MATX: A High-Performance Programming Language (Interpreter and Compiler) for Scientific and Engineering Computation

Masanobu Koga and Katsuhisa Furuta

Furuta Laboratory, Department of Control Engineering,
Tokyo Institute of Technology,
2-12-1, Oh-okayama, Meguro-ku, Tokyo 152, JAPAN

Abstract. **MaTX** is a high-performance programming language for scientific and engineering computation. It provides not only command-line *interpreter* **matx** whose interfaces are similar to the use of Matlab but also *compiler* **matc** which accepts the same syntax as that of matx and outputs portable C language code. The users can extend the functionality of a program by realizing algorithms as functions in "mm-files" which correspond to "m-files" of Matlab. The interpreter matx enables us to test out the mm-files interactively, file-by-file or line-by-line. Furthermore, it is possible to make the executable program from the same source code using matc. MaTX integrates numerical analysis, matrix computation, and symbolic computation in an easy-to-use environment where problems and solutions are expressed almost exactly as they are written mathematically. The translator which translates m-files to mm-files is under development.

1 Introduction

The interactive matrix manipulation package Matlab [2],[3] has become a standard among users and developers of numerical software. It has found widespread acceptance in application areas in which linear algebra plays an important role, such as signal processing, process identification and control system design. For these applications, special toolboxes are available, which are collections of small procedures (m-files) written in the Matlab language. Since programming m-files is very easy and also because the Matlab environment is very well suited for testing out new ideas interactively, many developers of numerical software today prefer programming in Matlab over traditional languages such as Fortran, Pascal, and C.

However there are at least two points in Matlab which should be improved. One is concerned with limited data structure. Matlab has only one data type, i.e., the Matrix, thus the other kinds of data are supposed to be coded using matrices. Such limited data structure doesn't allow a natural symbolic representation of fundamental mathematical objects such as polynomials and rational-functions. This makes problem descriptions hard to understand. The other is concerned with speed of computation. It takes much time to calculate complicated algorithms because m-files are interpreted rather than executed after compilation.

We have developed a high-performance programming language **MaTX** for scientific and engineering computation [1]. Many of the important ideas of matrix operation of MaTX stem from the matrix manipulation language Matlab developed by Cleve Moler [2],[3]. Although it shares several characteristic features with Matlab, MaTX is in no sense a dialect of it. Matlab is "typeless" language: the only data type is the Matrix, and operation of other kinds of objects is by special operators or function calls. MaTX is equipped to recognize several primitive data type such as Integer, Real-Number, Complex-Number, String, Polynomial, Rational-Function, Matrix, Array, and List. Each class object is stored in an appropriate data structure and treated suitably. It deals with symbolic formulae as well as numbers. It can solve polynomial equations and evaluate derivatives and integrals symbolically. MaTX has inherited the idea of C language such as declaration of variables, control-flow, style of function, and structure of a program. It is written all in C language and the parser has been developed by using the well known parser generator "**yacc**".

Most implementations of Matlab have a special interface which makes it possible to call separately compiled and linked Fortran (or C) routines ("mex-files") from the Matlab environment, just as if they were ordinary m-files. This feature is useful if the m-file implementation of an algorithm is not efficient enough. Creating mex-files, however, is no simple task. One needs to write a special "gateway function" for passing the function parameters from Matlab to Fortran and back. The Matlab gateway compiler has been developed [4], that automatically creates a gateway function for almost Fortran routines to be used in Matlab.

MaTX provides not only command-line *interpreter* **matx** whose interfaces are similar to the use of Matlab but also *compiler* **matc** which accepts the same syntax as that of matx and outputs portable C language code. The users can extend the functionality of a program by realizing algorithms as functions in "mm-files" which correspond to "m-files" of Matlab. It is very easy to call your own C functions from MaTX functions by linking C language routines to the output of matc. This allows us to utilize huge pre-exist C routines and speed up the rate of computation and promote the efficiency of memory usage.

The translator which translates m-files to mm-files is under development. When the translator comes out, the users of Matlab benefit from the MaTX interpreter and compiler facilities. But it is tough work to make the translator, because the syntax of Matlab is designed for interpreter rather than compiler. There is no class declaration statements in the Matlab language, since the interpreter knows everything about all variables at run time. Compilers, however, can't get class information of variables and functions at compile time, unless there are class declaration statements.

The disposition of the paper is as follows: The architecture of MaTX is outlined in Section 2, and the program developing process in MaTX is presented in Section 3. The matrix manipulation and the symbolic computation are described in Section 4 and Section 5, respectively. Section 6 deals with the List class and the multiple output function and demonstrates a practical example program. In Section 7, we touch on a few programming facilities in MaTX. Section 8 make a comparison of the performance between the interpreter (matx) and the compiled program which is produced by the compiler (matc). Section 9 concludes with some perspective remarks.

2 Architecture of MaTX

The architecture of MaTX is shown in Fig. 2.1. At the moment, MaTX provides three useful user-interface tools: interpreter "matx", compiler "matc", and matrix-editor "mated". They are built upon MaTX-Lib which is a collection of useful Class libraries such as **Complex-Number**, **Polynomial**, **Rational-Function**, **Matrix**, and **List**[1].

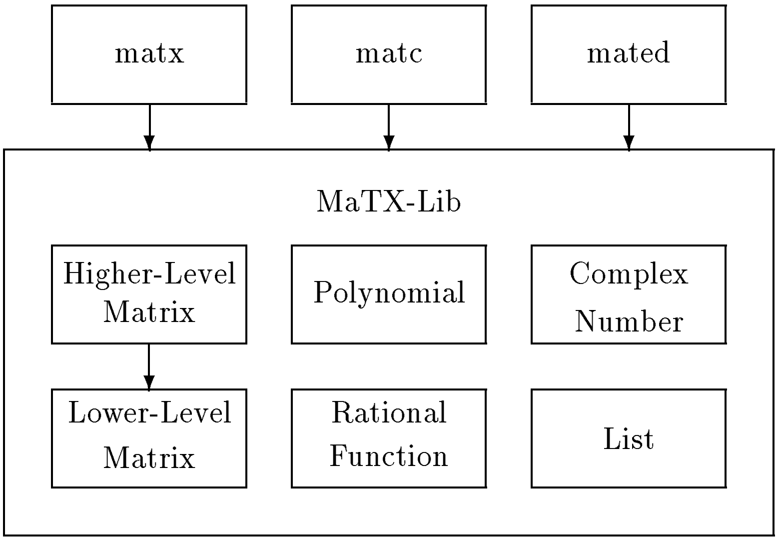

Fig. 2.1 The Architecture of MaTX

The library set MaTX-Lib can be utilized to make programs in C language by itself. Matrix class library is composed of higher-level library and lower-level library. The lower-level library consists of nearly 300 functions, which provide fast access to the objects, but do little to make programming easier. It handles the operation of the objects and includes some optimizations that encourage efficient memory usage. The higher-level library is built upon lower-level library and consists of about 200 functions. The purpose of the higher-level library is to provide the supervisory layer that supports the user-interface facilities. It handles the memory allocation of object and calls suitable lower-level functions according to the attribute of the object.

Figure 2.2 indicates the internal process of MaTX. A source file (`test.mm`) written in the MaTX language is given to the lexical analyzer which recognizes the expressions in the input stream and partitions it into strings matching the MaTX expressions. The Parser calls the lexical analyzer to pick up the partitioned basic items (tokens). These tokens are organized according to the MaTX input structure rules; when one of these rules is recognized, the corresponding action for the rule is invoked. In each action, both the interpreter and the compiler defect type mismatches, inconsistent argument usage, unused or apparently uninitialized variables, potential portability difficulties, and the like. In case of operations between distinct class objects, "Class Conversion" is done according to the "Automatic Class Conversion Rule" shown in Fig. 2.3. After the strict check to many aspects of a program, **matc** calls the "C Language Code Generator", while **matx** calls the "Internal Code Generator" and puts the generated code into execution.

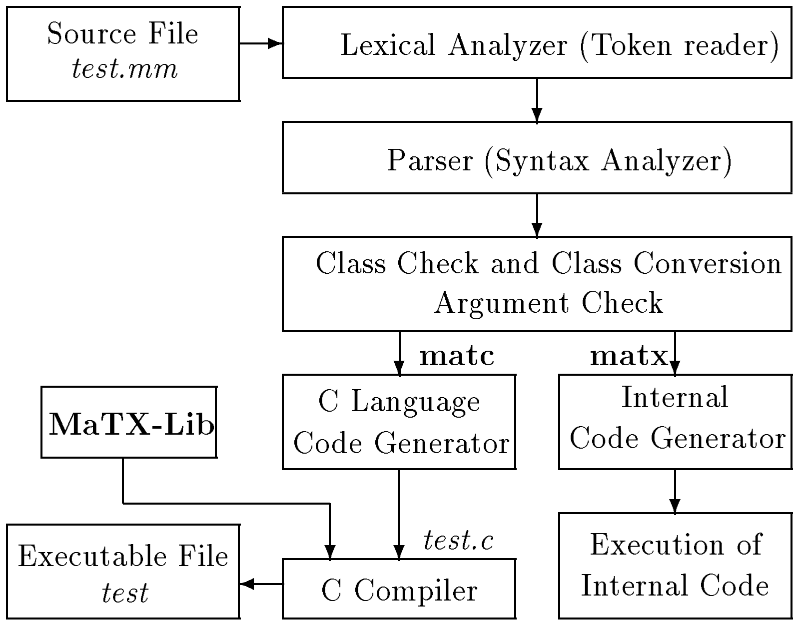

Fig. 2.2 The Internal Process of MaTX

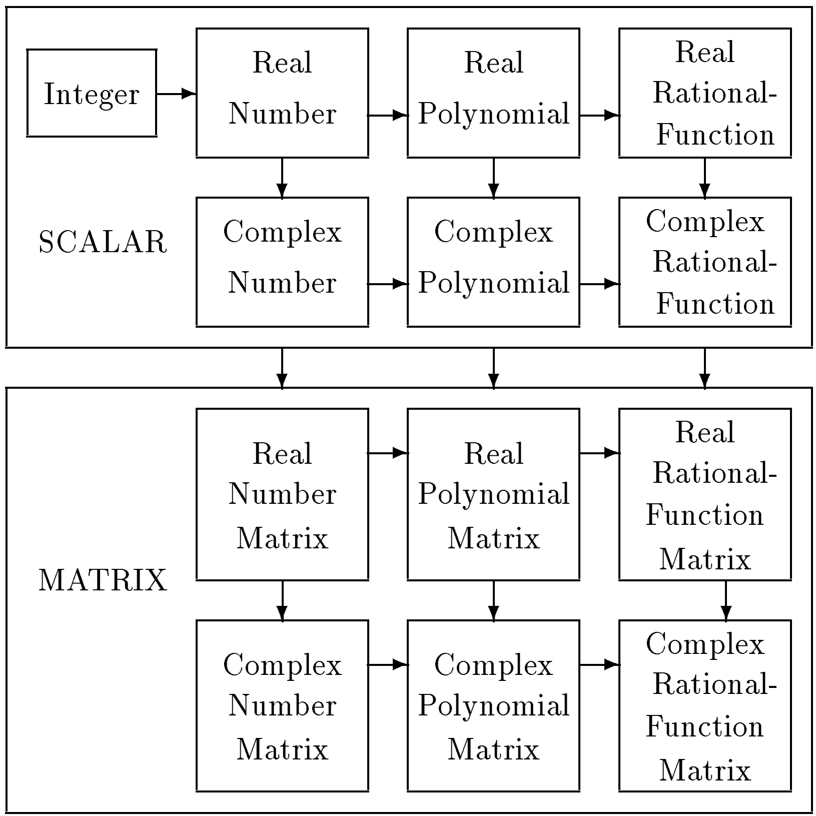

Fig. 2.3 The Automatic Class Conversion Rule

3 Program Developing Process

Figure 3.1 and Fig. 3.2 show the program developing process in Matlab and MaTX, respectively. In the program developing process in Matlab, the users usually realize algorithms in m-files (`test.m`) according to the requirements (`specs`). The m-files are tested several times until they satisfy the requirements. It is possible to call the user's C or Fortran subroutines (`ctest.mtex`), which are produced from source code (`ctest.c`) by C or Fortran compiler, from Matlab function. In the program developing process in MaTX, the users ordinarily make several procedures (mm-files) using a favorite editor (`vi, emacs, or so`) according to the requirements (`specs`). The interpreter (`matx`) enables us to test out the mm-files interactively file-by-file or line-by-line. The examined mm-files are compiled to portable C language files by the compiler (`matc`). It is possible to call your own C functions from MaTX functions by linking C language files (`ctest.c`) to the output of matc (`test.c`). When all C language files are completed, they are compiled and linked up with the run-time library (`MaTX-Lib`) by C compiler (`cc`) to produce the desired executable program (`test`). Finally, the executable file is executed to check whether all requirements are satisfied or not.

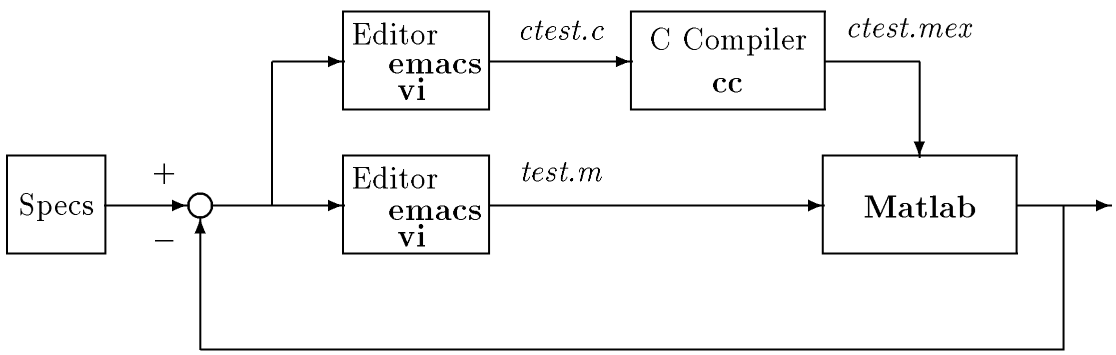

Fig. 3.1 Program Developing Process in Matlab

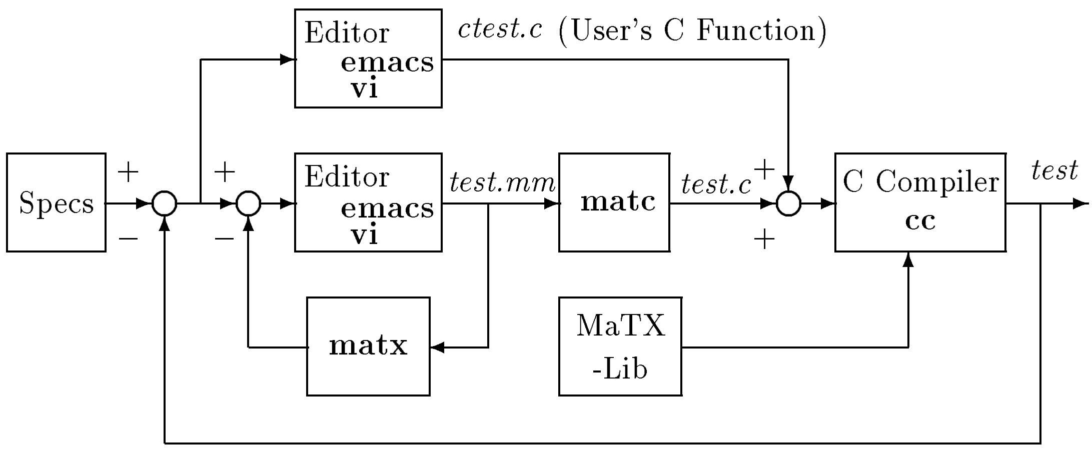

Fig. 3.2 Program Developing Process in MaTX

4 Matrix and Array

MaTX distinguishes Array class objects from Matrix class objects. Array class objects have the same data structure as that of Matrix class objects, but arithmetic operations of arrays are element-by-element in stead of the usual algebraic matrix operations. For array operations of matrices, Matrix class has array operators denoted by the symbols "`.*, ./, .\, .^`". The syntax of matrix manipulation in MaTX is very similar to that used in Matlab. The subscripting abilities of MaTX allow manipulation of rows, columns, individual elements, and submatrices of matrices.

4.1 Entering Simple Matrices

The easiest method of entering a small matrix is to use an explicit list, whose are elements separated by blanks, tabs, commas, or carriage returns, and surround it by square brackets, "`[`" and "`]`". Brackets are also used for indicating the begins and ends of the rows. For example, entering a statement

```
A = [[1 2 3][4 5 6][7 8 9]];
```

or in more natural fashion:

```
A = [[1 2 3]
     [4 5 6]
     [7 8 9]];
```

results in the output

```
=== [A] : (   3,   3) ===
           (   1)           (   2)           (   3)
(   1)  1.00000000E+00  2.00000000E+00  3.00000000E+00
(   2)  4.00000000E+00  5.00000000E+00  6.00000000E+00
(   3)  7.00000000E+00  8.00000000E+00  9.00000000E+00
```

4.2 Matrix Elements

Matrix elements can be any MaTX expressions except List class objects; for example,

```
x = [-1.3 , sqrt(3), (1+2+3)*4/5];
```

results in

```
=== [x] : (   1,   3) ===
           (   1)           (   2)           (   3)
(   1) -1.30000000E+00  1.73205081E+00  4.80000000E+00
```

Individual matrix elements can be specified with indices separated by a comma "`,`" inside parentheses, "`(`" and "`)`", i.e., `x(1,1)` is equal to `-1.3`. Big matrices can be constructed by using small matrices as elements. For example, we could attach another row vector to our matrix **A** with

```
B = [10 11 12];  C = [[A][B]];
```

which result in

```
=== [C] : (   4,   3) ===
            (   1)            (   2)            (   3)
(   1)  1.00000000E+00  2.00000000E+00  3.00000000E+00
(   2)  4.00000000E+00  5.00000000E+00  6.00000000E+00
(   3)  7.00000000E+00  8.00000000E+00  9.00000000E+00
(   4)  1.00000000E+01  1.10000000E+01  1.20000000E+01
```

Any submatrices can be extracted from the big matrix specifying the subportion with the left-top and the right-bottom locations, or the intervals of rows and columns. For example,

```
A = C(1,1,3,3);
```

or equivalently

```
A = C(1:3,1:3);
```

takes the 3×3 matrix to give us back the original matrix **A**. Furthermore, the dimensions of matrices can be changed. For example,

```
D(3:4) = C;
```

results in

```
=== [D] : (   3,   4) ===
            (   1)        (   2)        (   3)        (   4)
(   1)  1.00000E+00  2.00000E+00  3.00000E+00  4.00000E+00
(   2)  5.00000E+00  6.00000E+00  7.00000E+00  8.00000E+00
(   3)  9.00000E+00  1.00000E+01  1.10000E+01  1.20000E+01
```

The element ordering in the matrix `C` is left to right and top to bottom which explains the ordering of the re-arranged matrix in `D`. The i-th row vector and the j-th column vector can be extracted by `A(i,*)` and `A(*,j)` (or `A(i,:)` and `A(:,j)`), respectively.

4.3 Arithmetic Sequence

Arithmetic sequence can be created by using the special character, the colon ":", in side square brackets, "[" and "]". The statement

```
A = [1:4];
```

generates a row array containing the numbers from `1` to `4` with unit increment. It produces

```
=== [A] : (   1,   4) ===
          (   1)       (   2)       (   3)       (   4)
(   1)  1.0000E+00  2.0000E+00  3.0000E+00  4.0000E+00
```

Increments other than one can be used. The statement

```
B = [0 : PI/3 : PI];
```

results in

```
=== [B] : (   1,   5) ===
           (   1)       (   2)       (   3)       (   4)
(   1)  0.0000E+00  1.0472E+00  2.0944E+00  3.1416E+00
```

It should be noted that the vectors created by the colon notation are not matrices but arrays. The colon notation allows the easy generation of tables. To get a vertical tabular form, transpose the row array obtained from the colon notation, compute column of function values, then form a matrix from the two columns. For example,

```
x = [0.0 : 0.2 : 1.0]';   y = exp(x) * sin(x);
z = [x y];
```

produce

```
=== [z] : (   6,   2) ===
             (   1)          (   2)
(   1)  0.00000000E+00  0.00000000E+00
(   2)  2.00000000E-01  2.42655269E-01
(   3)  4.00000000E-01  5.80943901E-01
(   4)  6.00000000E-01  1.02884567E+00
(   5)  8.00000000E-01  1.59650534E+00
(   6)  1.00000000E+00  2.28735529E+00
```

5 Symbolic Computation in MaTX

Symbolic expressions such as polynomials and rational-functions, polynomial matrices, and rational-function matrices are essential to the study of system theory and signal processing. MaTX permits the definition and manipulation of these objects in a natural symbolic fashion. It can solve polynomial equations and evaluate derivatives and integrals symbolically.

5.1 Polynomials and Rational-Functions

MaTX handles polynomials and rational-functions of one variable with real and complex coefficients. Polynomials and rational-functions can be added, subtracted, multiplied, and divided, as usual. At least one polynomial variable must be defined in advance in order to describe symbolic expressions. For example, the statements

```
x = $s$;
A = 4 * x^2 + 5 * x + 6;
```

result in the output

```
A = 4 s^2 + 5 s + 6
```

where "`x = $s$`" is the definition of the polynomial variable and the character "`s`" is always used for the output polynomial variable. The class conversion function "`Polynomial()`" and "`Matrix()`" can be used to convert a row vector to a polynomial and vise versa. The coefficients of the polynomial are identical with the elements of the row vector in ascending order. For example, the statement

```
B = Polynomial([6 5 4]);
```

produces the same polynomial as the previous one. Individual polynomial coefficients can be specified with the index inside parentheses, "(" and ")". For example, the statements

```
A(2) = 1.0; A(1) = 2.0; A(0) = 3.0;
```

result in the output

```
A = s^2 + 2 s + 3
```

A rational-function is described as a division of a pair of polynomials. For example, the statements

```
x = $s$;
B = (x + 1)/(4*x^2 + 5*x + 6);
```

result in the output

```
          s + 1
B = ---------------
    4 s^2 + 5 s + 6
```

Numerator and denominator of a rational-function can be extracted by using the functions, "`Nu()`" and "`De()`", respectively. `Nu(B)` extracts a one-degree polynomial "`s + 1`" and `De(B)` takes a two-degree polynomial "`4s^2 + 5 s + 6`". Polynomials and rational-functions, like real-numbers and complex-numbers, can be used as elements in matrices. This is a very useful feature of MaTX for system theory, and is presented in the next section.

5.2 Symbolic Matrices

Essentially the manipulation of symbolic matrices, i.e., polynomial matrices and rational-function matrices, is identical with that for numerical matrices, i.e., real matrices, complex matrices. There are at lest two convenient ways to describe a polynomial matrix. They are illustrated by the following statements:

```
x = $s$;
A = [[1 2][3 4]]*x^2 + [[5 6][7 8]]*x
    + [[9 10][11 12]];
```

and

```
x = $s$;
A = [[  x^2+5*x+9 , 2*x^2+6*x+10]
     [3*x^2+7*x+11, 4*x^2+8*x+12]];
```

which produce the same result. The former statement uses three matrices as coefficients of a polynomial. The latter one explicitly uses four polynomials as elements of a matrix.

A method of describing rational-function matrices is to use explicitly rational-functions as matrix elements. For example, the statements

```
x = $s$;
A = [[(x+1)/(x+2) , (x+3)/(x+4)]
     [(x+5)/(x+6) , (x+7)/(x+8)]];
```

result in

$$A = \begin{bmatrix} \dfrac{s+1}{s+2} & \dfrac{s+3}{s+4} \\ \\ \dfrac{s+5}{s+6} & \dfrac{s+7}{s+8} \end{bmatrix}$$

Another convenient way to describe rational-function matrices is to utilize the array operation of matrices. For example, the statements

```
x = $s$;
A = [[1 3][5 7]];  B = [[2 4][6 8]];
C = (ONE(2)*x + A) ./ (ONE(2)*x + B);
```

result in the same rational-function matrix as the previous one. The symbol "`./`" denotes the array operator for element-by-element division of matrices. The functions "`Nu()`" and "`De()`" extract the numerator-part and the denominator-part from rational-function arrays, respectively. For example, the statements

```
N = Nu(C);  D = De(C);
```

result in

```
=== [N] : (   2,   2) ===
     [  (   1)    ]  [  (   2)    ]
(   1)    s + 1          s + 3
(   2)    s + 5          s + 7

=== [D] : (   2,   2) ===
     [  (   1)    ]  [  (   2)    ]
(   1)    s + 2          s + 4
(   2)    s + 6          s + 8
```

5.3 Evaluation of Symbolic Expressions

Evaluation of symbolic expressions at a specified value can be done by using the function "`eval()`". The evaluation of symbolic expressions at an array result in the array whose elements are simply the results of the evaluation at an individual elements. For example, the statements

```
x = $s$;
A = 3*x^2 + 4*x + 5;
B = eval(A, 2.0);
C = eval(A, x+1);
D = eval(A, (x+1)/(x+2));
F = eval(A, [1:4]);
```

result in

```
B = 25
C = 3 s^2 + 10 s + 12
```

```
    12 s^3 + 62 s^2 + 107 s + 62
D = ----------------------------
       s^3 + 6 s^2 + 12 s + 8

F = [12  25  44  69]
```

, respectively.

5.4 Derivative and Indefinite Integral

Derivatives and indefinite integrals of symbolic expressions can be obtained by using the functions "`derivative()`" and "`integral()`", respectively. The order of the derivative and the indefinite integral can be specified as the argument of the functions. For example, the statements

```
x = $s$;
A = 12*x^2 + 18*x + 10;  M = [A A];
DM = derivative(M);  DA2 = derivative(A,2);
IM = integral(M);    IA2 = integral(A,2);
```

result in

```
DM = [24 s + 18, 24s + 18]
IM = [4 s^3 + 9 s^2 + 10 s,
      4 s^3 + 9 s^2 + 10 s]
```

and

```
DA2 = 24
IA2 = s^4 + 3 s^3 + 5 s^2
```

, respectively.

5.5 Shift of Coefficients

The coefficients of the symbolic expressions can be shifted toward the lower and higher order by using the functions "`lower()`" and "`higher()`", respectively. The number of shift can be specified as the argument of the functions. For example, the statements

```
x = $s$;
A = 12*x^2 + 18*x + 10;  M = [A A];
LM = lower(M);   LA2 = lower(A,2);
HM = higher(M);  HA2 = higher(A,2);
```

result in

```
LM = [12 s + 18, 12s + 18]
HM = [12 s^3 + 18 s^2 + 10 s,
      12 s^3 + 18 s^2 + 10 s]
```

and

```
LA2 = 12
HA2 = 12 s^4 + 18 s^3 + 10 s^2
```

, respectively.

6 List Class

In the syntax of the Matlab language, functions have the general form :

```
[a, b, c, ...] = fun(d, e, f, ...)
```

where the `...` denotes ellipsis. The `a, b, c, ...` are "left-hand-side" arguments and the `d, e, f, ...` are "right-hand-side" arguments. In other words, functions take multiple input arguments and return multiple output arguments. In the syntax of traditional languages such as Fortran, Pascal, and C, functions use multiple input arguments and return only one value. In order to implement the multiple output functions, we introduced the "List Class" which is a collection of objects not necessarily of the same type. Lists can contain any of the data type as well as other list. Functions can return multiple output values as a list which contains them.

6.1 Entering Lists

The method of describing a List class object is to use an explicit list of objects which are separated by the commas "," and surrounded by a couple of braces, "{" and "}". For example, entering the statement

```
A = {4, (3,4), [1 2]};
```

results in a list which contains an integer, a complex-number, and a real-matrix.

6.2 List Elements

Any MaTX expressions can be used as elements of lists; For example, the statement

```
A = {sqrt(3.0), ((1+2)*3, 4), [1 2] + [3 4]};
```

results in a list which contains a real-number, a complex-number, and a real-matrix. Individual list elements can be extracted with the index and the class name of the element separated by a comma inside parentheses, "(" and ")". For example, the expression `A(1,Real)` is equal to `1.732` and the statements "`A(2) = (3,4); x = A(2,Complex);`" result in `x = (3,4)`. Class name is required for compilation. Long lists can be produced from several short lists. For example, we can append another list `B` to our list `A` by

```
B = {"Hello", 2.0};
C = {A, B};
```

which result in a 5-element list

```
C = {1.732, (9,4), [4 6], "Hello", 2.0}
```

Any part of a list can be extracted by specifying the interval with two indices. For example,

```
A = C(1,3);
```

extracts a 3-element list to give us back the original list `A`.

6.3 Multiple Output Functions

In the syntax of the MaTX language, multiple-output functions have the form

```
{a, b, c, ...} = fun(d, e, f, ...)
```

where the `...` denotes ellipsis. The `a, b, c, ...` are "left-hand-side" arguments and the `d, e, f, ...` are "right-hand-side" arguments. Function "`fun()`" returns multiple output values as a list which contains them. The number of left-hand-side arguments must be less than or equal to that of the elements of the returned list. If the number of left-hand-side arguments is less than that of the elements of the list, then the only top elements of the list which correspond to the left-hand-side arguments are used and the rest are discarded.

6.4 Example

The MaTX has been used to create an illustration of the capabilities and features described above. The demonstration, which is shown in List 1, involves calculation of Bode frequency response and Nyquist frequency response. The following statements are explanations of List 1.

- On the 1st, 19th, and 33rd line, beginning of functions.
- On the 3rd to 5th, 23rd, 24th, 37th, and 38th line, declaration of local variables.
- On the 6th line, declaration of functions.
- On the 8th line, definition of the polynomial variable.
- On the 9th line, generation of a logarithmically spaced vector.
- On the 10th line, entering a 3-degree real rational-function.
- On the 12th and 13th line, calling the multiple-output functions `Bode()` and `Nyquist()`.
- On the 20th, 21st, 34th, and 35th line, declaration of arguments of functions.
- On the 26th and 40th line, definition of the imaginary unit.
- On the 27th and 41st line, evaluation of rational-function arrays.
- On the 28th and 29th line, array operations.
- On the 30th and 42nd line, returning the list which contains two elements.

```
01:Func void main()
02:{
03:  Polynomial s;
04:  Rational G;
05:  Array w, ga, ph, re, im;
06:  List Bode(), Nyquist();
07:
08:  s = $s$;
09:  w = logspace(0.01, 100.0);
10:  G = (s^2 + 3*s + 4)/(s^3 + 5*s^2 + 6*s + 7);
11:
12:  {ga, ph} = Bode(G, w);
13:  {re, im} = Nyquist(G, w);
14:
15:  print w, ga, ph >> "bode-resp.mx";
16:  print re, im    >> "nyquist-resp.mx";
17:}
18:
19:Func List Bode(G, w)
```

```
  Rational G;
  Array w;
{
  Complex j;
  Array resp, ga, ph;

  j = (0,1);
  resp = eval(G, j*w);
  ga = 20.0 * log10(abs(resp));
  ph = arg(resp) * 180.0 / PI;
  return {ga, ph};
}

Func List Nyquist(G, w)
  Rational G;
  Array w;
{
  Complex j;
  Array resp;

  j = (0,1);
  resp = eval(G, j*w);
  return {Re(resp), Im(resp)};
}
```

List 1 Bode and Nyquist Frequency Response

7 Programming in MaTX

In this section we touch on a few programming facilities in MaTX.

7.1 Class Conversion Functions

Although the operations between distinct class objects invoke the automatic class conversions, some operations require explicit class conversions. The class conversion functions can be used for this. Their name is identical with the name of the destination class. For example, the statements

```
A = [[1 2][3 4]];  B = [[5 6][7 8]];
C = A .* B;
D = Array(A) * Array(B);
```

make matrix `C` equal to matrix `D`. The following statement uses the function "`Real()`" to convert a 1 by 1 real-matrix to a real number.

```
A = Real([1 2 3] * [4 5 6]') * [[1 2][3 4]];
```

The function "`String()`" in the statements:

```
A = [[1 2][3 4]];
for (i = 1; i <= 10; i++) {
  print A * i >> "A-" + String(i) + ".mx";
}
```

is used to convert an integer to a string.

7.2 Reference to Functions

In MaTX, a function is not a variable, but it is possible to pass its reference to functions. The reference to a function behaves like its name. We will illustrate this facility by showing the program of the continuous-time and hybrid-time simulation of a linear system

$$\dot{x} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ -2 & -3 & -4 \end{bmatrix} x + \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} u$$

to which the state feedback control

$$u = \begin{bmatrix} -1 & -1 & -1 \end{bmatrix} x$$

is applied. List 2 is the program of the simulation. The functions "`odesolve()`" and "`odediscsolve()`" in MaTX are capable of integrating a broad range of non-linear system functions of both explicit and implicit nature. The function "`odesolve()`" does the continuous-time simulation and the function "`odediscsolve()`" does the hybrid simulation, i.e., the simulation of continuous-time plants with the digital controller. The former function takes five arguments: the initial time (`0.0`), the terminal time (`5.0`), a vector initial condition (`x0`), the reference (`plant`) to the function which calculates the value of the derivative (`dx`) from the time `t` and the values of `x` and `u` at this time, the reference (`controller`) to the function which calculates the value of the input (`u`) from the time `t` and the value of `x` at this time. The last argument (`controller`) can be omitted in case of zero input. In addition to the five arguments, the function "`odediscsolve()`" requires the sampling interval (`dt`) between which the constant input (`u`) is applied to the system.

The functions (in this example `plant()` and `controller()`) which calculate the derivative and the input must have certain formats. The format of the former function specifies that the arguments list contains four variables and the latter one specifies that the arguments list contains three variables. In the derivative function, the first variable `dx` is the calculated derivative and has the same dimension as `x`, the second variable `t` is a real number which represents time, the third variable `x` is a vector (or matrix) of the sate values, and the fourth variable `u` is the input to the system. In the input function, the first variable `u` is the calculated input, the second variable `t` is a real number which represents time, the third variable `x` is a vector (or matrix) of the sate values.

```
Func void main()
{
  Real dt;
  List LC, LH;
  Matrix x0, TC, XC, UC, TH, XH, UH;
  void plant(), controller();

  dt = 0.01;      /* Sampling interval */
  x0 = [1 1 1]'; /* Initial state     */

  LC = odesolve(0.0, 5.0, x0, plant, controller);
  LH = odediscsolve(0.0, 5.0, dt,
                         x0, plant, controller);
```

```
 {TC, XC, UC} = LC;
 {TH, XH, UH} = LH;

 print TC, XC, UC >> "cont-simu.mx";
 print TH, XH, UH >> "hybr-simu.mx";
}

Func void plant(dx, t, x, u)
   Real t;
   Matrix x, dx, u;
{
 Matrix A, b;

 A = [[ 0   1   0]   /* System matrix   */
      [ 0   0   1]
      [-2, -3, -4]];
 b = [0 0 1]';       /* Input matrix    */
 dx = A * x + b * u; /* System equation */
}

Func void controller(u, t, x)
 Real t;
 Matrix u, x;
{
 Matrix f;

 f = [-1, -1, -1];
 u = f * x;
}
```

List 2 Simulation of Continuous-Time Systems

8 Interpreter vs. Compiler

Interpreters usually don't require the users to declare class of variables, because they know everything about all variables at run time. Compilers, however, can't get the information of variables and functions at compile time, unless there are declaration of variables. It seems reasonable to expect that the more information, in addition to algorithms themselves, are given to the interpreters or compilers, the more the rate of computation may be sped up and the efficiency of memory usage can be promoted. And it will get rid of the abuse of variables. Of course, the more information are required, the more time it takes the users to make programs. The relation between the speed of computation and the amount of information is depicted in Fig. 8.1.

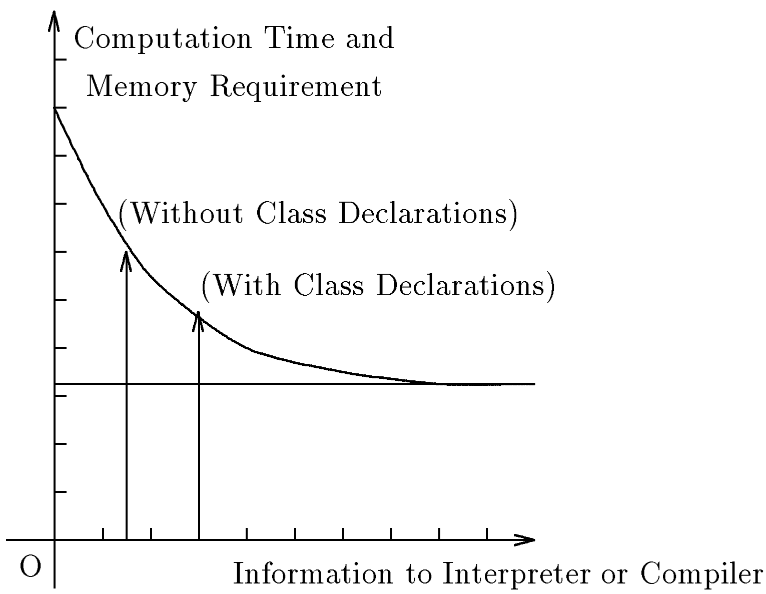

Fig. 8.1 Computation Time and the Amount of Information

8.1 matx vs. matc (Comparison 1)

Here we compare the performance of the interpreter **matx** and the compiled program which was produced from the same source code by using the compiler **matc**. We choose for this the pole-assignment function "PoleAssign.mm" which is similar to the control toolbox algorithm "acker.m" [2] for single input pole placement via linear state feedback. We compare the memory requirements and the execution times by taking random `(A, b)` pairs of state dimensions `n = 1, 2, 3, ..., 16` and asking for placing the eigenvalues of `A + b*f'` on prescribed locations, where is a vector of random (but complex conjugate) eigenvalues. Figure 8.2 and Fig. 8.3 show the memory requirements and the computation times of the interpreter and the compiled program, respectively.

Memory requirement of the interpreter is about twice as much as that of the compiled program. However, there are no remarkable difference in computation time between the interpreter and the compiled program. The following statements may help account for the results.

1. Both the interpreter and the compiled program call the same functions in the library MaTX-Lib which is a collection of matrix-operation and symbolic-computation functions. Accordingly, it takes them the same time to do operations of matrices and symbolic computations.
2. The interpreter (matx) translates functions to the internal code and stores it in the memory. In other words, the sequence of program is predetermined in advance of the execution, though the interpretation of the objects in all statements are done every execution. This preprocessing mechanism saves matx a lot of time.
3. The algorithm of "PoleAssign.mm" is written by using a powerful MaTX matrix notation, and there are few loops with scalar operations. If there are many loops with scalar operations in the program, then the interpreter will be significantly slower than the compiled program. Incidentally, such loops with scalar operations can be written by using integers and real numbers in MaTX, which are written as a vector operation in Matlab.

8.2 matx vs. matc (Comparison 2)

Here we choose for the comparison the continuous-time simulation of a linear system to which a state feedback control is applied. List 2 is the program of the simulation. We compare the

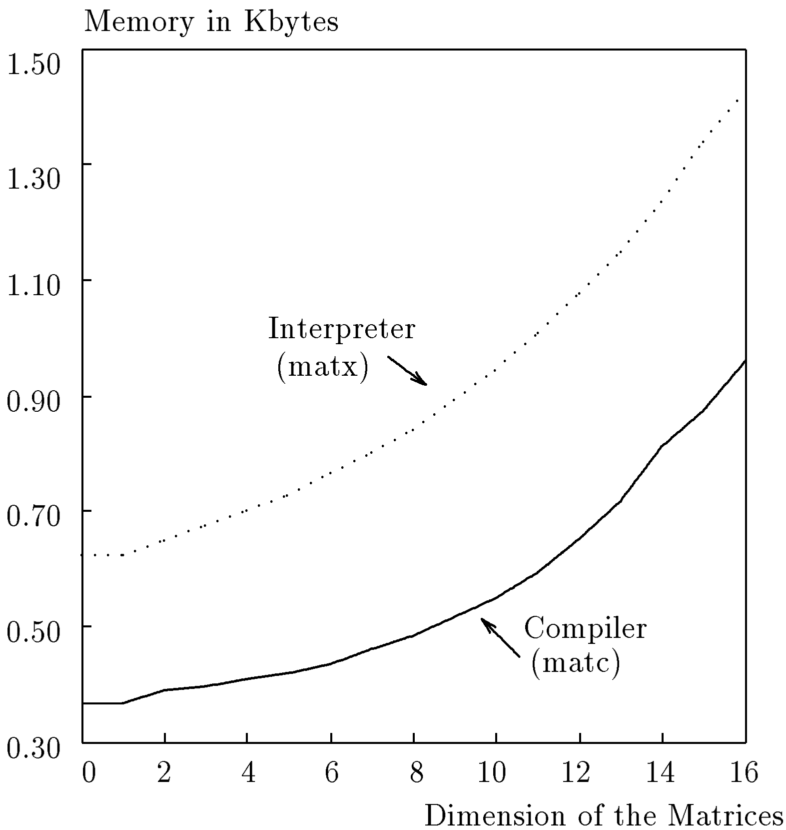

Fig. 8.2 Comparison of Memory Requirements

execution times on some machines, such as MIPS6280, SPARC station 1, Sun4/280, Sun3/260, Sun3/60, and NEWS831(SONY). The relative timings for both cases gave the ratios in favour of matc which is shown in Fig. 8.4. The main reason for the good performance of the compiled program with respect to the interpreter matx is that the integrating function "`odesolve()`" requires a lot of function calls which take the interpreter much more time than the compiled program.

9 Concluding Remarks

A new high-performance programming language MaTX for scientific and engineering computation was presented and its features were described. MaTX provides not only command-driven *interpreter* but also *compiler*. The users can extend the functionality of programs by realizing algorithms as functions in "mm-files". The interpreter (matx) enables us to test out the mm-files interactively. Furthermore, it is possible to make the executable program from the same source code using compiler (matc).

The bigger a matrix, the more complicated to enter a matrix. The user-interface tool, matrix-editor "mated" provides us a very convenient way to enter big matrices [1]. Scientific and engineering data, which was created in MaTX, are easily examined graphically in **XPLOT** [1]. It is a useful graphical drawing environment which runs under X Window System.

The flexibility and strength of MaTX is easily seen to be important for designers who want to combine the interactivity and the performance of optimized FORTRAN and C libraries. The translator which translates m-files to mm-files is under development. At the moment, MaTX is checked to run on the following machines under the operating system.

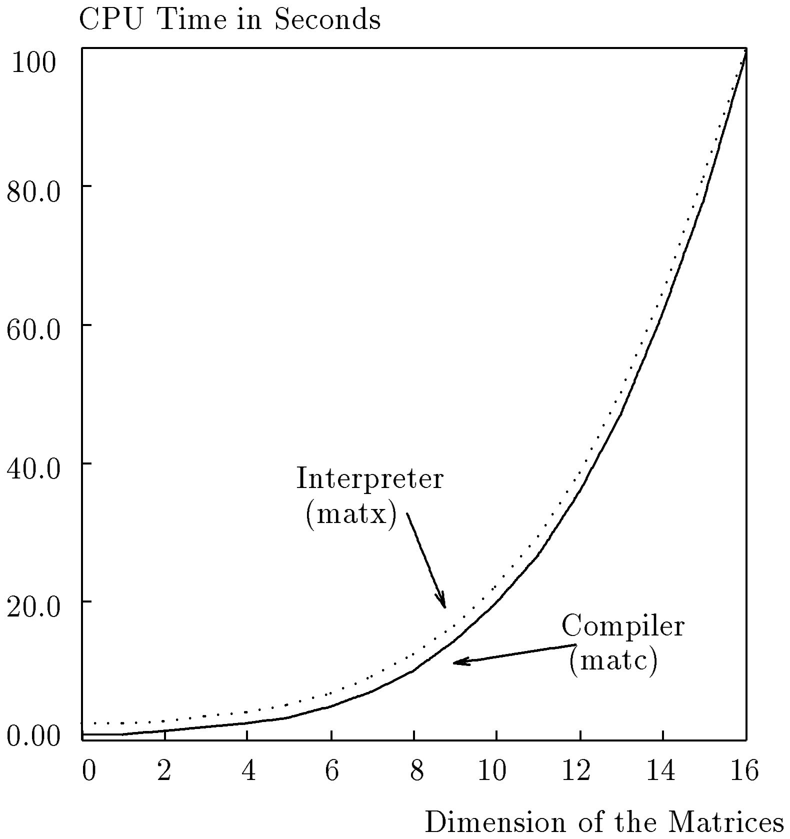

Fig. 8.3 Comparison of Computation Time

Mips RC6280 UMIPS4.52C
Sun4 SunOS4.0.0, SunOS4.0.3, SunOS4.1.1
Sun3 SunOS4.0.0, SunOS4.0.3, SunOS4.1.1
NEWS-830(SONY) NEWS-OS 2.0, 3.3
J3100 UNIX System V/386 3.0, PC-DOS 3.1
PC9801(NEC) MS-DOS 3.1, 3.30

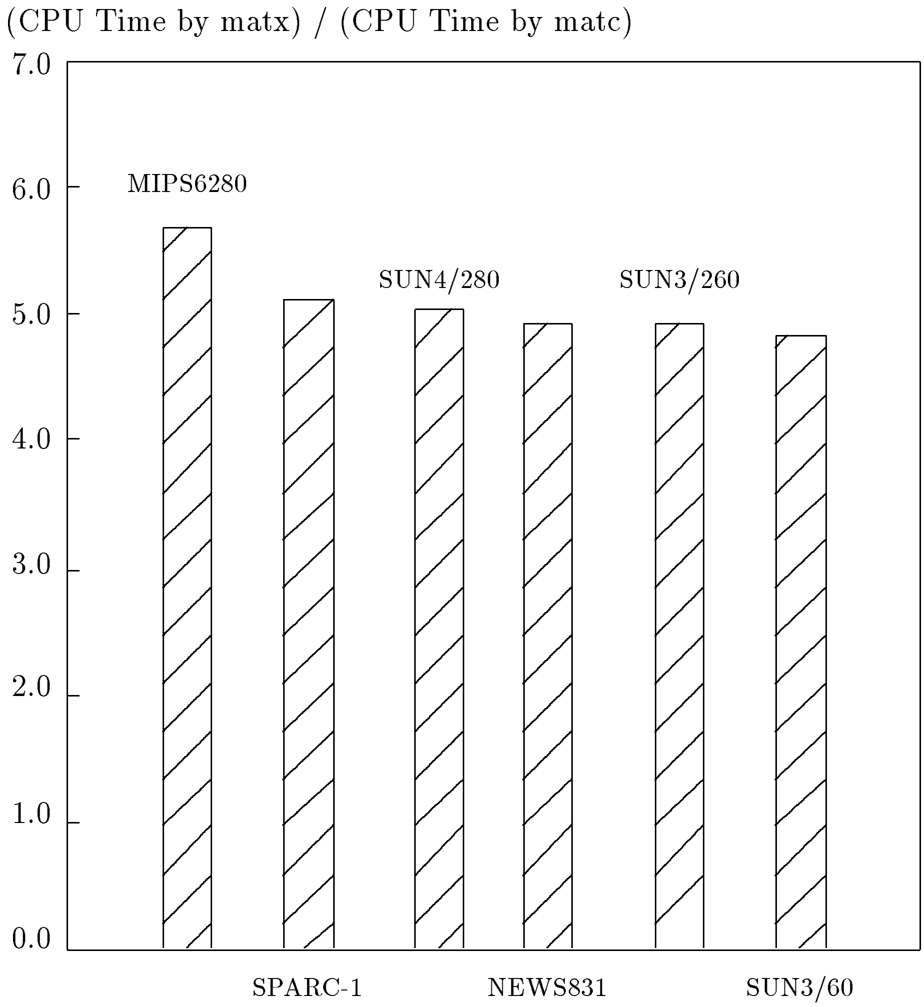

Fig. 8.4 Ratios of CPU Time in favour of matc

References

[1] Masanobu Koga and Katsuhisa Furuta. MaTX : A high-performance interactive software package for scientific and engineering computation. *Proc. of CADCS'91, Swansea, U.K.*, pages 39–44, 1991.

[2] C. Moler, J. Littel, and S. Bangert. *PC-MATLAB – User's Guide.* N. Main St., Sherborn, MA 01770, USA, 3.1 edition, 1987.

[3] Cleve Moler. *MATLAB – User's Guide.* Alberquerque, USA, 1980.

[4] W.A. Renes, M. Vanbegin, P. Van Dooren, and J.W.J.J. Beckers. The MATLAB gateway compiler. a tool for automatic linking of FORTRAN routines to MATLAB. *CADCS'91, Swansea, U.K.*, pages 95–100, 1991.