A C++ Class for Polynomial Operations

Eker, Johan; Åström, Karl Johan

1995

Document Version:
Publisher's PDF, also known as Version of record

ISSN 0280–5316
ISRN LUTFD2/TFRT--7541--SE

A C++ Class for Polynomial Operations

Johan Eker
Karl Johan Åström

Department of Automatic Control
Lund Institute of Technology
December 1993

Department of Automatic Control
Lund Institute of Technology
P.O. Box 118
S-221 00 Lund Sweden

Document name
INTERNAL REPORT

Date of issue
December 1995

Document Number
ISRN LUTFD2/TFRT--7541--SE

Author(s)
Johan Eker and Karl Johan Åström

Supervisor

Sponsoring organisation

Title and subtitle
A C++ Class for Polynomial Operations

Abstract

Programmer's manual for a C++ polynomial class. The class is specially implemented for use in control applications.

Key words
C++ Class, Polynomial, Adaptive Control.

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280–5316

ISBN

Language
English

Number of pages
15

Recipient's notes

Security classification

1. Introduction

This report is a part of a research project on autonomous control supported by NUTEK under contract 9204014P. The project aims at developing controllers that are highly independent. Such controllers contain modules for parameter estimators, control design, and diagnostics. Many of these functions are based on polynomial operations. This C++ implementation provides support for such operations. It also has features designed for use in real-time control. One advantage of using an object oriented language for implementing a polynomial package is that arithmetic operators can be redefined, which means that the polynomials can be treated like any predefined arithmetic data type. This simplifies programming and it makes the programs easy to read. The main design goals have been to create a safe, reasonably fast and easy-to-use package. The class includes overloaded operators for the normal C/C++ operations. It also has special functions required for control design. All functions are made reentrant, since the class is intended to be used in a real-time environment.

Parts of the code are adapted from old Pascal programs developed at the department, an interactive polynomial calculation program PCalc, written by Tomas Schönthal and a program Toolbox, for real time control, written by Micael Lundh.

This report is organized as follows. An overview of the implementation is presented in Section 2. All functions and their attributes are presented in Section 3. Some examples of use of the package is given in Section 4. It is assumed that the reader is familiar with control theory and the programming languages C and C++.

2. Structure and Implementation

The internal representation of a polynomial is a vector where the coefficient of the highest power is stored at the first position, i.e. position zero. A polynomial

$$A = a_0 z^n + a_1 z^{n-1} + \cdots + a_n$$

will thus be represented in the following way

$$A = [a_0, a_1, \cdots, a_n]$$

The data type of the polynomial coefficients is called `real` and is by default defined as `double`. If there should be any need to change this it is easily done by changing the real typedef in `polynomial.h`. The size of a polynomial can be arbitrarily large since the internal data structure is dynamically allocated.

To avoid small coefficients as a result of computational errors a constant `epsilon` is used to define zero. `Epsilon` is by default set to 10^{-6}. This figure has been found to work well in most cases, but the results of the polynomial calculations are very sensitive to changes of `epsilon`, so it could be wise, if results are getting strange, to try another value for `epsilon`.

A simple debug function for presenting the polynomials on screen is provided.

The class is developed at a SUN SPARCstation running Solaris 2.3. The compiler used is AT&T CFront 3.0.1 together with SUN C-compiler.

3. Operators

One of the greatest advantages of an object-oriented language like C++ is the possibility to overload operators.the following set of arithmetic operators are implemented in the package:

Operator	Function	Use
+	add.	<polynomial> + <polynomial>
−	sub.	<polynomial> − <polynomial>
*	mul.	<polynomial> * <polynomial> <polynomial> * <real> <real>*<polynomial>
/	div.	<polynomial> / <polynomial> <polynomial> / <real>
=	assign	<polynomial>=<polynomial>
%	mod. (remainder)	<polynomial> % <polynomial>
+ =	add. and assign	<polynomial> + =<polynomial>
− =	sub. and assign	<polynomial> − =<polynomial>
* =	mul. and assign	<polynomial> * =<polynomial> <polynomial> * =<real>
/ =	div. and assign.	<polynomial> / =<polynomial> <polynomial> / =<real>
% =	mod. and assign	<polynomial> % =<polynomial>

The results of the operations are always of type `polynomial`. Two comparison operators `==`(equal) and `!=`(not equal) are also included. It is difficult to find a good way to determine if two polynomials are equal. In this implementation a very straightforward approach is used. The second operand is simply subtracted from the first and the result is examined. If all the coefficients of the resulting polynomial are smaller than the constant epsilon then the operands are regarded as equal. The `!=` operator is implemented using the `==` operator, so at least the operators are mutually exclusive. An example of the convenience of using overloaded operators is illustrated below.

```
Polynomial A, B, C;
...
A = (B * C)/(2 * A) + A \%C;
B = A + 3.1416*C;
C *= 10;
...
```

4. Class Messages and Attributes

In this section a detailed description of polynomial class is given.

Constructors

```
Polynomial();
```

Constructs a polynomial of degree zero with the only coefficient equal to zero.

`Polynomial(int Degree,real *Coefficients);`

Constructs a polynomial of degree `Degree` with the coefficients from `Coefficients`. The first element in both the input vector `Coefficients` and the resulting polynomial represents the the highest power coefficient.

`Polynomial(int Degree,real *Coefficients, char *str);`

This constructor works in the same way as the one above, but has in addition the ability to give a name to the polynomial. This feature is mainly used for debug purposes. The program example below uses this constructor to create the polynomial $A = q^2 + 1.5q - 2$.

```
#include "polynomial.h"
main()
{
  int   deg;
  real  coeff[3];
  Polynomial *opA;

  deg = 2;
  coeff[0] = 1;
  coeff[1] = 1.5;
  coeff[2] = -2;

  opA = new Polynomial(deg, coeff, "A");
  opA->Display()
}
```

This program generates the following output.

```
A = 1.0 q^2 + 1.5 q - 2.0
```

Operators

Below is an overview of all the overloaded operators.

Member Operators:

```
Polynomial& operator =  (const Polynomial p);
Polynomial& operator += (const Polynomial p);
Polynomial& operator -= (const Polynomial p);
Polynomial& operator *= (const Polynomial p);
Polynomial& operator *= (real r);
Polynomial& operator /= (const Polynomial p);
Polynomial& operator /= (real r);
Polynomial& operator %= (const Polynomial p);
```

Friend Operators:

```
Polynomial operator * (const Polynomial& p, real r);
Polynomial operator * (real r, const Polynomial& p);
Polynomial operator * (const Polynomial& p1,
                       const Polynomial& p2);
Polynomial operator / (const Polynomial& p,real den);
```

```
Polynomial operator / (const Polynomial& num,
                       const Polynomial& den);
Polynomial operator % (const Polynomial& num,
                       const Polynomial& den);
Polynomial operator + (const Polynomial& p1,
                       const Polynomial& p2);
Polynomial operator - (const Polynomial& p1,
                       const Polynomial& p2);
void PolDiv(const Polynomial num, const Polynomial den,
            Polynomial& quotient, Polynomial& remainder);
int operator == (const Polynomial& p1, const Polynomial& p2);
int operator != (const Polynomial& p1, const Polynomial& p2);
```

Attributes

`real *coeff;`

This is the vector that contains the coefficients of the polynomial. The coefficient of the highest power is stored at the first position i.e. position zero.

`int deg;`

The degree of the polynomial.

`char name[namelength+1];`

This character vector holds the name of the polynomial instance and is used mainly for debug purposes. The `Display` function, see below uses `name` by default when writing the polynomial to the console. The constant `namelenght` is set in `Polynomial.h`

Operations on Attributes

These are member functions, which provide a way for the user to manipulate the class attributes without having direct access to them.

`void Set(int Degree,real *Coefficients);`

Sets the degree of the polynomial to `Degree` and the coefficients to `Coefficients`. `Coefficients` is a vector with the highest power coefficient at position zero.

```
real& operator[](int elem) const;
real& operator[](int elem);
```

This operator returns a reference to the element at position `elem`. It is valid both for assigning and returning values.

`int  GetDegree(void) const;`

Returns the degree of the polynomial.

`void SetDegree(int d);`

Sets the degree of the polynomial to `d`. If the degree is decreased the polynomial will simple be truncated and in the case where the new degree is higher then the previous then the new elements are set to zero.

```
void Display(char *str = "\0");
```

Displays the polynomial on the console. If the function is called with an argument it will write the string `str` followed by the polynomial to the console. If called with no argument it will use the class attribute `name` instead of `str`. Let the variable `opA` be defined as in the program example above, and make the following function call:

```
opA->Display("This is polynomial A:");
```

The following output is generated:

```
This is polynomial A:1.0 q^2 + 1.5 q - 2.0
```

```
void Display(char variable, signType sign = positive,
             char *str = "\0");
```

By default the polynomial is presented in the variable `q`. This `Display()` call takes two new arguments. The first argument `variable` tells the function which variable to use, and the second argument `sign` contains the sign of this variable. The third argument is optional and is used as in the other `Display()` call described above.

```
void Clear(int d);
```

Sets all the coefficients to zero and the degree to `d`.

```
void SetName(char *str)
```

Sets the class attribute `name` to str.

```
real Eval(real r);
real operator()(real r);
```

These are two identical functions which calculate the value of the polynomial for `q = r`, i.e. the scalar `A(r)`.

```
Polynomial Reciprocal()
```

Returns the reciprocal polynomial. The reciprocal polynomial is defined as follows:
Let $A = 3.5q^2 + 1.1q + 999$, [1] then the reciprocal polynomial denoted A^* is $A^* = 3.5 + 1.1q + 999^2 = q^2A(q^{-1})$.
More formal:

$$A^*(q) = 1 + a_1 * q + \cdots + a_{nq}^{na} = q^{na}A(q^{-1}) \tag{1}$$

See [Åström and Wittenmark, 1990] for a full definition.

```
int IsStable()
```

This function uses Jury's Criterion to determine if the polynomial is stable. The algorithm is described in [Åström and Wittenmark, 1990]. If the polynomial is stable 1 is returned otherwise 0 is returned.

---

1 q is the forward shift operator. The forward shift operator has the property $qf(k) = f(k+1)$.The backward shift operator denoted q^{-1} is the defined as $q^{-1}f(k) = f(k-1)$.

Miscellaneous Operations

`void ShiftForward(real r);`

This function shifts the coefficients of the polynomial to the left and sets the last coefficient to `r`. Let $A = 2.3q^2 + 3.5q + 1.1$, then the following operation

```
A.ShiftForward(999);
```

changes A to $A = 3.5q^2 + 1.1q + 999$.

`void ShiftBackward(real r);`

This function shifts the coefficients of the polynomial to the right and sets the highest power coefficient to `r`.

`Polynomial& MakeMonic();`

`MakeMonic` makes the polynomial monic by dividing all coefficients with the highest power coefficient. In a monic polynomial the highest coefficient is always 1. For example let $A = 2q^2 + q + 10$, then the following operation

```
B = A.MakeMonic();
```

sets $B = q^2 + 0.5q + 5$.

`void Cutpoly(real);`

Normalizes the polynomial, i.e. all coefficients that are smaller than the largest coefficient times `epsilon` are removed. If necessary the degree of the polynomial is changed

5. External Functions

In this section the numerical polynomial functions included in the package are described. The reader is assumed to have knowledge of polynomial design methods and to be familiar to the notations used in [Åström and Wittenmark, 1990] and [Åström and Wittenmark, 1995]. All algorithms in the functions below originate from those books unless otherwise stated.

```
void GCD(Polynomial A, Polynomial B, real reps,
         Polynomial& G, Polynomial& X, Polynomial& Y,
         Polynomial& U, Polynomial& V);
```

This function calculates the greatest common divisor of polynomials `A` and `B` using Euclid's algorithm. It also solves the two equations

$$AX + BY = G \tag{2}$$

$$AU + BV = 0 \tag{3}$$

G is the greatest common divisor of A and B. A more detailed description of the algorithm is found in [Åström and Wittenmark, 1995].

```
void DiophantineMDS(Polynomial A, Polynomial B, Polynomial Ac,
                    Polynomial& R, Polynomial& S);
```

`DiophantineMDS` finds the minimal degree solution to the equation $AR + BS = A_c$. The equation is solved in two steps. First the two equations 2 and 3 are solved using the GCD function. Then in the second step the general solution is found through

$$R = R^0 + QU$$

$$S = S^0 + QV$$

with

$$R^0 = X A_c \operatorname{div} G$$

$$S^0 = Y A_c \operatorname{div} G$$

where G is the greatest common divisor of A and B. The minimal degree solution is now given by simply choosing
$Q = S^0 \operatorname{div} V$. The greatest common G must divide A_c otherwise the equation has no solution.

EXAMPLE 1

```
#include"polynomial.h"

main()
{
  Polynomial A, B, C, XX, YY;

  real A_data[]= {1, 2.3, 3.5};
  A.Set(2, A_data, "A");

  real B_data[] = {1, 1.45};
  B.Set(1, B_data, "B");

  real C_data[] = {1, 3.4, 0.8, 2};
  C.Set(3, C_data, "C");

  DiophantineMDS(A, B, C, XX, YY);

  XX.Display("XX = ");
  YY.Display("YY = ");
}
```

When running the program above the following output is produced:

```
XX = 1.000 q + 3.629
YY = -2.529 q -7.379
```

```
void Sfactorize(Polynomial B, Polynomial& A);
```

Let

$$A(z) = z^{na} + a_1 z^{na-1} + \cdots + a_{na}. \tag{4}$$

The *reciprocal polynomial* of A, denoted A^* is obtained by reversing the order of the coefficients of A.

$$A^*(z) = 1 + a_1 z + \cdots + a_{na} z^{na} = z^{na} A(z^{-1}). \tag{5}$$

`Sfactorize` takes a polynomial $B = ee^*$ and returns a stable polynomial A so that $AA^* = ee^*$. This algorithm is described in [Kučera, 1979].

```
void DyadicReduction(Polynomial& A, Polynomial& B, real& Alpha,
                     real& Beta, int i0, int i1, int i2);
```

Given vectors

$$\begin{aligned} a &= [1\ a_2 \cdots a_n]^T \\ b &= [1\ b_2 \cdots b_n]^T \end{aligned} \tag{6}$$

and scalars α and β, find vectors

$$\begin{aligned} \tilde{a} &= [1\ \tilde{a_2} \cdots \tilde{a_n}]^T \\ \tilde{b} &= [1\ \tilde{b_2} \cdots \tilde{b_n}]^T \end{aligned} \tag{7}$$

such that

$$\alpha a a^T + \beta b b^T = \tilde{\alpha}\tilde{a}\tilde{a}^T + \tilde{\beta}\tilde{b}\tilde{b}^T \tag{8}$$

The vectors $\tilde{a}$ and $\tilde{b}$ can be found using *dyadic decomposition*. `DyadicReduction` is very useful when doing square root recursive least square estimations. For a closer look at the algorithm and its applications see [Åström and Wittenmark, 1995].

```
void RobustIntegralDesign(Polynomial  A, Polynomial  B,
                          Polynomial  Ac, Polynomial& R,
                          Polynomial& S, real x0, real x1);
```

This function calculates an integral controller with zero gain at the Nyquist frequency. Two additional closed loop poles are specified through `x0` and `x1` which are coefficients in the X-polynomial, see below.

First the minimal degree solutions R^0 and S^0 are calculated. If R^0 and S^0 satisfy

$$AR^0 + BS^0 = A_c$$

then

$$\begin{aligned} R &= XR^0 + YB \\ S &= XS^0 - YA \end{aligned}$$

are solutions to the equation

$$AR + BS = XA_c$$

This gives a controller with the characteristic polynomial A_cX, where $X = q^2 + x_1q + x_0$. To get the desired controller first let $Y = y_oq - y_1$. Then solve

$$R(1) = 0 \Leftrightarrow 0 = -X(1)R^0(1) + Y(1)B(1)$$

$$S(-1) = 0 \Leftrightarrow 0 = X(-1)S^0(-1) - Y(-1)A(-1)$$

By using those equations the coefficients of the Y polynomial can be calculated.

$$\begin{aligned} y_0 &= \frac{Y(1) - Y(-1)}{2} \\ y_1 &= \frac{Y(1) + Y(-1)}{2} \end{aligned}$$

EXAMPLE 2

```
#include <stream.h>
#include "polynomial.h"

main()
{
  Polynomial A, B, Ac, R, S, X, tmp;

  real A_data[]= {1, 2.3, 3.5};
  A.Set(2, A_data, "A");

  real B_data[] = {1, 1.45};
  B.Set(1, B_data, "B");

  real Ac_data[] = {1, 3.4, 0.8, 2, 2};
  Ac.Set(3, Ac_data, "Ac");

  RobustIntegralDesign(A, B, Ac, R, S, -1, 0.25);

  real X_data[] = {1, -1, 0.25};
  X.Set(2, X_data);
  R.Display("R = ");
  S.Display("S = ");

  cout << "Integral R(1)  = " <<  R(1) << endl;
  cout << "Robust   S(-1) = " <<  S(-1) << endl;
  Ac.Display();

  ((A*R + B*S)/X).Display("(A*R + B*S)/X = ");
}
```

When running the program above the following output is produced:

```
R = 1.000 q^3 + 4.873 q^2 -2.841 q -3.032
S = -4.773 q^3 -7.296 q^2 + 5.140 q + 7.663
Integral R(1)  = -8.88178e-16
Robust   S(-1) = -1.77636e-15
Ac = 1.000 q^3 + 3.400 q^2 + 0.800 q + 2.000
A*R + B*S)/X = 1.000 q^3 + 3.400 q^2 + 0.800 q + 2.000
```

In the example above the constant **epsilon** was set to 10^{-6}. The two following functions use the same method to calculate a robust controller and an integral controller.

```
void IntegralDesign(Polynomial  A, Polynomial  B, Polynomial C,
                    Polynomial& R, Polynomial& S, real x0);
```

This function first solves the diophantine equation and then forces the R polynomial to contain an integrator. This is done by designing the R polynomial so that $R(1) = 0$. The input parameter x0 specifies the additional closed loop pole. The characteristic polynomial now becomes $A_c X$, where $X = q - x_0$.

```
void RobustDesign(Polynomial  A, Polynomial  B, Polynomial C,
                  Polynomial& R, Polynomial& S, real x0);
```

`RobustDesign` works similar to `IntegralDesign` but the constraint on the controller design is instead $S(-1) = 0$. This condition gives a controller with zero gain at the Nyquist frequency. The input parameter `x0` specifies the additional closed loop pole. The characteristic polynomial now becomes $A_c X$, where $X = q - x_0$.

```
void LQGDesign(Polynomial A, Polynomial B, Polynomial C,
               Polynomial& R, Polynomial& S, real rho);
```

Calculates a LQG-controller for the system A, B, and C with the loss function coefficient ρ. The computational procedure is described in detail in section 12.5 in [Åström and Wittenmark, 1990]. This implementation only handles the case where $A(0) \neq 0$. Below is an example of how a LQG-problem can be solved using this function. It is taken from [Åström and Wittenmark, 1990], Example 12.7, with $A(z) = z + 0.5$, $B(z) = 0.5$, and $C = z + 0.8$.

EXAMPLE 3

```
#include <stream.h>
#include "polynomial.h"

main()
{
  Polynomial A, B, C, R, S;

  cout << "Example 12.7 in CCS" << endl;
  cout << "LQG Design with rho = 0.5." << endl;

  A.SetDegree(1);
  A[0] = 1;
  A[1] = 0.5;  // a = 0.5
  B.SetDegree(0);
  B[0] = 0.5;  // b = 0.5
  C.SetDegree(1);
  C[0] = 1;
  C[1] =0.8;   // c = 0.8

  LQGDesign(A, B, C, R, S, 0.5);

  R.Display("R = ");
  S.Display("S = ");
}
```

The following output is generated:

```
R = 1.000 q + 0.614
S = -0.112
```

```
void MDPPNZCDesign(Polynomial A, Polynomial B, Polynomial Am,
                   Polynomial Ao, Polynomial& R, Polynomial& S,
                   Polynomial& T);
```

The abbreviation stands for Minimal Degree Pole Placement with No Zero Cancelation. The function chooses $B^+ = 1$ and $B^- = B$. Furthermore B_m is chosen so that the stationary gain will be unity.

$$B_m(q) = \frac{A_m(1)B(q)}{B(1)}$$

The closed-loop characteristic equation to be solved now becomes

$$AR + BS = A_o A_m$$

The T polynomial is given by

$$T(q) = \frac{A_m(1)A_o(q)}{B(1)}$$

How the `MDPPNZCDesign`-function is used is demonstrated in the example below.

EXAMPLE 4

```
#include <stream.h>
#include "polynomial.h"

main()
{
  Polynomial A, B, Am, Ao, R, S, T;

  B.SetDegree(1);
  B[0] = 1;
  B[1] = 0.7;
  A.SetDegree(2);
  A[0] = 1;
  A[1] = -1.8;
  A[2] = 0.81;
  Am.SetDegree(2);
  Am[0] = 1;
  Am[1] = -1.5;
  Am[2] = 0.7;
  Ao.SetDegree(1);
  Ao[0] = 1;
  Ao[1] = 0;

  MDPPNZCDesign(A, B, Am, Ao, R, S, T);

  R.Display("R = ");
  S.Display("S = ");
  T.Display("T = ");
}
```

From this program the following output is generated:

```
R = 1.000 q + 0.088
S = 0.213 q + -0.101
T = 0.118 q + 0.000
```

```
void Roots(Polynomial A, Polynomial& rootRe,
           Polynomial& rootIm);
```

The roots of polynomial `A` are calculated and are returned in the two polynomials `rootRe` and `rootIm`. The real parts of the roots are stored in `rootRe` and the imaginary parts are stored in `rootIm`.

```
void LDFilter(double *l, Polynomial& d, Polynomial& phi,
              Polynomial& theta, double& lambda);
```

This function is an implementation of an estimator and would together with any of the design functions above form a complete adaptive controller. The algorithms behind `LDFilter` is taken from [Åström and Wittenmark, 1995]. Let the system to be estimated be on the form

$$y(t) = \varphi^T(t)\theta$$

where θ is a parameter vector and φ is a vector of signals. The following recursive least-square estimation algorithm is used

$$\begin{aligned}\hat{\theta}(t) &= \hat{\theta}(t-1) + K(t)(y(t) - \varphi^T(t)\hat{\theta}(t-1)) \\ K(t) &= P(t)\varphi(t) \\ P(t) &= (I - K(t)\varphi^T(t))P(t-1)\end{aligned}$$

The covariance matrix P has a decomposition $P = LDL^T$, where L is a lower triangular matrix and D is a diagonal matrix. Initially set $L = I$, which gives that $P(0) = D$. The function takes the following arguments:

- `double *l`
  An array of doubles with the size $deg(\theta) \times deg(\theta)$.
- `double *d`
  An array of doubles with the size $deg(\theta)$.
- `Polynomial& Phi`
  This a polynomial that contains old process values and old control signals. The polynomial is arranged on the following format:

  $$Phi = [y(t), -y(t-1), \ldots, -y(t-n), u(t-d), \ldots, u(t-d-m)]$$

  where $n = deg(A), m = deg(B)$, and $d = n - m$.
- `Polynomial& theta`
  This is a polynomial with the degree set to $(deg(A) + deg(B) + 1)$ and with the coefficients to be estimated on the following format:

  $$theta = [a_0, a_1, \ldots, a_n, b_0, \ldots, b_m]$$

- `double& l`
  This parameter is the forgetting factor λ.

6. Testing

The class is tested both in several real-time control applications and in several test programs. The test batch mentioned above is called `polyTest` and

is included with the other files in the package. It simply uses a number of polynomials, defined in the beginning of the program, and runs them through a number of equations and expression. Finally it checks if the results are the expected. The calculations made by the program are designed so that all the used polynomials should end up with their initial value. The result from an execution that detected an error is shown below.

```
*** RESULT ***
Polynomial A:  ...OK!
Polynomial B:  ...OK!
Polynomial C:  ...OK!
Polynomial D:  ...OK!
Polynomial H:  ...OK!
Polynomial I: --- ERROR IN POLYNOMIAL PACKAGE! ---
```

The source code for the test program is available together with the other files of the package.

7. A Small Example Program

This is small example program demonstrating how the Polynomial class can be used.

EXAMPLE 5

```
#include <stream.h>
#include "polynomial.h"

void main()
{
  Polynomial A, B, C;
  real Indata[4];
  int   Degree;
  signType sign;

  A.SetDegree(3);
  A[0] = 1;
  A[1] = 2.3;
  A[2] = 3.5;
  A[3] = 4.1;
  A.SetName("A");

  Degree = 2;
  Indata[0] = 1;
  Indata[1] = 1.45;
  Indata[2] = 7.4;
  B.Set(Degree, Indata, "B");

  C = A * B;
  C.SetName("C");

  A.Display();
  B.Display();
  C.Display();
```

```
  A += A;
  A.Display();

  B.MakeMonic().Display();

  sign = negative;
  ((A*C + B*A) % B.Reciprocal()).
     Display('z', sign, "(A*C + B*A) %% B.Reciprocal() = ");

  A.Display();
  if ( A.IsStable() )
    cout << "A is stable"<< endl;
  else
    cout << "A isn't stable" << endl;

  if ( A.Reciprocal().IsStable() )
    cout << "A.Reciprocal() is stable"<< endl;
  else
    cout << "A.Reciprocal() isn't stable" << endl;
}
```

This program gives the following output:

```
A = 1.00 q^3 + 2.30 q^2 + 3.50 q + 4.10
B = 1.00 q^2 + 1.45 q + 7.40
C = 1.00 q^5 + 3.75 q^4 + 14.235 q^3 + 26.195 q^2 +
    31.845 q + 30.34
A = 2.00 q^3 + 4.60 q^2 + 7.00 q + 8.20
B = 1.00 q^2 + 1.45 q + 7.40
(A*C + B*A) % B.Reciprocal() = 376.913345 z + 241.807917
A isn't stable
A.Reciprocal() is stable
A = 2.000000 q^3 + 4.600000 q^2 + 7.000000 q + 8.200000
```

8. Files

The polynomial package consists of the following files:

- `polynomial.h polynomial.c`
  Those files contain the definitions and declarations of the polynomial class and the controller design functions.
- `polyTest.c`
  This is a test program for testing the polynomial class.
- `Makefile`
  The Makefile for making `polyTest`.

9. References

ÅSTRÖM, K. J. and B. WITTENMARK (1990): *Computer Controlled Systems—Theory and Design*. Prentice-Hall, Englewood Cliffs, New Jersey, second edition.

ÅSTRÖM, K. J. and B. WITTENMARK (1995): *Adaptive Control.* Addison-Wesley, Reading, Massachusetts, second edition.

KOENIG, A. R. (1989):. "Effective use of C++.". Course material used in a course given by the author 1989 at the Department of Automatic Control, Lund, Sweden.

KUČERA, V. (1979): *Discrete Linear Control—The Polynomial Equation Approach.* Wiley, New York.

LIPMAN, S. B. (1991): *C++ Primer.* Addison-Wesley Publishing Company, 2nd edition.

SHOPIRO, J. E. (1991):. "Advanced C++.". Course material used in a course given by the author 1991 at the Department of Automatic Control, Lund, Sweden.