

MPCMath User Manual

Jørgen K. H. Knudsen

March 26, 2015

Contents

1	Introduction	7
1.1	Vectors	8
1.2	Matrices	8
1.3	Block vectors and matrices	11
1.4	Complex numbers and Vectors	12
1.5	Transfer Functions	13
1.6	ARX Models	15
1.7	State Space Models	18
1.8	Model Predictive Control	23
1.9	Intel Math Kernel Library (BLAS and LAPACK)	28
2	Console functions	31
2.1	Creating a new <i>MPCMath</i> application	31
2.2	Displaying and plotting variables	32
2.3	Using Console for test of GUI programs	37
2.4	MPCMathConsole	39
2.5	PlotSeries	46
2.6	Plot	49
2.7	MPCMathWindow.xaml	50
3	Vectors and Matrices	53
3.1	Sparse Matrices	54
3.2	Vector	54
3.3	MatrixForm	64
3.4	Matrix	65
3.5	CVector	76
4	Block Vectors and Matrices	81
4.1	Structure	83
4.2	BVector	85
4.3	BMatrix	91
5	Generic Vectors and Matrices	99
5.1	ICommon	100
5.2	TVector	103
5.3	TMatrix	107

6	Transfer Functions	115
6.1	TransferFunction	117
7	State Space Models	125
7.1	StateSpaceModel	126
8	Equation solvers	133
8.1	ISolver	133
8.2	LinearEquationSolver	134
8.3	LeastSquareEquationSolver	136
8.4	CholeskyEquationSolver	137
8.5	SymmetricEquationSolver	138
8.6	SymmetricalBlockEquationSolver	139
8.7	SparseEquationSolver	139
8.8	Banded matrix storage	141
8.9	BandEquationSolver	142
8.10	SymmetricBandEquationSolver	143
9	Linear Filter	145
9.1	LinearFilter	148
10	ARX models	157
10.1	ARXModel	157
11	Function and Model interfaces	163
11.1	IFun	163
11.2	IConFun	164
11.3	IFunt	166
11.4	IModel	167
11.5	IFeasible	170
11.6	Infeasible State exceptions	170
12	Unconstrained and constrained minimization	173
12.1	NewtonMethod	173
12.2	Extended linear-Quadratic Optimal Control problem	175
12.3	Function Minimization	178
12.4	FunMin	183
12.5	ConFunMin	186
12.6	Least Square Fitting	190
12.7	LeastSquareFit	194
12.8	RiccatiSolver	196
13	QP and LP solvers	199
13.1	Quadratic Optimization	202
13.2	Matrix Factorization	204
13.3	KKT solvers	204
13.4	QPSolver	205
13.5	IKKTSolver	207
13.6	KKTSolver	209
13.7	LPKKTSolver	209

14 Model Predictive Control, MPC	211
14.1 MiMoMPC	214
14.2 LinearModel	219
14.3 Four Tank Process	219
14.4 FourTankProcess	220
14.5 VanDerPol	221
15 ODE solvers	223
15.1 Introduction	223
15.2 ODEMethods	225
15.3 ODESolver	226
16 MPCMathLib	231
16.1 MPCMathLib-MinNorm	231
16.2 MPCMathLib-RK4	233
16.3 MPCMathLib-Discretize	234
16.4 MPCMathLib-SteadyState	235
16.5 MPCMathLib-Util	236
17 Miscellaneous functions	243
17.1 Complex	243
17.2 Polynomial	248
17.3 DelayChain	255
17.4 HuberFunction	257
17.5 Spline and Cubic Smoothing Spline functions	258
17.6 Spline	258
17.7 CubicSpline	260
17.8 BarrierFunction	261
17.9 Reports	262
17.10ReportBuilder	263
17.11File I/O and serialization	264
17.12BinaryIO	265
17.13XmlIO	266
17.14MPCMathLib-JacobianEstimation	266
A Installing <i>MPCMath</i>	271
A.1 Prerequisites	271
A.2 Installing <i>MPCMath</i>	271
A.3 Windows x86 and x64 platforms	272
A.4 Trouble shooting, Debugging and Exceptions	272
A.5 MPCMathTemplate	273
A.6 TestMiMoMPC	274
A.7 UnitTestMPCMath	276
A.8 MPCMathConsoleTemplate	276
B Debugging tool for Console applications	277
B.1 Cnsl	277

C	Base classes for Vectors and Matrices	285
C.1	VBase	285
C.2	MBase	286
D	Implementation of test fun using <i>IConFun</i>	291
E	Implementaion of Van der Pol equations using <i>IModel</i>	295
F	KKTSolver code	299
G	PARDISO sparse matrix storage format	307
H	<i>MPCMath</i> change history	309

Chapter 1

Introduction

MPCMath is a mathematical library for real time implementation of process identification, model predictive control and other optimization tasks

Introduction to *MPCMath*

MPCMath is a library of C#/.NET routines designed for real time implementation of Process Identification packages, Model Predictive Control and Optimizations tasks. The foundation of *MPCMath* is a comprehensive implementation of Matrix and Vector calculus.

Development and test of mathematical algorithms are often based on interactive environments as Matlab or Python providing a high level mathematical language and efficient debugging facilities with high productivity. These packages are not suited for on-line applications running continuously controlling chemical plants or embedded systems where the controllers are implemented on small processors. During the recent years Microsoft C# language combined with the very comprehensive .NET libraries, has been a very popular platform for development of client and server solutions. *MPCMath* provides a set of tools enabling the programmers to implement process control task, MPC controllers and optimization tasks on the .NET platform providing intelligent objects and functionality enabling high productivity and quality solutions.

Many of *MPCMath*'s functions are based on Intel Math Kernel Library Intel a highly optimized implementation of the public domain computing packages BLAS (Basic Linear Algebra Subroutines) and LAPCK (Linear Algebra Package). This gives *MPCMath* a boost in performance resulting more than an order of magnitude increased computational speed.

MPCMath are aimed at organizations and companies developing equipment for process control, having programmers mastering C# and object oriented programming. *MPCMath* is callable from any .NET language as C# , Visual-Basic.NET and F#.

This chapter gives a walk through how to program tasks related to MPC control in *MPCMath* , starting with an introduction to the basic Vector and Matrix objects.

1.1 Vectors

Vector functionality is implemented in the Vector class. A simple example:

```
console.WriteLine("\MPCMath Introduction");
console.WriteLine();

// Vector definition
Vector x = new Vector(5, Math.PI);
Vector y = Vector.Random(5, -100.0, 1000.0);
Vector z;
z = x + y;
Vector.Show("x", x);
Vector.Show("y", y);
Vector.Show("z", z);
```

The console.WriteLine writes a text string on the *MPCMath* console. The next statements defines a vector x with 5 elements all having the value $\pi = 3.141$, a vector y with 5 elements having random values between -100.0 and 100.0. The fourth statement adds vector x and y storing the result in Vector z. The Vector.Show statements displays the three vectors in a *MPCMath* console:

```
x Vector
   3,1416   3,1416   3,1416   3,1416   3,1416
y Vector
  732,8023  51,3077  691,7185  895,7004  672,5596
z Vector
  735,9439  54,4493  694,8601  898,8420  675,7012
```

Vector functionality includes Add, Subtract, Multiply, Clone, Negate, Equal, Normalize, OuterProduct, Sort, Sum and other functions. Vectors support the operators +, -, * as shown in the example above. Many functions are also implemented as methods

```
z.Add(x);
```

where the vector z is overwritten with $z + x$

1.2 Matrices

Matrices are implemented using the Matrix class:

```
// Matrix definition
Matrix A = new Matrix(MatrixForm.General, 3, 5);
Matrix B = Matrix.Random(3, 5, -100.0, 100.0);
Matrix C;

A[2,3] = 25.0;
C = A + B;
Matrix.Show("A", A);
Matrix.Show("B", B);
Matrix.Show("C", C);
```

These statements defines matrix A with 3 rows and 5 columns. Matrix B is defined as a 3 rows 5 columns matrix with random elements in the range -100.0 to 100.0. Element [2,3] of A is set to 25.0, and matrix C is the sum of matrix A and B. The Matrix.Show calls displays the matrices:

```
A Matrix.Form General[3,5]
    0      0      0      0      0
    0      0      0      0      0
    0      0      0  25,0000      0

B Matrix.Form General[3,5]
-87,4640 -74,9052  50,3092 -14,7262 -73,6282
-82,9797  27,8304   5,4292  -4,7578  61,5226
  7,0211  19,3501  98,1663  27,1058  58,8416

C Matrix.Form General[3,5]
-87,4640 -74,9052  50,3092 -14,7262 -73,6282
-82,9797  27,8304   5,4292  -4,7578  61,5226
  7,0211  19,3501  98,1663  52,1058  58,8416
```

The matrix standard functionality includes Add, Sub, Mul, Transpose, Mul-Transposed, Clone, Negate, Invert, Eigenvalues, SVD decompositions. As for Vectors the Matrix objects support +, -, * operators

```
int dim = 5;
x = Vector.Random(dim, -50.0, 50.0);
A = Matrix.Random(dim, dim, -50.0, 50.0);
B = Matrix.Random(dim, dim, -50.0, 50.0);

y = (A + B) * x;

Vector.Show("x", x);
Matrix.Show("A", A);
Matrix.Show("B", B);
Vector.Show("y", y);
```

with the following output to the *MPCMath* console

```
x Vector
  -3,3038  -47,6437   27,9289    6,2165   -8,0053
A Matrix.Form General[5,5]
  13,7145  -22,0332   24,2230   29,2905   43,0788
 -45,0651  -43,9516   46,3690    6,4084   -3,0715
 -33,1575    5,9085  -21,6215  -43,2811   22,5939
  44,6817   36,3091  -38,1826  -22,8073   11,0903
  37,0345   -5,7952   27,6559   32,9791    2,6289

B Matrix.Form General[5,5]
 -40,3434   23,3530  -48,6669    3,6267    1,2521
  43,3688  -14,1737   41,6677  -30,7539  -14,8256
  20,0622  -34,2421    8,8289   16,4055  -48,6315
  30,4909   -1,4764  -24,4871    4,8784  -46,8946
  28,3770  -43,4818  -14,0516   18,4139  -49,8697

y Vector
-807,8519 5225,6046 1077,2666 -3483,0406 3209,2488
```

MPCMath implements the following more advanced functionalities:

- LU decomposition
- QR decomposition
- Eigenvalue decomposition giving eigenvalues and Eigenvectors
- Linear equation solvers based on LU decompositions and Cholesky decompositions.
- And other functions ...

MPCMath keeps track of the form of the Matrix object as

- Null,
- ConstantDiagonal,
- Diagonal,
- General

enabling *MPCMath* to exploit matrix forms and an automatic speeding up of calculations.

1.3 Block vectors and matrices

Block vectors and matrices stores vector and matrices in their cells, enabling a high level programming style. Block vectors and matrices are implemented using the BVector and BMatrix classes.

The block matrix below, were A and I are submatrices

$$\mathbf{AS} = \begin{pmatrix} A & I & 0 & 0 & 0 \\ 0 & A & I & 0 & 0 \\ 0 & 0 & A & I & 0 \\ 0 & 0 & 0 & A & I \\ 0 & 0 & 0 & 0 & A \end{pmatrix}$$

is implemented with the code

```
dim = 5;
int n = 3;
A = Matrix.Random(n, n, -100.0, 100.0);
Matrix I = Matrix.UnityMatrix(n);
BMatrix AS = new BMatrix(MatrixForm.General, dim);
for (int pos = 0; pos < dim; pos++)
{
    AS[pos, pos] = A;
    if (pos < dim - 1)
    {
        AS[pos, pos + 1] = I;
    }
}

Matrix.Show("A", A);
BMatrix.Show("AS", AS);
```

```
A Matrix.Form General[3,3]
6,8836 -50,9906 -9,3580
8,8937 -5,6039 -33,8479
-29,8827 18,3255 -89,7406

AS Matrix.Form General[15,15]
6,8836 -50,9906 -9,3580 0 0 0 0 0 0 0 0 0 0 0 0
8,8937 -5,6039 -33,8479 0 0 0 0 0 0 0 0 0 0 0 0
-29,8827 18,3255 -89,7406 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 6,8836 -50,9906 -9,3580 1 0 0 0 0 0 0 0 0
0 0 0 8,8937 -5,6039 -33,8479 0 1 0 0 0 0 0 0 0
0 0 0 -29,8827 18,3255 -89,7406 0 0 0 1 0 0 0 0 0
0 0 0 0 0 0 6,8836 -50,9906 -9,3580 1 0 0 0 0
0 0 0 0 0 0 8,8937 -5,6039 -33,8479 0 1 0 0 0
0 0 0 0 0 0 -29,8827 18,3255 -89,7406 0 0 1 0 0
0 0 0 0 0 0 0 0 0 6,8836 -50,9906 -9,3580 1 0 0
0 0 0 0 0 0 0 0 0 8,8937 -5,6039 -33,8479 0 1 0
0 0 0 0 0 0 0 0 0 -29,8827 18,3255 -89,7406 0 0 1
0 0 0 0 0 0 0 0 0 0 0 6,8836 -50,9906 -9,3580
0 0 0 0 0 0 0 0 0 0 0 8,8937 -5,6039 -33,8479
0 0 0 0 0 0 0 0 0 0 0 -29,8827 18,3255 -89,7406
```

The Matrix object keeps track of the form of its submatrices, as shown by the statement:

```
BMatrix.ShowStructure("AS Structure", AS);
```

with the following output:

AS Structure Matrix.Form General				
General matrix[5][5]				
General	ConstantDiagonal	Null	Null	Null
Null	General	ConstantDiagonal	Null	Null
Null	Null	General	ConstantDiagonal	Null
Null	Null	Null	General	ConstantDiagonal
Null	Null	Null	Null	General

that AS is a structured 5x5 matrix where the diagonal A matrices are general matrices, the superdiagonal I matrices are diagonal with a constant value and all the other matrices are null matrices. The Matrix/Vector routines exploits this structure information to speed up computations.

Block Matrices and vectors support the operators $+$, $-$, $*$, clone etc ...

1.4 Complex numbers and Vectors

MPCMath implement complex numbers using the complex class

```
complex a = new complex(1.0, 3.0);
complex b = complex.Exp(a);
complex c = a + b;
console.Show("a", a);
console.Show("b", b);
console.Show("c", c);

complex i = complex.Sqrt(-1.0);
console.WriteLine();
console.Show("i", i);
```

with the output:

```
a  1,0000 + i* 3,0000
b  -2,6911 + i* 0,3836
c  -1,6911 + i* 3,3836

i  0,0000 + i* 1,0000
```

Complex objects supports Add, Sub, Mul, Div, RV(real value), IV (imaginary value), Mod (Modulus), Arg (Argument) and the operators $+$, $-$, $*$ and $/$

Complex vectors are implemented using the CVector class. Presently CVector implements a limited set of functionalities, which will be increased when needed. *MPCMath* does not support Complex Matrices presently.

1.5 Transfer Functions

Transfer function used in linear control applications are implemented using the `TransferFunction` class. The following code shows examples of definitions of transfer functions:

```
double gain = 1.0;
double delay = 0.0;

// Impulse
TransferFunction Impulse =
    new TransferFunction(delay);

//Step function
TransferFunction Step =
    new TransferFunction(gain, delay);
// Pole

double tau = 10.0;
TransferFunction Pole =
    new TransferFunction(gain, delay, tau);
// second order Oscillating
tau = 10.0;
double sigma = 0.1;
TransferFunction OscillationgSecondOrder =
    new TransferFunction(gain, delay, tau, sigma);
TransferFunction.Show("OscillationgSecondOrder"
    , OscillationgSecondOrder);

// tranfer function from poles and zeroes
int np = 5;
CVector roots = new CVector(np);
for (int pos = 0; pos < np; pos++)
{
    roots[pos] = -pos;
}

int nr = 4;
CVector zeroes = new CVector(nr);
for (int pos = 0; pos < nr; pos++)
{
    zeroes[pos] = pos;
}

gain = 5.0;
delay = 2.0;
TransferFunction G =
    new TransferFunction(gain, delay, roots, zeroes);
```

```
TransferFunction.Show("G", G);
```

with the output:

```
OscillationgSecondOrder TransferFunction
```

```

R
0,0100
P
0,0100    0,0200    1
Gain      0,0100
Roots Complex List
-0,0100 i    0,0995
-0,0100 i    -0,0995
```

```
G TransferFunction
```

```

Delay      2,0000 seconds
R
-30,0000    55,0000    -30,0000    5,0000
P
24,0000    50,0000    35,0000    10,0000    1
Gain      5,0000
Zeroes Complex List
3,0000 i    0
2,0000 i    0
1 i    0

Roots Complex List
-1 i    0
-2,0000 i    0
-3,0000 i    0
-4,0000 i    0
```

Transfer functions supports the +,-,* and / operators. Time responses are obtained using the Value function.

```

gain = 5.0;
delay = 2.0;
TransferFunction G =
    new TransferFunction(gain, delay, roots, zeroes);
TransferFunction.Show("G", G);

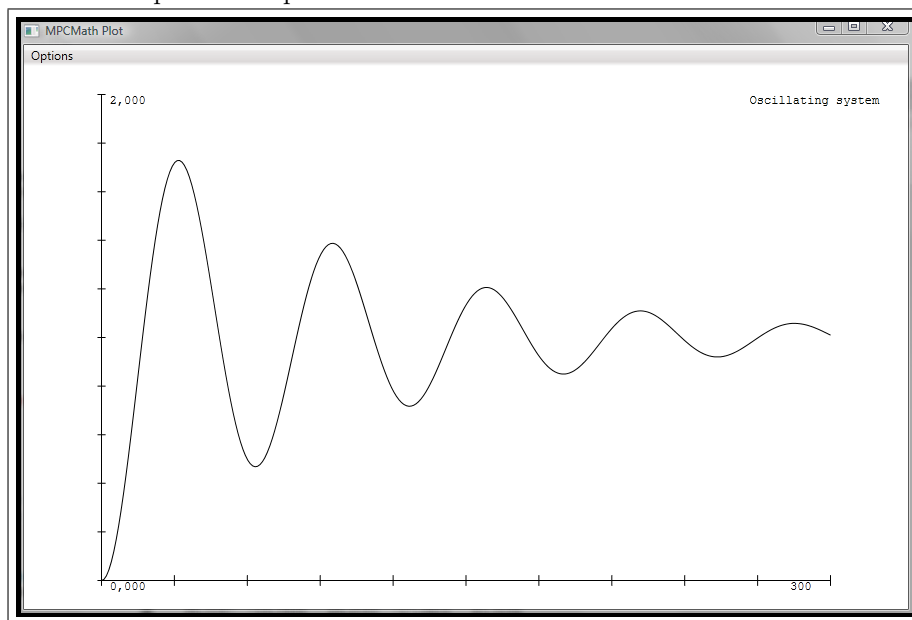
TransferFunction GR = OscillationSecondOrder * Step;
int steps = 300;

Vector resp = new Vector(steps);
for (int step = 0; step < steps; step++)
{
    double time = step;
    resp[step] = GR.Value(time);
}

console.Plot(
    new PlotSeries("Oscillating system",
        0.0, 2.0, resp));

```

with the plotted output



1.6 ARX Models

ARX models describes linear plant dynamics as

$$A(q)y(t) = B(q)u(t) + \epsilon(t)$$

$$A(q) = \sum_{i=0}^{n_y} a_i$$

$$B(q) = \sum_{i=0}^{n_u} b_i$$

q is the time shift operator $q_1 y(t) = y(t-1)$, and ϵ is the noise. The $E\Delta ARX$ model integrates the noise using the filter

$$\eta_k = \frac{1 - \alpha q^{-1}}{1 - q^{-1}} e_k$$

where α is in the range 0.0 to 1.0. The $E\Delta ARX$ integrates the noise in order to remove offsets in the closed control loop, see [Huusom et al., 2010]. The following code generates an ARX object *plant* and a $E\Delta ARX$ object *model*

```
// Arx model
Vector Aq = new Vector(1.0, -1.558, 0.5769);
Vector Bq = new Vector(0.0, 0.2094, 0.1744);
int delayCount = 25;
double dt = 2.0;

ARXModel plant =
    new ARXModel(delayCount, Aq, Bq, dt);
ARXModel.Show("plant", plant);

// Arx model with noise model
double alfa = 0.7;
ARXModel model =
    new ARXModel(delayCount, Aq, Bq, dt, alfa);
ARXModel.Show("model", model);

plant ARX Model

    Delay 25 steps
    A Vector
        1   -1,5580   0,5769
    B Vector
        0   0,2094   0,1744
model ARX Model

    Delay 25 steps
```

```

A Vector
  1   -2,5580   2,1349   -0,5769
B Vector
  0    0,2094   -0,0350   -0,1744

```

ARX and *EΔARX* objects can be directly created from transfer functions. The transfer function

$$G(s) = \frac{20}{(50s + 1)(4s + 1)} e^{-50s}$$

can be realized as *ARX* and *EΔARX* objects by

```

gain = 20.0;
delay = 50.0;
dt = 2.0;
Vector taus = new Vector(50.0, 4.0);
TransferFunction Furnace =
    TransferFunction.TauForm(gain, delay, taus);
TransferFunction.Show("Furnace", Furnace);
plant = new ArxModel(Furnace, dt);
ArxModel.Show("plant", plant);

```

```

Furnace TransferFunction
  Delay    50,0000 seconds
  R
  0,1000
  P
  0,0050    0,2700        1
  Gain      0,1000
  Roots Complex List
-0,0200 i      0
-0,2500 i      0

```

```

plant Arx Model

  Delay 25 steps
  A Vector
    1   -1,5673   0,5827
  B Vector
    0    0,1681   0,1405

```

The time responses from the *ARX* model can be calculated using the *NextState* and *Observed* functions

```

x= new Vector(plant.Dimension);
double u = 100.0;
double eps;
double epsVariance = 0.1;
double epsMean = 0.0;
resp = new Vector(steps);
for (int step = 0; step < steps; step++)
{
    eps = MPCMathLib.WhiteGaussianNoise(
        epsVariance,epsMean);
    x = plant.NextState(x, u, eps);
    resp[step] = plant.Observed(x, eps);
}

```

The state space matrices for the *ARX* and *EΔARX* objects are given in the ASP, BSP,KSP and C properties. The code sequence

```

for (int step = 0; step < steps; step++)
{
    eps = MPCMathLib.WhiteGaussianNoise(
        epsVariance, epsMean);
    x = plant.ASP * x + plant.BSP * u
        + plant.KSP * eps;
    resp[step] = plant.C * x + eps;
}

```

gives the same results obtained using NextState and Observed routines.

1.7 State Space Models

The state space model is used to describe linear multiple input multiple output, MIMO systems.

$$\begin{aligned}
 X_{k+1} &= AX_k + B * U_k + \epsilon_k \\
 Y_k &= CX_k + \eta_k
 \end{aligned}$$

Where ϵ_k is the process noise and η_k the measurement noise.

The StateSpace object can be created directly from a set of A,B,C and K matrices, in this case with matrices with random numbers:

```

int nx = 5;
int ny = 2;
int nu = 3;

A = Matrix.Random(nx, nx, -100.0, 100.0);
B = Matrix.Random(nx, nu, -100.0, 100.0);
K = Matrix.Random(nx, ny, -100.0, 100.0);
C = Matrix.Random(ny, nx, -100.0, 100.0);

StateSpaceModel spmodel =
    new StateSpaceModel(A, B, C, K);
StateSpaceModel.Show("spmodel", spmodel);

```

where nx are the number of states, nu the number of manipulated variables and ny the number of observed variables.

StateSpaceModel objects can also be created from an ARX model

```
spmodel = new StateSpaceModel(plant);
```

where *plant* is the ARX model from the previous section. As the ARX model is a Single Input Single Output, SISO, state space model, this conversion is rather trivial. It more interesting creating State space models object from an arrays of transfer functions or ARX functions.

A plant can be described by the following set of transfer functions, with 2 observed variable and 2 manipulated variables. In this case a model for a cement mill derived from experimental data.

$$G(s) = \begin{bmatrix} \frac{0.62}{(45s+1)(8s+1)}e^{-5s} & \frac{0.29(8s+1)}{(2s+1)(38s+1)}e^{-1.5s} \\ \frac{-15}{(60s+1)}e^{-5s} & \frac{5}{(14s+1)(s+1)}e^{-0.1s} \end{bmatrix}$$

These four transfer functions are inserted in an array of transfer functions with the code

```

// Cement mill MIMO system
Double T = 1.0;
int ny = 2;
int nu = 2;

Matrix<TransferFunction> CMModel =
    new Matrix<TransferFunction>
        (MatrixForm.General, ny, nu);

CMModel[0, 0] = TransferFunction.TauForm

```

and the state space model objects *Plant* and *Model* are generated

The StateSpaceModel object Model is created via *EΔARX* objects, with integration of noise, leading to offset free control.

In order to integrate the state space models the following objects are created:

```

// Define states of observed Variabes
Vector XPlant = new Vector(Plant.Dimension);
Vector XModel = new Vector(Model.Dimension);
Vector YPlant = new Vector(ny);
Vector YModel = new Vector(ny);
Vector UC = new Vector(nu);
Vector innovation;
Vector Zero = new Vector(ny);

// Measurement noise and process noise
Vector epsilonVar = new Vector(ny);
Vector xiVar = new Vector(nu);
if (noise)
{
    epsilonVar = new Vector(0.05, 10.0);
    xiVar = new Vector(0.0001, 0.1);
}

Vector xiMean = new Vector(0.0, 0.0);
Vector epsilonMean = new Vector(ny);

```

The simulation of the plant dynamics and tracking the plant with the model is done by the following code

```

steps = 1500;

// Tracking simulation
for (int step = 0; step < steps; step++)
{
    // Exercice manipulated vars
    // feed
    UC[0] = SetVal(step, 0, 300, 10.0);
    // Separator speed
    UC[1] = SetVal(step, 600, 700, 30.0);

    // umeasured disturbances
    // Load
    xiMean[0] = SetVal(step, 800, 900, 0.02);
    // Fineness
    xiMean[1] = SetVal(step, 1000, 1100, 1.0);

    // Measurement and process noise
    eps = MPCMathLib.WhiteGaussianNoise
        (epsilonVar, epsilonMean);
    xi = MPCMathLib.WhiteGaussianNoise

```

```

        (xiVar, xiMean);

// Plant measurement with measurement noise
YPlant = Plant.Observed(XPlant) + eps;
YModel = Model.Observed(XModel);

// Update Plant and Model state
innov = YPlant - YModel;
XModel = Model.NextState(XModel, UC, innov);
XPlant = Plant.NextState(XPlant, UC, xi);

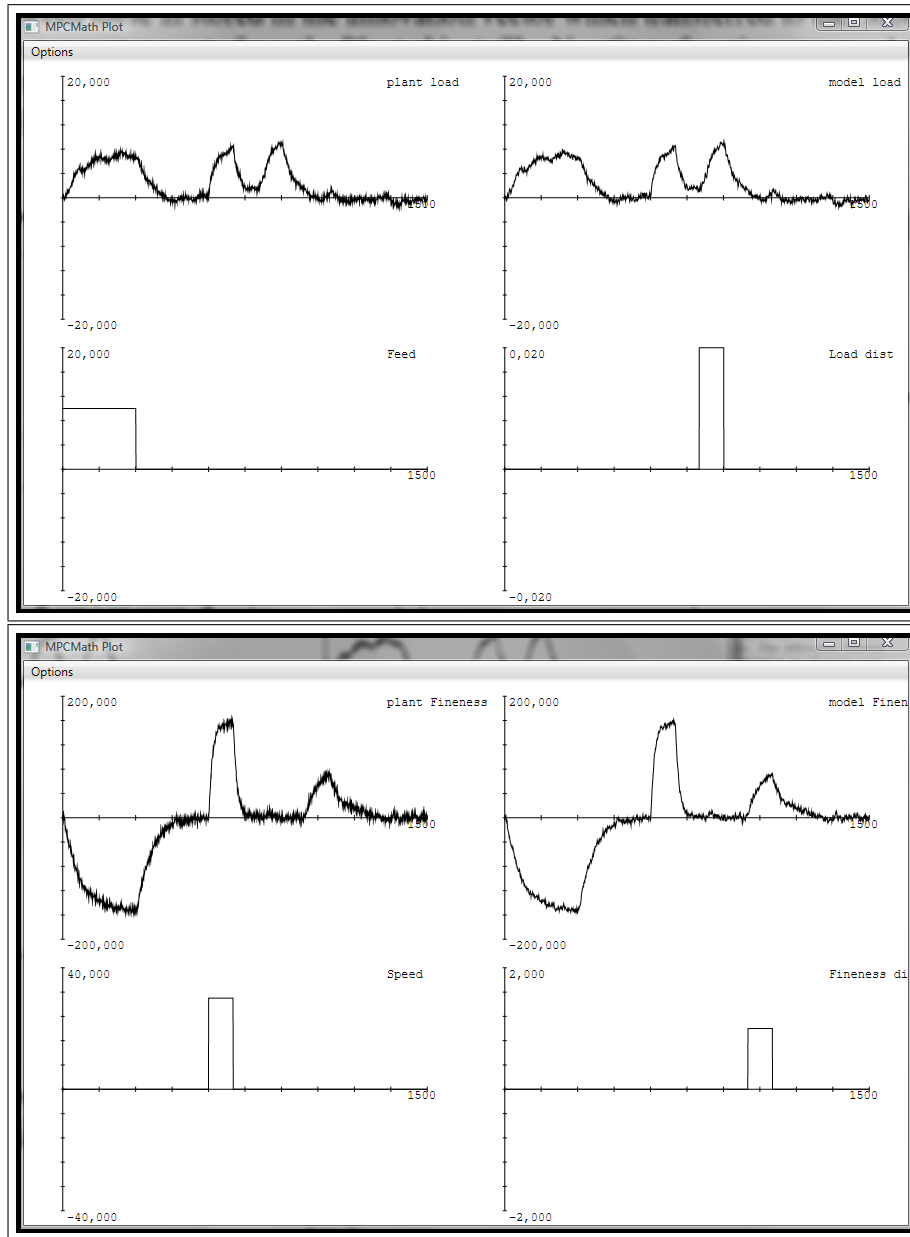
// collect plot indformation
ypload[step] = YPlant[0];
ymload[step] = YModel[0];
ypfineness[step] = YPlant[1];
ymfineness[step] = YModel[1];
Feed[step] = UC[0];
Speed[step] = UC[1];
dLoad[step] = xiMean[0];
dFineness[step] = xiMean[1];
}

console.Plot(sypload, symload,
             sFeed, sdLoad);
console.Plot(sypfineness, symfineness,
             sSpeed, sdFineness);

```

The manipulated variable feed, $UC[0]$, is increased to 10.0 from step 0 to 300. The separator Speed, $UC[1]$, is increased to 100.0 from step 600 to 700, and the unmeasured disturbances are active from step 800 to 900 and step 100 to 1100 respectively.

The simulated measurement noise and process noise are calculated and stored in vectors `eps` and `xi`. The `Yplant` and `Ymodel` are the observed responses from the plant and the Model respectively. The differences between these are stored in the innovation vector which is transferred to the Model in order to track the plant measurements from the Plant object. The `NextState` functions are used to update the Plant object (with process noise `xi`) and the Model object (with the innovation).

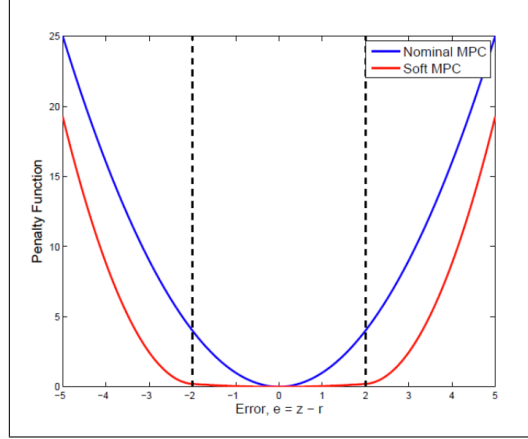


The Model objects act as a Kalman filter tracking the Plant state. Due to the noise integration, the Model is able to track the unmeasured disturbances without offsets.

1.8 Model Predictive Control

Model predictive control can be implemented using the MiMoMPC class. The MiMoMPC class implements both conventional MPC and Soft Constrained MPC. The conventional MPC has a quadratic penalty function and the Soft Constrained MPC has a dead band zone around the set point where the penalty

for not reaching the exact set point is low.



The SoftConstrained MPC minimizes the problem

$$\begin{aligned} \min_{\{z, u, \eta\}} \phi = & \frac{1}{2} \sum_{k=0}^{N-1} \|z_{k+1} - r_{k+1}\|_{Q_z}^2 + \|\Delta u_k\|_S^2 \\ & + \sum_{k=1}^N \frac{1}{2} \|\eta_k\|_{S_\eta}^2 + s'_\eta \eta_k \end{aligned}$$

subject to the constraints

$$\begin{aligned} z_k &= b_k + \sum_{i=1}^n H_i u_{k-i} & k &= 1, \dots, N \\ u_{\min} &\leq u_k \leq u_{\max} & k &= 0, \dots, N-1 \\ \Delta u_{\min} &\leq \Delta u_k \leq \Delta u_{\max} & k &= 0, \dots, N-1 \\ z_k &\leq z_{\max, k} + \eta_k & k &= 1, \dots, N \\ z_k &\geq z_{\min, k} - \eta_k & k &= 1, \dots, N \\ \eta_k &\geq 0 & k &= 1, \dots, N \end{aligned}$$

where z_k is the plant response, r_k the set-point, ΔU_k the movement of the manipulated variables. The Q_z , S and S_η are weighing matrices for reference error penalty, S movement of manipulated variables penalty and S_η the penalty for coming outside the dead-band zone. Setting S_η to zero result in a conventional MPC controller. The first constraint for z_k is the linear process model. For further details consult [Prasath and Jørgensen, 2010]

```
// MPC controller data
int history = 100;    // history for operator display
int horizon = 50;     // optimization horizon
Vector theta = null;
Vector my = null;
```

```

Vector ymax = null;
Vector ymin = null;
if (SoftConstraint)
{
    // soft constrained MPC

    // reference penalty
    theta = new Vector(0.1, 0.2);
    // soft constraint penalty
    my = new Vector(1000.0, 500.0);
    // soft constraint max values
    ymax = new Vector(2.0, 5.0);
    // soft constraint min values
    ymin = new Vector(-2.0, -5.0);
}
else
{
    // conventional MPC

    // reference penalty
    theta = new Vector(1000.0, 500.0);
}

// manipulated vars movement penalty
Vector rho = new Vector(100.0, 200.0);
// Hard constraint U max value
Vector umax = new Vector( 15.0, 30.0);
// Hard constraint U min value
Vector umin = new Vector(-30.0, -30.0);

```

The actual MPC controller and the control loop is defined by the code below (Model and Plant objects are the StateSpace models of the Cement mill described in the previous section).

```

// Setup MiMo MPC
MiMoMPC mpc = new MiMoMPC(Model, history, horizon,
    theta, my, ymin, ymax, rho, umin, umax);

// Get references to operator information
Matrix ypopr = mpc.YPlant;
Matrix ymopr = mpc.YModel;
Matrix yropr = mpc.YRef;
Matrix uopr = mpc.U;

// MPC control
for (int step = 0; step < steps; step++)
{

```

```

// Manipulate setpoints
// Fineness setpoint
YRef[1] = SetVal(step, 0, 100, 100.0);
// elevator load setpoint
YRef[0] = SetVal(step, 150, 250, 10.0);
mpc.SetRef(YRef);

// Measurement and process noise
eps = MPCMathLib.WhiteGaussianNoise
    (epsilonVar, epsilonMean);
xi = MPCMathLib.WhiteGaussianNoise
    (xiVar, xiMean);

// Plant measurement with measurement noise
YPlant = Plant.Observed(XPlant) + eps;

// MPC controller
UC = mpc.Next(YPlant);

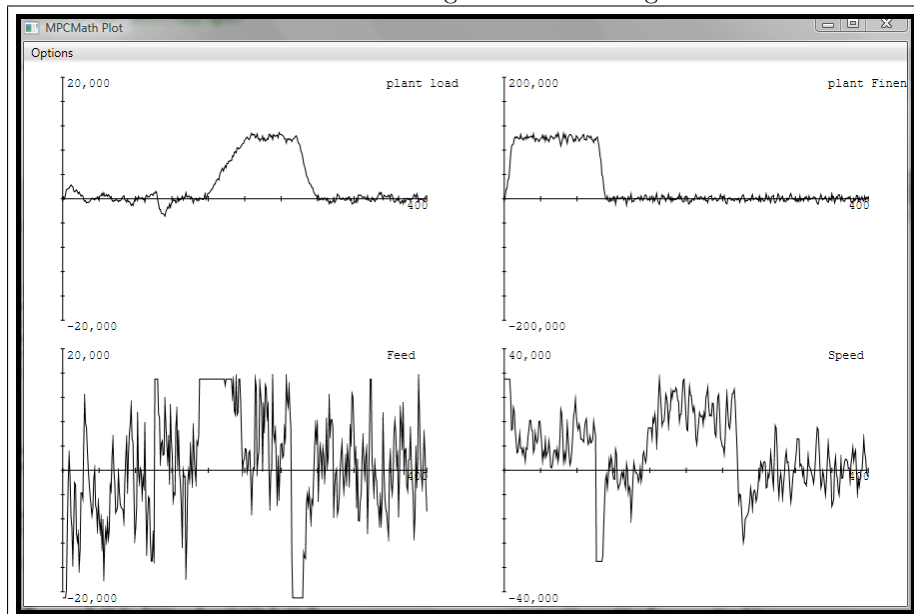
// send control signal and process noise to plant
XPlant = Plant.NextState(XPlant, UC, xi);

console.WriteLine("step " + step.ToString()
    + " iterations "
    + mpc.Solver.Iterations.ToString());
}

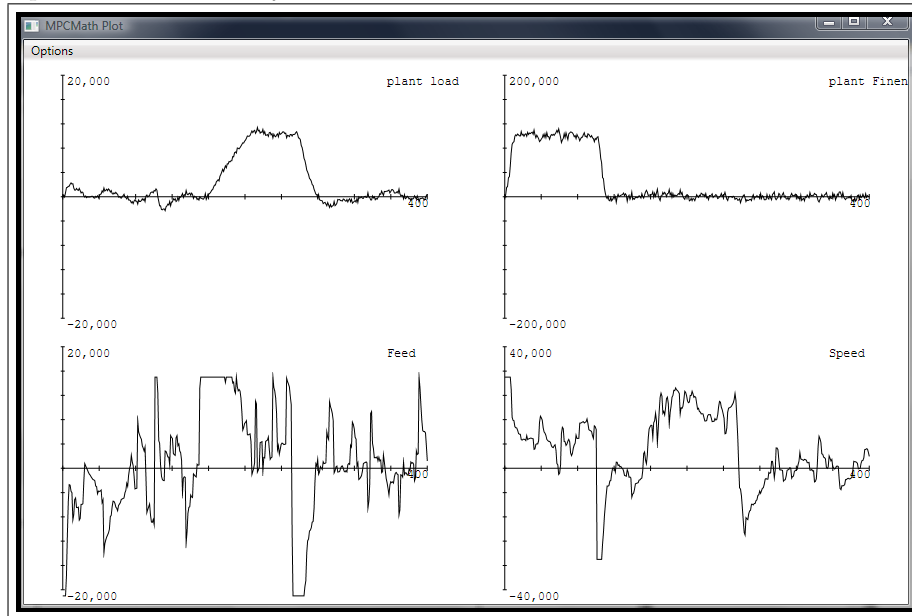
```

The set point for the Fineness, $YRef[2]$, is set to 100.0 for step 0 to 100, and the setpoint for the Elevator load, $YRef[0]$, is set to 100.0 for step 150 to 250.

The conventional MPC controller gives the following result



a good control of elevator load and the fineness are achieved, but the manipulated variables very active. The SoftConstrained MPC results in



The achieved control is as good the conventional MPC, but the manipulated variables are much better. The SoftConstrained MPC is much more robust for model errors than the conventional MPC, [Prasath and Jørgensen, 2010]

The MPC problems are solved using the QuadraticProblemSolver object. This object find the minimum of the object function:

```
// Minimize    X'G*X + g'*x
// s.t.        A*X = b
//            G*X >= d

QPSolver solver = new QPSolver(G, g, A, b, C, d);

solver.Solve();

console.Show("iterations", solver.Iterations);
console.Show("My", solver.My); // dual gap
BVector.Show("x", solver.X); // solution
// equality lagrange variables
BVector.Show("y", solver.Y);
// slack variables
BVector.Show("s", solver.S);
// inequality lagrange variables
BVector.Show("z", solver.Z);
```

The object uses a Primal-Dual Interior-Point algorithm to find the mini-

mum. Its a general routine, which exploit the structure of G , A , C to speed up calculations especially for problems with long time horizons. The QPSolver solves the KKT system by a call to a KKT solver defined by an interface. This makes it possible to develop KKT solver that exploit structures in the system matrices

```
solver = new QPSolver(new SoftConstraintKKT solver
    (G, g, A, b, C, d));
```

The complete source code for this example is given in the *MPCMath* download package from <http://www.2-control.dk> in the TestMiMoMPC project.

1.9 Intel Math Kernel Library (BLAS and LAPACK)

Many of *MPCMath* 's functions are based on Intel Math Kernel Library [1], a highly optimized implementation of the public domain computing packages BLAS (Basic Linear Algebra Subroutines) and LAPACK (Linear Algebra Package). This gives *MPCMath* a boost in performance resulting more than an order of magnitude.

The critical cpu consuming calculations in the MPC controllers and the QPSolver are the matrix multiplications and The Cholesky factorizations.

The speed of these operations can be measured with the following code

```
console.WriteLine("Test Cholesky Speed/\MPCMath");
console.WriteLine();

int dim = 32;
for (int test = 0; test < 7; test++)
{
    dim = 2 * dim;
    Matrix A = Matrix.Random
        (dim, dim, -100.0, 100.0);
    Matrix AT = Matrix.Transpose(A);
    Vector r = Vector.Random
        (dim, -1000.0, 1000.0);

    // Matrix multiplication
    DateTime start = DateTime.Now;
    A = AT * A;

    TimeSpan tm = DateTime.Now - start;
```

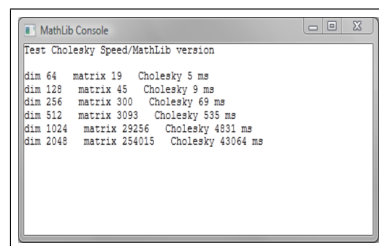
```
// Cholesky factorizatio + solve
start = DateTime.Now;
CholeskyEquationSolver solver =
    new CholeskyEquationSolver(A);
Vector x = solver.Solve(r);

TimeSpan tc = DateTime.Now - start;

console.WriteLine("dim " + dim.ToString()
    + " matrix "
    + tm.TotalMilliseconds.ToString()
    + " Cholesky "
    + tc.TotalMilliseconds.ToString());
}
```

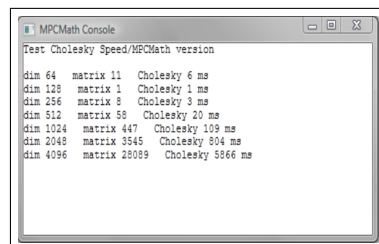
which measures calculation time for multiplication of random square matrices and subsequent Cholesky factorization.

The results obtained with the best possible C# code is



where multiplication of two 2048x 2048 matrices takes 254015 ms or 4.2 minutes (MathLib was a predecessor to *MPCMath* written entirely in C#).

With *MPCMath* using Intel MKL the results is



where multiplication of two 2048x 2048 matrices takes 3545 ms, a factor 70 improvement !

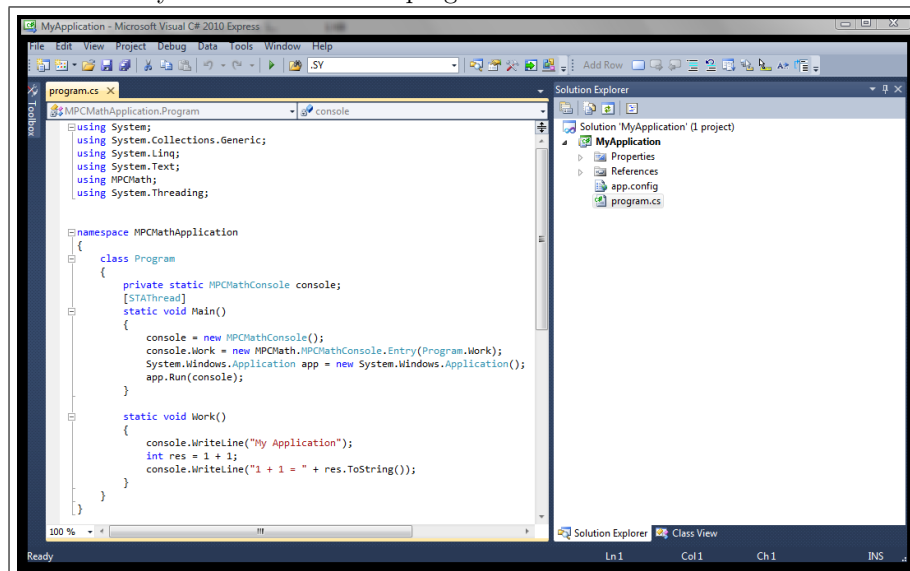
Chapter 2

Console functions

The MPCMathConsole is the development and debugging tool for *MPCMath* applications. The MPCMathConsole is normally created from the MPCMath template application delivered together with the *MPCMath* Package. Using MPCMathConsole requires a valid Development or a Demo license. It not available using a Runtime license.

2.1 Creating a new *MPCMath* application

Copy the folder C:2-controlMPCMathTemplate to a new folder and rename the new folder the name of your application i.e. "MyApplication". Open the new folder and double click the solution file MPCMathTemplate.sln. In the Solution explorer window rename the solution and the project file to your applications name. Finally enter some code in program work routine as shown below



The normal Main program for the application, creates the MPCMathConsole, set the entry of the Program.Work routine, and then starts the created MPCMathConsole by calling `app.Run(console);`. The MPCMath console runs

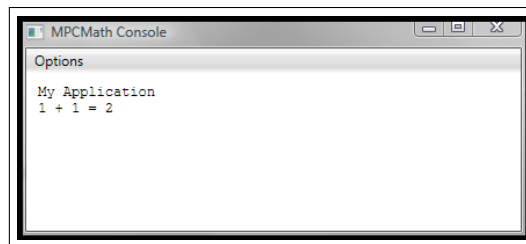
in a new parallel tread, execution the code specified in the *Program.Work* routine. Separating the MPCMathConsole main tread from the work tread, enables print out of information during program execution.

```
private static MPCMathConsole console;
[STAThread]
static void Main()
{
    console = new MPCMathConsole();
    console.Work =
        new MPCMath.MPCMathConsole.Entry(Program.Work);
    System.Windows.Application
        app = new System.Windows.Application();
    app.Run(console);
}
```

the work routine is:

```
static void Work()
{
    console.WriteLine("My Application");
    int res = 1 + 1;
    console.WriteLine("1 + 1 = " + res.ToString());
}
```

Running the application shows The MPCMathConsole



2.2 Displaying and plotting variables

Text output to the console is done using the Show and Writeline routines, some exaples are

```
// Show routines
console.WriteLine("output a line to console");
```

```
// output a blank line
console.WriteLine();

// output a bool var
bool Ok = true;
console.Show("Ok", Ok);

// output an array of integers
int[] intArr = { 1, 4, 5, 6, 7 };
console.Show("Integer array", intArr);

// output double
double pi = Math.PI;
console.Show("pi short", pi);

MPCMathConsole.Digits = 8;
console.Show("pi", pi);
```

with the following output

output a line to console

Ok True

Integer array

1 4 5 6 7

pi short 3,1416

pi 3,14159265

Integer array

1 4 5 6 7

If the content of the console window exceeds 300000 characters (appr. 2500 lines) the console throws an "MPCMathConsole" "Maximum text length exceeded" exception. Too long texts in the SchrollViewer slows down the performance and has the same effect as the good old memory leaks in C++ and c. The exception can be avoided by clearing the content of the console using `console.Clear()` 2.7 statements. If the output must be preserved use the `MPCMathConsole.Trace` to save the output in a text file.

Plots are generated using the `Plot` routine. The following code defines a vector and fills it with a sinus curve. The `PlotSeries` defines the label, min, max plot values (to be found automatically, and finally set `x` as the vector holding the values to be plotted. The call to `console.Plot` creates a new window with the plot.

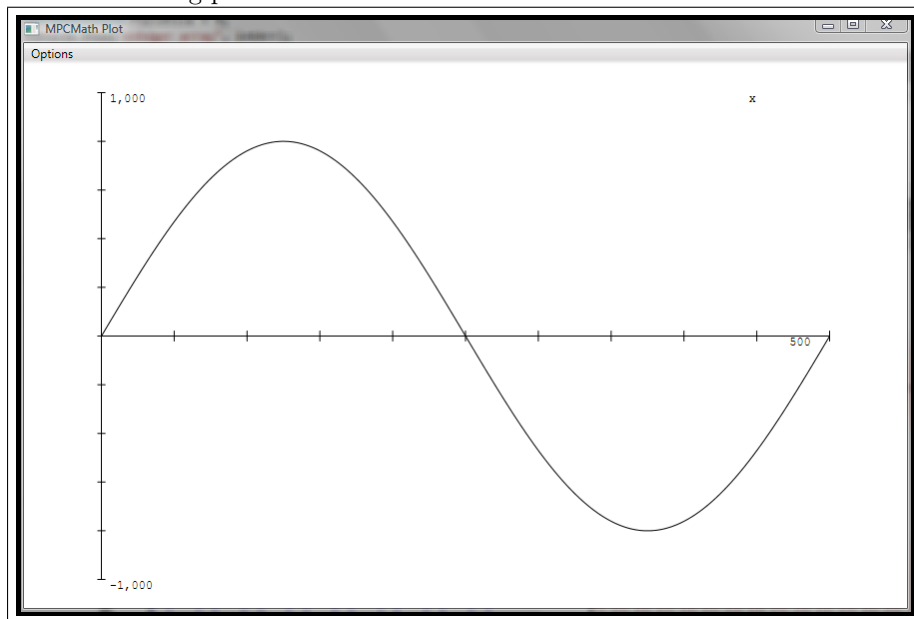
```

int steps = 500;
Vector x = new Vector(steps);
for (int step = 0; step < steps; step++)
{
    x[step] = 0.8 * Math.Sin(2.0 * pi * step / (steps - 1));
}

PlotSeries xs = new PlotSeries("x", -1.0, 1.0, x);
console.Plot(xs);

```

The resulting plot is



Plot can handle of to four plot series

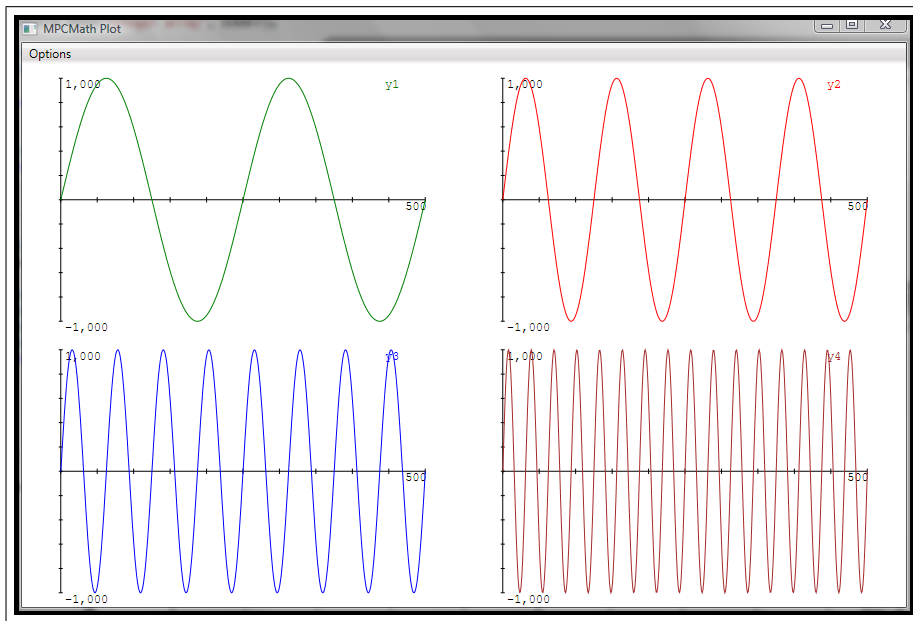
```

Vector y1 = new Vector(steps);
Vector y2 = new Vector(steps);
Vector y3 = new Vector(steps);
Vector y4 = new Vector(steps);
for (int step = 0; step < steps; step++)
{
    y1[step] = Math.Sin(4.0 * pi * step / (steps - 1));
    y2[step] = Math.Sin(8.0 * pi * step / (steps - 1));
    y3[step] = Math.Sin(16.0 * pi * step / (steps - 1));
    y4[step] = Math.Sin(32.0 * pi * step / (steps - 1));
}

console.Plot(
    new PlotSeries("y1", 0.0, 0.0, SeriesColor.Green, y1),
    new PlotSeries("y2", 0.0, 0.0, SeriesColor.Red, y2),

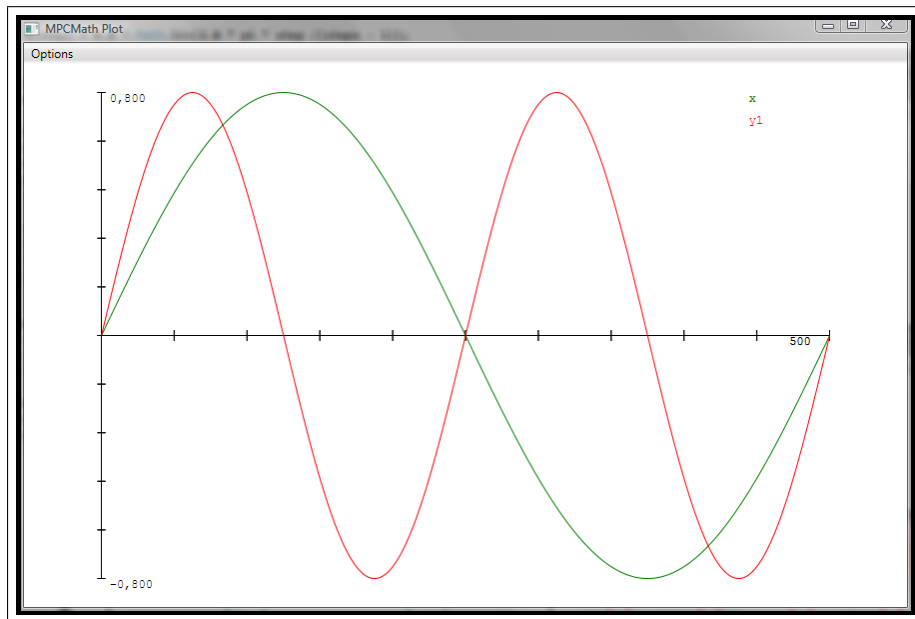
```

```
new PlotSeries("y3", 0.0, 0.0, SeriesColor.Blue, y3),
new PlotSeries("y4", 0.0, 0.0, SeriesColor.Brown, y4));
```



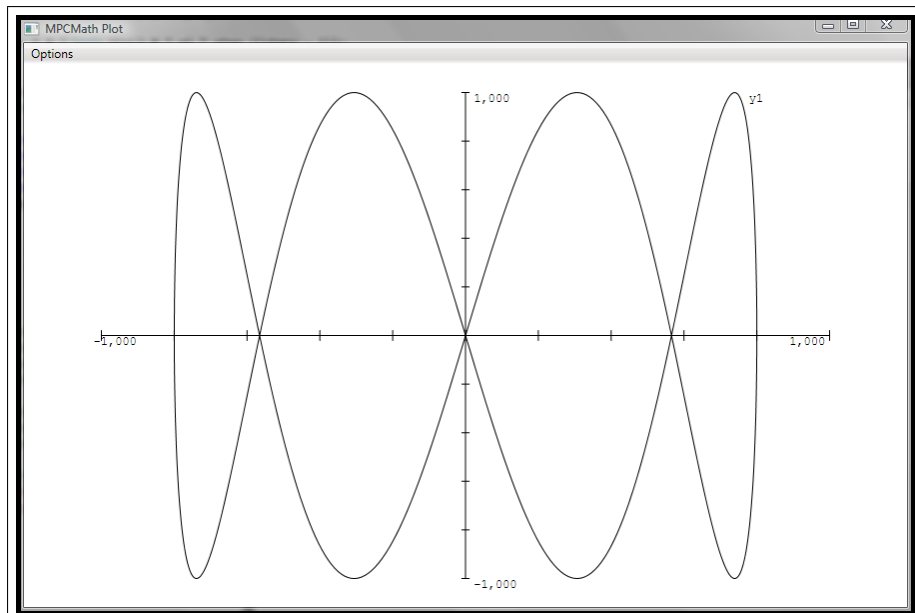
Using the Plot class several plot series can be shown in one coordinate system.

```
console.Plot(new Plot(
    new PlotSeries("x", 0.0, 0.0, SeriesColor.Green, x),
    new PlotSeries("y1", 0.0, 0.0, SeriesColor.Red, y1)));
```



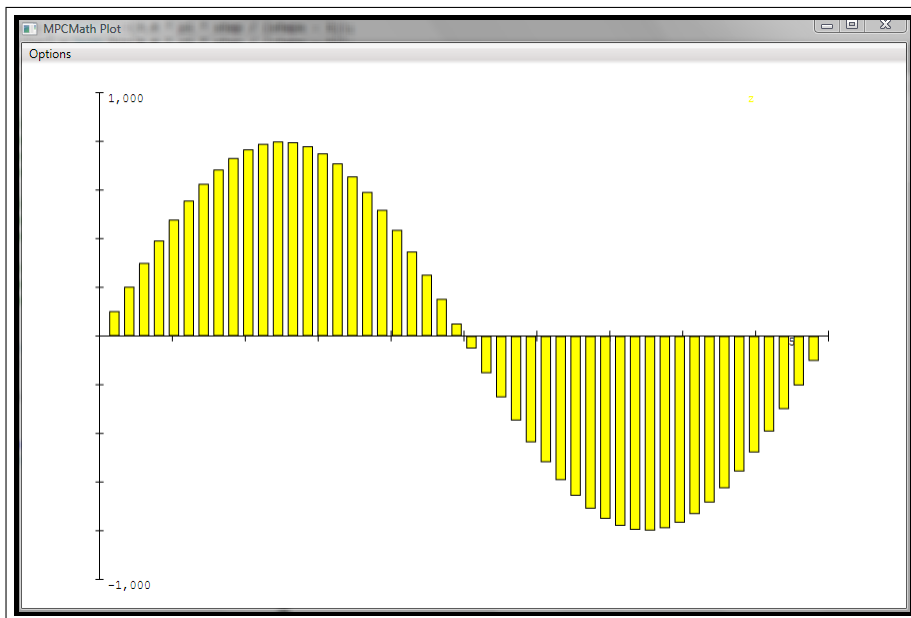
The Plot class also allows one to specify the plot type using the PlotType enumeration

```
console.Plot(new Plot(PlotType.XY,
    new PlotSeries("x", -1.0, 1.0, x),
    new PlotSeries("y1", -1.0, 1.0, y2)));
```



```
steps = 50;
Vector z = new Vector(steps);
for (int step = 0; step < steps; step++)
{
    z[step] = 0.8 * Math.Sin(2.0 * pi * step / (steps - 1));
}

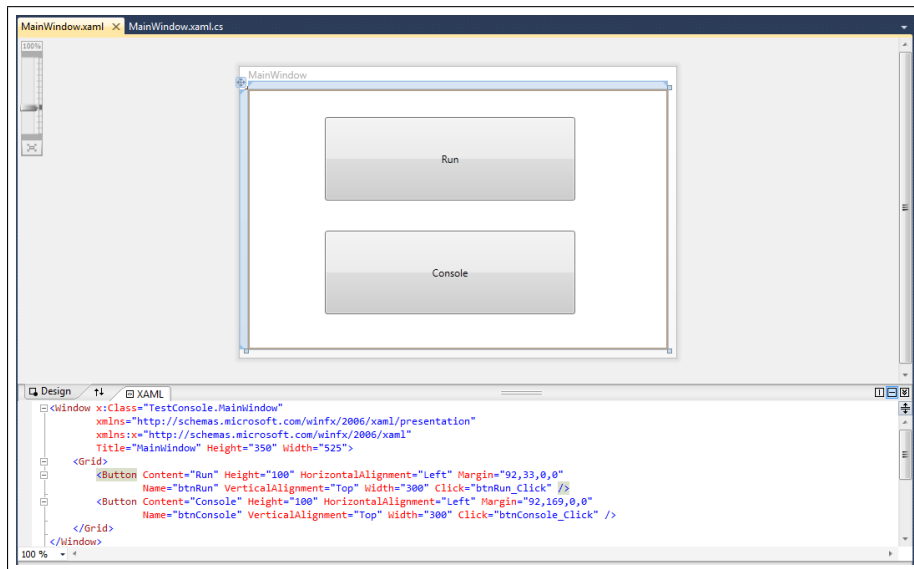
console.Plot(new Plot(PlotType.TBar,
    new PlotSeries("z", -1.0, 1.0, SeriesColor.Yellow, z)));
```



2.3 Using Console for test of GUI programs

It can be very convenient to use the Console as a part of a GUI program. Here console can be used to display debug information or detailed information about program operation.

Define a new WPF application and with two buttons *Run* and *Console*.



```
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
using MPCMath;
```

```
namespace TestConsole
```

```
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        private MPCMathConsole console;

        public MainWindow()
        {
            InitializeComponent();
            this.console = new MPCMathConsole(true);
        }

        private void btnRun_Click(object sender, RoutedEventArgs e)
        {
            MPCMath.Matrix A = MPCMath.Matrix.Random(5, 5, -100.0, 100.0);
            MPCMath.Matrix.Show("A", A);
        }

        private void btnConsole_Click(object sender, RoutedEventArgs e)
        {

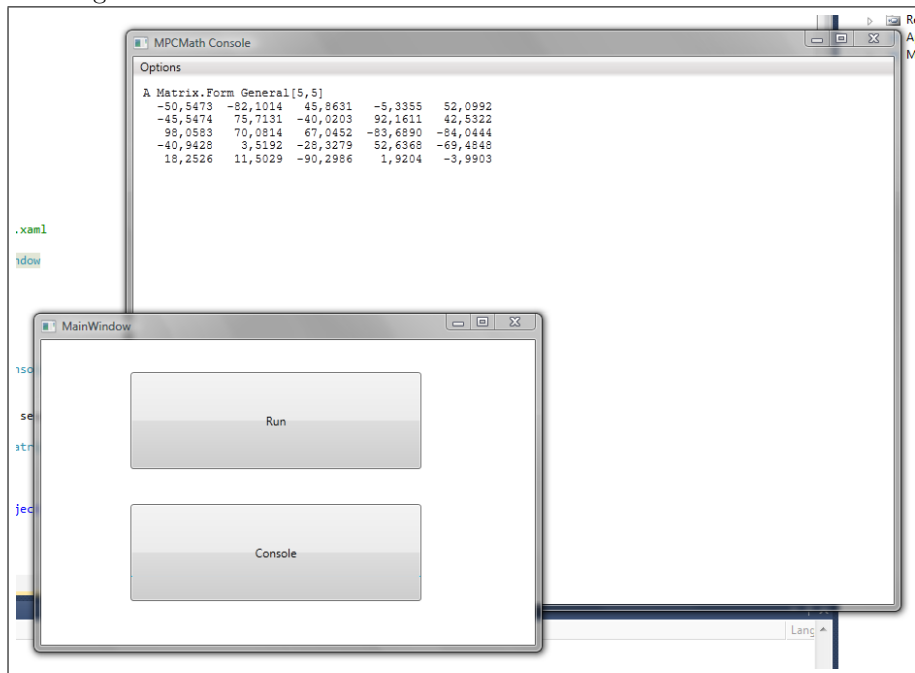
```

```

        this.console.Show();
    }
}
}

```

The constructor creates the MPCMathConsole and stores a reference to it in the private variable `console`. The *Run* button creates some output on the console. The *Run* makes the console visible. The console can be hidden by selecting the menu item *Hide*.



2.4 MPCMathConsole

Class MPCMathConsole

MPCMath console class for test and debugging of MPCMath programs. Running MPCMath Console requires a valid Development or Demo license

```
class MPCMathConsole : MPCMathWindow
```

Constructors

```
MPCMathConsole() : this(false)
```

Default constructor, without Hide option in console window

```
MPCMathConsole(bool EnableHide) : base(EnableHide)
```

Constructor

Parameters

EnableHide Enable Hide option in console window

Properties

Console

Get reference to MPCMath console

```
static MPCMathConsole Console {get;}
```

Digits

Digits for doubles

```
static int Digits {set; get;}
```

FieldSize

Field size of vector and matrix display

```
static int FieldSize {set; get;}
```

MaxTextLength

Maximum textlength in console window

```
static int MaxTextLength {set; get;}
```

NumbersPerLine

Numbers per Line

```
static int NumbersPerLine {set; get;}
```

TextLength

length of text in console window

```
static int TextLength {get;}
```

Trace

Trace console output to file "console.txt" in bin/debug or bin/release directory

```
static bool Trace {set; get;}
```

Work

Callback entry point defining work thread

```
Entry Work {set;}
```

Methods**Abort**

Abort execution of Work thread. Usefull for debugging code generating unhandled Exceptions, which prevents reading the output on the MPCMath console. The routine aborts the current work thread.

```
void Abort()
```

Abort

Abort execution of Work thread. Usefull for debugging code generating unhandled Exceptions, which prevents reading the output on the MPCMath console. The routine aborts the current work thread.

```
void Abort(string text)
```

Parameters

text Text string outputted to console

AssertEqual

Assert equal for bool variable Show actual and expected if they are not equal

```
void AssertEqual(string text, bool actual, bool expected)
```

Parameters

text variable desription

actual actual value

expected expected value

AssertEqual

Assert equal for integers Show actual and expected if they are not equal

```
void AssertEqual(string text, int actual, int expected)
```

Parameters

text variable desription

actual actual value

expected expected value

AssertEqual

Assert equal for integer array Show actual and expected if they are not equal

```
void AssertEqual(string text, int[] actual, int[] expected)
```

Parameters

text	variable description
actual	actual value
expected	expected value

AssertEqual

Assert equal for double Show actual and expected if they are not equal

```
void AssertEqual(string text, double actual, double expected)
```

Parameters

text	variable description
actual	actual value
expected	expected value

AssertEqual

Assert equal for complex Show actual and expected if they are not equal

```
void AssertEqual(string text, complex actual, complex expected)
```

Parameters

text	variable description
actual	actual value
expected	expected value

Clear

Clear Console window

```
void Clear()
```

DFormat

Format double variable

```
static string DFormat(double a)
```

Parameters

a	double variable
returns	Formatted output string

DFormat

Format double variable

```
static string DFormat(int Fieldsize, double a)
```

Parameters

Fieldsize Field size

a double variable

returns Formatted output string

Entry

Delegate specifying Work routine format

```
delegate void Entry()
```

Plot

Plot series in new window, each series have a separate coordinate system. Up to four series

```
void Plot(params PlotSeries[] Series)
```

Parameters

Series Series definition

Plot

Plot series i new or existing window

```
void Plot(string PlotID, params PlotSeries[] Series)
```

Parameters

PlotID Window identification string

Series Series definition

Plot

Plot plots in new window. Each plot can have a number of series. Up to four plots in the window

```
void Plot(params Plot[] Plots)
```

Parameters

Plots

Plot

Plot plots in new or existing window.

```
void Plot(string PlotID, params Plot[] Plots)
```

Parameters

PlotID window identification

Plots Plot definitions

Print

Print console window

```
void Print()
```

Print

Print text

```
void Print(string text)
```

Show

Show Console Window

```
void Show()
```

Show

Show bool

```
void Show(string text, bool a)
```

Parameters

text Variable description

a bool variable

Show

Show int, integer

```
void Show(string text, int r)
```

Parameters

text Variable description

r integer variable

Show

Show int[], integer array

```
void Show(string txt, int[] a)
```

Parameters

txt Variable description

a integer array

Show

Show double

```
void Show(string text, double r)
```

Parameters

text Variable description

r double variable

Show

Show complex

```
void Show(String text, complex c)
```

Parameters

text Variable description

c complex variable

Show

Show complex List

```
void Show(string txt, List<complex> a)
```

Parameters

txt Variable description

a list of complex variables

Show

Show double[], double array

```
void Show(string txt, double[] a)
```

Parameters

txt Variable description

a double array

WriteLine

Write empty line to console

```
void WriteLine()
```

WriteLine

Write line to console

```
void WriteLine(string txt)
```

Parameters

txt text string

2.5 PlotSeries*Class* **PlotSeries**

Class for definition of plot series.

```
[Serializable]
```

```
class PlotSeries
```

Constructors

```
PlotSeries()
```

Default constructor for serialization

```
PlotSeries(string Text, double Min, double Max, Vector Values):  
this(Text, Min, Max, SeriesColor.Black, Values)
```

Constructor. If both Min = 0.0 and Max = 0.0 uses the NiceMin and NiceMax routines to find suitable Min and Max values

Parameters

Text Series description
Min Minimum plot value
Max Maximum plot value
Values Vector with series values

```
PlotSeries(string Text, double Min, double Max, SeriesColor Color, Vector Values)
```

Parameters

Text Series description
Min Minimum plot value
Max Maximum plot value
Color series color
Values Vector with series values

Properties

Color

Series color

```
SeriesColor Color {set; get;}
```

Max

Maximum plot value

```
double Max {set; get;}
```

Min

Minimum plot value

```
double Min {set; get;}
```

Text

Series description

```
string Text {set; get;}
```

Values

Plot values

```
Vector Values {set; get;}
```

Methods

Equal

Equal Plotseries objects

```
static bool Equal(PlotSeries a, PlotSeries b)
```

Parameters

a Plotseries object a

b Plotseries object a

returns true if equal

NiceMax

Nice Maximum value for plot (example $x = 0.75464$, $\text{NiceMax}(x) = 0.8$)

```
static double NiceMax(double x)
```

Parameters

x value

NiceMin

Nice Minimum value for plot

```
static double NiceMin(double x)
```

Parameters

x

ScalePlot

Scale Plot

```
void ScalePlot()
```

Enumeration **SeriesColor**

Series Colors

```
enum SeriesColor
```

Fields

Black

Blue

Brown

Cyan

DarkBlue

DarkGreen

DarkRed

Gray

Green

LightGray

Magenta

Orange

Red

Yellow

2.6 Plot

Class Plot

Plot definition class

```
[Serializable]
class Plot
```

Constructors

Plot()

Constructor

```
Plot(params PlotSeries[] Series)
```

Constructor for default plot with PlotType.TY . Up to four series can be specified

Parameters

Series

```
Plot(PlotType Type)
```

Constructor for plot plot type, series must be set using property Series

Parameters

Type Plot type

```
Plot(PlotType Type, params PlotSeries[] Series)
```

Constructor for up to four series.

Parameters

Type Plot type

Series Plot Series

```
Plot(PlotType Type, double XMin, double XMax)
```

Constructor for a plot type, series must be set separately

Parameters

Type Plot type

XMin Min x-axis value

XMax Max x-axis value

```
Plot(PlotType Type, double XMin, double XMax, params PlotSeries[] Series)
```

Constructor a given plottype with up to four series

Parameters

Type Plot type

XMin Min x-axis value

XMax Max x-axis value

Series Plot Series

Properties**PlotSeries**

Plot Series

```
PlotSeries[] PlotSeries {set; get;}
```

Type

Plot Type

```
PlotType Type {set; get;}
```

XMax

X axis maximum value

```
double XMax {set; get;}
```

XMin

X axis minimum value

```
double XMin {set; get;}
```

Enumeration **PlotType**

Plot type

```
enum PlotType
```

Fields

MPC	MPC Operator plot
TBar	TY with bars (only one series)
TY	TY plots Time for x-axis , series for Y axis
XY	Y versus X Plot

2.7 MPCMathWindow.xaml*Class* **MPCMathWindow**

MPCMath Console window (MPCMathWindow.xaml). This is mainly for internal use

```
partial class MPCMathWindow : Window
```

Constructors

`MPCMathWindow(bool EnableHide)`

Constructor for MPCMathWindow

Parameters

EnableHide Enable Hide field in menu

Properties

Text

Get window text

```
string Text {get;}
```

Methods

AddLine

Add line to text area

```
void AddLine(string Line)
```

Parameters

Line Text line

Clear

Clear Text area

```
void Clear()
```


Chapter 3

Vectors and Matrices

Linear algebra is handled using Vector and Matrix objects.

Vector and Matrix objects supports operator over loads, documented as

operator

`static Vector operator *(Matrix a, Vector x)`

Parameters

a a matrix

x x vector

returns result vector

means that the compiler accepts the $A * x$ statement in the code below:

```
int dim = 5;
Vector x = Vector.Random(dim,-100.0, 100.0);
Matrix A = Matrix.Random(dim,dim, -100.0,100.0);
Vector res = A * x;
```

The individual elements of a vector can be read or set using `[pos]`, `[row,col,]` statements as below:

```
double velm = x[3];
double aelm = A[0,4];
x[2] = 1.3 * x[3] / A[4, 4] + velm;
```

The Vector class is build on the base class *VBase* `< double >` and implements the interface *ICommon* `< Vector >`. VBase provides basic vector functionalities and ICommon allows the Vector object to be included as elements in Block and Generic Vectors and Matrices.

The Matrix class is build on the base class *MBase* `< double >` and implements the interface *ICommon* `< Matrix >`. MBase provides basic matrix

functionalities and `ICommon` allows the `Matrix` object to be included as elements in `Block` and `Generic Vectors and Matrices`.

`VBase`, `Mbase` and `ICommon` are documented in appendix C.

Matrices are created as null matrices as a default. The different `MatrixForms` can be created using special constructors. Examples are:

```
// Null matrix with 5 rows and 5 columns
Matrix B = new Matrix(dim);

// Constant diagonal matrix with 5 rows and 5 columns
Matrix C = new Matrix(MatrixForm.ConstantDiagonal, 5);

// Diagonal matrix with 6 rows and 7 columns
Matrix D = new Matrix(MatrixForm.Diagonal, 6,7);

// General matrix with 10 rows and 20 columns
Matrix e = new Matrix(MatrixForm.General, 10, 20);
```

Addressing non diagonal element in a diagonal matrix, `C[3,4]`, creates an indexing error. The best way to locate the sinner is to look in the call stack until you find your call. Forgetting to specify the `MatrixForm` also generates a lot of indexing errors.

The form of a `Matrix` is read or changed using the `Form` property.

```
// Upgrade C matrix to a general matrix
C.Form = MatrixForm.General;
```

Upgrades to a `Matrix` preserves data. Attempts to degrade the status of a `Matrix` are ignored.

3.1 Sparse Matrices

MPCMath implements sparse matrices using the `PARDISO` storage scheme described in appendix G. The sparse matrices are used by the `SparseSolver 8.7` and the sparse format are supported for matrix / vector and matrix/ matrix multiplications.

3.2 Vector

Class Vector

Vector class

```
[Serializable]
class Vector : VBase<double>, ICommon<Vector>
```

Constructors

```
Vector()
: base(0)
```

Default constructor for serialization

```
Vector(int n)
: base(n)
```

Constructor
Parameters
n Dimension

```
Vector(int n, double val)
: base(n, val)
```

vector with equal values
Parameters
n Dimension
val Value

```
Vector(params double[] vals)
: base(vals)
```

Constructor from array of doubles. Vector dimension equal to array length
Parameters
vals Array of doubles

Methods

Add

Add Vector, $y = y + x$

```
void Add(Vector x)
```

Parameters
x x vector

Add

Add constant to vector $y[i] = y[i] + a$

```
void Add(double a)
```

Parameters
a constant

AssertEqual

assert equal, Compare two vectors and output value if not equal

```
static void AssertEqual(string text, Vector actual, Vector expected)
```

Parameters

text	Text string
actual	Vector with actual values
expected	Vector with expected values

Axpy

Add Vector ($y = a*x + y$)

```
void Axpy(double a, Vector x)
```

Parameters

a	a factor
x	x vector

Covr

Covariance vector cov[k]

```
static Vector Covr(int kFirst, int kLast, Vector a, Vector b)
```

Parameters

kFirst	first k
kLast	end k
a	Input vector
b	Input vector
returns	Vector of covariances

DivElements

Divide Vector elements, $res[i] = y[i]/x[i]$

```
static Vector DivElements(Vector y, Vector x)
```

Parameters

y	y vector
x	z vector
returns	result vector

Equal

Equal, compare two vectors. (Error limit given in MPCMathLib.MaxError)

```
static bool Equal(Vector y, Vector x)
```

Parameters

y Vector y
 x Vector x
 returns true if equal

Gemv

Gemv, $y := \alpha * \text{op}(a) * x + \beta * y$

```
void Gemv(Matrix a, Vector x, MatrixOp opr, double alpha, double beta)
```

Parameters

a Matrix a
 x Vector x
 opr MatrixOp N,T
 alpha constant
 beta constant

Get

Get subvector

```
Vector Get(int pos, int dim )
```

Parameters

pos Initial position
 dim Dimension of subvector
 returns Subvector

Get

Get subvector with increments

```
Vector Get(int pos, int incr, int dim)
```

Parameters

pos Initial position
 incr Increment
 dim Dimension of subvector
 returns Subvector

Max

Get Maximum value

```
double Max()
```

Parameters

returns Maximum value of vector elements

Mean

Get Mean value

```
double Mean()
```

Parameters

returns Mean value of vector elements

Min

Get Minimm value

```
double Min()
```

Parameters

returns Minimum value of vector elements

MulElements

Multiply Vecor elements, $\text{res}[i] = y[i]*x[i]$

```
static Vector MulElements(Vector y, Vector x)
```

Parameters

y y vector

x x vector

returns res vector

Norm

Euclidian norm of Vector

```
double Norm()
```

Parameters

returns Norm of vector elements

Normalize

Normalize vector (Making Norm = 1.0)

```
void Normalize()
```

NormInf

Infinity Norm of vector

```
double NormInf()
```

NormRel

Relative maximum element of vector, $\text{Max}(\text{---ai---}/(1 + \text{---ai---}))$

```
double NormRel()
```

operator

Clone vector operator, $\text{res} = +x$

```
static Vector operator +(Vector x)
```

Parameters

x Input vector

returns Clone of x

operator

Add vector operator, $\text{res} = y + x$

```
static Vector operator +(Vector y, Vector x)
```

Parameters

y y vector

x

operator

Negate vector, $\text{res} = -x$

```
static Vector operator -(Vector x)
```

Parameters

x x vector

operator

Sub vector operator, $\text{res} = y - x$

```
static Vector operator -(Vector y, Vector x)
```

Parameters

y y vector

x x vector

returns y - x

operator

Multiply operator $\text{res} = y' * x$

```
static double operator *(Vector y, Vector x)
```

Parameters

y y vector

x x vector

returns Inner product of vectors

operator

Matrix * vector operator, $\text{res} = A * x$

```
static Vector operator *(Matrix a, Vector x)
```

Parameters

a a matrix

x x vector

returns result vector

operator

Vector' * Matrix operator, $\text{res} = x' * A$

```
static Vector operator *(Vector x, Matrix a)
```

Parameters

x x vector

a a matrix

returns res vector

operator

Multiply scalar and Vector operator, $\text{res}[i] = \text{alpha} * x[i]$

```
static Vector operator *(double alpha, Vector x)
```

Parameters

alpha constant

x x vector

returns res vector

operator

Multiply scalar and Vector operator, $\text{res}[i] = x[i] * \text{alpha}$

```
static Vector operator *(Vector x, double alpha)
```

Parameters

x x vector

alpha constant

returns res vector

operator

Divide vector with scalar, $\text{res}[i] = x[i]/\text{alpha}$

```
static Vector operator /(Vector x, double alpha)
```

Parameters

x	x vector
alpha	constant
returns	res vector

operator

Add scalar to vector, $\text{res}[i] = x[i] + \text{alpha}$

```
static Vector operator +(Vector x, double alpha)
```

Parameters

x	x vector
alpha	constant
returns	res vector

operator

Add scalar to vector, $\text{res}[i] = \text{alpha} + x[i]$

```
static Vector operator +(double alpha, Vector x)
```

Parameters

alpha	constant
x	x vector
returns	res vector

operator

Subtract scalar from vector $\text{res}[i] = x[i] - \text{alpha}$

```
static Vector operator -(Vector x, double alpha)
```

Parameters

x	x vector
alpha	constant
returns	res vector

operator

Subtract scalar from vector $\text{res}[i] = \text{alpha} - x[i]$

```
static Vector operator -(double alpha, Vector x)
```

Parameters

alpha	constant
x	x vector
returns	res vector

ProperZeroes

Convert small values to proper zeroes

```
void ProperZeroes()
```

Random

Random vector

```
static Vector Random(int n, double min, double max)
```

Parameters

n	Dimension
min	Minimum value of random elements
max	Maximum value of random elements
returns	Vector with random values

Random

Random vector

```
static Vector Random(Vector XMin, Vector XMax)
```

Parameters

XMin	Minimum values
XMax	Maximum values
returns	Vector with random values

Set

Set, insert subvector

```
void Set(int pos, Vector a)
```

Parameters

pos	Start position
a	Subvector

Set

Set subvector with increments

```
void Set(int pos, int incr, Vector a)
```

Parameters

pos	Start position
incr	Increment
a	Subvector

Show

Show vector on console

```
static void Show(string txt, Vector a)
```

Parameters

txt Text string

a Vector

Show

Show Vector

```
void Show(string txt)
```

Parameters

txt Text string

Sort

Sort elements in vector descending

```
void Sort()
```

Sub

Sub Vector, $y = y - x$

```
void Sub(Vector x)
```

Parameters

x x vector

Sub

Subtract constant from vector $y[i] = y[i] - a$

```
void Sub(double a)
```

Parameters

a constant

Sum

Get Sum of elements

```
double Sum()
```

Parameters

returns Sum of all element in vector

Variance

Get variance

`double Variance()`

Parameters

returns Variance of vector elements

3.3 MatrixForm*Enumeration* **MatrixForm**

Matrix Form Enumerator

`enum MatrixForm`

Fields

ConstantDiagonal	Matrix with constant diagonal elements
Diagonal	Diagonal matrix
General	General matrix
Null	Null matrix
Sparse	Sparse matrix in PARDISO format

Enumeration **MatrixOp**

Operator Enumerator, for BLAS, LAPACK routines

`enum MatrixOp`

Fields

C	Conjugate Transpose (complex)
N	No tranpose
T	Transpose

Enumeration **UPLO**

Upper or Lower part of Matrix for BLAS, LAPACK routines

`enum UPLO`

Fields

Lower	Lower triangular
Upper	Upper triangular

Enumeration **Side**

Side of matrix, for BLAS, LAPACK routines

```
enum Side
```

Fields

```
Left    Left side
Right   Right side
```

Enumeration **Diagonal**

Type of diagonal matrix, for BLAS, LAPACK routines

```
enum Diagonal
```

Fields

```
NonUnit  Unit matrix
Unit     Non unit matrix
```

Enumeration **SparseMatrixType**

Sparse Matrix type enumeration.

```
enum SparseMatrixType
```

Fields

```
RealUnsymmetrical      Real and unsymmetrical matrix
StructurallySymmetrical  Real and structurally symmetrical matrix
SymmetricalIndefinite    Real and symmetric indefinite matrix
SymmetricalPositiveDefinit  Real and symmetrical positive definite matrix
```

3.4 Matrix*Class* **Matrix**

Matrix class

```
[Serializable]
```

```
class Matrix : MBase<double> , ICommon<Matrix>
```

Constructors

```
Matrix()
: base(0)
```

Constructor default for serialization

`Matrix(int n) : base(n)`

Constructor square matrix (Form = Null)

Parameters

n Dimension

`Matrix(MatrixForm Form, int n) : base(Form, n)`

Constructor square matrix

Parameters

Form Matrix form

n Dimension

`Matrix(int n, int m) : base(n, m)`

Constructor n x m matrix, (Form = Null)

Parameters

n Rows

m Columns

`Matrix(MatrixForm Form, int n, int m): base(Form, n, m)`

Constructor n x m matrix

Parameters

Form Matrix form

n Rows

m Columns

`Matrix(int n, double val) : base(n, val)`

Constructor ConstantDiagonal Matrix (Form = ConstantDiagonal)

Parameters

n Dimension

val value

`Matrix(int n, int m, double[] vals): base(n, m, vals)`

Constructor general matrix from array of values

Parameters

n Rows

m Columns

vals Value array

`Matrix(Vector a) : base(a.values)`

Constructor Diagonal matrix (Form = Diagonal)

Parameters

a Vector of diagonal values

`Matrix(params double[] vals) : base(vals)`

Constructor Diagonal matrix (Form = Diagonal)

Parameters

vals array of values

Properties

Density

Matrix density (0.0 - 1.0) Null -i 0.0 General -i 1.0

```
double Density {get;}
```

MatrixForm

Matrix form (Null, ConstantDiagonal, Diagonal, Sparse or General) Moving from Null towards General preserves data Moving from General to Null, ConstantDiagonal and Diagonal are ignored. Moving from General to Sparse is executed

```
override MatrixForm Form {set; get;}
```

SMType

Sparse Matrix type

```
SparseMatrixType SMType {set; get;}
```

Methods

Add

Add, $B = B + A$

```
void Add(Matrix a)
```

Parameters

a Matrix to be added

AddScaled

Add scaled this = $\alpha * a + \text{this}$;

```
void AddScaled(double alpha, Matrix a)
```

Parameters

alpha Constant

a Input matrix

AddScaled

res = $\alpha * \text{this}$

```
Matrix AddScaled(double alpha)
```

Parameters

a
txt
alpha

AssertEqual

Assert equal, Show actual and expected matrices are not equal

```
static bool AssertEqual(string text, Matrix actual, Matrix expected)
```

Parameters

text text string
actual actual matrix
expected expected matrix
returns true if equal

EigenValues

Eigenvalues

```
static CVector EigenValues(Matrix a)
```

Parameters

a Input matrix
returns Complex vector with eigenvalues

Equal

Equal matrices, test whether two Matrices are equal

```
static bool Equal(Matrix a, Matrix b)
```

Parameters

a Input matrix a
b Input Matrix b
returns true if equal

Exp

Exponential function

```
static Matrix Exp(Matrix A)
```

Parameters

A

Gemm

General Matrix Multiplication $\text{this} = \alpha * \text{Op}(\text{a}) * \text{Op}(\text{b}) + \beta * \text{this}$

```
void Gemm(Matrix a, Matrix b, MatrixOp opra,
MatrixOp oprb, double alpha, double beta)
```

Parameters

a	input matrix a
b	input matrix b
opra	MatrixOp direct or transposed
oprb	MatrixOp direct or transposed
alpha	constant
beta	constant

Ger

Rank1 update of Matrix $\text{a} := \alpha * \text{x} * \text{y}' + \text{a}$

```
void Ger(Vector x, Vector y, double alpha)
```

Parameters

x	input vector x
y	input vector y
alpha	constant

Get

Get Submatrix

```
Matrix Get(int row, int n, int col, int m)
```

Parameters

row	Row position
n	Rows
col	Column position
m	Columns
returns	Submatrix

GetColumn

Get GetColumn

```
Vector GetColumn(int col)
```

Parameters

col	Column
returns	Column vector

GetColumn

Get GetColumn

```
Vector GetColumn(int col, int indx, int length)
```

Parameters

col	Column
indx	First row
length	Number of rows

GetRow

Get Row

```
Vector GetRow(int row)
```

Parameters

row	Row
returns	row vector

GetRow

Get Row

```
Vector GetRow(int row, int indx, int length)
```

Parameters

row	Row
indx	First Column position
length	Number of columns

Invert

Invert Matrix

```
static Matrix Invert(Matrix a)
```

Parameters

a

Mul

outer product $A = x*y'$

```
static Matrix Mul(Vector x, Vector y)
```

Parameters

x	input vector x
y	input vector y
returns	outer product matrix

MulAdd

Multiply Add $r = r + a*b$

```
void MulAdd(Matrix a, Matrix b)
```

Parameters

a Input matrix a

b input matrix b

MulTransposed

Multiply tranpose $res = a'*b$

```
static Matrix MulTransposed(Matrix a, Matrix b)
```

Parameters

a Input matrix a

b Input matrix b

returns $a'*b$

Norm

Frobenius Norm of matrix

```
double Norm()
```

Norm1

Norm 1 of Matrix

```
double Norm1()
```

NormInf

Norm Infinity of Matrix

```
double NormInf()
```

operator

Clone Matrix, $A = +B$

```
static Matrix operator +(Matrix a)
```

Parameters

a Input matrix

returns clone of input matrix

operator

Negate Matrix, $A = -B$

```
static Matrix operator -(Matrix a)
```

Parameters

a Input matrix

returns Negated input matrix

operator

Add binary operator, $Res = A + B$

```
static Matrix operator +(Matrix a, Matrix b)
```

Parameters

a A matrix

b B matrix

returns Sum of input matrices

operator

Sub binary operator, $Res = A - B$

```
static Matrix operator -(Matrix a, Matrix b)
```

Parameters

a A matrix

b B matrix

operator

Matrix multiplication operator, $Res = A * B$

```
static Matrix operator *(Matrix a, Matrix b)
```

Parameters

a Input matrix a

b Input matrix b

returns Product

operator

Multiply matrix with constant, $Res = \alpha * A$

```
static Matrix operator *(double alpha, Matrix a)
```

Parameters

alpha constant

a Input matrix

returns Result matrix

operator

Multiply matrix with constant, $\text{Res} = A * \alpha$

```
static Matrix operator *(Matrix a, double alpha)
```

Parameters

a Input matrix
alpha constant
returns Result matrix

operator

Divide matrix with constant, $\text{Res} = A/\alpha$

```
static Matrix operator /(Matrix a, double alpha)
```

Parameters

a Input matrix
alpha constant
returns Result matrix

PseudoInvert

Pseudo Invert

```
static Matrix PseudoInvert(Matrix a)
```

Parameters

a Input matrix
returns Result matrix

Random

Random Matrix

```
static Matrix Random(int n, int m, double min, double max)
```

Parameters

n Number of Rows
m Number of Columns
min Minimum value
max Maximum value
returns Random Matrix

Set

Set sub matrix

```
void Set(int row, int col, Matrix a)
```

Parameters

row Row position
col Column position
a sub matrix

SetColumn

Set Column

```
void SetColumn(int col, Vector Vals)
```

Parameters

col Column

Vals Vector of values

SetRow

Set Row

```
void SetRow(int row, Vector Vals)
```

Parameters

row Row

Vals Vector of values

Show

Show matrix

```
static void Show(string text, Matrix a)
```

Parameters

text text string

a Matrix a

Sub

Sub B = B - A

```
void Sub(Matrix a)
```

Parameters

a Matrix to be subtracted

SVD

SVD Dcomposition $A = U \cdot S \cdot V^T$

```
static void SVD(Matrix a, out Matrix U, out Vector S, out Matrix VT)
```

Parameters

a Input matrix A

U Left side othogonal matrix

S Singular Values vector

VT Righth side orthogonal matrix

ToSparse

Copy Matrix to Sparse Matrix. equivalent to `Sparse(SparseMatrixType.RealUnsymmetrical, UPLO.Lower, A)`

```
static Matrix ToSparse(Matrix A)
```

Parameters

A Input matrix
returns Sparse matrix

ToSparse

Copy Matrix to Sparse Matrix. equivalent to `Sparse(type, UPLO.Lower, A)`

```
static Matrix ToSparse(SparseMatrixType type, Matrix A)
```

Parameters

type Sparse matrix Type
A Input matrix
returns Sparse matrix

ToSparse

Copy Matrix to Sparse Matrix

```
static Matrix ToSparse(SparseMatrixType type, UPLO UpLo, Matrix A)
```

Parameters

type Sparse matrix Type
UpLo Position for values in symmetric values
A Input matrix
returns Sparse matrix

Transpose

Transpose matrix

```
static Matrix Transpose(Matrix a)
```

Parameters

a Input matrix
returns Transposed input matrix

Trmm

Matrix product with triangular matrix
Side = Left this := $\alpha * \text{Opr}(A) *$
this side = Right this : $\alpha * \text{this} * \text{Opr}(A)$

```
void Trmm(Side side, UPLO uplo, MatrixOp opr, double alpha, Matrix A)
```

Parameters

side	Position of A matrix
uplo	Upper or lower matrix
opr	Transpose operator
alpha	constant
A	input matrix

UnityMatrix

Unity matrix

```
static Matrix UnityMatrix(int n)
```

Parameters

n	Dimension
returns	Unity matrix

3.5 CVector

Class CVector

Complex vector

[Serializable]

```
class CVector : VBase<complex>
```

Constructors

```
CVector()
```

```
: base(0)
```

Constructor for serialization

```
CVector(int n) : base(n)
```

Constructor

Parameters

n	Dimension
---	-----------

```
CVector(int n, complex val) : base(n, val)
```

Constructor, with equal elements

Parameters

n	Dimension
val	complex value

```
CVector(params complex[] vals)
```

```
: base(vals)
```

Constructor, form array of complex values

Parameters

vals

```
CVector(List<complex> a) : base(a.Count)
```

Constructor from List of complex values

Parameters

a List of Complex values

```
CVector(Vector rv, Vector iv)
: base(rv.Dimension)
```

Constructor from real and imaginary values

Parameters

rv real parts

iv imaginary parts

Methods

Arg

Get Argument

```
Vector Arg()
```

Parameters

returns Vector of argument

AssertEqual

Assert equal, Show actual and expected if no equal

```
static bool AssertEqual(string text, CVector actual, CVector expected)
```

Parameters

text text string

actual actual value

expected expected values

Equal

Equal

```
static bool Equal(CVector a, CVector b)
```

Parameters

a Vector a

b Vector b

returns true if equal

EqualRoots

Equal Roots

```
static bool EqualRoots(CVector a, CVector b)
```

Parameters

a Vector a
b Vector b
returns true if equal

Get

Get subvector

```
CVector Get(int pos, int dim)
```

Parameters

pos
dim

IV

Get imaginary values

```
Vector IV()
```

Parameters

returns Vector of Imaginary values

Mod

Get moduli

```
Vector Mod()
```

Parameters

returns Vector of Moduli

operator

Clone vector, $res = + x$

```
static CVector operator +(CVector x)
```

Parameters

x Input vector

RV

Get real values

Vector RV()

Parameters

returns Vector of real values

Show

Show Complex Vector

```
static void Show(string txt, CVector a)
```

Parameters

txt text string

a complex vector

Sort

Sort elements in vector descenting according 1 according to real part, 2 according to imag part

```
void Sort()
```


Chapter 4

Block Vectors and Matrices

A block vector is a vector whose elements consist of vectors. A Block matrix is a matrix whose elements consist of matrices. Block vectors and block matrices are very convenient for two reasons. It simplifies the programming task, making it easier to configure matrices for Optimization tasks. MPC controllers are often described in literature using block matrices. Secondly *MPCMath* stores information about the structure of the block matrix and its sub matrices making it possible to exploit structures automatically during program execution. The MatrixForm enumeration is described in C.2.

There are some rules for block vectors and block matrices that must be obeyed. All sub matrices in a block matrix row, must have an equal number rows. All sub matrices in a block matrix column must have the same number of columns. The structure of a block matrix or block vector is recorded in a *Structure* C.1 object.

For a block vector the *Structure* object defines the number of sub vectors and the dimension of each sub vector. The code below defines a block vector *X* with three sub vectors. The dimension of the sub vectors are 3, 3 and 1. Three random sub vectors are stored in *X* and finally a sub vector, *xsub*, is retrieved from *X*

```
Structure strc = new Structure(3);
strc[0] = 3;
strc[1] = 2;
strc[2] = 1;

BVector X = new BVector(strc);
Structure.Show("x structure", X.Structure);

X[0] = Vector.Random(3, -100.0, 100.0);
X[1] = Vector.Random(2, -100.0, 100.0);
X[2] = Vector.Random(1, -100.0, 100.0);

BVector.Show("x", X);
```

```
Vector xsub = X[1];
Vector.Show("xsub", xsub);
```

The MPCMathConole output is

```
x structure
      3      2      1

x Vector
-60,3688 -48,1193 -98,3050 -22,4367 -49,2921 82,2724
xsub Vector
-22,4367 -49,2921
```

The following code defines a block matrix A with random sub matrices. It displays the structure of A and finally demonstrates a multiplication between a block matrix and a block vector.

```
Structure rowStr = Structure.Values(1, 2, 3, 1);
Structure colStr = Structure.Values(3, 2, 1);
BMatrix A = new BMatrix(MatrixForm.General, rowStr, colStr);

A[0, 0] = Matrix.Random(1, 3, -100.0, 100.0);
A[1, 1] = Matrix.UnityMatrix(2);
A[2, 1] = Matrix.Random(3, 2, -100.0, 100.0);

BMatrix.ShowStructure("A structure", A);
BMatrix.Show("A", A);

BVector R = A * X;
BVector.Show("R", R);
```

The MPCMathConole output is

```
A structure Matrix.Form General
General matrix[4][3]

General      Null      Null
Null         ConstantDiagonal Null
Null         General    Null
Null         Null       Null

A Matrix.Form General[7,6]
```

-23,0554	-85,4168	79,6318	0	0	0
0	0	0	1	0	0
0	0	0	0	1	0
0	0	0	-27,7048	-23,4950	0
0	0	0	19,7783	26,8926	0
0	0	0	34,7124	74,2146	0
0	0	0	0	0	0

R Vector

-1705,0789	-2,0829	90,7942	-2075,5061	2400,4981	6665,9496	0
------------	---------	---------	------------	-----------	-----------	---

Block vectors and block matrices can be created from vector dimensions or numbers of rows and columns. Then the block matrix builds the structure information, when the sub matrix object are stored in the matrix. In general its safer, and prevents a lot of indexing errors, if you one carefully plan the structure of the task and use *Structure* objects to define the block vectors and matrices.

4.1 Structure

Class Structure

Structure class

```
[Serializable]
class Structure
```

Constructors

`Structure()`

Default constructor for serialization

`Structure(int n)`

Constructor for basic structure

Parameters

n Dimension

`Structure(int n, int val)`

Structure with equal substructures

Parameters

n Dimension

val Substructure dimensions

Properties**Dimension**

Dimension

```
int Dimension {get;}
```

FlatDimension

Flat dimension

```
int FlatDimension {get;}
```

Indexer

Get or set value

```
int this[int p] {set; get;}
```

Methods**AssertEqual**

assert equal, Compare two Structures and output value if not equal

```
static void AssertEqual(string text, Structure actual, Structure expected)
```

Parameters

text	Text string
actual	Vector with actual values
expected	Vector with expected values

Equal

Test if two structures are equal

```
static bool Equal(Structure a, Structure b)
```

Parameters

a	Input a
b	Input b
returns	true if equal

operator

Merge structures (warning $a + b \neq b + a$)

```
static Structure operator +(Structure a, Structure b)
```

Parameters
 a Input a
 b Input b
 returns a+b

Show

Show Structure

```
static void Show(string txt, Structure a)
```

Parameters
 txt text string
 a Structure object

Values

Construct new Structure from array of values

```
static Structure Values(params int[] vals)
```

Parameters
 vals array of dimensions
 returns Structure

4.2 BVector

Class **BVector**

Block Vector class

```
[Serializable]  
class BVector : Vector<Vector>
```

Constructors

```
BVector()  
: base()
```

Constructor default

```
BVector(int n) : base(n)
```

Constructor

Parameters
 n Dimension

```
BVector(int n, Vector val) : base(n, val)
```

Constructor vector with equal values

Parameters

n Dimension
value Value

```
BVector(params Vector[] vals)
: base(vals)
```

Generate vector from array of Vectors

Parameters

vals Vector array

```
BVector(Vector<Vector> a)
: base(a.Dimension)
```

Constructor BVector from *Vector* < *Vector* >

Parameters

a

```
BVector(Structure s)
```

Construct BVector from structure

Parameters

s Structure

```
BVector(Structure s, double val)
```

Construct BVector from structure with equal values

Parameters

s Structue
val value

```
BVector(Vector a)
: base(1)
```

Constructor, Wrap Vector a in block vector

Parameters

a

Methods

DivElements

Divide Vector elements, $z[i] = y[i]/x[i]$

```
static BVector DivElements(BVector y, BVector x)
```

Parameters

y Input y
x Input X
returns $y[i]/x[i]$

Flatten

Flatten BVector to plain Vector

```
static Vector Flatten(BVector a)
```

Parameters

a Input BVector

returns Vector

Get

Get subvector

```
new BVector Get(int pos, int dim)
```

Parameters

pos Start position

dim Dimension

Get

Get subvectorwith increments

```
new BVector Get(int pos,int incr, int dim)
```

Parameters

pos Start position

incr Position incremenst

dim Dimension

Max

Maximum value

```
double Max()
```

Parameters

returns Maximum value of all elements in member vectors

Min

Minimum value

```
double Min()
```

Parameters

returns Minimum value of all elements in member vectors

MulElements

Multiply Vecor elements, $z[i] = y[i]*x[i]$

```
static BVector MulElements(BVector y, BVector x)
```

Parameters

y Input y

x Input X

returns $y[i]*x[i]$

Norm

Euclidian norm of Vector

```
double Norm()
```

Parameters

returns Norm of vector elements

operator

Clone, unary + operator

```
static BVector operator +(BVector a)
```

Parameters

a input a

returns +a

operator

Negate, unary -

```
static BVector operator -(BVector a)
```

Parameters

a Input a

returns -a

operator

Binary add operator

```
static BVector operator +(BVector a, BVector b)
```

Parameters

a Input a

b Input b

returns $a+b$

operator

Binary Sub

```
static BVector operator -(BVector a, BVector b)
```

Parameters

a Input a

b Input b

returns a-b

operator

Multiply constant and BVector

```
static BVector operator *(double alpha, BVector a)
```

Parameters

alpha constant

a Input a

returns alpha*a

operator

Multiply BVector and constant

```
static BVector operator *(BVector a, double alpha)
```

Parameters

a Input a

alpha Constant

returns alpha*a

operator

Divide BVector and constant

```
static BVector operator /(BVector a, double alpha)
```

Parameters

alpha Constant

a Input a

returns a/alpha

operator

Multiply , inner product

```
static double operator *(BVector a, BVector b)
```

Parameters

a Input a

b Input b

returns a*b

operator

Matrix * vector operator

```
static BVector operator *(BMatrix a, BVector x)
```

Parameters

a Input BMatrix a
 x Input BVector x
 returns a*x

operator

Vector * matrix operator

```
static BVector operator *(BVector x, BMatrix a)
```

Parameters

x BVector x
 a BMatrix a
 returns x*a

Random

Random BVector

```
static BVector Random(Structure strc, double min, double max)
```

Parameters

strc Structure
 min Minimum value
 max Maximum value

Restructure

Restructure Vector from plain Vector

```
static BVector Restructure(Structure strct, Vector a)
```

Parameters

strct Structure
 a Input vecto
 returns Block vector

Show

Show

```
static void Show(string text, BVector a)
```

Parameters

text text string
 a BVector object

Show

Show first elements of each subvector

```
static void Show(string text, BVector a, int dim)
```

Parameters

text text string

a BVector object

dim Number of elements to show from each subvector

4.3 BMatrix*Class* **BMatrix**

Blok Matrix class

```
[Serializable]
```

```
class BMatrix : Matrix<Matrix>
```

Constructors

```
BMatrix()
```

```
: base()
```

Constructor default

```
BMatrix(int n)
```

```
: base(n)
```

Constructor square matrix as Null matrix

Parameters

n Dimension

```
BMatrix(MatrixForm Form, int n)
```

```
: base(Form, n)
```

Constructor square matrix

Parameters

Form Matrix form

n Dimension

```
BMatrix(int n, int m)
```

```
: base(n, m)
```

Constructor n x m matrix as Null matrix

Parameters

n Rows

m Columns

```
BMatrix(MatrixForm Form, int n, int m)
: base(Form, n, m)
```

Constructor n x m matrix

Parameters
 Form Matrix Form
 n Rows
 m Columns

```
BMatrix(int n, Matrix val)
: base(n, val)
```

Constructor ConstantDiagonal Matrix

Parameters
 n Dimension
 val Value

```
BMatrix(BVector a)
: base(a.Dimension)
```

Constructor Diagonal matrix, from BVector

Parameters
 a Diagonal elements

```
BMatrix(params Matrix[] vals)
: base(vals)
```

Constructor Diagonal matrix

Parameters
 vals Diagonal sub matrices

```
BMatrix(Structure Structure) : base(Structure.Dimension)
```

Constructor for square matrix as null matrix

Parameters
 Structure Structure for rows and Columns

```
BMatrix(MatrixForm form, Structure Structure)
: base(form, Structure.Dimension)
```

Constructor for square matrix

Parameters
 form Matrix Form
 Structure Structure for rows and Columns

```
BMatrix(Structure RowStructure, Structure ColStructure)
: base(RowStructure.Dimension, ColStructure.Dimension)
```

Constructor for general matrix as Null matrix

Parameters
 RowStructure Row Structure
 ColStructure Column Structure

```
BMatrix(MatrixForm form, Structure RowStructure, Structure ColStructure)
: base(RowStructure.Dimension, ColStructure.Dimension)
```

Constructor for general matrix

Parameters

form	Matrix Form
RowStructure	Row Structure
ColStructure	Column Structure

```
BMatrix(Matrix<Matrix> a)
: base(a.form, a.rows, a.columns)
```

Constructor BMatrix from generic Matrix; Matrix; Obs Values and structures are copied by reference, not cloned

Parameters

a Input a

```
BMatrix(Matrix a)
: base(1)
```

Constructor , Wrap matrix a in Block Matrix

Parameters

a Input matrix

Methods

Allocate

Allocate Matrix element. If null element a new matrix is allocated. If element exist matrix form is changed:

```
void Allocate(int row, int col)
```

Parameters

row
col
form

Allocate

Allocate Matrix element. (Empty elements of a Block matrix are nor allocated by the constructor).

```
void Allocate(int row, int col, MatrixForm form)
```

Parameters

row
col
form

Flatten

Flatten BMatrix

```
static Matrix Flatten(BMatrix a)
```

Parameters

a Input Bmatrix
 returns Plain Matrix

Get

Get submatrix

```
new BMatrix Get(int row, int rows, int col, int cols)
```

Parameters

row Row position
 rows Number of Rows
 col Column position
 cols Number of Columns
 returns Submatrix

MulTransposed

Multiply transposed $a'b$

```
static BMatrix MulTransposed(BMatrix a, BMatrix b)
```

Parameters

a Input a
 b Input b
 returns $a'b$

operator

Clone, + operator

```
static BMatrix operator +(BMatrix a)
```

Parameters

a Input a
 returns +a

operator

Negate, - operator

```
static BMatrix operator -(BMatrix a)
```

Parameters

a Input a
 returns -a

operator

Binary add

```
static BMatrix operator +(BMatrix a, BMatrix b)
```

Parameters

a	Input a
b	Input b
returns	a+b

operator

Binary sub

```
static BMatrix operator -(BMatrix a, BMatrix b)
```

Parameters

a	Input a
b	Input b
returns	a-b

operator

Multiply

```
static BMatrix operator *(BMatrix a, BMatrix b)
```

Parameters

a	Input a
b	Input b
returns	a*b

operator

Multiply with constant

```
static BMatrix operator *(double alpha, BMatrix a)
```

Parameters

alpha	constant
a	Input a
returns	alpha*a

operator

Multiply with constant

```
static BMatrix operator *(BMatrix a, double alpha)
```

Parameters

a	Input a
alpha	Constant

operator

Divede by constant

```
static BMatrix operator /(BMatrix a, double alpha)
```

Parameters

a	Input a
alpha	constant
returns	a/alpha

OuterProduct

Outer Product of two BVectors

```
static BMatrix OuterProduct(BVector a, BVector b)
```

Parameters

a	Input a
b	Input b
returns	BMatrix

Random

Random Block Matrix

```
static BMatrix Random(Structure rowStrc, Structure colStrc, double min, double max)
```

Parameters

rowStrc	Row structure
colStrc	Column structure
min	Minimum value
max	Maximum value
returns	Random BMatrix

Restructure

Restructure Matrix

```
static BMatrix Restructure(Structure rowStr, Structure colStr, Matrix a)
```

Parameters

rowStr	Row Structure
colStr	Columns structure
a	Input matrix
returns	BMatrix

Show

Show

```
static void Show(string text, BMatrix a)
```

Parameters

text Text string
a BMatrix object

ShowStructure

Show matrix structure

```
static void ShowStructure(string text, BMatrix A)
```

Parameters

text text string
A BMatrix object

ToSparse

Copy Block Matrix to sparse matrix Equivalent to ToSparse(SparseMatrixType.RealUnsymmetrical, UPLO.Lower, A);

```
static Matrix ToSparse(BMatrix A)
```

Parameters

A Input matrix
returns Sparse matrix

ToSparse

Copy Block Matrix to sparse matrix Equivalent to ToSparse(type, UPLO.Lower, A);

```
static Matrix ToSparse(SparseMatrixType type, BMatrix A)
```

Parameters

type Sparse matrix Type
A Input matrix
returns Sparse matrix

ToSparse

Copy Block Matrix to sparse matrix

```
static Matrix ToSparse(SparseMatrixType type, UPLO UpLo, BMatrix A)
```

Parameters

type Sparse matrix Type
UpLo Position for values in symmetric values
A Input matrix
returns Sparse matrix

Transpose

Transpose

```
static BMatrix Transpose(BMatrix a)
```

Parameters

a Input a

returns a'

Chapter 5

Generic Vectors and Matrices

The main part of the block vector 4.2 and block matrix 4.3 functionality is build using the generic Vector and generic matrices classes.

Generic vector and matrices are populated with objects that implements the interface *ICommon* 5.1 and a default new T() constructor. In *MPCMath* this is the case for the Classes complex ??, Vector 3.2, Matrix 3.4. For the classes ArxModel 10.1, LinearFilter 9.1, TransferFunction 6.1 there is a limited implementation of ICommon, that does not support calculations.

The following code demonstrates generic vector and matrices for complex objects.

```
int dim = 4;
Vector<complex> x = new Vector<complex>(dim);
Matrix<complex> A =
    new Matrix<complex>(MatrixForm.General, dim, dim);

// fill in some crazy numbers
for (int row = 0; row < dim; row++)
{
    x[row] = complex.Sqrt(-row);
    for (int col = 0; col < dim; col++)
    {
        A[row, col] = new complex(row, col);
    }
}

Vector<complex>.Show("x", x);
Matrix<complex>.Show("A", A);

// Matrix calculation
Vector<complex> Res = A * x;
```

```
Vector<complex>.Show("Res", Res);
```

The console output is

```
Test Generic Matrix x MPCMath.Vector'1[MPCMath.complex] [0] 0,0000 + i*
0,0000
[1] 0,0000 + i* 1,0000
[2] 0,0000 + i* 1,4142
[3] 0,0000 + i* 1,7321
A Matrix.Form General Size [4,4] Pos [0,0] 0,0000 + i* 0,0000
Pos [0,1] 0,0000 + i* 1,0000
Pos [0,2] 0,0000 + i* 2,0000
Pos [0,3] 0,0000 + i* 3,0000
Pos [1,0] 1,0000 + i* 0,0000
Pos [1,1] 1,0000 + i* 1,0000
Pos [1,2] 1,0000 + i* 2,0000
Pos [1,3] 1,0000 + i* 3,0000
Pos [2,0] 2,0000 + i* 0,0000
Pos [2,1] 2,0000 + i* 1,0000
Pos [2,2] 2,0000 + i* 2,0000
Pos [2,3] 2,0000 + i* 3,0000
Pos [3,0] 3,0000 + i* 0,0000
Pos [3,1] 3,0000 + i* 1,0000
Pos [3,2] 3,0000 + i* 2,0000
Pos [3,3] 3,0000 + i* 3,0000
Res MPCMath.Vector'1[MPCMath.complex] [0] -9,0246 + i* 0,0000
[1] -9,0246 + i* 4,1463
[2] -9,0246 + i* 8,2925
[3] -9,0246 + i* 12,4388
```

The generic Show routine cannot produce compact printouts.

5.1 ICommon

Interface ICommon

Common functions for elements in Generic Vectors and Matrices

```
interface ICommon<T>
```

Properties**Columns**

Columns

```
int Columns {get;}
```

Form

Object form

```
MatrixForm Form {get;}
```

Rows

Rows

```
int Rows {get;}
```

Methods**AddScaled**

Add/Clone +, alpha * this

```
T AddScaled(double alpha)
```

Parameters

alpha	constant
-------	----------

AddScaled

Add this + alpha * a

```
T AddScaled(double alpha, T a)
```

Parameters

alpha	constant
a	object

Equal

Equal

```
bool Equal(T a)
```

Parameters

a	Compared to
returns	true if equal

MulAdd

MulAdd this = this + a x b

```
void MulAdd(T a, T b)
```

Parameters

a object a

b object b

returns Result object

Restructure

Restructure Object

```
void Restructure(int Rows, int Cols)
```

Parameters

form

Rows

Cols

Show

Show object

```
void Show(string txt)
```

Parameters

txt text string

a object

TMulAdd

Transpose a and MulAdd this = this + a' x b

```
void TMulAdd(T a, T b)
```

Parameters

a object a

b object b

Transpose

Transpose

```
T Transpose()
```

Parameters

returns Transposed object

Enumeration **ObjectType**

Object type

`enum ObjectType`

Fields

Element

Matrix

Vector

5.2 TVector*Class* **Vector**

Generic Vector class

`[Serializable]``class Vector<T> : VBase<T>, ICommon<VBase<T>> where T : ICommon<T>, new()`**Constructors**`Vector(int n)``: base(n)`

Constructor

Parameters

n

`Vector(int n, T val)``: base(n, val)`

vector with equal values

Parameters

n

val

`Vector(params T[] vals): base(vals)`

Constructor

Parameters

vals

`Vector(Structure s):base(s.Dimension)`

Construct Vector from structure

Parameters

s

Properties**T**

Set or get matrix value

```
override T this[int pos] {set; get;}
```

Methods**AssertEqual**

assert equal, Show if not equal

```
static void AssertEqual(string text, Vector<T> actual, Vector<T> expected)
```

Parameters

text
actual
expected

Equal

Equal

```
static bool Equal(Vector<T> y, Vector<T> x)
```

Parameters

y Vector y
x Vector x
returns true if equal

Get

Get subvector

```
Vector<T> Get(int pos, int dim )
```

Parameters

pos Position
dim Dimension
returns Subvector

Get

Get subvector with increment

```
Vector<T> Get(int pos, int incr, int dim)
```

Parameters

pos Position
incr Increment
dim Dimension
returns Subvector

operator

Unary add/clone operator +

```
static Vector<T> operator +(Vector<T> a)
```

Parameters

a Input a

returns +a

operator

Unary Sub/clone operator -

```
static Vector<T> operator -(Vector<T> a)
```

Parameters

a Input a

returns -a

operator

Add operator +

```
static Vector<T> operator +(Vector<T> a, Vector<T> b)
```

Parameters

a Input a

b Input b

returns a+b

operator

Vector * Vector , inner product

```
static Vector<T> operator *(Vector<T> a, Vector<T> b)
```

Parameters

a Input a

b Input b

returns a+b

operator

Vector * Matrix operator

```
static Vector<T> operator *(Matrix<T> a, Vector<T> x)
```

Parameters

a Input a

x Input b

returns a*x

operator

Matrix * Vector operator

```
static Vector<T> operator *(Vector<T> x, Matrix<T> a)
```

Parameters

a Input a

x Input b

returns x'*a

operator

Scalar vector multiply

```
static Vector<T> operator *(double alpha, Vector<T>a)
```

Parameters

alpha constant

a Input a

returns alpha*a

operator

Scalar vector multiply

```
static Vector<T> operator *(Vector<T> a, double alpha)
```

Parameters

a Input a

alpha constant

returns alpha*a

operator

Vector Scalar divide

```
static Vector<T> operator /(Vector<T> a, double alpha)
```

Parameters

alpha constant

a Input a

operator

Sub operator -

```
static Vector<T> operator -(Vector<T> a, Vector<T> b)
```

Parameters

a Input a

b Input b

returns a-b

Set

Set subvector

```
void Set(int pos , Vector<T>a)
```

Parameters

pos position

a sub vector

Set

Set subvector with increments

```
void Set(int pos, int incr ,Vector<T> a)
```

Parameters

pos position

incr increment

a sub vector

Show

Show

```
static void Show(string txt, Vector<T> a)
```

Parameters

txt text string

a Object

5.3 TMatrix*Class* **Matrix**

Generic Matrix class

[Serializable]

```
class Matrix<T>: MBase<T>, ICommon<Matrix<T>> where T : ICommon<T>, new()
```

Constructors

```
Matrix(int n)
```

```
: base(n)
```

Constructor square matrix, null form

Parameters

n Dimension

```
Matrix(MatrixForm Form, int n)
```

```
: base(Form, n)
```

Constructor square matrix

Parameters
 Form Matrix Form
 n Dimension

```
Matrix(int n, int m)
: base(n, m)
```

Constructor n x m matrix

Parameters
 n
 m

```
Matrix(MatrixForm Form, int n, int m)
: base(Form, n, m)
```

Constructor n x m matrix

Parameters
 n Rows
 m Columns

```
Matrix(int n, T val)
: base(n, val)
```

Constructor ConstantDiagonal Matrix

Parameters
 n Dimension
 val value of all diagonal elements

```
Matrix(Vector<T> a)
: base(a.values)
```

Constructor Diagonal matrix

Parameters
 a Vector with diagonal values

```
Matrix(params T[] vals)
: base(vals)
```

Constructor Diagonal matrix

Parameters
 vals Array of diagonal values

```
Matrix(Structure Structure) : base(Structure.Dimension)
```

Constructor for square matrix Null matrix

Parameters
 Structure Structure

```
Matrix(MatrixForm form, Structure Structure)
: base(form, Structure.Dimension)
```

Constructor for square matrix

Parameters
 form Matrix Form
 Structure Structure

```
Matrix(Structure RowStructure, Structure ColStructure)
: base(RowStructure.Dimension, ColStructure.Dimension)
```

Constructor, null matrix

Parameters

Structure Structure

```
Matrix(MatrixForm form, Structure RowStructure, Structure ColStructure)
: base(RowStructure.Dimension, ColStructure.Dimension)
```

Constructor

Parameters

form	Matrix Form
RowStructure	Row structure
ColStructure	Column structure

Properties

T

Set or get matrix value

```
override T this[int row, int col] {set; get;}
```

Methods

Add

Add $r = r + a$

```
void Add(Matrix<T> a)
```

Parameters

a Input a

AddScaled

Add scaled $r = \alpha * a + r$;

```
void AddScaled(double alpha, Matrix<T> a)
```

Parameters

alpha	constant
a	Input a

AssertEqual

Assert equal, Show if a not equal to b

```
static bool AssertEqual(string text, Matrix<T> actual, Matrix<T> expected)
```

Parameters

text
actual
expected

Equal

Equal generic Matrix

```
static bool Equal(Matrix<T> a, Matrix<T> b)
```

Parameters

a Input a
b Input b
returns true if a equal to b

Get

Get submatrix

```
Matrix<T> Get(int row, int col, int rows, int cols)
```

Parameters

row Row position
col Column position
rows Number of rows
cols Number of columns
returns Sub matrix

MulAdd

MulAdd $r = rs + a*b$

```
void MulAdd(Matrix<T> a, Matrix<T> b)
```

Parameters

a Input a
b Input b

MulTransposed

Multiply Transposed $res = a'*b$

```
static Matrix<T> MulTransposed(Matrix<T> a, Matrix<T> b)
```

Parameters

a Input a
b Input b
returns $a'*b$

operator

Clone Matrix

```
static Matrix<T> operator +(Matrix<T> a)
```

Parameters

a Input a
returns -a

operator

Negate Matrix

```
static Matrix<T> operator -(Matrix<T> a)
```

Parameters

a Input a

returns -a

operator

Add binary operator

```
static Matrix<T> operator +(Matrix<T> a, Matrix<T> b)
```

Parameters

a Input a

b Input b

returns a+b

operator

Sub binary operator

```
static Matrix<T> operator -(Matrix<T> a, Matrix<T> b)
```

Parameters

a Input a

b Input b

returns a+b

operator

Matrix constant multiplikation

```
static Matrix<T> operator *(Matrix<T> a, double alpha)
```

Parameters

a Input a

alpha constant

returns alpha*a

operator

Constrant matrix multiplikation

```
static Matrix<T> operator *(double alpha, Matrix<T> a)
```

Parameters

alpha constant

a Input a

returns alpha*a

operator

Matrix constant division

```
static Matrix<T> operator /(Matrix<T> a, double alpha)
```

Parameters

a Input a
alpha constant
returns a/alpha

operator

Matrix multiply operator

```
static Matrix<T> operator *(Matrix<T> a, Matrix<T> b)
```

Parameters

a Input a
b Input b
returns a*b

Set

Set submatrix

```
void Set(int row, int col, Matrix<T> a)
```

Parameters

row Row Position
col Column position
a Sub matrix

Show

Show matrix

```
static void Show(string text, Matrix<T> a)
```

Parameters

text
A

Sub

Sub, $r = r - a$

```
void Sub(Matrix<T> a)
```

Parameters

a Input a

TMulAdd

TMulAdd $r = r + a * b$

```
void TMulAdd(Matrix<T> a, Matrix<T> b)
```

Parameters

a Input a

b Input b

Transpose

Transpose

```
static Matrix<T> Transpose(Matrix<T> a)
```

Parameters

a Input a

returns a'

Chapter 6

Transfer Functions

Transfer Functions are the classical way to describe the dynamics of a linear system. Creating transfer functions using delay and time constants are an intuitive way to specify process dynamic. The transfer function can be used directly or the can be used to as input to Linear Filter, which are the discrete form of transfer functions.

The use of the TransferFunction class can be illustrated by a simple example. The example shows the closed loop response for a PI controller controlling a Plant describes as a second order systems.

The *Plant* is described by the transfer function

$$Plant(s) = \frac{1}{(\tau s)^2 + 2\sigma s + 1}$$

The PI *Controller* is defined by

$$Controller(s) = P(1 + \frac{1}{\tau_i s})$$

where P is the controller gain and τ_i the integration time.

```
double gain = 1.0;
double delay = 0.0;
double tau = 10.0;
double sigma = 0.5;
// second order process
TransferFunction Plant = new TransferFunction(gain, delay, tau, sigma);
double P = 3.0;
double TI = 25.0;
// Integrator
TransferFunction Integrator = new TransferFunction(1.0 / TI, 0.0);
TransferFunction Controller = P * (1.0 + Integrator);
```

creates the *Plant* and *controller* objects.

The closed loop dynamic G_{cl} is defined by

$$G_{cl}(s) = \frac{Controller(s)Plant(s)}{1 + Controller(s)Plant(s)}$$

```
// Closed loop dynamics
TransferFunction Gcl = Controller * Plant / (1.0 + Controller * Plant);

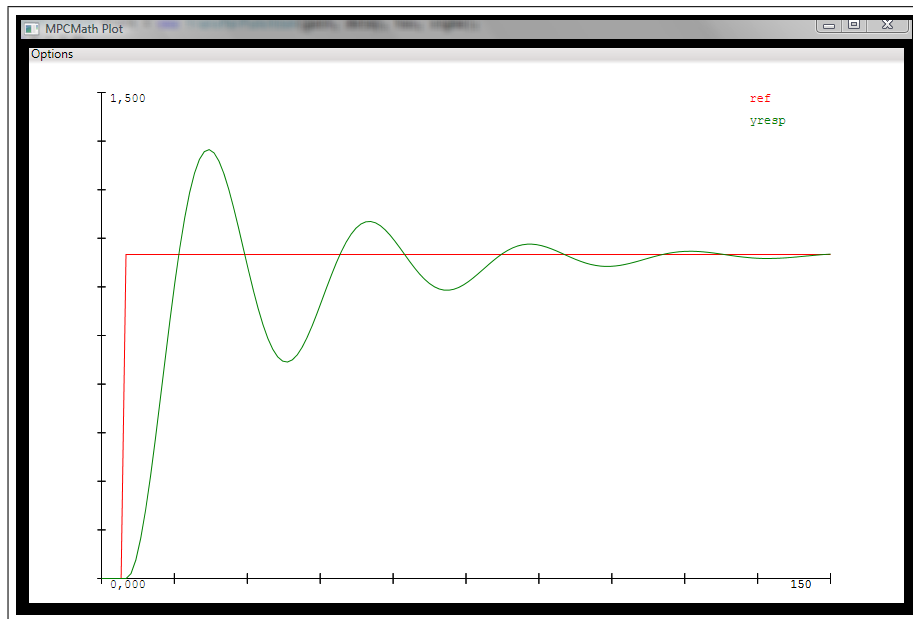
// reference signal, step after 5 time units
TransferFunction Ref = new TransferFunction(1.0, 5.0);
// Closed loop response
TransferFunction Y = Gcl * Ref;
```

creates the transfer function of the closed loop dynamic G_{cl} and the a reference signal Ref (a step delayed 5 seconds) and the response signal Y.

Plotting of the Ref and Y signal are performed by the code

```
int steps = 150;
Vector yresp = new Vector(steps);
Vector Reference = new Vector(steps);
for (int step = 0; step < steps; step++)
{
    Reference[step] = Ref.Value(step);
    yresp[step] = Y.Value(step);
}

double max = 1.5;
console.Plot(new Plot(
    new PlotSeries("ref", 0.0, max, SeriesColor.Red, Reference),
    new PlotSeries("yresp", 0.0, max, SeriesColor.Green, yresp)));
}
```



6.1 TransferFunction

Class TransferFunction

Linear Transfer Function $G(S) = e^{-\theta S} \frac{R(S)}{P(S)}$

```
[Serializable]
class TransferFunction : ICommon<TransferFunction>
```

Constructors

```
TransferFunction()
: this(0.0, 0.0)
```

Default constructor

```
TransferFunction(double Gain, double Delay, CVector Roots)
```

TransferFunction Constructor $G(s) = \gamma e^{-\theta s} \frac{1}{\prod(s-p_i)}$

Parameters

Gain	Gain γ
Delay	Delay θ
Roots	Roots p_1

```
TransferFunction(double Gain, double Delay, CVector Roots, CVector Zeroes)
```

TransferFunction Constructor $G(s) = \gamma e^{-\theta s} \frac{\Pi(s-r_i)}{\Pi(s-p_i)}$

Parameters

Gain Gain γ
 Delay Delay θ
 Roots Roots p_i
 Zeroes Zeroes r_i

TransferFunction(double Delay, Polynomial P)

TransferFunction constructor $G(s) = e^{-\theta s} \frac{1}{P(s)}$

Parameters

Delay Delay θ
 P Root polynomial $P(s)$

TransferFunction(double Delay, Polynomial P, Polynomial R)

TransferFunction constructor $G(s) = e^{-\theta s} \frac{R(s)}{P(s)}$

Parameters

Delay Delay θ
 P Root polynomial $P(s)$
 R Zero polynomial $R(s)$

TransferFunction(double Delay)

Constructor Transfer function Impulse $G(s) = e^{-\theta s}$

Parameters

Delay Delay θ

TransferFunction(double Gain, double Delay)

Constructor Transfer function, Step $G(s) = \gamma e^{-\theta s} / s$

Parameters

Gain Gain γ
 Delay Delay θ

TransferFunction(double Gain, double Delay, double Tau)

constructor Transfer function, one pole $G(s) = \gamma e^{-\theta s} / (\tau s + 1)$

Parameters

Gain Gain γ
 Delay Delay θ
 Tau time constant $\tau(\tau \neq 0)$

TransferFunction(double Gain, double Delay, double Tau, double Sigma)

constructor Transfer function, two poles $G(s) = \gamma e^{-\theta s} / ((\tau s)^2 + 2\sigma\tau s + 1)$

Parameters

Gain Gain γ
 Delay Delay θ
 Tau Time constant $\tau(\tau \neq 0)$
 Sigma Damping σ

Properties**Delay**

Transfer function Delay

```
double Delay {set; get;}
```

Gain

Transfer function Gain

```
double Gain {set; get;}
```

P

Denominator Polynomial $G(s) = e^{-\theta s} \frac{R(s)}{P(s)}$

```
Polynomial P {get;}
```

R

Numerator Polynomial $G(s) = e^{-\theta s} \frac{R(s)}{P(s)}$

```
Polynomial R {get;}
```

Roots

Transfer function Roots

```
CVector Roots {get;}
```

Zeroes

Transfer function Zeroes

```
CVector Zeroes {get;}
```

Methods**Add**

Add transfer functions

```
static TransferFunction Add(TransferFunction a, TransferFunction b)
```

Parameters

a input a

b input b

returns a+b

AssertEqual

Assert equal, Show transfer function if not equal

```
static void AssertEqual(string text,  
TransferFunction actual, TransferFunction expected)
```

Parameters

text	text string
actual	actual transfer function
expected	expected transfer function

Clone

Clone Transfer function

```
static TransferFunction Clone(TransferFunction a)
```

Parameters

a	input a
returns	Clone TransferFunction

Div

Divide transferfunctions

```
static TransferFunction Div(TransferFunction a, TransferFunction b)
```

Parameters

a	input a
b	input b
returns	a/b

Equal

Equal function

```
static bool Equal(TransferFunction a, TransferFunction b)
```

Parameters

a	Input a
b	Input b
returns	true if a equal b

Mul

Multiply transferfunctions

```
static TransferFunction Mul(TransferFunction a, TransferFunction b)
```

Parameters

a	input a
b	input b
returns	a*b

Mul

Multiply Transferfunction with constant

```
static TransferFunction Mul(double a, TransferFunction b)
```

Parameters

a constant

b input b

returns a*b

operator

TransferFunction clone operator

```
static TransferFunction operator +(TransferFunction a)
```

Parameters

a input TransferFunction

returns Cloned TransferFunction

operator

TransferFunction Negate operator

```
static TransferFunction operator -(TransferFunction a)
```

Parameters

a Input TransferFunction

returns Negated TransferFunction

operator

Transfer function Add operator MPCMath requires the two Transferfunction a
b to have equal delay !

```
static TransferFunction operator +(TransferFunction a, TransferFunction b)
```

Parameters

a input a

b input b

returns a+b

operator

Transfer function Add operator MPCMath requires the two Transferfunction b
to have equal delay = 0!

```
static TransferFunction operator +(double a, TransferFunction b)
```

Parameters

a constant

b TransferFunction

returns TransferFunction

operator

Transfer function Add operator MPCMath requires the two Transferfunction a to have equal delay = 0!

```
static TransferFunction operator +(TransferFunction a, double b)
```

Parameters

a TransferFunction

b constant

returns TransferFunction

operator

Transferfunction Sub

```
static TransferFunction operator -(TransferFunction a, TransferFunction b)
```

Parameters

a input a

b input b

returns a-b

operator

Multiply transferfunctions operator

```
static TransferFunction operator *(TransferFunction a, TransferFunction b)
```

Parameters

a input a

b input b

returns a*b

operator

Multiply TransferFunction with Constant operator

```
static TransferFunction operator *(double a, TransferFunction b)
```

Parameters

a constant

b Input b

returns a*b

operator

Multiply TransferFunction with Constant operator

```
static TransferFunction operator *(TransferFunction a, double b)
```

Parameters

a input a

b constant

returns a*b

operator

Divide TransferFunction with constant operator

```
static TransferFunction operator /(TransferFunction a, double b)
```

Parameters

a input a
b constant
returns a/b

operator

Divide transferfunctions operator

```
static TransferFunction operator /(TransferFunction a, TransferFunction b)
```

Parameters

a input a
b input b
returns a/b

Show

Show Transfer function

```
static void Show(string txt, TransferFunction a)
```

Parameters

txt text string
a TransferFunction

Sub

Transferfunction Sub

```
static TransferFunction Sub(TransferFunction a, TransferFunction b)
```

Parameters

a input a
b input b
returns a-b

TauForm

”Constructor” for Tau form $G(s) = \gamma e^{-\theta s} \frac{1}{\prod(\tau_i s + 1)}$

```
static TransferFunction TauForm(double Gain, double Delay, Vector Taus)
```

Parameters

Gain Gain γ
Delay Delay θ
Tau Time constants τ
returns Transfer function

TauForm

”Constructor” for Tau form $G(s) = \gamma e^{-\theta s} \frac{\prod(z_i s + 1)}{\prod(\tau_i s + 1)}$

```
static TransferFunction TauForm(double Gain, double Delay, Vector Taus, Vector Zeroes)
```

Parameters

Gain Gain γ

Delay Delay θ

Tau time constants τ

zeroes

returns Transfer function

Value

Time response of Transfer function

```
double Value(double t)
```

Parameters

t time

returns f(t)

Chapter 7

State Space Models

The *StateSpaceModel* class implements the discrete state space model

$$\mathbf{x}_{k+1} = A\mathbf{x}_k + Bu_k + K\epsilon_k + off \quad (7.1a)$$

$$\mathbf{y}_k = C\mathbf{x}_k + \epsilon_k \quad (7.1b)$$

where

x_k	Process states at time k	Dimension nx
u_k	Process inputs at time k	Dimension nu
ϵ_k	Innovation at time k	Dimension ny
y_k	Process outputs at time k	Dimension ny

The matrix K and the innovation ϵ_k implements a Kalman filter, estimating the process state based on measurements of the plant outputs y_k . The innovation at sample k is defined by

$$\epsilon_k = y_k - Cx_k \quad (7.2)$$

where the term Cx_k is the models estimate of the plant output based on x_{k-1} , u_{k-1} and ϵ_{k-1} . For State space models derived from transfer functions 6.1 or ARXModels 10.1 the Kalman gain K are calculated analytically.

SISO State space models with one process input ($nu = 1$) and one process output ($ny = 1$) can be constructed directly from a transfersfunction or an ARXModel. MIMO State space models with more than one input or output are can be generated from generic matrices 5 of TransferFunctions or ARXModels.

If the state space model derived from transfer functions or ARXModels has more than one process input, the distribution of the innovation between the individual process inputs ($u_j, j = 0, \dots, nu - 1$) and process outputs ($y_i, i = 0, \dots, ny - 1$) must be specified

$$\mathbf{x}_i^+ = A_{ij}\mathbf{x} + B_{ij}u_j + K_{ij}\epsilon_i + off_{ij} \quad (7.3a)$$

$$\mathbf{y}_i = CA_{ij}\mathbf{x}_i + \epsilon_i \quad (7.3b)$$

ϵ_{ij} is calculated as a fraction w_{ij} of the total innovation given by 7.2

$$\epsilon_{ij} = w_{ij}\epsilon \quad i = 0, \dots, ny - 1 \quad j = 0, \dots, nu - 1 \quad (7.4a)$$

$$\sum_{j=0}^{nu-1} w_{ij} = 1.0 \quad (7.4b)$$

In the *StateSpaceModel* constructors the fractions are specified as a matrix of ErrorLevels *EL* with dimension *ny, nu*. The fractions w_{ij} are calculated from *EL*

$$w_{ij} = EL_{ij} / \sum_{j=0}^{nu-1} EL_{ij} \quad i = 0, \dots, ny - 1 \quad (7.5)$$

A low Errorlevel $EL_{ij} \rightarrow 0$ indicates that model of the dynamic between y_i and u_j is very accurate, a high value indicates a an inaccurate model.

The default ErrorLevel is a matrix with $EL_{ij} = 1/nu$, i.e. with even distribution of the ErrorLevels.

7.1 StateSpaceModel

Class StateSpaceModel

State Space model class $X+ = A*X + B*U + K * \text{eps} + \text{off} ; Y = C*X + \text{eps}$

```
[Serializable]
class StateSpaceModel
```

Constructors

```
StateSpaceModel()
```

Constructor for serialization

```
StateSpaceModel(Matrix A, Matrix B, Matrix C, Matrix K)
```

Constructor from system matrices

Parameters

```
A   System matrix A
B   System matrix B
D   System matrix C
K   System matrix K
```

```
StateSpaceModel(ARXModel arxModel)
```

Constructor ArxModel to StateSpace Model

Parameters

```
arxModel  ARX model
```

```
StateSpaceModel(TransferFunction G, double T):
this(new ARXModel(G, T))
```

Constructor TransferFunction to StateSpace Model

Parameters

G Transfer function

T Sample time

```
StateSpaceModel(Matrix<TransferFunction> model, double T)
: this(model, T, new Vector(model.rows, 1.0))
```

Constructor MiMo TransferFunction to StateSpace model

Parameters

model Generic matrix of TransferFunction objects

T Sample time

```
StateSpaceModel(Matrix<TransferFunction> model, double T, Vector Alfa)
: this(model, T, Alfa, null)
```

Constructor MiMo TransferFunction to StateSpace Models

Parameters

model Generic matrix of TransferFunction objects

T Sample time

alpha Noise Integrator Coificients

```
StateSpaceModel(Matrix<TransferFunction> model, double T, Matrix EL)
: this(model, T, new Vector(model.rows, 1.0), EL)
```

Constructor MiMo TransferFunction to StateSpace Models

Parameters

model Generic matrix of TransferFunction objects

T Sample time

EL Error level

```
StateSpaceModel(Matrix<TransferFunction> model, double T, Vector Alfa, Matrix EL)
```

Constructor MiMo TransferFunction to StateSpace Models

Parameters

model Generic matrix of TransferFunction objects

T Sample time

alpha Noise Integrator Coificients

EL Error level

```
StateSpaceModel(Matrix<ARXModel> model)
: this(model, null)
```

Constructor MiMo ARX model to StateSpace

Parameters

model Generic matrix of ArxModel objects

```
StateSpaceModel(Matrix<ARXModel> model, Matrix EL)
```

Constructor MiMo ARX model to StateSpace

Parameters

model Generic matrix of ArxModel objects

EL ErrorLevel

Properties**A**

System Matrix A

```
Matrix A {set; get;}
```

B

System Matrix B

```
Matrix B {set; get;}
```

C

System Matrix C

```
Matrix C {set; get;}
```

Created

Created (Administrative field)

```
DateTime Created {set; get;}
```

Delay

Minimum delay for all manipulated vars

```
int Delay {get;}
```

Delays

Delay chains for U 17.3

```
DelayChain[] Delays {set; get;}
```

Description

Description (Administrative field)

```
string Description {set; get;}
```

Dimension

States

```
int Dimension {set; get;}
```

K

System Matrix K

```
Matrix K {set; get;}
```

NU

Manipulated vars

```
int NU {set; get;}
```

NY

Observed vars

```
int NY {set; get;}
```

Off

Offset ($X+ = A * X + B * U + K * Eps + Off$)

```
Vector Off {get;}
```

US

Operating point for inputs (Administrative field)

```
Vector US {set; get;}
```

XS

Operating point for State

```
Vector XS {set; get;}
```

Methods**AssertEqual**

Assert equal, and show objects if not equal

```
static void AssertEqual(string text, StateSpaceModel actual, StateSpaceModel expected)
```

Parameters

text	text string
actual	actual statespace model
expected	expected statespace model

Equal

Equal compare two StateSpace models

```
static bool Equal(StateSpaceModel actual, StateSpaceModel expected)
```

Parameters

actual actual statespace model
expected expected statespace model

NextState

Next state $X_+ = A*X + B*U + K*Eps$

```
Vector NextState(Vector X, Vector U, Vector Eps)
```

Parameters

X State vector
U Manipulated vector
Eps Noise
returns Next state

Observed

Observed $Y = C*X$

```
Vector Observed(Vector X)
```

Parameters

X
returns Observed vector

Reset

Reset delay chains

```
void Reset()
```

SetKalmanGain

Set Kalman Gain from noise spectrum

```
void SetKalmanGain(Vector q, Vector r)
```

Parameters

q Process Noise
r Measurement Noise

Show

Show State Space model

```
static void Show(string txt, StateSpaceModel model)
```

Parameters

txt

model

Chapter 8

Equation solvers

Equation solvers are used to solve linear equations $A * x = r$. The factorization of the coefficient matrix is performed during the creation of the solver object, making multiple solutions equations with equal coefficient matrices very efficient.

```
int dim = 5;
Matrix A = Matrix.Random(dim, dim, -100.0, 100.0);

ISolver solver = new LinearEquationSolver(A);

Vector r = Vector.Random(dim, -100.0, 100.0);
Vector x = solver.Solve(r);
Vector.AssertEqual("r", A * x, r);

Matrix R = Matrix.Random(dim, 3, -100.0, 100.0);
Matrix X = solver.Solve(R);
Matrix.AssertEqual("R", A * X, R);
```

In this example the *Vector.AssertEqual* and *Matrix.AssertEqual* tests whether the correct solutions to the equations has been obtained.

The *LinearEquationSolver* performs the $A = P * L * U$ factorizations, and can be used to invert matrices.

The equation solvers implements the interface *ISolver* facilitating easy switch between solvers.

The *SparseEquationSolver* is using Intel's PARDISO solver.

8.1 ISolver

Interface ISolver

Interface for Equation solvers

```
interface ISolver
```

Methods

Solve

Solve $A \cdot x = r$

```
Vector Solve(Vector r)
```

Parameters

r Vector r

returns Vector x

Solve

Solve $A \cdot X = R$

```
Matrix Solve(Matrix R)
```

Parameters

R Matrix R

returns Matrix X

8.2 LinearEquationSolver

Class LinearEquationSolver

Linear equation solver, PLU factorization. Matrix determinant. Solve $A \cdot x = r$ and $A \cdot X = R$

```
class LinearEquationSolver : ISolver
```

Constructors

```
LinearEquationSolver(Matrix A)
```

Constructor

Parameters

A Coefficient matrix

Properties

L

Lower factor in $A = PLU$ factorization

```
Matrix L {get;}
```

U

Upper factor in $A = PLU$ factiorization

`Matrix U {get;}`

Methods**Determinant**

Determinant

`double Determinant()`

Parameters

returns determinant of A

Invert

Invert matrix

`Matrix Invert()`

Parameters

returns A^{-1}

Pivot

`void Pivot(Matrix LU)`

Parameters

LU $L*U$

Solve

Solve $A*x = r$

`Vector Solve(Vector r)`

Parameters

r vector r

returns vector x

Solve

Solve $A*X = R$

`Matrix Solve(Matrix R)`

Parameters

r Matrix R

returns Matrix X

8.3 LeastSquareEquationSolver

Class LeastSquareEquationSolver

Least Square Equation Solver. QR Factorization

```
class LeastSquareEquationSolver : ISolver
```

Constructors

```
LeastSquareEquationSolver(Matrix A)
```

Constructor

Parameters

A Coefficient matrix

Properties

Q

Q matrix, $A = Q \cdot R$ Q orthogonal

```
Matrix Q {get;}
```

R

R matrix , $A = Q \cdot R$, R upper triangular

```
Matrix R {get;}
```

R1

R1 matrix R upper triangular

$$A = Q \begin{pmatrix} R_1 \\ 0 \end{pmatrix}$$

```
Matrix R1 {get;}
```

Methods

Solve

Solve $A \cdot x = r$

```
Vector Solve(Vector r)
```

Parameters

r vector r

returns vector x

Solve

Solve $A \cdot X = R$

`Matrix Solve(Matrix R)`

Parameters

`R` `Matrix R`

returns `Matrix X`

8.4 CholeskyEquationSolver

Class **CholeskyEquationSolver**

Cholesky equations solver

`class CholeskyEquationSolver : ISolver`

Constructors

`CholeskyEquationSolver(Matrix a)`

Constructor Set Equations and do Cholesky decomposition

Parameters

`a` Coificient matrix

Properties**LTR**

Factor lower triangular

`Matrix LTR {get;}`

Methods**Solve**

Solve $A \cdot x = r$

`Vector Solve(Vector r)`

Parameters

`r` `vector r`

returns `vector x`

SolveSolve $A \cdot X = R$ **Matrix** Solve(**Matrix** R)

Parameters

R **Matrix** Rreturns **Matrix** X**8.5 SymmetricEquationSolver***Class* **SymmetricEquationSolver**

Symmetrical Equation solver class

class SymmetricEquationSolver : ISolver**Constructors**SymmetricEquationSolver(**Matrix** A)

Constructor

Parameters

A Symmetrical coefficient matrix

Methods**Invert**

Invert matrix

Matrix Invert()**Solve**Solve $A \cdot x = r$ **Vector** Solve(**Vector** r)

Parameters

r **vector** rreturns **vector** x**Solve**Solve $A \cdot X = R$ **Matrix** Solve(**Matrix** R)

Parameters

R **Matrix** Rreturns **Matrix** X

8.6 SymmetricalBlockEquationSolver

Class SymmetricalBlockEquationSolver

Symmetrical Block Equation Solver, exploiting diagonal elements in diagonal of block matrix.

```
class SymmetricalBlockEquationSolver
```

Constructors

```
SymmetricalBlockEquationSolver(BMatrix A)
```

Constructor

Parameters

A Symmetrical block matrix with values stored in lower triangle

Methods

Solve

Solve $A * x = r$

```
BVector Solve(BVector r)
```

Parameters

r coefficient block vector

8.7 SparseEquationSolver

Class SparseEquationSolver

Sparse Linear Equation Solver, (PARDISO, Parallel Direct Sparse Solver) Solve $A*x = r$ and $A*X = R$

```
class SparseEquationSolver : ISolver
```

Constructors

```
SparseEquationSolver(Matrix A)
```

```
: this(SparseMatrixType.RealUnsymmetrical, UPL0.Upper, A)
```

Constructor for unsymmetrical matrix

Parameters

A Coefficient matrix

```
SparseEquationSolver(SparseMatrixType type, UPL0 UpLo, Matrix A)
```

Constructor

Parameters

type Sparse matrix type
 UpLo Upper or lower triangular for symmetric matrices
 A Coefficient matrix

`SparseEquationSolver(BMatrix A)`

`: this(SparseMatrixType.RealUnsymmetrical, UPLO.Upper, A)`

Constructor for Block Matrix/Vector equations

Parameters

A Coefficient matrix

`SparseEquationSolver(SparseMatrixType type, UPLO UpLo, BMatrix A)`

Constructor for Block Matrix/Vector equations

Parameters

type Sparse matrix type
 UpLo Upper or lower triangular for symmetric matrices
 A Coefficient matrix

Methods

Solve

Solve equations

`Vector Solve(Vector r)`

Parameters

r right hand side coefficients
 returns solution vector

Solve

Solve equations

`Matrix Solve(Matrix R)`

Parameters

R right hand side coefficients
 returns solution matrix

Solve

Solve equations

`BVector Solve(BVector r)`

Parameters

r right hand side coefficients
 returns solution vector

Solve

Solve equations

BMatrix Solve(BMatrix R)

Parameters

R right hand side coefficients

returns solution matrix

8.8 Banded matrix storage

The equation solvers *BandEquationSolver* 8.9 and *SymmetricBandEquationSolver* 8.10 stores the data in a compact format as follows. The m by n band matrix A with $ku = 1$ non-zero super-diagonals and $kl = 2$ sub-diagonals is stored in a matrix AB with $m = kl + 1 + ku$ rows and n columns. Columns of the matrix is stored in the corresponding columns in the AB matrix and diagonals of the matrix are stored in the rows of the array. Thus a_{ij} is stored in $ab[ku + 1 + i - j, j]$ for $\max(1, j - ku, \leq 1 \leq \min(n, j + kl))$. The elements marked NaN in AB are ignored.

A Matrix.Form General [8,8]

-78,7643	65,9043	0	0	0	0	0	0
-94,1250	-2,6987	9,3032	0	0	0	0	0
-26,2801	-61,2631	61,8430	18,1162	0	0	0	0
0	-21,6376	-80,8072	-79,1933	42,3467	0	0	0
0	0	-30,0282	41,7612	26,4536	-3,5851	0	0
0	0	0	76,4280	35,7596	-72,9231	-81,3391	0
0	0	0	0	-83,8892	34,8921	-34,3952	39,6899
0	0	0	0	0	75,5007	-84,5224	10,9481

AB Matrix.Form General [4,8]

NaN	65,9043	9,3032	18,1162	42,3467	-3,5851	-81,3391	39,6899
-78,7643	-2,6987	61,8430	-79,1933	26,4536	-72,9231	-34,3952	10,9481
-94,1250	-61,2631	-80,8072	41,7612	35,7596	34,8921	-84,5224	NaN
-26,2801	-21,6376	-30,0282	76,4280	-83,8892	75,5007	NaN	NaN

Symmetrical banded matrix A with n rows and columns and $kd = 1$ non-zero sub and super diagonals is stored in a compact matrix AB with $1 + kd$ rows and n columns. The first row is the diagonal of A and the following rows are the sub-diagonals of A . The elements marked NaN in AB are ignored.

A Matrix.Form General [8,8]

62,0381	20,6993	0	0	0	0	0	0
20,6993	50,3402	-40,3750	0	0	0	0	0

	0	-40,3750	42,2135	-13,3814	0	0	0	
	0	0	-13,3814	103,5437	-14,6392	0	0	
	0	0	0	-14,6392	65,9977	-60,5258	0	
	0	0	0	0	-60,5258	76,3448	15,1430	
	0	0	0	0	0	15,1430	83,1616	61,1
	0	0	0	0	0	0	61,1746	65,3
AB Matrix.Form General[2,8]								
	62,0381	50,3402	42,2135	103,5437	65,9977	76,3448	83,1616	65,3
	20,6993	-40,3750	-13,3814	-14,6392	-60,5258	15,1430	61,1746	

8.9 BandEquationSolver

Class BandEquationSolver

Band Equation solver class

```
class BandEquationSolver : ISolver
```

Constructors

```
BandEquationSolver(int Kl, int Ku, Matrix AB)
```

Constructor

Parameters

- Kl Number of sub-diagonals
- Ku Number of Super-diagonals
- AB Coefficient matrix for banded matrix

Methods

Solve

Solve $A \cdot x = r$

```
Vector Solve(Vector r)
```

Parameters

- r vector r
- returns vector x

Solve

Solve $A \cdot X = R$

```
Matrix Solve(Matrix R)
```

Parameters

- R Matrix R
- returns Matrix X

8.10 SymmetricBandEquationSolver

Class SymmetricBandEquationSolver

Symmetrical positive definite band matrix Equation solver class

```
class SymmetricBandEquationSolver : ISolver
```

Constructors

```
SymmetricBandEquationSolver(int Kd, Matrix A)
```

Constructor

Parameters

Kd number of super- and sup- diagonals

A Coefficient matrix

Methods

Solve

Solve $A \cdot x = r$

```
Vector Solve(Vector r)
```

Parameters

r vector r

returns vector x

Solve

Solve $A \cdot X = R$

```
Matrix Solve(Matrix R)
```

Parameters

R Matrix R

returns Matrix X

Chapter 9

Linear Filter

The linear filter or the discrete transfer function is defined by

$$F(q) = q^{-k} R(q) / P(q) \quad (9.1)$$

where q is the shift operator, $u(t+1) = qu(t)$ and $q^{-1}u(t) = u(t-1)$

The numerator and denominator polynomials are defined by

$$R(q) = \sum_{i=0}^{nz} r_i q^{-i} \quad P(q) = \sum_{i=0}^{np} p_i q^{-i} \quad (9.2)$$

The following example illustrates the use of linear filters. We have an ARX process modelled as

$$Y_{Plant} = Y_{Det} + Y_{Stoc} \quad (9.3)$$

$$= G(q)u(t) + H(q)e(t) \quad (9.4)$$

$$= q^{-k} \frac{R(q)}{P(q)} u(t) + \frac{1}{P(q)} e(t) \quad (9.5)$$

where the input signal $u(t)$ is a pseudo random binary signal, and $e(t)$ is white noise.

The process is a second order oscillating process defined by the transfer function $Plant$. The LinearFilter $G(q)$ describing the deterministic part of Y_{Plant} is derived directly from $Plant$ and the sample time DT . The LinearFilter $H(q)$ is derived from $G(q)$ denominator polynomial.

The input signal $u(t)$ is generated using the PBR9.1 routine. The deterministic part comes from $Y_{Det} = G(q)u(t)$. The stochastic part $Y_{Stoc} = H(q)e(t)$ where the white noise series is generated using the *Niid* 9.1 routine. The plant response comes from $Y_{Plant} = Y_{Det} + Y_{Stoc}$

Having an input and an output series a LinearFilter $\hat{G}(q)$ can be estimated using the Regress 9.1 routine. An estimate $\hat{Y}(t)$ of the deterministic response is calculated. The estimated deterministic response is calculated from $\hat{Y}(t) = \hat{G}(q)u(t)$

The code is

```

console.WriteLine("Test Linear Filter");

int N = 1000;
double mean = 0.0;
double var = 1.0e-4;

double gain = 1.0;
int delay = 5;
double tau = 10.0;
double sigma = 0.3;
double DT = 1.0;

// second order system
TransferFunction Plant = new TransferFunction(gain, delay, tau, sigma);

// model for deterministic part
LinearFilter G = new LinearFilter(Plant, DT);
// model for stochastic part
LinearFilter H = new LinearFilter(new Polynomial(1.0), G.P);

LinearFilter.Show("G", G);
LinearFilter.Show("H", H);
console.Show("H.Gain", H.Gain);

// Pseudo Random Binary Signal
Vector u = LinearFilter.PBRS(N, 0.005);

// Deterministic response
Vector yDet = G * u;

// Stochastic response
Vector yStoch = H * LinearFilter.Niid(N, mean, var);
Vector yPlant = yDet + yStoch;

int np = 3;
int nz = 2;
int k = 6;
// Estimate model from input signal u and output signal yPlant
LinearFilter GHat = LinearFilter.Regress(np, yPlant, k, nz, u);
LinearFilter.Show("GHat", GHat);

// Estimated deterministic response
Vector yHat = GHat * u;

double max = 2.0;
double min = -max;
console.Plot(
    new Plot(new PlotSeries("u", min, max, u),
        new PlotSeries("yPlant", min, max, SeriesColor.Red, yPlant),
        new PlotSeries("yDet", min, max, SeriesColor.Green, yDet),

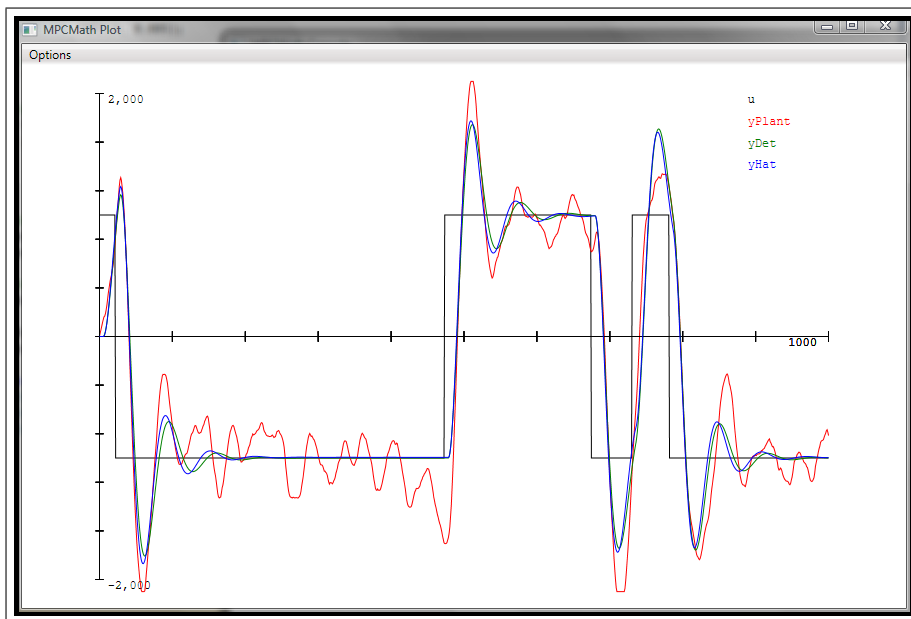
```

```
new PlotSeries("yHat", min, max, SeriesColor.Blue, yHat));
```

with the following output

```
Test Linear Filter
G ShiftFunction K = 6
  Z
  0,0049    0,0048
  P
      1    -1,9321    0,9418
H ShiftFunction
  Z
      1
  P
      1    -1,9321    0,9418
H.Gain 103,1159
GHat ShiftFunction K = 6
  Z
  0,0023    0,0087
  P
      1    -1,9298    0,9409
```

end the plotted time series



9.1 LinearFilter

Class LinearFilter

LinearFilter (Discrete Transfer function)

[Serializable]

class LinearFilter : ICommon<LinearFilter>

Constructors

LinearFilter() : this (new Polynomial(1.0))

Default constructor, $F(q) = 1$

LinearFilter(int K, Polynomial R, Polynomial P)

Constructor $F(q) = q^{-k}R(q)/P(q)$

Parameters

k Delay

R R Polynomial

P P Polynomial

LinearFilter(Polynomial R, Polynomial P)

Constructor $F(q) = R(q)/P(q)$

Parameters

R R Polynomial

P P Polynomial

LinearFilter(Polynomial R)

Constructor $F(q) = R(q)$

Parameters

R R Polynomial

LinearFilter(params double[] vals)

Constructor $F(q) = R(q)$

Parameters

vals R coefficient array

LinearFilter(TransferFunction TF, double T)

Constructor LinearFilter from TransferFunction

Parameters

TF TransferFunction object

T Sample time

LinearFilter(int K, params double[] vals)

Constructor $F(q) = q^{-k}R(q)$

Parameters

K Delay

vals R coefficient array

Properties**Gain**

Gain (stationary)

```
double Gain {get;}
```

InitialValue

Initial value for input time series, used by LinearFilter object for $y(t)$ where $t < 0$

```
double InitialValue {set; get;}
```

K

Delay

```
int K {set; get;}
```

P

Denominator Polynomial

```
Polynomial P {set; get;}
```

R

Numerator Polynomial

```
Polynomial R {set; get;}
```

Roots

Roots

```
CVector Roots {get;}
```

Zeroes

Zeroes

```
CVector Zeroes {get;}
```

Methods

AssertEqual

Assert Equal, show if not equal

```
static bool AssertEqual(string text, LinearFilter actual, LinearFilter expected)
```

Parameters

text	text string
actual	actual filter object
expected	expected filter object
returns	true if equal

Equal

Test if Equal

```
static bool Equal(LinearFilter A, LinearFilter B)
```

Parameters

A	A Filter object
B	B Filter object
returns	true if equal

Invert

Invert LinearFilter

```
LinearFilter Invert()
```

Parameters

returns	F^{-1}
---------	----------

Invertible

Check if LinearFilter is invertible. If $v = H(q)e(t)$ implies that $e(t) = H^{-1}(q)v(t)$

```
bool Invertible()
```

Parameters

returns	true if invertible
---------	--------------------

Niid

Normal independent identical distributed process

```
static Vector Niid(int N, double Mean, double Variance)
```

Parameters

N	Series length
Mean	Series mean
Variance	Series variance

operator

Filter time series

```
static Vector operator *(LinearFilter F, Vector u)
```

Parameters

F Filter object
 u Input time series
 returns $F(q)u(t)$

operator

Filter time series

```
static Vector operator *(Vector u, LinearFilter H)
```

Parameters

u Input time series
 F Filter object
 returns $F(q)u(t)$

operator

Operator + , clone

```
static LinearFilter operator +(LinearFilter F)
```

Parameters

F Filter object
 returns $+F$

operator

Operator -

```
static LinearFilter operator -(LinearFilter F)
```

Parameters

F Filter object
 returns $-F$

operator

Multiply by constant

```
static LinearFilter operator *(double fak, LinearFilter F)
```

Parameters

fak constant
 F Filter object
 returns $fak * F$

operator

Multiply by constant

```
static LinearFilter operator *(LinearFilter F, double fak)
```

Parameters

F Filter object

fak constant

returns $fak * F$

operator

Divide by constant

```
static LinearFilter operator /(LinearFilter F, double fak)
```

Parameters

F Filter object

fak constant

returns F/fak

operator

Add Linear Filters

```
static LinearFilter operator +(LinearFilter A, LinearFilter B)
```

Parameters

A A Filter object

B B Filter object

returns $A + B$

operator

Subtract Linear Filters

```
static LinearFilter operator -(LinearFilter A, LinearFilter B)
```

Parameters

A A Filter object

B B Filter object

returns $A - B$

operator

Add constant and LinearFilter

```
static LinearFilter operator +(double alpha, LinearFilter A)
```

Parameters

alpha constant

A A Filter object

returns $A + alpha$

operator

Add LinearFilter and constant

```
static LinearFilter operator +(LinearFilter A, double alpha)
```

Parameters

A A Filter object

alpha constant

returns $\alpha + A$

operator

Subtract constant and LinearFilter

```
static LinearFilter operator -(double alpha, LinearFilter A)
```

Parameters

alpha constant

A A Filter object

returns $A - \alpha$

operator

Subtract LinearFilter and constant

```
static LinearFilter operator -(LinearFilter A, double alpha)
```

Parameters

A A Filter object

alpha constant

returns $A - \alpha$

operator

Multiply operator

```
static LinearFilter operator *(LinearFilter A, LinearFilter B)
```

Parameters

A A Filter object

B B Filter object

returns $A * B$

operator

Divide operator

```
static LinearFilter operator /(LinearFilter A, LinearFilter B)
```

Parameters

A A Filter object

B B Filter object

returns A/B

operator

Operator divide constant with linear filter

```
static LinearFilter operator /(double alpha, LinearFilter A)
```

Parameters

alpha constant
A A Filter object
returns *alpha/A*

PBRS

Pseudo Binary Random Signal series. Produces a PRBS series u of length N , where $u[i]$ switches between -1 and 1 with probability *prob*

```
static Vector PBRS(int N, double prob)
```

Parameters

N Series length
prob Probability for switch
returns u series

Regress

Find Shift Function $F(q) = 1/P(q)$ where $P(q)y(t) = e(t)$, $P(q) = \sum_{i=0}^{np} p_i q^{-i}$ and $e(t) = N_{iid}(0.0, 1.0)$

```
static LinearFilter Regress(int np, Vector y)
```

Parameters

np Number of poles
y Output series
returns $F(q) = 1/P(q)$

Regress

Find Shift Function $F(q) = q^{-k}R(q)/P(q)$ where $R(q) = \sum_{i=0}^{nz} r_i q^{-i}$, $P(q) = \sum_{i=0}^{np} p_i q^{-i}$ and $y(t) = F(q)u(t)$

```
static LinearFilter Regress(int np, Vector y, int k, int nz, Vector u)
```

Parameters

np Number of poles
y Output series
k Delay
nz Number of poles
u Output series

Regress

Find Shift Function $F(q) = q^{-k}R(q)/P(q)$ where $R(q) = \sum_{i=0}^{nz} r_i q^{-i}$, $P(q) = \sum_{i=0}^{np} p_i q^{-i}$ and $y(t) = F(q)u(t)$

```
static LinearFilter Regress(Norm norm, double gamma,
int np, Vector y, int k, int nz, Vector u)
```

Parameters

norm	Regression norm
gamma	Huber norm parameter
np	Number of poles
y	Output series
k	Delay
nz	Number of poles
u	Output series

Show

Show LinearFilter

```
static void Show(string txt, LinearFilter A)
```

Parameters

txt	text string
A	A Filter object

Chapter 10

ARX models

The ARX model describes the process

$$A(q)y(t) = B(q)u(t) + e(t) \quad (10.1)$$

q is the time shift operator, and the polynomials are

$$A(q) = 1 + \sum_{j=1}^{ny} a_j q^{-j} \quad (10.2)$$

$$B(q) = \sum_{j=1}^{nu} B_j q^{-j} \quad (10.3)$$

The *ExtendedDeltaARX* [Huusom et al., 2010] model is

$$A(q)y(t) = B(q)u(t) + \frac{1 - \alpha q^{-1}}{1 - q^{-1}} e(t) \quad (10.4)$$

see description given in Introduction 1.6.

10.1 ARXModel

Class ARXModel

ARX model and Extended Delta ARX Model

```
[Serializable]
class ARXModel : ICommon<ARXModel>
```

Constructors

```
ARXModel() : this(0, new Vector(1, 1.0), new Vector(1), 1.0)
```

Default constructor for serialization

```
ARXModel(int Delay, Vector A, Vector B, double T)
```

Constructor ARX Model

Parameters

Delay Delay
 A A polynomial
 B B polynomial
 T Sample time

`ARXModel(int Delay, Vector A, Vector B, double T, double Alfa)`

Constructor Extended Delta ARX Model

Parameters

Delay Delay
 A A polynomial
 B B polynomial
 T Sample time
 Alfa Alfa coefficient

`ARXModel(TransferFunction TF, double T)`
`: this(0, new Vector(1, 1.0), new Vector(1), 1.0)`

Constructor ARX model

Parameters

TF System transfer function
 T Sample time

`ARXModel(TransferFunction TF, double T, double Alfa)`

Constructor Extended Delta ARX Model

Parameters

TF System transfer function
 T Sample time
 Alfa Alfa coefficient

Properties

A

ARX coefficients A, $A(q-1)y(t) = (q\text{-delay}) * B(q-1) * u(t)$

`Vector A {get;}`

Alfa

Extended Delta ARX model alpha coefficient

`double Alfa {get;}`

ASP

State space matrix ASP

$X+ = ASP * X + BSP * U + KSP * \text{eps}$

$Y = C * X + \text{eps}$

`Matrix ASP {get;}`

B

ARX coefficients B, $A(q-1)y(t) = (q\text{-delay})*B(q-1)*u(t)$

Vector B {get;}

BSP

State space matrix BSP

$X+ = ASP*X + BSP*U + KSP* \text{eps}$

$Y = C*X + \text{eps}$

Vector BSP {get;}

C

State space vector C

$X+ = ASP*X + BSP*U + KSP* \text{eps}$

$Y = C*X + \text{eps}$

Vector C {get;}

Delay

Delay states

int Delay {get;}

Dimension

State dimension

int Dimension {get;}

ExtendedDeltaARX

Extended Delta ARX model

bool ExtendedDeltaARX {get;}

Gain

ARX function gain

double Gain {get;}

KSP

State space matrix KSP

$$X+ = ASP*X + BSP*U + KSP* \text{eps}$$

$$Y = C*X + \text{eps}$$

Vector KSP {get;}

T

Sampling time (in seconds);

double T {get;}

Methods**AssertEqual**

Assert equal and show function if nor equal

```
static void AssertEqual(string txt, ARXModel a, ARXModel b)
```

Parameters

txt text string

a Actual ARX model

b Expected ARX model

NextState

Next state

```
Vector NextState(Vector X, double u, double eps)
```

Parameters

X state vector

u Manipulated variables

eps Noise vector

returns Next state x

Observed

Observed values

```
double Observed(Vector X)
```

Parameters

X state vector

returns Y vector

Reset

Reset u delay chain

```
void Reset()
```

Show

Show ARX Model

```
static void Show(string txt, ARXModel a)
```

Parameters

txt text string

a ARXModel object

Chapter 11

Function and Model interfaces

MPCMath provides the *IFun* and *IConFun* interfaces as general interfaces to unconstrained and Constrained functions.

The function interface *IFun* is useful for fitting function parameters, as described in section Least Square Fitting 12.6

The interface *IFeasible* is inherited by all the interfaces above. *IFeasible* makes it possible to define boxlimits for the feasible region of a model or a function.

The *IModel* interfaces is general interface to plant objects.

11.1 IFun

Interface IFun

Interface for function $\mathbb{R}_n \Rightarrow \mathbb{R}$

```
interface IFun : IFeasible
```

Properties

N

Number of variables in x

```
int N {get;}
```

Methods

DDx

Hessian matrix

Matrix DDx(Vector x)

Parameters

x state

returns Hessian matrix. null if Hessian undefined or x infeasible

Dx

Jacobian vector

Vector Dx(Vector x)

Parameters

x state

returns Jacobian vector or null if x infeasible

Value

Function value

double Value(Vector x)

Parameters

x State

returns function value or double. MaxValue if x infeasible

11.2 IConFun

Interface IConFun

Interface for Constrained functions $\mathbb{R}_n \Rightarrow \mathbb{R}$. Equality constraints $Ce(X) = 0$ and Inequality Constraints $Ci(x) \geq 0$

interface IConFun : IFun

Properties

Ne

Number of Equality conditions

int Ne {get;}

Ni

Number of Inequality conditions

int Ni {get;}

Methods

DDxe

Hessians of Equality Constraints

BMatrix DDxe(**Vector** x)

Parameters

x state

returns Hessian of constraints. null if Hessian undefined or x infeasible

DDxi

Hessians of Inequality Constraints

BMatrix DDxi(**Vector** x)

Parameters

x state

returns Hessian of constraints. null if Hessian undefined or x infeasible

Dxe

Gradients of Equality constraints

Matrix Dxe(**Vector** x)

Parameters

x state

returns Gradients of constraints

Dxi

Gradients of Inequality constraints

Matrix Dxi(**Vector** x)

Parameters

x state

returns Gradients of constraints

Vale

Equality Constraint Values

Vector Vale(**Vector** x)

Parameters

x State

returns Constraint values

Vali

Inequality Constraint Values $Vali \geq 0$

Vector Vali(**Vector** x)

Parameters

x State

returns Constraint values

11.3 IFunt*Interface* **IFunt**

Interface for function $\mathbb{R}_n \Rightarrow \mathbb{R}$

```
interface IFunt : IFeasible
```

Properties

N

Number of variables in x

```
int N {get;}
```

Methods

DDx

Hessian

Matrix DDx(**Vector** x, double t)

Parameters

x

t

returns Hessian matrix. null if Hessian undefined or x infeasible

Dx

Jacobian

Vector Dx(**Vector** x, double t)

Parameters

x parameters

t t

returns Jacobian or null if infeasible

Value

Function value, (return infinity if x infeasible)

```
double Value(Vector x, double t)
```

Parameters

x parameters

t t

returns function value or infinity if infeasible

11.4 IModel*Interface IModel*

Process model interface

```
interface IModel : IFeasible
```

Properties**Dimension**

Number of states

```
int Dimension {get;}
```

NU

Number of manipulated vars

```
int NU {get;}
```

NY

Number of Observed vars

```
int NY {get;}
```

Parameters

Model parameters, useful for iterative optimization programs

```
Vector Parameters {set; get;}
```

Sparse

Sparse matrices supported

```
bool Sparse {get;}
```

T

Sample time for Next and LinearModel functions

```
double T {get;}
```

Methods**Derivative**

Derivative function dx/dt

```
Vector Derivative(Vector X, Vector U)
```

Parameters

X Present state

U Manipulated variables

returns Time derivative of state X or null

Ju

Jacobian with respect to Inputs

```
Matrix Ju(Vector X, Vector U)
```

Parameters

X State vector

U Manipulated vars

returns Jacobian or Null

Jx

Jacobian with respect to outputs

```
Matrix Jx(Vector X, Vector U)
```

Parameters

X State vector

U Manipulated vars

returns Jacobian or Null

LinearModel

Linear State Space Model

```
StateSpaceModel LinearModel(Vector XS, Vector US)
```

Parameters
 XS Stationary State
 XY
 returns StateSpaceModel

Next

Next step function

```
Vector Next(Vector X, Vector U, Vector W)
```

Parameters
 X of state X
 U Manipulated variables
 W Disturbance/Process Noise
 returns Next value of state X

Next

Next step with sensitivity matrices

```
void Next(Vector X0, Vector U, Vector W, out Vector X,  
out Matrix A, out Matrix B)
```

Parameters
 X0 Initial state value
 U Manipulated variables
 W Disturbance/Process Noise
 X Next value of state X
 A Sensitivity $dX/dX0$
 B Sensitivity dX/dU

Observed

Observed output

```
Vector Observed(Vector X, Vector U)
```

Parameters
 X State X
 U Manipulated variables

Observed

Observed output with sensitivities

```
void Observed(Vector X, Vector U, out Vector Y, out Matrix C, out Matrix D)
```

Parameters

X State X
 U Manipulated variables
 Y Output
 C sensitivity Dy/DX
 D sensitivity Dy/DU

11.5 IFeasible*Interface* **IFeasible**

Feasible interface, defining the feasible region of a function as boxlimits

```
interface IFeasible
```

Properties**XFMax**

Upper limit on feasible region

```
Vector XFMax {set; get;}
```

XFMin

Lower limit on feasible region

```
Vector XFMin {set; get;}
```

11.6 Infeasible State exceptions

Programs implementing Models using the IModel interface can notify the calling program if it has entered into an infeasible state. For the U-Loop reactor example below, the states are Chemical concentrations, which must be positive. The ODESolver 15.3 and the SteadyState 16.4 routines use these exceptions to back track if error parameter set to 1, otherwise the exception is passed on to the calling program.

```
/// <summary>
/// Test whether input are feasible
/// </summary>
/// <param name="X"></param>
/// <param name="U"></param>
private void Feasible(Vector X, Vector U)
{
    X.ProperZeroes();
    for (int p = 0; p < X.Dimension; p++)
    {
        if (X[p] < 0.0)
        {
            throw new MPCMathException("ULoop reactor", "Infeasible state", 1);
        }
    }
}
```

In the example above "ULoop reactor" can be replaced by your own identifying text.

Chapter 12

Unconstrained and constrained minimization

Unconstrained minimization defined by

$$\min_{x \in \mathbb{R}^n} \theta = fun(x) \quad (12.1a)$$

and equality constrained minimization problems

$$\min_{x \in \mathbb{R}^n} \theta = fun(x) \quad (12.2a)$$

$$s.t. \quad Ax = b \quad (12.2b)$$

are solved using the Newton method [Boyd and Vanderberghe, 2004]. The function to be minimized must implement the IFun 11.1 interface.

12.1 NewtonMethod

Class NewtonMethod

Newtons Method

```
class NewtonMethod
```

Constructors

```
NewtonMethod(IFun function)
```

Constructor for unconstrained solver

Parameters

function function definition

```
NewtonMethod(IFun function, Matrix A, Vector B)
```

Constructor for equality constrained solver

Parameters

function function definition

Properties

Alfa

Back tracking parameter ($0.0 < Alfa < 0.5$)

```
double Alfa {set; get;}
```

Beta

Back tracking parameter ($0.0 < Beta < 1.0$)

```
double Beta {set; get;}
```

Epsilon

Error margin

```
double Epsilon {set; get;}
```

Lambda

Newton decrement

```
double Lambda {get;}
```

MaxBacktrack

Maximum number of backtrack steps

```
long MaxBacktrack {set; get;}
```

MaxIterationSteps

Maximum number of iteration steps

```
long MaxIterationSteps {set; get;}
```

Mu

Dual variables

```
Vector Mu {get;}
```

Steps

number of Newton steps

```
long Steps {get;}
```

X

Position of optimum

```
Vector X {get;}
```

Methods**Minimize**

Minimize function

```
double Minimize(Vector xo)
```

Parameters

xo Start position of x

returns Minimum value

12.2 Extended linear-Quadratic Optimal Control problem

The extended Linear-Quadratic Optimal Control Problem is a special equality constrained minimization problem, which are encountered in solvers for MPC and LQR controllers. This problem can be solved extremely cpu efficient using the RiccatiSolver class 12.8. For a detailed description of the mathematics behind the Extended Linear-Quadratic Optimal Control problem and the the Riccati series a description can be found in [Jørgensen, 2004]

The RiccatiSolver class solves the problem

$$\min_{\{x_{k+1}, u_k\}_{k=0}^{N-1}} \phi = \sum_{k=0}^{N-1} l_k(x_k, u_k) + l_N(x_N) \quad (12.3a)$$

subject to

$$x_{k+1} = A_k x_k + B_k u_k + b_k \quad k = 0, \dots, N-1 \quad (12.3b)$$

with the stage costs

$$l_k(x_k, u_k) = \frac{1}{2} x_k' Q_k x_k + x_k' M_k u_k + \frac{1}{2} u_k' R_k u_k + q_k' x_k + r_k' u_k + f_k \quad (12.4a)$$

$$l_N(x_N) = \frac{1}{2} x_N' Q_N x_N + p_N' x_N + \gamma_k \quad (12.4b)$$

The initial state x_0 is a parameter and not a decision variable.

The following code example illustrates the use of the Riccati Solver class

```

console.WriteLine("Test Riccati");

int N = 100;

Double T = 1.0;
// Second order system
double gain = 1.0;
double delay = 0.0;
double tau = 10.0;
double sigma = 0.1;
TransferFunction G = new TransferFunction(gain, delay, tau, sigma);
StateSpaceModel Plant = new StateSpaceModel(G, T);
StateSpaceModel.Show("Model", Plant);
int nx = Plant.Dimension;
int nu = Plant.NU;
int ny = Plant.NY;

// initial condition
Vector X0 = new Vector(nx, 1.0);

Matrix Ak = Plant.A;
Matrix Bk = Plant.B;
Vector bk = Plant.Off;

Vector theta = new Vector(ny, 1.0);
Vector rho = new Vector(nu, 1.0);

// define stage penalties
Matrix Qk = Matrix.MulTransposed(Plant.C, new Matrix(theta)) * Plant.C;
Vector qk = new Vector(nx);
Matrix Rk = new Matrix(rho);
Vector rk = new Vector(nu);
Matrix Mk = new Matrix(nx, nu);
// end condition penalties
Matrix PN = Ak;
Vector pN = qk;

Matrix.Show("Rk", Rk);
Vector.Show("rk", rk);
Matrix.Show("Qk", Qk);
Vector.Show("qk", qk);
Matrix.Show("Mk", Mk);
Matrix.Show("Pn", PN);
Vector.Show("pn", pN);

```

```

Matrix.Show("Ak", Ak);
Matrix.Show("Bk", Bk);
Vector.Show("bk", bk);

// store stage penalites as diagonal elements in Block matrices
BMatrix Q = new BMatrix(N, Qk);
BMatrix R = new BMatrix(N, Rk);
BMatrix M = new BMatrix(N, Mk);
BMatrix A = new BMatrix(N, Ak);
BMatrix B = new BMatrix(N, Bk);

RiccatiSolver ricattiSolver = new RiccatiSolver(PN, Q, M, R, A, B);

BVector q = new BVector(N, qk);
BVector r = new BVector(N, rk);
BVector b = new BVector(N, bk);

BVector U;
BVector X;

ricattiSolver.Solve(X0, pN, q, r, b, out U, out X);

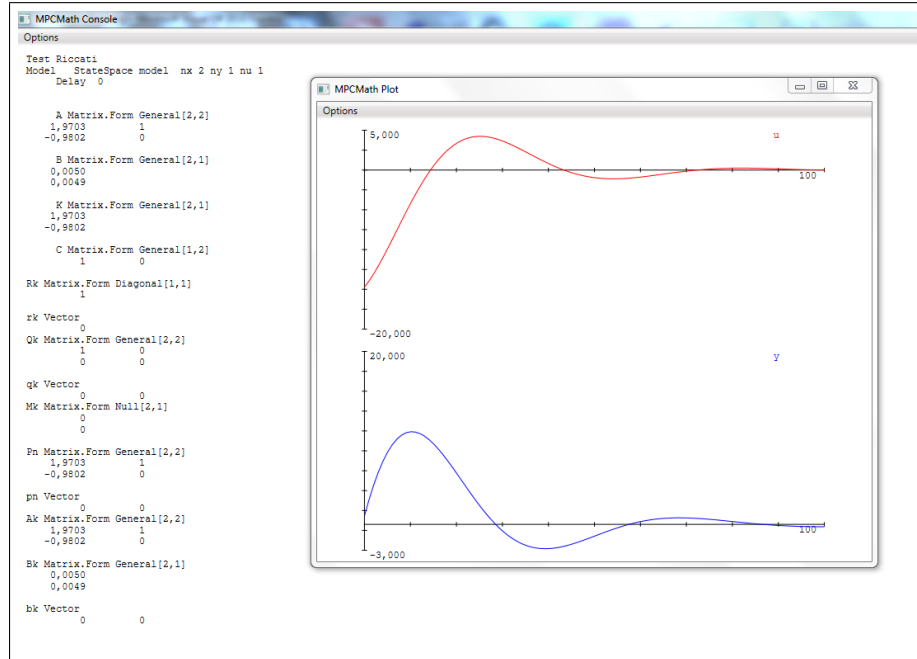
// Plot results

Vector y = new Vector(N);
Vector u = new Vector(N);
for(int k = 0; k < N; k++)
{
    y[k] = X[k][0];
    u[k] = U[k][0];
}

console.Plot(new PlotSeries("u", 0.0, 0.0, SeriesColor.Red, u),
    new PlotSeries("y", 0.0, 0.0, SeriesColor.Blue, y));

```

The code produces the following output



12.3 Function Minimization

The function minimizer *FunMin* finds a local minima of a non-linear function, using a quasi Newton method (BFGS).

$$\min_{x \in \mathbb{R}^n} \quad fun(x) \quad (12.5a)$$

The function to be minimized must implement the *IFun* interface , 11.1, with the property *N* and methods *Value* and *Dx*. *FunMin* does not use the Hessian function *DDx*.

The constrained function minimizer *ConFunMin* 12.5 find a locals minima of fun(x) subject to equality and inequality constraints

$$\min_{x \in \mathbb{R}^n} \quad fun(x) \quad (12.6a)$$

$$s.t. \quad C_e(x) = 0 \quad (12.6b)$$

$$C_i(x) \geq 0 \quad (12.6c)$$

Where C_e is a vector of n_e equality conditions and C_i is a vector of n_i inequality conditions.

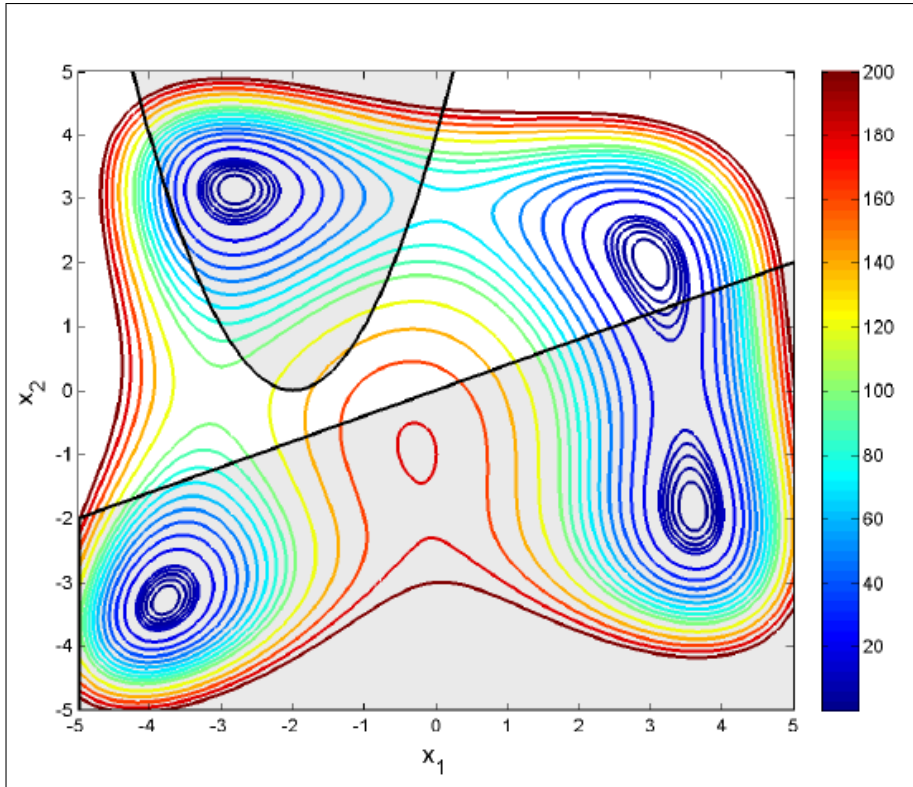
The function to be minimized must implement the *IConFun* interface , 11.2, with the properties *N*, *Ne*, *Ni*, and methods *value*, *Dx*, *Vale*, *Dxe*, *Vali* and *Dxi*. *ConFunMin* does not require the Hessian functions *DDx*, *DDxe*, *DDxi*.

Use of *FunMin* and *ConFunMin* are illustrated on a simple non-linear function:

$$fun(x_1, x_2) = (x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2 \quad (12.7a)$$

$$s.t. \quad C_1(x_1, x_2) = (x_1 + 2)^2 - x_2 \geq 0 \quad (12.7b)$$

$$C_2(x_1, x_2) = -4x_1 + 10x_2 \geq 0 \quad (12.7c)$$



The implementation of *fun* with the *IconFun* interface is shown in appendix D

The use of *FunMin* is demonstrated in the code example below:

```
console.WriteLine("Test Constrained Minimizer");

IconFun fun = new TestConFun();

Vector X0 = new Vector(3.5, 1.5);
Vector XMin = new Vector(-5.0, -5.0);
Vector XMax = new Vector( 5.0,  5.0);

Vector.Show("X0", X0);
Vector.Show("fun.Dx", fun.Dx(X0));
Matrix.Show("fun.DDx", fun.DDx(X0));
```

```

double val;
Vector X;

// Unconstrained minimization

FunMin min = new FunMin(fun);
val = min.Minimize(X0, XMin, XMax);
console.WriteLine("Iterations " + min.Iterations +
    " Minimum value " + val);

X = min.X;
Vector.Show("X*", X);
Vector.Show("fun.Dx", fun.Dx(X));
Matrix.Show("fun.DDx", fun.DDx(X));

```

Notice that the interface *IConFun* inherits the interface *IFun* , making the function *TestConFun* usable as argument requiring an *IFun* interface.

The resulting output on the console is:

```

X0 Vector
      3,5000      1,5000
fun.Dx Vector
      36,0000     -2,0000
fun.DDx Matrix.Form General[2,2]
      31,0000     20,0000
      20,0000     15,0000

Iterations 32 Minimum value 3,09858265117597E-18
X* Vector
      3,0000      2,0000
fun.Dx Vector
      0,0000      0,0000
fun.DDx Matrix.Form General[2,2]
      34,0000     20,0000
      20,0000     34,0000

```

The *Consistent* routine can be used to check the coding of the function to be optimized. *Consistent* checks the approximation:

$$\tilde{f}(x + \Delta x) \simeq f(x) + \nabla_x f(x)' \Delta x \quad (12.8)$$

The use of *Consistent* is demonstrated in the code below

```

FunMin min = new FunMin(fun);

if (min.Consistent(X0))
{

```

```

    val = min.Minimize(X0, XMin, XMax);
    console.WriteLine("Iterations " + min.Iterations +
        " Minimum value " + val);

    X = min.X;
    Vector.Show("X*", X);
    Vector.Show("fun.Dx", fun.Dx(X));
    Matrix.Show("fun.DDx", fun.DDx(X));
}
else
{
    console.WriteLine("Inconsistent function");
}

```

The constrained minimizer *ConFunMin* is based on the Augmented Lagrange multiplier method. The Lagrange function for the optimization problem, 12.6, is

$$L(x, \lambda_e, \lambda_i, z) = fun(x) - \lambda_e C_e(x) - \lambda_i (C_i(x) - z) \quad (12.9a)$$

$$s.t. \quad z \geq 0 \quad (12.9b)$$

where z is a vector of slack variables. This Lagrange function is augmented to the Lagrange function

$$\begin{aligned}
 L(x, \lambda_e, \lambda_i, z) = & fun(x) - \lambda_e C_e(x) - \lambda_i (C_i(x) - z) \\
 & + \frac{1}{2} \mu (C_e(x)' C_e(x) + (C_i(x) - z)' (C_i(x) - z))
 \end{aligned} \quad (12.10)$$

The square term $\frac{1}{2} \mu (C_e(x)' C_e(x) + (C_i(x) - z)' (C_i(x) - z))$ gives a penalty on constraint violations. *ConFunMin* fixes a set of $(\lambda_e, \lambda_i, \mu, z)_0$ and then uses the unconstrained minimizer *FunMin* to minimize the augmented Lagrange function over x . If the constraint violations are unacceptable, the value of μ is increased, and a new minimization is performed. *ConFunMin* continues to iterate on the parameter set $(\lambda_e, \lambda_i, \mu, z)_k$ until an optimal solutions is found. For further details see [Nocedal and J.Wright, 2006] or [K.Madsen et al., 2004]

The following code demonstrate how to locate all four minima's in the simple problem 12.7

```

IConFun fun = new TestConFun();
Vector XMin = new Vector(-5.0, -5.0);
Vector XMax = new Vector( 5.0,  5.0);

double val;
Vector X;

```

```

// Constrained Minimization

console.WriteLine("Constrained minimizer ConFunMin");
ConFunMin lmin = new ConFunMin(fun);

int ntry = 50;
List<Vector> minimas = new List<Vector>();
for (int i = 0; i < ntry; i++)
{
    Vector X0 = Vector.Random(XMin, XMax);
    val = lmin.Minimize(X0, XMin, XMax);
    X = lmin.X;

    if (lmin.Ok)
    {
        bool insert = true;
        for (int p = 0; p < minimas.Count; p++)
        {
            Vector Xminima = minimas[p];
            if ((X - Xminima).Norm() < 1.0e-2)
            {
                insert = false;
                break;
            }
        }

        if (insert)
        {
            minimas.Add(X);
        }
    }
}

console.WriteLine("Search end, found " + minimas.Count + " minimas");
for (int p = 0; p < minimas.Count; p++)
{
    X = minimas[p];
    val = fun.Value(X);
    console.Show("val", val);
    Vector.Show("X", X);
}

```

The program starts minimization for *ntry* random start points. Found minimas are collected in the list *minimas*, if it is not already in the list.

The console output is;

Test Constrained Minimizer Constrained minimizer ConFunMin Search end,
found 4 minimas val 65,4261 X Vector -0,2983 2,8956 val 72,8556 X Vector -
3,5485 -1,4194 val 0,0000 X Vector 3,0000 2,0000 val 35,9298 X Vector -3,6546
2,7377

12.4 FunMin

Class FunMin

Unconstrained function minimizer. Quasi Newton method (BFGS) .

```
class FunMin
```

Constructors

```
FunMin(IFun Fun)
```

Constructor

Parameters

Fun Function to be minimized

Properties

Alpha0

Initial Steepest descent gain

```
double Alpha0 {set; get;}
```

BackTracks

Total number of backtracks in minimization

```
int BackTracks {get;}
```

C1

Wolfe condition constant C1

```
double C1 {set; get;}
```

C2

Wolfe condition constant C2

```
double C2 {set; get;}
```

EpsGrad

Gradient residual limit

```
double EpsGrad {set; get;}
```

EpsStep

Step size residual limit

```
double EpsStep {set; get;}
```

ExitReason

Reason for leaving Minimizer

```
string ExitReason {get;}
```

Iterations

Number of Iterations

```
int Iterations {get;}
```

MaxBacktracks

Maximum number of backtracks in each line search

```
int MaxBacktracks {set; get;}
```

MaxIterations

Maximum number of iteration steps

```
int MaxIterations {set; get;}
```

Ok

Minimize or Consistent result

```
bool Ok {get;}
```

ResGrad

Gradient residual

```
double ResGrad {get;}
```

ResStep

Step size residual

```
double ResStep {get;}
```

Trace

Trace iterations

```
bool Trace {set; get;}
```

TraceLevel

Trace level

```
int TraceLevel {set; get;}
```

UseBFGS

Use BFGS methis to appaximate inverse Hessian

```
bool UseBFGS {set; get;}
```

W

Inverse Hessian BFGS approximation;

```
Matrix W {get;}
```

WarmStart

Warm start, (keep BFGS Hesssian estimate between calls to Minimize)

```
bool WarmStart {set; get;}
```

X

Argmin X

```
Vector X {get;}
```

XUnchanged

X was not changed during minimixation ($X = X_0$)

```
bool XUnchanged {get;}
```

Methods**Consistent**

Check consistence of function to be minimized

```
bool Consistent(Vector x)
```

Parameters

x X operation point
returns true if consistent

Consistent

Check consistence of function to be minimized

```
bool Consistent(Vector x, Vector dx)
```

Parameters

x X operation point
dx variations of x
returns true if consistent

Minimize

Minimize function, find local minimum

```
double Minimize(Vector X0)
```

Parameters

X0 Initial X
returns minimum value

Minimize

Minimize function, find local minimum, with box constraints on variables

```
double Minimize(Vector X0, Vector XMin, Vector XMax)
```

Parameters

X0 Initial X guess
XMin Min values for X, null if no check wanted
XMax Max values for X, null if no check wanted
returns minimum value at local minimum

12.5 ConFunMin*Class* **ConFunMin**

Constrained Minimization of Function using Augmented Lagrangian method.
(se Nocedal and Wright: Numerical optimization chap 17.4. and K.Madsen,

H.B. Nielsen, O.Tingleff Optimization with Constraints, 2nd Edition, march 2004 chap 4.1)

```
class ConFunMin : IFun
```

Constructors

```
ConFunMin(IconFun Fun)
```

Constructor

Parameters

Fun

Properties

EpsGradL

Lagrange iteration residual limit

```
double EpsGradL {set; get;}
```

EpsGradX

X iterations limit on gradient

```
double EpsGradX {set; get;}
```

EpsStepX

X iterations limit on step length

```
double EpsStepX {set; get;}
```

ExitReason

Reason for leaving Minimizer

```
string ExitReason {get;}
```

Iterations

Number of Iterations

```
int Iterations {get;}
```

LambdaE

Equality Lagrange multipliers

```
Vector LambdaE {set; get;}
```

LambdaI

Inequality Lagrange multipliers

```
Vector LambdaI {set; get;}
```

MaxIterations

Maximum number of iteration steps

```
int MaxIterations {set; get;}
```

Mu

Penalty of squared constraints

```
double Mu {set; get;}
```

Mu0

Initial penalty of squared constraints

```
double Mu0 {set; get;}
```

Ok

Minimize or Consistent results OK

```
bool Ok {get;}
```

ResGradL

Lagrange iteration residual

```
double ResGradL {get;}
```

ResGradX

X iteration residual on gradient

```
double ResGradX {get;}
```

ResStepX

X iteration residual on step length

```
double ResStepX {get;}
```

Trace

Trace iterations

```
bool Trace {set; get;}
```

TraceLevel

Trace level

```
int TraceLevel {set; get;}
```

X

Argmin X

```
Vector X {get;}
```

XIterations

Total number of iterations i XMinimizer

```
int XIterations {get;}
```

XMinimizer

Reference to Minimizer for augmented Lagrangian function

```
FunMin XMinimizer {get;}
```

Methods**Consistent**

Check consistence of function to be minimized

```
bool Consistent(Vector x)
```

Parameters

x X operation point

returns true if consistent

Consistent

Check consistence of function to be minimized

```
bool Consistent(Vector x, Vector dx)
```

Parameters

x X operation point
dx variations of x
returns true if consistent

Minimize

Minimize Constrained function

```
double Minimize(Vector X0)
```

Parameters

X0 Initial X
returns minimum value

Minimize

```
double Minimize(Vector X0, Vector XMin, Vector XMax)
```

Parameters

X0 Initial X guess
XMin Min values for X, null if no check wanted
XMax Max values for X, null if no check wanted
returns minimum value

12.6 Least Square Fitting

The class *LeastSquareFit* 12.7 is useful for fitting the parameters x of a given function to a set of experimental results.

An example is

$$fun(t, x) = x_2 e^{X_0 t} + x_3 e^{x_1 t} + \varepsilon(t) \quad (12.11)$$

where ε is measurement noise. The function *fun* must implement the *IFunt* interface as shown in the code example below:

```
/// <summary>
/// test function for parameter fitting, Least square fit
/// </summary>
public class Function : IFunt
{
```

```

    /// <summary>
    /// X dimension
    /// </summary>
    int IFunt.N
    {
        get
        {
            return 4;
        }
    }

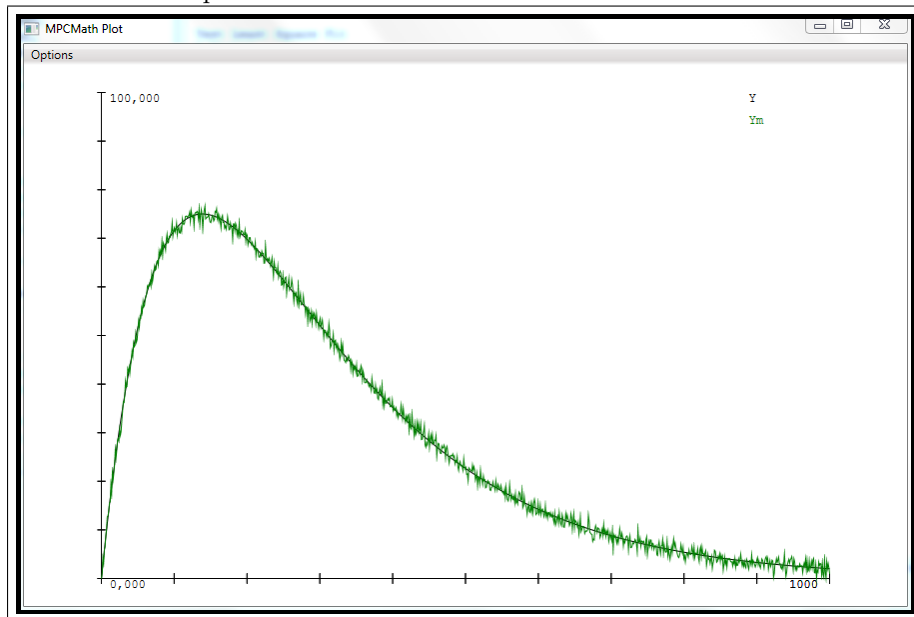
    /// <summary>
    /// Function value
    /// </summary>
    /// <param name="x"></param>
    /// <param name="t"></param>
    /// <returns></returns>
    double IFunt.Value(Vector x, double t)
    {
        return x[2] * Math.Exp(x[0] * t) + x[3] * Math.Exp(x[1] * t); ;
    }

    /// <summary>
    /// Jacobian
    /// </summary>
    /// <param name="x"></param>
    /// <param name="t"></param>
    /// <returns></returns>
    Vector IFunt.Dx(Vector x, double t)
    {
        Vector dx = new Vector(4);
        dx[2] = Math.Exp(x[0] * t);
        dx[3] = Math.Exp(x[1] * t);
        dx[0] = t * x[2] * dx[2];
        dx[1] = t * x[3] * dx[3];
        return dx;
    }

    /// <summary>
    /// Hessian (not implemented)
    /// </summary>
    /// <param name="x"></param>
    /// <param name="t"></param>
    /// <returns></returns>
    Matrix IFunt.DDx(Vector x, double t)
    {
        return null;
    }
}

```

Plot of the input data with and without noise.



The program below illustrates the use of *LeastSquareFit* to estimate the four parameters of the function

```
console.WriteLine("Test Least Square Fit");
double tau1 = 0.005;
double tau2 = 0.01;
double gain = 300.0;
double var = 1.0;
Vector X = new Vector(-tau1, -tau2, gain, -gain);
Vector.Show("X", X);

IFunt fun = new Function();

int m = 1000;
Vector T = new Vector(m);
Vector Y = new Vector(m);
Vector Ym = new Vector(m);
for (int i = 0; i < m; i++)
{
    double time = (1000.0 * i)/m;
    T[i] = time;
    Y[i] = fun.Value(X, time);
    Ym[i] = Y[i] + MPCMathLib.WhiteGaussianNoise(0.0, var);
}

console.Plot(new Plot(
    new PlotSeries("Y", 0.0, 100.0, Y),
```

```

        new PlotSeries("Ym", 0.0, 100.0, SeriesColor.Green, Ym)));

LeastSquareFit lsqfit = new LeastSquareFit(T, Ym, fun);
Vector X0 = new Vector(0.0, 0.0, 100.0, -100.0);
Vector Xhat = lsqfit.Fit(X0);

console.WriteLine("Iterations " + lsqfit.Iterations);
Vector.Show("X", X);
Vector.Show("Xhat", Xhat);

// constrain the function parameters

Vector XMin = new Vector(-10.0, -10.0, 0.0, -500.0);
Vector XMax = new Vector(0.0, 0.0, 290.0, -310.0);

Xhat = lsqfit.Fit(X0, XMin, XMax);
console.WriteLine();
console.WriteLine("Constrained result");
Vector.Show("XMin", XMin);
Vector.Show("XMax", XMax);
console.WriteLine("Iterations " + lsqfit.Iterations);
Vector.Show("X", X);
Vector.Show("Xhat", Xhat);
}

```

The console output is

```

Test Least Square Fit
X Vector
    -0,0050    -0,0100    300,0000    -300,0000
Iterations 50
X Vector
    -0,0050    -0,0100    300,0000    -300,0000
Xhat Vector
    -0,0050    -0,0098    311,1983    -311,1504

Constrained result
XMin Vector
    -10,0000    -10,0000         0    -500,0000
XMax Vector
         0         0    290,0000    -310,0000
Iterations 56
X Vector
    -0,0050    -0,0100    300,0000    -300,0000
Xhat Vector
    -0,0050    -0,0098    290,0000    -311,1504

```

The *LeastSquareFit* uses the Levenberg-Marquardt method as described in

[Nocedal and J. Wright, 2006] and [Madsen et al., 2004]

12.7 LeastSquareFit

Class LeastSquareFit

```
class LeastSquareFit
```

Constructors

```
LeastSquareFit(Vector T, Vector Y, IFunt Fun)
```

Constructor Least Square Fit

Parameters

T t values

Y y values

X0 Initial guess for parameter vector X

Fun Approximation function

Properties

EpsGrad

Gradient residual limit

```
double EpsGrad {set; get;}
```

EpsStep

Step size residual limit

```
double EpsStep {set; get;}
```

Iterations

Number of Iterations

```
int Iterations {get;}
```

MaxIterations

Maximum number of iteration steps

```
int MaxIterations {set; get;}
```

ResGrad

Gradient residual

```
double ResGrad {get;}
```

ResStep

Step size residual

```
double ResStep {get;}
```

Tau

Tau, initial mu factor. Tau equal 0.0 , start with Newton direction. Tau equal infinity, start with steepest descent direction.

```
double Tau {set; get;}
```

Trace

Trace iterations

```
bool Trace {set; get;}
```

Methods**Fit**

Fit parameter

```
Vector Fit(Vector X0)
```

Parameters

X0 Initial parameter guess

returns Local Minimizer parameters

Fit

Fit parameter, constrained

```
Vector Fit(Vector X0, Vector XMin, Vector XMax)
```

Parameters

X0 Initial parameter guess

XMin Min values for X, null if no check wanted

XMax Max values for X, null if no check wanted

returns Local Minimizer parameters

12.8 RiccatiSolver

Class RiccatiSolver

Riccati solver for extended linear-quadratic optimal control problem

```
class RiccatiSolver
```

Constructors

```
RiccatiSolver(Matrix Pn, BMatrix Q, BMatrix M, BMatrix R, BMatrix A, BMatrix B)
```

Constructor with Factorization

Parameters

- N Horizon
- Pn Penalty on final state
- Q State penalties
- M cross term penalties. (null is legal)
- R Penalties on inputs
- A Linear state equation matrices
- B Linear state equation matrices

Properties

K0

Initial Gain

```
Matrix K0 {get;}
```

p0

Initial cost

```
Vector p0 {get;}
```

P0

Initial cost P0

```
Matrix P0 {get;}
```

Methods

Solve

Solve Riccati series

```
void Solve(Vector X0, Vector pn, BVector q, BVector r,  
BVector b, out BVector U, out BVector X)
```

Parameters

X0	Initial state
pn	linear penalty on final state
q	linear penalty on state
r	linear penalty on inputs
b	linear state space term
U	Optimal inputs
X	Optimal plant state

Solve

Solve Riccati series

```
void Solve(Vector X0, Vector pn, BVector q, BVector r,  
BVector b, out BVector U, out BVector X, out BVector Lambda)
```

Parameters

X0	Initial state
pn	linear penalty on final state
q	linear penalty on state
r	linear penalty on inputs
b	linear state space term
U	Optimal inputs
X	Optimal plant state
Lambda	Dual variable

Chapter 13

QP and LP solvers

The *QPSolver* class is used to solve quadratic and linear optimization tasks. *QPSolver* implements a Primal-Dual Interior-Point Algorithm including Mehrotra's modifications. For quadratic problems the problem

$$\min_{x \in \mathbb{R}^n} \quad \frac{1}{2}x'Gx + g'x \quad (13.1a)$$

$$s.t. \quad Ax = b \quad (13.1b)$$

$$Cx \geq d \quad (13.1c)$$

is solved. For linear problems the problem

$$\min_{x \in \mathbb{R}^n} \quad g'x \quad (13.2a)$$

$$s.t. \quad Ax = b \quad (13.2b)$$

$$Cx \geq d \quad (13.2c)$$

is solved.

An quadratic problem example:

$$\min_{x \in \mathbb{R}^n} \quad \frac{1}{2}(x_1 - 1)^2 + \frac{1}{2}(x_2 - 2)^2 + \frac{1}{2}(x_3 - 1)^2 \quad (13.3a)$$

$$s.t. \quad x_1 = x_2 + 0.5 \quad (13.3b)$$

$$-1 \leq x_1 \leq 1 \quad (13.3c)$$

$$-1 \leq x_2 \leq 1 \quad (13.3d)$$

$$-1 \leq x_3 \leq 1 \quad (13.3e)$$

reformulations of (13.3) to the standard form (13.1) gives

$$G = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad (13.4)$$

$$g' = \begin{pmatrix} -1 & -2 & -1 \end{pmatrix} \quad (13.5)$$

$$A = \begin{pmatrix} 1 & -1 & 0 \end{pmatrix} \quad (13.6)$$

$$b' = \begin{pmatrix} 0.5 \end{pmatrix} \quad (13.7)$$

$$C = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{pmatrix} \quad (13.8)$$

$$d' = \begin{pmatrix} -1 & -1 & -1 & -1 & -1 & -1 \end{pmatrix} \quad (13.9)$$

The *MPCMath* code for this problem is

```
static void Work()
{
    console.WriteLine("Test QP solver");

    int nx = 3;
    int ny = 1;
    int nz = 3;
    Structure sx = new Structure(1, nx);
    Structure sy = new Structure(1, ny);
    Structure sz = new Structure(2, nz);
    BMatrix G = new BMatrix(MatrixForm.General, sx);
    BVector g = new BVector(sx);
    BMatrix A = new BMatrix(MatrixForm.General, sy, sx);
    BVector b = new BVector(sy);
    BMatrix C = new BMatrix(MatrixForm.General, sz, sx);
    BVector d = new BVector(sz);

    Matrix I = Matrix.UnityMatrix(nx);
    G[0, 0] = I;
    g[0] = new Vector(-1, -2, -1);

    A[0, 0] = new Matrix(MatrixForm.General, ny, nx);
    A[0, 0][0, 0] = 1.0;
    A[0, 0][0, 1] = -1.0;
    b[0] = new Vector(1, 0.5);

    C[0, 0] = I;
    C[1, 0] = -I;
    d[0] = new Vector(-1, -1, -1);
    d[1] = new Vector(-1, -1, -1);
}
```

```

        Structure.Show("sx", sx);
        Structure.Show("sy", sy);
        Structure.Show("sz", sz);
        BMatrix.Show("G", G);
        BVector.Show("g", g);
        BMatrix.Show("A", A);
        BVector.Show("b", b);
        BMatrix.Show("C", C);
        BVector.Show("d", d);

        QPSolver solver = new QPSolver(G, g, A, b, C, d);
        solver.Solve();
        BVector.Show("x optimal", solver.X);
        console.WriteLine();
        console.WriteLine("KKT conditions");
        console.Show("Dual gap", solver.My);
        BVector.Show("A*X - b", A * solver.X - b);
        BVector.Show("C*X - d", C * solver.X - d);
        BVector.Show("S*E", BVector.MulElements(solver.S , solver.Z));
    }
}

```

with the output

Test QP solver

sx

3

sy

1

sz

3

3

G Matrix.Form General[3,3]

1	0	0
0	1	0
0	0	1

g Vector

-1	-2,0000	-1
----	---------	----

A Matrix.Form General[1,3]

1	-1	0
---	----	---

b Vector

0,5000

C Matrix.Form General[6,3]

1	0	0
---	---	---

```

      0      1      0
      0      0      1
     -1      0      0
      0     -1      0
      0      0     -1

d Vector
      -1      -1      -1      -1      -1      -1
x optimal Vector
      1,0000      0,5000      0,9934

KKT conditions
Dual gap 0,0000
A*X - b Vector
      0
C*X - d Vector
      2,0000      1,5000      1,9934      0,0000      0,5000      0,0066
S*E Vector
      0,0000      0,0000      0,0000      0,0000      0,0000      0,0000

```

The interior point algorithm requires that the initial value of X strictly fulfils the inequality conditions $Cx > d$. The equality condition $Ax = b$ don't have to be fulfilled. For most MPC problems $X = 0$ fulfils this requirement.

13.1 Quadratic Optimization

Quadratic program

$$\min_{x \in \mathbb{R}^n} \frac{1}{2}x'Gx + g'x \quad (13.10a)$$

$$s.t. \quad Ax = b \quad (13.10b)$$

$$Cx \geq d \quad (13.10c)$$

Lagrange function

$$L(x, y, z) = \frac{1}{2}x'Gx + g'x - y'(Ax - b) - z'(Cx - d) \quad (13.11)$$

Optimality Conditions

$$\nabla_x L(x, y, z) = Gx + g - A'y - C'z = 0 \quad (13.12a)$$

$$\nabla_y L(x, y, z) = -(Ax - b) = 0 \quad (13.12b)$$

$$\nabla_z L(x, y, z) = -(Cx - d) \leq 0 \quad (13.12c)$$

$$z \geq 0 \quad (13.12d)$$

$$(Cx - d)_i z_i = 0 \quad i = 1, 2, \dots, m_c \quad (13.12e)$$

Slack variables

$$s \triangleq Cx - d \geq 0 \quad (13.13)$$

implies

$$-Cx + s + d = 0 \quad (13.14a)$$

$$s \geq 0 \quad (13.14b)$$

Optimality Conditions

$$r_L = Gx + g - A'y - C'z = 0 \quad (13.15a)$$

$$r_A = -Ax + b = 0 \quad (13.15b)$$

$$r_C = -Cx + s + d = 0 \quad (13.15c)$$

$$z \geq 0 \quad (13.15d)$$

$$s \geq 0 \quad (13.15e)$$

$$s_i z_i = 0 \quad i = 1, 2, \dots, m_c \quad (13.15f)$$

Notation

$$S = \begin{bmatrix} s_1 & & & \\ & s_2 & & \\ & & \ddots & \\ & & & s_{m_c} \end{bmatrix} \quad z = \begin{bmatrix} z_1 & & & \\ & z_2 & & \\ & & \ddots & \\ & & & z_{m_c} \end{bmatrix} \quad e = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} \quad (13.16)$$

The complementarity conditions

$$s_i z_i = 0 \quad i = 1, 2, \dots, m_c \quad (13.17)$$

can be expressed as

$$SZe = 0 \quad (13.18)$$

The optimality conditions can be expressed as

$$r_L = Gx + g - A'y - C'z = 0 \quad (13.19a)$$

$$r_A = -Ax + b = 0 \quad (13.19b)$$

$$r_C = -Cx + s + d = 0 \quad (13.19c)$$

$$r_{SZ} = SZe = 0 \quad (13.19d)$$

$$s \geq 0, \quad z \geq 0 \quad (13.19e)$$

which is the same as

$$F(x, y, z, s) = \begin{bmatrix} r_L \\ r_A \\ r_C \\ r_{SZ} \end{bmatrix} = \begin{bmatrix} Gx + g - A'y - C'z \\ -Ax + b \\ -Cx + s + d \\ SZe \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad (13.20a)$$

$$(z, s) \geq 0 \quad (13.20b)$$

Solve $F(x, y, z, s) = 0$ such that $(z, s) \geq 0$ using Newton's method.

13.2 Matrix Factorization

Matrix factorizations are used by *QPSolver* to speed up calculation. The KKT equations

$$\begin{bmatrix} G & -A' & -C' & 0 \\ -A & 0 & 0 & 0 \\ -C & 0 & 0 & I \\ 0 & 0 & S & Z \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \\ \Delta s \end{bmatrix} = \begin{bmatrix} -r_L \\ -r_A \\ -r_C \\ -\bar{r}_{SZ} \end{bmatrix} \quad (13.21)$$

can be factorized as follows. The fourth equation gives

$$S\Delta z + Z\Delta s = -\bar{r}_{SZ} \Rightarrow \Delta s = -Z^{-1}\bar{r}_{SZ} - Z^{-1}S\Delta z \quad (13.22)$$

Substitution in the third equation gives

$$-r_C = -C'\Delta x + \Delta s = -C'\Delta x - Z^{-1}S\Delta z - Z^{-1}\bar{r}_{SZ} \quad (13.23)$$

and

$$-C'\Delta x - Z^{-1}S\Delta z = -r_C + Z^{-1}\bar{r}_{SZ} \quad (13.24)$$

Augmented form

$$\begin{bmatrix} G & -A' & -C' \\ -A & 0 & 0 \\ -C & 0 & -Z^{-1}S \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \end{bmatrix} = \begin{bmatrix} -r_L \\ -r_A \\ -r_C + Z^{-1}\bar{r}_{SZ} \end{bmatrix} \quad (13.25)$$

13.3 KKT solvers

QPSolver solves the KKT equations (13.20) by calling a *KKTSolver* routine. The KKT solver determines the type of optimization problem quadratic, linear, second order cone problems. The KKT solver must implement the interface *IKKTSolver* making it possible to switch between KKT solvers and making it possible to produce custom KKT solvers exploiting matrix structures.

The relevant *KKTSolver* is passed to *QPSolver* using a constructor. The two following calls are equivalent (as *KKTSolver* is the default KKT solver for *QPSolver*)

```
QPSolver solver = new QPSolver(G, g, A, b, C, d);
QPSolver solver = new QPSolver(new KKTSolver(G, g, A, b, C, d));
```

The *IKKTSolver* defines three methods used by *QPSolver*. The KKT solves the augmented KKT equations (13.25).

- Calculate the residuals r_L , r_A and r_C .
- Factorize

$$\begin{bmatrix} G & -A' & -C' \\ -A & 0 & 0 \\ -C & 0 & -Z^{-1}S \end{bmatrix} \quad (13.26)$$

- Calculate Newton search directions Δx , Δy and Δz

The default KKT Solver perform a further factorization of (13.25) to the form:

$$\begin{bmatrix} G + C' (S^{-1}Z) C & -A' \\ -A & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} -r_L + C (S^{-1}Z) (r_C - Z^{-1} \bar{r}_{SZ}) \\ -r_A \end{bmatrix} \quad (13.27)$$

The main computational effort for QPSolver and the associated KKT Solver is the computation of the term $G + C' (S^{-1}Z) C$ and the factorization of (13.27) using a SymmetricalBlockEquationSolver 8.6.

MPCMath automatically exploits structures in the block matrices, but further exploitation of matrix structures or other factorization methods can be implemented by writing a new KKT Solver. The source code for the default KKT Solver is given in appendix F.

13.4 QPSolver

Class QPSolver

IP Solver class Primal-Dual Interior-Point Algorithm

```
class QPSolver
```

Constructors

```
QPSolver(BMatrix G, BVector g, BMatrix A, BVector b, BMatrix C, BVector d)
: this(new KKT Solver(G, g, A, b, C, d))
```

Constructor with default KKT Solver

min. $X'Gx + g'x$

s.t. $Ax = b$

$Gx \geq d$

Parameters

G G coefficients

g g coefficients

A A coefficients

b b coefficients

C C coefficients

d d coefficients

```
QPSolver(IKKT Solver Solver)
```

Solver with specific KKT solver

Parameters

Solver KKT solver

Properties**Disabled**

Disabled inequality constraints. Included = 0, disabled = 1;

```
BVector Disabled {get;}
```

ErrorLimit

Error limit

```
double ErrorLimit {set; get;}
```

Iterations

Iterations in last solve call

```
int Iterations {get;}
```

MaxIterations

Maximum iterations

```
int MaxIterations {set; get;}
```

Mu

Duality gap

```
double Mu {get;}
```

S

Slack variables for (Cx - d) reference

```
BVector S {set; get;}
```

Time

Execution Time for last Solve call

```
TimeSpan Time {get;}
```

TraceIterations

Trace iterations

```
bool TraceIterations {set; get;}
```

TraceLevel

Trace level 0 trace residual 1 trace residual and x 2 trace residual and x,y,z,s 3 trace residual, x,y,z,s and detailed info

```
int TraceLevel {set; get;}
```

X

Solution vector x

```
BVector X {set; get;}
```

Y

Lagrange multiplier for $(Ax - b)$ reference

```
BVector Y {set; get;}
```

Z

Lagrange multiplier for $(CX-d)$ reference

```
BVector Z {set; get;}
```

Methods**Solve**

Solve

```
void Solve()
```

13.5 IKKTSolver*Interface IKKTSolver*

KKT Solver interface for solving KKT equation system in augmented form, see equation (13.25)

```
interface IKKTSolver
```

Properties**d**

```
BVector d {get;}
```

Strcx

X structure

Structure Strcx {get;}

Strcy

Y structure

Structure Strcy {get;}

Strcz

Z structure

Structure Strcz {get;}

Methods**Factorize**

Factorize KKT system

void Factorize(BVector SInvZ)

Parameters

SInvZ $S^{-1}Z$ diagonal vector**Residuals**

Calculate residulas

```
void Residuals(BVector x, BVector y, BVector z, BVector s,
out BVector rL, out BVector rA, out BVector rC, out BVector rSZ)
```

Parameters

```
x    state variables
y    Lagrange multipliers for equality conditions
z    Lagrange multipliers for inequality conditions
s    Slack variables
rL   residuals
rA   residuals
rC   residuals
rSZ  residuals
```

Solve

Find Newton search directions

```
void Solve(BVector rL, BVector rA, BVector rC,
out BVector DX, out BVector DY, out BVector DZ)
```

Parameters

```
rL    residual
rA    residual
rC    residual
DX    X search direction
DY    Y search direction
DZ    Z search direction
```

13.6 KKTSolver*Class* **KKTSolver**

Quadratic Program KKT solver, default KKT solver for QPSolver. Q must be positive semidefinit. Solves Linear problems by setting Q = null.

```
class KKTSolver : IKKTSolver
```

Constructors

```
KKTSolver(BMatrix G, BVector g, BMatrix A,
BVector b, BMatrix C, BVector d)
```

Constructor for KKT solver

Parameters

```
G    G coefficients
g    g coefficients
A    A coefficients
b    b coefficients
C    C coefficients
d    d coefficients
```

13.7 LPKKTSolver*Class* **LPKKTSolver**

Linear problem KKT solver for problems defined in equation (13.2)

```
class LPKKTSolver : IKKTSolver
```

Constructors

LPKKTsolver(BVector g, BMatrix A, BVector b, BMatrix C, BVector d)

Constructor, Linear Programming KKTsolver

Parameters

g g coefficients
A A coefficients
b b coefficients
C C coefficients
d d coefficients

Chapter 14

Model Predictive Control, MPC

This chapter describes how to set up MPC controller using *MPCMath* . The implementation of MPC is based on the MiMoMPC object. Source code for MiMoMPC is available for the *MPCMath* user, making it possible to tailor the MPC controllers specifications and functionality to the users requirement

The MiMoMPC class implements both conventional MPC and Soft Constrained MPC [Prasath and Jørgensen, 2010], [Prasath et al., 2010] . The conventional MPC has a quadratic penalty function and the Soft Constrained MPC has a dead band zone around the set point where the penalty for not reaching the exact set point is low.

The plant is assumed to be a linear state space system in innovation form.

$$\mathbf{x}_{k+1} = A\mathbf{x}_k + Bu_k + K\epsilon_k \quad (14.1a)$$

$$\mathbf{y}_k = C\mathbf{x}_k + \epsilon_k \quad (14.1b)$$

where $\mathbf{x}_k$ is the plant state vector, u_k the manipulated variables. ϵ_k noise and $\mathbf{y}_k$ is the observed plant output. Knowing the present plant state $\mathbf{x}_k$ and the measured plant output $\mathbf{y}_k$ the noise ϵ_k can be estimated from

$$\epsilon_k = y_k - CAx_k \quad (14.2)$$

The MPC control problem is formulated as minimization of

$$\min_{\{y,u,\eta\}} \phi = \frac{1}{2} \sum_{k=0}^{N-1} (\|y_{k+1} - r_{k+1}\|_{Q_y}^2 + \|\Delta u_k\|_{S_u}^2) + \frac{1}{2} \sum_{k=1}^N \|\eta_k\|_{S_\eta}^2 \quad (14.3a)$$

subject to the constraints

$$\mathbf{x}_{k+1} = A\mathbf{x}_k + Bu_k + K\epsilon_k \quad k = 0, \dots, N-1 \quad (14.3b)$$

$$\mathbf{y}_k = C\mathbf{x}_k + \epsilon_k \quad k = 0, \dots, N \quad (14.3c)$$

$$u_{\min} \leq u_k \leq u_{\max} \quad k = 0, \dots, N-1 \quad (14.3d)$$

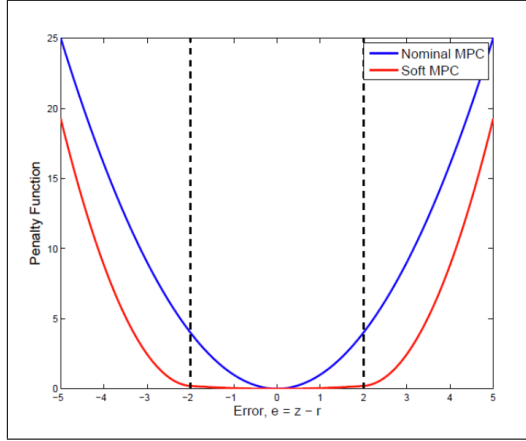
$$\Delta u_{\min} \leq \Delta u_k \leq \Delta u_{\max} \quad k = 0, \dots, N-1 \quad (14.3e)$$

$$y_k \leq y_{\max} + \eta_k \quad k = 1, \dots, N \quad (14.3f)$$

$$y_k \geq y_{\min} - \eta_k \quad k = 1, \dots, N \quad (14.3g)$$

$$\eta_k \geq 0 \quad k = 1, \dots, N \quad (14.3h)$$

In which $\Delta u_k = u_k - u_{k-1}$. The term $\|y_{k+1} - r_{k+1}\|_{Q_y}^2$ in (14.3a) penalizes the deviation between plant output and the reference r_k . The term $\|\Delta u_k\|_{S_u}^2$ penalizes the movements of the manipulated variables. $y_{\min}$ and $y_{\max}$ are the soft constraint limits on the controlled variables. η_k is the violation of soft constraints. $\|\eta_k\|_{S_\eta}^2$ is the penalty for violation of the soft constraints.



The state space description of the plant (14.1) can be rearranged to

$$y_k = CA^k x_0 + CA^{k-1} K \epsilon_0 + \sum_{i=1}^{k-1} CA^{k-i-1} B u_i \quad 1 \leq k \leq N \quad (14.4)$$

where the predicted future value of y_{k+i} is a function of x_k , ϵ_k and the future values of the manipulated variables u_k .

Defining $y_{free,k} = CA^k x_0 + CA^{k-1} K \epsilon_0$ and $H_i = CA^{i-1} B$ (The Markov parameters) gives

$$y_k = y_{free,k} + \sum_{j=1}^i H_i u_{j-1} \quad 1 \leq i < N \quad (14.5)$$

Define the vectors Y, Y_{free}, R and η

$$Y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix} \quad Y_{free} = \begin{bmatrix} y_{free,1} \\ y_{free,2} \\ \vdots \\ y_{free,N} \end{bmatrix} \quad R = \begin{bmatrix} r_1 \\ r_2 \\ \vdots \\ r_N \end{bmatrix} \quad U = \begin{bmatrix} u_0 \\ u_1 \\ \vdots \\ u_{N-1} \end{bmatrix} \quad \eta = \begin{bmatrix} \eta_1 \\ \eta_2 \\ \vdots \\ \eta_N \end{bmatrix} \quad (14.6)$$

and the matrix Γ

$$\Gamma = \begin{bmatrix} H_1 & 0 & \dots & 0 & 0 \\ H_2 & H_1 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ H_{N-1} & H_{N-2} & & H_1 & 0 \\ H_N & H_{N-1} & \dots & H_2 & H_1 \end{bmatrix} \quad (14.7)$$

Then the prediction from (14.5) can be expressed as

$$Y = Y_{free} + \Gamma U \quad (14.8)$$

Define the matrix Λ and vector I_0 by
 Λ and I_0 by

$$\Lambda = \begin{bmatrix} I & 0 & 0 & \dots & 0 & 0 \\ -I & I & 0 & \dots & 0 & 0 \\ 0 & -I & I & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & I & 0 \\ 0 & 0 & 0 & \dots & -I & I \end{bmatrix} \quad I_0 = \begin{bmatrix} I \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad (14.9)$$

Define $\mathcal{Q}_y$, $\mathcal{S}_u$ and $\mathcal{S}_\eta$

$$\mathcal{Q}_y = \begin{bmatrix} Q_y & & & \\ & Q_y & & \\ & & \ddots & \\ & & & Q_y \end{bmatrix} \quad \mathcal{S}_u = \begin{bmatrix} S_u & & & \\ & S_u & & \\ & & \ddots & \\ & & & S_u \end{bmatrix} \quad \mathcal{S}_\eta = \begin{bmatrix} S_\eta & & & \\ & S_\eta & & \\ & & \ddots & \\ & & & S_\eta \end{bmatrix} \quad (14.10)$$

Then the objective function (14.3) may be expressed as

$$\begin{aligned} \phi &= \frac{1}{2} \sum_{k=0}^{N-1} \|y_{k+1} - r_{k+1}\|_{\mathcal{Q}_y}^2 + \|\Delta u_k\|_{\mathcal{S}_u}^2 + \frac{1}{2} \|\eta_{k+1}\|_{\mathcal{S}_\eta}^2 \\ &= \frac{1}{2} \|Y - R\|_{\mathcal{Q}_y}^2 + \frac{1}{2} \|\Lambda U - I_0 u_{-1}\|_{\mathcal{S}_u}^2 + \frac{1}{2} \|\eta\|_{\mathcal{S}_\eta}^2 \\ &= \frac{1}{2} \|Y_{free} + \Gamma U - R\|_{\mathcal{Q}_y}^2 + \frac{1}{2} \|\Lambda U - I_0 u_{-1}\|_{\mathcal{S}_u}^2 + \frac{1}{2} \|\eta\|_{\mathcal{S}_\eta}^2 \\ &= \frac{1}{2} U' (\Gamma' \mathcal{Q}_y \Gamma + \Lambda' \mathcal{S}_u \Lambda) U + (\Gamma' \mathcal{Q}_y (Y_{free} - R) - \Lambda' \mathcal{S}_u I_0 u_{-1})' U \quad (14.11) \\ &\quad + \left(\frac{1}{2} \|Y_{free} - R\|_{\mathcal{Q}_y}^2 + \frac{1}{2} \|I_0 u_{-1}\|_{\mathcal{S}_u}^2 \right) + \frac{1}{2} \eta' \mathcal{S}_\eta \eta \\ &= \frac{1}{2} U' G U + g' U + \rho + \frac{1}{2} \eta' \mathcal{S}_\eta \eta \\ &= \frac{1}{2} x' \bar{G} x + \bar{g}' x + \rho \end{aligned}$$

with

$$G = \Gamma' \mathcal{Q}_y \Gamma + \Lambda' \mathcal{S}_u \Lambda \quad (14.12a)$$

$$g = \Gamma' \mathcal{Q}_y (c - R) - \Lambda' \mathcal{S}_u I_0 u_{-1} \quad (14.12b)$$

$$\rho = \frac{1}{2} \|Y_{free} - R\|_{\mathcal{Q}_y}^2 + \frac{1}{2} \|u_{-1}\|_{\mathcal{S}_u}^2 \quad (14.12c)$$

$$x = \begin{bmatrix} U \\ \eta \end{bmatrix} \quad \bar{G} = \begin{bmatrix} G & 0 \\ 0 & \mathcal{S}_\eta \end{bmatrix} \quad \bar{g} = \begin{bmatrix} g \\ s_\eta \end{bmatrix} \quad (14.12d)$$

Consequently, we may solve MPC regulator problem (14.3) by solution of the following convex quadratic program

$$\min_x \quad \psi = \frac{1}{2} x' \bar{G} x + \bar{g}' x \quad (14.13a)$$

$$s.t. \quad x_{\min} \leq x \leq x_{\max} \quad (14.13b)$$

$$b_l \leq \bar{C} x \leq b_u \quad (14.13c)$$

in which

$$x_{\min} = \begin{bmatrix} U_{\min} \\ 0 \end{bmatrix} \quad x_{\max} = \begin{bmatrix} U_{\max} \\ \infty \end{bmatrix} \quad (14.14a)$$

$$b_l = \begin{bmatrix} \Delta U_{\min} \\ -\infty \\ Z_{\min} - c \end{bmatrix} \quad C = \begin{bmatrix} \Lambda & 0 \\ \Gamma & -I \\ \Gamma & I \end{bmatrix} \quad b_u = \begin{bmatrix} \Delta U_{\max} \\ Z_{\max} - c \\ \infty \end{bmatrix} \quad (14.14b)$$

In a model predictive controller only the first vector, u_0^* , of $U^* = [(u_0^*)' \quad (u_1^*)' \quad \dots \quad (u_{N-1}^*)']'$, is implemented on the process. At the next sample time the open-loop optimization is repeated with new information due to a new measurement.

14.1 MiMoMPC

Class MiMoMPC

Multiple Input Multiple Output MPC *MPCMath*

```
class MiMoMPC
```

Constructors

```
MiMoMPC(StateSpaceModel Model, int History, int Horizon,
Vector Theta, Vector Mu, Vector YMin, Vector YMax,
Vector Rho, Vector UMin, Vector UMax)
```

Constructor for MiMoMPC object

Parameters

Model	Controller model
History	History length
Horizon	Control horizon, N
Theta	Reference violation penalty, Q_y
Mu	Soft constraint violation penalty, S_η
YMin	Y Max soft constraint, y_{max}
YMax	Y Min soft constraint, y_{min}
Rho	U change penalty, S_u
UMin	U max value, u_{max}
UMax	U min value, u_{max}

Properties

Alfa

Error integration coefficient Alfa = 0.0 Dedadbeat elimination of stationary error
 Alfa = 1.0 No elimination of stationary error

Vector Alfa {set; get;}

History

Length of History part

int History {get;}

Horizon

Length of control Horizon

int Horizon {get;}

Model

reference to Arx model for MPC controller

StateSpaceModel Model {get;}

Mu

Soft Constrained penalty, Q_η

Vector Mu {set; get;}

Rho

Delta U penalty, S_u

Vector Rho {set; get;}

SoftConStrained

MPC controller soft constrained

```
bool SoftConStrained {get;}
```

Solver

Reference to QP solver

```
QPSolver Solver {get;}
```

Theta

Delta Y penalty, Q_y

```
Vector Theta {set; get;}
```

U

Reference to U response

```
Matrix U {get;}
```

UMax

U maximum value, U_{max}

```
Vector UMax {set; get;}
```

UMin

U minimum value, U_{min}

```
Vector UMin {set; get;}
```

X

Reference to state

```
Vector X {get;}
```

YFree

Reference to YFree response

```
Matrix YFree {get;}
```

YMax

Soft Constrained Uppper Limit, y_{max}

Vector YMax {set; get;}

YMin

Soft Constrained Lower Limit, y_{min}

Vector YMin {set; get;}

YModel

Reference to YModel response

Matrix YModel {get;}

YPlant

Reference to YPlant response

Matrix YPlant {get;}

YRef

Reference to YRef

Matrix YRef {get;}

Methods**Next**

Next control step

Vector Next(Vector YPlant)

Parameters

YPlant Plant value

returns U value

NextY

Next expected Y value

Vector NextY()

SetRef

Set future reference

```
void SetRef(Matrix Ref)
```

Parameters

Ref future reference Matrix

SetRef

Set future reference

```
void SetRef(Vector Ref)
```

Parameters

Ref future reference value

14.2 LinearModel

Class LinearModel

Linear Model $dx/dt = A * X + B * U + W$
 $Y = C * X + V$

```
class LinearModel : IModel
```

Constructors

```
LinearModel(Matrix A, Matrix B, Matrix C, double T)
: this(A, B, C, T, T)
```

Constructor

Parameters

A System matrix A
 B System matrix A
 C System matrix A
 T Step size for NextSate

```
LinearModel(Matrix A, Matrix B, Matrix C, double T, double DT)
```

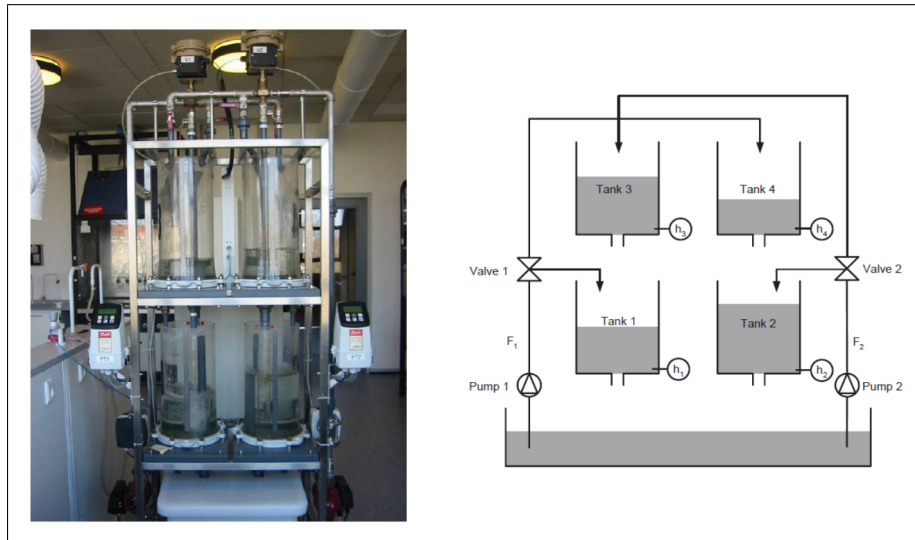
Constructor

Parameters

A System matrix A
 B System matrix A
 C System matrix A
 T Step size for NextSate
 DT Integration step size

14.3 Four Tank Process

The four tank process is an simple multi variable process used to demonstrate MIMO process dynamic in education . The four tank system was introduced by Johanson [2000] as a benchmark for control design.



The Controlled variables are the water levels in the four tanks, H_1 , H_2 , H_3 and H_4 . The manipulated variables are the two inflows, F_1 and F_2 .

The parameters in the IModel interface are

Parameters[0] cross sectional area of outlet a1 [cm²]
 Parameters[1] cross sectional area of outlet a2 [cm²]
 Parameters[2] cross sectional area of outlet a3 [cm²]
 Parameters[3] cross sectional area of outlet a4 [cm²]

Parameters[4] cross sectional area of inlet A1 [cm²]
 Parameters[5] cross sectional area of inlet A2 [cm²]
 Parameters[6] cross sectional area of inlet A3 [cm²]
 Parameters[7] cross sectional area of inlet A4 [cm²]

Parameters[8] valve position valve γ_1
 Parameters[9] valve position valve γ_2

14.4 FourTankProcess

Class FourTankProcess

Four Tank Process model

```
class FourTankProcess : IModel
```

Constructors

```
FourTankProcess(double T)
: this(T, false)
```

Constructor

Parameters

T Step size

```
FourTankProcess(double T, bool Extended)
```

Constructor

Parameters

T Step size

Extended Extended model

14.5 VanDerPol

Class **VanDerPol**

Van der Pol's problem $y''(t) = \mu(1 - y(t)^2)y'(t) - y(t)$

```
class VanDerPol : IModel
```

Constructors

```
VanDerPol(double Mu, double T)
```

Constructor

Parameters

Mu μ parameter

T Step size

Implementation of Van der Pol equation is shown in appendix E

Chapter 15

ODE solvers

15.1 Introduction

Ordinary Differential Equations can be solved using the ODESolver class. ODESolver implements a number of Explicit Runge-Kutta methods where the solutions can be found without iterations, and a number of ESDIRK methods, where the solution in each integration step requires a number of iteration steps. The simpler Explicit Runge-Kutta methods are suitable for differential equation system, where the eigenvalues have an equal order of magnitude. The ESDIRK methods are recommended for stiff differential equation systems where there is a large difference between the systems eigenvalues. The test example below shows how to integrate the Van der Pol equation, where the parameter μ determines the systems stiffness. $\mu = 0.0$ gives a non - stiff sinus curve, values of $\mu = 100.0$ gives a very stiff problem.

The differential equations to be integrated must be implemented using the *IModel* interface 11.4. Implementation of Van der Pol equation is shown in appendix E. The ERK methods does not require implementation of the Jacobian method, but the ESDIRK methods require that the Jacobian method is implemented.

The ODESolver class can be run with fixed step length and with adaptive step length, if you specify a tolerance for the integration. The step length is adjusted based on calculation of the index

$$r = \max \frac{e_i}{\max(abstol_i, |x_i| reltol_i)} \quad i \in 1, \dots, n \quad (15.1)$$

The error e_i is estimated based on comparing integration result with of one step with two integration steps with half the step length. Values of $r > 1.0$ rejects the step and decreases the step length. Values of $r \leq 1.0$ increases the step length and the step is accepted.

The program for integration of the Van der Pol equation is:

```
console.WriteLine("Test Ordinary Differential Equation solvers");

double dt = 0.01;
```

```

double T = 200.0;
ODEMethods method = ODEMethods.ESDIRK34;

// Create Van der Pol model
double mu = 10.0;
IModel fun = new VanDerPol(mu, dt);

double tol = 1.0e-6;
Vector absTol = new Vector(2, tol);
Vector relTol = new Vector(2, tol);
// Create ODE solver
ODESolver solver = new ODESolver(method, fun, dt, absTol, relTol);

// integrate Van der Pol
Vector tval = null;
Matrix X = null;
Vector XS = new Vector(fun.Dimension, 1.0);
Vector US = new Vector(fun.NU);
DateTime start = DateTime.Now;
solver.Integrate(T, XS, US, out tval, out X);
TimeSpan elapsed = DateTime.Now - start;

console.WriteLine();
console.WriteLine("Method " + method.ToString()
    + " absTol " + tol.ToString()
    + " Tolerance " + solver.Tolerance.ToString());
console.Show("Time ms ", elapsed.TotalMilliseconds);
console.Show("stepsize", solver.Step);
console.Show("steps", tval.Dimension);
console.Show("evaluations", solver.Evaluations);
console.Show("accepted", solver.AcceptedSteps);
console.Show("rejected", solver.RejectedSteps);
console.Show("Factorizations", solver.Factorizations);

// Plot results
Vector Y0 = X.GetRow(0);
Vector Y1 = X.GetRow(1);

console.Plot(new Plot(PlotType.XY,
    new PlotSeries("t", 0.0, T, tval),
    new PlotSeries("Y0", 0.0, 0.0, SeriesColor.Green, Y0)),
    new Plot(PlotType.XY,
    new PlotSeries("t", 0.0, T, tval),
    new PlotSeries("Y1", 0.0, 0.0, SeriesColor.Green, Y1)));
}

```

The console output is

Test Ordinary Differential Equation solvers

Method ESDIRK34 absTol 1E-06 Tolerance 1E-16

Time ms 2005,0000

stepsize 0,1242

steps 4099

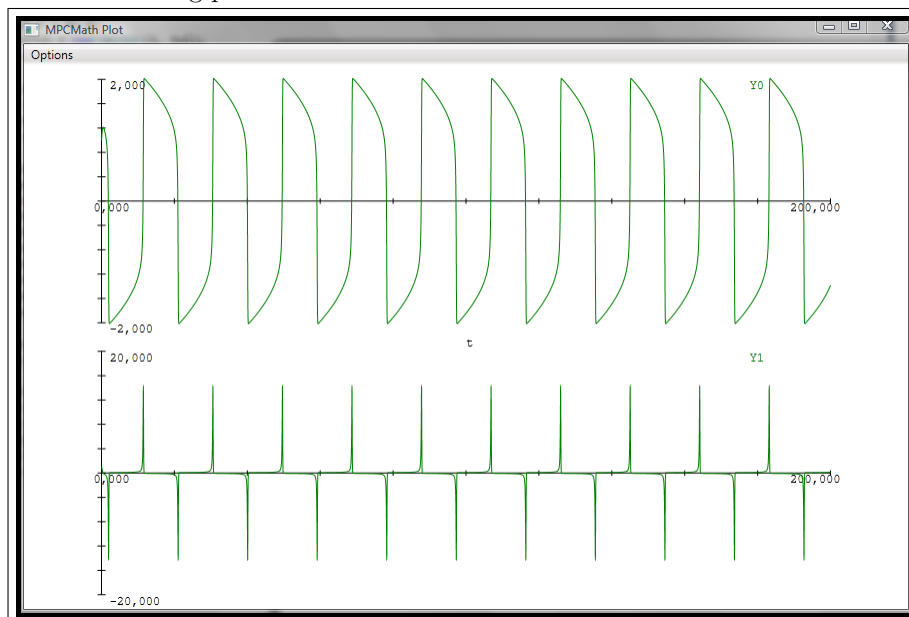
evaluations 208592

accepted 4098

rejected 1171

Factorizations 15848

The resulting plots are



15.2 ODEMethods

Enumeration ODEMethods

ODE solver methods enumeration

```
enum ODEMethods
```

Fields	
DOPRI5	DOPRI5(4) 7 stages, order 4
ERK1	Explicit Euler
ERK2A	Explicit Trapezoidal $\alpha = 1$;
ERK2B	Explicit Trapezoidal $\alpha =$;
ERK3A	Three stage ERK
ERK3B	Three stage ERK
ERK4A	Four Stage ERK
ERK4B	Four Stage ERK
ERK4C	Classical Runge Kutta (4 order, 4 tages)
ERK4D	Four Stage ERK
ESDIRK12	Explicit Singly Diagonally Implicit Runge-Kutta method 12
ESDIRK23	Explicit Singly Diagonally Implicit Runge-Kutta method 23
ESDIRK34	Explicit Singly Diagonally Implicit Runge-Kutta method 34

15.3 ODESolver

Class ODESolver

ODE solver for ordinary differential equations

```
class ODESolver
```

Constructors

```
ODESolver(ODEMethods Method, IModel Model, double Step)
: this(Method, Model, Step, null, null)
```

Constructor for fixed step size

Parameters

Method	Integration Method
Model	Differential equations
Step	Step size

```
ODESolver(ODEMethods Method, IModel Model,
double InitialStep, Vector AbsTol, Vector RelTol)
```

Constructor for adaptive step size

Parameters

Method	Integration Method
Model	Differential equations
Step	Initial step size
AbsTol	Absolute tolerance
RelTol	Relative tolerance

Properties

A

Butcher tableau A

Matrix A {get;}

AcceptedSteps

Number of Accepted steps

int AcceptedSteps {set; get;}

B

Butcher tableau B

Vector B {get;}

C

Butcher tableau C

Vector C {get;}

Evaluations

Number of evaluations of Model.Derivative

int Evaluations {set; get;}

Factorizations

Number of factorizations

int Factorizations {set; get;}

InfeasibleSteps

Number of Backtracks due to infeasible states

int InfeasibleSteps {set; get;}

Iterations

Number of Newton iterations used by ESDIRK step

int Iterations {set; get;}

MaxIterations

Max Iterartions for ESDIRK Newton steps

```
int MaxIterations {set; get;}
```

NewtonErrors

Number of Newton iterations not converged

```
int NewtonErrors {set; get;}
```

RejectedSteps

Number of Rejected steps

```
int RejectedSteps {set; get;}
```

Step

Integration step size

```
double Step {set; get;}
```

Tolerance

Tolerance for ESDIRK Newton iterations

```
double Tolerance {set; get;}
```

Methods**Integrate**

Integrate ODE for T time units, and return all integration steps

```
void Integrate(double T, Vector X0, Vector U, out Vector Steps, out Matrix Xnew)
```

Parameters

T	Integration time
X0	Initial state
U	Manipulated variables
Steps	Integration steps (time values)
Xnew	Integration states, stored column wise

Next

Integrate ODE for T time units and calculate Sensitivity matrices

```
void Next(double T, Vector X0, Vector U, out Vector X, out Matrix SX, out Matrix SU)
```

Parameters

T Integration time
 X0 Initial state
 U Manipulated variables
 X Next State
 SX Sensitivity $DX/DX0$
 SU Sensitivity $DX/DX0$

Next

Integrate ODE for T time units

```
Vector Next(double T, Vector X0, Vector U)
```

Parameters

T Integration time
 X0 Initial state
 U Manipulated variables
 returns new State

Chapter 16

MPCMathLib

16.1 MPCMathLib-MinNorm

Class MPCMathLib

Minimize $\text{sum}(\text{norm}(y - A \cdot x))$

```
partial class MPCMathLib
```

Methods

MinNorm

Minimize $\text{sum}(\text{norm}(y - A \cdot x))$

```
static Vector MinNorm(Norm norm, double gamma, Matrix A, Vector y)
```

Parameters

norm	Norm
gamma	Huber norm coefficient
A	Coefficient matrix
y	vector of observed values
returns	vector x

Enumeration Norm

```
[0] = Zeroes;  
g[1] = Ones;  
Cqp[0, 0] = +A;  
Cqp[0, 1] = I;  
Cqp[1, 0] = -A;  
Cqp[1, 1] = I;
```

```

Cqp[2, 1] = I;
d[0] = y;
d[1] = -y;
errorlim = 1.0e-9;
break;
case Norm.NormInfinity:
Ones = new Vector(1, 1.0);
Zeroes = new Vector(cols);
I = Matrix.Random(rows, 1, 1.0, 1.0);
strx = Structure.Values(cols, 1);
stry = new Structure(0);
strz = Structure.Values(rows, rows, 1);
g = new BVector(strx);
Aqp = new BMatrix(MatrixForm.General, stry, strx);
b = new BVector(stry);
Cqp = new BMatrix(MatrixForm.General, strz, strx);
d = new BVector(strz);
Gqp = new BMatrix(strx);
g[0] = Zeroes;
g[1] = Ones;
Cqp[0, 0] = +A;
Cqp[0, 1] = I;
Cqp[1, 0] = -A;
Cqp[1, 1] = I;
Cqp[2, 1] = new Matrix(1, 1.0);
d[0] = y;
d[1] = -y;
break;
case Norm.Huber:
strx = Structure.Values(cols, rows, rows, rows);
stry = Structure.Values(rows);
strz = Structure.Values(rows, rows, rows, rows);
Gqp = new BMatrix(MatrixForm.Diagonal, strx);
g = new BVector(strx);
Aqp = new BMatrix(MatrixForm.General, stry, strx);
b = new BVector(stry);
Cqp = new BMatrix(MatrixForm.General, strz, strx);
d = new BVector(strz);
I = Matrix.UnityMatrix(rows);
Vector gammaI = new Vector(rows, gamma);
Gqp[1, 1] = I;
g[2] = +gammaI;
g[3] = +gammaI;
Aqp[0, 0] = A;
Aqp[0, 1] = I;
Aqp[0, 2] = I;
Aqp[0, 3] = -I;
b[0] = y;
Cqp[0, 1] = I;
Cqp[1, 1] = -I;

```

```

Cqp[2, 2] = I;
Cqp[3, 3] = I;
d[0] = -gammaI;
d[1] = -gammaI;
break;
}
KKTsSolver kktsolver = new KKTsSolver(Gqp, g, Aqp, b, Cqp, d);
qpsolver = new QPSolver(kktsolver);
qpsolver.MaxIterations = 100;
qpsolver.ErrorLimit = errorlim;
qpsolver.Solve();
x = qpsolver.X[0];
if (qpsolver.Iterations >= qpsolver.MaxIterations)
{
throw new MPCMathException(" + norm.ToString(), ");
}
}
else
{
lssolver = new LeastSquareEquationSolver(A);
x = lssolver.Solve(y);
}
return x;
}
else
{
throw new MPCMathException(", ");
}
}
}
enum Norm

```

Fields	
Huber	Huber norm
Norm1	Norm 1, absolute value
Norm2	Norm2, Euclid norm
NormInfinity	Norm Infinity

16.2 MPCMathLib-RK4

Class MPCMathLib

```
partial class MPCMathLib
```

Methods

RK4

Runge Kutta 4 integrator

```
static Vector RK4(IModel model, Vector X, Vector U, double T)
```

Parameters

model	Process model
X	Present state
U	Manipulated variables
T	End time
returns	State at endtime

RK4

Runge Kutta 4 integrator

```
static Vector RK4(IModel model, Vector X, Vector U, double T, double step)
```

Parameters

model	Process model
X	Present state
U	Manipulated variables
T	End time
step	step size
returns	State at endtime

16.3 MPCMathLib-Discretize

Class MPCMathLib

```
partial class MPCMathLib
```

Methods

Discretize

Calculate discretized linear equations from Linear differential equations.

$$\frac{dx(t)}{dt} = Ax(t) + Bu(t)$$

$$x(k+1) = AdX(k) + BdU(k)$$

```
static void Discretize(Matrix A, Matrix B, double T,
out Matrix AD, out Matrix BD)
```

Parameters

A	A matrix
B	B matrix
T	Discretization time
AD	AD matrix for discrete system
BD	BD matrix for discrete system

Discretize

Discretize linear or unlinear Plant Model around an operating point

$$\frac{dx(t)}{dt} = Fun(X(t), U(t))$$

$$x(k+1) = AdX(k) + BdU(k)$$

```
static void Discretize(IModel model, Vector XS, Vector US,
double T, out Matrix AD, out Matrix BD)
```

Parameters

model	IModel model
XS	State of operating point
US	State of manipulated var operating point
AD	AD matrix for discrete system
BD	BD matrix for discrete system

16.4 MPCMathLib-SteadyState*Class* **MPCMathLib**

Find stationary state of Process with manipulated variable US given

```
partial class MPCMathLib
```

Properties**Iterations**

Iterations

```
static int Iterations {get;}
```

MaxIterations

Maximum number of iterations (default 100)

```
static int MaxIterations {set; get;}
```

Methods

SteadyState

Find Steady State of Process

```
static Vector SteadyState(IModel model, Vector XI, Vector US)
```

Parameters

model	Plant model
XI	Initial guess off state variables
US	Manipulated variables
returns	XS state variable

16.5 MPCMathLib-Util

Class MPCMathLib

MPCMathLib Utility routines

```
partial class MPCMathLib
```

Properties

MaxError

Maximum error for comparing doubles

```
static double MaxError {set; get;}
```

Ran

Random number generator

```
static Random Ran {get;}
```

Methods

BFGS

BFGS Hessian approximation. (Nocedal, Wright: Numerical optimization, chapter 6)

```
static Matrix BFGS(Matrix H, Vector s, Vector y)
```

Parameters

H	Hessian apprimation matrix to be updated, must be symmetrical and positive definit
s	iteration step
y	curvature change

BFGSInv

BFGS Inverse Hessian approximation update. (Nocedal, Wright: Numerical optimization, chapter 6)

```
static Matrix BFGSInv(Matrix H, Vector s, Vector y)
```

Parameters

Hinv Inverse Hessian approximation matrix to be updated, must be symmetrical and positive definite
 s iteration step
 y curvature change

Difference

Difference, $\text{Math.Abs}(a - b) / (1.0 + \text{Math.Abs}(b))$

```
static double Difference(double a, double b)
```

Parameters

a
 b

Difference

Difference between vectors, $\text{Max} (\text{Abs}(A[i] - B[i]) / (1.0 + \text{Abs}(B[i])))$

```
static double Difference(Vector A, Vector B)
```

Parameters

A
 B

Difference

Difference from Zero, $\text{Max} (\text{Abs}(A[p]) / (1.0 + \text{Math.Abs}(A[p])))$

```
static double Difference(Vector A)
```

Parameters

A

Distribution

Density curve for a values

```
static Vector Distribution(int N, double Range, Vector a)
```

Parameters

N Groups
 Range max variation
 a input vector

Fact

Factorial $n!$

```
static int Fact(int n)
```

Parameters
 n input
 returns $n!$

Gauss

Gauss density function

```
static double Gauss(double x, double mean, double var)
```

Parameters
 x Input
 mean Mean
 var Variation

GaussDistribution

Gauss Distribution

```
static Vector GaussDistribution(int N, double Range, double mean, double var)
```

Parameters
 N Groups
 Range max variation
 mean
 var

IsMiminimum

Is X a local minimizer for Fun

```
static bool IsMiminimum(IFun Fun, Vector X, double Eps)
```

Parameters
 Fun Function (Ifun or IConFun)
 X Minimizer candidate
 Eps Accuracy
 returns true if x is a local minimizer

IsMiminimum

Is X a local minimizer for Fun

```
static bool IsMiminimum(IFun Fun, Vector X, double Eps, bool Trace)
```

Parameters

Fun Function (IFun or IConFun)
 X Minimizer candidate
 Eps Accuracy
 Trace Trace result
 returns true if x is a local minimizer

IsMiminimum

Is X a local minimizer for Fun

```
static bool IsMiminimum(IFun Fun, Vector X, Vector XMin, Vector XMax, double Eps)
```

Parameters

Fun Function (IFun or IConFun)
 X Minimizer candidate
 XMin Lower box limits
 XMax Upper box limits
 Eps Accuracy
 returns true if x is a local minimizer

IsMiminimum

Is X a local minimizer for Fun

```
static bool IsMiminimum(IFun Fun, Vector X, Vector XMin, Vector XMax, double Eps, bool Trace)
```

Parameters

Fun Function (IFun or IConFun)
 X Minimizer candidate
 XMin Lower box limits
 XMax Upper box limits
 Eps Accuracy
 Trace Trace result
 returns true if x is a local minimizer

IsZero

Is Zero, test wheter $-MaxError \geq a \leq MaxError$

```
static bool IsZero(double a)
```

Parameters

a
 returns true if $-a \leq MaxError$

IsZero

Is Zero, test wheter $-MaxError \geq a \leq Eps$

```
static bool IsZero(double a, double Eps)
```

Parameters

a
Eps
returns true if $\|a\|_j = \text{Eps}$

KalmanGain

Kalman filter gain

```
static Matrix KalmanGain(Matrix a, Matrix c, Vector q, Vector r)
```

Parameters

a system matrix ($x_+ = a*x + b*u$)
c system matrix ($y = c*x$)
q Process Noise
r Measurement Noise

KalmanGain

Kalman filter gain

```
static void KalmanGain(Matrix a, Matrix c,  
Vector q, Vector r, out Matrix K, out Matrix P)
```

Parameters

a system matrix ($x_+ = a*x + b*u$)
c system matrix ($y = c*x$)
q Process Noise
r Measurement Noise
K Kalman Gain
P Error Covariance

LimitViolation

Test for Limit violation. $\text{Math.Abs}(a - b) / (1.0 + \text{Math.Abs}(b)) \geq \text{Limit}$;

```
static bool LimitViolation(double a, double b, double Limit)
```

Parameters

Value
Limit

LimitViolation

Test for Limit violation. $\text{Abs}(A[i] - B[i]) / (1.0 + \text{Abs}(B[i])) \geq \text{Limit}$ for all elements in Vectors A and B

```
static bool LimitViolation(Vector A, Vector B, double Limit)
```

Parameters

A
B
Limit

ProperZero

Return a proper 0.0 if $|a|$ less than MaxError

```
static double ProperZero(double a)
```

Parameters

a

ProperZero

Return a proper 0.0 if $|a|$ less than Eps

```
static double ProperZero(double a, double Eps)
```

Parameters

a

Eps

PseudoRandom

Set Pseudo random mode if seed != 0 Needed for debugging purposes, get the same random number every time the program is started

```
static void PseudoRandom(int seed)
```

Parameters

seed

WhiteGaussianNoise

White Gaussian Noise

```
static double WhiteGaussianNoise(double mean, double var)
```

Parameters

var mean

mean variance

WhiteGaussianNoise

White Gaussian Noise

```
static Vector WhiteGaussianNoise(Vector mean, Vector var)
```

Parameters

mean mean

var variance

Chapter 17

Miscellaneous functions

17.1 Complex

Class **complex**

Basic complex functionality

[Serializable]

```
class complex : ICommon<complex> , IComparable<complex>, IEquatable<complex>
```

Constructors

```
complex()
```

Constructor

```
complex(double RV, double IV)
```

Constructor

Parameters

RV real value

IV imaginary value

```
static implicit operator complex(double a)
```

implicit conversion from double to complex

Parameters

a input var

Properties

Arg

Angle

```
double Arg {get;}
```

Con

Complex conjugate

```
complex Con {get;}
```

IV

Imaginary value

```
double IV {set; get;}
```

Mod

Modulus

```
double Mod {get;}
```

RV

Real value

```
double RV {set; get;}
```

Methods**Equals**

```
override bool Equals(object o)
```

Exp

Complex exponential

```
static complex Exp(complex a)
```

Parameters

a input var

returns exp(a)

GetHashCode

```
override int GetHashCode()
```

Log

Complex logarithm

```
static complex Log(complex a)
```

Parameters

a input var

returns Log(a)

operator

Unary + operator/ clone

```
static complex operator +(complex a)
```

Parameters

a Input complex

returns Out complex

operator

Add complex numbers

```
static complex operator +(complex a, complex b)
```

Parameters

a input a

b input b

returns a+b

operator

Add complex numbers

```
static complex operator +(double a, complex b)
```

Parameters

a input a

b input b

returns a+b

operator

Add complex numbers

```
static complex operator +(complex a, double b)
```

Parameters

a input a

b input b

returns a + b

operator

Unary - operator

```
static complex operator -(complex a)
```

Parameters

a input var

returns -a

operator

Subtract complex numbers

```
static complex operator -(complex a, complex b)
```

Parameters

a input var

b input var

returns a-b

operator

Subtract complex numbers

```
static complex operator -(double a, complex b)
```

Parameters

a input var

b input var

returns a-b

operator

Subtract complex numbers

```
static complex operator -(complex a, double b)
```

Parameters

a input var

b input var

returns a-b

operator

Multiply complex numbers

```
static complex operator *(complex a, complex b)
```

Parameters

a input var

b input var

returns a*b

operator

Multiply complex numbers

```
static complex operator *(double a, complex b)
```

Parameters

a input var

b input var

returns a*b

operator

Multiply complex numbers

```
static complex operator *(complex a, double b)
```

Parameters

a input var

b input var

returns a*b

operator

Divide complex numbers

```
static complex operator /(complex a, complex b)
```

Parameters

a input var

b input var

returns a/b

operator

Divide complex numbers

```
static complex operator /(double a, complex b)
```

Parameters

a input var

b input var

returns a/b

operator

Divide complex numbers

```
static complex operator /(complex a, double b)
```

Parameters

a input var

b input var

returns a/b

operator

Equal operator

```
static bool operator ==(complex a, complex b)
```

Parameters

a input var
b input var
returns true if equal

operator

Not equal operator

```
static bool operator !=(complex a, complex b)
```

Parameters

a input var
b input var
returns true if not equal

Sqrt

Complex Sqrt

```
static complex Sqrt(complex a)
```

Parameters

a input var
returns sqrt(a)

17.2 Polynomial*Class* **Polynomial**

Polynomial class with $p(x) = c[0] + c[1] * x + c[2] * x^2 + \dots + c[n] * x^n$

```
[Serializable]
```

```
class Polynomial
```

Constructors

```
Polynomial():this(0)
```

Constructor fro serialization

```
Polynomial(double Gain, CVector Roots)
```

Constructor from a vector of roots

Parameters

dim Dimension

Gain Gain

Roots Roots

`Polynomial(Vector A)`

Constructor from vector of coefficients

Parameters

A *Coefficients* $a_0 + a_1 * x + a_2 * x^2 + \dots$

`Polynomial(params double[] vals)`

Constructor from an array of coefficients

Parameters

vals *Coefficients* $a_0 + a_1 * x + a_2 * x^2 + \dots$

Properties

Coefficients

Polynomial coefficients

`Vector Coefficients {get;}`

Gain

Gain

`double Gain {get;}`

N

Polynomial dimension

`int N {get;}`

Roots

Polynomial roots

`CVector Roots {get;}`

Methods

Add

Add polynomials

```
static Polynomial Add(Polynomial a, Polynomial b)
```

Parameters

a Input a

b Input b

returns a+b

AssertEqual

Assert equal show actual and expected if not equal

```
static bool AssertEqual(string text, Polynomial actual, Polynomial expected)
```

Parameters

text text string

actual actual value

expected expected value

Clone

Clone Polynomial

```
static Polynomial Clone(Polynomial a)
```

Parameters

a input value

returns Cloned polynomial

Derivative

Derivative of polynomial

```
Polynomial Derivative()
```

Parameters

returns Derivative polynomial

Div

Divide Polynomials, $\text{res} = a/b$

```
static Polynomial Div(Polynomial a, Polynomial b)
```

Parameters

a Polynomial a

b Polynomial b

returns a/b

Equal

Equal Polynomial

```
static bool Equal(Polynomial a, Polynomial b)
```

Parameters

a Input a

b Input b

returns true if equal

Mul

Multiply Polynomial with constant $\text{res} = \text{fak} * b$

```
static Polynomial Mul(double fak, Polynomial b)
```

Parameters

fak Constant

b Polynomila

Mul

Multiply Polynomials operator $\text{res} = P * Q$

```
static Polynomial Mul(Polynomial P, Polynomial Q)
```

Parameters

P Polynomial P

Q Polynomial Q

returns $P * Q$

operator

Clone operator, $\text{res} = +a$

```
static Polynomial operator +(Polynomial a)
```

Parameters

a input a

returns Cloned polynomial

operator

Negate operator, $\text{res} = -a$;

```
static Polynomial operator -(Polynomial a)
```

Parameters

a input a

returns $-a$

operator

Add operator

```
static Polynomial operator +(Polynomial a, Polynomial b)
```

Parameters

a Input a

b Input b

returns a+b

operator

Subtract operator

```
static Polynomial operator -(Polynomial a, Polynomial b)
```

Parameters

a Input a

? Input b

returns a-b

operator

Multiply operator, res = fak*a

```
static Polynomial operator *(double fak, Polynomial a)
```

Parameters

fak Constant

a Polynomial a

returns fak*a

operator

Multiply operator, res = a*fak

```
static Polynomial operator *(Polynomial a, double fak)
```

Parameters

a Polynomial a

fak Constant

returns a*fak

operator

```
static Polynomial operator /(Polynomial a, double fak)
```

Parameters

a Polynomial a

fak constant

returns a/fak

operator

Multiply Polynomials operator $\text{res} = P * Q$

```
static Polynomial operator *(Polynomial P, Polynomial Q)
```

Parameters

P Polynomial P

Q Polynomial Q

returns $P * Q$

operator

Div operator, $\text{res} = a/b$

```
static Polynomial operator /(Polynomial a, Polynomial b)
```

Parameters

a Polynomial a

b Polynomial b

returns a/b

operator

Remainder operator

```
static Polynomial operator %(Polynomial a, Polynomial b)
```

Parameters

a Polynomial a

b Polynomial a

returns a

Random

Generate random Polynomial

```
static Polynomial Random(int n, double min, double max)
```

Parameters

n dimension

min min coefficients

max max coefficients

Rem

Remainder

```
static Polynomial Rem(Polynomial a, Polynomial b)
```

Parameters

a Polynomial a

b Polynomial a

returns a

Show

Show Polynomial

```
static void Show(string txt, Polynomial a)
```

Parameters

txt text string
a Polynomial

Sub

Subtract

```
static Polynomial Sub(Polynomial a, Polynomial b)
```

Parameters

a Input a
b Input b
returns a-b

Truncate

Truncate Polynomial to effective dimension

```
void Truncate()
```

Value

Polynomial value

```
double Value(double x)
```

Parameters

x input x
returns Value

Value

Polynomial value

```
complex Value(complex x)
```

Parameters

x Complex value
returns Complex value

17.3 DelayChain

Class DelayChain

Delay Chain for delay of manipulated variables

```
[Serializable]  
class DelayChain
```

Constructors

```
DelayChain()
```

Constructor for serialization

```
DelayChain(int Delay)
```

Constructor

Parameters

Delay Delay

```
DelayChain(int Delay, double Value)
```

Constructor

Parameters

Delay Delay

Value Initial value

Properties

Delay

Delay

```
int Delay {set; get;}
```

ResetValue

Reset value

```
double ResetValue {set; get;}
```

Methods

AssertEqual

assert equal, Compare two DelayChains and output value if not equal

```
static void AssertEqual(string text, DelayChain actual, DelayChain expected)
```

Parameters
text Text string
actual DelayChain with actual values
expected DelayChain with expected values

Equal

Equal, compare two Delay chains. (Error limit given in MPCMathLib.MaxError)

```
static bool Equal(DelayChain y, DelayChain x)
```

Parameters
y DelayChain y
x DelayChain x
returns true if equal

Get

Get delayed value

```
double Get()
```

Parameters
returns u[k - delay]

Get

Get delayed value with offset

```
double Get(int Offset)
```

Parameters
Offset Offset
returns u[k + Offset - delay]

Reset

Reset DelayChain

```
void Reset()
```

Set

Set value

```
void Set(double value)
```

Parameters
value u[k]

Show

Show DelayChain

```
static void Show(string text, DelayChain x)
```

Parameters

text text string

x DelayChain object

17.4 HuberFunction*Class* **HuberFunction**

Huber penalty function.

$$val = \sum(\phi(Y - A * X))$$

$$\phi(e) = e^2, |e| \leq \gamma$$

$$\phi(e) = \gamma * |e| - \gamma^2, |e| > \gamma$$

```
class HuberFunction : IFun
```

Constructors

```
HuberFunction(double Gamma, Matrix A, Vector Y)
```

Constructor

Parameters

Gamma Threshold, γ

A Coefficient matrix

Y Value vector

Properties**A**

A in $val = \sum(\phi(Y - A * X))$

```
Matrix A {get;}
```

Gamma

Threshold, γ

```
double Gamma {set; get;}
```

Y

y in $val = \sum(\phi(Y - A * X))$

```
Vector Y {get;}
```

17.5 Spline and Cubic Smoothing Spline functions

A spline function is a curve constructed from polynomial segments (Splines) that are subject to continuity conditions at their joints. A spline function is approximating a set of data $(x_0, y_0), (x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ of the function $y = y(x)$. The Spline function segments are:

$$S_i(x) = a_i * (x - x_{0_i})^3 + b_i * (x - x_{0_i}) * 2 + c_i * (x + x_{0_i}) + d_i \quad (17.1)$$

For a simple Spline function the spline function segments fulfils the conditions:

$$S_{i-1}(x_i) = S_i(x_i) \quad (17.2a)$$

$$S'_{i-1}(x_i) = S'_i(x_i) \quad (17.2b)$$

$$S''_{i-1}(x_i) = S''_i(x_i) \quad (17.2c)$$

$$S_i(x_i) = y_i \quad (17.2d)$$

In the cubic smoothing Spline function the spline function is allowed to deviate from the data points. The cubic smoothing spline function is minimizing the value

$$\min_{\{x\}} L(x) = \lambda \sum_0^n \sigma_i (y_i - S_i(x_i))^2 + (1 - \lambda) \int_{x_0}^{x_n} S''(x)^2 \quad (17.3)$$

In the limiting case, where $\lambda = 0.0$ the spline function will become a straight line. At the other extreme, where $\lambda = 1.0$ the smoothing spline converges to the simple spline function.

17.6 Spline

Class Spline

Spline function. $S(x) = a * (x - x_0)^3 + b * (x - x_0) * 2 + c * (x + x_0) + d$

```
class Spline : IComparable
```

Properties

A

Spline cubic coefficient

```
double A {set; get;}
```

B

Spline square coefficient

```
double B {set; get;}
```

C

Spline linear coefficient

```
double C {set; get;}
```

D

Spline constant coefficient

```
double D {set; get;}
```

H

Spline length

```
double H {set; get;}
```

Sigma

Spline data weight

```
double Sigma {set; get;}
```

X0

Spline starting point

```
double X0 {set; get;}
```

Y

Spline data value

```
double Y {set; get;}
```

Methods**Position**

Is x inside Spline interval

```
int Position(double X)
```

Parameters

X

returns -1 if x below spline, 0 if inside, 1 if above

Value

Spline value

```
double Value(double X)
```

Parameters

X

17.7 CubicSpline*Class* **CubicSpline**

Smoothing Cubic Spline

```
class CubicSpline
```

Constructors

```
CubicSpline(Vector X, Vector Y) : this(X, Y, 1.0, null)
```

Constructor for non smoothing Spline

Parameters

X data values

Y funtion value

```
CubicSpline(Vector X, Vector Y, double Lambda)
```

```
: this(X, Y, Lambda, null)
```

Constructor for smoothing Spline

Parameters

X data values

Y funtion value

Lambda Smoothing factor

```
CubicSpline(Vector X, Vector Y, double Lambda, Vector Sigma)
```

Constructor for smoothing Spline

Parameters

X data values

Y funtion value

Lambda Smoothing factor

Sigma Weighths for data points

Properties**Lambda**

Smoothing factor ($0.0 < \textit{Lambda} \leq 1.0$)

```
double Lambda {set; get;}
```

Splines

Array of Spline functions

```
Spline[] Splines {get;}
```

X

Spline x points

```
Vector X {get;}
```

Y

Spline y points

```
Vector Y {get;}
```

Methods**Value**

Value of smoothed spline

```
double Value(double x)
```

Parameters

x

17.8 BarrierFunction*Class* **BarrierFunction**

Logarithmic barrier function.

Given a set of inequality conditions $B \geq Ax$ the barrier function is

$$\phi = - \sum_{i=0}^{N-1} \ln(b_i - a_i x) \quad (17.4)$$

where a_i is the i row of A and N is the number of rows in A

```
class BarrierFunction : IFun
```

Constructors

BarrierFunction(Matrix A, Vector B)

Constructor

Parameters

A A matrix

B B vector

17.9 Reports

In C# it is surprisingly difficult to program a simple plain report with some values in nice columns. One way is to use Windows Presentation Foundation, WPF, function to make the columns, but in many cases its an over kill. The ReportBuilder object provides a simple solution to the problem.

The following program demonstrates the use of the report builder object

```
console.WriteLine("Test Report Builder");
console.WriteLine();

string[] names = { "Sten", "Peter", "John", "Jorgen", "Niels" };
double[] values = { 1000.00, 33234.12, 27945.28, 1677.45, 56.44 };

ReportBuilder rb = new ReportBuilder(500);

int pname = 0;          // position of name column
int pvalue = 15;         // position of value column
int fieldsize = 10;
int digits = 2;
rb.Write(pname, "Name");
rb.Write(pvalue, "value");
rb.NewLine();
for (int p = 0; p < names.Length; p++)
{
    rb.Write(pname, names[p]);
    rb.Write(pvalue, values[p], fieldsize, digits);
    rb.NewLine();
}

console.WriteLine(rb.ToString());
```

with the output

Test Report Builder

Name	value
Sten	1000,00
Peter	33234,12

John	27945,28
Jorgen	1677,45
Niels	56,44

17.10 ReportBuilder

Class ReportBuilder

Report builder

```
class ReportBuilder
```

Constructors

```
ReportBuilder(int Capacity)
```

Constructor for Report builder

Parameters

Capacity Capacity of report builder

Methods

NewLine

New line

```
void NewLine()
```

NewLine

New Lines

```
void NewLine(int lines)
```

Parameters

lines Number of new lines

ToString

ToString

```
override string ToString()
```

Write

Write

```
void Write(int Pos, object Value)
```

Parameters

Pos

Value

Write

Write string

```
void Write(int Pos, string Value)
```

Parameters

Pos position

Value text string

Write

Write double

```
void Write(int Pos, double Value, int Size, int Digits)
```

Parameters

Pos Position

Value Value

Size Field size

Digits Digits

17.11 File I/O and serialization

Serialization refers to the term of converting or eventually transferring the state of an object into a stream (e.g. a file stream or a memory stream). The stream contains all the information needed to reconstruct, deserialize, the object for later use. This can be used to store object on disc or transferring object between applications or computers.

MPCMath objects defined as [Serializable] can be serialized using .NET's BinaryFormatter. *MPCMath* provides the BinaryIO class to save and read object from binary files and the XmlIO class to save and read from XML files. As shown below.

```
Vector x = Vector.Random(10, -100.0, 100.0);
```

```
// save and read binary file
```

```
string file = "object.txt";
```

```

BinaryIO.Save(file, x);
Vector y = (Vector)BinaryIO.Read(file);
Vector.AssertEqual("BinaryIO Vector", y, x);

// save and read XML file
string xfile = "xobject.xml";
XmlIO<Vector>.Save(xfile, x);
y = XmlIO<Vector>.Read(xfile);
Vector.AssertEqual("XmlIO Vector", y, x);

```

Some of the *MPCMath* objects marked as [Serializable] can be serialized using .NET's SoapFormatter as ASCII streams. The SoapFormatter does not support generic classes as *BMatrix*, *BVector*.

To support XmlSerialization a number of properties with names like xvl and xvla has been defined. These routines are not for general use.

17.12 BinaryIO

Class BinaryIO

Binary Input Output routines. Saves or read a serializable object to or from binary file.

```
static class BinaryIO
```

Methods

Read

Read object from file

```
static Object Read(string FileName)
```

Parameters

FileName	File name
returns	read object

Save

Save object to disc

```
static void Save(string FileName, Object ObjGraph)
```

Parameters

FileName	File name
ObjGraph	Object to be saved

17.13 XmlIO

Class XmlIO

XML Input Output routines. Saves or read a serializable object to or from XML file.

```
static class XmlIO<T> where T : new()
```

Methods

Read

Read object from file

```
static T Read(string FileName)
```

Parameters

FileName File name

returns read object

Save

Save object to disc

```
static void Save(string FileName, T ObjGraph)
```

Parameters

FileName File name

ObjGraph Object to be saved

17.14 MPCMathLib-JacobianEstimation

Class MPCMathLib

Calculate Jacobian estimation

```
partial class MPCMathLib
```

Methods

JuEstimate

Jacobian Estimation, , for debugging of models implemented using IModel interface. The initial variation of U, DU is reduced until norm of error is less than eps

```
static Matrix JuEstimate(IModel model, Vector X, Vector U, Vector DU, double eps)
```

Parameters

model	Plant model
X	Operating state
U	Manipulated variables
DU	Initial Variation of U
eps	Error limit
returns	Approximated Jacobian

JxEstimate

Jacobian Estimation, , for debugging of models implemented using IModel interface. The initial variation of X, DX is reduced until norm of error is less than eps

```
static Matrix JxEstimate(IModel model, Vector X, Vector U, Vector DX, double eps)
```

Parameters

model	Plant model
X	Operating state
U	Manipulated variables
DX	Initial Variation of X
eps	Error limit
returns	Approximated Jacobian

Bibliography

- Stephen Boyd and Lieven Vanderberghe. *Convex Optimization*. Cambridge University Press, 2004.
- Jakob Kjøbsted Huusom, Niels Kjølstad Poulsen, Sten Bay Jørgensen, and John Bagterp Jørgensen. Offset-free model predictive control based on arx models. In *American Control Conference, Marriot Waterfront, Baltimore, MD, USA, June-July 02, 2010*.
- Intel. Intel math kernel library. "<http://software.intel.com/en-us/articles/intel-mkl/>".
- K. H. Johanson. The quadruple-tank process: A multivariable laboratory process with an adjustable zero. *IEEE Transactions on control systems technology*, 8(3):456–465, 2000.
- John Bagterp Jørgensen. *Moving Horizon Estimation and Control*. Ph.D thesis, Department of Chemical Engineering, Technical University of Denmark, 2004.
- K.Madsen, H.B Nielsen, and O.Tinglev. *Optimization optimization with Constraints*. IMM, DTU, 2004.
- K. Madsen, H.B. Nielsen, and O. Tinglev. *MethiOp for Non-Linear Least Squares Problems*. IMM - DTU, 2004.
- Jorge Nocedal and Stephen J.Wright. *Numerical Optimization*. Springer, 2006.
- G. Prasath and John Bagterp Jørgensen. Soft constraints for robust mpc of uncertain systems. In *DYCOPS 2010, pag 288-293, Leuven, Belgium, 2010*.
- G. Prasath, B. Recke, M. Chidambaram, and J.B. Jørgensen. Application of soft constrained mpc to a cement mill circuit. In *9th International Symposium on Dynamics and Control of Process Systems, DYCOPS 2010, 2010*.

Appendix A

Installing *MPCMath*

This appendix describes how to get *MPCMath* up and running on your machine. If you get any problems or have any questions, please don't hesitate to contact me.

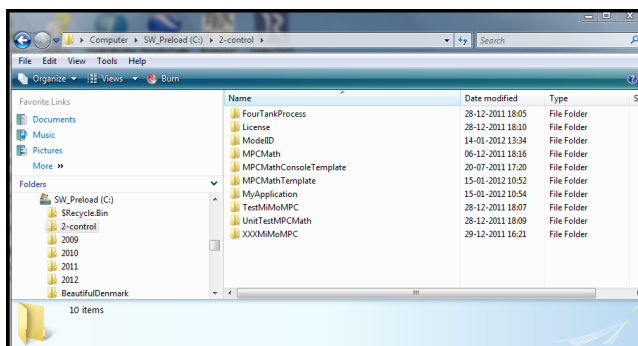
A.1 Prerequisites

MPCMath requires that visual Studio is installed on your computer. A free version of Microsoft Visual C# 2010 Express can be downloaded from Microsoft <http://www.microsoft.com/express/Downloads/#2010-Visual-CS>.

Remember to register Microsoft Visual C# 2010 Express, otherwise it becomes inactive after 30 days.

A.2 Installing *MPCMath*

Unzip the file "2-control 14-feb-2011.zip" to c:\creating the folder c:\2-control. (If the file you received has the extension .sip, rename it to .zip). Copy the license file MPCMath00007.lic to c:\2-control\ License. That's all, now you should be ready to use *MPCMath* . The c:\2-control folder should look something like this



The c:\2-control folder contains the following folders:

License	Holding the license file
<i>MPCMath</i>	The binary <i>MPCMath</i> .dll
MPCMathConsoleTemplate	Template for Console based <i>MPCMath</i> applications. Can be used with MPCMath Runtime licenses.
MPCMathWPFTemplate	Template for WPF based application, this is the normal starting point for development of a new <i>MPCMath</i> application. Using the WPF based console requires a Demo or a Development license.
TestMiMoMPC	Multiple Input Multiple Output MPC program, used in the?Introduction to <i>MPCMath</i> ? chapter.
UnitTestMPCMath	Unit test of the <i>MPCMath</i> library. It contains many examples of calls to <i>MPCMath</i> .
XXMiMoMPC	As TestMiMoMPC with included source code for MI-MOMPC object

A.3 Windows x86 and x64 platforms

MPCMath supports both the 32 bits platform x86 and the 64 bits platform x64 . The x86 platform is the default platform, where the MPCMath routines are delivered in the MPCMath.Dll. The x64 platform requires the use of the library MPCMathx64.dll Both libraries can be found in the c:\2-control\MPCMath directory. The Intel MKL routines are provided in the folders c:\2-control\MPCMath\MKLx86 and c:\2-control\MPCMath\MKLx64 folders.

A.4 Trouble shooting, Debugging and Exceptions

The MPCMath.dll requires that the directories "C:\2-control\MPCMath" and "C:\2-control\License" exists. Its also required that a valid license file is present in the License directory.

If the MPCMathTemplate runs properly and display the license information. Then the License information is ok. If MPCMathTemplate executes and MPCMathConsoleTemplate don't execute properly, then there is a problems with the Intel MKL libraries:c:\2-control\MPCMath\MKLx86 and c:\2-control\MPCMath\MKLx64 folders. Use "DependencyWalker" to test whether your operating system is missing some sub dll in MPCmathMKLx86.dll or MPCMathMKLX64.dll DependencyWalker is free sw which can be downloaded from the net.

If you move programs as MIMOMPC ,UnitTestMPCMath or MPCMathTemplate away from the c:\2-control directory the reference til MPCMath must be redefined. Its done in the solution explorer Add references. Browse to the routine "c:\2-control\MPCMath*MPCMath* .dll"

The *MPCMath* function checks parameter values and error situations. If an error happens the function creates an MPCMathException, as show below where the vector x and y should have the same dimension.

```

static void Work()
{
    console.WriteLine("Test Exceptions");

    try
    {
        Vector x = Vector.Random(4, -100.0, 100.0);
        Vector y = Vector.Random(3, -100.0, 100.0);
        double res = x * y;
    }
    catch (MPCMathException e)
    {
        console.WriteLine(e.Message);
    }
}

```

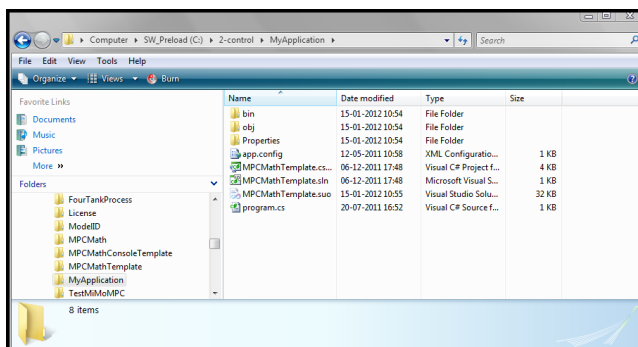
the console print out is

Test Exceptions
MPCMath.Vector.Mul/Illegal dimensions

Normally its easier debug programs without the *try/catch* construction, and use the information from the call stack to locate the sinner. In the final program its advisable to include *try/catch* construction.

A.5 MPCMathTemplate

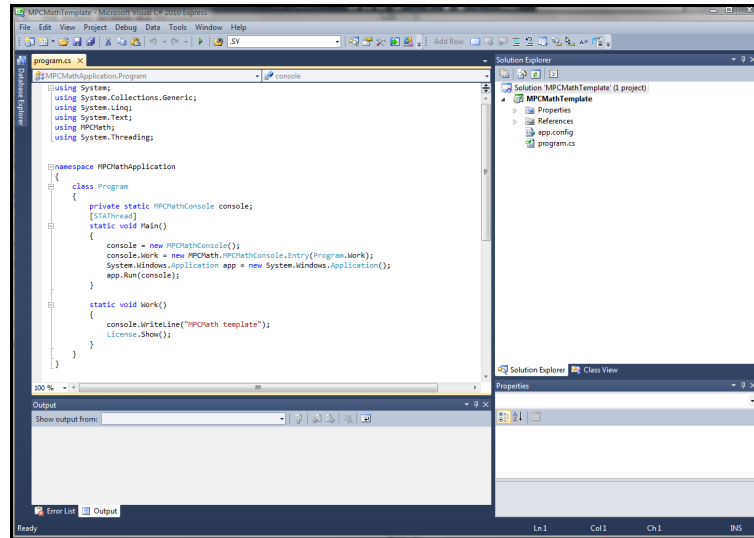
This is the starting point for development of a new *MPCMath* application. Copy the contents of MPCMathTemplate to a new folder named "MyApplication".



Double click the solution file MPCMathTemplate.sln to start Microsoft Visual C# 2010 Express. Click the menu View / Other Windows / Solution Explorer to open the Solution explorer.

If "MyApplication" is not located in the c:\2-control directory the reference to MPCMath must be redefined. Its done in the solution explorer Add references. Browse to the routine "c:\2-control\MPCMath\MPCMath.dll"

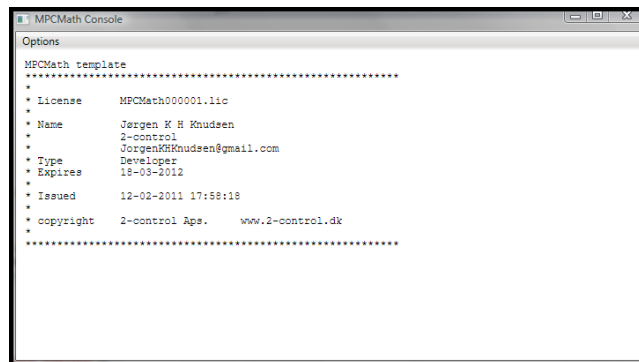
Rename the Solution and the project to MyApplication. (Right click / Rename). Open the main program program.cs by double clicking it. Now the screen should look like this:



The first routine `staticvoidMain()` starts the *MPCMath* console in a separate thread, this section should not be modified by you. Your code should be placed in the `staticvoidWork()` routine.

Start the application by clicking the run button. 

If everything is ok, the *MPCMath* console should look like this:

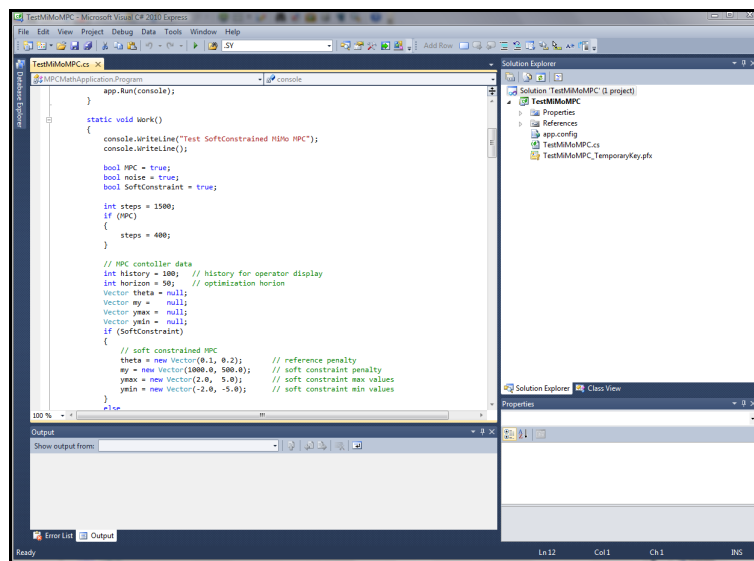


Closing the *MPCMath* console window or clicking the  button returns the system to development mode.

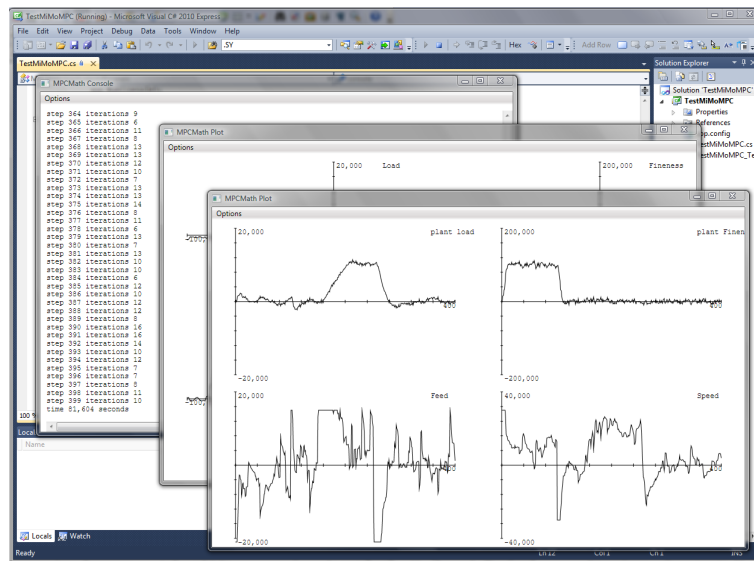
Before starting developing new code, I recommend that you study the Test-MiMoMPC application and the chapter "Introduction to *MPCMath*".

A.6 TestMiMoMPC

This folder contains the Multiple Input Multiple Output MPC program, used in the "Introduction to *MPCMath*" document. Start the application by opening the solution file "TestMiMoMPC.sln".



Press the run button to start the MiMo MPC controller.



The MPC controller runs 500 steps and display an operator display with history and predicted performance. After 500 step an overview plot is displayed. Press the stop button to revert to edit mode.

Play with the application parameter and study the code.

```
bool MPC = true;           // MPC or trackin exaple
bool noise = true;        // noise on/off
bool SoftConstraint = true; // Conventional or softconstrained MPC
```

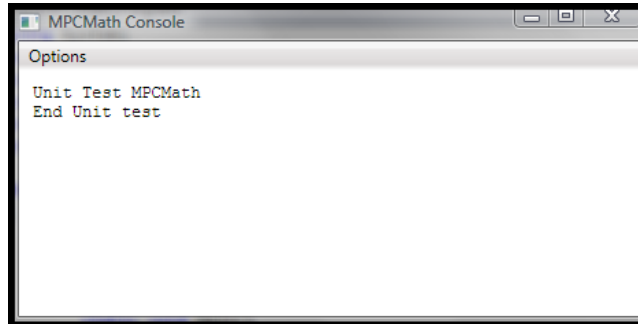
The MiMoMPC history and prediction horizon are defined by:

```
// MPC contoller data
int history = 100; // history for operator display
int horizon = 50;  // optimization horion
```

The history value must be equal or larger the the state space model's time delay.

A.7 UnitTestMPCMath

The UnitTestMPCMath application test the functionality of *MPCMath* . Run the application. If everything is OK the output will look like this:



In the source code for the unit test presents a lot of *MPCMath* code examples.

The solution file UnitTestMPCMath test the 32 version (x86) of MPCMath. The solution file UnitTestMPCMathx64 test the 64 bits (x64) version

A.8 MPCMathConsoleTemplate

This template can be used to develop or test *MPCMath* applications without *MPCMath* 's console. Typically intended for the final runtime systems.

Appendix B

Debugging tool for Console applications

B.1 Cnsl

Class Cnsl

MPCMath Cnsl class for test and debugging of MPCMath programs. Running DOS Command console. It requires a valid Demo or Developer license. Output is ignored when running under runtime licenses

```
static class Cnsl
```

Properties

Digits

Digits for doubles

```
static int Digits {set; get;}
```

FieldSize

Field size of vector and matrix display

```
static int FieldSize {set; get;}
```

NumbersPerLine

Numbers per Line

```
static int NumbersPerLine {set; get;}
```

TraceText

Trace console output to file "console.txt" in bin/debug or bin/release directory

```
static bool TraceText {set; get;}
```

Methods**AssertEqual**

Assert equal for bool variable Show actual and expected if they are not equal

```
static void AssertEqual(string text, bool actual, bool expected)
```

Parameters

text	variable description
actual	actual value
expected	expected value

AssertEqual

Assert equal for integers Show actual and expected if they are not equal

```
static void AssertEqual(string text, int actual, int expected)
```

Parameters

text	variable description
actual	actual value
expected	expected value

AssertEqual

Assert equal for integer array Show actual and expected if they are not equal

```
static void AssertEqual(string text, int[] actual, int[] expected)
```

Parameters

text	variable description
actual	actual value
expected	expected value

AssertEqual

Assert equal for double Show actual and expected if they are not equal

```
static void AssertEqual(string text, double actual, double expected)
```

Parameters

text	variable description
actual	actual value
expected	expected value

AssertEqual

Assert equal for complex Show actual and expected if they are not equal

```
static void AssertEqual(string text, complex actual, complex expected)
```

Parameters

text	variable description
actual	actual value
expected	expected value

DFormat

Format double variable

```
static string DFormat(double a)
```

Parameters

a	double variable
returns	Formatted output string

DFormat

Format double variable

```
static string DFormat(int Fieldsize, double a)
```

Parameters

Fieldsize	Field size
a	double variable
returns	Formatted output string

ReadLine

ReadLine from console

```
static void ReadLine()
```

Show

Show bool

```
static void Show(string text, bool a)
```

Parameters

text	Variable description
a	bool variable

Show

Show int, integer

```
static void Show(string text, int r)
```

Parameters

text Variable description

r integer variable

Show

Show int[], integer array

```
static void Show(string txt, int[] a)
```

Parameters

txt Variable description

a integer array

Show

Show double

```
static void Show(string text, double r)
```

Parameters

text Variable description

r double variable

Show

Show complex

```
static void Show(String text, complex c)
```

Parameters

text Variable description

c complex variable

Show

Show complex List

```
static void Show(string txt, List<complex> a)
```

Parameters

txt Variable description

a list of complex variables

Show

Show double[], double array

```
static void Show(string txt, double[] a)
```

Parameters

txt Variable description
a double array

Show

Show vector on console

```
static void Show(string txt, Vector a)
```

Parameters

txt Text string
a Vector

Show

Show matrix

```
static void Show(string text, Matrix a)
```

Parameters

text text string
a Matrix a

Show

Show ARX Model

```
static void Show(string txt, ARXModel a)
```

Parameters

txt text string
a ARXModel object

Show

Show Structure

```
static void Show(string txt, Structure a)
```

Parameters

txt text string
a Structure object

Show

Show

```
static void Show(string text, BMatrix a)
```

Parameters

```
text  Text string
a     BMatrix object
```

Show

Show

```
static void Show(string text, BVector a)
```

Parameters

```
text  text string
a     BVector object
```

Show

Show Polynomial

```
static void Show(string txt, Polynomial a)
```

Parameters

```
txt  text string
a    Polynomial
```

Show

Show Transfer function

```
static void Show(string txt, TransferFunction a)
```

Parameters

```
txt  text string
a    TransferFunction
```

Show

Show Complex Vector

```
static void Show(string txt, CVector a)
```

Parameters

```
txt  text string
a    complex vector
```

Show

Show LinearFilter

```
static void Show(string txt, LinearFilter A)
```

Parameters

txt text string
A A Filter object

Show

Show State Space model

```
static void Show(string txt, StateSpaceModel model)
```

Parameters

txt
model

ShowStructure

Show matrix structure

```
static void ShowStructure(string text, BMatrix A)
```

Parameters

text text string
A BMatrix object

WriteLine

Write empty line to console

```
static void WriteLine()
```

WriteLine

Write line to console

```
static void WriteLine(string txt)
```

Parameters

txt text string

Appendix C

Base classes for Vectors and Matrices

C.1 VBase

Class VBase

Base class for Vector holding values of type T

```
[Serializable]  
abstract class VBase<T>
```

Constructors

```
VBase(int n)
```

Constructor
Parameters
n

```
VBase(int n, T val)
```

Constructor vector with equal values
Parameters
n Dimension
value value

```
VBase(T[] vals)
```

Generate vector from array
Parameters
vals array of values

Properties**Dimension**

Get vector Dimension

```
int Dimension {get;}
```

Structure

Get vector structure

```
Structure Structure {get;}
```

T

Set or get vector value

```
virtual T this[int pos] {set; get;}
```

Methods**G**

Get value (equivalent to $\text{res} = \text{a}[\text{pos}]$)

```
T G(int pos)
```

Parameters

pos position

returns value

S

Set value (equivalent to $\text{a}[\text{pos}] = \text{val}$)

```
void S(int pos, T val)
```

Parameters

pos

val

C.2 MBase*Class* **MBase**

Base class for Matrix objects type T

```
[Serializable]
```

```
abstract class MBase<T>
```

Constructors

`MBase(int n)`

Constructor for square matrix

Parameters

n Dimension

`MBase(MatrixForm Frm, int n)`

Constructor for square matrix

Parameters

n

`MBase(int n, int m)`

Constructor for matrix

Parameters

n Rows

m Columns

`MBase(MatrixForm Frm, int n, int m)`

Constructor for matrix

Parameters

n Rows

m Columns

`MBase(int n, int m, T[] vals)`

Constructor for general matrix

Parameters

n Rows

m Columns

vals Values stored row wise

`MBase(int n, T val)`

Constructor for diagonal matrix with constant diagonal values

Parameters

n Dimension

val value

`MBase(T[] vals)`

Constructor for diagonal matrix from vector $a[i,i] = x[i]$

Parameters

vals vector of values

Properties

Columns

Columns

`int Columns {get;}`

ColumnStructure

Get Column structure

```
Structure ColumnStructure {get;}
```

MatrixForm

Matrix form (Null, ConstantDiagonal, Diagonal or General) Moving from Null towards General preserves data Moving from General to Null, ConstantDiagonal and Diagonal are ignored. Moving from General to Sparse is executed

```
virtual MatrixForm Form {set; get;}
```

NZ

Get or set space for Non zero values in sparse matrix. If NZ is set to less than the actual number of of non-zero values, the value buffer is shortened to actual number of non- zero values.

```
int NZ {set; get;}
```

Rows

Rows

```
int Rows {get;}
```

RowStructure

Get Row structure

```
Structure RowStructure {get;}
```

T

Set or get matrix value , res = A[row,col]

```
virtual T this[int row, int col] {set; get;}
```

Type

Object type

```
ObjectType Type {get;}
```

Methods

G

Get value (equivalent to `res = a[row,col]`)

```
T G(int row, int col)
```

Parameters

row Row
col

S

Set value (equivalent to `a[row,col] = res`)

```
void S(int row, int col, T val)
```

Parameters

row
col
val

Appendix D

Implementation of test fun using *IconFun*

Implementation of non -linear test function :

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using MPCMath;

namespace MPCMathApplication
{
    /// <summary>
    /// Non linear function with four minimas and two inequality constraints \index{IconFun}
    /// </summary>
    public class TestConFun : IconFun
    {
        // fun(x0,x1) = (x0^2 + x1 .11)^2 + (x0 + x1^2 -7)^2
        int IFun.N
        {
            get
            {
                return 2;
            }
        }

        /// <summary>
        /// Value
        /// </summary>
        /// <param name="x"></param>
        /// <returns></returns>
    }
}
```

```

double IFun.Value(Vector x)
{
    double ra = x[0] * x[0] + x[1] - 11.0;
    double rb = x[0] + x[1] * x[1] - 7.0;
    double res = ra*ra + rb*rb;
    return res;
}

/// <summary>
/// Jacobian
/// </summary>
/// <param name="x"></param>
/// <returns></returns>
Vector IFun.Dx(Vector x)
{
    double ra = x[0] * x[0] + x[1] - 11.0;
    double rb = x[0] + x[1] * x[1] - 7.0;
    Vector dx = new Vector(2);
    dx[0] = 4.0 * x[0] * ra + 2.0 * rb;
    dx[1] = 2.0 * ra + 4.0 * x[1] * rb;
    return dx;
}

/// <summary>
/// Hessian
/// </summary>
/// <param name="x"></param>
/// <returns></returns>
Matrix IFun.DDx(Vector x)
{
    Matrix DDX = null;
    double ra = x[0] * x[0] + x[1] - 11.0;
    double rb = x[0] + x[1] * x[1] - 7.0;
    DDX = new Matrix(MatrixForm.General, 2);
    DDX[0, 0] = 4.0 * ra + 8.0 * x[1] * x[1] + 2.0;
    DDX[0, 1] = 4.0 * (x[0] + x[1]);
    DDX[1, 0] = DDX[0, 1];
    DDX[1, 1] = 2.0 + 4.0 * rb + 8.0 * x[1] * x[1];
    return DDX;
}

/// <summary>
/// Number of Equality Constraints
/// </summary>
int IConFun.Ne
{
    get
    {
        return 0;
    }
}

```

```

    }
}

/// <summary>
/// Equality coinstraint values
/// </summary>
/// <param name="x"></param>
/// <returns></returns>
Vector IConFun.Vale(Vector x)
{
    Vector Cval = new Vector(0);
    return Cval;
}

/// <summary>
/// Equality constraint gradients
/// </summary>
/// <param name="x"></param>
/// <returns></returns>
Matrix IConFun.Dxe(Vector x)
{
    Matrix Dx = new Matrix(0, 2);
    return Dx;
}

/// <summary>
/// Equality constraint Hessians
/// </summary>
/// <param name="x"></param>
/// <returns></returns>
BMatrix IConFun.DDxe(Vector x)
{
    BMatrix DDx = null;
    DDx = new BMatrix(new Structure(1, 2));
    return DDx;
}

/// <summary>
/// Number of Inequality Constraints
/// </summary>
int IConFun.Ni
{
    get
    {
        return 2;
    }
}

/// <summary>
/// Constraint values

```

```

    /// </summary>
    /// <param name="x"></param>
    /// <returns></returns>
    Vector IConFun.Vali(Vector x)
    {
        Vector Cval = new Vector(2);
        Cval[0] = Math.Pow(x[0] + 2.0, 2.0) - x[1];
        Cval[1] = -4.0 * x[0] + 10.0 * x[1];
        return Cval;
    }

    /// <summary>
    /// Cosntraint gradients
    /// </summary>
    /// <param name="x"></param>
    /// <returns></returns>
    Matrix IConFun.Dxi(Vector x)
    {
        Matrix Dx = new Matrix(MatrixForm.General, 2, 2);
        Dx[0, 0] = 2.0 * (x[0] + 2.0);
        Dx[0, 1] = -1.0;
        Dx[1, 0] = -4.0;
        Dx[1, 1] = 10.0;
        return Dx;
    }

    /// <summary>
    /// Constraint Hessians
    /// </summary>
    /// <param name="x"></param>
    /// <returns></returns>
    BMatrix IConFun.DDxi(Vector x)
    {
        BMatrix DDx = null;
        DDx = new BMatrix(MatrixForm.Diagonal, new Structure(1, 2));
        Matrix ddx00 = new Matrix(MatrixForm.Diagonal, 2);
        ddx00[0, 0] = 2.0;
        DDx[0, 0] = ddx00;
        return DDx;
    }
}
}

```

Appendix E

Implementaion of Van der Pol equations using *IModel*

Implementation of the Van der Pol equations using IModel is shown below:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using MPCMath;

namespace MPCMath
{
    /// <summary>
    /// Van der Pol's problem
    /// $ y''(t) = \mu (1- y(t)^2)y'(t) -y(t) $
    /// </summary>
    public class VanDerPol : IModel
    {
        private double mu;
        private double t;

        /// <summary>
        /// Construtor
        /// </summary>
        /// <param name="Mu">$\mu$ parameter</param>
        /// <param name="T">Step size</param>
        public VanDerPol(double Mu, double T)
        {
            this.mu = Mu;
            this.t = T;
        }
    }
}
```

```

int IModel.Dimension
{
    get
    {
        return 2;
    }
}

int IModel.NU
{
    get
    {
        return 0;
    }
}

int IModel.NY
{
    get
    {
        return 1;
    }
}

Vector IModel.Derivative(Vector X, Vector U)
{
    Vector res = new Vector(2);
    res[0] = X[1];
    res[1] = this.mu*(1.0 - X[0]*X[0])* X[1] - X[0];
    return res;
}

Matrix IModel.Jacobian(Vector X, Vector U)
{
    Matrix res = new Matrix(MatrixForm.General, 2);
    res[0,1] = 1.0;
    res[1,0] = 2.0 * this.mu * X[0]*X[1] - 1.0;
    res[1,1] = this.mu*(1.0 - X[0]*X[0]);
    return res;
}

BMatrix IModel.BJacobian(Vector X, Vector U)
{
    return null;
}

Vector IModel.NextState(Vector X, Vector U, Vector W)
{
    throw new NotImplementedException();
}

```

```

Vector IModel.Observed(Vector X, Vector U)
{
    return +X;
}

StateSpaceModel IModel.LinearModel(Vector XS, Vector US)
{
    throw new NotImplementedException();
}

Vector IModel.Parameters
{
    get
    {
        return new Vector(1,this.mu);
    }
    set
    {
        this.mu = value[0];
    }
}
}
}

```


Appendix F

KKTSolver code

Source code listing for default KKTSolver ?? implementing interface IKKT-Solver 13.5.

```
// <code-header>
// <file>KKTSolver.cs</file>
// <author>Jrgen K H Knudsen</author>
// <copyright>Copyright (c) 2-control Aps. 2011</copyright>
// </code-header> using System;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace MPCMath
{
    // Quadratic Program KKT solver ( Q >= 0)
    // Linear Program solver ( Q = Null matrix or Q.Form == MatrixForm.Null);
    // min.    xGx + g'
    // s.t     Ax  = b
    //         Cx >= d
    //
    //      |G   -A'   -C'   |   | DX |   | -rL |
    //      |-A   0     0   | * | DY | = | -rA |
    //      |-C   0   -ZInvS |   | DZ |   | -rC |
    //
    //      Factorized into
    //
    //      |G - C'*SinvZ*C   -A'| * |DX| = | -rL + C'* SinvZ* rc|
    //      |- A              0 |   |DY| = | -rA              |
    //
    //      DZ = DX'* C' * SinvZ + SinvZ*rC

    /// <summary>
    /// Quadratic Program KKT solver, default KKT solver for QPSolver.
    /// Q must be positive semidefinit.
}
```

```

/// Solves Linear problems by setting Q = null.
/// </summary>
public class KKTsolver : IKKTSolver
{
    private static MPCMathConsole console = MPCMathConsole.Console;

    // Linear equation solver
    private SymmetricalBlockEquationSolver solver;

    private Structure strc;
    private Structure strcx;
    private Structure strcy;
    private Structure strcz;

    private BMatrix mG;
    private BVector vg;
    private BMatrix mA;
    private BVector vb;
    private BMatrix mC;
    private BVector vd;

    private BMatrix J;

    private BMatrix W;
    private BMatrix mCTSinvZ;
    /// <summary>
    /// Constructor for KKT solver
    /// </summary>
    /// <param name="G">G coefficients</param>
    /// <param name="g">g coefficients</param>
    /// <param name="A">A coefficients</param>
    /// <param name="b">b coefficients</param>
    /// <param name="C">C coefficients </param>
    /// <param name="d">d coefficients</param>
    public KKTsolver(BMatrix G, BVector g, BMatrix A, BVector b, BMatrix C, BVect
    {
        if (G != null)
        {
            if (!Structure.Equal(G.rowStructure, G.columnStructure))
            {
                throw new MPCMathException("MPCMath.KKTSolver", "Illegal G struct
            }

            this.mG = G;
        }
        else
        {
            this.mG = new BMatrix(g.structure);
        }
    }
}

```

```

if (!Structure.Equal(G.rowStructure, g.structure))
{
    throw new MPCMathException("MPCMath.KKTSolver", "Illegal g structure");
}

if (!Structure.Equal(G.rowStructure, A.columnStructure))
{
    throw new MPCMathException("MPCMath.KKTSolver", "Illegal A structure");
}

if (!Structure.Equal(A.rowStructure, b.structure))
{
    throw new MPCMathException("MPCMath.KKTSolver", "Illegal b structure");
}

if (!Structure.Equal(G.rowStructure, C.columnStructure))
{
    throw new MPCMathException("MPCMath.KKTSolver", "Illegal C structure");
}

if (!Structure.Equal(C.rowStructure, d.structure))
{
    throw new MPCMathException("MPCMath.KKTSolver", "Illegal d structure");
}

this.vg = g;
this.mA = A;
this.vb = b;
this.mC = C;
this.vd = d;

this.strcx = this.mG.rowStructure;
this.strcy = this.mA.rowStructure;
this.strcz = this.mC.rowStructure;
this.strc = new Structure(G.rows + A.rows);

for (int row = 0; row < this.strcx.Dimension; row++)
{
    this.strc[row] = this.strcx[row];
}

int p = this.strcx.Dimension;
for (int row = 0; row < A.rows; row++)
{
    this.strc[p] = A.rowStructure[row];
    p++;
}

this.J = new BMatrix(MatrixForm.General, strc);

```

```

    p = this.strcx.Dimension;
    for (int row = 0; row < this.strcy.Dimension; row++)
    {
        for (int col = 0; col < p; col++)
        {
            J[row + p, col] = -this.mA[row, col];
        }
    }
}

/// <summary>
/// X Structure
/// </summary>
Structure IKKTSolver.Strcx
{
    get
    {
        return this.strcx;
    }
}

/// <summary>
/// Y Structure
/// </summary>
Structure IKKTSolver.Strcy
{
    get
    {
        return this.strcy;
    }
}

/// <summary>
/// X Structure
/// </summary>
Structure IKKTSolver.Strcz
{
    get
    {
        return this.strcz;
    }
}

/// <summary>
/// Slacks
/// </summary>
BVector IKKTSolver.d
{
    get
    {

```

```

        return this.vd;
    }
}

/// <summary>
/// Calculate residulas
/// </summary>
/// <param name="x">state variables</param>
/// <param name="y">Lagrange multipliers for equality conditions</param>
/// <param name="z">Lagrange multipliers for inequality conditions</param>
/// <param name="s">Slack variables</param>
/// <param name="rL">residuals</param>
/// <param name="rA">residuals</param>
/// <param name="rC">residuals</param>
/// <param name="rSZ">residuals</param>
void IKKTSolver.Residuals(BVector x, BVector y,
    BVector z, BVector s,
    out BVector rL, out BVector rA,
    out BVector rC, out BVector rSZ)
{
    //  $rL = Gx + g - A'y - C'z$ 
    rL = this.mG * x + this.vg - y * this.mA - z * this.mC;

    //  $rA = b - Ax$ 
    rA = this.vb - this.mA * x;

    //  $rC = d - Cx + s$ 
    rC = this.vd - this.mC * x + s;

    //  $rSZ = SZe$ 
    rSZ = BVector.MulElements(s, z);
}

/// <summary>
/// Factorize KKT system
/// </summary>
/// <param name="SInvZ">diagonal vector</param>
void IKKTSolver.Factorize(BVector SInvZ)
{
    this.W = new BMatrix(SInvZ);

    this.mCTSInvZ = BMatrix.MulTransposed(this.mC, W);

    //  $J = \begin{bmatrix} G + C'SInvZ * C & -A' \\ -A & 0 \end{bmatrix}$ 
    BMatrix GM = this.mG + this.mCTSInvZ * this.mC;

    int dim = this.strcx.Dimension;
    for (int row = 0; row < dim; row++)

```

```

    {
        for (int col = 0; col < dim; col++)
        {
            this.J[row, col] = GM[row, col];
        }
    }

    this.solver = new SymmetricalBlockEquationSolver(J);
}

/// <summary>
/// Find Newton search directions
/// </summary>
/// <param name="rL">residual</param>
/// <param name="rA">residual</param>
/// <param name="rC">residual</param>
/// <param name="DX">X search direction</param>
/// <param name="DY">Y search direction</param>
/// <param name="DZ">Z search direction</param>
void IKKTSolver.Solve(BVector rL, BVector rA, BVector rC,
    out BVector DX, out BVector DY, out BVector DZ)
{
    BVector rX = -rL + this.mCTSinvZ * rC;
    BVector R = new BVector(this.strcx);
    int nx = this.strcx.Dimension;
    for (int p = 0; p < nx; p++)
    {
        R[p] = rX[p];
    }

    for (int p = 0; p < this.strcy.Dimension; p++)
    {
        R[p + nx] = -rA[p];
    }

    BVector res = this.solver.Solve(R);

    DX = new BVector(this.strcx);
    DY = new BVector(this.strcy);

    for (int p = 0; p < nx; p++)
    {
        DX[p] = res[p];
    }

    for (int p = 0; p < this.strcy.Dimension; p++)
    {
        DY[p] = res[p + nx];
    }
}

```

```
        DZ = -DX * this.mCTSinvZ + this.W * rC;  
        return;  
    }  
}  
}
```


Appendix G

PARDISO sparse matrix storage format

The compression of the non-zeros of a sparse matrix A into a linear array is done by walking down each column (column major format) or across each row (row major format) in order, and writing the non-zero elements to a linear array in the order that they appear in the walk.

When storing symmetric matrices, it is necessary to store only the upper triangular half of the matrix (upper triangular format) or the lower triangular half of the matrix (lower triangular format).

The Intel MKL direct sparse solver uses a row major upper triangular storage format. That is, the matrix is compressed row-by-row and for symmetric matrices only non-zeros in the upper triangular half of the matrix are stored.

The Intel MKL storage format accepted for the PARDISO software for sparse matrices consists of three arrays, which are called the values, columns, and rowIndex arrays. The following table describes the arrays in terms of the values, row, and column positions of the non-zero elements in a sparse matrix A .

<i>values</i>	A real or complex array that contains the non-zero entries of A . The non-zero values of A are mapped into the values array using the row major, upper triangular storage mapping described above.
<i>columns</i>	Element i of the integer array columns contains the number of the column in A that contained the value in values(i).
<i>rowIndex</i>	Element j of the integer array rowIndex gives the index into the values array that contains the first non-zero element in a row j of A .

The length of the values and columns arrays is equal to the number of non-zeros in A .

Since the rowIndex array gives the location of the first non-zero within a row, and the non-zeros are stored consecutively, then we would like to be able to compute the number of non-zeros in the i -th row as the difference of rowIndex(i) and rowIndex($i+1$).

In order to have this relationship hold for the last row of A , we need to add an entry (dummy entry) to the end of rowIndex whose value is equal to the number of non-zeros in A , plus one. This makes the total length of the rowIndex

array one larger than the number of rows of A.

The Intel MKL sparse storage scheme uses the Fortran programming language convention of starting array indices at 1, rather than the C programming language convention of starting at 0.

MPCMath takes care of all updating of the indexes of this storage scheme, and the indexes `RowIndex` and `Columns` are not presented for the *MPCMath* programmer. This description is provided solely to give an understanding of the sparse matrix storage mechanism.

Appendix H

MPCMath change history

Feb 25, 2012. Moved NiceMax 2.5 and NiceMin 2.5 from *Plot* 2.6 class to *PlotSeries* 2.6 class.

March 1, 2012 Support for serialization 17.11 of *MPCMath* objects. New *BinaryIO* 17.12 for saving and reading objects to disc files

March 6, 2012 IModel 11.4 property Hessian renamed to Jacobian

March 13, 2012 ODESolver included, chapter 15 . Van der Pol routines 14.5 and appendix E for demonstration of ODESolver included.

March 14, 2012 Support for serialization 17.11 of *MPCMath* objects to XML files . New *XmlIO* 17.13 for saving and reading objects to XML files

May 8, 2102 New Vector functionality ProperZeroes 3.2. IModel interface expanded with BJacobian ?? . SparseMatrixSolver 8.7 based in Intel's PARDISO routine. ODESolver 15.3 and SteadyState 16.4support for BJacobian ?? and support for Infeasible State Exception ?? . JacobianApprox ?? for debugging models implemented using IModel 11.4 interface.

May 12, 2012 IModel 11.4 interface, included parameter for sample time T and renamed NextStep to Next. Added administrative properties to StateSpaceModel 7.1

June 8, 2012 Sparse matrices implemented using PARDISO matrix storage. Matrix GetSubmatrix and SetSubmatrix changed to Get and Set.

June 23, 2012 Sparse matrices implemented as one of the possible Matrix Forms C.2. A quite big jobs, with a lot of testing. IModel 11.4 changed as usual. It will probably not stabilize until work with Non linear MPC is finished.

July 4, 2012 BMatrix 4.3 Get parameter sequence changed, to same form as Matrix Get routine

August 4, 2013. Minor changes to StateSpaceModel 7.1. Removed property YS, included properties XS and Off. New constructor included. Included RiccatiSolver 12.8 object. Included Cnsl class B.1 for debugging of Console based programs

December 15, 2014. Added BandEquationSolver 8.9, SymmetricBandEquationSolver 8.10, Spline 17.6, CubicSpline 17.7, IConFun 11.2, IFunt 11.3, FunMin , 12.4, ConFunMin 12.5, LeastSquareFit 12.7 and an additional version of Vector.Random 3.2

March 23, 2014 Modified *StateSpaceModel* 7.1 calculation of Kalman gain for MIMO state space models derived from generic arrays of Transferfunctions or ARXModels.

September 29, 2014 JacobianApprox replaced by routines JxEstimate 17.14 and JuEstimate 17.14

October 26, 2014 Imodel ?? modified once again, and as consequence hereof the LinearModel 14.2, VanderPol 14.5 and FourTankProcess 14.4 has been updated. 64 bit operating system support A.3.

November 25, 2014 Debugging and improvement of FunMin 12.4 and ConFunMin 12.5 functions. Added the following utility routines BFGD 16.5 , BFGSInv 16.5 and IsMimimum 16.5.

February 9, 2015 Debugging and improvement of FunMin 12.4 and ConFunMin 12.5 functions once again. Now they are working quite good, but there will always be room for improvements for these functions. Added a new interface *IFeasible* 11.5 used to define the feasible regions for models and functions

February 19, 2015 Complex logarithm included 17.1

Index

A, 128, 158, 226, 257, 258
Abort, 41
AcceptedSteps, 227
Add, 55, 67, 109, 119, 250
AddLine, 51
AddScaled, 67, 101, 109
Alfa, 158, 174, 215
Allocate, 93
Alpha0, 183
Arg, 77, 243
ARXModel, 15, 157
ASP, 158
AssertEqual, 41, 42, 56, 68, 77, 84, 104, 109, 120, 129, 150, 160, 250, 255, 278, 279
Axy, 56

B, 128, 159, 227, 259
BackTracks, 183
BandEquationSolver, 141, 142
BarrierFunction, 261
Beta, 174
BFGS, 236
BFGSInv, 237
BinaryIO, 265
BLAS, 28
BMatrix, 11, 91
BSP, 159
BVector, 11, 85

C, 128, 159, 227, 259
C1, 183
C2, 183
Cholesky factorization, 137
CholeskyEquationSolver, 137
Clear, 42, 51
Clone, 120, 250
Cnsl, 277
Coefficients, 249
Color, 47
Columns, 101, 287
ColumnStructure, 288
Complex, 243
complex, 243
Con, 244
ConFunMin, 178, 181, 186
Consistent, 180, 186, 189, 190
Console, 40
Covr, 56
Created, 128
CubicSpline, 260
CVector, 76

D, 259
d, 207
DDx, 163, 166
DDxe, 165
DDxi, 165
Delay, 119, 128, 159, 255
DelayChain, 255
Delays, 128
Density, 67
Derivative, 168, 250
Description, 128
Determinant, 134, 135
DFormat, 42, 43, 279
Diagonal, 65
Difference, 237
Digits, 40, 277
Dimension, 84, 128, 159, 167, 286
Disabled, 206
Discretize, 234, 235
Distribution, 237
Div, 120, 250
DivElements, 56, 86
Dx, 164, 166
Dxe, 165
Dxi, 165

EigenValues, 68
Entry, 43
EpsGrad, 184, 194

- EpsGradL, 187
- EpsGradX, 187
- Epsilon, 174
- EpsStep, 184, 194
- EpsStepX, 187
- Equal, 47, 56, 68, 77, 84, 101, 104, 110, 120, 130, 150, 251, 256
- EqualRoots, 78
- Equals, 244
- ErrorLimit, 206
- Evaluations, 227
- ExitReason, 184, 187
- Exp, 68, 244
- ExtendedDeltaARX, 159

- Fact, 238
- Factorizations, 227
- Factorize, 208
- FieldSize, 40, 277
- Fit, 195
- FlatDimension, 84
- Flatten, 87, 93
- Form, 101
- FourTankProcess, 220
- FunMin, 178, 179, 181, 183

- G, 286, 289
- Gain, 119, 149, 159, 249
- Gamma, 257
- Gauss, 238
- GaussDistribution, 238
- Gemm, 69
- Gemv, 57
- Ger, 69
- Get, 57, 69, 78, 87, 94, 104, 110, 256
- GetColumn, 69, 70
- GetHashCode, 244
- GetRow, 70

- H, 259
- History, 215
- Horizon, 215
- HuberFunction, 257

- ICommon, 100
- IconFun, 164, 179, 180
- IconFunMin, 178
- IFeasible, 170
- IFun, 163, 178
- IFunt, 166, 190

- IKKTSolver, 204, 207, 299
- IModel, 167, 235, 236, 295
- Indexer, 84
- InfeasibleSteps, 227
- InitialValue, 149
- Integrate, 228
- Invert, 70, 135, 138, 150
- Invertible, 150
- IsMiminimum, 238, 239
- ISolver, 133
- IsZero, 239
- Iterations, 184, 187, 194, 206, 227, 235
- IV, 78, 244

- Ju, 168
- JuEstimate, 266
- Jx, 168
- JxEstimate, 267

- K, 129, 149
- K0, 196
- KalmanGain, 240
- KKTSolver, 204, 209, 299
- KSP, 160

- L, 134
- Lambda, 174, 261
- LambdaE, 188
- LambdaI, 188
- LAPACK, 28
- LeastSquareEquationSolver, 136
- LeastSquareFit, 190, 194
- LimitViolation, 240
- LinearEquationSolver, 134
- LinearFilter, 148
- LinearModel, 169, 219
- Log, 245
- LPKKTSolver, 209
- LTR, 137
- LU factorization, 134, 135

- Matrix, 8, 65, 107
- MatrixForm, 10, 64, 67, 288
- MatrixOp, 64
- Max, 47, 57, 87
- MaxBacktrack, 174
- MaxBacktracks, 184
- MaxError, 236
- MaxIterations, 184, 188, 194, 206, 228, 235

- MaxIterationSteps, 174
- MaxTextLength, 40
- MBase, 286
- Mean, 58
- MiMoMPC, 23, 214
- Min, 47, 58, 87
- Minimize, 175, 186, 190
- MinNorm, 231
- MKL, 28
- Mod, 78, 244
- Model, 215
- MPCMathConsole, 39
- MPCMathLib, 231, 233–236, 266
- MPCMathLib-Discretize, 234
- MPCMathLib-JacobianEstimation, 266
- MPCMathLib-MinNorm, 231
- MPCMathLib-RK4, 233
- MPCMathLib-SteadyState, 235
- MPCMathLib-Util, 236
- MPCMathWindow, 50
- MPCMathWindow.xaml, 50
- Mu, 174, 188, 206, 215
- Mu0, 188
- Mul, 70, 120, 121, 251
- MulAdd, 71, 102, 110
- MulElements, 58, 88
- MulTransposed, 71, 94, 110

- N, 163, 166, 249
- Ne, 164
- NewLine, 263
- NewtonErrors, 228
- NewtonMethod, 173
- Next, 169, 217, 229
- NextState, 130, 160
- NextY, 217
- Ni, 164
- NiceMax, 48
- NiceMin, 48
- Niid, 150
- Norm, 58, 71, 88, 231
- Norm1, 71
- Normalize, 58
- NormInf, 58, 71
- NormRel, 59
- NU, 129, 167
- NumbersPerLine, 40, 277
- NY, 129, 167
- NZ, 288

- ObjectType, 103
- Observed, 130, 160, 169, 170
- ODEMethods, 225
- ODESolver, 226
- Off, 129
- Ok, 184, 188
- operator, 53, 59–61, 71–73, 78, 84, 88–90, 94–96, 105, 106, 110–112, 121–123, 151–154, 245–248, 251–253
- OuterProduct, 96

- P, 119, 149
- P0, 196
- p0, 196
- Parameters, 167
- PARDISO, 54, 307
- PBRs, 154
- Pivot, 135
- Plot, 43, 44, 49
- PlotSeries, 46, 50
- PlotType, 50
- PLU, 134
- Polynomial, 248
- Position, 260
- Print, 44
- ProperZero, 241
- ProperZeroes, 62
- PseudoInvert, 73
- PseudoRandom, 241

- Q, 136
- QPSolver, 205
- QR, 136
- QR factorization, 136

- R, 119, 136, 149
- R1, 136
- Ran, 236
- Random, 62, 73, 90, 96, 253
- Read, 265, 266
- ReadLine, 279
- Regress, 154, 155
- RejectedSteps, 228
- Rem, 253
- ReportBuilder, 263
- Reset, 130, 161, 256
- ResetValue, 255
- ResGrad, 184, 195
- ResGradL, 188

- ResGradX, 188
- Residuals, 208
- ResStep, 185, 195
- ResStepX, 189
- Restructure, 90, 96, 102
- Rho, 215
- RiccatiSolver, 196
- RK4, 234
- Roots, 119, 149, 249
- Rows, 101, 288
- RowStructure, 288
- Runge Kutta, 235
- Runge Kutta 4, 234
- RV, 79, 244
- S, 206, 286, 289
- Save, 265, 266
- ScalePlot, 48
- Serialization, 264
- SeriesColor, 48
- Set, 62, 73, 107, 112, 256
- SetColumn, 74
- SetKalmanGain, 130
- SetRef, 218
- SetRow, 74
- Show, 44, 45, 63, 74, 79, 85, 90, 91, 97, 102, 107, 112, 123, 131, 155, 161, 254, 257, 279–283
- ShowStructure, 97, 283
- Side, 65
- Sigma, 259
- SMTType, 67
- SoftConStrained, 216
- Solve, 134–143, 196, 197, 207, 209
- Solver, 216
- Sort, 63, 79
- Sparse, 168
- SparseEquationSolver, 139
- SparseMatrixType, 65
- Spline, 258
- Splines, 261
- Sqrt, 248
- StateSpaceModel, 18, 126
- SteadyState, 236
- Step, 228
- Steps, 175
- Strcx, 208
- Strcy, 208
- Strcz, 208
- Structure, 83, 286
- Sub, 63, 74, 112, 123, 254
- Sum, 63
- SVD, 74
- SymmetricalBlockEquationSolver, 139
- SymmetricBandEquationSolver, 141, 143
- SymmetricEquationSolver, 138
- T, 104, 109, 160, 168, 286, 288
- Tau, 195
- TauForm, 123, 124
- Text, 47, 51
- TextLength, 40
- Theta, 216
- Time, 206
- TMatrix, 107
- TMulAdd, 102, 113
- Tolerance, 228
- ToSparse, 75, 97
- ToString, 263
- Trace, 40, 185, 189, 195
- TraceIterations, 206
- TraceLevel, 185, 189, 207
- TraceText, 278
- TransferFunction, 13, 117
- Transpose, 75, 98, 102, 113
- Trmm, 75
- Truncate, 254
- TVector, 103
- Type, 50, 288
- U, 135, 216
- UMax, 216
- UMin, 216
- UnityMatrix, 76
- UPLO, 64
- US, 129
- UseBFGS, 185
- Vale, 165
- Vali, 166
- Value, 124, 164, 167, 254, 260, 261
- Values, 47, 85
- VanDerPol, 221, 223, 291, 295
- Variance, 64
- VBase, 285
- Vector, 8, 54, 103
- W, 185
- WarmStart, 185
- WhiteGaussianNoise, 241

Work, 41
Write, 264
WriteLine, 46, 283

X, 175, 185, 189, 207, 216, 261
X0, 259
x64, 272, 276
x86, 272, 276
XFMax, 170
XFMin, 170
XIterations, 189
XMax, 50
XMin, 50
XMinimizer, 189
XmlIO, 266
XS, 129
XUnchanged, 185
xv1, 265

Y, 207, 257, 259, 261
YFree, 216
YMax, 217
YMin, 217
YModel, 217
YPlant, 217
YRef, 217

Z, 207
Zeroes, 119, 149