. For example, some computational loops are skipped when a factor is zero under the assumption that the skipped loop would have no impact on the computed results [7]. But this may fail to propagate nonfinite values or fail to detect invalid operations.

Version 1.2 of the Ponderosa Computing Linear Algebra .NET class library does not provide special handling of nonfinite floating point matrix or vector elements, and its class methods might not provide reliable results for method arguments containing such elements.

4 LinearAlgebra Class Methods

The Ponderosa Computing Linear Algebra .NET class library implements its functions using version 3.2.1 of CLAPACK [3]. This implementation of CLAPACK was machine translated by Netlib from Fortran 77 to ANSI C using the f2c tool and version 3.2.1 of LAPACK [2]. The Ponderosa Computing Linear Algebra .NET class library uses the reference BLAS library included with this CLAPACK distribution.

4.1 Matrix Creation and Extraction

The Ponderosa Computing Linear Algebra .NET class library provides the following class methods for creating matrices and extracting the diagonals of a matrix:

Create a constant matrix	`GenerateConstant()`
Create a random matrix	`GenerateRandom()`
Create an identity matrix	`GenerateIdentity()`
Create a diagonal matrix	`GenerateDiagonal()`
Extract the diagonal elements of a matrix	`ExtractDiagonals()`
Extract a row vector from a matrix	`ExtractRow()`
Extract a column vector from a matrix	`ExtractColumn()`
Extract rows from a matrix	`ExtractRows()`
Extract columns from a matrix	`ExtractColumns()`

4.1.1 Create a Constant Matrix

The class method

```
LinearAlgebra.GenerateConstant(m,n,value)
```

returns an m x n matrix with entries all equal to value. If m < 1 or n < 1 then this method throws an **eBadParamError** LinearAlgebraException.

4.1.2 Create a Random Matrix

The class method

```
LinearAlgebra.GenerateRandom(m,n,minimum,maximum)
```

returns an m x n matrix with random entries in [minimum, maximum). If m < 1 or n < 1 or maximum < minimum then this method throws an **eBadParamError** LinearAlgebraException.

This class method generates uniformly distributed random elements in the interval [minimum, maximum] using a Mersenne Twister pseudo-random generator of 32-bit numbers with a state size of 19937 bits provided by the C++11 standard library. This generator is initialized when the **PonderosaComputing.LinearAlgebra** class instance is constructed and a class method is first invoked through this instance.

4.1.3 Create an Identity Matrix

The class method

```
LinearAlgebra.GenerateIdentity(n)
```

returns an n x n identity matrix, $\boldsymbol{I_n}$, of order n. If n < 1 then this method throws an **eBadParamError** LinearAlgebraException.

4.1.4 Create a Diagonal Matrix

The class method

```
LinearAlgebra.GenerateDiagonal(d)
```

returns an n x n diagonal matrix $\boldsymbol{D}$ where n is the size of the argument vector $\boldsymbol{d}$ containing the diagonal elements of the returned diagonal matrix.

4.1.5 Extract Diagonal Elements

The class method

```
LinearAlgebra.ExtractDiagonals(A)
```

returns the min(m, n) main diagonal elements of an m x n matrix $\boldsymbol{A}$ as a min(m, n)-element vector.

4.1.6 Extract Row Elements

The class method

```
LinearAlgebra.ExtractRow(A,r)
```

returns the n elements of the r-th row of an m x n matrix $\boldsymbol{A}$ as a n-element column vector $\boldsymbol{v}$. The row number r is base 0. If r < 0 or r ≥ m then this method throws an **eBadParamError** LinearAlgebraException.

4.1.7 Extract Column Elements

The class method

`LinearAlgebra.ExtractColumn(A,c)`

returns the m elements of the c-th column of an m x n matrix $\boldsymbol{A}$ as an m-element vector $\boldsymbol{v}$. The column number c is base 0. If $c < 0$ or $c \geq n$ then this method throws an **eBadParamError** LinearAlgebraException.

4.1.8 Extract Rows

The class method

`LinearAlgebra.ExtractRows(A,r₁,r₂)`

returns the rows r_1 through r_2 of an m x n matrix $\boldsymbol{A}$ as a (r_2-r_1+1) x n matrix. The row numbers r_1 and r_2 are base 0. If $r_1 < 0$ or $r_1 > r_2$ or $r_2 \geq m$ then this method throws an **eBadParamError** LinearAlgebraException.

4.1.9 Extract Columns

The class method

`LinearAlgebra.ExtractColumns(A,c₁,c₂)`

returns the columns c_1 through c_2 of an m x n matrix $\boldsymbol{A}$ as a m x (c_2-c_1+1) matrix. The column numbers c_1 and c_2 are base 0. If $c_1 < 0$ or $c_1 > c_2$ or $c_2 \geq m$ then this method throws an **eBadParamError** LinearAlgebraException.

4.2 Matrix Addition, Subtraction, Multiplication, and Transpose Operations

The Ponderosa Computing Linear Algebra .NET class library provides the following class methods for vector and matrix addition and subtraction and for matrix multiplication and transpose operations:

Vector or Matrix addition	`Add()`
Vector or Matrix subtraction	`Subtract()`
Matrix multiplication	`Multiply()`
Matrix transpose multiplication	`TransposeMultiply()`
Matrix multiply transpose	`MultiplyTranspose()`
Transpose	`Transpose()`

4.2.1 Vector or Matrix Addition

The class method

```
LinearAlgebra.Add(a,b)
```

returns the element-wise sum of n-element column vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ as an n-element column vector $\boldsymbol{c} = \boldsymbol{a} + \boldsymbol{b}$. If the number of elements of $\boldsymbol{a}$ does not match the number of elements of $\boldsymbol{b}$ then this method throws an **eBadParamError** LinearAlgebraException.

The class method

```
LinearAlgebra.Add(A,B)
```

returns the element-wise sum of m x n matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ as an m x n matrix $\boldsymbol{C} = \boldsymbol{A} + \boldsymbol{B}$. If the number of rows and columns of $\boldsymbol{A}$ do not match the number of rows and columns of $\boldsymbol{B}$ then this method throws an **eBadParamError** LinearAlgebraException.

4.2.2 Vector or Matrix Subtraction

The class method

```
LinearAlgebra.Subtract(a,b)
```

returns the element-wise difference of n-element column vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ as an n-element column vector $\boldsymbol{c} = \boldsymbol{a} - \boldsymbol{b}$. If the number of elements of $\boldsymbol{a}$ does not match the number of elements of $\boldsymbol{b}$ then this method throws an **eBadParamError** LinearAlgebraException.

The class method

```
LinearAlgebra.Subtract(A,B)
```

returns the element-wise difference of m x n matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ as an m x n matrix $\boldsymbol{C} = \boldsymbol{A} - \boldsymbol{B}$. If the number of rows and columns of $\boldsymbol{A}$ do not match the number of rows and columns of $\boldsymbol{B}$ then this method throws an **eBadParamError** LinearAlgebraException.

4.2.3 Matrix Multiplication

The class method

```
LinearAlgebra.Multiply(A,b)
```

returns the product of an m x n matrix $\boldsymbol{A}$ and a n-element column vector $\boldsymbol{b}$ as an m-element column vector $\boldsymbol{v} = \boldsymbol{A}\,\boldsymbol{b}$. If the number of columns of $\boldsymbol{A}$ does not match the number of elements of $\boldsymbol{b}$ then this method throws an **eBadParamError** LinearAlgebraException.

The class method

```
LinearAlgebra.Multiply(A,B)
```

returns the product of an m x n matrix $\boldsymbol{A}$ and a n x p matrix $\boldsymbol{B}$ as an m x p matrix $\boldsymbol{C} = \boldsymbol{A}\,\boldsymbol{B}$. If the number of columns of $\boldsymbol{A}$ does not match the number of rows of $\boldsymbol{B}$ then this method throws an **eBadParamError** LinearAlgebraException.

These methods compute the matrix product using routine `dgemm_()` from the CLAPACK v 3.2.1 BLAS package.

4.2.4 Matrix Transpose Multiplication

The class method

```
LinearAlgebra.TransposeMultiply(A,b)
```

returns the product of the transpose of an n x m matrix $\boldsymbol{A}$ and a n-element column vector $\boldsymbol{b}$ as an m-element column vector $\boldsymbol{v} = \boldsymbol{A}^T\boldsymbol{b}$. If the number of rows of $\boldsymbol{A}$ does not match the number of elements of $\boldsymbol{b}$ then this method throws an **eBadParamError** LinearAlgebraException.

The class method

```
LinearAlgebra.TransposeMultiply(A,B)
```

returns the product of an m x n matrix $\boldsymbol{A}$ and a n x p matrix $\boldsymbol{B}$ as an m x p matrix $\boldsymbol{C} = \boldsymbol{A}^T\boldsymbol{B}$. If the number of columns of $\boldsymbol{A}$ does not match the number of rows of $\boldsymbol{B}$ then this method throws an **eBadParamError** LinearAlgebraException.

These methods use routine `dgemm_()` from the CLAPACK v 3.2.1 BLAS package.

4.2.5 Matrix Multiply Transpose

The class method

```
LinearAlgebra.MultiplyTranspose(A,B)
```

returns the product of an m x n matrix $\boldsymbol{A}$ and the transpose of a p x n matrix $\boldsymbol{B}$ as an m x p matrix $\boldsymbol{C} = \boldsymbol{A}\,\boldsymbol{B}^T$. If the number of columns of $\boldsymbol{A}$ does not match the number of columns of $\boldsymbol{B}$ then this method throws an **eBadParamError** LinearAlgebraException.

This method uses routine `dgemm_()` from the CLAPACK v 3.2.1 BLAS package.

4.2.6 Transpose

The class method

```
LinearAlgebra.Transpose(v)
```

returns the transpose of an m-element column vector $\boldsymbol{v}$ as the 1 x m matrix representing the row vector $\boldsymbol{u} = \boldsymbol{v}^T$.

The class method

```
LinearAlgebra.Transpose(A)
```

returns the transpose of an m x n matrix $\boldsymbol{A}$ as the n x m matrix $\boldsymbol{A}^T$.

4.3 Scalar-Valued Functions

The Ponderosa Computing Linear Algebra .NET class library provides the following scalar-valued vector and matrix methods:

Vector dot product	`Dot()`
1-norm (column norm) of a vector or matrix	`OneNorm()`
Infinity-norm (row norm) of a vector or matrix	`InfinityNorm()`
Frobenius norm of a vector or matrix	`FrobeniusNorm()`
2-norm (spectral norm) of a vector or matrix	`TwoNorm()`
Rank of a matrix	`Rank()`
Spectral radius of a symmetric matrix	`SpectralRadius()`
Trace of a square matrix	`Trace()`
1-norm inverse condition number estimate of a square matrix	`InverseOneNormConditionNumberEstimate()`
Infinity-norm inverse condition number estimate of a square matrix	`InverseInfinityNormConditionNumberEstimate()`
Determinant of a square matrix	`Determinant()`

4.3.1 Vector Dot Product

The class method

`LinearAlgebra.Dot(u,v)`

returns the dot product of two n-element column vectors $\boldsymbol{u}$ and $\boldsymbol{v}$. This is the sum of the products of the corresponding elements of $\boldsymbol{u}$ and $\boldsymbol{v}$:

$$u \cdot v = \sum_{i=1}^{n} u_i * v_i$$

These methods compute the dot product using routine `ddot_()` from the CLAPACK v 3.2.1 BLAS package.

4.3.2 1-Norm (Column Norm)

The class methods

`LinearAlgebra.OneNorm(A)`

`LinearAlgebra.OneNorm(v)`

return the 1-norm (column norm, $\|\boldsymbol{A}\|_1$, $\|\boldsymbol{v}\|_1$) of a matrix $\boldsymbol{A}$ or column vector $\boldsymbol{v}$, respectively.

This is the maximum absolute column sum of the elements of an m x n matrix $\boldsymbol{A}$:

$$\|\boldsymbol{A}\|_1 = \max_{1\le j\le n} \sum_{i=1}^{m} |a_{ij}|$$

It is the absolute column sum of the elements of an n element column vector $\boldsymbol{v}$:

$$\|\boldsymbol{v}\|_1 = \sum_{i=1}^{n} |v_i|$$

This method computes the 1-norm using the routine `dlange_()` from CLAPACK v 3.2.1.

4.3.3 Infinity-Norm (Row Norm)

The class methods

`LinearAlgebra.InfinityNorm(A)`

`LinearAlgebra.InfinityNorm(v)`

return the infinity-norm (row norm, $\|\boldsymbol{A}\|_\infty$, $\|\boldsymbol{v}\|_\infty$) of a matrix $\boldsymbol{A}$ or column vector $\boldsymbol{v}$, respectively.

This is the maximum absolute row sum of the elements of an m x n matrix $\boldsymbol{A}$:

$$\|\boldsymbol{A}\|_\infty = \max_{1\le i\le m} \sum_{j=1}^{n} |a_{ij}|$$

It is the maximum absolute value of the elements of an n element column vector $\boldsymbol{v}$:

$$\|\boldsymbol{v}\|_\infty = \max_{1\le i\le n} |v_i|$$

This method computes the infinity-norm using routine `dlange_()` from CLAPACK v 3.2.1.

4.3.4 Frobenius-Norm

The class methods

```
LinearAlgebra.ForbeniusNorm(A)

LinearAlgebra.ForbeniusNorm(v)
```

return the Frobenius-norm ($\|\boldsymbol{A}\|_F$, $\|\boldsymbol{v}\|_F$) of a matrix $\boldsymbol{A}$ or vector $\boldsymbol{v}$, respectively.

This is the square root of the sum of absolute squares (root mean square) of the elements of an m x n matrix $\boldsymbol{A}$:

$$\|\boldsymbol{A}\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} |a_{ij}|^2}$$

It is the square root of the sum of absolute squares (root mean square) of the elements of an n vector $\boldsymbol{v}$:

$$\|\boldsymbol{v}\|_F = \sqrt{\sum_{i=1}^{n} |v_i|^2}$$

This method computes the Frobenius norm using routine `dlange_()` from CLAPACK v 3.2.1.

4.3.5 2-Norm (Spectral Norm)

The class methods

```
LinearAlgebra.TwoNorm(A)
LinearAlgebra.TwoNorm(v)
```

return the 2-norm (spectral norm, $\|\boldsymbol{A}\|_2$, $\|\boldsymbol{v}\|_2$) of a matrix $\boldsymbol{A}$ or column vector $\boldsymbol{v}$, respectively.

This is the largest singular value of an m x n matrix $\boldsymbol{A}$:

$$\|\boldsymbol{A}\|_2 = \sigma_{\max}(\boldsymbol{A}) \ .$$

It is the square root of the sum of absolute squares (root mean square) of the elements of an n vector $\boldsymbol{v}$:

$$\|\boldsymbol{v}\|_F = \sqrt{\sum_{i=1}^{n} |v_i|^2}$$

This method computes the min(m, n) singular values of a m x n matrix $\boldsymbol{A}$ using routine `dgesvd_()` from CLAPACK v 3.2.1.

Routine `dgesvd_()` uses routine `dbdsqr_()` to compute the singular values of an (upper or lower) bidiagonal matrix using the implicit zero-shift QR algorithm. If this algorithm fails to find all the singular values of $\boldsymbol{A}$ then this method throws an **eInternalError** LinearAlgebraException.

This method computes the 2-norm of a vector using routine `dlange_()` from CLAPACK v 3.2.1.

4.3.6 Rank

The **column rank** of an m x n matrix $\boldsymbol{A}$ is the maximum number of linearly independent column vectors of A. The **row rank** of an m x n matrix $\boldsymbol{A}$ is the maximum number of linearly independent row vectors of $\boldsymbol{A}$. The column rank and the row rank are equal, and this is called the **rank** of the matrix $\boldsymbol{A}$. The rank of an m x n matrix $\boldsymbol{A}$ is also the number of nonzero singular values of $\boldsymbol{A}$.

The class method

```
LinearAlgebra.Rank(A)
```

returns the rank of a m x n matrix $\boldsymbol{A}$. This method computes the min(m, n) singular values of $\boldsymbol{A}$ using routine `dgesvd_()` from CLAPACK v 3.2.1. The rank is then determined as the number of computed singular values that are significantly greater than zero.

The Ponderosa Computing Linear Algebra .NET class library uses the threshold value

$$threshold = \max(sfmin\,,\ \max(m,n) * eps * \sigma_{\max}(\boldsymbol{A})) \quad .$$

A computed singular value is considered significantly greater than zero if it exceeds this threshold. Here $safemin$ is the minimum positive floating point value such that $1/safemin$ does not overflow, eps is the relative machine precision, and $\sigma_{\max}(\boldsymbol{A})$ is the largest singular value of $\boldsymbol{A}$.

Routine `dgesvd_()` uses routine `dbdsqr_()` to compute the singular values of an (upper or lower) bidiagonal matrix using the implicit zero-shift QR algorithm. If this algorithm fails to find all the singular values of $\boldsymbol{A}$ then this method throws an **eInternalError** LinearAlgebraException.

4.3.7 Spectral Radius

Let $\lambda_1, \lambda_2, \ldots, \lambda_n$ be the (real) eigenvalues of an n x n symmetric (real) matrix $\boldsymbol{A}$. Then the spectral radius of $\boldsymbol{A}$ is:

$$\rho(\boldsymbol{A}) = \max_i(|\lambda_i|) .$$

The class method

`LinearAlgebra.SpectralRadius(A)`

returns the spectral radius of a symmetric matrix $\boldsymbol{A}$. If $\boldsymbol{A}$ is not symmetric then this method throws an **eBadParamError** LinearAlgebraException.

This method computes the eigenvalues of $\boldsymbol{A}$ using routine `dsyev_()` from CLAPACK v 3.2.1. Routine `dsyev_()` uses routine `dsterf_()` which computes all eigenvalues of a symmetric tridiagonal matrix using the Pal-Walker-Kahan variant of the QL or QR algorithm. If this algorithm fails to find all of the eigenvalues of $\boldsymbol{A}$ in at most 30*n iterations then this method throws an **eInternalError** LinearAlgebraException.

4.3.8 Trace

The trace of an n x n matrix $\boldsymbol{A}$ is the sum of the main diagonal elements of $\boldsymbol{A}$:

$$tr(\boldsymbol{A}) = \sum_{i=1}^{n} a_{ii}$$

The class method

`LinearAlgebra.Trace(A)`

returns the trace of a square matrix $\boldsymbol{A}$ directly from its definition. If $\boldsymbol{A}$ is not square then this method throws an **eBadParamError** LinearAlgebraException.

4.3.9 1-Norm Inverse Condition Number Estimate

The class method

```
    LinearAlgebra.InverseOneNormConditionNumberEstimate(A)
```

returns the inverse of a 1-norm condition number estimate of a square matrix $\boldsymbol{A}$. If $\boldsymbol{A}$ is not square this method throws an **eBadParamError** LinearAlgebraException.

This method starts by computing $\|\boldsymbol{A}\|_1$ using routine `dlange_()` from CLAPACK v 3.2.1 and returns zero if $\|\boldsymbol{A}\|_1 = 0$.

This method then uses routine `dgetrf_()` from CLAPACK v 3.2.1 to compute the LU factorization $\boldsymbol{A} = \boldsymbol{P}\ \boldsymbol{L}\ \boldsymbol{U}$ using partial pivoting with row interchanges. If routine `dgetrf_()` determines the matrix $\boldsymbol{A}$ is exactly singular then this method returns 0.

Otherwise, this method estimates $\left\|\boldsymbol{A}^{-\mathbf{1}}\right\|_1$ using the LU factorization and routine `dgecon_()` from CLAPACK v 3.2.1 and returns zero if the $\left\|\boldsymbol{A}^{-\mathbf{1}}\right\|_1$ estimate is zero. Otherwise this method returns the inverse condition number estimate

$$c = \frac{1}{\|\boldsymbol{A}\|_1 \|\boldsymbol{A}^{-\mathbf{1}}\|_1}$$

`LinearAlgebra.InverseOneNormConditionNumberEstimate()` returns the inverse of the 1-norm condition number estimate of a matrix, rather than the condition number estimate of the matrix, itself, to provide a zero return value for singular matrices.

4.3.10 Infinity-Norm Inverse Condition Number Estimate

The class method

```
    LinearAlgebra.InverseInfinityNormConditionNumberEstimate(A)
```

returns the inverse of an infinity-norm condition number estimate of a square matrix $\boldsymbol{A}$. If $\boldsymbol{A}$ is not square this method throws an **eBadParamError** LinearAlgebraException.

This method starts by computing $\|\boldsymbol{A}\|_\infty$ using routine `dlange_()` from CLAPACK v 3.2.1 and returns zero if $\|\boldsymbol{A}\|_\infty = 0$.

This method then uses routine `dgetrf_()` from CLAPACK v 3.2.1 to compute the LU factorization $\boldsymbol{A} = \boldsymbol{P}\ \boldsymbol{L}\ \boldsymbol{U}$ using partial pivoting with row interchanges. If routine `dgetrf_()`determines the matrix $\boldsymbol{A}$ is exactly singular then this method returns 0.

Otherwise, This method estimates $\left\|\boldsymbol{A}^{-1}\right\|_\infty$ using the LU factorization and routine `dgecon_()` from CLAPACK v 3.2.1 and returns zero if the $\left\|\boldsymbol{A}^{-1}\right\|_\infty$ estimate is zero. Otherwise This method returns the inverse condition number estimate

$$c = \frac{1}{\|\boldsymbol{A}\|_\infty \left\|\boldsymbol{A}^{-1}\right\|_\infty}$$

`LinearAlgebra.InverseInfinityNormConditionNumberEstimate()` returns the inverse of the infinity-norm condition number estimate of a matrix, rather than the condition number estimate of the matrix, itself, to provide a zero return value for singular matrices.

4.3.11 Determinant

The class method

```
LinearAlgebra.Determinant(A)
```

returns the determinant of a square matrix $\boldsymbol{A}$. If $\boldsymbol{A}$ is not square then this method throws an **eBadParamError** LinearAlgebraException.

This method computes the determinant of $\boldsymbol{A}$ by using routine `dgetrf_()` from CLAPACK v 3.2.1 to compute the LU factorization $\boldsymbol{A} = \boldsymbol{P}\,\boldsymbol{L}\,\boldsymbol{U}$ using partial pivoting with row interchanges. If routine `dgetrf_()`determines the matrix $\boldsymbol{A}$ is exactly singular then this method returns 0. Otherwise, This method accumulates the product of the diagonal entries of $\boldsymbol{U}$ with sign adjustment according to pivot row interchanges and scaling to avoid intermediate overflow.

4.4 Full-Rank Square Linear System Solvers

The Ponderosa Computing Linear Algebra .NET class library provides the following full-rank square linear system solver methods:

Full rank square system solvers	`SolveFullRankLinearSystem()`
	`SolveFullRankLinearSystemEB()`

4.4.1 System Solution

The class method

```
LinearAlgebra. SolveFullRankLinearSystem(A,b)
```

returns the solution vector $\boldsymbol{x}$ to a full-rank real linear system:

$$\boldsymbol{A}\,\boldsymbol{x} = \boldsymbol{b}$$

using equilibration and iterative refinement as needed. Here $\boldsymbol{A}$ is an n x n matrix and $\boldsymbol{b}$ is a n-element column vector. The solution vector $\boldsymbol{x}$ is a n-element column vector. If $\boldsymbol{A}$ is not square or if vector $\boldsymbol{b}$ does not have n elements this method throws an **eBadParamError** LinearAlgebraException.

The class method

```
    LinearAlgebra. SolveFullRankLinearSystem(A,B)
```

returns the solution matrix $\boldsymbol{X}$ to a full-rank real linear system:

$$\boldsymbol{A}\,\boldsymbol{X} = \boldsymbol{B}$$

using equilibration and iterative refinement as needed. Here $\boldsymbol{A}$ is an n x n matrix and $\boldsymbol{B}$ is a n x p matrix. The solution matrix $\boldsymbol{X}$ is n x p. If $\boldsymbol{A}$ is not square or if matrix $\boldsymbol{B}$ does not have n rows this method throws an **eBadParamError** LinearAlgebraException.

These methods compute the solution matrix $\boldsymbol{X}$ by using routine `dgesvx_()` from CLAPACK v 3.2.1, using equilibration and iterative refinement as needed. If routine `dgesvx_()`determines the matrix $\boldsymbol{A}$ is exactly singular then these methods throw an **eInternalError** LinearAlgebraException.

4.4.2 System Solution with Error Bounds

The class method

```
    LinearAlgebra. SolveFullRankLinearSystemEB(A,b)
```

returns the solution vector $\boldsymbol{x}$ and forward error bound estimate e to a full-rank real linear system:

$$\boldsymbol{A}\,\boldsymbol{x} = \boldsymbol{b}$$

using equilibration and iterative refinement as needed. Here $\boldsymbol{A}$ is an n x n matrix and $\boldsymbol{b}$ is a n-element column vector. The solution vector $\boldsymbol{x}$ is a n-element column vector and the forward error bound estimate e is a scalar. If $\boldsymbol{A}$ is not square or if vector $\boldsymbol{b}$ does not have n elements this method throws an **eBadParamError** LinearAlgebraException.

The computed solution vector $\boldsymbol{x}$ is returned as the first n elements of a (n+1)-element augmented solution vector $\boldsymbol{x_A}$. The forward error bound estimate e is returned as the last element of $\boldsymbol{x_A}$.

$$\boldsymbol{x_A} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \\ e \end{bmatrix}.$$

The forward error bound estimate e bounds the relative error in the computed solution. If $\boldsymbol{x}$ is the computed solution and $\boldsymbol{x_t}$ is the true solution, then the relative error is bounded by:

$$\frac{\|\boldsymbol{x}-\boldsymbol{x_t}\|_\infty}{\|\boldsymbol{x}\|_\infty} \le e$$

That is, e is an estimated upper bound for the magnitude of the largest element in $\boldsymbol{x}-\boldsymbol{x_t}$ divided by the magnitude of the largest element in $\boldsymbol{x}$. This estimate is almost always a slight overestimate of the true error.

The class method

```
    LinearAlgebra. SolveFullRankLinearSystemEB(A,B)
```

returns the solution matrix $\boldsymbol{X}$ and forward error bound estimates *e* to a full-rank real linear system:

$$\boldsymbol{A}\,\boldsymbol{X} = \boldsymbol{B}$$

using equilibration and iterative refinement as needed. Here $\boldsymbol{A}$ is an n x n matrix and $\boldsymbol{B}$ is a n x p matrix. The solution matrix $\boldsymbol{X}$ is n x p and the forward error bound estimates *e* is a row vector. If $\boldsymbol{A}$ is not square or if matrix $\boldsymbol{B}$ does not have n rows this method throws an **eBadParamError** LinearAlgebraException.

The computed matrix solution $\boldsymbol{X}$ is returned as the first n rows of a (n+1) x p augmented solution matrix $\boldsymbol{X_A}$. The forward error bound estimate row vector *e* is returned as the last row of $\boldsymbol{X_A}$.

$$\boldsymbol{X_A} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1p} \\ x_{21} & x_{22} & \cdots & x_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{np} \\ e_1 & e_2 & \cdots & e_p \end{bmatrix}.$$

For each column j of the augmented solution matrix, the forward error bound estimate e_j bounds the relative error in the computed solution column $\boldsymbol{x_j}$. If $\boldsymbol{x_j}$ is the computed solution column and $\boldsymbol{x_{jt}}$ is the true solution column, then the relative error for that column is bounded by:

$$\frac{\left\|\boldsymbol{x_j}-\boldsymbol{x_{jt}}\right\|_\infty}{\left\|\boldsymbol{x_j}\right\|_\infty} \le e_j$$

That is, e_j is an estimated upper bound for the magnitude of the largest element in $\boldsymbol{x_j}-\boldsymbol{x_{jt}}$ divided by the magnitude of the largest element in $\boldsymbol{x_j}$. This estimate is almost always a slight overestimate of the true error.

These methods compute the augmented solution matrix $\boldsymbol{X}$ by using routine `dgesvx_()` from CLAPACK v 3.2.1, using equilibration and iterative refinement as needed. If routine `dgesvx_()`determines the matrix $\boldsymbol{A}$ is exactly singular then these methods throw an **eInternalError** LinearAlgebraException.

4.5 General Least-Squares System Solvers

The Ponderosa Computing Linear Algebra .NET class library provides the following real linear least squares system solver methods:

General least-squares system solvers (via LQ/QR)	`SolveLsqLinearSystem()` `SolveLsqLinearSystemEB()`
General least-squares system solvers (via SVD)	`SolveLsqLinearSystemSvd()` `SolveLsqLinearSystemSvdEB()`

4.5.1 General Least-Squares System Solvers (via Orthogonal Factorizations)

4.5.1.1 System Solution

The class method

```
LinearAlgebra.SolveLsqLinearSystem(A,b)
```

returns the minimum-norm solution vector $\boldsymbol{x}$ to a real linear least squares system:

$$\mathbf{min}_x \|\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}\|_2 \ , with\ minimum\ \|\boldsymbol{x}\|_2 \ .$$

Here $\boldsymbol{A}$ is an m x n matrix which may be rank-deficient and $\boldsymbol{b}$ is an m-element column vector. The solution is the minimum-2-norm n-element column vector $\boldsymbol{x}$ achieving the minimum 2-norm residual. This method uses a complete orthogonal factorization of $\boldsymbol{A}$. If the number of rows of $\boldsymbol{A}$ does not match the number of elements of $\boldsymbol{b}$ then this method throws an **eBadParamError** LinearAlgebraException.

Multiple real linear least squares systems sharing the same system matrix $\boldsymbol{A}$ can be solved in a single method call. The p right hand side m-element column vectors $\boldsymbol{b_j}$ for the least squares systems:

$$\min_{\boldsymbol{x}} \left\|\boldsymbol{b_j} - \boldsymbol{A}\boldsymbol{x_j}\right\|_2 \ , with\ minimum\ \left\|\boldsymbol{x_j}\right\|_2, \qquad j = 1 \dots p$$

are stored as the columns of a m x p matrix $\boldsymbol{B}$.

The class method:

`LinearAlgebra.SolveLsqLinearSystem(A,B)`

returns the minimum-norm solution vector $\boldsymbol{x_j}$ for the j-th real linear least squares system as the j-th column of the n x p solution matrix $\boldsymbol{X}$. If the number of rows of $\boldsymbol{A}$ does not match the number of rows of $\boldsymbol{B}$ then this method throws an **eBadParamError** LinearAlgebraException.

These methods compute the solution vector $\boldsymbol{x}$ and matrix $\boldsymbol{X}$ by using routine `dgelsy_()` from CLAPACK v 3.2.1. This routine first computes a QR factorization of $\boldsymbol{A}$ with column pivoting. It then determines the effective rank r of $\boldsymbol{A}$ and then refactors a reduced system into an r x r triangular system which is then solved. If routine `dgelsy_()`determines the matrix $\boldsymbol{A}$ has effective rank 0 then these methods return the zero solution vector $\boldsymbol{x}$ or matrix $\boldsymbol{X}$.

4.5.1.2 System Solution with Error Bounds

The class method

`LinearAlgebra.SolveLsqLinearSystemEB(A,b)`

returns the minimum-norm solution vector x and forward error bound estimate e to a real linear least squares system:

$$\mathbf{min}_x \|\boldsymbol{b} - \boldsymbol{Ax}\|_2 \ , with\ minimum\ \|\boldsymbol{x}\|_2 \ .$$

Here $\boldsymbol{A}$ is an m x n matrix which may be rank-deficient and $\boldsymbol{b}$ is an m-element column vector. The solution is the minimum-2-norm n-element column vector $\boldsymbol{x}$ achieving the minimum 2-norm residual. The forward error bound estimate e is a scalar. This method uses a complete orthogonal factorization of $\boldsymbol{A}$. If the number of rows of $\boldsymbol{A}$ does not match the number of elements of $\boldsymbol{b}$ then this method throws an **eBadParamError** LinearAlgebraException.

The computed solution vector x is returned as the first n elements of a (n+1)-element augmented solution vector $\boldsymbol{x_A}$. If the matrix $\boldsymbol{A}$ has m ≥ n (we have an overdetermined system) with full column rank then this method also returns a forward error bound estimate e as the last element of x_A. Otherwise this method returns the value -1 as the last element of $\boldsymbol{x_A}$.

$$\boldsymbol{x_A} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \\ e \end{bmatrix} .$$

The forward error bound estimate e bounds the relative error in the computed solution. If $\boldsymbol{x}$ is the computed solution and $\boldsymbol{x_t}$ is the true solution, then the relative error is bounded by:

$$\frac{\|\boldsymbol{x} - \boldsymbol{x_t}\|_2}{\|\boldsymbol{x_t}\|_2} \le e$$

That is, e is an estimated upper bound for the root mean square of $\boldsymbol{x} - \boldsymbol{x_t}$ divided by the root mean square of $\boldsymbol{x_t}$. This estimate is almost always a slight overestimate of the true error.

Multiple real linear least squares systems sharing the same system matrix $\boldsymbol{A}$ can be solved with forward error bound estimates in a single method call. The p right hand side m-element column vectors b_j for the least squares systems:

$$\min_{\boldsymbol{x}} \left\| \boldsymbol{b_j} - \boldsymbol{A}\boldsymbol{x_j} \right\|_{\boldsymbol{2}}\ , with\ minimum\ \left\| \boldsymbol{x_j} \right\|_2, \qquad j = 1 \dots p$$

are stored as the columns of a m x p matrix $\boldsymbol{B}$.

The class method:

```
    LinearAlgebra.SolveLsqLinearSystemEB(A,B)
```

returns the minimum-norm solution vector $\boldsymbol{x_j}$ for the j-th real linear least squares system as the j-th column of the first n rows of the augmented (n+1) x p augmented solution matrix $\boldsymbol{X_A}$. If the matrix $\boldsymbol{A}$ has m ≥ n (we have an overdetermined system) with full column rank then this method also returns a forward error bound estimate e_j returned as the last element of column j of $\boldsymbol{X_A}$. Otherwise this method returns the value -1 as the last element of column j of $\boldsymbol{X_A}$.

$$\boldsymbol{X_A} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1p} \\ x_{21} & x_{22} & \cdots & x_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{np} \\ e_1 & e_2 & \cdots & e_p \end{bmatrix}.$$

If the number of rows of $\boldsymbol{A}$ does not match the number of rows of $\boldsymbol{B}$ then this method throws an **eBadParamError** LinearAlgebraException.

For each computed solution vector $\boldsymbol{x_j}$, the forward error bound estimate e_j bounds the relative error in the solution $\boldsymbol{x_j}$. If $\boldsymbol{x_j}$ is the computed solution vector and $\boldsymbol{x_{jt}}$ is the true solution vector, then the relative error for that solution vector is bounded by:

$$\frac{\left\| \boldsymbol{x_j} - \boldsymbol{x_{jt}} \right\|_2}{\left\| \boldsymbol{x_{jt}} \right\|_2} \le e$$

That is, e_j is an estimated upper bound for the root mean square of $\boldsymbol{x_j} - \boldsymbol{x_{jt}}$ divided by the root mean square of $\boldsymbol{x_{jt}}$. This estimate is almost always a slight overestimate of the true error.

These methods compute the solution vector $\boldsymbol{x}$ and matrix $\boldsymbol{X}$ by using routine `dgelsy_()` from CLAPACK v 3.2.1. This routine first computes a QR factorization of $\boldsymbol{A}$ with column pivoting. It then determines the effective rank r of $\boldsymbol{A}$ and then refactors a reduced system into an r x r triangular system which is then solved. If routine `dgelsy_()`determines the matrix $\boldsymbol{A}$ has effective rank 0 then these methods return the zero solution vector $\boldsymbol{x}$ or matrix $\boldsymbol{X}$.

The error bounds for the full column rank overdetermined case are computed in the manner described here: http://www.netlib.org/lapack/lug/node82.html.

4.5.2 General Least-Squares System Solvers (via SVD)

4.5.2.1 System Solution

The class method

`LinearAlgebra.SolveSvdLinearSystem(A,b)`

returns the minimum-norm solution vector $\boldsymbol{x}$ to a real linear least squares system:

$$\mathbf{min}_x \|\boldsymbol{b} - \boldsymbol{Ax}\|_2 \text{ ,} \mathit{with\ minimum}\ \|\boldsymbol{x}\|_2 \text{ .}$$

Here $\boldsymbol{A}$ is an m x n matrix which may be rank-deficient and $\boldsymbol{b}$ is an m-element column vector. The solution is the minimum-2-norm n-element column vector $\boldsymbol{x}$ achieving the minimum 2-norm residual. This method uses the singular value decomposition of $\boldsymbol{A}$. If the number of rows of $\boldsymbol{A}$ does not match the number of elements of $\boldsymbol{b}$ then this method throws an **eBadParamError** LinearAlgebraException.

Multiple real linear least squares systems sharing the same system matrix $\boldsymbol{A}$ can be solved in a single method call. The p right hand side m-element column vectors $\boldsymbol{b_j}$ for the least squares systems:

$$\mathbf{\min_{\boldsymbol{x}}} \left\|\boldsymbol{b}_j - \boldsymbol{Ax}_j\right\|_2 \text{ ,} \mathit{with\ minimum}\ \left\|\boldsymbol{x}_j\right\|_2, \qquad j = 1 \dots p$$

are stored as the columns of a m x p matrix $\boldsymbol{B}$.

The class method:

`LinearAlgebra.SolveLsqLinearSystem(A,B)`

returns the minimum-norm solution vector $\boldsymbol{x_j}$ for the j-th real linear least squares system as the j-th column of the n x p solution matrix $\boldsymbol{X}$. If the number of rows of $\boldsymbol{A}$ does not match the number of rows of $\boldsymbol{B}$ then this method throws an **eBadParamError** LinearAlgebraException.

These methods compute the solution vector $\boldsymbol{x}$ and matrix $\boldsymbol{X}$ by using routine `dgelss_()` from CLAPACK v 3.2.1. This routine first computes the singular value decomposition of $\boldsymbol{A}$. It then determines the effective rank r of $\boldsymbol{A}$, zeros out the insignificant portion of the SVD, and solves the reduced system. If routine `dgelss_()`determines the matrix $\boldsymbol{A}$ has effective rank 0 then these methods return the zero solution vector $\boldsymbol{x}$ or matrix $\boldsymbol{X}$.

4.5.2.2 System Solution with Error Bounds

The class method

```
LinearAlgebra.SolveSvdLinearSystemEB(A,b)
```

returns the minimum-norm solution vector $\boldsymbol{x}$ and forward error bound estimate e to a real linear least squares system:

$$\mathbf{min}_x \|\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}\|_2 \ , with\ minimum\ \|\boldsymbol{x}\|_2 \ .$$

Here $\boldsymbol{A}$ is an m x n matrix which may be rank-deficient and $\boldsymbol{b}$ is an m-element column vector. The solution is the minimum-2-norm n-element column vector $\boldsymbol{x}$ achieving the minimum 2-norm residual. The forward error bound estimate e is a scalar. This method uses the singular value decomposition of $\boldsymbol{A}$. If the number of rows of $\boldsymbol{A}$ does not match the number of elements of $\boldsymbol{b}$ then this method throws an **eBadParamError** LinearAlgebraException.

The computed solution vector $\boldsymbol{x}$ is returned as the first n elements of a (n+1)-element augmented solution vector $\boldsymbol{x_A}$. If the matrix $\boldsymbol{A}$ has m ≥ n (we have an overdetermined system) with full column rank then this method also returns a forward error bound estimate e as the last element of $\boldsymbol{x_A}$. Otherwise this method returns the value -1 as the last element of $\boldsymbol{x_A}$.

$$\boldsymbol{x_A} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \\ e \end{bmatrix} .$$

The forward error bound estimate e bounds the relative error in the computed solution. If $\boldsymbol{x}$ is the computed solution and $\boldsymbol{x_t}$ is the true solution, then the relative error is bounded by:

$$\frac{\|\boldsymbol{x} - \boldsymbol{x_t}\|_2}{\|\boldsymbol{x_t}\|_2} \leq e$$

That is, e is an estimated upper bound for the root mean square of $\boldsymbol{x} - \boldsymbol{x_t}$ divided by the root mean square of $\boldsymbol{x_t}$. This estimate is almost always a slight overestimate of the true error.

Multiple real linear least squares systems sharing the same system matrix $\boldsymbol{A}$ can be solved with forward error bound estimates in a single method call. The p right hand side m-element column vectors $\boldsymbol{b_j}$ for the least squares systems:

$$\min_{x} \left\|\boldsymbol{b_j} - \boldsymbol{A}\boldsymbol{x_j}\right\|_2 \ , with\ minimum\ \left\|\boldsymbol{x_j}\right\|_2, \qquad j = 1 \dots p$$

are stored as the columns of a m x p matrix $\boldsymbol{B}$.

The class method:

```
LinearAlgebra.SolveSvdLinearSystemEB(A,B)
```

returns the minimum-norm solution vector $\boldsymbol{x_j}$ for the j-th real linear least squares system as the j-th column of the first n rows of the augmented (n+1) x p augmented solution matrix $\boldsymbol{X_A}$. If the matrix $\boldsymbol{A}$ has m ≥ n (we have an overdetermined system) with full column rank then this method also returns a forward error bound estimate e_j returned as the last element of column j of $\boldsymbol{X_A}$. Otherwise this method returns the value -1 as the last element of column j of $\boldsymbol{X_A}$.

$$\boldsymbol{X_A} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1p} \\ x_{21} & x_{22} & \cdots & x_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{np} \\ e_1 & e_2 & \cdots & e_p \end{bmatrix}.$$

If the number of rows of $\boldsymbol{A}$ does not match the number of rows of $\boldsymbol{B}$ then this method throws an **eBadParamError** LinearAlgebraException.

For each computed solution vector $\boldsymbol{x_j}$, the forward error bound estimate e_j bounds the relative error in the solution $\boldsymbol{x_j}$. If $\boldsymbol{x_j}$ is the computed solution vector and $\boldsymbol{x_{jt}}$ is the true solution vector, then the relative error for that solution vector is bounded by:

$$\frac{\left\|\boldsymbol{x_j} - \boldsymbol{x_{jt}}\right\|_2}{\left\|\boldsymbol{x_{jt}}\right\|_2} \le e$$

That is, e_j is an estimated upper bound for the root mean square of $\boldsymbol{x_j} - \boldsymbol{x_{jt}}$ divided by the root mean square of $\boldsymbol{x_{jt}}$. This estimate is almost always a slight overestimate of the true error.

These methods compute the solution vector $\boldsymbol{x}$ and matrix $\boldsymbol{X}$ by using routine `dgelss_()` from CLAPACK v 3.2.1. This routine first computes the singular value decomposition of $\boldsymbol{A}$. It then determines the effective rank r of $\boldsymbol{A}$, zeros out the insignificant portion of the SVD, and solves the reduced system. If routine `dgelss_()`determines the matrix $\boldsymbol{A}$ has effective rank 0 then these methods return the zero solution vector $\boldsymbol{x}$ or matrix $\boldsymbol{X}$.

The error bounds for the full column rank overdetermined case are computed in the manner described here: http://www.netlib.org/lapack/lug/node82.html.

4.6 Singular Value Decomposition

The **singular value decomposition** of an m x n (real) matrix $\boldsymbol{A}$ is a factorization of $\boldsymbol{A}$ into the form:

$$\boldsymbol{A} = \boldsymbol{U}\,\boldsymbol{\Sigma}\,\boldsymbol{V}^T$$

where $\boldsymbol{U}$ is a m x m matrix, $\boldsymbol{V}^T$ is an n x n matrix, and $\boldsymbol{\Sigma}$ is a m x n rectangular diagonal matrix with p = min(m,n) nonnegative entries on the main diagonal.

The nonnegative diagonal entries of $\boldsymbol{\Sigma}$ are the **singular values** of $\boldsymbol{A}$, $\{\sigma_1, \sigma_2, \ldots \sigma_{\min(m,n)}\}$, arranged in descending order: $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_{\min(m,n)} \geq 0$.

The matrices $\boldsymbol{U}$ and $\boldsymbol{V}$ are orthonormal:

$$\boldsymbol{U}^T\boldsymbol{U} = \boldsymbol{U}\,\boldsymbol{U}^T = \boldsymbol{I_m}\,, \qquad \boldsymbol{V}^T\boldsymbol{V} = \boldsymbol{V}\,\boldsymbol{V}^T = \boldsymbol{I_n}\,,$$

where $\boldsymbol{I_m}$ and $\boldsymbol{I_n}$ denote the identity matrices of order m and n, respectively. The m columns of $\boldsymbol{U}$, $\{\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots \boldsymbol{u}_m\}$, are called the **left singular vectors** of $\boldsymbol{A}$. The n columns of $\boldsymbol{V}$, $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots \boldsymbol{v}_n\}$, are called the **right singular vectors** of $\boldsymbol{A}$.

Since $\boldsymbol{\Sigma}$ is a m x n rectangular diagonal matrix with p = min(m,n), the singular value decomposition can be represented as the **thin singular value decomposition**:

$$\boldsymbol{A} = \boldsymbol{U_p}\,\boldsymbol{\Sigma_p}\,\boldsymbol{V_p}^T\,,$$

Where $\boldsymbol{U_p}$ is the m x p matrix consisting of the first p columns of $\boldsymbol{U}$, $\boldsymbol{\Sigma_p}$ is the p x p upper diagonal portion of $\boldsymbol{\Sigma}$, and $\boldsymbol{V_p}$ is the n x p matrix consisting of the first p columns of $\boldsymbol{V}$. The p columns of $\boldsymbol{U_p}$ and the p columns of $\boldsymbol{V_p}$ are orthonormal:

$$\boldsymbol{U_p}^T\boldsymbol{U_p} = \boldsymbol{I_p}\,, \qquad \boldsymbol{V_p}^T\boldsymbol{V_p} = \boldsymbol{I_p}\,,$$

where $\boldsymbol{I_p}$ denote the identity matrix of order p.

The Ponderosa Computing Linear Algebra .NET class library provides the following methods computing the thin singular value decomposition or parts thereof of a matrix:

Singular value decomposition	`SingularValueDecomposition()`

Singular values	`SingularValues()`
	`SingularValuesEB()`

Left or right singular vectors	`SingularValuesLeftVectors()`
	`SingularValuesRightVectors()`

4.6.1 Singular Value Decomposition

The class method

```
LinearAlgebra.SingularValueDecomposition(A)
```

returns the thin singular value decomposition of an m x n matrix $\boldsymbol{A}$. This function returns the p = min(m,n) singular values, their associated left singular vectors, and their associated right singular vectors of $\boldsymbol{A}$ in the p columns of a (m+n+1) x p matrix $\boldsymbol{S}$:

$$\boldsymbol{S} = \begin{bmatrix} \sigma_1 & \sigma_2 & \cdots & \sigma_p \\ u_{11} & u_{21} & \cdots & u_{p1} \\ \vdots & \vdots & \ddots & \vdots \\ u_{1m} & u_{2m} & \cdots & u_{pm} \\ v_{11} & v_{21} & \cdots & v_{p1} \\ \vdots & \vdots & \ddots & \vdots \\ v_{1n} & v_{2n} & \cdots & v_{pn} \end{bmatrix}$$

The first row of $\boldsymbol{S}$ contains the p singular values of $\boldsymbol{A}$, $\{\sigma_1, \sigma_2, \ldots \sigma_\mathrm{p}\}$, in descending order. The columns of the next m rows contain the first p left singular vectors, $\{\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots \boldsymbol{u}_p\}$. The columns of the last n rows contain the first p right singular vectors, $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots \boldsymbol{v}_p\}$.

This method computes the singular value decomposition of $\boldsymbol{A}$ using routine `dgesvd_()` from CLAPACK v 3.2.1. Routine `dgesvd_()` uses routine `dbdsqr_()` to compute the singular values and singular vectors of an (upper or lower) bidiagonal matrix using the implicit zero-shift QR algorithm. If this algorithm fails to find all the singular values of $\boldsymbol{A}$ then this method throws an **eInternalError** LinearAlgebraException.

4.6.2 Singular Values

The class method

```
LinearAlgebra.SingularValues(A)
```

returns the p = min(m,n) singular values of a m x n matrix $\boldsymbol{A}$, $\{\sigma_1, \sigma_2, \ldots \sigma_\mathrm{p}\}$, in descending order as an p-element column vector $\boldsymbol{s}$:

$$\boldsymbol{s} = \begin{bmatrix} \sigma_1 \\ \sigma_2 \\ \vdots \\ \sigma_\mathrm{p} \end{bmatrix} .$$

This method computes the singular values of $\boldsymbol{A}$ using routine `dgesvd_()` from CLAPACK v 3.2.1. Routine `dgesvd_()` uses routine `dbdsqr_()` to compute the singular values of an (upper or lower) bidiagonal matrix using the implicit zero-shift QR algorithm. If this algorithm fails to find all the singular values of $\boldsymbol{A}$ then this method throws an **eInternalError** LinearAlgebraException.

4.6.3 Singular Values With Error Bound

The class method

```
LinearAlgebra.SingularValuesEB(A)
```

returns the p = min(m,n) singular values of a m x n matrix $\boldsymbol{A}$, $\{\sigma_1, \sigma_2, \ldots \sigma_\mathrm{p}\}$, in descending order followed by a forward error bound estimate e as an (p+1)-element column vector $\boldsymbol{s_A}$:

$$\boldsymbol{s_A} = \begin{bmatrix} \sigma_1 \\ \sigma_2 \\ \vdots \\ \sigma_\mathrm{p} \\ e \end{bmatrix} .$$

The forward error bound estimate e bounds the absolute error in the computed singular values. If σ_i is the i-th computed singular value and σ_{it} is the i-th true singular value, then the absolute error in σ_i is bounded by:

$$|\sigma_i - \sigma_{it}| \le e$$

This estimate is almost always a slight overestimate of the true error.

This method computes the singular values of $\boldsymbol{A}$ using routine `dgesvd_()` from CLAPACK v 3.2.1. Routine `dgesvd_()` uses routine `dbdsqr_()` to compute the singular values of an (upper or lower) bidiagonal matrix using the implicit zero-shift QR algorithm. If this algorithm fails to find all the singular values of $\boldsymbol{A}$ then this method throws an **eInternalError** LinearAlgebraException.

The singular values forward error bound e is:

$$e = \max(sfmin\,, \max(m,n) * eps * \sigma_1) \ .$$

This error bound is described here: http://www.netlib.org/lapack/lug/node97.html, but this methods uses p(m,n) = max(m,n) in place of p(m,n) = 1 as the "modestly growing function of n" and uses machine epsilon dlamch_("P") in place of dlamch_("E"). The computation of e is similar to that found in the example program: http://www.nag.com/lapack-ex/node128.html .

4.6.4 Left Singular Vectors

The class method

```
LinearAlgebra.SingularValuesLeftVectors(A)
```

returns the p = min(m,n) singular values and first p left singular vectors of a m x n matrix $\boldsymbol{A}$ in the p columns of a (m+1) x p matrix $\boldsymbol{S}$:

$$\boldsymbol{S} = \begin{bmatrix} \sigma_1 & \sigma_2 & \cdots & \sigma_p \\ u_{11} & u_{21} & \cdots & u_{p1} \\ \vdots & \vdots & \ddots & \vdots \\ u_{1m} & u_{2m} & \cdots & u_{pm} \end{bmatrix}$$

The first row of $\boldsymbol{S}$ contains the p singular values of $\boldsymbol{A}$, $\{\sigma_1, \sigma_2, \dots \sigma_\mathrm{p}\}$, in descending order. The columns of the remaining m rows contain the first p left singular vectors, $\{\boldsymbol{u}_1, \boldsymbol{u}_2, \dots \boldsymbol{u}_p\}$.

This method computes the singular value decomposition of $\boldsymbol{A}$ using routine `dgesvd_()` from CLAPACK v 3.2.1. Routine `dgesvd_()` uses routine `dbdsqr_()` to compute the singular values and singular vectors of an (upper or lower) bidiagonal matrix using the implicit zero-shift QR algorithm. If this algorithm fails to find all the singular values of $\boldsymbol{A}$ then this method throws an **eInternalError** LinearAlgebraException.

4.6.5 Right Singular Vectors

The class method

```
LinearAlgebra.SingularValuesRightVectors(A)
```

returns the p = min(m,n) singular values and first p right singular vectors of a m x n matrix $\boldsymbol{A}$ in the p rows of a p x (n+1) matrix $\boldsymbol{S}$:

$$\boldsymbol{S} = \begin{bmatrix} \sigma_1 & v_{11} & v_{12} & \cdots & v_{1n} \\ \sigma_2 & v_{21} & v_{22} & \cdots & v_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \sigma_p & v_{p1} & v_{p2} & \cdots & v_{pn} \end{bmatrix}$$

The first column of $\boldsymbol{S}$ contains the p singular values of $\boldsymbol{A}$ in descending order. The rows of the remaining n columns of $\boldsymbol{S}$ contain the first p right singular vectors, $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \dots \boldsymbol{v}_p\}$.

This method computes the singular value decomposition of $\boldsymbol{A}$ using routine `dgesvd_()` from CLAPACK v 3.2.1. Routine `dgesvd_()` uses routine `dbdsqr_()` to compute the singular values and singular vectors of an (upper or lower) bidiagonal matrix using the implicit zero-shift QR algorithm. If this algorithm fails to find all the singular values of $\boldsymbol{A}$ then this method throws an **eInternalError** LinearAlgebraException.

4.7 Eigenvalues and Eigenvectors

A **right eigenvector** of an n x n symmetric (real) matrix $\boldsymbol{A}$ is a nonzero n-vector $\boldsymbol{v}$ such that the matrix product:

$$\boldsymbol{A}\boldsymbol{v} = \lambda \boldsymbol{v} \, .$$

Here λ is a real number (scalar) called the **eigenvalue** of $\boldsymbol{A}$ corresponding to the right eigenvector $\boldsymbol{v}$. The set of all eigenvectors of $\boldsymbol{A}$, $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots \boldsymbol{v}_n\}$, each paired with its corresponding eigenvalue, $\{\lambda_1, \lambda_2, \ldots \lambda_n\}$, is called the **eigensystem** of the matrix $\boldsymbol{A}$. It can be expressed as:

$$\boldsymbol{A}\,\boldsymbol{V} = \boldsymbol{V}\,\boldsymbol{\Lambda}.$$

Here $\boldsymbol{\Lambda}$ is a n x n diagonal matrix with diagonal entries $\{\lambda_1, \lambda_2, \ldots \lambda_n\}$ and $\boldsymbol{V}$ is a n x n matrix with columns $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots \boldsymbol{v}_n\}$.

The Ponderosa Computing Linear Algebra .NET class library provides the following methods computing the eigenvalues and right eigenvectors of a symmetric (real) matrix:

Eigenvalues	`Eigenvalues()`
	`EigenvaluesEB()`

Eigenvalues and right eigenvectors	`EigenvaluesRightEigenvectors()`

4.7.1 Eigenvalues

The class method

```
LinearAlgebra.Eigenvalues(A)
```

returns the eigenvalues of an n x n symmetric (real) matrix $\boldsymbol{A}$, $\{\lambda_1, \lambda_2, \ldots \lambda_n\}$, in ascending order, $\lambda_1 \le \lambda_2 \le \cdots \le \lambda_n$, as an n-element column vector $\boldsymbol{\lambda}$:

$$\boldsymbol{\lambda} = \begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \vdots \\ \lambda_n \end{bmatrix}.$$

If $\boldsymbol{A}$ is not symmetric then this method throws an **eBadParamError** LinearAlgebraException.

This method computes the eigenvalues of $\boldsymbol{A}$ using routine `dsyev_()` from CLAPACK v 3.2.1. Routine `dsyev_()` uses routine `dsterf_()` which computes all eigenvalues of a symmetric tridiagonal matrix using the Pal-Walker-Kahan variant of the QL or QR algorithm. If this algorithm fails to find all of the eigenvalues of $\boldsymbol{A}$ in at most 30*n iterations then this method throws an **eInternalError** LinearAlgebraException.

4.7.2 Eigenvalues with Error Bounds

The class method

```
LinearAlgebra.EigenvaluesEB(A)
```

returns the eigenvalues of an n x n symmetric (real) matrix $\boldsymbol{A}$, $\{\lambda_1, \lambda_2, \ldots \lambda_n\}$, in ascending order, $\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_\mathrm{n}$, followed by a forward error bound estimate e as an (n+1)-element column vector $\boldsymbol{\lambda_A}$:

$$\boldsymbol{\lambda_A} = \begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \vdots \\ \lambda_\mathrm{n} \\ e \end{bmatrix} .$$

If $\boldsymbol{A}$ is not symmetric then this method throws an **eBadParamError** LinearAlgebraException.

The forward error bound estimate e bounds the absolute error in the computed eigenvalues. If λ_i is the i-th computed eigenvalue and λ_{it} is the i-th true eigenvalue, then the absolute error in λ_i is bounded by:

$$|\lambda_i - \lambda_{it}| \leq e$$

This estimate is almost always a slight overestimate of the true error.

This method computes the eigenvalues of $\boldsymbol{A}$ using routine `dsyev_()` from CLAPACK v 3.2.1. Routine `dsyev_()` uses routine `dsterf_()` which computes all eigenvalues of a symmetric tridiagonal matrix using the Pal-Walker-Kahan variant of the QL or QR algorithm. If this algorithm fails to find all of the eigenvalues of $\boldsymbol{A}$ in at most 30*n iterations then this method throws an **eInternalError** LinearAlgebraException.

The eigenvalues forward error bound e is:

$$e = \max(sfmin\,,\ n * eps * \max_i(|\lambda_i|)) \quad .$$

This error bound is described here: http://www.netlib.org/lapack/lug/node90.html, but this method uses p(n) = n in place of p(n) = 1 as the "modestly growing function of n" and uses machine epsilon dlamch_("P") in place of dlamch_("E"). The computation of e is similar to that found in the example program: http://www.nag.com/lapack-ex/node71.html.

4.7.3 Eigenvalues and Right Eigenvectors

The class method

`LinearAlgebra.EigenvaluesRightEigenvectors(A)`

returns the eigenvalues and right eigenvectors of an n x n symmetric (real) matrix $\boldsymbol{A}$ in the n columns of the (n+1) x n matrix $\boldsymbol{V_A}$.

$$\boldsymbol{V_A} = \begin{bmatrix} \lambda_1 & \lambda_2 & \cdots & \lambda_n \\ v_{11} & v_{21} & \cdots & v_{n1} \\ \vdots & \vdots & \ddots & \vdots \\ v_{1n} & v_{2n} & \cdots & v_{nn} \end{bmatrix}$$

The first row of $\boldsymbol{V_A}$ contains the n eigenvalues of $\boldsymbol{A}$, $\{\lambda_1, \lambda_2, \ldots \lambda_n\}$, in ascending order, $\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_{\mathrm{n}}$. The columns of the remaining n rows of $\boldsymbol{V_A}$ contain the corresponding n right eigenvectors of $\boldsymbol{A}$, $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots \boldsymbol{v}_n\}$.

If $\boldsymbol{A}$ is not symmetric then this method throws an **eBadParamError** LinearAlgebraException.

This method computes the eigenvalues of $\boldsymbol{A}$ using routine `dsyev_()` from CLAPACK v 3.2.1. Routine `dsyev_()` uses routine `dsterf_()` which computes all eigenvalues of a symmetric tridiagonal matrix using the Pal-Walker-Kahan variant of the QL or QR algorithm. If this algorithm fails to find all of the eigenvalues of $\boldsymbol{A}$ in at most 30*n iterations then this method throws an **eInternalError** LinearAlgebraException.

4.8 Cholesky Factorization

The **upper triangular Cholesky factorization** (or decomposition) of an n x n symmetric positive definite matrix A is the unique factorization of $\boldsymbol{A}$ into

$$\boldsymbol{A} = \boldsymbol{U}^T \boldsymbol{U}$$

where $\boldsymbol{U}$ is an n x n upper triangular matrix with positive entries on the diagonal.

The **lower triangular Cholesky factorization** (or decomposition) of an n x n symmetric positive definite matrix A is the unique factorization of $\boldsymbol{A}$ into

$$\boldsymbol{A} = \boldsymbol{L}\,\boldsymbol{L}^T$$

where $\boldsymbol{L}$ is an n x n lower triangular matrix with positive entries on the diagonal.

The Ponderosa Computing Linear Algebra .NET class library provides the following methods computing the Cholesky factorizations of a symmetric positive definite matrix:

Upper Cholesky factorization	`UpperCholeskyFactorization()`
Lower Cholesky factorization	`LowerCholeskyFactorization()`

4.8.1 Upper and Lower Cholesky Factorizations

The class method

```
LinearAlgebra.UpperCholeskyFactorization(A)
```

returns the upper triangular Cholesky factorization of an n x n symmetric positive definite matrix ***A*** as an n x n upper triangular matrix ***U*** with positive entries on the diagonal.

The class method

```
    LinearAlgebra.LowerCholeskyFactorization(A)
```

returns the lower triangular Cholesky factorization of an n x n symmetric positive definite matrix ***A*** as an n x n lower triangular matrix ***L*** with positive entries on the diagonal.

These methods compute a Cholesky factorization of an n x n symmetric positive definite matrix ***A*** using routine `dpotrf_()` from CLAPACK v 3.2.1. If this algorithm determines the the matrix ***A*** is not positive definite then these methods throw an **eBadParamError** LinearAlgebraException.

5 References

[1] Anderson, E. et al., LAPACK Users' Guide, Third Edition (Society for Industrial and Applied Mathematics, Philadelphia, PA, 1999) ISBN 0-89871-447-8.

[2] LAPACK on the Netlib Repository at UTK and ORNL. http://www.netlib.org/lapack/

[3] CLAPACK (f2c'ed version of LAPACK) version 3.2.1 on the Netlib Repository at UTK and ORNL. http://www.netlib.org/clapack/. See clapack-3.2.1-CMAKE.tgz.

[4] CLAPACK for Windows at the University of Tennessee Innovative Computing Laboratory. http://icl.cs.utk.edu/lapack-for-windows/clapack/

[5] How to: Wrap Native Class for Use by C#. https://msdn.microsoft.com/en-us/library/ms235281.aspx

[6] Wikipedia: IEEE floating point. This discussion includes references to the standard publications.

[7] LAPACK/ScaLAPACK Development Forum. http://icl.cs.utk.edu/lapack-forum/

6 License Notice

The Ponderosa Computing Linear Algebra .NET class library was built using the CLAPACK implementation of LAPACK, a freely-available software package from netlib at http://www.netlib.org/lapack. The license used for the LAPACK software is the modified BSD license: