CSML Documentation
v1.1

hanzzoid

July 2, 2007

hanzzoid@gmx.net

Contents

1 Introduction

1.1 What is CSML?

CSML – CSharp Matrix Library – is an open source class library for the .NET environment providing basic numerical linear algebra routines. Its main component is the *Matrix*-Class containing numerous matrix calculations and manipulations such as computation of trace, determinant and norm of a matrix. You can also transpose, invert and concatenate matrices and use commonly known operators for matrix calculations.

If you ever encountered the quarrels of using MATLAB from foreign applications, you might be glad to have found this .NET solution. Since it comes open source, you have the possibility of tuning and continuing this surely neverending project.

2 The Matrix Class

2.1 Matrix Representation

Internally, an m by n matrix is saved as an ArrayList of length m of rows, each row is considered to be an ArrayList of length n. Thus, and this is important, row operations (e.g. line swapping) are always faster than column operations (e.g. column swapping) by a factor n.

This representation has – contrary to the `Complex[,]` idea – one major advantage: The matrix dimensions can be dynamically expanded. For instance, if the program tries to access the matrix entry at [3,3] of a 2 by 2 matrix, a new line and a new column of zeros are added to fit the demanded dimensions.

2.2 Constructors

`public Matrix()` — This constructor creates an empty matrix with zero lines and zero columns. Use it only if there is absolute incertainty about the dimension to work with, since dynamic expansion of matrix dimension is expensive.

`public Matrix(int m, int n)` — Creates an m by n matrix filled with zeros.

`public Matrix(int n)` — Creates an n by n matrix filled with zeros.

`public Matrix(double x)` — Creates a 1 by 1 matrix containing just x. This constructor might by useful if a vector is to be extended by one row or column, resp.

`public Matrix(Complex x)` — Creates a 1 by 1 matrix containing just x.

`public Matrix(Complex[,] values)` — Creates matrix from Complex array.

`public Matrix(double[,] values)` — Creates matrix from double array. Example:

```
Matrix M = new Matrix(new double[,] {{1, 2},{0, 3}});
```

leads to

$$M = \begin{pmatrix} 1 & 2 \\ 0 & 3 \end{pmatrix}.$$

`public Matrix(Complex[] values)` — Creates a column vector from a Complex array.

`public Matrix(double[] values)` — Creates a column vector from a double array, e.i. if n = values.Length, it produces an n by 1 matrix.
Example:

```
Matrix M = new Matrix(new double[] { -7, -3 });
```

gives

$$M = \begin{pmatrix} -7 \\ -3 \end{pmatrix}.$$

`public Matrix(string matrix_string)` — Creates matrix from string. This is to be considered *very* slow, but it is useful, when medium size matrices are to be entered by hand or read from text files.
Example:

```
Matrix M = new Matrix("1,4,5,1; 0,1,0,2; 2,7,9,3");
```

gives

$$M = \begin{pmatrix} 1 & 4 & 5 & 1 \\ 0 & 1 & 0 & 2 \\ 2 & 7 & 9 & 3 \end{pmatrix}.$$

Rows are delimited by a semicolon, the elements within each row are delimited by commas. The spaces in the example above have been inserted for improved readability, they are deleted anyway.

2.3 Static Methods

2.3.1 Diag

`public static Matrix Diag(Matrix diag_vector)` creates a diagonal matrix with diagonal elements from the vector `diag_vector`, which can be either a column or a row vector.
Example:

```
Matrix M = Matrix.Diag(5*Matrix.Ones(3, 1));
```

gives

$$M = \begin{pmatrix} 5 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 5 \end{pmatrix}.$$

`public static Matrix Diag(Matrix diag_vector, int offset)` creates matrix with a shifted diagonal vector. If `offset` = 0, this is equal to the method above.

Example:

```
Matrix M = Matrix.Diag(2*Matrix.Ones(3, 1), -1)
    + Matrix.Diag(3*Matrix.Ones(3, 1), 1);
```

gives

$$M = \begin{pmatrix} 0 & 3 & 0 & 0 \\ 2 & 0 & 3 & 0 \\ 0 & 2 & 0 & 3 \\ 0 & 0 & 2 & 0 \end{pmatrix}.$$

2.3.2 Dot

`public static Complex Dot(Matrix v, Matrix w)` implements the dot product of two vectors. v and w can be either row or column vectors.

Example 1:

```
Matrix vec1 = new Matrix("1,2,3");
Matrix vec2 = new Matrix("5,7,9");
Complex d = Matrix.Dot(vec1, vec2);
```

gives $d = 5 + 14 + 27 = 46$.

Example 2 (euclidian norm via dot product):

```
double d = Complex.Sqrt(2*Matrix.Dot(Matrix.Ones(4, 1),
                2*Matrix.Ones(4, 1)).Re;
```

gives

$$d = \left| \begin{pmatrix} 2 \\ 2 \\ 2 \\ 2 \end{pmatrix} \right| = \sqrt{16} = 4.$$

2.3.3 E

`public static Matrix E(int n, int j)` retrieves the j-th canonical basis vector of the $\mathbf{R}^n$, e.i. `E(4,2)` gives

$$e_2 = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}.$$

2.3.4 Fib

`public static double Fib(int n)` computes the n-th Fibonacci-number F_n with just $8n$ multiplications by using powers of the matrix

$$M = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}.$$

It is $F_n = (M^{n-1})_{11}$.

2.3.5 HorizontalConcat

`public static Matrix HorizontalConcat(Matrix A, Matrix B)` horizontally concats two matrices A and B, which don't have to be of the same width, e.g.

```
Matrix M = Matrix.HorizontalConcat(-Matrix.Ones(1, 3),
                -2*Matrix.Ones(1, 2));
```

gives

$$M = \begin{pmatrix} -1 & -1 & -1 \\ -2 & -2 & 0 \end{pmatrix}.$$

Missing columns are filled up with zeros.

There is also the version `public static Matrix HorizontalConcat(Matrix[] A)`, which horizontally concatenates the matrices of the array A; this is a simple gernalization of the method above.

2.3.6 Identity

`public static Matrix Identity(int n)` creates the n by n identity matrix. For instance,

```
Matrix M = Matrix.Identity(3);
```

gives

$$M = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

The $n \times n$ identity matrix I can also be considered the neutral element of multiplication in the vector space $M(n \times n, K)$ of all matrices of a field K (here: complex numbers), such that for each $n \times n$ matrix X we find

$$XI = IX = X.$$

2.3.7 Ones

`public static Matrix Ones(int m, int n)` creates an m by n matrix filled with ones.

`public static Matrix Ones(int n)` creates an n by n square matrix filled with ones.

2.3.8 Random

`public static Matrix Random(int m, int n)` creates m by n matrix filled with random entries in $[0, 1]$.

`public static Matrix Random(int n)` creates n by n square matrix filled with random entries in $[0, 1]$.

2.3.9 RandomGraph

`public static Matrix RandomGraph(int n)` creates an n by n square matrix filled with random entries in $[0, 1]$ and with zeros on the main diagonal. The returned matrix can be considered as the adjacence matrix of the complete directed graph with n vertices – K^n – with random edge weights in $[0, 1]$.

Example:

```
Matrix M = Matrix.RandomGraph(3);
```

could give

$$M = \begin{pmatrix} 0 & 0.9174 & 0.0145 \\ 0.4152 & 0 & 0.6111 \\ 0.4686 & 0.2321 & 0 \end{pmatrix}.$$

See picture below.

2.3.10 Solve

`public static void Solve(b)` solves the linear equation $Ax = b$, where A is the matrix in scope, b is submitted, and the solution x is saved within b. The solving is done via `LU`-decomposition (see `LUSafe`) and `BackwardInsertion`/`ForwardInsertion` (see there as well). This method only works for square matrices.

If you have a non-square matrix A with full rank, $Ax = b$ is equivalent to

$$A^T Ax = A^T b,$$

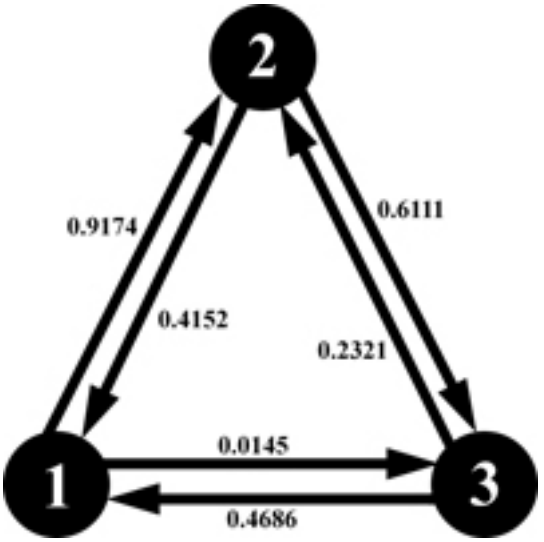


Figure 1: The graph described by the matrix above.

and since $A^T A$ is square, symmetric and positive definite, the equation above has a unique solution which can be retrieved by the `Solve` command. This solution equals the solution of $Ax = b$. If A is non-square and has not full rank, the solution space is at least one-dimensional, so....

2.3.11 TriDiag

`public static Matrix TriDiag(Complex l, Complex d, Complex u, int n)` creates an n by n tridiagonal matrix with constant values d on main diagonal, l and u on the lower and upper secondary diagonal, resp.

Example:

```
Matrix M = Matrix.TriDiag(-1,.5,-2,4);
```

gives

$$M = \begin{pmatrix} 0.5 & -2 & 0 & 0 \\ -1 & 0.5 & -2 & 0 \\ 0 & -1 & 0.5 & -2 \\ 0 & 0 & -1 & 0.5 \end{pmatrix}.$$

The generalization of the tridiag method is `public static Matrix TriDiag(Matrix l, Matrix d, Matrix u)`, which creates a tridiagonal matrix with main diagonal d, lower secondary diagonal l and upper secondary diagonal u, where u and l must vectors be of the same length (no matter whether row or column vectors), and d must have exactly one element more than the others, for instance

```
Matrix L = new Matrix("1,2,3");
Matrix D = new Matrix("-1,-4,-5,-2");
Matrix U = new Matrix("9,8,7");
```

```
Matrix M = Matrix.TriDiag(L, D, U);
```

gives

$$M = \begin{pmatrix} -1 & 9 & 0 & 0 \\ 1 & -4 & 8 & 0 \\ 0 & 2 & -5 & 7 \\ 0 & 0 & 3 & -2 \end{pmatrix}.$$

2.3.12 VerticalConcat

`public static Matrix VerticalConcat(Matrix A, Matrix B)` vertically concats two matrices A and B, which don't have to be of the same height, e.g.

```
Matrix M = Matrix.VerticalConcat(3*Matrix.Identity(2),
                -2*Matrix.Ones(3, 1));
```

gives

$$M = \begin{pmatrix} 3 & 0 & -2 \\ 0 & 3 & -2 \\ 0 & 0 & -2 \end{pmatrix}.$$

Missing lines are filled up with zeros.

There is also the version `public static Matrix VerticalConcat(Matrix[] A)`, which vertically concatenates the matrices of the array A; this is a simple gernalization of the method above.

2.3.13 Vandermonde

`public static Matrix Vandermonde(Complex[] x)` computes the specific Vandermonde matrix V for a given set x of values, where

$$V := \begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{pmatrix}.$$

2.3.14 ZeroOneRandom

`public static ZeroOneRandom(m, n, p)` generates a random matrix filled entirely with zeros and ones, where $p \in [0, 1]$ denotes the probability of an entry being one.

2.3.15 Zeros

`public static Matrix Zeros(int m, int n)` creates an m by n matrix filled with zeros.

`public static Matrix Zeros(int n)` creates an n by n square matrix filled with zeros.

2.4 Dynamic Methods

2.4.1 AbsColumnSum

`public double AbsColumnSum(int j)` computes the sum of the absolute values of column j.

2.4.2 AbsRowSum

`public double AbsRowSum(int i)` computes the sum of the absolute values of row i.

2.4.3 BackwardInsertion

`public void BackwardInsertion(Matrix b)` performs backward insertion of a vector b for a non-singular upper triangular matrix U, e.i. it solves the equation $Ux = b$, where the solution vector $x = (x_1, \ldots, x_n)$ is saved within b.

Example: Consider the linear equation system

$$\begin{cases} -x_1 + x_2 + 3x_3 = 1 \\ 2x_2 + x_3 = -2 \\ 3x_3 = -1 \end{cases}$$

This equation system is equivalent to the matrix equation

$$Ax = \begin{pmatrix} -1 & 1 & 3 \\ 0 & 2 & 1 \\ 0 & 0 & 3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ -2 \\ -1 \end{pmatrix} = b.$$

For triangular matrices, the determinant is the product of the diagonal entries, hence $det(A) = -6 \neq 0$, thus A is non-singular. We can apply backward insertion:

$$x_3 = b_3/A_{33} = -0.3333$$

$$x_2 = (b_2 - A_{23}x_3)/A_{22} = -0.8333$$

$$x_1 = (b_1 - A_{12}x_2 - A_{13}x_3)/A_{11} = -2.8333.$$

2.4.4 Cholesky

`public void Cholesky()` performs cholesky decomposition of a square, symmetric and positive definite matrix A such that

$$A = LL^T,$$

where L is a lower triangular matrix, which is stored in the lower half of A. To restore A, use `CholeskyUndo()`.

2.4.5 CholeskyUndo

`public void CholeskyUndo()` retrieves initial matrix A, if cholesky decomposition has been performed and the lower triangular matrix L being saved within the lower half of A is no longer needed.

Hint: You can also isolate L before restoring the initial matrix. Let M be symmetric and positive definite.

```
M.Cholesky(); // perform cholesky decomposition

Matrix L = M.ExtractLowerTrapeze(); // extract L

M.CholeskyUndo(); // now M has its initial content
```

2.4.6 Clear

`public Matrix Clear(int i, int j)` returns the matrix which results in clearing of row i and column j. **Warning:** This will lead to a collision, if the matrix is one by one. Since this method is exclusivly used by the `Invert()` method, no error catching has been included for the sake of velocity.

Example:

```
Matrix M = (new Matrix("1,2; 3,4")).Clear(1, 2);
```

gives

$$M = \begin{pmatrix} 3 \end{pmatrix}.$$

2.4.7 Clone

`public Matrix Clone()` returns a shallow copy of the current instance of a Matrix. The main purpose is to bypass the problems arising from reference initialization.

Example:

```
Matrix Q = Matrix.Random(4);
       Matrix R = Q;
```

This might look proper, but it isn't. If you init R this way, all changes of R directly result in the same changes at Q. The clean way is

```
Matrix Q = Matrix.Random(4);
   Matrix R = Q.Clone();
```

Now R and Q are independent.

2.4.8 Column

`public Matrix Column(int j)` returns the j-th column vector of a matrix.

2.4.9 ColumnSum

`public Complex ColumnSum(int j)` returns the sum of the entries within the j-th column of a matrix.

2.4.10 ColumnSumNorm

`public double ColumnSumNorm()` returns `this.TaxiNorm()`, see there.

2.4.11 ColumnVectorize

`public Matrix[] ColumnVectorize()` splits a matrix into an array of its column vectors.

2.4.12 Condition

`public double Condition()` computes the condition number of an invertible square matrix with respect to the column sum norm. If you want to compute the condition with respect to another norm, use the overloaded method `public double Condition(int p)`, which uses p norm. p should be in $[1, +\infty]$.

If you want to compute the condition number with respect to the frobenius norm, use the overloaded method `public double ConditionFro()`.

Anyway, keep in mind, that for any norm p

$$\kappa_p(A) = \|A\|_p \|A^{-1}\|_p,$$

and computation of the inverse is expensive.

An exception is thrown when A is not invertible.

2.4.13 ConjTranspose

`public Matrix ConjTranspose()` computes the conjugated transpose A^H of a matrix $A = (a_{ij})$, such that

$$A^H := (\widetilde{a}_{ij}) = \overline{a_{ji}},$$

e.i. A^H is the transpose (see there) of the conjugated elements of A.

2.4.14 Conjugate

`public Matrix Conjugate()` computes the conjugation of a complex Matrix, e.i. each entry $z = x + iy$ is replaced by its complex conjugated $\overline{z} = x - iy$.

2.4.15 Definiteness

`public DefinitenessType Definiteness()` checks a matrix A for definiteness. The following property of symmetric matrices is used: Let $\{y_1, \ldots, y_n\}$ be an orthogonal basis with respect to A, e.i. $y_i^T A y_j = 0 \; \forall i \neq j$, then

$$A > 0 \Leftrightarrow y_i^T A y_i > 0 \; \forall i,$$

$$A \geq 0 \Leftrightarrow y_i^T A y_i \geq 0 \; \forall i,$$

$$A < 0 \Leftrightarrow y_i^T A y_i < 0 \; \forall i,$$

$$A \leq 0 \Leftrightarrow y_i^T A y_i \leq 0 \; \forall i,$$

where $A > 0 := A$ is positive definite, $A \geq 0 := A$ is positive semidefinite, $A < 0 := A$ is negative definite, $A \leq 0 := A$ is negative semidefinite.

If none of the above-mentioned properties can be found, A is indefinite.

2.4.16 DeleteColumn

`public void DeleteColumn(int j)` removes the column with one-based index j from the matrix.

2.4.17 DeleteRow

`public void DeleteRow(int i)` removes the row with one-based index i from the matrix.

2.4.18 Determinant

`public Complex Determinant()` computes the determinant of a matrix via LU-decomposition and `DiagProd()`, which takes

$$\frac{n^3}{3} + 3n^2$$

multiplications. It is $PA = LU$ for a permutation matrix P, and

$$det(A) = sgn(P)det(U) = sgn(P) \prod_{i=1}^{n} U_{ii}.$$

This is relatively slow, since internally this method uses `LUSafe()`. If you are interested in best performance despite of possible instabilities, directly change the source code of `Determinant()`.

See `LU` for more information.

2.4.19 DiagProd

`public Complex DiagProd()` computes the product of the main diagonal entries of a matrix.

2.4.20 DiagVector

`public Matrix DiagVector()` extracts the main diagonal of a matrix as column vector.

Example:

```
Matrix b = Matrix.Identity(3).DiagVector();
```

results in

$$b = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}.$$

2.4.21 Eigenvalues

`public Matrix Eigenvalues()` computes the eigenvalues (aka spectrum) of the matrix A in scope via the lamest mule in town, known as QR iteration (see there). It can be proved that the QR iteration converges towards a matrix T whose diagonal entries are the eigenvalues of A.

2.4.22 Extract

`public Matrix Extract(i1, i2, j1, j2)` extracts a sub matrix $\widetilde{A}$ from a matrix A, such that A remains unchanged, and $\widetilde{A}$ is an $(i2 - i1 + 1)$ by $(j2 - j1 + 1)$ matrix, with

$$\widetilde{A}_{ij} = A_{i+i_1 j+j_1}.$$

2.4.23 ExtractColumn

`public Matrix ExtractColumn(int j)` retrieves a column vector with one-based index j of a matrix and deletes it from the matrix.

2.4.24 ExtractLowerTrapeze

`public Matrix ExtractLowerTrapeze()` retrieves the lower trapeze part of a matrix.

Example: For

$$M = \begin{pmatrix} -1 & 5 & 2 \\ 4 & -2 & 4 \\ -2 & 6 & 1 \end{pmatrix},$$

```
Matrix L = M.ExtractLowerTrapeze();
```

produces

$$L = \begin{pmatrix} -1 & 0 & 0 \\ 4 & -2 & 0 \\ -2 & 6 & 1 \end{pmatrix}.$$

2.4.25 ExtractRow

`public Matrix ExtractRow(int i)` retrieves a row vector with one-based index i of a matrix and deletes it from the matrix.

2.4.26 ExtractUpperTrapeze

`public Matrix ExtractUpperTrapeze()` retrieves the upper trapeze part of a matrix.

Example: For

$$M = \begin{pmatrix} -1 & 5 & 2 \\ 4 & -2 & 4 \\ -2 & 6 & 1 \end{pmatrix},$$

```
Matrix L = M.ExtractUpperTrapeze();
```

produces

$$L = \begin{pmatrix} -1 & 5 & 2 \\ 0 & -2 & 4 \\ 0 & 0 & 1 \end{pmatrix}.$$

2.4.27 ForwardInsertion

`public void ForwardInsertion(Matrix b)` performs forward insertion of a vector b for a non-singular lower triangular matrix L, e.i. it solves the equation $Lx = b$, where the solution vector $x = (x_1, \ldots, x_n)$ is saved within b.

2.4.28 FrobeniusNorm

`public double FrobeniusNorm()` computes the Frobenius norm of a matrix A:

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

2.4.29 HorizontalFlip

`public void HorizontalFlip()` horizontally flips a matrix A, e.i. the columns of A are reversed.

2.4.30 InsertColumn

`public void InsertColumn(Matrix col, int j)` inserts the vector col at column with one-based index j into a matrix A. col does not need to have the same number of rows as A, and j can be greater than the number of columns of A — new rows and columns filled with zeros are added dynamically.

Example:

```
Matrix M = Matrix.Identity(3);
M.InsertColumn(2 * Matrix.Ones(4, 1), 5);
```

gives

$$M = \begin{pmatrix} 1 & 0 & 0 & 0 & 2 \\ 0 & 1 & 0 & 0 & 2 \\ 0 & 0 & 1 & 0 & 2 \\ 0 & 0 & 0 & 0 & 2 \end{pmatrix}.$$

2.4.31 InsertRow

`public void InsertRow(Matrix row, int i)` inserts the vector row at row with one-based index i into a matrix A. row does not need to have the same number of columns as A, and i can be greater than the number of rows of A — new rows and columns filled with zeros are added dynamically.

2.4.32 Inverse

`public Matrix Inverse()` returns the inverse of a square matrix A, as long as

$$det(A) \neq 0.$$

Then, of course, the inverse matrix holds

$$A^{-1}A = AA^{-1} = I_n,$$

where I_n is the identity matrix with n rows and columns each.

This method uses the LAPLACE formula

$$A^{-1} = \frac{1}{det(A)}(\Delta_{ij})^T_{1 \leq i,j \leq n}$$

and

$$\Delta_{ij} = (-1)^{i+j} det(A^{\#}_{ij})$$

and $A^{\#}_{ij} =$ `A.Clear(i, j)`.

As `Determinant()` in its factory shape uses `LUSafe()`, the computation of the inverse is slower than necessary (for explanation of the reasons, see `LU()`). Thus, if you are working with huge matrices, or if you need to compute many inverse matrices, start speed optimization at `Determinant()`.

Caution: Matrix inversion is under most circumstances *very* expensive (exception: matrices are very small, empty, diagonal or orthogonal). Use it sparsely. Although formally the equation $Ax = b$ is solved by $x = A^{-1}b$, rather use `A.Solve(b)`, which performs LU-decomposition with column pivoting, forward and backward insertion to retrieve the solution in $\mathcal{O}(n^3)$.

Again, if speed is of primary concern, remove the checks for diagonality and orthogonality at the beginning of `Invert()` (where especially the former is expensive), if you can distinctly exclude the occurance of those special matrices. Otherwise, those checks can speed up things a little:

Orthogonal matrices are inverted in $\mathcal{O}(n^2)$, diagonal matrices are inverted in $\mathcal{O}(n)$ even.

General matrix inversion needs $\mathcal{O}(n^4)$ with a not-so-good constant, for those who know what I mean... It needs one n by n determinant and n $(n-1)$x$(n-1)$ determinants, and each involves an LU-decomposition.

2.4.33 InverseLeverrier

`public Matrix InverseLeverrier()` is an alternative approach to the inversion of matrices. The iterative formula is simpler than the LAPLACE method, less preliminary checks are necessary, and the code is clean and straightforward.

Here comes the algorithm, known as LEVERRIER formula or FADDEV formula: We want to invert a complex $n \times n$ matrix A; assume $B = I_n$, the $n \times n$ identity matrix. Then compute recursively

$$\alpha_k = \frac{1}{k} tr(AB_{k-1}),\ 0 \leq k \leq n,$$

$$B_k = -AB_{k-1} + \alpha_k I,\ 0 \leq k \leq n-1,$$

where $tr(X)$ denotes the trace of X, see there. Since $B_n = 0$, we receive

$$A^{-1} = \frac{1}{\alpha_n} B_{n-1}.$$

If $\alpha_n = 0$, an exception is thrown.

2.4.34 IsDiagonal

`public bool IsDiagonal()` checks if a matrix A is a diagonal matrix, which is true iff

$$\forall i \neq j : a_{ij} = 0.$$

The entire code for this method is

```
return (this.Clone() - Diag(this.DiagVector()) ==
          Zeros(RowCount, ColumnCount));
```

Despite of its simplicity, this check is rather expensive,and although `Clone()`, `Diag()` and `DiagVector()` have a $\mathcal{O}(n)$ complexity, `Zeros()` and the equality test == are $\mathcal{O}(n^2)$.

2.4.35 IsInvolutary

`public bool IsInvolutary()` returns true iff a matrix A is involutary, which is true iff

$$A^2 = I_n.$$

2.4.36 IsHermitian

`public bool IsHermitian()` checks if a matrix A is Hermitian, which is true for any complex $n \times n$ matrix with

$$A^H = A,$$

where A^H is the conjugated tranpose of A, see `ConjTranpose()`.

2.4.37 IsLowerTrapeze

`public bool IsLowerTrapeze()` returns true iff a matrix A is a lower trapeze matrix, which is the case iff $A_{ij} = 0$ for $i > j$.

2.4.38 IsLowerTriangular

`public bool IsLowerTriangulat()` returns true iff a matrix A is a lower triangular matrix, which is true iff A is a lower trapeze matrix and square.

2.4.39 IsNormal

`public bool IsNormal()` checks if a matrix A is normal, which is true iff

$$AA^H = A^H A,$$

where A^H is the conjugated transpose of A, see `ConjTranspose()`.

2.4.40 IsOrthogonal

`public bool IsOrthogonal()` returns true iff a matrix A is an orthogonal matrix, which is true iff A is square, real and $AA' = id$, e.i. the transpose of A equals the inverse of A.

2.4.41 IsPermutation

`public bool IsPermutation()` checks if a matrix A is a permutation of the identity matrix, which is true for this example:

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

The check is done via `IsZeroOneMatrix()` and `IsInvolutary()`, since each permutation matrix P holds $P^2 = I_n$ (compare `Identity()`).

2.4.42 IsSPD

Equivalent to `public bool IsSymmetricPositiveDefinite()`, see there.

2.4.43 IsSquare

`public bool IsSquare()` checks if a matrix A is a square matrix, e.i. if number of rows equals number of columns.

2.4.44 IsSymmetric

`public bool IsSymmetric()` checks if a matrix A is a symmetrix matrix by checking if $A_{ij} = A_{ji}$ for all i, j.

2.4.45 IsSymmetricPositiveDefinite

`public bool IsSymmetricPositiveDefinite()` checks, if a matrix is symmetric and positive definite via `Definiteness`, see there.

2.4.46 IsTrapeze

`public bool IsTrapeze()` checks if a matrix A is a trapeze matrix by checking if A is either lower or upper trapeze.

2.4.47 IsTriangular

`public bool IsTriangular()` checks if a matrix A is a triangular matrix by checking if it is lower or upper triagular.

2.4.48 IsUnitary

`public bool IsUnitary()` checks if a matrix A is unitary, with is true iff

$$A^H A = A A^H = I,$$

where A^H is the conjugated transpose of A (see `ConjTranspose()`) and I is the identity matrix (see `Identity()`). As immediate corollary, we receive

$$A^{-1} = A^H.$$

Considering the effort needed to directly compute the inverse via LAPLACE or LEVERRIER formula, especially for huge unitary matrices, it is much faster to check for unitarity first and then return the conjugated transpose ($\mathcal{O}(n^2)$ now versus $\mathcal{O}(n^4)$ before).

2.4.49 IsUpperTrapeze

`public bool IsUpperTrapeze()` checks if a matrix A is an upper trapeze matrix by checking if $A_{ij} = 0$ for $i < j$.

2.4.50 IsUpperTriangular

`public bool IsUpperTriangular()` checks if a matrix A is an upper triangular matrix by checking if A is square and upper trapeze matrix.

2.4.51 IsZeroOneMatrix

`public bool IsZeroOnematrix()` checks if a matrix A consists only of zeros and ones. This method is only used within `IsPermutation()`, see there for more explanation.

2.4.52 LU

`public void LU()` performs LU-decomposition of a quare matrix A, where L is a lower triangular matrix with ones on its diagonal, U is an upper triangular matrix and

$$A = LU.$$

Note that this method is declared void, since U is saved in the upper triangular part of A and the part of L below the diagonal is saved in the lower triangular part of A. Thereby, the orginal matrix A can easily be retrieved:

```
A = (A.EctractLowerTrapeze
+ Matrix.Diag(Matrix.Ones(A.Size()[0], 1)))
* A.ExtractUpperTrapeze;
```

2.4.53 LUSafe

`public void LUSafe()` performs LU-decomposition of A with additional column pivoting, which is sometimes vitally necessary for the gaussian elimination to work properly. Nevertheless, this leeds to addtional time $\mathcal{O}(n^2)$.

2.4.54 MaxNorm

`public Complex MaxNorm()` returns the maximum norm of a matrix A, which is also known as the row-sum norm: Let r_i be the sum of the absolute values of the entries of row i of A, then $max\{r_1, \ldots, r_m\}$ is returned.

2.4.55 Minor

`public Matrix Minor()` returns the minor $A^{\#}_{ij}$ of A, which is the matrix defined by clearing row i and column j. The minor is of central interest for the calculation of the inverse, see there for more information.

2.4.56 Norm

`public Complex Norm()` computes the Euclidian norm for vectors. If the matrix in scope is not a vector, an exception is thrown. For matrices, use `MaxNorm`, `FrobeniusNorm` or `TaxiNorm` instead.

2.4.57 Permanent

`public Complex Permanent()` computes the permanent of a square matrix A, which is defined similarly to the determinant:

$$per(A) := \sum_{\sigma \in \Pi_n} a_{1\sigma(1)} \cdots a_{n\sigma(n)},$$

where $A = (a_{ij})_{1 \leq i,j \leq n}$. The main problem with this method is: There is not a polynomial time algorithm for the computation of the permanent known at this time. Even worse, a polynomial time algorithm for per would in particular imply

P = NP.

In this algorithm, we used a Laplacian-style formula with roughly $\mathcal{O}(2^n)$, which is just way to much for any but very small matrices.

2.4.58 PNorm

`public Complex PNorm()` computes the p-norm of a matrix $A = (a_{ij})$, $1 \leq i \leq m$, $1 \leq j \leq n$, which is defined by

$$\|A\|_p := \left(\sum_{i=1}^{m} \sum_{j=1}^{n} |a_{ij}|^p \right)^{\frac{1}{p}} (p \neq \infty),$$

and $\|A\|_\infty$ is the row sum norm of A. For $p = 2$, the p-norm equals the Frobenius norm, and for $p = 1$, the p-norm equals the column-sum norm or taxi norm.

2.4.59 QRGramSchmidt

`public Matrix[] QRGramSchmidt()` computes a basic QR-decomposition of an $m \times n$ matrix A such that a 1-dimensional matrix array containing $\{Q, R\}$ is returned, where

$$A = QR,$$

and Q is $m \times n$ and orthogonal, R is $n \times n$ and upper triangular. We use a slightly modified version of the canonical Gram-Schmidtian orthogonalization.

2.4.60 Row

`public Matrix Row(i)` retrieves the row with one-based index i of the matrix in scope.

2.4.61 RowSum

`public Complex RowSum(i)` computes the sum the the elements of the row with one-based index i of the matrix in scope.

2.4.62 RowSumNorm

`public Complex RowSumNorm()` computes the row-sum norm by retrieving `this.TaxiNorm()`, see there.

2.4.63 RowVectorize

`public Matrix[] RowVectorize()` splits the matrix in scope into a matrix array of its row vectors.

2.4.64 Signum

`public short Signum()` returns the signum of a permutation matrix P (compare `IsPermutation`), which us 1 for an even number of row/column swaps and -1 for an odd number of swaps needed to get the identity matrix.

Example:

$$sgn \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = -1.$$

The method uses the well known algorithm

$$sgn(\pi) = \prod_{1<i\leq j\leq n} \frac{f(i) - f(j)}{i - j},$$

where $f(i)$ is such that $p_{f(i)i} = 1$ with $P = (p_{ij})$.

2.4.65 Spectrum

See `Eigenvalues`.

2.4.66 SwapColumns

`public void SwapColumns(j1, j2)` swaps the columns with one-based index $j1$ and $j2$ of the matrix in scope.

2.4.67 SwapRows

`public void SwapRows(i1, i2)` swaps the rows with one-based index $i1$ and $i2$ of the matrix in scope.

2.4.68 SymmetrizeDown

`public void SymmetrizeDown()` makes the given matrix symmetric by copying the upper half to the lower half, e.g. if we have

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix},$$

then `A.SymmetrizeDown()` leads to

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 5 & 6 \\ 3 & 6 & 9 \end{pmatrix}.$$

The matrix in scope must be square.

2.4.69 SymmetrizeUp

`public static void SymmetrizeUp()` makes the given matrix symmetric by copying the lower half to the upper half; compare `SymmetrizeDown`.

2.4.70 TaxiNorm

`public double TaxiNorm()` computes the taxi norm, also known as column sum norm of a matrix: If c_i is the sum of the absolute values of the entries of column number i, then $max\{c_0, \ldots, c_n\}$ is returned.

2.4.71 ToString

`public override ToString()` returns a formatted string displaying the matrix in scope.

2.4.72 Trace

`public Complex Trace()` computes the trace of a square matrix A, which is defined as the sum of the diagonal elements of A, e.i.

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

2.4.73 Transpose

`public void Transpose()` transposes a given matrix A, which is done by swapping the elements a_{ij} and a_{ji}; more precisely:

$$A^T = (a_{ij})^T = (a_{ji}).$$

2.4.74 VectorLength

`public int VectorLength()` checks if a matrix v is a vector, e.i. if at least one dimension of v is 1. If positive, the length of the v is returned, else 0 is returned.

2.4.75 VerticalFlip

`public void VerticalFlip()` vertically flips the matrix in scope, e.i. the rows are swapped.

3 The Complex Class

3.1 What are complex numbers anyway?

Many people encounter the problem of not being able to find the roots of a more than simple polynomial like

$$p(x) = x^2 + 1.$$

In school you are taught: Roots of a polynomial p are the intersections of the graph of p with the x-axis. The above mentioned polynomial obvious does not intersect the x-axis and therefore does not have any roots. Or does it? Why not define an imaginary number i via

$$i := \sqrt{-1}?$$

Then we could factorize $p(x) = x^2 + 1 = (x - i)(x + i)$, and p has two roots. In fact, the complex numbers, defined as all tuples of real numbers (x, y), where x is the real part and y is the imaginary part, are a field extension of the real numbers, that means roughly spoken: You can do anything as usual with real numbers — and more.

Contrary to real numbers, which are thought to live on a straight line, a complex number (x, y), also written as $x + iy$, can be located in the GAUSSIAN plane (see image below). Since the Eulerian formula states $e^{i\varphi} = cos(\varphi) + i\ sin(\varphi)$, we can write a complex number $z = (x, y)$ also as $z = re^{i\varphi}$, where $r = |z| = \sqrt{x^2 + y^2}$ is the distance of z to the origin and $\varphi = arctan(\frac{y}{x})$ is the argument of z, e.i. the angle enclosed with the real axis. Summarized:

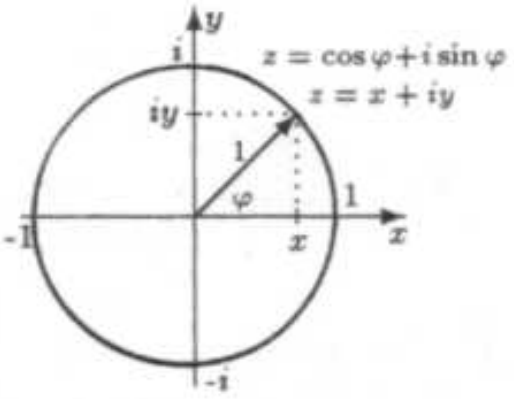
$$z = (x, y) = x + iy = re^{i\varphi} = r(cos(\varphi) + i\ sin(\varphi))$$

$$= |z|(cos(\varphi) + i\ sin(\varphi))$$

where $\overline{z} = x - iy$ is the complex conjugated of z.

3.2 Complex Number Representation

Internally, each complex number is represented as a pair of double values, `re` and `im`, corresponding to real part and imaginary part, resp. These values are both readable and writable.



3.3 Constructors

3.3.1 Empty

`public Complex()` inititates the complex number 0.

Example:

```
Complex z = new Complex();
```

and

```
Complex z = Complex.Zero;
```

are equivalent.

3.3.2 From Double Value

`public Complex(double re)` inits complex number with real part `re` and imaginary part 0.

Example:

```
Complex z = new Complex(3);
```

and

```
Complex z = 3 * Complex.One;
```

are equivalent.

3.3.3 From Pair of Double Values

`public Complex(double re, double im)` inits complex number with real part `re` and imaginary part `im`.

Example:

```
Complex z = new Complex(-2, 4);
```

and

```
Complex I = Complex.I; // imaginary unit
Complex z = -2 + 4 * I;
```

are equivalent.

3.4 Operators

3.4.1 $+$

Addition of two complex numbers $x = x_1 + ix_2$ and $y = y_1 + iy_2$ is easily done component-wise:

$$x + y = (x_1 + y_1) + i(x_2 + y_2).$$

3.4.2 $*$

Multiplication of two complex numbers $x = x_1 + ix_2$ and $y = y_1 + iy_2$ is defined by

$$xy := (x_1y_1 - x_2y_2) + i(x_1y_2 + x_2y_1).$$

3.4.3 $/$

Division of two complex numbers $x = x_1 + ix_2$ and $y = y_1 + iy_2$ is reduced to multiplication of complex numbers via

$$\frac{x}{y} = x\overline{y}\frac{1}{|y|^2}.$$

Note that $|y|^2$ is real.

3.5 Static Methods

3.5.1 Abs

`public static double Abs(Complex z)` computes the absolute value of the complex number z, which is defined by

$$|z| := \sqrt{z\overline{z}} = \sqrt{x^2 + y^2},$$

where $\overline{z} := x - iy$ and $z = x + iy$.

$|z|$ is also the distance of the point z in the GAUSSIAN plane to the origin 0. Note that the absolute value of a complex number is always real.

3.5.2 Arg

`public static double Arg(Complex z)` computes the argument of the complex number z, which is the angle enclosed by the line through 0 and z and the real axis; thus

$$\arg(z) := \arctan(\frac{y}{x}),$$

where $z = x + iy$. Note that the argument of z is always a real number.

3.5.3 Conj

`public static Complex Conj(Complex z)` returns the complex conjugate $\overline{z} = x - iy$ of a complex number $z = x + iy$.

3.5.4 Cos

`public static Complex Cos(Complex z)` computes the complex cosine of a complex number z using the formula

$$\cos(z) := \frac{1}{2}\left(e^{iz} + e^{-iz}\right).$$

3.5.5 Cosh

`public static Complex Cosh(Complex z)` computes the hyberbolic cosine of a complex number z via

$$\cosh(z) = \frac{e^z + e^{-z}}{2}.$$

3.5.6 Cot

`public static Complex Cot(Complex z)` computes the cotangent of the complex number z, which is

$$\cot(z) = \frac{\cos(z)}{\sin(z)}.$$

3.5.7 Coth

`public static Complex Coth(Complex z)` computes the hyperbolic cotangent of the complex number z via

$$\coth = \frac{\cosh(z)}{\sinh(z)} = \frac{e^{2z} + 1}{e^{2z} - 1}.$$

3.5.8 Csch

public static Complex Csch(Complex z) computes the hyperbolic cotangent of the complex number z via

$$\operatorname{csch}(z) = \frac{1}{\sinh(z)}.$$

3.5.9 Exp

`public static Complex Exp(Complex z)` computes the value of the complex exponential function e^z via

$$e^z := e^x(\cos(y) + i\,\sin(y)),$$

where $z = x + iy$. Note that *real* sin and *real* cosine are used. This is necessary, since complex sin and cosine are defined upon the complex eponential function.

3.5.10 I

`public static Complex I` returns the imaginary unit i as a complex value.

3.5.11 Inv

`public static Complex Inv(Complex z)` inverts the complex number z via

$$z^{-1} = \frac{x}{x^2 + y^2} - i\frac{y}{x^2 + y^2},$$

where $z = x + iy$. The same can be done by simply typing

```
Complex zinv = 1 / z;
```

3.5.12 Log

`public static Complex Log(Complex z)` computes the complex logarithm of a complex number z. Note that the complex logarithm is *not* one-to-one, since

$$\log(z) = \ln(|z|) + i\mathrm{arg}(z) + 2\pi ik,$$

where $k \in \mathbf{N}$ and $\ln$ the real natural logarithm (this sounds weird!).

This method returns the so-called main value of the complex logarithm, which is defined by

$$\log(z) = \ln(|z|) + i\mathrm{arg}(z).$$

3.5.13 One

`public static Complex One` returns 1 as a complex value.

3.5.14 Pow

public static Complex Pow(Complex a, Complex b) computes the complex power

$$a^b = e^{b\,\log(a)}.$$

3.5.15 Sech

`public static Complex Sech(Complex z)` computes the hyperbolic secant of the complex number z via

$$\operatorname{sech}(z) = \frac{1}{\cosh(z)}.$$

3.5.16 Sin

`public static Complex Sin(Complex z)` computes the complex sine of a complex number z using the formula

$$\sin(z) = \frac{1}{2i}\left(e^{iz} - e^{-iz}\right).$$

3.5.17 Sinh

`public static Complex Sinh(Complex z)` computes the hyberbolic sin of a complex number z via

$$\sinh(z) = \frac{e^z - e^{-z}}{2}.$$

3.5.18 Sqrt

`public static Complex Sqrt(Complex z)` computes the square root of a complex number z via

$$\sqrt{z} = z^{\frac{1}{2}}.$$

Compare `Pow`.

There is an overloaded method taking a double argument x, which returns the real square root, if x is non-negative, and which returns $-i\sqrt{x}$ if $x < 0$.

3.5.19 Tan

`public static Complex Tan(Complex z)` computes the tangent of the complex number z, which is

$$\tan(z) = \frac{\sin(z)}{\cos(z)}.$$

3.5.20 Tanh

`public static Complex Tanh(Complex z)` computes the hyperbolic tangent of the complex number z, which is

$$\tanh(z) = \frac{\sinh(z)}{\cosh(z)} = \frac{e^{2z} - 1}{e^{2z} + 1}.$$

3.5.21 Zero

`public static Complex Zero` returns 0 as a complex value.

3.6 Dynamic Methods

3.6.1 IsReal

`public bool IsReal()` returns true iff the imaginary part of the compex number in scope equals 0.

3.6.2 IsImaginary

`public bool IsImaginary()` returns true iff the real part of the compex number in scope equals 0.