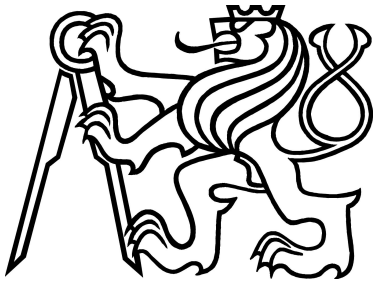
Department of Control Engineering, Faculty of Electrical Engineering,
Czech Technical University in Prague



Numerical Algorithms for Polynomial Matrices in Java

Graduate Diploma Thesis

Michal Paděra

2004

České vysoké učení technické v Praze - Fakulta elektrotechnická

Katedra řídicí techniky **Školní rok:** 2002/2003

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: Michal P a d ě r a

Obor: Technická kybernetika

Název tématu: Numerické algoritmy pro polynomiální matice v jazyce Java

Zásady pro vypracování:

1. Posuďte vhodnost jazyka Java pro numerické výpočty, vyberte knihovnu algoritmů numerické lineární algebry a zdůvodněte tento výběr.
2. Navrhněte a efektivně implementujte třídu PolynomialMatrix a její základní metody pro aritmetické operace.
3. Implementujte efektivně násobení dvou polynomiálních matic.
4. Implementujte algoritmus pro výpočet determinantu polynomiální matice pomocí rychlé Fourierovy transformace.

Seznam odborné literatury: Dodá vedoucí práce.

Vedoucí diplomové práce: Ing. Zdeněk Hurák

Datum zadání diplomové práce: prosinec 2002

Termín odevzdání diplomové práce: květen 2004

L.S.

doc. Ing. Michael Šebek, DrSc.
vedoucí katedry

v zast.

prof. Ing. Vladimír Kučera, DrSc.
děkan

V Praze dne 14.11.2003

Prohlášení

Prohlašuji, že jsem svou diplomovou (bakalářskou) práci vypracoval samostatně a použil jsem pouze podklady (literaturu, projekty, SW atd.) uvedené v přiloženém seznamu.

Nemám závažný důvod proti užití tohoto školního díla ve smyslu § 60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze, dne ....

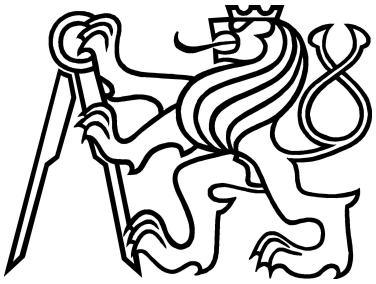
podpis

Proclamation of the Author

I hereby declare that I am the author of the proposed graduate diploma thesis and that proper references were given to the resources that I used.

Prague ....
signature

Department of Control Engineering, Faculty of Electrical Engineering,
Czech Technical University in Prague



Numerical Algorithms for Polynomial Matrices in Java

Graduate Diploma Thesis

Author: Michal Paděra
paderam@fel.cvut.cz

Advisor: Ing. Zdeněk Hurák
Center for Applied Cybernetics, CTU FEE, Prague, Czech Republic
z.hurak@c-a-k.cz

Reviewer: Dr. Didier Henrion
LAAS Toulouse, France
henrion@laas.fr

Submitted: January 23, 2004

Abstract

Object-oriented Java library for operating on polynomial matrices was created in this work. It provides developers and control engineers with a programming interface for implementing applications relying on manipulation with polynomial matrices. A polynomial matrix is a mathematical tool for description of both continuous and discrete dynamical systems. A new efficient object structure of classes storing various information about polynomial matrices was designed and it enables very efficient and reliable object oriented development of applications. Basic linear algebra operations with polynomial matrices were implemented. These algorithms are based on computation with constant matrices. Therefore existing Java library for computing with constant matrices was chosen. Syntax and semantics of implemented methods were documented and shown in simple examples of usage. Functionality of all methods was exceedingly tested and computational performance of more demanding operations was assessed by means of exhaustive numerical experiments. This initial version of Java library is fully functional and can be used in practice by applications that require operations on polynomial matrices.

Abstrakt

V této práci byla vytvořena objektová knihovna v jazyku Java pro práci s polynomiálními maticemi. Knihovna poskytuje vývojářům a návrhářům regulátorů programátorské rozhraní pro implementaci aplikací, které využívají polynomiální matice. Polynomiální matice je matematický nástroj sloužící k popisu dynamických systémů jak spojitých tak diskrétních. Byla navržena objektová struktura tříd uchovávajících informace o polynomiální matici. Byly implementovány základní algebraické operace nad polynomiálními maticemi. Tyto algoritmy jsou založeny na výpočtech s konstantními maticemi, proto byla použita existující javovská knihovna pro práci s konstantními maticemi. Je popsána syntaxe a sémantika implementovaných metod a jejich použití je ukázáno na jednoduchých příkladech. Správná funkčnost všech metod byla otestována. Dále byla změřena a vyhodnocena doba výpočtu časově náročnějších operací. Tato první verze javovské knihovny je plně funkční a může být použita v praxi v aplikacích, které vyžadují výpočty s polynomiálními maticemi.

Contents

Chapter 1

Introduction

Polynomial matrix is a mathematical tool for description of dynamical systems. A dynamical system can be described by set of differential or difference equations or by polynomial matrices. Properties of dynamical system can be found by solving differential equations or by studying algebraic properties of polynomial matrix. Furthermore, polynomial matrix can be used for advanced controller design (LQG, H_2, H_∞, etc.) or in signal processing applications (Wiener filters, Kalman filters, etc.) [18].

Only a few packages for computing with polynomial matrices exist. Namely, Polynomial Toolbox [19] for Matlab, a commercial package developed and distributed by PolyX company, Scilab [12], a library in a free Matlab clone developed by researchers at INRIA and finally, Maple [14], a commercial CAS system (Computer Algebra System), whose support of polynomial matrices is very simple. But things are changing now, reflecting a growing need for reliable tools for polynomial matrices on diverse platforms. There are also rumours that some polynomial package is being prepared for Mathematica, a commercial CAS system produced by Wolfram Research company. At the same time, a very mature Mathematica package for polynomial matrices has been developed by Jiri Kujan and the development still continues. An astonishingly efficient C++ package named PolPack++ [7] has been developed by Leos Halmo. A simple library for TI-89/TI-92 programmable calculators was coded by Petr Stefko. Last but not least, a purely symbolic package is being developed by Petr Augusta for commercial MuPAD package. All these new packages have been developed at the Department of Control Engineering at CTU FEE in Prague.

The aim of this work is to create object-oriented package for polynomial matrices similar to mentioned products. It should be operating system independent and therefore Java language is used for implementation. It should offer functional and fast enough application programming interface for basic linear algebra operations on polynomial matrices to enable develop software using polynomial matrices. Created package should be basis of graphical user interface of applications for design of robust controllers or optimal filters. These applications should run especially on the Internet and Java is the best solution for creating web applications. The portability of Java allows deploying applications both on Unix and Windows platforms without any changes in code. The advantage is that Java is distributed for free, it can be extended with existing libraries and it is independent on universal, expensive environments as Matlab or Mathematica.

1.1 Contribution of the Work

1. A rigorous analysis of suitability of Java for numerical computation.
2. Complete list of available software packages and their comparison.
3. Design of classes.
4. Implementation of the package.
5. Web page.

1.2 Features of the Library

I created package that is collection of classes that enable storing continuous and discrete-time (two-sided) polynomial matrices. These classes also enable performing basic linear algebra operations - addition, subtraction, multiplication, conjugation, transposition, norm, value, scaling, determinant, rank and roots of polynomial matrix. I based algorithms for computing determinant, rank and roots of polynomial matrix on evaluation of a polynomial matrix at a set of points equally distributed along the unit circle using FFT and on interpolation of a set of constant matrices by a polynomial matrix using inverse FFT. The package offers solvers of linear equations $\mathbf{A}(s)\mathbf{B}(s)=\mathbf{X}(s)$, $\mathbf{A}(z)\mathbf{B}(z)=\mathbf{X}(z)$ and $\mathbf{A}(s)\mathbf{X}(s)+\mathbf{B}(s)\mathbf{Y}(s)=\mathbf{C}(s)$. Furthermore, I created supportive classes for exporting polynomial matrices into MathML and other formats, functionality tests with graphical outputs, testing framework for performance tests. Executed performance tests generate outputs for further statistical processing and corresponding tests of Polynomial Toolbox for Matlab. The package is fully documented programming interface with number of examples for better understanding.

1.3 Overview of the Document

This document is divided into chapters describing proposal of library, implemented application programming interface (API) and tests. The *Analysis and Design* chapter starts with discussing issue of numerical computing in Java. Then the most suitable existing Java library for basic linear algebra computations with constant matrices and complex numbers is chosen. Basic classes and their structure are designed in the final part. The *Library Description* chapter describes implemented API. Syntax and semantics of each method are described. Examples of usage and basic idea of implemented algorithm are shown for more complex methods. Functionality and performance tests are discussed in the *Tests* chapter. Frameworks used for both types of tests are described and pieces of implemented tests are shown. Test results are summarized in the *Conclusion* chapter.

Source codes (i.e. method names, code examples. class names, etc.) are written in courier font (e.g. `toString()`). Comments of code are written in courier font in italic (e.g. *`this is a comment`*).

Attached CD

The CD attached to this document contains:

- This document in PDF format,
- Offline version of project's web pages, includes:
  - browsable examples,
  - outputs of performance tests,
  - builds to download,
  - documents related to project (proposal, poster, polynomial research group information),
  - browsable version of design diagrams,
  - links related to project,
  - contact information,
- Build of Java package for operating on polynomial matrices from January 2004,
- Complete source codes including examples,
- Java documentation of created programming interface,
- Useful downloads (Java Runtime Environment, MathPlayer, etc.).

For more information see *index.html* file in the root directory of CD.

Chapter 2

Analysis and Design

It is necessary to consider suitability of Java for numerical computing and find an existing Java library for basic linear algebra algorithms. Design of the important objects (especially classes keeping information about polynomial matrices) must be done. A few Java libraries that enable performing basic linear algebra algorithms on constant matrices exist. It is necessary to choose one of them, which fits best the needs of implementation polynomial matrices library. Its structure must be taken into account during design as well. All these mentioned issues are discussed in this chapter.

2.1 Java and Numerics

In this section general advantages and disadvantages of Java language are mentioned first, then issues concerned to Java and numerics are described more particularly and the best solutions reducing disadvantages are explained.

Java is the platform independent language (the same code works on Windows, Unix and the other platforms). It supports modern technologies as Internet applications, working with databases, multithreading, networking, etc. It has protected memory access (i.e. there are no pointers, only object references exist), automatic memory allocation and deallocation (garbage collector). Java is the object language easy to understand for developers. Java documentation is standard documentation that very effectively introduces to developers usage of API (application programming interface) [6].

Java and numerics issues are discussed at the Numerics Working Group of the Java Grande Forum [5]. The goals of the Numerics Working Group of the Java Grande Forum are to assess the suitability of Java for numerical computation, to work towards community consensus on actions which can be taken to overcome deficiencies of the language and its run-time environment, and to encourage the development of APIs for core mathematical operations. The Group hosts open meetings and other forums to discuss these issues. It is supported by the Mathematical and Computational Sciences Division of the NIST Information Technology Laboratory. The Numerics Working Group has contributors from many companies and universities involved in Java and numerics (Sun, IBM, MathWorks - professor Cleve Moler contributed, Visual Numerics, etc.).

Java language is not always suitable for implementation of numerical algorithms. Java as interpreted language is slower than compiled languages and executing of code is therefore slower. This disadvantage is partly removed by Just-In-Time compiler [10], included in the Java HotSpot Virtual Machine [22], that increases performance of executed code. Arrays and multidimensional arrays (need for operating on matrices) are in Java treated similarly as objects. There is no primitive data type for complex numbers. It is not possible to define overloaded operator.

2.1.1 Multidimensional Arrays

Arrays and multidimensional arrays (needed for operating on matrices) are treated similarly as objects in Java. It means that each array element is referenced and it is the reason why multidimensional arrays are allowed to have different sizes of arrays in each dimension - multidimensional array is array with references to arrays (e.g. in two-dimensional case each row of matrix might have different numbers of columns). Because of reference behaviour array elements might access the same memory area.

Once allocated arrays cannot be resized any more. Java automatically checks array bounds whenever an arrays is accessed.

It is proposed [5] to store matrices, i.e. multidimensional arrays of primitive data types (e.g. `double`, `int`), as classes. Constructors of such classes should check the rectangularity of an array. New array must be allocated when a matrix is resized.

2.1.2 Complex Numbers

There is no primitive data type for complex numbers. Primitive data types are kept in stack memory, which has fast access. Dynamically allocated objects are stored in heap having slower access [6]. Complex numbers must be defined as objects having defined operations on complex numbers. It is necessary to distinguish between semantics of operators. Operator `=` sets value of primitive data type or it sets reference to memory area with an object instance. Operator `==` compares values of primitive data types or it compares two references pointing to memory areas with object instances [5].

2.1.3 Operator Overloading And Lightweight Classes

It is more readable to use operators instead of methods substituting operator (e.g. `a + b` instead of `a.add(b)`) but Java does not enable overload operators (which would be useful e.g. for class storing complex numbers, see 2.1.2) [5].

It would be useful to create classes having some properties of primitive data types (e.g. operator semantics, see 2.1.2) which could have positive influence on performance (object would be stored in stack with faster access than heap). Properties of lightweight classes are described in [5]. Lightweight classes are not taken in account because it is non-standard Java usage.

2.2 Library Choice

Algorithms for polynomial matrices are based on operating on constant matrices. It is therefore a right strategy to base this polynomial package on some existing high quality library for constant matrices. There are both commercial and open source Java libraries for basic linear algebra algorithms. Their properties are listed in table 2.1[1].

	JNL	**JMSL**	**JMAT**	**NINJA**	**JAMA**	**COLT**
Open Source	no	no	yes	yes	yes	yes
Last Update	1997	2003	2003	1999	1999	2001
Java Grande Req.	no	no	yes	yes	yes	yes
Real Vector	yes	yes	yes	yes	yes	yes
Real Matrix	yes	yes	yes	yes	yes	yes
Complex Vector	yes	yes	no	yes	no	no
Complex Matrix	yes	yes	no	yes	no	no
Linear Algebra	yes	yes	yes	no	yes	yes

Table 2.1: Libraries for basic linear algorithms

JMSL 2.0 by Visual Numerics has been chosen for its complexity, availability, graphical interface, web application interface and support as the most suitable library for further design and implementation.

2.2.1 The JMSL Library and Visual Numerics

The JMSL Library [24] marketed by Visual Numerics is a complete collection of mathematical, statistical and charting classes that are used for developing network-centric, cost effective applications. It is written 100% in Java and easily fits into any Java application. It includes linear algebra, zero finding, splines, ordinary differential equations, linear programming, nonlinear optimization, FFTs, special functions, regression, ANOVA, ARMA, Kalman filters. It can be used in both standalone and Web environments. The library can be typically applied in these areas:

- Risk Management and Portfolio Optimization in Finance and Insurance,
- Manufacturing Yield Analysis, Process Control,
- R&D Analytical Tools for Data Analysis and Product Optimization,
- Energy Consumption Analysis,
- Customer and Market Visual Data Analysis,

[1] Information were gathered in February 2003

- Extending Analysis and Visualization Capabilities for ISVs
  - Business Intelligence
  - Databases
  - Supply Chain.

Created library for polynomial matrices uses JMSL 2.0 but new JMSL 2.5 was released in June 2003. New version includes new, more robust, non-linear optimization routines, curve fitting functions for graphical display, data mining and statistical algorithms and charting enhancements.

There is a few disadvantages or missing functionalities of JMSL 2.0 required for implementation and using library for polynomial matrices:

- JMSL is a commercial product,
- Class for complex matrix is missing some linear algebra methods,
- 2-D and 3-D FFT algorithm is missing.

Cooperation with Visual Numerics

The license was obtained from Visual Numerics for free. It is valid until this work is finished (spring 2004). People from Visual Numerics are interested in the package for polynomial matrices and would like negotiate further cooperation with Faculty of Electrical Engineering, CTU in Prague.

2.3 Design

The purpose of created library is to provide a developer with an efficient application programming interface (API) for operating on polynomial matrices. It should be primarily object-oriented library and its structure and algorithms should consider Java limits for numerical computing mentioned above (see 2.1). Algorithms for polynomial matrices should use methods for calculating with constant matrices provided by JMSL 2.0 from Visual Numerics (see [23], 2.2).

The API should enable to developer define object of polynomial matrix, its properties and execute mathematical operations using interface methods.

The most important issue of design is to decide how to store coefficients of polynomial matrix in array. Often used mathematical operations on polynomial matrices are effectively calculated when coefficients of polynomial matrix are stored as 3-dimensional arrays having degrees in the first dimension, rows in the second dimension and columns in the third dimension (i.e. polynomial matrix is a polynomial having constant matrices as coefficients). An example of coefficients storage is shown in figure A.7. Polynomial matrix object should keep information about degree of polynomial matrix, its number of rows and columns, etc. All of these information are needed later when performing operations on polynomial matrix.

It is useful to distinguish continuous-time polynomial matrices and discrete-time polynomial matrices. Discrete polynomial matrices are allowed to have polynomials with negative powers. Discrete-time polynomial matrices are also called two-sided polynomials or Laurent polynomial matrices. They are supported only by the Polynomial Toolbox for Matlab out of the existing few packages (see chapter 1). The advantage of having both positive and negative powers in one polynomial matrix stands out in optimal control and estimation problems. Most of algorithms for both continuous and discrete polynomial matrices have common basis and therefore it is reasonable to define abstract class (`PolynomialMatrix`). `PolynomialMatrix` class is ancestor of continuous and discrete polynomial matrix classes (`ContinuousPolynomial-Matrix`, `DiscretePolynomialMatrix`).

Designed principal classes, their structure and relations between classes are found in appendix A. Semantics of diagrams is explained in [1]. The following chapter describes implemented API and its usage.

Chapter 3

Library Description

In this chapter application programming interface (API) of implemented library is described. This chapter is divided into sections according to existing classes. They are ancestor of polynomial matrix objects (`PolynomialMatrix`), class for operating on continuous polynomial matrices (`ContinuousPolynomialMatrix`), class for operating on discrete polynomial matrices (`DiscretePolynomialMatrix`), class performing fast Fourier transform on polynomial matrices (`PolynomialMatrixFFT`), class solving linear equation with polynomial matrices (`ContinuousAXB, DiscreteAXB, AXBYC`) and class enabling exporting polynomial matrix into MathML format and its transforms (`MathMl`). Class usage is shortly explained and its location (package) is given at the beginning of each section. Each section is divided into subsections describing constants and API methods of class (public methods). The functionality of each method is shortly explained, the syntax is described (i.e. method's parameters, return values, thrown exceptions), usage of method is shown on a simple example and finally implementation of more complex algorithms is described. Special section describing exceptions used for handling errors is found at the beginning of this chapter. More detailed information about all implemented classes and their API can be found in Java documentation [16], see example on figure E.7.

3.1 Exceptions

All exceptions are stored in one package reserved for them. All of them are inherited from `java.lang.Exception` [22, 10, 6]. Only these exceptions can be thrown by library methods. It is said in each section describing method whether and which exceptions is thrown. In case of throwing exception by method, it must be wrapped in try-catch block [10, 6]. Each exception contains text describing reason for throwing exception.

Package

```
cz.ctu.fee.dce.polynomial.exceptions
```

All exceptions are listed in table 3.1.

Exception	Description
IllegalPMCoefficientsException	Exception indicates illegal polynomial matrix coefficients (e.g. coefficients, i.e. constant matrices, have different sizes for each degree or are not rectangular.)
PMAXBException	Exception indicates that linear equation **A**(s)**X**(s) = **B**(s) with polynomial matrices cannot be computed.
PMAXBYCException	Exception indicates that linear equation **A**(s)**X**(s) + **B**(s)**Y**(s) = **C**(s) with polynomial matrices cannot be computed.
PMDeterminantException	Exception indicates that determinant of polynomial matrix cannot be computed (e.g. when polynomial matrix has different number of rows and columns).
PMNormException	Exception indicates than norm of polynomial matrix cannot be computed.
PMRootsException	Exception indicates that roots of polynomial matrix cannot be computed.
PMsAddException	Exception indicates that polynomial matrices cannot be added or subtracted.
PMScaleException	Exception indicates that polynomial matrix cannot be scaled.
PMsMultiplyException	Exception indicates the polynomial matrices cannot be multiplied.

Table 3.1: List of exceptions

3.2 Class `PolynomialMatrix`

`PolynomialMatrix` class is the ancestor of `ContinuousPolynomialMatrix` and `DiscretePolynomialMatrix` classes. It is abstract class and therefore it cannot be instanced. It contains implemented functionalities common for both descendants. Some functionality is the same for both continuous and discrete polynomial matrix. It is the reason why some methods implemented in `PolynomialMatrix` class do not need to be overloaded in its descendants. Some of the overloaded methods might use functionality from this class. In this case subsections describing descendant class's (`ContinuousPolynomialMatrix`, `DiscretePolynomialMatrix`) methods will refer to subsections described in this section explaining inherited functionality.

Package

```
cz.ctu.fee.dce.polynomial
```

Constants

- `public static final char S_SYMBOL` - symbol "`s`" used for displaying polynomials
- `public static final int MULTIPLY_DEFAULT` - default multiplication of polynomial matrices (see 3.3.10)
- `public static final int MULTIPLY_DFT` - multiplication of polynomial matrices using discrete Fourier transform (see 3.3.10)
- `public static final int NORM_ABSOLUTE` - absolute norm (1-norm) of polynomial matrix (see 3.2.17)
- `public static final int NORM_QUADRATIC` - quadratic norm (2-norm) of polynomial matrix (see 3.2.17)
- `public static final int NORM_INFINITE` - infinite norm (∞-norm) of polynomial matrix (see 3.2.17)
- `public static final int NORM_FROBENIUS` - Frobenius norm of polynomial matrix (see 3.2.17)
- `public static final int NORM_METHOD_BLOCK` - block method used for computation of norm of polynomial matrix (see 3.2.17)
- `public static final int NORM_METHOD_LEAD` - leading coefficient method used for computation of norm of polynomial matrix (see 3.2.17)
- `public static final int NORM_METHOD_MAX` - maximal norm method used for computation of norm of polynomial matrix (see 3.2.17)

3.2.1 Constructor `(double[][][], char)`

Creates instance of polynomial matrix. It cannot be used for creating instance of `PolynomialMatrix` class because `PolynomialMatrix` is the abstract class. It used by constructors of descendants (`ContinuousPolynomialMatrix` and `DiscretePolynomialMatrix` classes).

Syntax

```
public PolynomialMatrix(double[][][] aCoef, char aSymbol)
throws IllegalPMCoefficientsException
```

Example

See 3.3.1 or 3.4.2.

Algorithm

Constructor `PolynomialMatrix(double[][][])` (see 3.2.2) is called for passed coefficients and passed symbol for displaying polynomials is set to private attribute holding its value.

3.2.2 Constructor `(double[][][])`

Creates instance of polynomial matrix. It cannot be used for creating instance of `PolynomialMatrix` class because `PolynomialMatrix` is the abstract class. It used by constructors of descendants (`ContinuousPolynomialMatrix` and `DiscretePolynomialMatrix` classes).

Syntax

```
public PolynomialMatrix(double[][][] aCoef)
throws IllegalPMCoefficientsException
```

Example

See 3.3.2 or 3.4.1.

Algorithm

First of all rectangularity of passed coefficients is checked. It means that each coefficient of polynomial must be matrix (i.e. each row must have the same number of columns) and all coefficients of polynomial must have the same sizes. If any of mentioned conditions is not satisfied, then the exception is thrown.

3.2.3 Constructor `(PolynomialMatrix)`

Creates instance of polynomial matrix. It cannot be used for creating instance of `PolynomialMatrix` class because `PolynomialMatrix` is the abstract class. It used by constructors of descendants (`ContinuousPolynomialMatrix` and `DiscretePolynomialMatrix` classes).

Syntax

```
public PolynomialMatrix(PolynomialMatrix aPm)
```

Example

See 3.3.3 or 3.4.5.

Algorithm

All encapsulated attributes (i.e. coefficients of polynomial matrix, its degree, its number of rows, its number of columns, symbol used for displaying and zeroing coefficient) are copied from passed polynomial matrix (passed argument). Attributes are just copied. Rectangularity and sizes of coefficients do not need to be checked because they have been already checked when instance of passed polynomial matrix has been created.

3.2.4 Method `equals()`

Compares polynomial matrix with another object.

Syntax

```
public boolean equals(Object aObject)
```

Algorithm

Object compared to polynomial matrix must be an instance of `PolynomialMatrix`, degrees and matrix sizes must equal, symbols used for displaying polynomials must be the same and coefficients must equal.

This method is used in JUnit tests for comparing expected and computed polynomial matrices (see chapter 4).

Equal objects must have equal hash code and therefore method `hashCode()` (see 3.2.13) must be implemented when method `equals` exists [22].

3.2.5 Method `getCoefficients()`

Returns coefficients of polynomial matrix.

Syntax

```
public final double[][][] getCoefficients()
```

3.2.6 Method `getColumnDegrees()`

Returns array containing degrees of columns. The i-th array element corresponds to degree of i-th column of polynomial matrix.

Syntax

```
public int[] getColumnDegrees()
```

Example

Figure 3.1 shows the example of getting column degrees of polynomial matrix. The polynomial matrix

$$\mathbf{A}(s) = \begin{pmatrix} 7+s & 2s & 9 \\ 4 & -s & 6 \end{pmatrix} \tag{3.1}$$

is created and its column degrees are

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

```
// coefficients of polynomial matrix A
double[][][] aCoef = {
  {{ 7, 0, 9}, { 4, 0, 6}}, // coefficients at s^0
  {{ 1, 2, 0}, { 0,-1, 0}}  // coefficients at s^1
};

// polynomial matrix A
ContinuousPolynomialMatrix cpmA =
  new ContinuousPolynomialMatrix(aCoef);

// gets column degrees of polynomial matrix A
int[] columnDegrees = cpmA.getColumnDegrees();
```

Figure 3.1: Example of `getColumnDegrees()` method usage

Algorithm

Non-zero coefficient belonging to highest degree in each column is found and the degree is stored in return value for given column.

3.2.7 Method `getNumberOfColumns()`

Returns number of columns of polynomial matrix.

Syntax

```
public final int getNumberOfColumns()
```

3.2.8 Method `getNumberOfRows()`

Returns number of rows of polynomial matrix.

Syntax

```
public final int getNumberOfRows()
```

3.2.9 Method `getRowDegrees()`

Returns array containing degrees of rows. The i-th array element corresponds to degree of i-th row of polynomial matrix.

Syntax

```
public int[] getRowDegrees()
```

Example

Figure 3.2 shows the example of getting row degrees of polynomial matrix. The polynomial matrix 3.1 is created and its column degrees are

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

Algorithm

Non-zero coefficient belonging to highest degree in each row is found and the degree is stored in return value for given row.

3.2.10 Method `getSylvesterMatrix(int)`

Creates Sylvester matrix [8] from coefficients of polynomial matrix. Coefficients are put in column.

Syntax

```
public double[][] getSylvesterMatrix(int aNumberOfColumns)
```

```
// coefficients of polynomial matrix A
double[][][] aCoef = {
  {{ 7, 0, 9}, { 4, 0, 6}}, // coefficients at s^0
  {{ 1, 2, 0}, { 0,-1, 0}}  // coefficients at s^1
};

// polynomial matrix A
ContinuousPolynomialMatrix cpmA =
  new ContinuousPolynomialMatrix(aCoef);

// gets row degrees of polynomial matrix A
int[] rowDegrees = cpmA.getRowDegrees();
```

Figure 3.2: Example of `getRowDegrees()` method usage

Example

Figure 3.3 shows creation of Sylvester matrix. The continuous polynomial matrix 3.1 is created and then Sylvester matrix

$$\begin{pmatrix} 7 & 0 & 9 & 0 & 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & 6 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 2 & 0 & 7 & 0 & 9 & 0 & 0 & 0 \\ 0 & -1 & 0 & 4 & 0 & 6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 2 & 0 & 7 & 0 & 9 \\ 0 & 0 & 0 & 0 & -1 & 0 & 4 & 0 & 6 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \end{pmatrix}$$

having three coefficients in a row is created.

Algorithm

There is polynomial matrix

$$\mathbf{A}(s) = \mathbf{A}_0 + \mathbf{A}_1 s + \ldots + \mathbf{A}_n s^n$$

At first the constant matrix

$$\mathbf{B} = \begin{pmatrix} \mathbf{A}_0 \\ \mathbf{A}_1 \\ \vdots \\ \mathbf{A}_n \end{pmatrix}$$

```
// coefficients of polynomial matrix A
double[][][] aCoef = {
  {{ 7, 0, 9}, { 4, 0, 6}}, // coefficients at s^0
  {{ 1, 2, 0}, { 0,-1, 0}}  // coefficients at s^1
};

// polynomial matrix A
ContinuousPolynomialMatrix cpmA = new
  ContinuousPolynomialMatrix(aCoef);

// creates Sylvester matrix from A
// having 3 coefficients in a row
double[][] sylvester = cpmA.getSylvesterMatrix(3);
```

Figure 3.3: Example of `getSylvesterMatrix()` method usage

is created from polynomial matrix's coefficients by putting them under each other. Sylvester matrix **S** is created by putting matrix **B** in diagonal of zero matrix having desired number of columns

$$\mathbf{S} = \begin{pmatrix} \mathbf{B} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \mathbf{B} & & \mathbf{0} \\ \vdots & & \ddots & \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{B} \end{pmatrix}.$$

3.2.11 Method `getSymbol()`

Returns symbol for displaying polynomials in polynomial matrix.

Syntax

```
public final char getSymbol()
```

3.2.12 Method `getZeroingCoefficient()`

Returns number of decimal places used for zeroing (evaluating numbers as zeros). It is number of decimal places following decimal point which must be equal to zero to evaluate double number as zero. Used for "inaccurate" operations (e.g. computation of determinant using FFT).

Syntax

```
public final int getZeroingCoefficient()
```

3.2.13 Method `hashCode()`

Returns hash code of polynomial matrix object. Equal objects must have equal hash code and therefore method `hashCode()` must be implemented when method `equals()` (see 3.2.13) exists [22].

Syntax

```
public int hashCode()
```

3.2.14 Method `isSquare()`

Returns true when polynomial matrix is square, otherwise returns false.

Syntax

```
public final boolean isSquare()
```

Algorithm

Number of rows and number columns must be the same when a polynomial matrix is square matrix.

3.2.15 Method `isZero()`

Checks whether polynomial matrix is zero polynomial matrix.

Syntax

```
public final boolean isZero()
```

Algorithm

Zero polynomial matrix must have degree 0 and its coefficient must be zero matrix.

3.2.16 Method `multiply(double)`

Multiplies polynomial matrix by real number.

Syntax

```
public final void multiply(double aNumber)
```

Example

Figure 3.4 shows the example of multiplication polynomial matrix by real number. The polynomial matrix 3.1 is created. After multiplication by *2* polynomial matrix

$$\mathbf{A}(s)=\begin{pmatrix} 14+2s & 4s & 18 \\ 8 & -2s & 12 \end{pmatrix}.$$

```
// coefficients of polynomial matrix A
double[][][] aCoef = {
  {{ 7, 0, 9}, { 4, 0, 6}}, // coefficients at s^0
  {{ 1, 2, 0}, { 0,-1, 0}}  // coefficients at s^1
};

// polynomial matrix A
ContinuousPolynomialMatrix cpmA =
  new ContinuousPolynomialMatrix(aCoef);

// multiplies polynomial matrix A by 2
cpmA.multiply(2);
```

Figure 3.4: Example of `multiply()` method usage

Algorithm

Each coefficient (i.e. constant matrix belonging to degree) is multiplied by given number.

3.2.17 Method `norm(int, int)`

Computes norm of polynomial matrix.

Syntax

```
public double norm(int aType, int aMethod)
throws IllegalArgumentException, PMNormException
```

Example

Figure 3.5 shows the example of computation norm of polynomial matrix. The polynomial matrix 3.1 is created and its absolute norm computed using "block" method is *15*.

```
// coefficients of polynomial matrix A
double[][][] aCoef = {
  {{ 7, 0, 9}, { 4, 0, 6}}, // coefficients at s^0
  {{ 1, 2, 0}, { 0,-1, 0}}  // coefficients at s^1
};

// polynomial matrix A
ContinuousPolynomialMatrix cpmA =
  new ContinuousPolynomialMatrix(aCoef);

// computes absolute norm of polynomial matrix A
// using "block" method
double norm = cpmA.norm(
  PolynomialMatrix.NORM_ABSOLUTE,
  PolynomialMatrix.NORM_METHOD_BLOCK
);
```

Figure 3.5: Example of `norm()` method usage

Algorithm

Norm of polynomial matrix is computed as norm of constant matrix [21, 23] created from coefficients of polynomial matrix. There are several ways for creating constant matrix from coefficients of polynomial matrix [17]:

- Block method (`PolynomialMatrix.NORM_METHOD_BLOCK`)
  There is polynomial matrix

  $$\mathbf{A}(s) = \mathbf{A}_0 + \mathbf{A}_1 s + \ldots + \mathbf{A}_n s^n.$$

  Constant matrix

  $$\mathbf{A} = \begin{pmatrix} \mathbf{A}_0 & \mathbf{A}_1 & \cdots & \mathbf{A}_n \end{pmatrix}$$

  is created from coefficients of polynomial matrix **A**(s).

- Leading coefficient method (`PolynomialMatrix.NORM_METHOD_LEAD`)
  Leading coefficient is coefficient of polynomial matrix by highest degree.

- Maximal norm method (`PolynomialMatrix.NORM_METHOD_MAX`)
  Norm for each coefficient of polynomial matrix is computed and norm of polynomial matrix is the maximum of these norms.

3.2.18 Method `rank()`

Computes rank of polynomial matrix. Uses interpolation algorithm [9].

Syntax

```
public int rank()
```

Example

Figure 3.6 shows the example of evaluating rank of polynomial matrix. The polynomial matrix 3.1 is created and its computed rank is *2*.

```
// coefficients of polynomial matrix A
double[][][] aCoef = {
  {{ 7, 0, 9}, { 4, 0, 6}}, // coefficients at s^0
  {{ 1, 2, 0}, { 0,-1, 0}}  // coefficients at s^1
};

// polynomial matrix A
ContinuousPolynomialMatrix cpmA =
  new ContinuousPolynomialMatrix(aCoef);

// computes rank of polynomial matrix A
int rank = cpmA.rank();
```

Figure 3.6: Example of `rank()` method usage

Algorithm

Polynomial matrix is evaluated in set of complex values. The appropriate method for evaluating polynomial matrix in set of complex values is direct discrete Fourier transform (see 3.5). Number of iterations and number of complex values evaluating polynomial matrix are given by estimated degree of determinant of polynomial matrix [9]. Rank of constant complex matrix is computed in each iteration of interpolation and maximal value of computed ranks is chosen. If

some of evaluations has maximal possible rank of polynomial matrix, rank of polynomial matrix was found and algorithm stops.

3.2.19 Method `roots()`

Computes roots of polynomial matrix. Roots of polynomial matrix are defined as roots of its determinant [17].

Syntax

```
public Complex[] roots() throws PMRootsException
```

Example

Figure 3.7 shows the example of computing roots of polynomial matrix. The polynomial matrix 3.1 is created and no roots were found.

```
// coefficients of polynomial matrix A
double[][][] aCoef = {
  {{ 7, 0, 9}, { 4, 0, 6}}, // coefficients at s^0
  {{ 1, 2, 0}, { 0,-1, 0}}  // coefficients at s^1
};

// polynomial matrix A
ContinuousPolynomialMatrix cpmA =
  new ContinuousPolynomialMatrix(aCoef);

// computes roots of polynomial matrix A
Complex[] roots = cpmA.roots();
```

Figure 3.7: Example of `roots()` method usage

Algorithm

Roots of determinant (i.e. roots of polynomial) are found in case of square polynomial matrix. In case of non-square or singular polynomial matrix two square polynomial matrices are created by multiplying original polynomial matrix by random constant matrices from left and right. Determinants for both ”squared” polynomial matrices are found. Then roots of first ”squared”

polynomial matrix are found. Each of these found roots is used as point for evaluation of determinant of second ”squared” matrix and it is root of original polynomial matrix when evaluation of determinant is zero (i.e. both ”squared” polynomial matrices have common roots).

3.2.20 Method `setSymbol(char)`

Sets symbol used for displaying polynomials in polynomial matrix.

Syntax

```
public final void setSymbol(char aSymbol)
```

3.2.21 Method `setZeroingCoefficient(int)`

Sets number of decimal places used for zeroing (evaluating numbers as zeros). It is number of places following decimal point which must be equal to zero to evaluate double number as zero. Used for "inaccurate" operations (e.g. computation of determinant using FFT).

Syntax

```
public final void setZeroingCoefficient(int aDigits)
throws IllegalArgumentException
```

3.2.22 Method `valueAt(Complex)`

Computes value of polynomial matrix at specified complex point. Horner scheme algorithm is used computation [25].

Syntax

```
public Complex[][] valueAt(Complex aPoint)
```

Example

Figure 3.8 shows the example of evaluating polynomial matrix in complex number. The polynomial matrix 3.1 is created and its value in complex point *j* is

$$\mathbf{A}(j)=\begin{pmatrix} 7+j & \mathbf{2}j & 9 \\ 4 & -j & 6 \end{pmatrix}.$$

3.2.23 Method `valueAt(double)`

Computes value of polynomial matrix at specified real point. Horner scheme is used computation [25].

```
// coefficients of polynomial matrix A
double[][][] aCoef = {
  {{ 7, 0, 9}, { 4, 0, 6}}, // coefficients at s^0
  {{ 1, 2, 0}, { 0,-1, 0}}  // coefficients at s^1
};

// polynomial matrix A
ContinuousPolynomialMatrix cpmA =
  new ContinuousPolynomialMatrix(aCoef);

// computes value of polynomial matrix A
// at complex point j
Complex[][] value = cpmA.valueAt(new Complex(0, 1));
```

Figure 3.8: Example of `valueAt()` method usage

Syntax

```
public double[][] valueAt(double aPoint)
```

Example

Figure 3.9 shows the example of evaluating polynomial matrix in real number. The polynomial matrix 3.1 is created and its value in real point *1* is

$$\mathbf{A}(1) = \begin{pmatrix} 8 & 2 & 9 \\ 4 & -1 & 6 \end{pmatrix}.$$

3.3 Class `ContinuousPolynomialMatrix`

This class enables operating on continuous polynomial matrices.

Package

```
cz.ctu.fee.dce.polynomial
```

3.3.1 Constructor `(double[][][], char)`

Creates instance of continuous polynomial matrix.

```
// coefficients of polynomial matrix A
double[][][] aCoef = {
  {{ 7, 0, 9}, { 4, 0, 6}}, // coefficients at s^0
  {{ 1, 2, 0}, { 0,-1, 0}}  // coefficients at s^1
};

// polynomial matrix A
ContinuousPolynomialMatrix cpmA =
  new ContinuousPolynomialMatrix(aCoef);

// computes value of polynomial matrix A at point 1
double[][] value = cpmA.valueAt(1);
```

Figure 3.9: Example of `valueAt()` method usage

Syntax

```
public ContinuousPolynomialMatrix(double[][][] aCoef,
char aSymbol) throws IllegalPMCoefficientsException
```

Example

Figure 3.10 shows how the instance of continuous polynomial matrix

$$\begin{pmatrix} 1-3x^2 & 8x-4x^2 & 3-5x^2 \\ -6x^2 & 2-7x^2 & -2x-8x^2 \end{pmatrix}$$

is created.

Algorithm

Constructor `PolynomialMatrix(double[][][], char)` from ancestor class is called (see 3.2.1).

3.3.2 Constructor `(double[][][])`

Creates instance of continuous polynomial matrix.

Syntax

```
public ContinuousPolynomialMatrix(double[][][] aCoef)
throws IllegalPMCoefficientsException
```

```
// coefficients of polynomial matrix A
double[][][] aCoef = {
  {{ 1, 0, 3}, { 0, 2, 0}}, // coefficients at x^0
  {{ 0, 8, 0}, { 0, 0,-2}}, // coefficients at x^1
  {{-3,-4,-5}, {-6,-7,-8}}  // coefficients at x^2
};

// polynomial matrix A is created
ContinuousPolynomialMatrix cpmA = new
  ContinuousPolynomialMatrix(aCoef, 'x');
```

Figure 3.10: Creation of continuous polynomial matrix instance

Example

Figure 3.11 shows how the instance of continuous polynomial matrix

$$\begin{pmatrix} 1-3s^2 & 8s-4s^2 & 3-5s^2 \\ -6s^2 & 2-7s^2 & -2s-8s^2 \end{pmatrix}$$

is created.

```
// coefficients of polynomial matrix A
double[][][] aCoef = {
  {{ 1, 0, 3}, { 0, 2, 0}}, // coefficients at s^0
  {{ 0, 8, 0}, { 0, 0,-2}}, // coefficients at s^1
  {{-3,-4,-5}, {-6,-7,-8}}  // coefficients at s^2
};

// polynomial matrix A is created
ContinuousPolynomialMatrix cpmA = new
  ContinuousPolynomialMatrix(aCoef);
```

Figure 3.11: Creation of continuous polynomial matrix instance

Algorithm

Constructor `PolynomialMatrix(double[][][])` from ancestor class is called (see 3.2.2).

3.3.3 Constructor (`ContinuousPolynomialMatrix`)

Creates instance of continuous polynomial matrix.

Syntax

```
public ContinuousPolynomialMatrix(
ContinuousPolynomialMatrix aCpm)
```

Example

Figure 3.12 shows how the instance of continuous polynomial matrix

$$\begin{pmatrix} 1-3s^2 & 8s-4s^2 & 3-5s^2 \\ -6s^2 & 2-7s^2 & -2s-8s^2 \end{pmatrix}$$

is created from the existing instance of continuous polynomial matrix.

```
// coefficients of polynomial matrix A
double[][][] aCoef = {
  {{ 1, 0, 3}, { 0, 2, 0}}, // coefficients at s^0
  {{ 0, 8, 0}, { 0, 0,-2}}, // coefficients at s^1
  {{-3,-4,-5}, {-6,-7,-8}}  // coefficients at s^2
};

// polynomial matrix A is created
ContinuousPolynomialMatrix cpmA = new
  ContinuousPolynomialMatrix(aCoef);

// polynomial matrix B is created from existing
// continuous polynomial matrix A
ContinuousPolynomialMatrix cpmB = new
  ContinuousPolynomialMatrix(cpmA);
```

Figure 3.12: Creation of continuous polynomial matrix instance

Algorithm

Constructor `PolynomialMatrix(PolynomialMatrix)` from ancestor class is called (see 3.2.3).

3.3.4 Method `add(ContinuousPolynomialMatrix)`

Adds continuous polynomial matrix.

Syntax

```
public void add(ContinuousPolynomialMatrix aCpm)
throws PMsAddException
```

Example

Figure 3.13 shows the example of polynomial matrices addition. Polynomial matrices

$$\mathbf{A}(s) = \begin{pmatrix} 1-3s^2 & 8s-4s^2 & 3-5s^2 \\ -6s^2 & 2-7s^2 & -2s-8s^2 \end{pmatrix} \tag{3.2}$$

and

$$\mathbf{B}(s) = \begin{pmatrix} 7+s & 2s & 9 \\ 4 & -1s & 6 \end{pmatrix}$$

are created. Polynomial matrix **B**(s) is added to polynomial matrix **A**(s). After addition

$$\mathbf{A}(s) = \begin{pmatrix} 8+s-3s^2 & 10s-4s^2 & 12-5s^2 \\ 4-6s^2 & 2-1s-7s^2 & 6-2s-8s^2 \end{pmatrix}$$

and **B**(s) remains unchanged.

Algorithm

Sizes of added polynomial matrices are checked at first. The exception is thrown in case of different sizes. Then corresponding coefficients (constant matrices) are added. Finally degree of result is lowered when coefficients at highest degrees are zero matrices.

3.3.5 Method `conjugate()`

Conjugates polynomial matrix [17].

Syntax

```
public void conjugate()
```

```
// coefficients of polynomial matrix A
double[][][] aCoef = {
  {{ 1, 0, 3}, { 0, 2, 0}}, // coefficients at s^0
  {{ 0, 8, 0}, { 0, 0,-2}}, // coefficients at s^1
  {{-3,-4,-5}, {-6,-7,-8}}  // coefficients at s^2
};

// coefficients of polynomial matrix B
double[][][] bCoef = {
  {{ 7, 0, 9}, { 4, 0, 6}}, // coefficients at s^0
  {{ 1, 2, 0}, { 0,-1, 0}}  // coefficients at s^1
};

// polynomial matrix A
ContinuousPolynomialMatrix cpmA =
  new ContinuousPolynomialMatrix(aCoef);

// polynomial matrix B
ContinuousPolynomialMatrix cpmB =
  new ContinuousPolynomialMatrix(bCoef);

// adds B to A
// A changes: A=A+B
// B is unchanged
cpmA.add(cpmB);
```

Figure 3.13: Example of `add()` method usage

Example

Figure 3.14 shows the example of polynomial matrix conjugation. The polynomial matrix 3.2 is created. After conjugation

$$\mathbf{A}(s) = \begin{pmatrix} 1-3s^2 & -8s-4s^2 & 3-5s^2 \\ -6s^2 & 2-7s^2 & 2s-8s^2 \end{pmatrix}.$$

Algorithm

Each coefficient at odd degree is multiplied by *-1*.

```
// coefficients of polynomial matrix A
double[][][] aCoef = {
  {{ 1, 0, 3}, { 0, 2, 0}}, // coefficients at s^0
  {{ 0, 8, 0}, { 0, 0,-2}}, // coefficients at s^1
  {{-3,-4,-5}, {-6,-7,-8}}  // coefficients at s^2
};

// polynomial matrix A
ContinuousPolynomialMatrix cpmA =
  new ContinuousPolynomialMatrix(aCoef);

// conjugates A
cpmA.conjugate();
```

Figure 3.14: Example of `conjugate()` method usage

3.3.6 Method `conjugateAndTranspose()`

Transposes and conjugates polynomial matrix [17].

Syntax

```
public void conjugateAndTranspose()
```

Example

Figure 3.15 shows the example of polynomial matrix conjugation and transposition. The polynomial matrix 3.2 is created. After conjugation and transposition

$$\mathbf{A}(s) = \begin{pmatrix} 1-3s^2 & -6s^2 \\ -8s-4s^2 & 2-7s^2 \\ 3-5s^2 & 2s-8s^2 \end{pmatrix}.$$

Algorithm

Polynomial matrix is transposed and then it is conjugated (see 3.3.5).

3.3.7 Method `determinant()`

Computes determinant of continuous polynomial matrix.

```
// coefficients of polynomial matrix A
double[][][] aCoef = {
  {{ 1, 0, 3}, { 0, 2, 0}}, // coefficients at s^0
  {{ 0, 8, 0}, { 0, 0,-2}}, // coefficients at s^1
  {{-3,-4,-5}, {-6,-7,-8}}  // coefficients at s^2
};

// polynomial matrix A
ContinuousPolynomialMatrix cpmA =
  new ContinuousPolynomialMatrix(aCoef);

// conjugates and transposes A
cpmA.conjugateAndTranspose();
```

Figure 3.15: Example of `conjugateAndTranspose()` method usage

Syntax

```
public ContinuousPolynomialMatrix determinant()
throws PMDeterminantException
```

Example

Figure 3.16 shows the example of computing determinant of polynomial matrix. The polynomial matrix

$$\mathbf{A}(s) = \begin{pmatrix} 1-3s^2 & 8s-4s^2 \\ -6s^2 & 2-7s^2 \end{pmatrix}$$

is created and its determinant

$$det\mathbf{A}(s) = \begin{pmatrix} 2-13s^2+48s^3-3s^4 \end{pmatrix}$$

is computed.

Algorithm

The FFT algorithm [11] is used. At first it is checked whether polynomial matrix is square matrix (if not exception is thrown). Then degree of determinant is estimated. Polynomial matrix is transformed using direct FFT algorithm (see 3.5), where number of points used for transformation equals to estimated degree. Direct FFT produces set of constant complex matrices.

```
// coefficients of polynomial matrix A
double[][][] aCoef = {
  {{ 1, 0}, { 0, 2}}, // coefficients at s^0
  {{ 0, 8}, { 0, 0}}, // coefficients at s^1
  {{-3,-4}, {-6,-7}}  // coefficients at s^2
};

// polynomial matrix A
ContinuousPolynomialMatrix cpmA =
  new ContinuousPolynomialMatrix(aCoef);

// computes determinant of A
ContinuousPolynomialMatrix det = cpmA.determinant();
```

Figure 3.16: Example of `determinant()` method usage

Determinant for each of this constant matrix is computed, i.e. set of constant complex numbers is produced. These numbers are transformed using inverse FFT (see 3.5). This transformation produces determinant (coefficients of determinant) of polynomial matrix.

3.3.8 Method `getDegree()`

Returns degree of continuous polynomial matrix.

Syntax

```
public int getDegree()
```

3.3.9 Method `multiply(ContinuousPolynomialMatrix)`

Multiplies polynomial matrix by polynomial matrix using default method (see 3.3.10)

Syntax

```
public void multiply(ContinuousPolynomialMatrix aCpm)
throws PMsMultiplyException
```

3.3.10 Method `multiply(ContinuousPolynomialMatrix, int)`

Multiplies polynomial matrix by polynomial matrix using given method for multiplication.

Syntax

```
public void multiply(ContinuousPolynomialMatrix aCpm,
int aMethod) throws PMsMultiplyException
```

Example

Figure 3.17 shows the example of polynomial matrices multiplication. Polynomial matrices 3.2 and

$$\mathbf{B}(s) = \begin{pmatrix} 7+s & 4 \\ 2s & -1s \\ 9 & 6 \end{pmatrix}$$

are created. Polynomial matrix **A**(s) is multiplied with polynomial matrix **B**(s) using DFT method. After multiplication

$$\mathbf{A}(s) = \begin{pmatrix} 34+s-50s^2-11s^3 & 22-50s^2+4s^3 \\ -14s-114s^2-20s^3 & -14s-72s^2+7s^3 \end{pmatrix}$$

and **B**(s) remains unchanged.

Algorithm

First of all algorithm for multiplication is chosen, then sizes of matrices for multiplication are checked (in case of incorrect sizes exception `PMsMultiplyException` is thrown). It is possible to choose one of these multiplication methods:

- default method (`PolynomialMatrix.MULTIPLY_DEFAULT`) - coefficients (constant matrices) of multiplied polynomial matrices are multiplied, added and set to corresponding degrees of result polynomial matrix. Degree of polynomial matrix is lowered in case of zero coefficients by highest degrees.
- DFT method [17, 11] (`PolynomialMatrix.MULTIPLY_DFT`) - multiplied polynomial matrices are transformed using direct FFT algorithm (see 3.5). Direct FFT produces set of constant complex matrices, where number of points used for transformation equals to estimated degree of result matrix. Corresponding matrices from each set are multiplied, i.e. set of complex multiplied matrices is produced. This set is transformed using inverse FFT (see 3.5). This transformation produces result of multiplication. Result is zeroed (see 3.2.21).

3.3.11 Method `scale()`

Scales polynomial matrix with automatically set scaling coefficient.

```
// coefficients of polynomial matrix A
double[][][] aCoef = {
  {{ 1, 0, 3}, { 0, 2, 0}}, // coefficients at s^0
  {{ 0, 8, 0}, { 0, 0,-2}}, // coefficients at s^1
  {{-3,-4,-5}, {-6,-7,-8}}  // coefficients at s^2
};

// coefficients of polynomial matrix B
double[][][] bCoef = {
  {{ 7, 4}, { 0, 0}, { 9, 6}}, // coefficients at s^0
  {{ 1, 0}, { 2,-1}, { 0, 0}}  // coefficients at s^1
};

// polynomial matrix A
ContinuousPolynomialMatrix cpmA =
  new ContinuousPolynomialMatrix(aCoef);

// polynomial matrix B
ContinuousPolynomialMatrix cpmB =
  new ContinuousPolynomialMatrix(bCoef);

// multiplies A with B using DFT method
// A changes: A=A*B
// B is unchanged
cpmA.multiply(cpmB, PolynomialMatrix.MULTIPLY_DFT);
```

Figure 3.17: Example of `multiply()` method usage

Syntax

```
public ContinuousPolynomialMatrix scale()
throws PMScaleException
```

Example

Figure 3.18 shows the example of scaling polynomial matrix. The polynomial matrix 3.2 is created. Method `scale()` produces scaled polynomial matrix

$$\begin{pmatrix} 4-3s^2 & 17s-4s^2 & 13-5s^2 \\ -6s^2 & 9-7s^2 & -4s-8s^2 \end{pmatrix}.$$

```
// coefficients of polynomial matrix A
double[][][] aCoef = {
  {{ 1, 0, 3}, { 0, 2, 0}}, // coefficients at s^0
  {{ 0, 8, 0}, { 0, 0,-2}}, // coefficients at s^1
  {{-3,-4,-5}, {-6,-7,-8}}  // coefficients at s^2
};

// polynomial matrix A
ContinuousPolynomialMatrix cpmA =
  new ContinuousPolynomialMatrix(aCoef);

// scales matrix A with scaling coefficient
// set automatically
ContinuousPolynomialMatrix scaledCpm = cpmA.scale();
```

Figure 3.18: Example of `scale()` method usage

Algorithm

Scaling coefficient [17] is set as

$$\sqrt[d]{\frac{norm\mathbf{A}_n}{norm\mathbf{A}_x}},$$

where

$$\mathbf{A}(s) = \mathbf{A}_0 + \mathbf{A}_1 s + \ldots + \mathbf{A}_n s^n, \tag{3.3}$$

$\mathbf{A_x}$ is first non-zero coefficient of polynomial matrix $\mathbf{A}(s)$ by lowest degree and $d = n - x$. Then method `scale(double)` (see 3.3.12) with passed scaling coefficient is called.

3.3.12 Method `scale(double)`

Scales polynomial matrix with given scaling coefficient.

Syntax

```
public ContinuousPolynomialMatrix scale(double aScaling)
throws PMScaleException
```

Example

Figure 3.19 shows the example of scaling polynomial matrix. The polynomial matrix 3.2 is created. Created polynomial matrix is scaled with scaling coefficient *0.1* and scaled polynomial matrix

$$\begin{pmatrix} 0.01 - 3s^2 & 0.8s - 4s^2 & 0.03 - 5s^2 \\ -6s^2 & 0.02 - 7s^2 & -0.2 - 8s^2 \end{pmatrix}$$

is produced.

```
// coefficients of polynomial matrix A
double[][][] aCoef = {
  {{ 1, 0, 3}, { 0, 2, 0}}, // coefficients at s^0
  {{ 0, 8, 0}, { 0, 0,-2}}, // coefficients at s^1
  {{-3,-4,-5}, {-6,-7,-8}}  // coefficients at s^2
};

// polynomial matrix A
ContinuousPolynomialMatrix cpmA =
  new ContinuousPolynomialMatrix(aCoef);

// scales matrix A with scaling coefficient 0.1
ContinuousPolynomialMatrix scaledCpm =
  cpmA.scale(0.1);
```

Figure 3.19: Example of `scale()` method usage

Algorithm

Polynomial matrix 3.3 scaled with coefficient *a* is polynomial matrix

$$a^n \mathbf{A}_0 + a^{n-1} \mathbf{A}_1 \frac{s}{a} + \ldots + \mathbf{A}_n \left(\frac{s}{a}\right)^n,$$

i.e. each coefficient of polynomial matrix is multiplied by given scaling [17].

3.3.13 Method `subtract(ContinuousPolynomialMatrix)`

Subtracts continuous polynomial matrix.

Syntax

```
public void subtract(ContinuousPolynomialMatrix aCpm)
throws PMsAddException
```

Example

Figure 3.20 shows the example of polynomial matrices subtraction. Polynomial matrices 3.2 and

$$\mathbf{B}(s) = \begin{pmatrix} 7+s & 2s & 9 \\ 4 & -1s & 6 \end{pmatrix}$$

are created. Polynomial matrix **B**(s) is subtracted from polynomial matrix **A**(s). After subtraction

$$\mathbf{A}(s) = \begin{pmatrix} -6-1s-3s^2 & 6s-4s^2 & -6-5s^2 \\ -4-6s^2 & 2+s-7s^2 & -6-2s-8s^2 \end{pmatrix}$$

and **B**(s) remains unchanged.

Algorithm

Subtracted polynomial matrix is multiplied by *-1* (see 3.2.16) and then it is added (see 3.3.4).

3.3.14 Method `toString()`

Converts continuous polynomial matrix to `String`. Method should be used for debugging purposes only.

Syntax

```
public String toString()
```

3.3.15 Method `transpose()`

Transposes continuous polynomial matrix.

Syntax

```
public void transpose()
```

```
// coefficients of polynomial matrix A
double[][][] aCoef = {
  {{ 1, 0, 3}, { 0, 2, 0}}, // coefficients at s^0
  {{ 0, 8, 0}, { 0, 0,-2}}, // coefficients at s^1
  {{-3,-4,-5}, {-6,-7,-8}}  // coefficients at s^2
};

// coefficients of polynomial matrix B
double[][][] bCoef = {
  {{ 7, 0, 9}, { 4, 0, 6}}, // coefficients at s^0
  {{ 1, 2, 0}, { 0,-1, 0}}  // coefficients at s^1
};

// polynomial matrix A
ContinuousPolynomialMatrix cpmA =
  new ContinuousPolynomialMatrix(aCoef);

// polynomial matrix B
ContinuousPolynomialMatrix cpmB =
  new ContinuousPolynomialMatrix(bCoef);

// subtracts B from A
// A changes: A=A-B
// B is unchanged
cpmA.subtract(cpmB);
```

Figure 3.20: Example of `subtract()` method usage

Example

Figure 3.21 shows the example of polynomial matrix transposition. The polynomial matrix 3.2 is created. After transposition

$$\mathbf{A}(s) = \begin{pmatrix} 1-3s^2 & -6s^2 \\ 8s-4s^2 & 2-7s^2 \\ 3-5s^2 & -2s-8s^2 \end{pmatrix}.$$

Algorithm

Each coefficient of polynomial matrix is transposed.

```
// coefficients of polynomial matrix A
double[][][] aCoef = {
  {{ 1, 0, 3}, { 0, 2, 0}}, // coefficients at s^0
  {{ 0, 8, 0}, { 0, 0,-2}}, // coefficients at s^1
  {{-3,-4,-5}, {-6,-7,-8}}  // coefficients at s^2
};

// polynomial matrix A
ContinuousPolynomialMatrix cpmA =
  new ContinuousPolynomialMatrix(aCoef);

// transposes A
cpmA.transpose();
```

Figure 3.21: Example of `transpose()` method usage

3.4 Class `DiscretePolynomialMatrix`

This class enables operating on discrete or two-sided polynomial matrices.

Package

`cz.ctu.fee.dce.polynomial`

Constants

- `public static final char Z_SYMBOL` - symbol “z” used for displaying polynomials

3.4.1 Constructor `(double[][][])`

Creates instance of discrete polynomial matrix.

Syntax

```
public DiscretePolynomialMatrix(double[][][] aCoef)
throws IllegalPMCoefficientsException
```

Example

Figure 3.22 shows how the instance of discrete polynomial matrix

$$\begin{pmatrix} 1-3z^2 & 8z-4z^2 & 3-5z^2 \\ -6z^2 & 2-7z^2 & -2z-8z^2 \end{pmatrix}$$

is created.

```
// coefficients of polynomial matrix A
double[][][] aCoef = {
  {{ 1, 0, 3}, { 0, 2, 0}}, // coefficients at z^0
  {{ 0, 8, 0}, { 0, 0,-2}}, // coefficients at z^1
  {{-3,-4,-5}, {-6,-7,-8}}  // coefficients at z^2
};

// polynomial matrix A is created
DiscretePolynomialMatrix dpmA = new
  DiscretePolynomialMatrix(aCoef);
```

Figure 3.22: Creation of discrete polynomial matrix instance

Algorithm

Constructor `PolynomialMatrix(double[][][], char)` from ancestor class with attribute `aSymbol` set to value of constant `Z_SYMBOL` (see 3.4) is called (see 3.2.1) and the attribute holding value of lowest power of polynomial is set to zero.

3.4.2 Constructor `(double[][][], char)`

Creates instance of discrete polynomial matrix.

Syntax

```
public DiscretePolynomialMatrix(double[][][] aCoef,
char aSymbol) throws IllegalPMCoefficientsException
```

Example

Figure 3.23 shows how the instance of discrete polynomial matrix

$$\begin{pmatrix} 1-3x^2 & 8x-4x^2 & 3-5x^2 \\ -6x^2 & 2-7x^2 & -2x-8x^2 \end{pmatrix}$$

is created.

```
// coefficients of polynomial matrix A
double[][][] aCoef = {
  {{ 1, 0, 3}, { 0, 2, 0}}, // coefficients at x^0
  {{ 0, 8, 0}, { 0, 0,-2}}, // coefficients at x^1
  {{-3,-4,-5}, {-6,-7,-8}}  // coefficients at x^2
};

// polynomial matrix A is created
DiscretePolynomialMatrix dpmA = new
  DiscretePolynomialMatrix(aCoef, 'x');
```

Figure 3.23: Creation of discrete polynomial matrix instance

Algorithm

Constructor `PolynomialMatrix(double[][][], char)` from ancestor class is called (see 3.2.1) and the attribute holding value of lowest power of polynomial is set to zero.

3.4.3 Constructor `(double[][][], int)`

Creates instance of discrete polynomial matrix.

Syntax

```
public DiscretePolynomialMatrix(double[][][] aCoef,
int aLowestPower) throws IllegalPMCoefficientsException
```

Example

Figure 3.24 shows how the instance of discrete polynomial matrix

$$\begin{pmatrix} z^{-1}-3z & 8-4z & 3z^{-1}-5z \\ -6z & 2z^{-1}-7z & -2-8z \end{pmatrix}$$

is created.

```
// coefficients of polynomial matrix A
double[][][] aCoef = {
  {{ 1, 0, 3}, { 0, 2, 0}}, // coefficients at z^-1
  {{ 0, 8, 0}, { 0, 0,-2}}, // coefficients at z^0
  {{-3,-4,-5}, {-6,-7,-8}}  // coefficients at z^1
};

// polynomial matrix A is created
DiscretePolynomialMatrix dpmA = new
  DiscretePolynomialMatrix(aCoef, 1);
```

Figure 3.24: Creation of discrete polynomial matrix instance

Algorithm

Constructor `DiscretePolynomialMatrix(double[][][], int, char)` with attribute `aSymbol` set to value of constant `Z_SYMBOL` (see 3.4) is called (see 3.4.4).

3.4.4 Constructor `(double[][][], int, char)`

Creates instance of discrete polynomial matrix.

Syntax

```
public DiscretePolynomialMatrix(double[][][] aCoef,
int aLowestPower, char aSymbol)
throws IllegalPMCoefficientsException
```

Example

Figure 3.25 shows how the instance of discrete polynomial matrix

$$\begin{pmatrix} x^{-4} - 3x^{-2} & 8x^{-3} - 4x^{-2} & 3x^{-4} - 5x^{-2} \\ -6x^{-2} & 2x^{-4} - 7x^{-2} & -2x^{-3} - 8x^{-2} \end{pmatrix}$$

is created.

```
// coefficients of polynomial matrix A
double[][][] aCoef = {
  {{ 1, 0, 3}, { 0, 2, 0}}, // coefficients at x^-4
  {{ 0, 8, 0}, { 0, 0,-2}}, // coefficients at x^-3
  {{-3,-4,-5}, {-6,-7,-8}}  // coefficients at x^-2
};

// polynomial matrix A is created
DiscretePolynomialMatrix dpmA = new
  DiscretePolynomialMatrix(aCoef, 4, 'x');
```

Figure 3.25: Creation of discrete polynomial matrix instance

Algorithm

Constructor `PolynomialMatrix(double[][][], char)` from ancestor class is called (see 3.2.1). Positive value of parameter `aLowestPower` is checked and the attribute holding value of lowest power of polynomial is set to value of parameter `aLowestPower`. In case or negative or zero value of parameter `aLowestPower` the `IllegalPMCoefficients-Exception` informing about wrong value of parameter is thrown.

3.4.5 Constructor (`DiscretePolynomialMatrix`)

Creates instance of discrete polynomial matrix.

Syntax

```
public DiscretePolynomialMatrix(
DiscretePolynomialMatrix aDpm)
```

Example

Figure 3.26 shows how the instance of discrete polynomial matrix

$$\begin{pmatrix} z^{-2}-3 & 8z^{-1}-4 & 3z^{-2}-5 \\ -6 & 2z^{-2}-7 & -2z^{-1}-8 \end{pmatrix}$$

is created from the existing instance of discrete polynomial matrix.

```
// coefficients of polynomial matrix A
double[][][] aCoef = {
  {{ 1, 0, 3}, { 0, 2, 0}}, // coefficients at z^-2
  {{ 0, 8, 0}, { 0, 0,-2}}, // coefficients at z^-1
  {{-3,-4,-5}, {-6,-7,-8}}  // coefficients at z^0
};

// polynomial matrix A is created
DiscretePolynomialMatrix dpmA = new
  DiscretePolynomialMatrix(aCoef, 2);

// polynomial matrix B is created from existing
// continuous polynomial matrix A
DiscretePolynomialMatrix dpmB = new
  DiscretePolynomialMatrix(dpmA);
```

Figure 3.26: Creation of discrete polynomial matrix instance

Algorithm

Constructor `PolynomialMatrix(PolynomialMatrix)` from ancestor class is called (see 3.2.3) and the attribute holding value of the lowest power of polynomial is the lowest power of passed discrete polynomial matrix instance.

3.4.6 Method `add(DiscretePolynomialMatrix)`

Adds discrete polynomial matrix.

Syntax

```
public void add(DiscretePolynomialMatrix aDpm)
throws PMsAddException
```

Example

Figure 3.27 shows the example of polynomial matrices addition. Polynomial matrices

$$\mathbf{A}(s) = \begin{pmatrix} 2+z & 1 \\ z^{-1}+1 & z \end{pmatrix} \tag{3.4}$$

and

$$\mathbf{B}(s) = \begin{pmatrix} 7z^{-1}+1 & 9z^{-1} \\ 4z^{-1} & 6z^{-1} \end{pmatrix}$$

are created. Polynomial matrix **B**(s) is added to polynomial matrix **A**(s). After addition

$$\mathbf{A}(s) = \begin{pmatrix} 7z^{-1}+3+z & 9z^{-1}+1 \\ 5z^{-1}+1 & 6z^{-1}+z \end{pmatrix}$$

and **B**(s) remains unchanged.

Algorithm

See algorithm in 3.3.4.

3.4.7 Method `conjugate()`

Conjugates discrete polynomial matrix.

Syntax

```
public void conjugate()
```

Example and Algorithm

See example and algorithm in 3.3.5.

3.4.8 Method `conjugateAndTranspose()`

Transposes and conjugates discrete polynomial matrix.

Syntax

```
public void conjugateAndTranspose()
```

Example and Algorithm

See example and algorithm in 3.3.6.

```
// coefficients of polynomial matrix A
double[][][] aCoef = {
  {{ 0, 0}, { 1, 0}}, // coefficients at z^-1
  {{ 2, 1}, { 1, 0}}, // coefficients at z^0
  {{ 1, 0}, { 0, 1}}  // coefficients at z^1
};

// coefficients of polynomial matrix B
double[][][] bCoef = {
  {{ 7, 9}, { 4, 6}}, // coefficients at z^-1
  {{ 1, 0}, { 0, 0}}  // coefficients at z^0
};

// polynomial matrix A
DiscretePolynomialMatrix dpmA =
  new DiscretePolynomialMatrix(aCoef, 1);

// polynomial matrix B
DiscretePolynomialMatrix dpmB =
  new DiscretePolynomialMatrix(bCoef, 1);

// adds B to A
// A changes: A=A+B
// B is unchanged
dpmA.add(dpmB);
```

Figure 3.27: Example of `add()` method usage

3.4.9 Method `determinant()`

Computes determinant of discrete polynomial matrix.

Syntax

```
public DiscretePolynomialMatrix determinant()
throws PMDeterminantException
```

Example and Algorithm

See example and algorithm in 3.3.7.

3.4.10 Method `equals()`

Compares discrete polynomial matrix with another object.

Syntax

```
public boolean equals(Object object)
```

Algorithm

See algorithm in 3.2.4

3.4.11 Method `getDegree()`

Returns degree of discrete polynomial matrix part with positive powers.

Syntax

```
public int getDegree()
```

3.4.12 Method `getLowestPower()`

Returns degree of discrete polynomial matrix part with negative powers.

Syntax

```
public int getLowestPower()
```

3.4.13 Method `isTwoSided()`

Returns true if polynomial matrix is two-sided (it has both positive and negative powers), otherwise returns false.

Syntax

```
public boolean isTwoSided()
```

3.4.14 Method `multiply(DiscretePolynomialMatrix)`

Multiplies discrete polynomial matrix by discrete polynomial matrix using default method (see 3.4.15).

Syntax

```
public void multiply(DiscretePolynomialMatrix aDpm)
throws PMsMultiplyException
```

Example and Algorithm

See example and algorithm in 3.3.10.

3.4.15 Method `multiply(DiscretePolynomialMatrix, int)`

Multiplies discrete polynomial matrix by discrete polynomial matrix choosing method for multiplication.

Syntax

```
public void multiply(DiscretePolynomialMatrix aDpm,
int method) throws PMsMultiplyException
```

Example and Algorithm

See example and algorithm in 3.3.10.

3.4.16 Method `scale()`

Scales polynomial matrix with scaling coefficient set automatically.

Syntax

```
public DiscretePolynomialMatrix scale()
throws PMScaleException
```

Example and Algorithm

See example and algorithm in 3.3.11.

3.4.17 Method `scale(double)`

Scales polynomial matrix with given scaling coefficient.

Syntax

```
public DiscretePolynomialMatrix scale(double scaling)
throws PMScaleException
```

Example and Algorithm

See example and algorithm in 3.3.12.

3.4.18 Method `subtract()`

Subtracts discrete polynomial matrix.

Syntax

```
public void subtract(DiscretePolynomialMatrix aDpm)
throws PMsAddException
```

Example and Algorithm

See example and algorithm in 3.3.13.

3.4.19 Method `toString()`

Converts polynomial matrix to `String`. Method should be used for debugging purposes only.

Syntax

```
public String toString()
```

3.4.20 Method `transpose()`

Transposes discrete polynomial matrix.

Syntax

```
public void transpose()
```

Example and Algorithm

See example and algorithm in 3.3.15.

3.4.21 Method `valueAt(Complex)`

Computes value of polynomial matrix at specified complex point. Horner scheme [25] is used computation.

Syntax

```
public Complex[][] valueAt(Complex point)
```

Example

See example in 3.2.22.

3.4.22 Method `valueAt(double)`

Computes value of polynomial matrix at specified complex point. Horner scheme [25] is used computation.

Syntax

```
public double[][] valueAt(double point)
```

Example

See example in 3.2.23.

3.5 Class `PolynomialMatrixFFT`

This class enables performing direct and inverse fast Fourier transform (FFT) on polynomial matrices.

Package

```
cz.ctu.fee.dce.polynomial
```

3.5.1 Method `directFFT(ContinuousPolynomialMatrix, int)`

Evaluates real continuous polynomial matrix at given number of points equally distributed along the unit circle in complex plane. Direct fast Fourier transform (FFT) is used for evaluation.

Syntax

```
public static Complex[][][] directFFT(
ContinuousPolynomialMatrix aCpm, int aSamples)
```

Example

Figure 3.28 shows the example of direct and inverse FFT on continuous polynomial matrix. The polynomial matrix

$$\mathbf{A}(s) = \begin{pmatrix} 1-3s^2 & 8s-4s^2 & 3-5s^2 \\ -6s^2 & 2-7s^2 & -2s-8s^2 \end{pmatrix}$$

is created. Direct FFT is applied on polynomial matrix **A**(s) at three points. It produces set of constant complex matrices

$$\left\{ \begin{pmatrix} -2 & 4 & -2 \\ -6 & -5 & -10 \end{pmatrix}, \begin{pmatrix} 2.5 \pm j2.6 & -2 \pm j10.39 & -5.5 \pm j4.33 \\ 3 \pm j5.2 & 5.5 \pm j6.06 & 5 \pm j5.2 \end{pmatrix} \right\}.$$

This set is transformed back using inverse FFT and polynomial matrix **A**(s) is reconstructed.

```
// coefficients of polynomial matrix A
double[][][] aCoef = {
  {{ 1, 0, 3}, { 0, 2, 0}}, // coefficients at s^0
  {{ 0, 8, 0}, { 0, 0,-2}}, // coefficients at s^1
  {{-3,-4,-5}, {-6,-7,-8}}  // coefficients at s^2
};

// polynomial matrix A
ContinuousPolynomialMatrix cpmA =
  new ContinuousPolynomialMatrix(aCoef);

// direct FFT transform on polynomial
// matrix A in 3 samples
Complex[][][] cpmATransformed =
  PolynomialMatrixFFT.directFFT(cpmA, 3);

// inverse FFT transform producing continuous
// polynomial matrix of degree 3
ContinuousPolynomialMatrix cpmB =
 PolynomialMatrixFFT.inverseFFTContinuous(
  cpmATransformed, 3);
```

Figure 3.28: Example of direct and inverse FFT on continuous polynomial matrix

Algorithm

Discrete Fourier transform of polynomial matrix is described in [11]. Set of polynomials

$$p_{ij} = [p_{0,ij}, p_{1,ij}, \ldots p_{r,ij}], i \in < 1; x >, j \in < 1; y > \tag{3.5}$$

from polynomial matrix

$$\mathbf{P}(s) = \mathbf{P}_0 + \mathbf{P}_1 s + \ldots + \mathbf{P}_r s^r \tag{3.6}$$

of degree r with x rows and y columns, where

$$\mathbf{P}_k = \begin{pmatrix} p_{k,11} & p_{k,12} & \cdots & p_{k,1y} \\ p_{k,21} & p_{k,22} & \cdots & p_{k,2y} \\ \vdots & \vdots & \ddots & \vdots \\ p_{k,x1} & p_{k,x2} & \cdots & p_{k,xy} \end{pmatrix}, k \in < 0; r >, \tag{3.7}$$

is created. Each polynomial p_{ij} is transformed in s samples using FFT algorithm [23], i.e. sets of complex numbers q_{ij} are computed. Finally set of constant complex matrices

$$\mathbf{Q}_l = \begin{pmatrix} q_{l,11} & q_{l,12} & \cdots & q_{l,1y} \\ q_{l,21} & q_{l,22} & \cdots & q_{l,2y} \\ \vdots & \vdots & \ddots & \vdots \\ q_{l,x1} & q_{l,x2} & \cdots & q_{l,xy} \end{pmatrix}, l \in < 0; s > \tag{3.8}$$

is reconstructed from q_{ij} corresponding to discrete Fourier transform of polynomial matrix **P**(s).

3.5.2 Method `directFFT(DiscretePolynomialMatrix, int)`

Evaluates real discrete polynomial matrix at given number of points equally distributed along the unit circle in complex plane. Direct fast Fourier transform (FFT) is used for evaluation.

Syntax

```
public static Complex[][][] directFFT(
DiscretePolynomialMatrix aDpm, int aSamples)
```

Example

See example in 3.28.

Algorithm

See algorithm in 3.5.1.

3.5.3 Method `inverseFFTContinuous(Complex[][][], int)`

Interpolates real continuous polynomial matrix from set of constant complex matrices. These constant complex matrices correspond to real polynomial matrix evaluated at points equally distributed along the unit circle. Inverse fast Fourier transform (FFT) is used for interpolation.

Syntax

```
public static ContinuousPolynomialMatrix
inverseFFTContinuous(Complex[][][] aTransformedCPM,
int aDegree)
```

Example

See example in 3.28.

Algorithm

Inverse discrete Fourier transform of polynomial matrix is described in [11]. Sets of complex numbers

$$q_{ij} = [q_{0,ij}, q_{1,ij}, \dots q_{s,ij}], i \in < 1; x >, j \in < 1; y >$$

are created from transformed polynomial matrix 3.8. Each set q_{ij} is transformed in r samples using inverse FFT algorithm [23], i.e. sets of numbers p_{ij} are computed. Finally coefficients 3.7 of polynomial matrix 3.6 are reconstructed from set of polynomials 3.5 corresponding to inverse discrete Fourier transform of 3.8.

3.5.4 Method `inverseFFTDiscrete(Complex[][][], int, int)`

Interpolates real discrete polynomial matrix from set of constant complex matrices. These constant complex matrices correspond to real polynomial matrix evaluated at points equally distributed along the unit circle. Inverse fast Fourier transform (FFT) is used for interpolation.

Syntax

```
public static DiscretePolynomialMatrix inverseFFTDiscrete(
Complex[][][] aTransformedDPM, int aDegree, int aLowestPower)
```

Example

See example in 3.28.

Algorithm

See algorithm in 3.5.3.

3.6 Class `ContinuousAXB`

This class enables solving of linear equation **A**(s)**X**(s) = **B**(s) with continuous polynomial matrices.

Package

```
cz.ctu.fee.dce.polynomial
```

3.6.1 Constructor

Solves linear equation **A**(s)**X**(s) = **B**(s) with polynomial matrices.

Syntax

```
public ContinuousAXB(ContinuousPolynomialMatrix aA,
ContinuousPolynomialMatrix aB) throws PMAXBException
```

Example

Figure 3.29 shows how linear equation

$$\begin{pmatrix} 1 & 0 \\ s & 1 \end{pmatrix} \mathbf{X}(s) = \begin{pmatrix} s \\ 1 \end{pmatrix}$$

is solved. The solution

$$\mathbf{X}(s) = \begin{pmatrix} s \\ 1 - 1s^2 \end{pmatrix}$$

is found.

```
// coefficients of polynomial matrix A
double[][][] aCoef = {
  {{1, 0}, {0, 1}}, // coefficient at s^0
  {{0, 0}, {1, 0}}  // coefficient at s^1
};
// coefficients of polynomial matrix B
double[][][] bCoef = {
  {{1, 0}, {0, 1}}, // coefficient at s^0
  {{0, 0}, {1, 0}}  // coefficient at s^1
};
// polynomial matrix A
ContinuousPolynomialMatrix cpmA = new
  ContinuousPolynomialMatrix(aCoef);

// polynomial matrix B
ContinuousPolynomialMatrix cpmB = new
  ContinuousPolynomialMatrix(bCoef);

// instance of A(s)X(s) = B(s)
// equation solver is created
// solution X(s) is found
ContinuousAXB axb = new ContinuousAXB(cpmA, cpmB);
ContinuousPolynomialMatrix cpmX = axb.getX();
```

Figure 3.29: Solving of linear equation

Algorithm

The algorithm is described in [8, 17]. The linear equation with continuous polynomial matrices **A**(s)**X**(s) = **B**(s), where **A**(s)**,** **B**(s) are known polynomial matrices and **X**(s) is searched polynomial matrix, is converted to the linear equation with constant matrices

$$\mathbf{A_S X_C} = \mathbf{B_C}, \tag{3.9}$$

where

$$\mathbf{A}_S = \begin{pmatrix} \mathbf{A}_0 & 0 & \cdots & 0 \\ \mathbf{A}_1 & \mathbf{A}_0 & \cdots & 0 \\ \vdots & \vdots & \ddots & 0 \\ \mathbf{A}_{n_A} & \mathbf{A}_{n_{A-1}} & \ddots & \mathbf{A}_0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \mathbf{A}_{n_A} \end{pmatrix}, \mathbf{X_C} = \begin{pmatrix} \mathbf{X}_0 \\ \mathbf{X}_1 \\ \vdots \\ \mathbf{X}_{n_X} \end{pmatrix}, \mathbf{B_C} = \begin{pmatrix} \mathbf{B}_0 \\ \mathbf{B}_1 \\ \vdots \\ \mathbf{B}_{n_B} \\ \vdots \\ 0 \end{pmatrix}$$

and

$$\mathbf{A}(s) = \mathbf{A}_0 + \mathbf{A}_1 s + \ldots + \mathbf{A}_{n_A} s^{n_A},$$

$$\mathbf{X}(s) = \mathbf{X}_0 + \mathbf{X}_1 s + \ldots + \mathbf{X}_{n_X} s^{n_X},$$

$$\mathbf{B}(s) = \mathbf{B}_0 + \mathbf{B}_1 s + \ldots + \mathbf{B}_{n_B} s^{n_B}.$$

At first lower and upper bounds of degree n_X of searched polynomial matrix **X**(s) are found. Binary halving algorithm finds the minimal degree n_X, if exists (ranks of Sylvester matrix $\mathbf{A_S}$must equal to rank of extended matrix $\begin{pmatrix} \mathbf{A_S} & \mathbf{B_C} \end{pmatrix}$). Then solution of equation 3.9 is found. And finally polynomial matrix **X**(s) is recreated from column matrix $\mathbf{X_C}$.

3.6.2 Method `getX()`

Gets solution **X**(s) of linear equation **A**(s)**X**(s) = **B**(s).

Syntax

```
public ContinuousPolynomialMatrix getX()
```

Example

See example in 3.29.

3.7 Class `DiscreteAXB`

This class enables solving of linear equation **A**(z)**X**(z) = **B**(z) with discrete (one-sided only) polynomial matrices.

Package

```
cz.ctu.fee.dce.polynomial
```

3.7.1 Constructor

Solves linear equation **A**(z)**X**(z) = **B**(z) with polynomial matrices.

Syntax

```
public DiscreteAXB(DiscretePolynomialMatrix aA,
DiscretePolynomialMatrix aB) throws PMAXBException
```

Example

Figure 3.30 shows how linear equation

$$\begin{pmatrix} 1 & 0 \\ z^{-1} & 1 \end{pmatrix} \mathbf{X}(z) = \begin{pmatrix} z^{-1} \\ 1 \end{pmatrix}$$

is solved. The solution

$$\mathbf{X}(z) = \begin{pmatrix} z^{-1} \\ -1z^{-2} + 1 \end{pmatrix}$$

is found.

Algorithm

One sided discrete polynomial matrices **A**(z) and **B**(z) are converted to continuous polynomial matrices. Continuous solution (see 3.6) is found and it is converted to discrete polynomial matrix.

3.7.2 Method `getX()`

Gets solution **X**(z) of linear equation **A**(z)**X**(z) = **B**(z).

Syntax

```
public DiscretePolynomialMatrix getX()
```

Example

See example in 3.30.

3.8 Class `AXBYC`

This class enables solving of linear equation **A**(s)**X**(s) + **B**(s)**Y**(s) = **C**(s) with continuous polynomial matrices.

Package

```
cz.ctu.fee.dce.polynomial
```

```
// coefficients of polynomial matrix A
double[][][] aCoef = {
  {{ 0, 0}, { 1, 0}}, // coefficients at z^-1
  {{ 1, 0}, { 0, 1}}  // coefficients at z^0
};

// coefficients of polynomial matrix B
double[][][] bCoef = {
  {{ 1}, { 0}}, // coefficients at z^-1
  {{ 0}, { 1}}  // coefficients at z^0
};

// polynomial matrix A
DiscretePolynomialMatrix dpmA =
  new DiscretePolynomialMatrix(aCoef, 1);

// polynomial matrix B
DiscretePolynomialMatrix dpmB =
  new DiscretePolynomialMatrix(bCoef, 1);

// instance of A(z)X(z) = B(z)
// equation solver is created
// solution X(z) is found
DiscreteAXB axb = new DiscreteAXB(dpmA, dpmB);
DiscretePolynomialMatrix dpmX = axb.getX();
```

Figure 3.30: Solving of linear equation

3.8.1 Constructor

Solves linear equation **A**(s)**X**(s) + **B**(s)**Y**(s) = **C**(s) with polynomial matrices.

Syntax

```
public AXBYC (ContinuousPolynomialMatrix aA,
ContinuousPolynomialMatrix aB, ContinuousPolynomialMatrix aC)
throws PMAXBYCException
```

Example

Figure B.1 shows how linear equation

$$\begin{pmatrix} 1 & 0 \\ s & 1 \end{pmatrix} \mathbf{X}(s) + \begin{pmatrix} 1 & 0 \\ s & 1 \end{pmatrix} \mathbf{Y}(s) = \begin{pmatrix} 2s \\ 2 \end{pmatrix}$$

is solved. Solutions

$$\mathbf{X}(s) = \begin{pmatrix} 2s \\ -2s^2 \end{pmatrix} \text{ and } \mathbf{Y}(s) = \begin{pmatrix} 0 \\ 2 \end{pmatrix}$$

are found.

Algorithm

The linear equation with continuous polynomial matrices **A**(s)**X**(s) + **B**(s)**Y**(s) = **C**(s) is converted to the linear equation

$$\begin{pmatrix} \mathbf{A}(s) & \mathbf{B}(s) \end{pmatrix} \begin{pmatrix} \mathbf{X}(s) \\ \mathbf{Y}(s) \end{pmatrix} = \mathbf{C}(s), \tag{3.10}$$

where **A**(s)**, B**(s), **C**(s) are known polynomial matrices and **X**(s), **Y**(s) are searched polynomial matrices. Solution of equation 3.10is found with usage of class `ContinuousAXB` (see 3.6). This solution in split into two searched polynomial matrices **X**(s), **Y**(s).

3.8.2 Method `getX()`

Gets solution **X**(s) of linear equation **A**(s)**X**(s) + **B**(s)**Y**(s) = **C**(s).

Syntax

```
public ContinuousPolynomialMatrix getX()
```

Example

See example in B.1.

3.8.3 Method `getY()`

Gets solution **Y**(s) of linear equation **A**(s)**X**(s) + **B**(s)**Y**(s) = **C**(s).

Syntax

```
public ContinuousPolynomialMatrix getY()
```

Example

See example in B.1.

3.9 Class `MathMl`

This class enables exporting polynomial matrices into commonly used formats.

Package

```
cz.ctu.fee.dce.polynomial.utils
```

3.9.1 The MathML Format

MathML defined by W3C is intended to facilitate the use and re-use of mathematical and scientific content on the web, and for other applications such as computer algebra systems, print typesetting, and voice synthesis. MathML can be used to encode both the presentation of mathematical notation for high-quality visual display, and mathematical content, for applications where the semantics plays more of a key role such as scientific software or voice synthesis.

MathML is cast as an application of XML. As such, with adequate style sheet support, it will ultimately be possible for browsers to natively render mathematical expressions. For the immediate future, several vendors (e.g. MathPlayer by Design Science) offer applets and plug-ins which can render MathML in place in a browser [3].

Robert Hornych, a graduate of class 2001 at CTU FEE Department of Control Engineering, devised a couple of Matlab functions for converting polynomial matrix objects from Matlab to MathML and vice versa. Mathematica and Maple claim the same features in their latest releases. Converting polynomial matrix objects to MathML is also available in this package using class MathMl. An example of polynomial matrix displayed by an Internet browser is shown in figure E.3.

3.9.2 Method `pmToMml(String, PolynomialMatrix)`

This method converts polynomial matrix (both continuous and discrete) into MathML format. Polynomial matrix stored in MathML format can be for example very easily presented on web pages [15].

Syntax

```
public static void pmToMml(String aFile, PolynomialMatrix aPm)
throws Exception
```

Example

Figure 3.31 shows implementation of conversion polynomial matrix

$$\begin{pmatrix} 1-3s^2 & 8s-4s^2 \\ -6s^2 & 2-7s^2 \end{pmatrix}$$

into MathML format. Generated file is shown in figure B.2.

```
// coefficients of polynomial matrix A
double[][][] aCoef = {
  {{ 1, 0}, { 0, 2}}, // coefficients at s^0
  {{ 0, 8}, { 0, 0}}, // coefficients at s^1
  {{-3,-4}, {-6,-7}}  // coefficients at s^2
};

// polynomial matrix A
ContinuousPolynomialMatrix cpmA =
  new ContinuousPolynomialMatrix(aCoef);

// saves matrix A into matrix.mml
// file(MathML format)
MathMl.pmToMml("matrix.mml", cpmA);
```

Figure 3.31: Conversion of polynomial matrix into MathML

Algorithm

At first DOM (Document Object Model) [2]is created from existing polynomial matrix object. Created DOM corresponds to MathML structure having presentation markup elements [15]. It is transformed [2] to the standard XML format and saved as a file.

3.9.3 Method `transformMml()`

This method transforms polynomial matrix stored in MathML format into format defined by given transformation.

As it was said in subsection 3.9.1 the MathML format is universal format for mathematical notations. It is transformable by XSLT into any other format by existing transformation file (XSL) [2]. For example polynomial matrix saved in MathML format can be transformed into both HTML or TEX format.

Syntax

```
public static void transformMml(String, String, String)
throws Exception
```

Example

Figure 3.32 shows conversion of file in MathML format (it can be polynomial matrix) into file having TEX format using XSLT transformation.

```
// transforms existing matrix matrix.mml in MathML
// format into matrix.tex file in TeX format using
// XSLT transformation defined in mmltex.xsl file
MathMl.transformMml("matrix.mml", "xsl/mmltex.xsl",
  "matrix.tex");
```

Figure 3.32: Conversion of MathML file into TEX file

Algorithm

The XSLT transformation [2] is used.

Chapter 4

Tests

It is necessary to test implemented methods in order to provide usable library. All methods must prove correct functionality of implemented algorithms. Functionality tests are discussed in the first section. If an error was reported by functionality test, it was fixed. Functionality tests can be launched by library user to see the error report. It is also necessary to have as fast as possible algorithms enabling on-line usage. Performance tests of methods are described in the second section. The performance of algorithms was taken in account concerning Java and numerical computing during implementation. The performance was improved where necessary and possible.

4.1 Functionality Tests

In this section it is described how functionality tests are constructed. It is explained how tests are implemented using JUnit framework and simple example of functionality test is shown.

Functionality test are called JUnit tests because JUnit testing framework [4, 13] is used. Functional testing is based on comparing expected and computed output data for the same input data. Input and expected data are easily generated by Polynomial Toolbox for Matlab [17] and exported to Java format or they can be created by hand. Expected and computed data are compared using some of overloaded static methods `Assert.assertEquals()`. Each method of provided programming interface (all public methods are tested) is tested for several different data. It is tested for randomly generated data and then for special cases, i.e. case when exception is thrown or zero division, etc. Not only output data are tested but input data are tested that they were not changed as well. Test of each method for particular data is implemented in particular *test methods*. Figure 4.1 shows the example of test method for polynomial matrices addition. The following paragraph describes tests structure.

The test class called *test case* extending `TestCase` class is created for each class of provided programming interface. It contains test methods. Most of test methods uses same input data. This data are set in overloaded method `setUp()` and destroyed in overloaded method `tearDown()`. These methods are launched before, respectively after, running of each test method automatically by JUnit framework. All test cases are launched from class called *test suite*.

```
public void testAdd() {

  // expected matrix A for comparison
  ContinuousPolynomialMatrix expectedA =
    new ContinuousPolynomialMatrix(expectedCoef1);

  // expected matrix B for comparison
  ContinuousPolynomialMatrix expectedB =
    new ContinuousPolynomialMatrix(bCoef);

  try {
    // tested method variables cpmA and cpmB
    // were set in setUp() method
    cpmA.add(cpmB);

  } catch (Exception e) {
    // handle exception
    System.out.println(e);
  }

   // comparison of expected and output data
   Assert.assertEquals(expectedA, cpmA);
   Assert.assertEquals(expectedB, cpmB);
}
```

Figure 4.1: The example of test method

Test suite displays tests structure and their results. Methods `Assert.assertEquals()` produce messages when errors occur. Classes with JUnit tests are placed in package `cz.ctu.fee.dce.polynomial.tests.junit` and they are listed in appendix A. Examples of JUnit test output can be found in appendix C.

4.2 Performance Tests

It is described how performance tests are constructed in this section. It is explained how tests are implemented using framework for performance tests and simple example of performance test is shown.

The framework for launching performance tests (see appendix A) was written. Performance tests are based on measuring duration of tested method. Input data are randomly generated for

tests. Time is measured by using `Time.init()` and `Time.getElapsedTime()` methods (see figure A.4, [16]). Test of each method is implemented in *test method*. Each test method writes result of test (elapsed time) to a text file and generates Matlab command into m-file for launching the same test in Polynomial Toolbox for Matlab [17]. Figure 4.2 shows the example of test method for polynomial matrices addition. The following paragraph describes tests structure.

```
public void testMutliplyDefault() throws Exception {

  // saves the initial time
  Time.init();

  // performs test method, cpmA and cpmB
  // are created by setUp() method
  cpmA.multiply(cpmB,
    PolynomialMatrix.MULTIPLY_DEFAULT
  );

  // saves elapsed time to file and
  // generates Matlab command A*B to m-file
  writeResult(Time.getElapsedTime(), "A*B");
}
```

Figure 4.2: The example of test method

The test class called *test case* extending `PerformanceTest` class is created for each class of provided programming interface. It contains test methods. Most of test methods use the same input data. This data are set in overloaded method `setUp()` and destroyed in overloaded method `tearDown()`. These methods are launched before, respectively after, running of each test method automatically by test framework. Test cases are launched from `PerformanceRunner` class. It uses Java Reflection [22] for launching test cases. Each test case is launched several times for different sizes of matrix and different degrees of matrix. Number of runs, maximal matrix size and maximal matrix degree are given as parameters. Results of test (test case) are written to the text file for further statistical processing and Matlab m-file is generated for performing the same test case in Polynomial Toolbox for Matlab [17]. All test cases are launched from class `AllPerformanceTests` called *test suite*. Classes with performance tests are placed in package `cz.ctu.fee.dce.polynomial.tests.performance` and they are listed in appendix A.

Both JUnit and performance tests were described. Graphical results of performance tests can be found in appendix D. Test results are assessed in the following chapter.

Chapter 5

Conclusion

This chapter includes summary of the whole thesis with comparison of the proposed package with Polynomial Toolbox for Matlab. Some future plans for the package are outlined.

5.1 Comparison with Polynomial Toolbox for Matlab

Methods that use more complex algorithms were tested for their time performance on several environments and compared to corresponding methods of Polynomial Toolbox version 3.0.10 for Matlab 6.5 (see appendix D). These results were found out:

- Computational times increase slower with increasing degree than with increasing size of the polynomial matrix. It is due to proposed storage of coefficients of polynomial matrix. Coefficients are stored as 3-dimensional array where degree index is at the first dimension and row and column indexes are at the second and at the third index of array.
- The method for computing value of polynomial matrix at given point is faster than corresponding method in Polynomial Toolbox for Matlab. Polynomial Toolbox converts scalar to constant matrix of the same sizes as sizes of polynomial matrix and evaluates polynomial matrix in three nested loops contrary to one loop without any conversions of scalar in Java package.
- Scaling method has got much better performance than the scaling method in Polynomial Toolbox for Matlab. It might be caused by several checks of input arguments correctness in Polynomial Toolbox contrary to no need to check input arguments in Java package.
- All algorithms for two-sided polynomial matrices seem to be as fast as the algorithms for one-sided polynomial matrices contrary to slower methods for two-sided polynomial matrices in Polynomial Toolbox for Matlab.
- Methods which use discrete Fourier transform (determinant, roots, rank) have worse performance than methods in Polynomial Toolbox for Matlab. Class `PolynomialMatrixFFT` uses fast Fourier transform algorithm of set of points. It would be more efficient if

existed method for fast Fourier transform of set of constant matrices (at the time of implementation library JMSL 2.0 included classes for fast Fourier transform of set of points only). This is the reason why default algorithm for multiplication of polynomial matrices is faster than the algorithm using discrete Fourier transform of polynomial matrix.

- The performance of the other methods is comparable to performance of corresponding methods in Polynomial Toolbox for Matlab.

5.2 Summary

An object-oriented library application programming interface enabling operating on polynomial matrices was created. It was designed, implemented and properly tested both for functionality and for performance.

The structure of classes was designed while considering advantages and disadvantages of Java language for numerical computing. Working with multidimensional arrays and complex numbers are the most important issues that were considered during analysis. These aspects had the main influence on choice of Java library for computing with constant matrices. The algorithms for operating on polynomial matrices are based on algorithms for operating on constant matrices. The JMSL 2.0 library by Visual Numerics was chosen. There are two classes `ContinuousPolynomialMatrix` and `DiscretePolynomialMatrix` inheriting from abstract class `PolynomialMatrix`. Base functionality common for both child classes is included in `PolynomialMatrix`. `ContinuousPolynomialMatrix` enables operating on continuous polynomial matrices and `DiscretePolynomialMatrix` enables operating discrete-time polynomial matrices or two-sided polynomial matrices.

Base classes were implemented in correspondence with proposed structure. They contain methods that enable performing basic linear algebra operations on polynomial matrices. The class `PolynomialMatrixFFT` for inverse and direct discrete Fourier transform of polynomial matrix was implemented using fast Fourier transform algorithm. This algorithm is used in some algorithms like computing determinant or rank of polynomial matrix. Classes `ContinuousAXB`, `DiscreteAXB` and `AXBYC` were created. They can be used for solving linear equations with polynomial matrices.

All methods of programming interface were tested for functionality using JUnit framework. About 100 successful tests were launched. Methods that use more complex algorithms were tested for their time performance on several environments and compared to corresponding methods of Polynomial Toolbox for Matlab.

This is fully functional, usable and documented initial version of library providing application programming interface for operating on polynomial matrices. It is operating system independent. The JDK (Java Development Kit) distributed for free and JMSL 2.0 by Visual Numerics are needed for its usage. Such a library can be used by programmers who want to develop software (deployed on the Internet or locally) for automatic control system design and signal processing applications, it can be used for educational purposes or by researchers. More information about project can be found at [16], screenshots of project's home page are found in appendix E.

5.3 Future Extensions

It is planned to implement other functionalities like:

- Greatest common divisor (right and left) of two polynomial matrices,
- Fast solvers for $\mathbf{A}(s)\mathbf{X}(s)=0$ based on displacement rank theory for block Toeplitz matrices [26],
- Reliable triangularization of a polynomial matrix [8],
- Spectral factorization of a para-Hermitian polynomial matrix,
  i.e. a quadratic equation $\mathbf{X}(-s)\mathbf{X}(s)=\mathbf{A}(s)$ and $\mathbf{X}(z^{-1})\mathbf{X}(z)=\mathbf{A}(z,z^{-1})$ [20],
- J-spectral factorization of a para-Hermitian polynomial matrix,
  i.e. a quadratic equation $\mathbf{X}(-s)\mathbf{J}\mathbf{X}(s)=\mathbf{A}(s)$ and $\mathbf{X}(z^{-1})\mathbf{J}\mathbf{X}(z)=\mathbf{A}(z,z^{-1})$ [20].

Bibliography

[1] Arlow, J., Neustadt, I. *UML and the Unified Process Practical Object-Oriented Analysis and Design*. Computer Press, 1st edition, 2003.

[2] Armstrong, E., Ball, J., Bodoff, S., Carson, D. B., Fisher, M., Fordin, S., Green, D., Haase, K., Jendrock, E. *The Java Web Services Tutorial <http://java.sun.com/webservices/docs/1.2/tutorial/doc/index.html>*. July 2003.

[3] Ausbrooks, R., Buswell, S., Dalmas, S., Devitt, S., Diaz, A., Hunter, R., Smith, B., Soiffer, N., Sutor, R., Watt, S. *Mathematical Markup Language (MathML) Version 2.0 <http://www.w3.org/TR/MathML2/>*. February 2001.

[4] Beck, K., Gamma, E. *JUnit Cookbook <http://junit.sourceforge.net/doc/cookbook/cookbook.htm>*. December 2003.

[5] Boisvert, R. F., Moreira, J., Philippsen, M., Pozo, R. *Java Grande Forum Report: Making Java Work for High-End Computing <http://www.javagrande.org/sc98/sc98grande.ps>*. Java Grande Forum Panel, SC1998, Orlando, Florida, November 1998.

[6] Eckel, B. *Thinking in Java*. Prentice Hall, 1st edition, 1998.

[7] Halmo, L. *PolPack++ <http://sourceforge.net/projects/polpackplusplus>*. 2004.

[8] Henrion, D. *Reliable Algorithms for Polynomial Matrices <http://www.laas.fr/~henrion/Papers/thesis.ps.gz>*. Ph. D. Thesis, Institute of Information Theory and Automation, Czech Academy of Sciences, Prague, Czech Republic, December 1998. LAAS-CNRS Research Report No. 99003.

[9] Henrion, D., Sebek, M. *Numerical Methods for Polynomial Matrix Rank Evaluation <http://www.laas.fr/~henrion/Papers/rank_IFAC.ps.gz>*. LAAS-CNRS Research Report No. 97356, Proceedings of the IFAC Conference on Systems Structure and Control, pp. 385-390, Nantes, France, July 1998.

[10] Herout, P. *Ucebnice jazyka Java*. Kopp, 1st edition, 2000.

[11] Hromcik, M., Sebek, M. *New Algorithm for Polynomial Matrix Determinant Based on FFT*. In: European Control Conference. ECC '99. (CD-ROM). VDI/VDE GMA, Karlsruhe 1999.

[12] INRIA. *Scilab <http://scilabsoft.inria.fr>*. 2004.

[13] JUnit.org. *JUnit 3.8.1 Javadocs <http://www.junit.org/junit/javadoc/3.8.1/index.htm>*. December 2003.

[14] Maplesoft. *Maple <http://www.maplesoft.com/products/maple>*. 2004.

[15] Miner, R., Schaeffer, J. *A Gentle Introduction to MathML <http://www.dessci.com/en/support/tutorials/mathml/default.htm>*. October 2001.

[16] Padera, M. *Polynomial Matrices in Java, graduate diploma thesis homepage <http://dce.felk.cvut.cz/~paderam.kstudent>*. December 2003.

[17] Polyx Ltd. *The Polynomial Toolbox for Matlab - Commands <http://www.polyx.com/download/commands.pdf.gz>*. 1999.

[18] Polyx Ltd. *The Polynomial Toolbox for Matlab - Manual <http://www.polyx.com/download/manual.pdf.gz>*. 1999.

[19] Polyx Ltd. *Polynomial Toolbox for Matlab <www.polyx.com>*. 2004.

[20] Sebek, M., Kwakernaak, H. *J-spectral factorization.* In Proceedings of the 30th IEEE Conference on Decision and Control, (Brighton, England), pp. 1278-1283, IEEE Control Systems Society, December 1991.

[21] Stewart, G. W. *Matrix Algorithms,* volume Basic Decompositions. SIAM, 1st edition, 1998.

[22] Sun Microsystems, Inc. *JavaTM 2 SDK, Standard Edition Documentation Version 1.4.2 <http://java.sun.com/j2se/1.4.2/docs/index.html>*. 2003.

[23] Visual Numerics. *JMSL Reference Manual <http://www.vni.com/products/imsl/documentation>*. 2002.

[24] Visual Numerics. *JMSL 2.5 Product Sheet <http://www.vni.com/books/dod/pdf/JMSL25PS051503.pdf>*. 2003.

[25] Weisstein, E. W. *Eric Weisstein's World of Mathematics <http://mathworld.wolfram.com/>*. CRC Press LLC, Wolfram Research, Inc., 1999.

[26] Zuniga, J., C., Henrion, D. *Comparison of Algorithms for Computing Infinite Structural Indices of Polynomial Matrices <http://www.laas.fr/~jczuniga/publications_fichiers/ecc03d.pdf>*. LAAS-CNRS Research Report No. 02536, November 2002. Proceedings of the European Control Conference, Cambridge, UK, September 2003.

Appendix A

Design

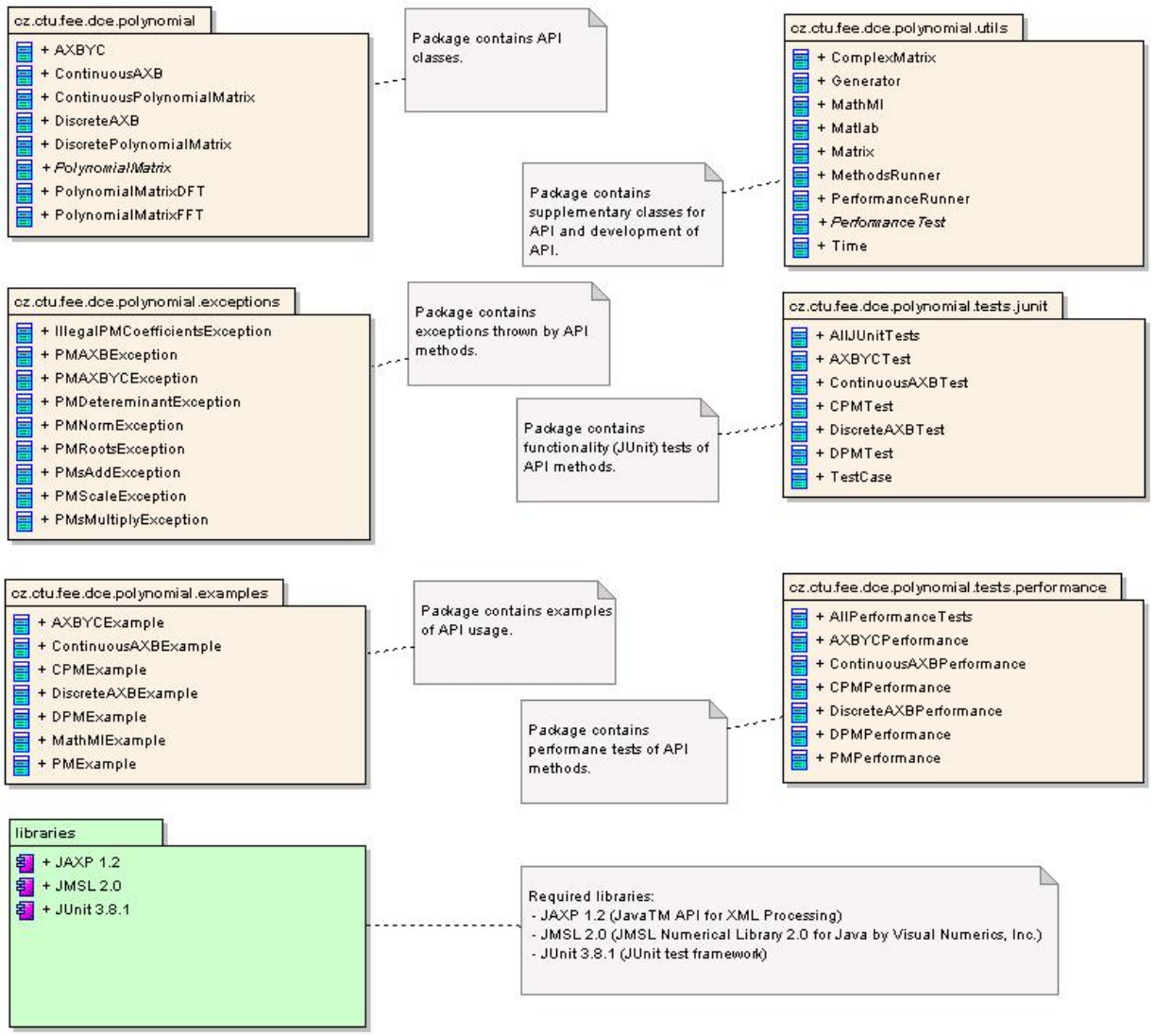


Figure A.1: Package diagram

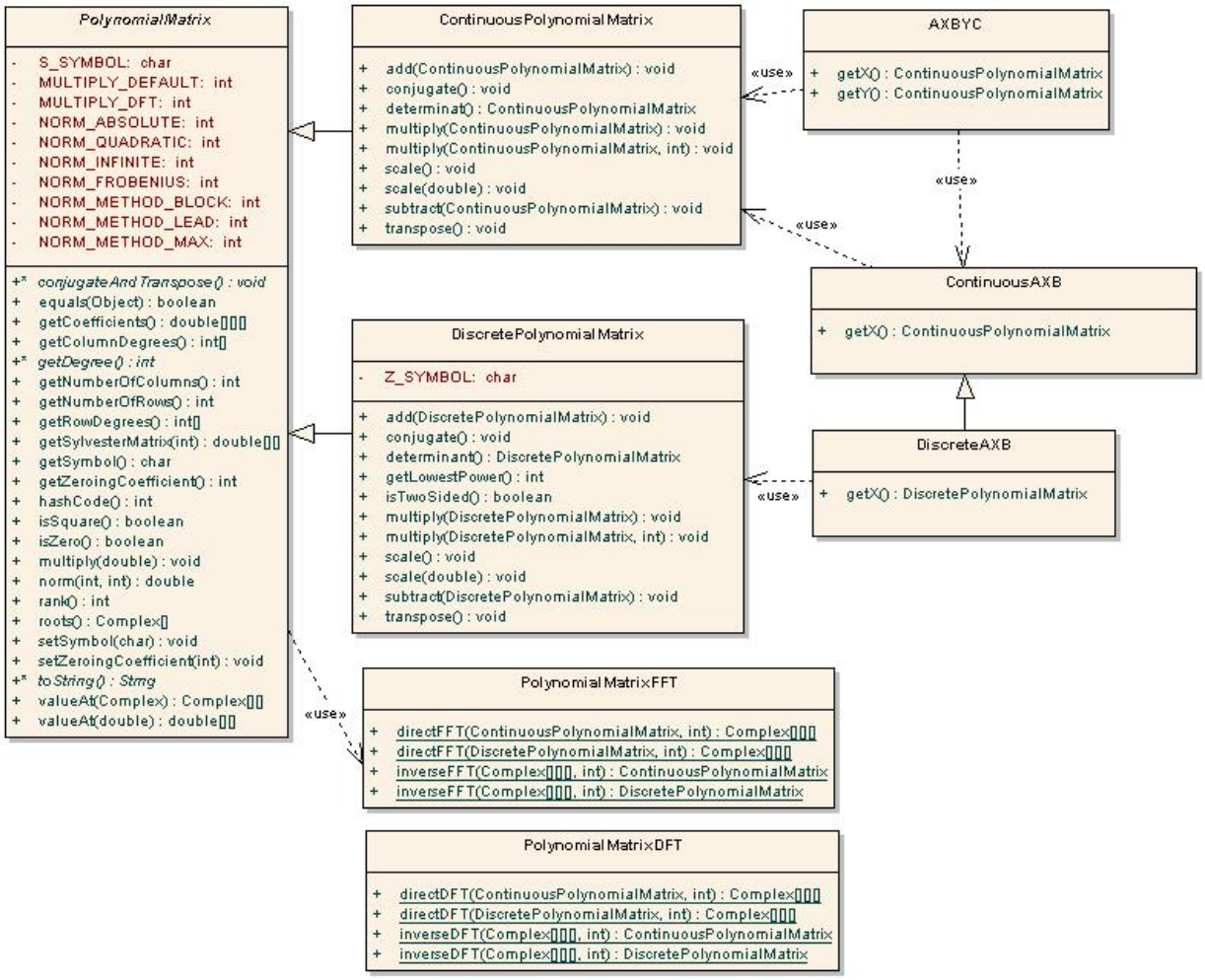


Figure A.2: Class diagram of package `cz.ctu.fee.dce.polynomial`

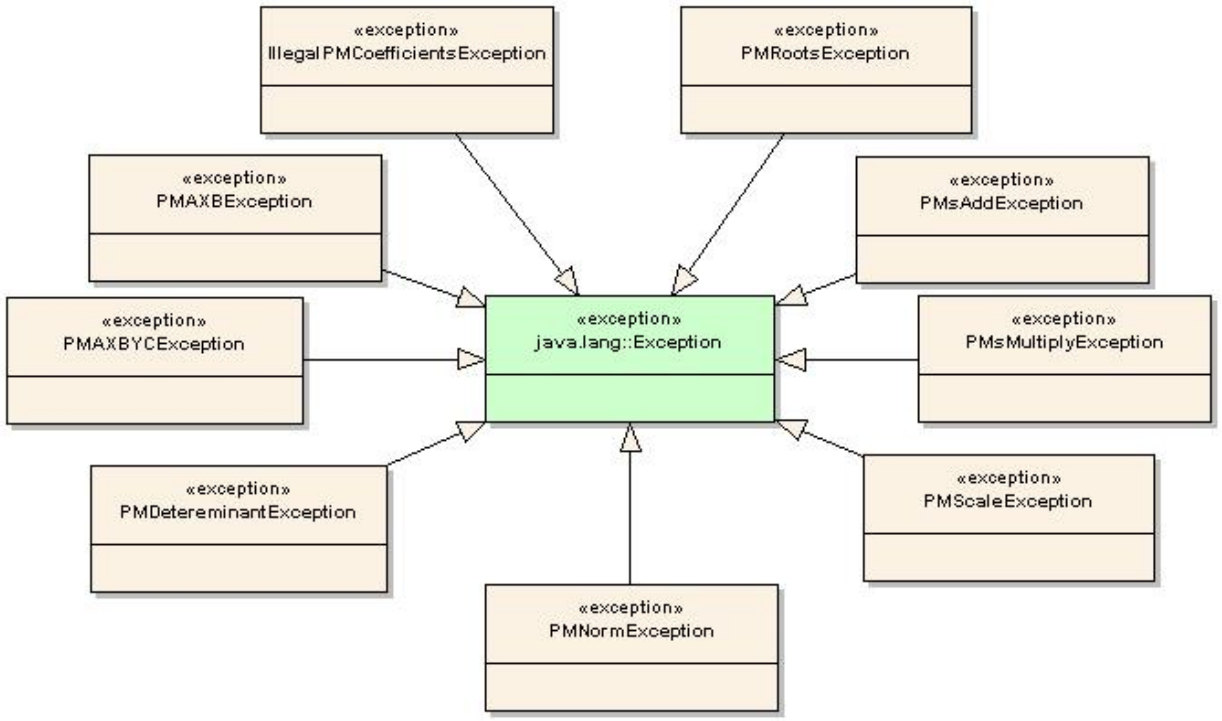


Figure A.3: Class diagram of package `cz.ctu.fee.dce.polynomial.exceptions`

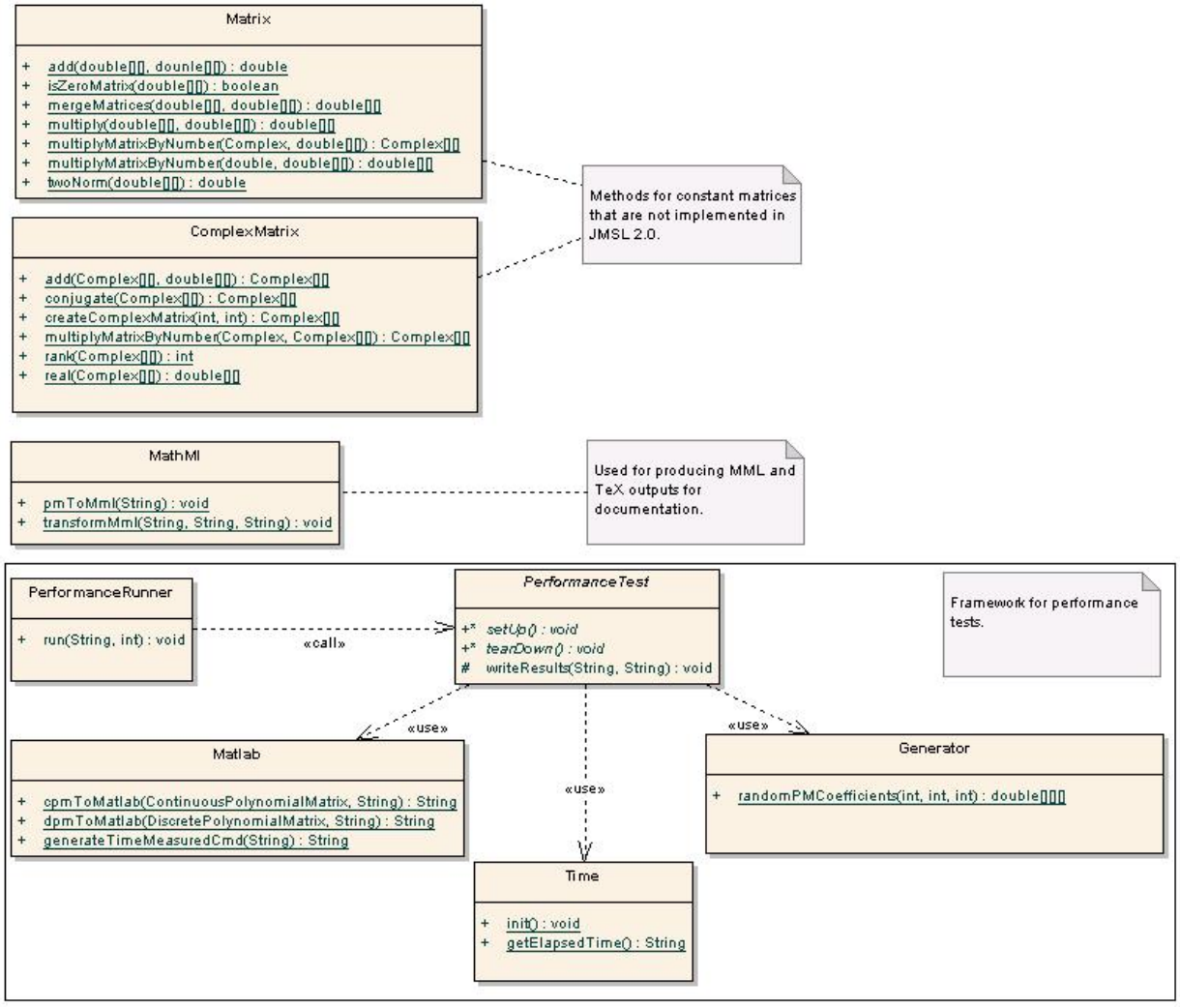


Figure A.4: Class diagram of package `cz.ctu.fee.dce.polynomial.utils`

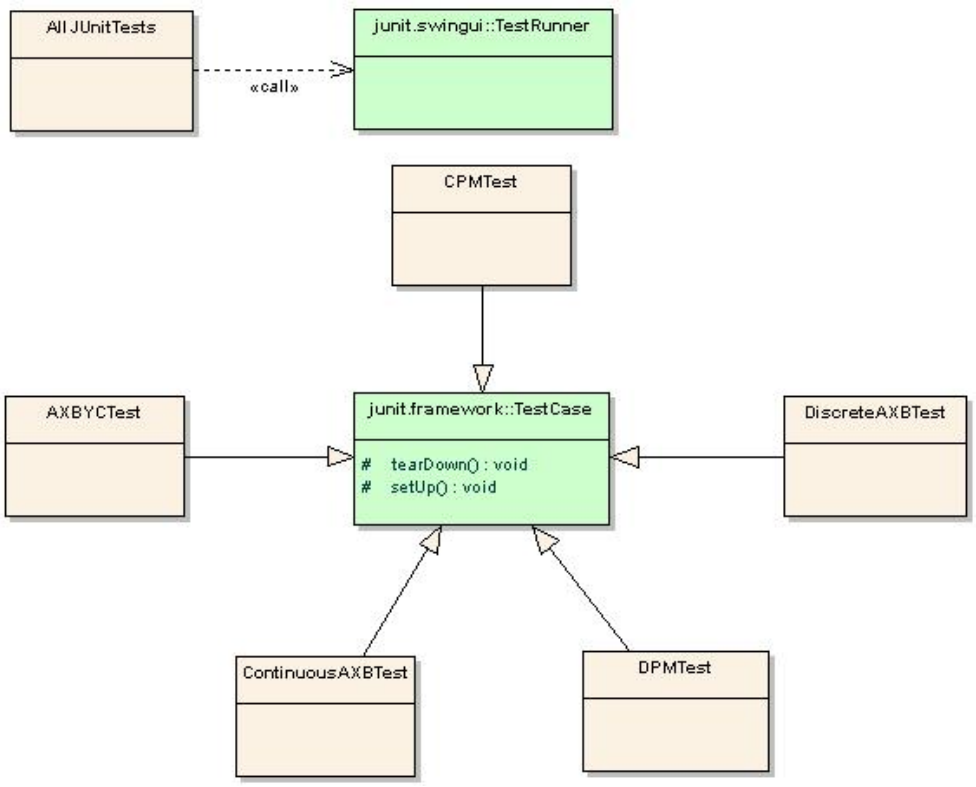


Figure A.5: Class diagram of package `cz.ctu.fee.dce.polynomial.tests.junit`

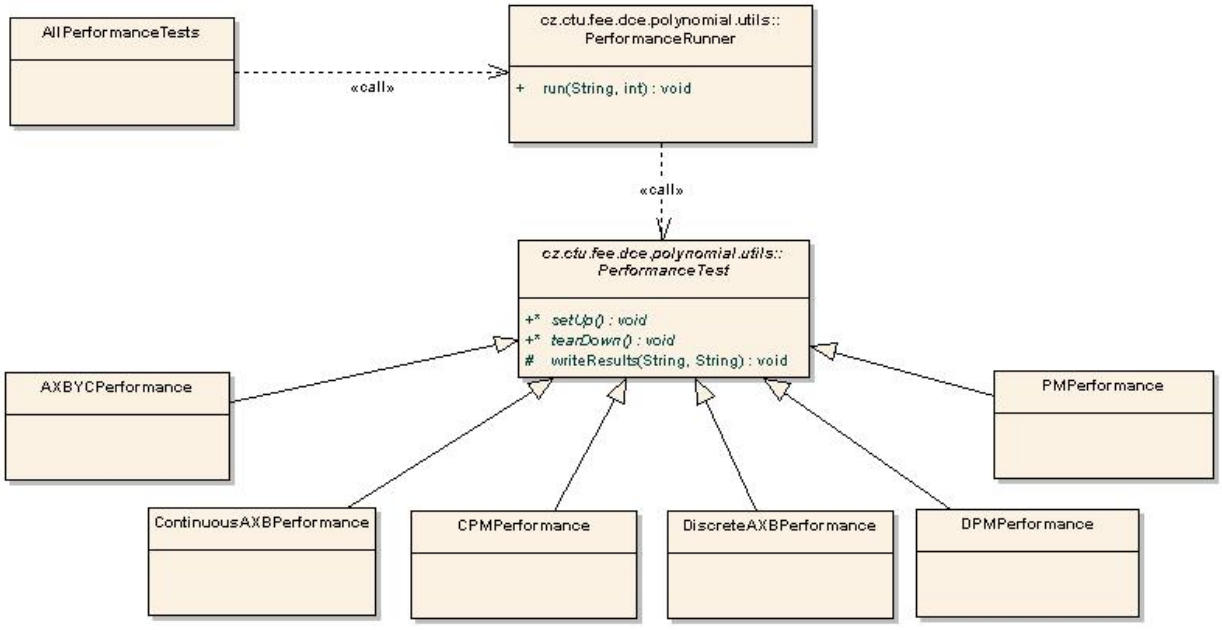


Figure A.6: Class diagram of package `cz.ctu.fee.dce.polynomial.tests.performance`

Polynomial matrix:

$$\begin{pmatrix} 1-3s^2 & 8s-4s^2 & 3-5s^2 \\ -6s^2 & 2-7s^2 & -2s-8s^2 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 3 \\ 0 & 2 & 0 \end{pmatrix} s^0 + \begin{pmatrix} 0 & 8 & 0 \\ 0 & 0 & -2 \end{pmatrix} s^1 + \begin{pmatrix} -3 & -4 & -5 \\ -6 & -7 & -8 \end{pmatrix} s^2$$

Coefficients in Java:

```
// coefficients of polynomial matrix
double[][][] coefficients = {
    {{ 1, 0, 3}, { 0, 2, 0}}, // coefficients at s^0
    {{ 0, 8, 0}, { 0, 0,-2}}, // coefficients at s^1
    {{-3,-4,-5}, {-6,-7,-8}}  // coefficients at s^2
};
```

Storage of coefficients:

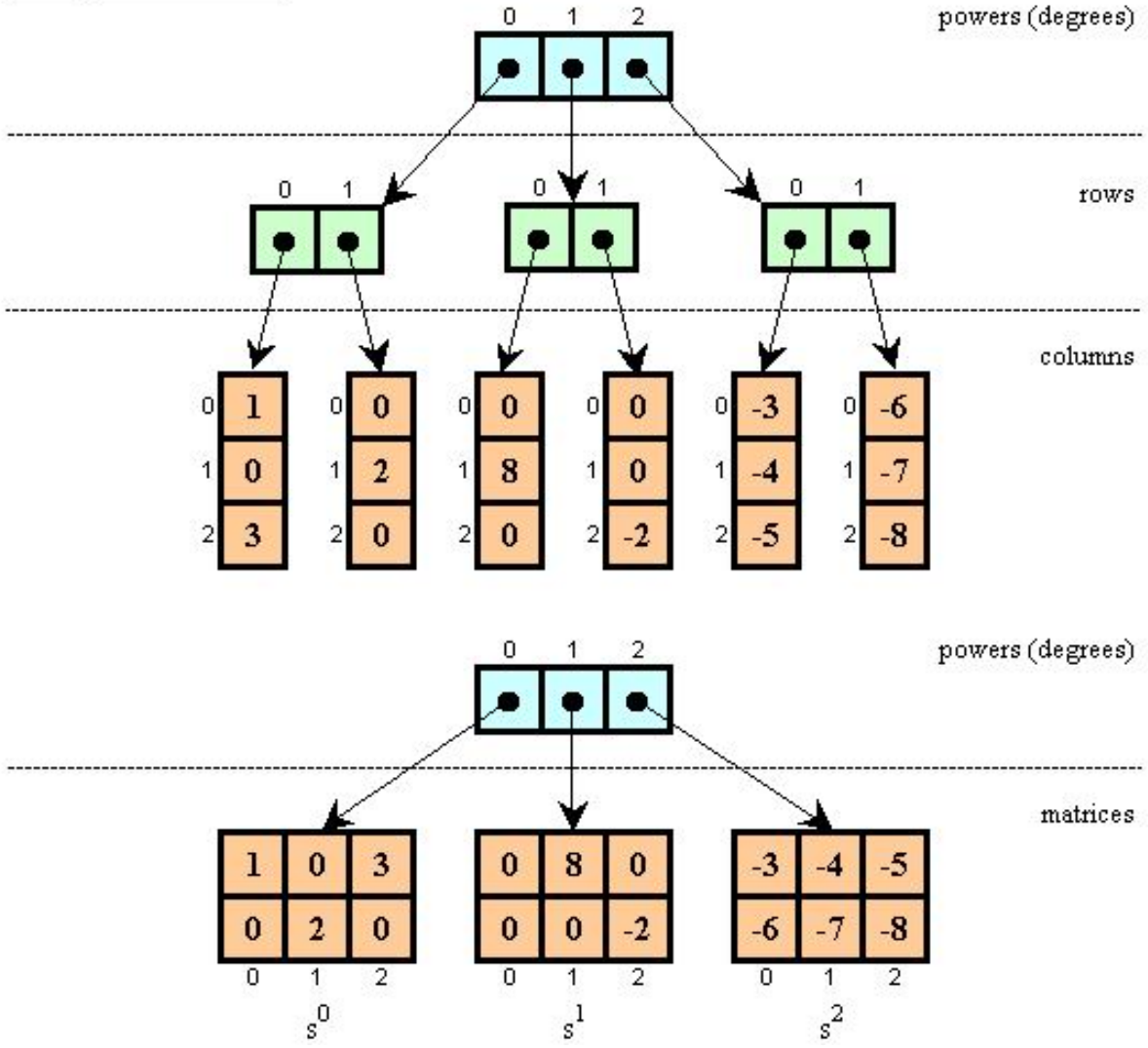


Figure A.7: Storage of coefficients

Appendix B

Code Examples

```
// coefficients of polynomial matrix A
double[][][] aCoef = {
  {{1, 0}, {0, 1}}, // coefficient at s^0
  {{0, 0}, {1, 0}}  // coefficient at s^1
};
// coefficients of polynomial matrix B
double[][][] bCoef = {
  {{1, 0}, {0, 1}}, // coefficient at s^0
  {{0, 0}, {1, 0}}  // coefficient at s^1
};
// coefficients of polynomial matrix C
double[][][] cCoef = {
  {{0}, {2}}, // coefficient at s^0
  {{2}, {0}}  // coefficient at s^1
};

// polynomial matrix A
ContinuousPolynomialMatrix cpmA = new
  ContinuousPolynomialMatrix(aCoef);

// polynomial matrix B
ContinuousPolynomialMatrix cpmB = new
  ContinuousPolynomialMatrix(bCoef);

// polynomial matrix C
ContinuousPolynomialMatrix cpmC = new
  ContinuousPolynomialMatrix(cCoef);

// instance of A(s)X(s) + B(s)Y(s) = C(s)
// equation solver is created
// solution X(s) and Y(s) is found
AXBYC axbyc = new AXBYC(cpmA, cpmB, cpmC);

// solution X(s) and Y(s)
ContinuousPolynomialMatrix cpmX = axbyc.getX();
ContinuousPolynomialMatrix cpmY = axbyc.getY();
```

Figure B.1: Solving of linear equation

```
<?xml version="1.0" encoding="UTF-8"?>
<math xmlns="http://www.w3.org/1998/Math/MathML">
  <mfenced close=")" open="(">
    <mtable>
      <mtr>
        <mtd><mrow>
            <mo/><mn>1</mn>
            <mo>-</mo><mn>3</mn>
            <msup><mi>s</mi><mn>2</mn></msup>
        </mrow></mtd>
        <mtd><mrow>
            <mo/><mn>8</mn><mi>s</mi>
            <mo>-</mo><mn>4</mn>
            <msup><mi>s</mi><mn>2</mn></msup>
        </mrow></mtd>
      </mtr>
      <mtr>
        <mtd><mrow>
            <mo>-</mo><mn>6</mn>
            <msup><mi>s</mi><mn>2</mn>
            </msup>
        </mrow></mtd>
        <mtd><mrow>
            <mo/><mn>2</mn>
            <mo>-</mo><mn>7</mn>
            <msup><mi>s</mi><mn>2</mn>
            </msup>
        </mrow></mtd>
      </mtr>
    </mtable>
  </mfenced>
</math>
```

Figure B.2: Polynomial matrix in MathML format

Appendix C

Functionality Tests

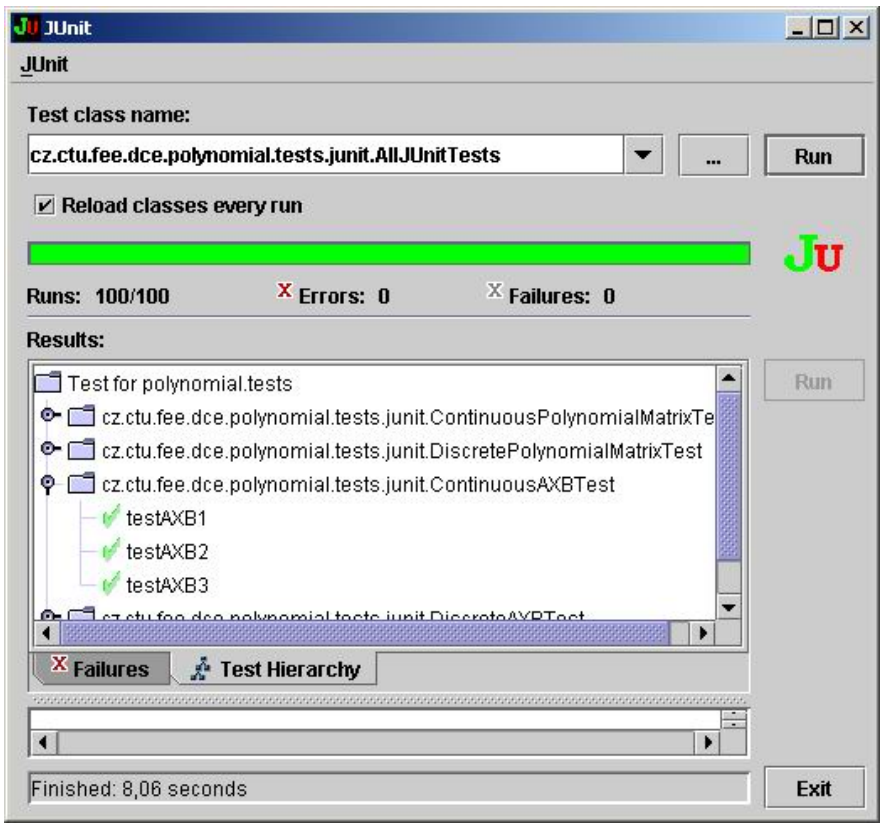


Figure C.1: Output of JUnit tests - all tests passed

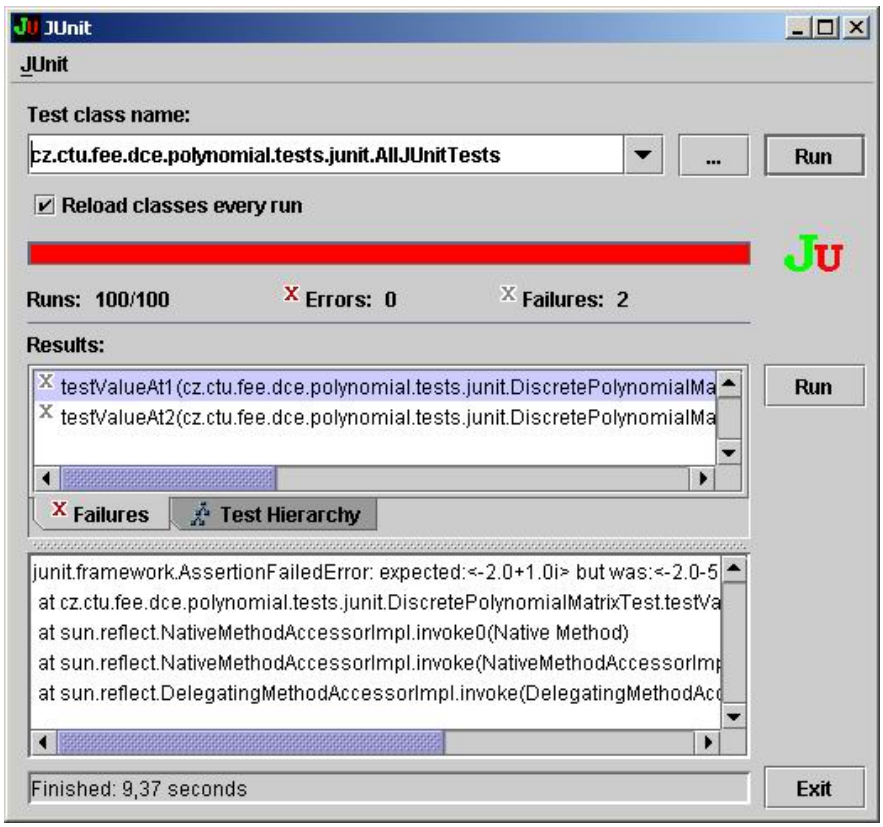


Figure C.2: Output of JUnit test - tests with errors

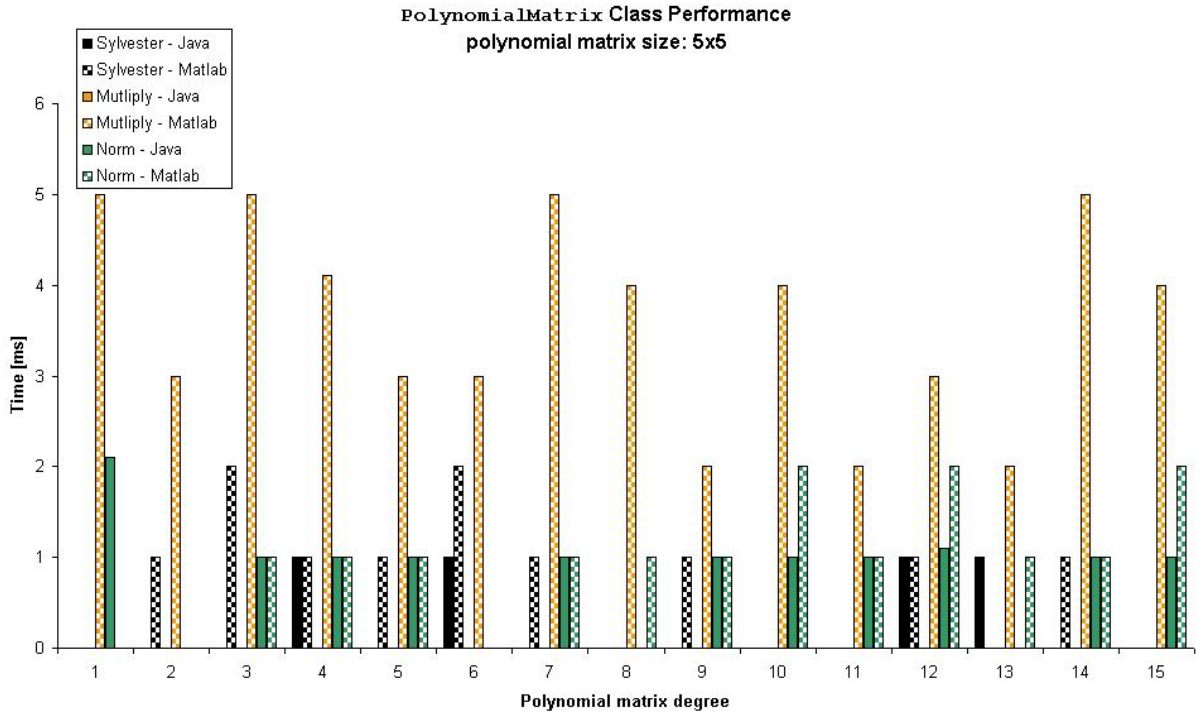
Appendix D

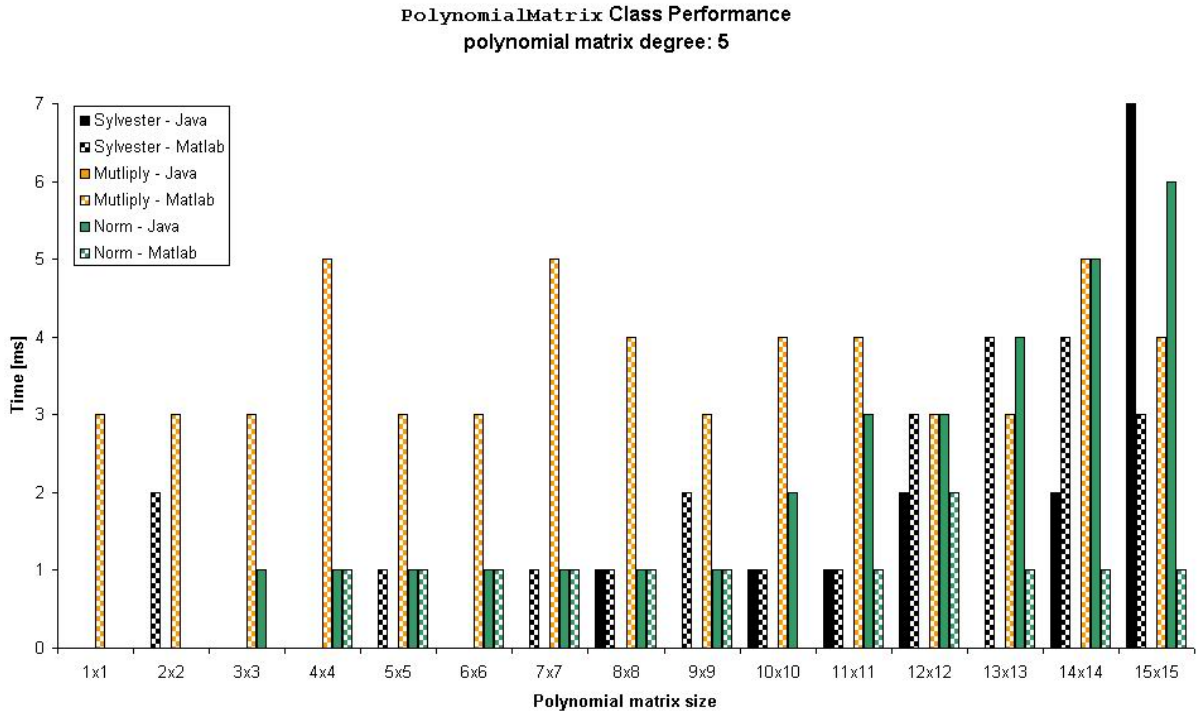
Performance Tests

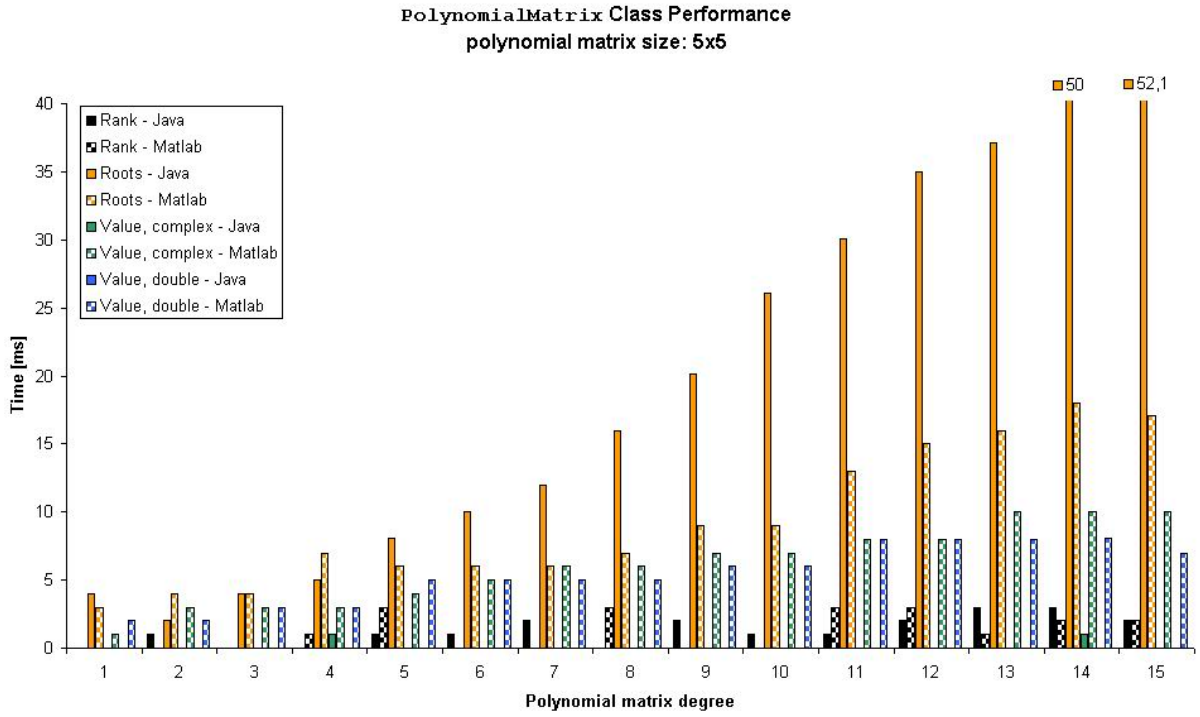
Test Protocol

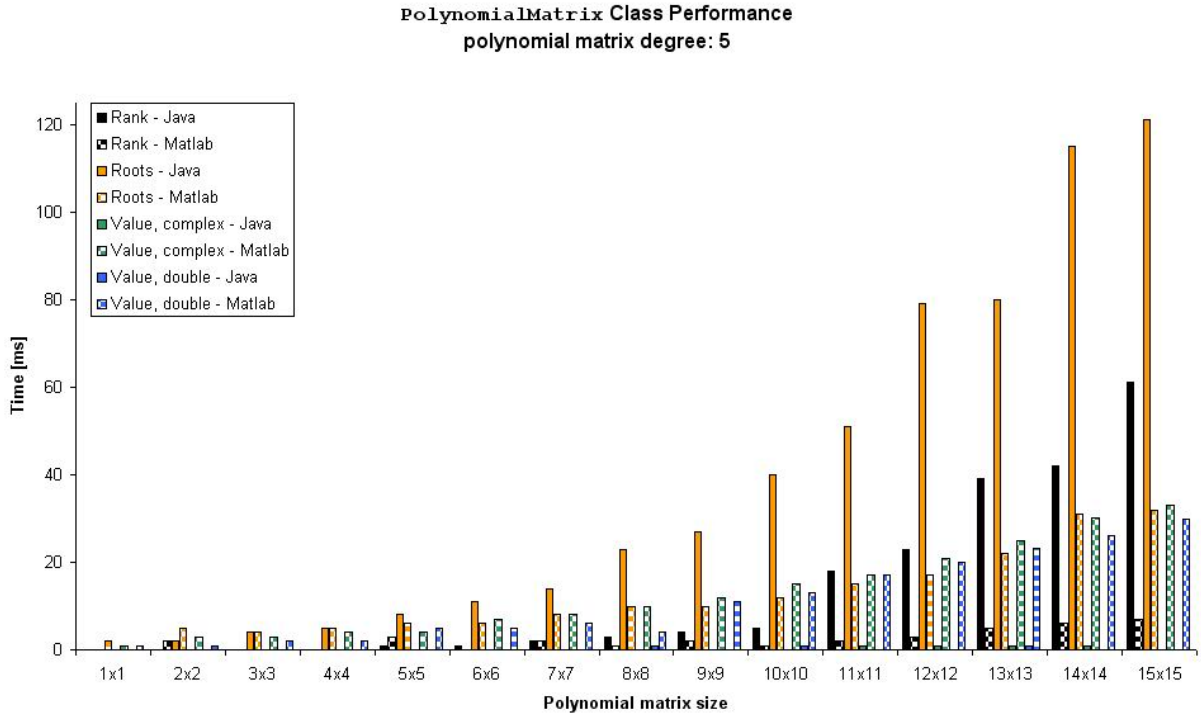
Date	2003/12/13	
Polynomial Matrices in Java	build 200311131804 (2003/11/13 18:04)	
OS	Windows XP Home Edition (version 5.1, build 2600)	
JRE	Sun Java 2 SE 1.4.2	
Matlab	6.5.0.180913a (R13)	
Polynomial Toolbox	3.0.10	
CPU	Intel Celeron, 1200 MHz, 256 kB L2 Cache	
RAM	384 MB	
Java	**Before Test**	**After Test**
Total Memory	945 MB	945 MB
Total Memory Occupied	88 MB	88 MB
Total Memory Occupied - Peak	148 MB	164 MB
Matlab	**Before Test**	**After Test**
Total Memory	945 MB	945 MB
Total Memory Occupied	133 MB	247 MB
Total Memory Occupied - Peak	164 B	269 MB

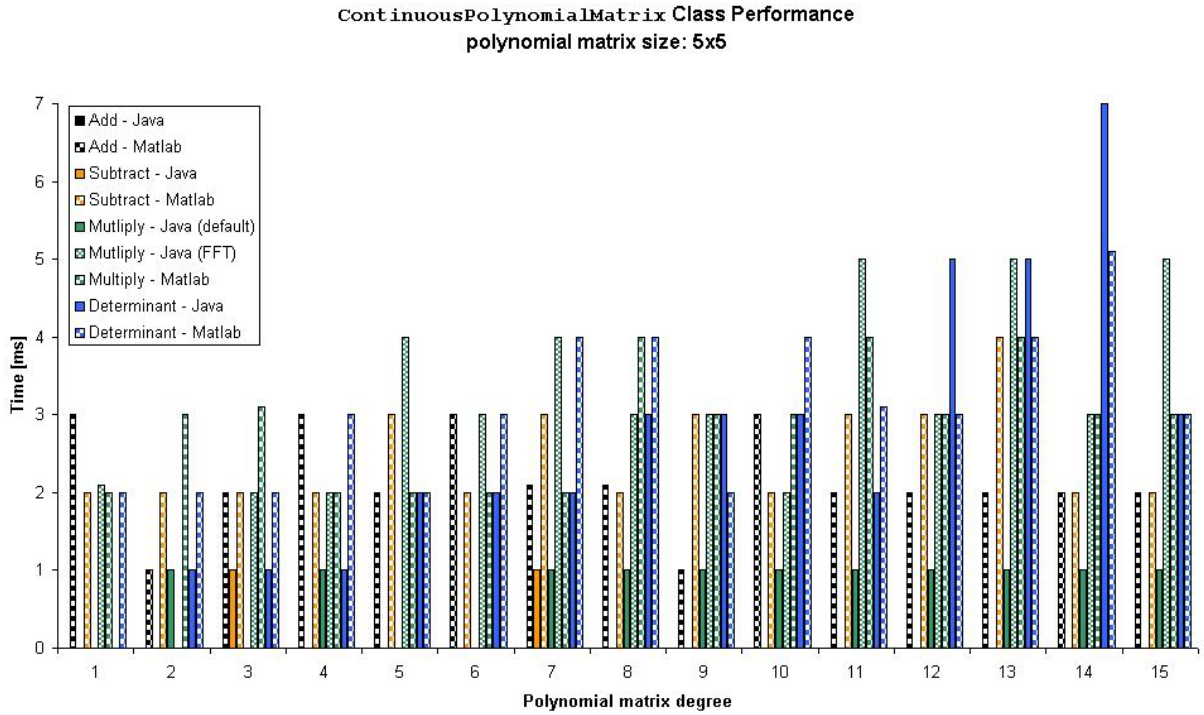
Figure D.1: Test protocol of performance tests

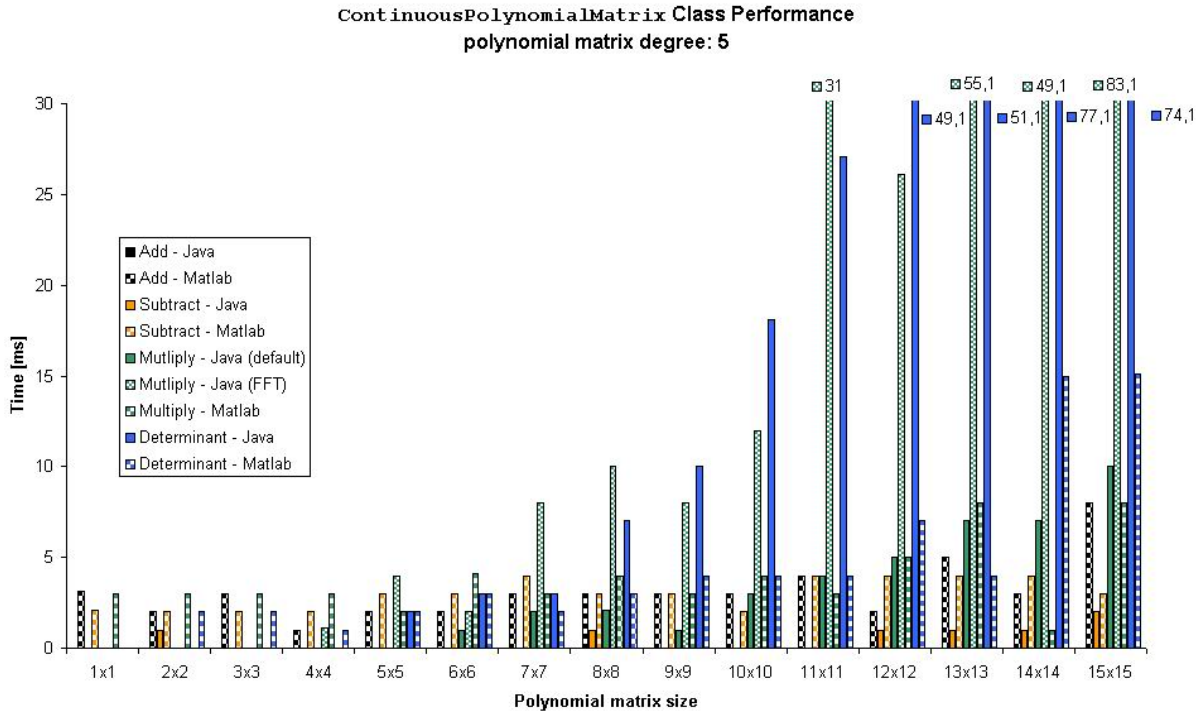

PolynomialMatrix Class Performance
polynomial matrix size: 5x5
Sylvester - Java
Sylvester - Matlab
Mutliply - Java
Mutliply - Matlab
Norm - Java
Norm - Matlab
Time [ms]
0
1
2
3
4
5
6
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Polynomial matrix degree

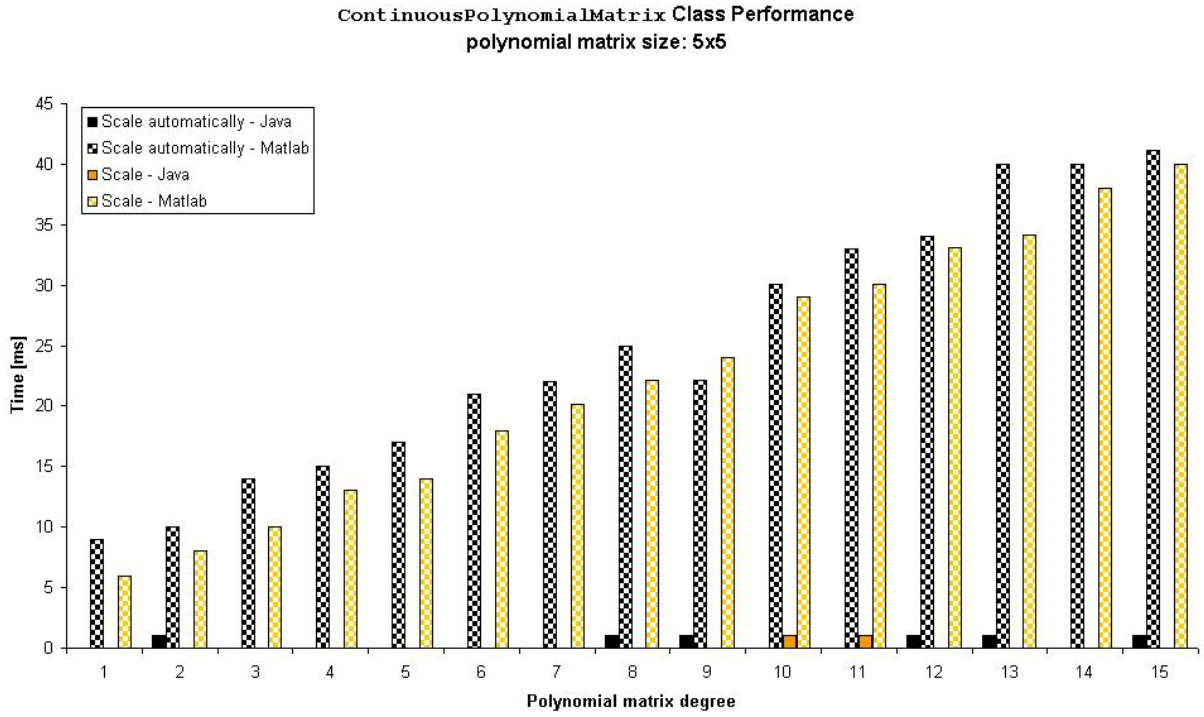

PolynomialMatrix Class Performance
polynomial matrix degree: 5
Sylvester - Java
Sylvester - Matlab
Mutliply - Java
Mutliply - Matlab
Norm - Java
Norm - Matlab
Time [ms]
0
1
2
3
4
5
6
7
1x1
2x2
3x3
4x4
5x5
6x6
7x7
8x8
9x9
10x10
11x11
12x12
13x13
14x14
15x15
Polynomial matrix size

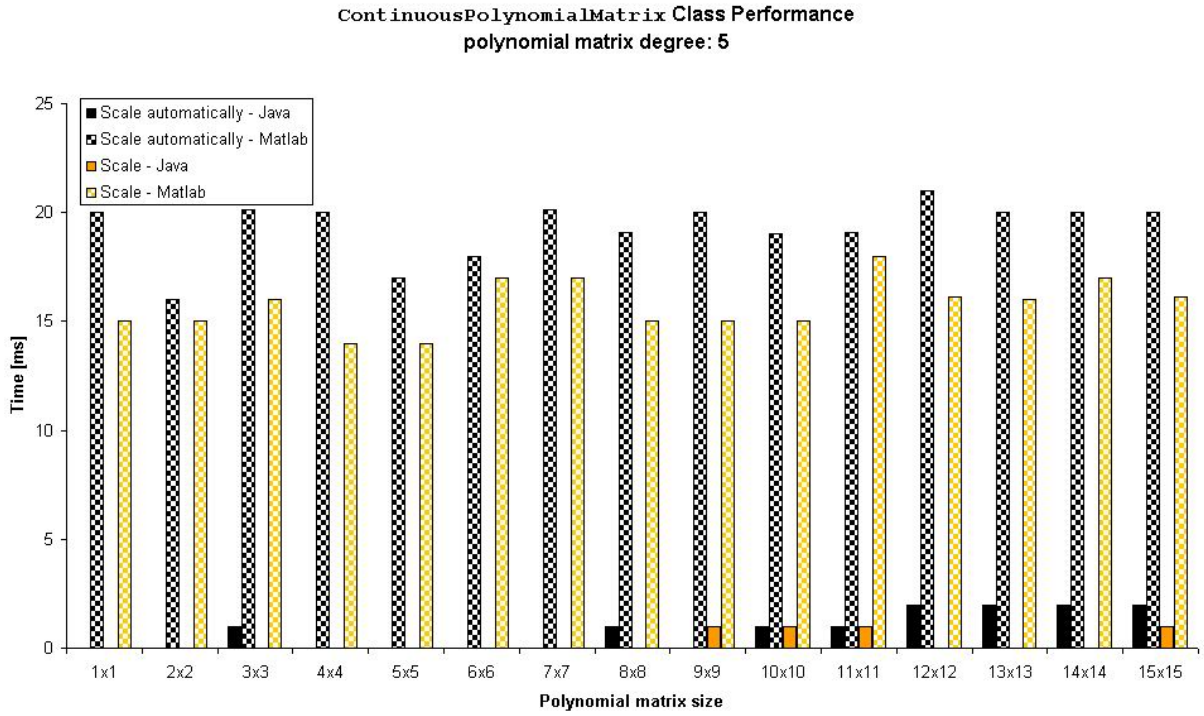

PolynomialMatrix Class Performance
polynomial matrix size: 5x5
Rank - Java
Rank - Matlab
Roots - Java
Roots - Matlab
Value, complex - Java
Value, complex - Matlab
Value, double - Java
Value, double - Matlab
50
52,1
Time [ms]
Polynomial matrix degree

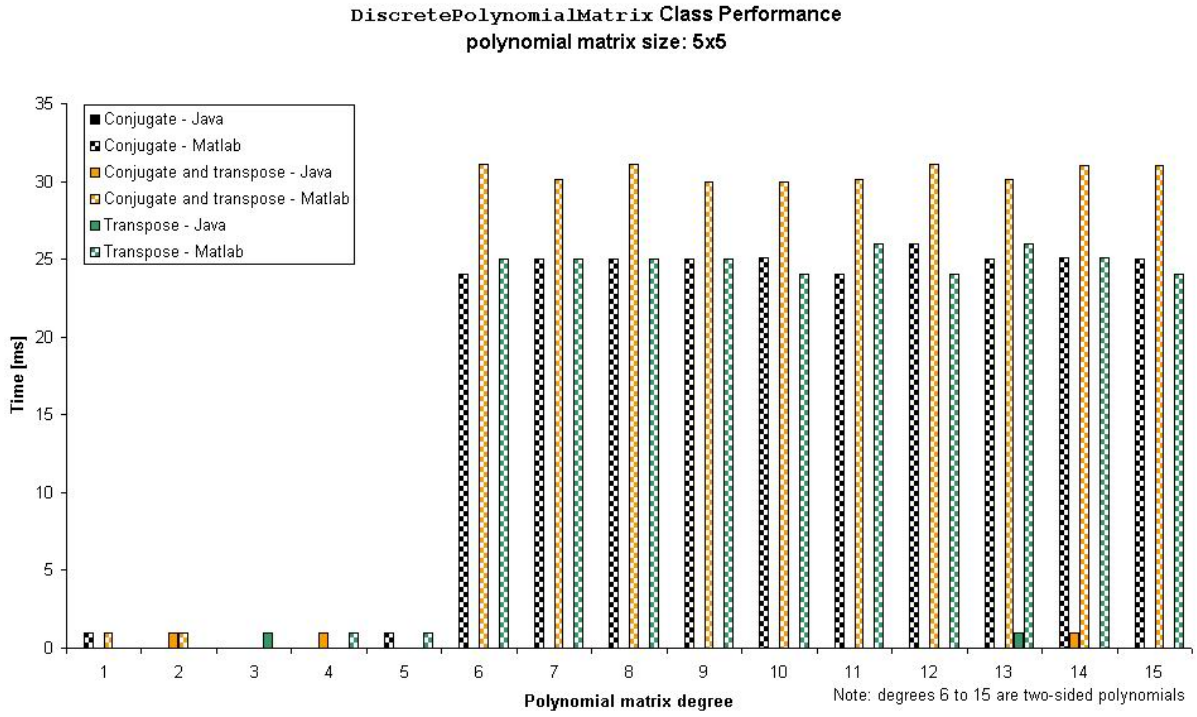

PolynomialMatrix Class Performance
polynomial matrix degree: 5
Rank - Java
Rank - Matlab
Roots - Java
Roots - Matlab
Value, complex - Java
Value, complex - Matlab
Value, double - Java
Value, double - Matlab
Time [ms]
1x1
2x2
3x3
4x4
5x5
6x6
7x7
8x8
9x9
10x10
11x11
12x12
13x13
14x14
15x15
Polynomial matrix size

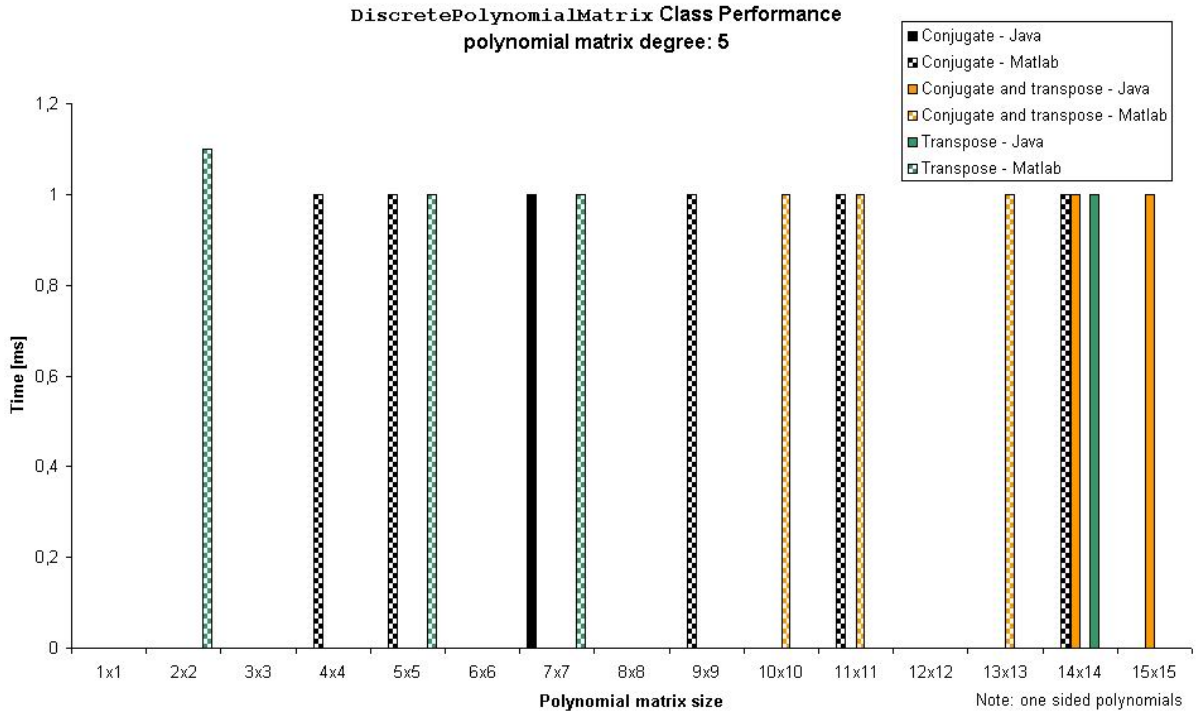

ContinuousPolynomialMatrix Class Performance
polynomial matrix size: 5x5
Add - Java
Add - Matlab
Subtract - Java
Subtract - Matlab
Mutliply - Java (default)
Mutliply - Java (FFT)
Multiply - Matlab
Determinant - Java
Determinant - Matlab
Time [ms]
Polynomial matrix degree

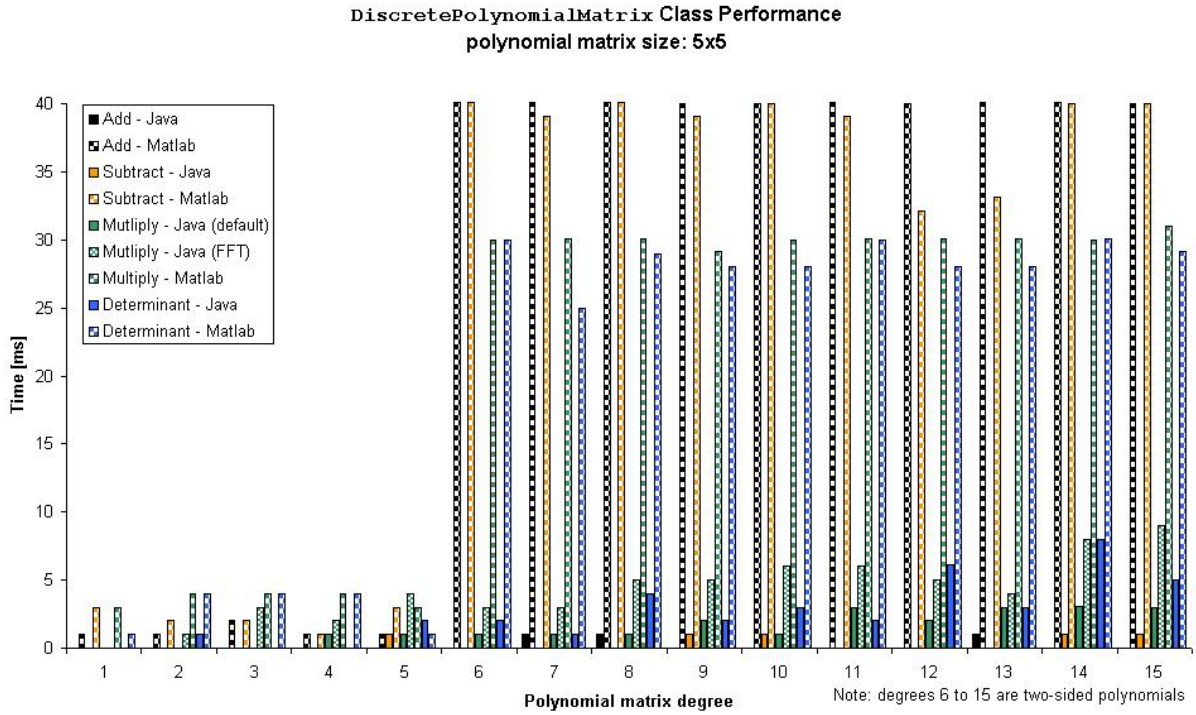

ContinuousPolynomialMatrix Class Performance
polynomial matrix degree: 5
31
55,1
49,1
83,1
49,1
51,1
77,1
74,1
Add - Java
Add - Matlab
Subtract - Java
Subtract - Matlab
Mutliply - Java (default)
Mutliply - Java (FFT)
Multiply - Matlab
Determinant - Java
Determinant - Matlab
Time [ms]
Polynomial matrix size

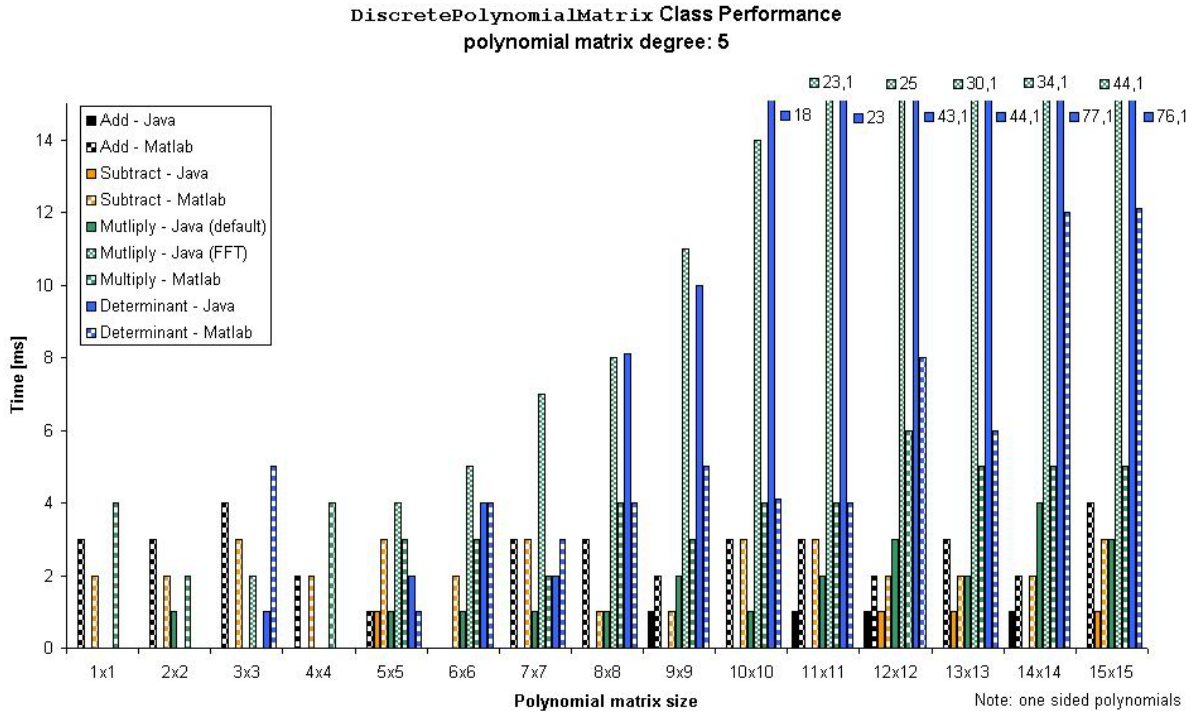

ContinuousPolynomialMatrix Class Performance
polynomial matrix size: 5x5
Scale automatically - Java
Scale automatically - Matlab
Scale - Java
Scale - Matlab
Time [ms]
0
5
10
15
20
25
30
35
40
45
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Polynomial matrix degree


ContinuousPolynomialMatrix Class Performance
polynomial matrix degree: 5
Scale automatically - Java
Scale automatically - Matlab
Scale - Java
Scale - Matlab
Time [ms]
0
5
10
15
20
25
1x1
2x2
3x3
4x4
5x5
6x6
7x7
8x8
9x9
10x10
11x11
12x12
13x13
14x14
15x15
Polynomial matrix size


DiscretePolynomialMatrix Class Performance
polynomial matrix size: 5x5
Conjugate - Java
Conjugate - Matlab
Conjugate and transpose - Java
Conjugate and transpose - Matlab
Transpose - Java
Transpose - Matlab
Time [ms]
0
5
10
15
20
25
30
35
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Polynomial matrix degree
Note: degrees 6 to 15 are two-sided polynomials


DiscretePolynomialMatrix Class Performance
polynomial matrix degree: 5
Conjugate - Java
Conjugate - Matlab
Conjugate and transpose - Java
Conjugate and transpose - Matlab
Transpose - Java
Transpose - Matlab
Time [ms]
0
0,2
0,4
0,6
0,8
1
1,2
1x1
2x2
3x3
4x4
5x5
6x6
7x7
8x8
9x9
10x10
11x11
12x12
13x13
14x14
15x15
Polynomial matrix size
Note: one sided polynomials


DiscretePolynomialMatrix Class Performance
polynomial matrix size: 5x5
Add - Java
Add - Matlab
Subtract - Java
Subtract - Matlab
Mutliply - Java (default)
Mutliply - Java (FFT)
Multiply - Matlab
Determinant - Java
Determinant - Matlab
Time [ms]
Polynomial matrix degree
Note: degrees 6 to 15 are two-sided polynomials


DiscretePolynomialMatrix Class Performance
polynomial matrix degree: 5
Add - Java
Add - Matlab
Subtract - Java
Subtract - Matlab
Mutliply - Java (default)
Mutliply - Java (FFT)
Multiply - Matlab
Determinant - Java
Determinant - Matlab
Time [ms]
Polynomial matrix size
Note: one sided polynomials

Appendix E

Project’s Home Page

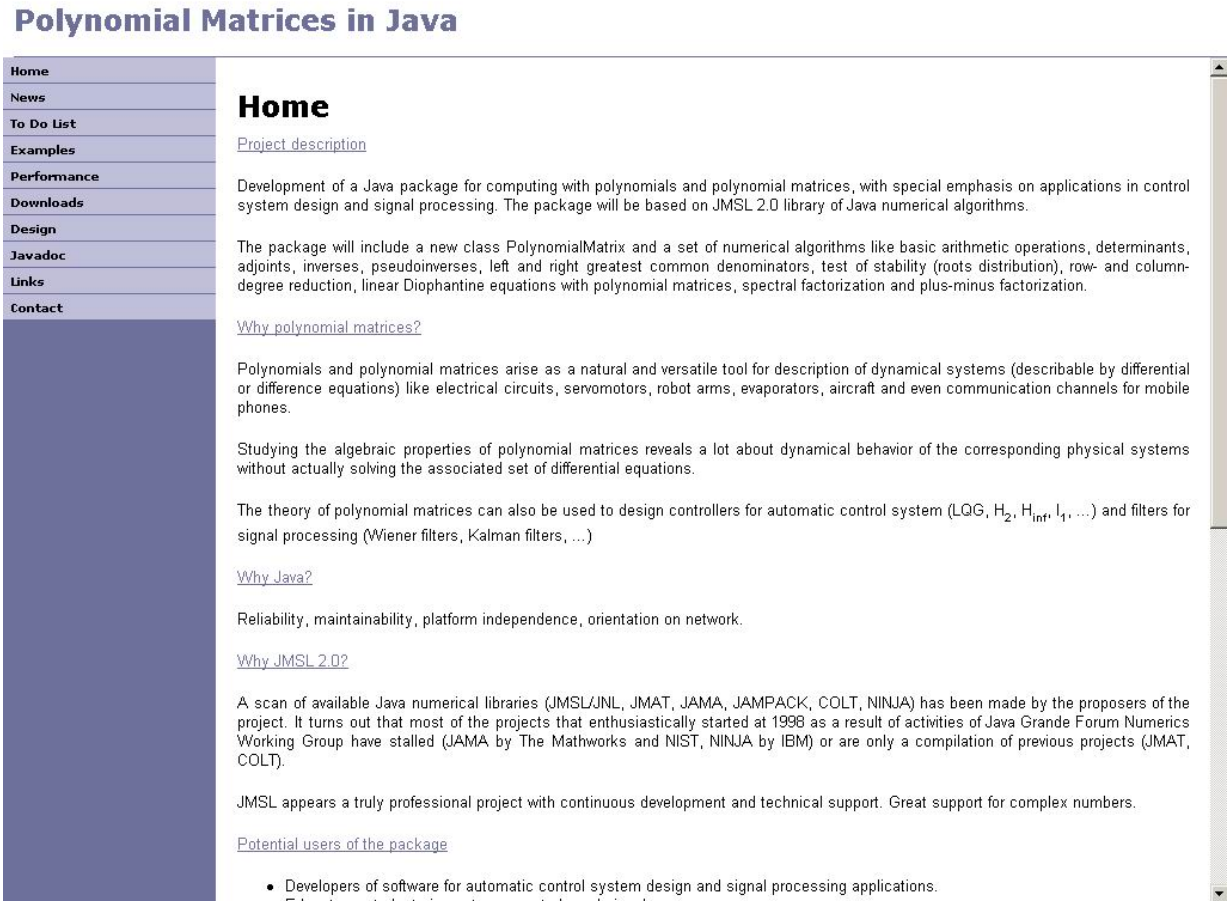
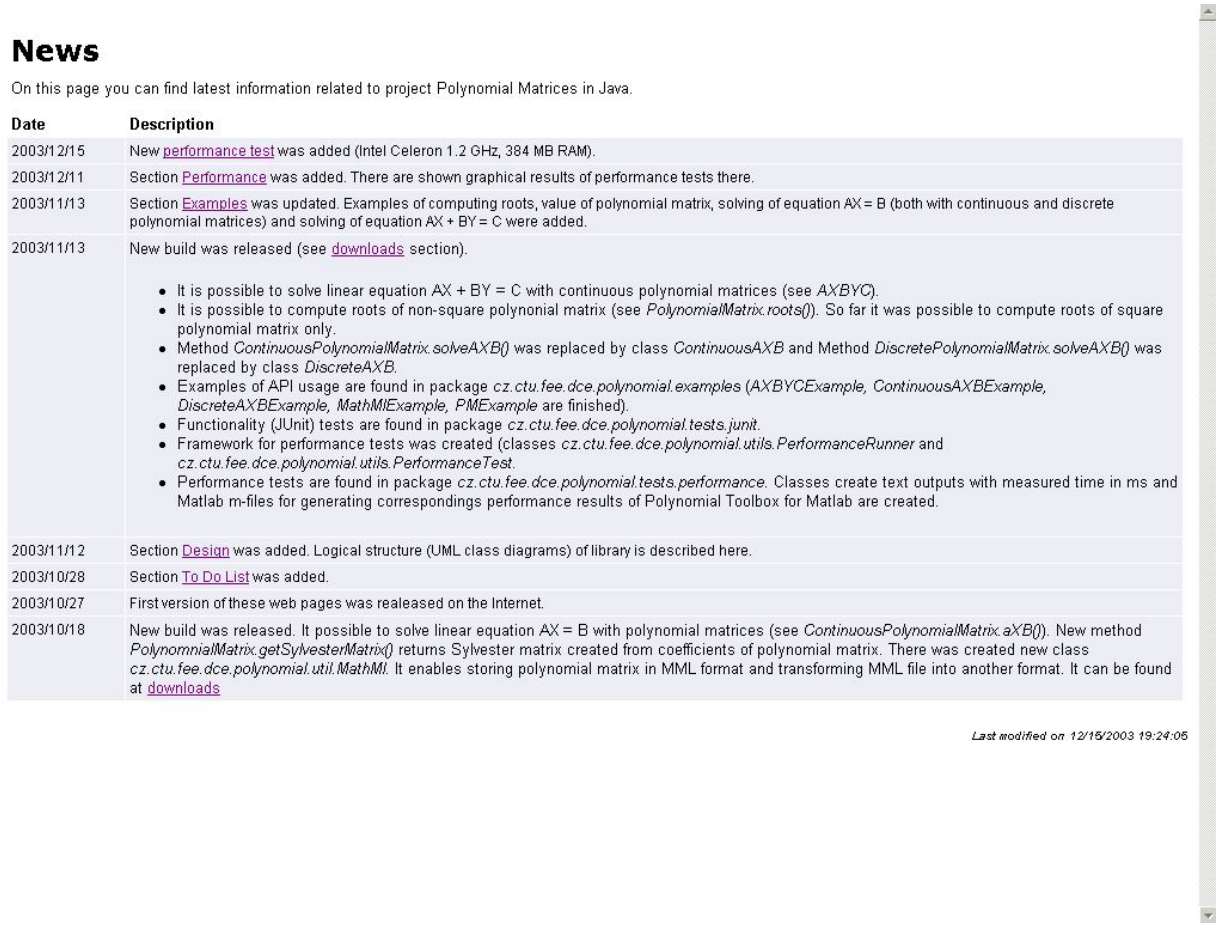


Figure E.1: Home page



News

On this page you can find latest information related to project Polynomial Matrices in Java.

Date	Description
2003/12/15	New performance test was added (Intel Celeron 1.2 GHz, 384 MB RAM).
2003/12/11	Section Performance was added. There are shown graphical results of performance tests there.
2003/11/13	Section Examples was updated. Examples of computing roots, value of polynomial matrix, solving of equation AX = B (both with continuous and discrete polynomial matrices) and solving of equation AX + BY = C were added.
2003/11/13	New build was released (see downloads section). • It is possible to solve linear equation AX + BY = C with continuous polynomial matrices (see *AXBYC*). • It is possible to compute roots of non-square polynonial matrix (see *PolynomialMatrix.roots()*). So far it was possible to compute roots of square polynomial matrix only. • Method *ContinuousPolynomialMatrix.solveAXB()* was replaced by class *ContinuousAXB* and Method *DiscretePolynomialMatrix.solveAXB()* was replaced by class *DiscreteAXB*. • Examples of API usage are found in package *cz.ctu.fee.dce.polynomial.examples* (*AXBYCExample, ContinuousAXBExample, DiscreteAXBExample, MathMlExample, PMExample* are finished). • Functionality (JUnit) tests are found in package *cz.ctu.fee.dce.polynomial.tests.junit*. • Framework for performance tests was created (classes *cz.ctu.fee.dce.polynomial.utils.PerformanceRunner* and *cz.ctu.fee.dce.polynomial.utils.PerformanceTest*. • Performance tests are found in package *cz.ctu.fee.dce.polynomial.tests.performance*. Classes create text outputs with measured time in ms and Matlab m-files for generating correspondings performance results of Polynomial Toolbox for Matlab are created.
2003/11/12	Section Design was added. Logical structure (UML class diagrams) of library is described here.
2003/10/28	Section To Do List was added.
2003/10/27	First version of these web pages was realeased on the Internet.
2003/10/18	New build was released. It possible to solve linear equation AX = B with polynomial matrices (see *ContinuousPolynomialMatrix.aXB()*). New method *PolynomnialMatrix.getSylvesterMatrix()* returns Sylvester matrix created from coefficients of polynomial matrix. There was created new class *cz.ctu.fee.dce.polynomial.util.MathMl*. It enables storing polynomial matrix in MML format and transforming MML file into another format. It can be found at downloads

Last modified on 12/15/2003 19:24:05

Figure E.2: News page

<< back

Let's have linear equation with polynomial matrices **A**(s)**X**(s) + **B**(s)**Y**(s)= **C**(s), where

$\mathbf{A}(s) = \begin{pmatrix} 1 & 0 \\ s & 1 \end{pmatrix}$, $\mathbf{B}(s) = \begin{pmatrix} 1 & 0 \\ s & 1 \end{pmatrix}$, $\mathbf{C}(s) = \begin{pmatrix} 2s \\ 2 \end{pmatrix}$ and **X**(s), **Y**(s) are searched solutions of this equation.

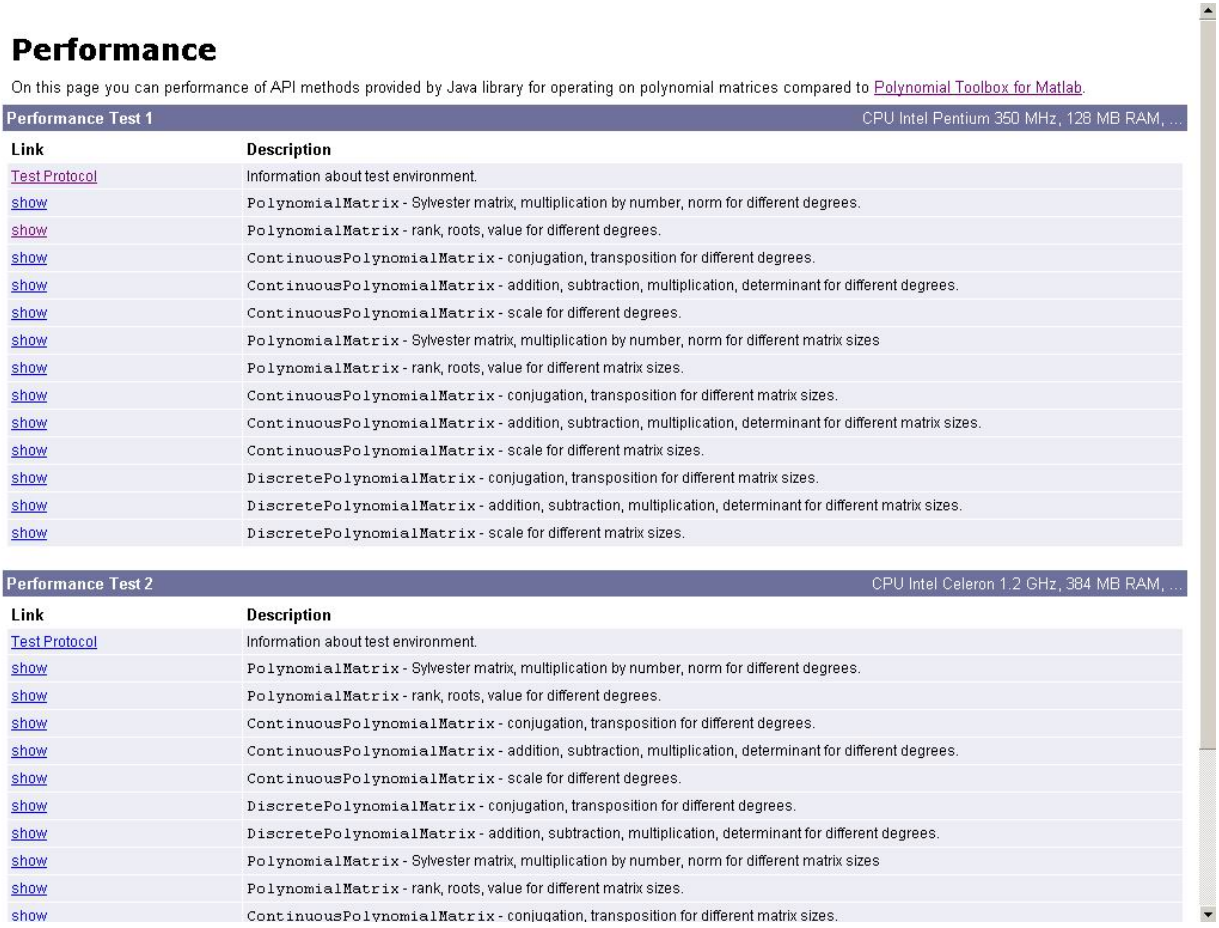
With usage of polynomial.jar library we find solutions of linear equation:

```
// coefficients of polynomial matrix A
double[][][] aCoef = {
    {{1, 0}, {0, 1}}, // coefficient at s^0
    {{0, 0}, {1, 0}}  // coefficient at s^1
};

// coefficients of polynomial matrix B
double[][][] bCoef = {
    {{1, 0}, {0, 1}}, // coefficient at s^0
    {{0, 0}, {1, 0}}  // coefficient at s^1
};

// coefficients of polynomial matrix C
double[][][] cCoef = {
    {{0}, {2}}, // coefficient at s^0
    {{2}, {0}}  // coefficient at s^1
};

// polynomial matrix A
ContinuousPolynomialMatrix cpmA = new ContinuousPolynomialMatrix(aCoef);

// polynomial matrix B
ContinuousPolynomialMatrix cpmB = new ContinuousPolynomialMatrix(bCoef);

// polynomial matrix C
ContinuousPolynomialMatrix cpmC = new ContinuousPolynomialMatrix(cCoef);

// instace of A(s)X(s) + B(s)Y(s) = C(s) equation solver is created
// solution X(s) and Y(s) is found
AXBYC axbyc = new AXBYC(cpmA, cpmA, cpmB);

// soulution X(s)
ContinuousPolynomialMatrix cpmX = axbyc.getX();

// soulution Y(s)
ContinuousPolynomialMatrix cpmY = axbyc.getY();
```

There were found solutions $\mathbf{X}(s) = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}$ and $\mathbf{Y}(s) = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$.

Last modified on 01/07/2004 12:34:13

Figure E.3: Code example



Performance

On this page you can performance of API methods provided by Java library for operating on polynomial matrices compared to Polynomial Toolbox for Matlab.

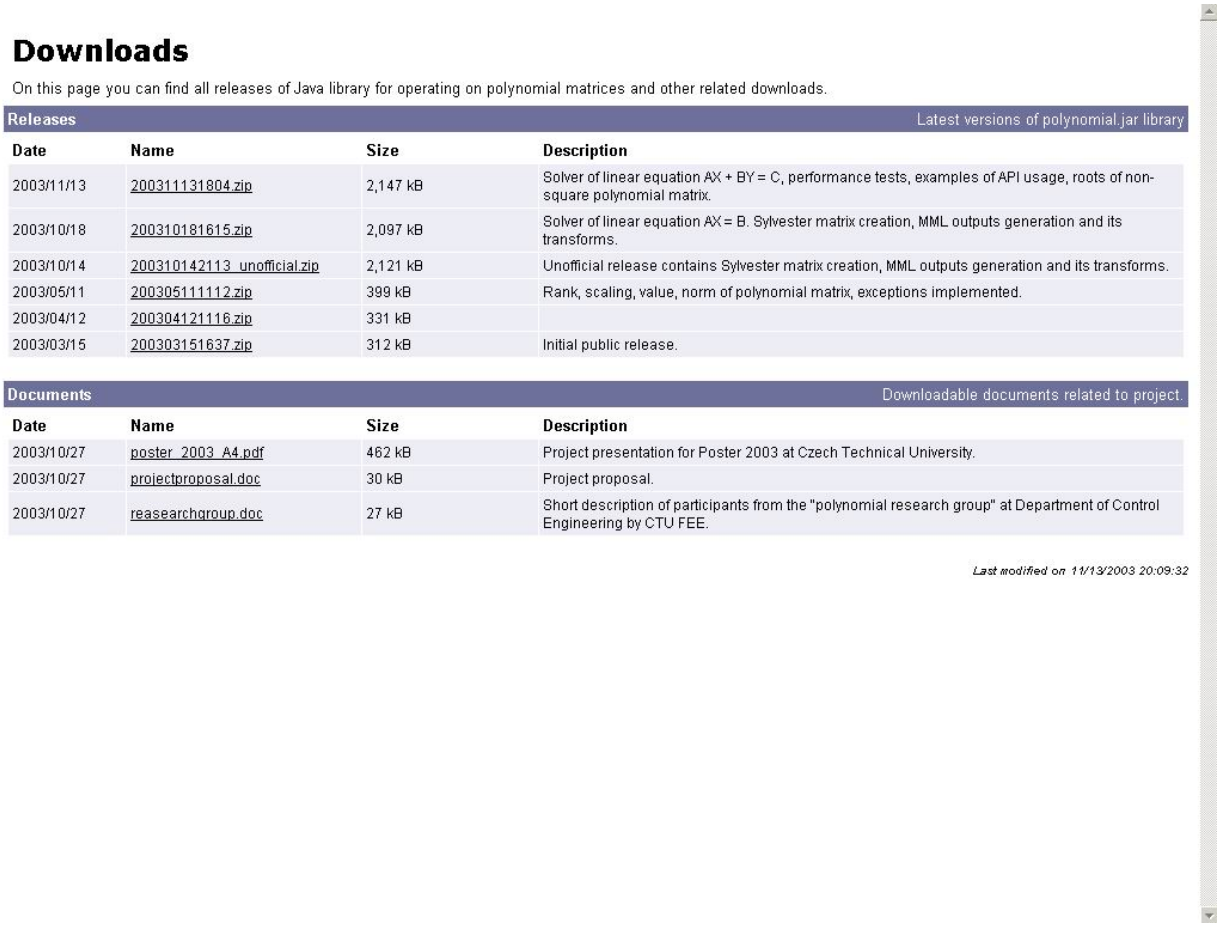
Performance Test 1 — CPU Intel Pentium 350 MHz, 128 MB RAM, ...

Link	Description
Test Protocol	Information about test environment.
show	`PolynomialMatrix` - Sylvester matrix, multiplication by number, norm for different degrees.
show	`PolynomialMatrix` - rank, roots, value for different degrees.
show	`ContinuousPolynomialMatrix` - conjugation, transposition for different degrees.
show	`ContinuousPolynomialMatrix` - addition, subtraction, multiplication, determinant for different degrees.
show	`ContinuousPolynomialMatrix` - scale for different degrees.
show	`PolynomialMatrix` - Sylvester matrix, multiplication by number, norm for different matrix sizes
show	`PolynomialMatrix` - rank, roots, value for different matrix sizes.
show	`ContinuousPolynomialMatrix` - conjugation, transposition for different matrix sizes.
show	`ContinuousPolynomialMatrix` - addition, subtraction, multiplication, determinant for different matrix sizes.
show	`ContinuousPolynomialMatrix` - scale for different matrix sizes.
show	`DiscretePolynomialMatrix` - conjugation, transposition for different matrix sizes.
show	`DiscretePolynomialMatrix` - addition, subtraction, multiplication, determinant for different matrix sizes.
show	`DiscretePolynomialMatrix` - scale for different matrix sizes.

Performance Test 2 — CPU Intel Celeron 1.2 GHz, 384 MB RAM, ...

Link	Description
Test Protocol	Information about test environment.
show	`PolynomialMatrix` - Sylvester matrix, multiplication by number, norm for different degrees.
show	`PolynomialMatrix` - rank, roots, value for different degrees.
show	`ContinuousPolynomialMatrix` - conjugation, transposition for different degrees.
show	`ContinuousPolynomialMatrix` - addition, subtraction, multiplication, determinant for different degrees.
show	`ContinuousPolynomialMatrix` - scale for different degrees.
show	`DiscretePolynomialMatrix` - conjugation, transposition for different degrees.
show	`DiscretePolynomialMatrix` - addition, subtraction, multiplication, determinant for different degrees.
show	`PolynomialMatrix` - Sylvester matrix, multiplication by number, norm for different matrix sizes
show	`PolynomialMatrix` - rank, roots, value for different matrix sizes.
show	`ContinuousPolynomialMatrix` - conjugation, transposition for different matrix sizes.

Figure E.4: Performance tests



Downloads

On this page you can find all releases of Java library for operating on polynomial matrices and other related downloads.

Releases — Latest versions of polynomial.jar library

Date	Name	Size	Description
2003/11/13	200311131804.zip	2,147 kB	Solver of linear equation AX + BY = C, performance tests, examples of API usage, roots of non-square polynomial matrix.
2003/10/18	200310181615.zip	2,097 kB	Solver of linear equation AX = B. Sylvester matrix creation, MML outputs generation and its transforms.
2003/10/14	200310142113_unofficial.zip	2,121 kB	Unofficial release contains Sylvester matrix creation, MML outputs generation and its transforms.
2003/05/11	200305111112.zip	399 kB	Rank, scaling, value, norm of polynomial matrix, exceptions implemented.
2003/04/12	200304121116.zip	331 kB	
2003/03/15	200303151637.zip	312 kB	Initial public release.

Documents — Downloadable documents related to project.

Date	Name	Size	Description
2003/10/27	poster_2003_A4.pdf	462 kB	Project presentation for Poster 2003 at Czech Technical University.
2003/10/27	projectproposal.doc	30 kB	Project proposal.
2003/10/27	reasearchgroup.doc	27 kB	Short description of participants from the "polynomial research group" at Department of Control Engineering by CTU FEE.

Last modified on 11/13/2003 20:09:32

Figure E.5: Downloads

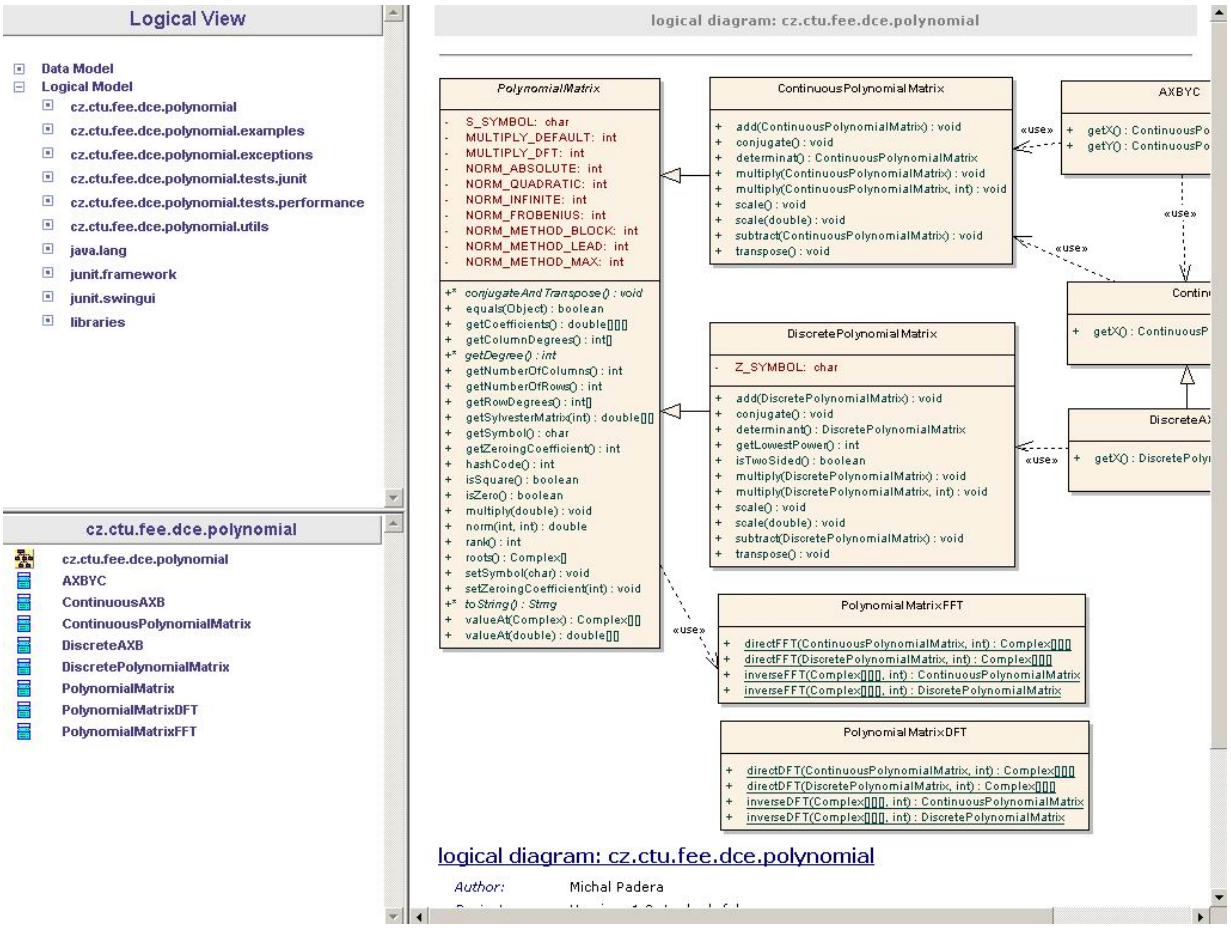


Figure E.6: Design

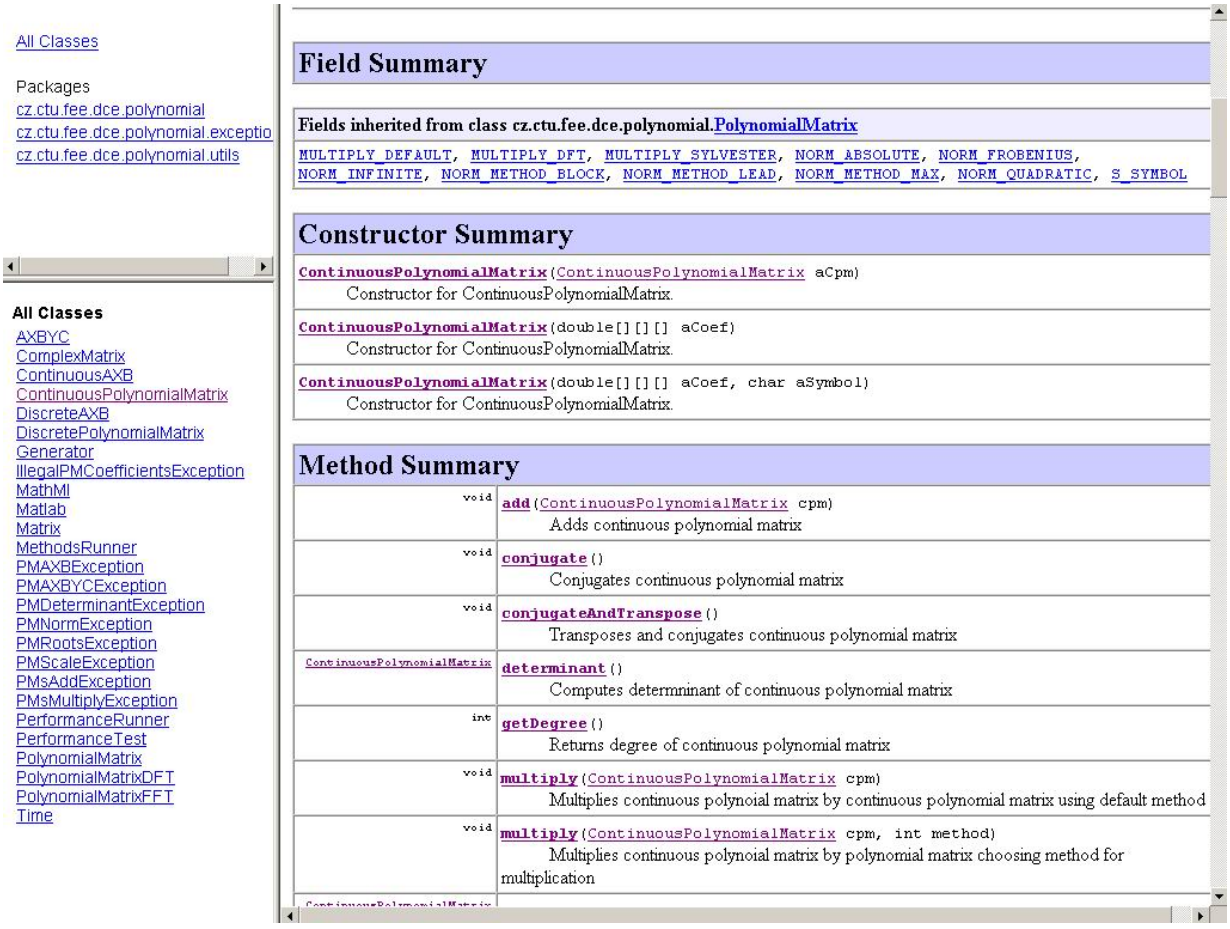


Figure E.7: Java documentation

Links

On this page you can find links to sites concerning to project of Polynomial Matrices in Java.

General — General links

Link	Description
http://www.vni.com	Visual Numerics
http://www.polyx.com	Polyx - producer of the Polynomial Toolbox for Matlab
http://dce.felk.cvut.cz	Department of Control Engineering by Czech Technical University Faculty of Electrical Engineering

Java — Links concerned to development in Java

Link	Description
http://java.sun.com	Sun Java
http://www.vni.com/products/imsl/jmsl.html	JMSL Numerical Library for Java Applications by Visual Numerics
http://www.eclipse.org	Eclipse - Java IDE
http://www.junit.org	JUnit - library used for functional testing
http://ant.apache.org	Ant - tool for creating builds

Java and Numerics — Links related to Java, numerics and linear algebra.

Link	Description
http://math.nist.gov/javanumerics	Java Grande Forum Numerics Working Group
http://java.sun.com/people/jag/FP.html	The Evolution of Numerical Computing in Java
http://jmat.sourceforge.net	Java MATrix tools package
http://hoschek.home.cern.ch/hoschek/colt	Open Source Libraries for High Performance Scientific and Technical Computing in Java

Algorithms — Linear algebra algorithms.

Link	Description
http://www.nr.com	Numerical Recipes
http://www.netlib.org/blas	BLAS (Basic Linear Algebra Subprograms)
http://www.laas.fr/~henrion/publis.html	Didier Henrion's publications, linear equations, rank, Sylvester matrix, norms
http://mathworld.wolfram.com/HornersMethod.html	Horner scheme

Other — Other unfolded links.

Link	Description
http://www.w3.org/Math/	W3C Math Home
http://www.dessci.com/en/products/mathplayer	MathPlayer is used for displaying MathML equations on web pages

Figure E.8: Links

Vita

Michal Paděra was born on July 1979 in Jilemnice, Czech Republic. After graduating at grammar school in Jicin in 1997 he was accepted at Faculty of Electrical Engineering by Czech Technical University in Prague with major field in control engineering. In 2000 he passed The Cambridge First Certificate in English exam. Since August 2001 he has been part-time developer at Unicorn in Prague, Czech Republic. In 2001 he participated in Erasmus programme by European Union at Department of Applied Information Technology and Multimedia by Technological Educational Institute of Crete in Greece. From November 2001 till February 2002 he attended English lessons at Access Language Centre in Sydney, Australia.