SYMBOLIC TOOLS

Symbolic Tools for GAUSS

ECONOTRON SOFTWARE, INC.

Version 1.0

Jon Breslaw

June, 2003

Support:

Econotron Software
447 Grosvenor Ave.
Westmount, P.Q. Canada
H3Y-2S5

Tel: (514) 939-3092
Fax: (514) 938-4994
Eml: support@econotron.com
Web: http://www.econotron.com

Contents

1 Introduction **1**

1.1 Concept 1

1.2 Symbolic Modes 3

1.3 Example files 5

2 Installation and Testing **7**

2.1 Installation Requirements 7

2.2 Installing Symbolic Tools 7

2.3 Testing Symbolic Tools 8

3 Tutorial **9**

3.1 Example 1 9

3.2 Example 2 11

3.3 Example 3 12

3.4 Example 4 13

4 Symbolic Data Types and Operations **19**

4.1 Data types 19

4.2 Symbolic Elements 20

4.3 Symbolic List 20

4.4 Symbolic Vector 21

4.5 Symbolic Matrix 22

5 Symbolic Tools Commands - Summary **25**

5.1 Summary 25

5.1.1 GAUSS Commands 25

5.1.2 Maple Commands 26

6 Reference - GAUSS commands **27**

7 Reference - Maple commands **45**

8 GAUSS functions **53**

Chapter 1

Introduction

1.1 Concept

The concept of symbolic manipulation is to augment the numeric and graphical capabilities of GAUSS with additional types of mathematical computations. These include:

- Symbolic Algebra. This includes analytic differentiation and integration, as well as simplification.
- Linear Algebra. This capability allows for the exact (as opposed to the numeric) evaluation of matrix forms, including inverses, determinants and eigenvalues.
- Language Extension. This permits the evaluation of a wide range of mathematical functions and matrix forms, effectively extending the GAUSS language.
- Precision. Numerical evaluation of functions can occur at any specified level of accuracy.

The computational work is carried out by the Maple kernel using the Open Maple interface. Maple is a symbolic mathematics package developed at the University of Waterloo, and distributed by Waterloo Maple, Inc.

Symbolic Tools consists of a set of functions that provide an interface between GAUSS and the Maple kernel. These functions provide the means of sending variables or code from GAUSS to the kernel, running GAUSS code within the kernel, and returning variables back to GAUSS.

GAUSS is a programming language in which each variable is given a name - `gnp, coeff, foo` – any legal GAUSS name. Associated with each name is a type - a scalar, matrix, string, stringarray, etc. Each variable has associated value(s) - 6.4 for a scalar, "Hello World" for a string, {2,4,6} for a vector, etc. Symbolic Tools adds a new datatype - a symbol - to the set of datatypes used by GAUSS. So in normal GAUSS mode:

```
z = x + y;
```

results in z being generated as a scalar taking the value 5 assuming that x is a scalar with the value 2 and y is a scalar with the value 3. The same command in which both x and y are symbols results in z being created as a symbol, with the value $x + y$. Thus in symbolic mode, the same GAUSS syntax is used, but instead of manipulating value or strings, one manipulates symbols. Symbols can represent scalars, vectors or matrices, but the content of these variables does not need to be specified.

As a simple example, if vectors `v1` and `v2` are specified as:

```
v1 = {a, b};
v2 = {c, d};
```

then:

```
v1'v2 = a*c + b*d
```

The ability to use symbolic arithmetic enables a GAUSS optimization process to use analytic gradients and Hessians – this is called automatic differentiation. There exists a number of GAUSS optimization packages – `Maxlik, Optmum, CML`, etc – that require the gradient and Hessian of the function being optimized in order to evaluate the appropriate search direction. In the default, these are calculated numerically using forward differencing:

$$df/db_i = [f(b_1, ..b_i + h, ..b_n) - f(b)]/h$$

The Hessian is similarly calculated using the second derivative. For the Hessian, the number of function evaluations increases quadratically with the number of parameters. Thus, as the gradients and Hessians have to be recalculated at each iteration, this process becomes very time intensive if there are a large number of parameters.

These optimization packages provide mechanisms whereby the user can specify the procedures that will return values of the gradient and/or Hessian, instead of doing forward differencing. Thus, as a trivial example, if the function were:

$$F = ln(b);$$

then the user could specify functions for the gradient and Hessian:

$$\begin{aligned} GF &= 1/b; \\ HF &= -1/b^2; \end{aligned}$$

Symbolic Tools can create GAUSS procs for the gradients and Hessian of the specified function, and this approach works very well for enabling automatic differentiation in GAUSS. Using Monte Carlo simulation of a Tobit example with 2000 observations and 11 parameters, the AD gradient took 10% of the time required for forward differences using `gradp` - ie. approximately a 10 fold speed improvement. Similar results were also obtained for the Hessian, with the additional advantage that the AD methodology generated much more precise estimates of the gradients and Hessian.

1.2 Symbolic Modes

Symbolic operations are carried out using the Maple kernel. There are two main modes utilized:

1. Direct Mode: Symbolic code is sent to Maple, where the symbolic operations are stored. Values are then sent to Maple, the symbolic operations are carried out using these values, and the numeric data is returned to GAUSS.

2. Compiled Mode: Symbolic code is sent to Maple, which then creates a GAUSS proc which replicate the symbolic operations carried out in Maple.

A trivial example demonstrates both ideas. We wish to evaluate the gradient of the function $sin(x * y)$ at the point $x = .5$, $y = .75$.

```
txt = "      slist = {x,y};
             llf = sin(x*y);
             llfg = gradp(llf,slist);
      ";
```

Using this direct mode method, this chunk of code is executed using the `symrun` command, the values for x and y are sent to the Maple kernel, and the results retrieved.

```
call symrun(txt);
call symput(x,"x");
call symput(y,"y");
rslt = symget("llfg");
```

The `symrun` command generates (in Maple) the symbolic variables `slist, llf` and `llfg`. `llfg` is a symbolic vector:

```
llfg = { y*cos(x*y), x*cos(x*y) }
```

We evaluate the value `llfg` takes at $x = .5, y = .75$ by specifying the values of the symbols `x` and `y` using the `symput` statement, and retrieving the new value of `llfg` using the `symget` statement.

In compiled mode, the same code is executed using the `symproc` command:

```
call symproc("diffsin","x,y","llfg",txt);
```

This generates a GAUSS procedure called `diffsin` as a string, with an input argument `x,y`, and creating an output `llfg`. The procedure is also compiled, and can be immediately called as a standard GAUSS proc.

```
proc diffsin(x,y);
local t0, t2, unknown;
unknown = zeros(2,1);
t2 = cos(x .* y);
unknown[1+0] = sumc( t2 .* y) ;
unknown[1+1] = sumc( t2 .* x) ;
retp(unknown);
endp;
```

Thus the difference between the two modes is that in Direct Mode, the numeric evaluation takes place under Maple, while under Compiled Mode it takes place under GAUSS. The latter is between 10 and 100 times faster, and thus for most operations where speed is essential, such as automatic differentiation, Compiled Mode is more suitable. The two example files intro1.e and introl2.e in the symbolic\samples\tutorial folder demonstrate these two modes.

Besides symbolic analysis, Symbolic Tools can be used to augment GAUSS functionality by using Maple commands that are not available in GAUSS. To take a trivial example, one wishes to know the value of the norm of matrix. This is undertaken using the `symmaple` statement:

```
x = rndu(3,4);
xnorm = symmaple("norm(x)",0);
```

These examples are coded in the file symbolic\samples\tutorial\into3.e.

1.3 Example files

The symbolic\samples folder contains a large number of example files that demonstrate the capabilities of Symbolic Tools.

Tutorial These files are described above – the new user should start with these files, before progressing to the other examples.

GAUSS Each file in this folder demonstrates how the specified GAUSS command is used within Symbolic Tools.

AD The Automatic Differentiation folder contains two subfolders (Maxlik and Optmum). See the file readme.txt which describes AD, and the difference between the two folders.

Applications A number of applications, including integration and non-linear estimation.

Chapter 2

Installation and Testing

This chapter describes the hardware and software configuration required to run Symbolic Tools on your computer.

2.1 Installation Requirements

The Symbolic Tools (vsn. 5) system requirements are:

- Windows 9x, NT4, ME, 2000, or XP.
- GAUSS for Windows 4.0 or higher, or the GAUSS Engine for Windows 4.0 or higher.
- Maple 9 for Windows or higher. An evaluation version of Maple 9 available at http://register.maplesoft.com/TrialDownload.asp

2.2 Installing Symbolic Tools

Before you open the product package, please read the license agreement that accompanies Symbolic Tools. By installing and using the product, you accept the terms of this agreement.

The program files on the CD are compressed, so you cannot simply copy them to your computer. Rather, you must run the installation program which decompresses the files and copies them to your hard disk in the appropriate directories.

1. Insert the Symbolic Tools CD into the appropriate drive.
2. From the Windows Start menu, chose Run.
3. Type d:\setup.exe (where d: is the letter for your CD drive).
4. Choose OK.
5. Follow the instructions on the screen.

The installation routine creates a folder called symbolic on the GAUSS or GAUSS Engine folder. Within this folder, the following subfolders are created

doc	This folder contains the Symbolic Tools help files and manual.
lib	This folder contains the Symbolic Tools Maple package.
samples	This folder contains demonstration files for using Symbolic Tools.

2.3 Testing Symbolic Tools

Launch GAUSS (or engauss.exe for the GAUSS Engine) and run the file:
symbolic\samples\tutorial\symtest.e.

This opens the Maple kernel, and reports some statistics about the kernel. There are three other files in the tutorial folder - intro1.e, intro2.e and intro3.e, which you should also run to check that the installation has been correctly performed. These files also demonstrate programming techniques in Symbolic Tools.

Chapter 3

Tutorial

GAUSS is a numeric based application, in that it operates on variables that have specified values, such as $x = 2, y = 4$ or $z = "abc"$. Thus $x * y$ results in 8. Symbolic applications - such as Maple - do not operate solely on numerical values - they can in addition, act on symbolic values. So if $x = a$, and $y = b$, then $x * y$ results in $a * b$. This can be very useful - for example, we might want to have the indefinite integral of x^2, but not yet want it evaluated. So we would like to have this indefinite integral available as $x^3/3$.

Symbolic Tools makes these symbolic results available to GAUSS as procedures. Four examples are presented below:

3.1 Example 1

We require the determinant of a matrix where the elements on the principal diagonal are symbolic. Obviously this could be evaluated using GAUSS, but the idea might be that one had a huge matrix, and one wanted to know the determinant if only a couple of elements change.

The GAUSS program to do this is fairly simple, and is shown below in the string `txt`. Besides the GAUSS command `symmat`, which is used to define a symbolic matrix, the code in `txt` is conventional GAUSS . The `symproc` command is used to create a GAUSS procedure – this command takes the procedure name, the procedure input arguments, the procedure output arguments and the procedure code as arguments. In this example, `symproc` takes the code in the string `txt`, creates a GAUSS proc called `symdet`, and compiles it. It is then called with numeric arguments in the standard manner. Thus Symbolic Tools uses procedures to map symbolic arithmetic to numerical output.

```
*******************************************************************
The GAUSS program:

 library symbolic;
 call symstate(reset);
 proc symdet; endp;
 txt = "
       x11 = x[1];
       x22 = x[2];
       amat = symmat(2,2,{x11,2,8,x22});
       rslt = det(amat);
       ";
 call symproc("symdet","x","rslt",txt);    // compile the procedure
 let xdiag = 1 9;                          // some intial values
 rslt = symdet(xdiag);                     // call the compiled proc
"rslt \n" rslt;

*******************************************************************

The Output:

rslt
      -7.0000000
*******************************************************************

The computer generated proc:

proc symdet(x);
local t0,  t2, unknown;
      t2 = x[1+0] .* x[1+1]-16.0;
  retp( t2 );
endp;

*******************************************************************
```

3.2 Example 2

We require the analytic gradient of a function. The example below shows a simple example, but this provides the basis for automatic differentiation.

The GAUSS program to do is shown below. The string `txt` creates the gradient of the function $sin(x*y)^2$, which is then created as a proc called diffsin. Note both that `gradp` is overloaded, so that it can accept a symbol list (`slist`) instead of numeric values. Also note that the proc allows arguments that are both scalars and matrices.

```
*******************************************************************
The Gauss program:

 library symbolic;
 proc diffsin; endp;                          // dummy proc
 call symstate(reset);                        // symbolic reset
 txt = "
        slist = {x,y};
        llf = sin(x*y)^2;
        llfg = gradp(llf,slist);
       ";

 call symproc("diffsin","x,y","llfg",txt);  // compile the proc
 rslt = diffsin(0.5,0.75);                    // call the proc
 "\n rslt " rslt;


*******************************************************************

The Output:

rslt
      0.51122907
      0.34081938

*******************************************************************
```

```
The computer generated proc:

proc diffsin(x,y);
local t0,     t1,     t2,     t3,     t4,  unknown;
      unknown = zeros(2,1);
      t1 = x .* y;
      t2 = sin(t1);
      t3 = cos(t1);
      t4 = t2 .* t3;
      unknown[1+0] = sumc( 2.0 .* t4 .* y) ;
      unknown[1+1] = sumc( 2.0 .* t4 .* x) ;
      retp(unknown);
endp;

*******************************************************************
```

3.3 Example 3

We require the indefinite integral of a function. The example below shows a very simple example, but much more complicated cases exist.[1] The GAUSS program to do this is shown below. The `txt` provides the integral of $y * x^2$ (which is not very exciting) as a proc called `intsim`. `Intquad` is used - we don't need `intquad1, intquad2`, etc. since the level of integration is given by the length of `slist`. Again, `intquad` is overloaded to permit both numeric and symbolic integration.

```
*******************************************************************
The Gauss program:

 library symbolic ;
 proc intsim; endp;                               // dummy proc
 call symstate(on);                               // symbolic reset
 txt = "
        slist = {x,y};
        llf = y*(x^2);
        intg = intquad(llf,slist);
```

[1] This technique was used to create the integral of the Pearson function in "Simulated Latent Variable Estimation of Models with Ordered Categorical Data", (J. Breslaw and J. McIntosh) *Journal of Econometrics*, 87, 1998, pp 25-47.

```
       ";
 call symproc("intsim","x,y","intg",txt);    // compile the proc
 rslt = intsim(0.75,0.5);                    // call the proc
 "rslt \n" rslt;

******************************************************************

The Output:

rslt
     0.017578125

******************************************************************

The computer generated proc:

proc intsim(x,y);
local t0,  t1,  t2,  t5,  unknown;
     t1 = y .* y;
     t2 = x .* x;
     t5 = t1 .* t2 .* x ./ 6.0;
     retp( t5 );
endp;

******************************************************************
```

3.4 Example 4

Finally, a real world example - a Tobit estimation. The code shown is a Monte Carlo simulation of the Hessian, based on 2000 observations, 4 parameters, and 200 replications. The results show element [2,1] of the Hessian for the first 10 replications, while the time in each case is for the full 100 replications. As can be seen, the symbolic code is over 10 times faster than `hessp`.

```
******************************************************************
/* Symbolic example: Tobit.e

       proc tobit(y,indx,sigma);
       y is the censored variable
       indx is the vector of the index
       sigma is the parameter for the std. error of the residual
```

```
*/

/*****************************************************************
 **                        Initialization                      **
 *****************************************************************/


 library symbolic;
 call symstate(reset);          // Initialize Symbolic tools
 call symdebug(off);            // Debug mode shows line by line

 rndseed 12345;
 mode = 1;                      // 0 - gradient; 1 - Hessian
 replics =  200;                // number of replications
 num = 2000;                    // number of observations

                                // create the data
  sigma = 2;
  xmat = ones(num,1) ~rndu(num,2);
  y =  2*sigma*rndn(num,1);
  x1= xmat[.,1];
  x2= xmat[.,2];
  x3= xmat[.,3];
  xrnd = rndu(replics,4);       // parameters

  rslt = zeros(replics,2);      // initialization
  clear time1, time2;
  jj = 2;
  title = "Tobit process";
  process = "Gradient" $| "Hessian";
  cls;
  print /flush title ;



/*****************************************************************
 **                    Code for a  Tobit process               **
 *****************************************************************/

 // GAUSS code as proc

proc llfn(bpar);
   local llf1,  indx, bvec, sigma, llf2,llf,h;
```

```
   h = .000001;
   bvec = bpar[1:3];
   sigma = bpar[4];
   indx = xmat*bvec;
   sigma = maxc(sigma|h);
   llf1 = -((y-indx)^2) / (2*sigma) - .5*ln(2*pi*sigma);
   llf2 = ln(h+cdfnc(indx/sqrt(sigma)));
   llf = (y .gt 0).*llf1 + (y .le 0).*llf2;
   retp(sumc(llf));
 endp;

 // Gauss symbolic representation as string

 txt =  "
     h = .000001;
     slist = symset(bpar,b,4);
     indx = x1*b1+x2*b2+x3*b3;
     sigma = maxc(symvec({h,b4}));
     llf1 = -(y-indx)^2 / (2*sigma) - .5*ln(2*pi*sigma);
     llf2 = ln(h+cdfnc(indx/sqrt(sigma)));
     llf = sumc((y .gt 0).*llf1 + (y .le 0).*llf2);
     llfg = gradp(llf,slist);
     llfh = hessp(llf,slist);
";


/*******************************************************************
 **              create a GAUSS proc that does the AD         **
 *******************************************************************/

 proc llfproc; endp;                             // dummy proc
 if mode == 0;
   call symproc("llfproc","bpar","llfg",txt);   // gradient
 else;
   call symproc("llfproc","bpar","llfh",txt);   // hessian
 endif;


/*******************************************************************
 **             Monte Carlo using forward difference          **
 *******************************************************************/

 cls;
```

```
print /flush title ;
print /flush
  "\n Evaluating " process[mode+1]  " using Forward Differencing...";;
d1=date;
j = 1;
do while j <= replics;
    if mode == 0;
       gvec = gradp(&llfn,xrnd[j,.]');
    else;
       gvec = hessp(&llfn,xrnd[j,.]');
    endif;
    rslt[j,1] = gvec[1,jj];
    j = j+1;
endo;
time1 = ethsec(d1,date)/100;

/*******************************************************************
**           Monte Carlo using symbolic code                **
*******************************************************************/

print "";
print /flush
  "\n Evaluating "  process[mode+1]  " analytically...";;
d1=date;
j = 1;
do while j <= replics;
    x = vec(xrnd[j,.]);
    gvec = llfproc(x);
    rslt[j,2] = gvec[1,jj];
    j = j+1;
endo;
time2 = ethsec(d1,date)/100;

/*******************************************************************
**                  Monte Carlo results                          **
*******************************************************************/

cls;
print /flush title  ":  " process[mode+1];
" ";
"      numerical        analytical ";
rslt[1:10,.];
```

```
 " \nTime   (secs)\n " time1~time2;
 " \nSpeed factor:   " time1/time2;
 " \n";
```

```
*******************************************************************

The Output:

   numerical       analytic

      -1365.4884       -1364.2409
      -979.19588       -976.33761
      -1899.4885       -1900.6809
      -4442.5751       -4448.6259
      -1238.0001       -1231.6229
      -2540.5553       -2560.5614
      -1148.6819       -1149.3890
      -1311.6106       -1268.6765
      -1065.7274       -1063.2483
      -1157.2675       -1158.1068

time (secs)
        10.532000        1.0310000
*******************************************************************
```

Chapter 4

Symbolic Data Types and Operations

As in GAUSS, each variable refers to an entity - these can be individual scalars, or more complex groupings such as vectors and matrices. In GAUSS, each element is given a value, such as `x = 4.7;` for a numeric component, or `txt = "abc";` for a string component. In symbolic mode, the same GAUSS syntax is used, but instead of manipulating value or strings, one manipulates symbols. Symbols can occurs as elements, or as compontents of a list, a vector, or a matrix.

4.1 Data types

Symbolic Tools supports the following data types:

scalar (real, complex, symbol)
vector (real, complex, symbol)
matrix (real, complex, symbol)
string

The following types are not supported:

arrays
string arrays
structures
GAUSS data sets
band and sparse matrices
character vectors
date and time types
graphics
fuzzy operators

4.2 Symbolic Elements

Each symbol is designated by a name, which is just the GAUSS variable name - `x, y, foo` - any legal GAUSS name. Thus the command:

```
sigma = b;
```

defines a GAUSS variable, `sigma` which has a value of `b`. `b` is neither numeric, nor a string; rather it is just a symbol.

4.3 Symbolic List

The main structure used in the symbolic arithmetic is a list - an ordered list of symbols:

```
slist = {a,b,c,d};
```

These lists are the basis for most symbolic operations.

List Creation Lists can be created in a number of ways:

1. `slist = {a,b,c,d};`
2. `slist = symlist(4,x);` This creates `slist = {x1,x2,x3,x4};`.
3. `slist = symlist(vect);` This converts a vector to a list.
4. `slist = symset(bpar,b,4);` This creates `slist = {b1,b2,b3,b4}` and assigns each symbol to the corresponding element of the GAUSS vector `bpar`.

5. `slist = symdat(dta,b,4);` This creates `slist =` {`b1,b2,b3,b4`} and assigns each symbol to the corresponding column of the GAUSS matrix `dta`.

Lists are used in creating symbolic vectors and matrices (see below), and as arguments of functions that evaluate accross a list, such as symbolic differentiation and integration.

Element Identification

```
slist = {a,b,c,d};
slist[3] is 'c'
slist[2:3] is {b,c}
```

List Concatenation If vlst1 and vlst2 are two lists, then concatenation occurs by:

```
vlst1 | vlst2;
vlst1 ~ vlst2;
symlist({vlst1,vlst2}).
```

List Operations

```
blst = {b1,b2,b3};
xlst = {x1,x2,x3};

blst+xlst  ->   {b1+x1,b2+x2,b3+x3}
blst.*xlst ->   {b1*x1,b2*x2,b3*x3}
blst*xlst  ->   b1*x1+b2*x2+b3*x3
```

4.4 Symbolic Vector

A vector consisting of symbols is defined as:

```
v = symvec({x,y,z});
```

or

```
slist = {x,y,z};
v = symvec(slist);
```

All symbolic vectors are symbolic matrices with a single column. Thus all the operations applicable to symbolic matrices apply to symbolic vectors.

4.5 Symbolic Matrix

A 2x3 matrix consisting of symbols is defined as:

```
m = symmat(2,3,{a,b,c,d,e,f});
```

or

```
matlst = {a,b,c,d,e,f};
m = symmat(2,3,matlst);
```

Symbolic matrix operations closely resemble numeric operations:

Element Identification

```
xmat = matrix(2,2,{a,b,c,d});
xmat[2,1] is 'c'
```

Concatenation

```
q = {a,b,a}|{b,b,c};
v1 = symmat(1,2,{a,b});
v2 = symmat(1,2,{b,b});
v3 = symmat(1,2,{a,c});
v = v1|v2|v3;
m = {a}~v1;
```

Transpose $m = x'$; You may need to use (x') to get the precedence correct.

Transpose Multiply $xx = x' * x$ (not $x'x$).

Element Operations $.*$ and $./$ work as expected but may need parenthesis to get precedence correct.

Matrix Multiplication $x * y$ works as in GAUSS if the type of x and y are known. Thus if x and y are defined by a `symmat` statement, then there is no problem. If x and y are symbols, then it is not clear whether these symbols represent scalars or matrices. In the default, when x and y are both symbols, $x * y$ generates $x. * y$. To force matrix multiplication, use the Maple syntax $x\& * y$. $x' * y$ is always taken as matrix multiplication.

Example:

```
bhat = inv(x'*x) &* (x'*y);
```

x and y are symbols, so they could be matrices, or scalars. The $x' * x$ term is recognized as requiring a matrix type mulitplication, since this operation occurs almost exclusively on matrices. However, inv$(x' * x)$ mulitplied by $(x' * y)$ is a symbol times a symbol, so to ensure the required matrix multiplication, we use the &* format.

Matrix Division x/y works as in GAUSS if the type of x and y are known. If x and y are symbols, then it is not clear whether these symbols represent scalars or matrices. In the default, when x and y are both symbols, then x/y generates $x./y$. To force the matrix interpretation of x/y, use the syntax $x\&/y$.

Other Operations The following work as in GAUSS :

$x!$	factorial
$x\%y$	modulo division
$x.*.y$	kronecker product
$x*~y$	horizontal dircect product

Logic Operations GAUSS assumes that true = 1 and false = 0. Maple does not use this rule. Symbolic Tools evaluates dot functions (., ., .and etc) so as to return unity or zero as expected. Relational and boolean operators (non dot) are used for flow control, and return true or false.

Chapter 5

Symbolic Tools Commands - Summary

5.1 Summary

5.1.1 GAUSS Commands

These commands are called from GAUSS.

SYMARG	— Specifies data argument in Sympro.
SYMDEBUG	— Controls debugging mode
SYMGAUSS	— Sends a GAUSS command to the Maple kernel
SYMGET	— Retrieves a variable from the Maple kernel
SYMHELP	— Displays online help
SYMMAPLE	— Sends a GAUSS command to the Maple kernel
SYMMODE	— Controls syntax mode
SYMOUT	— Controls output buffer
SYMPROC	— Compiles a GAUSS proc derived from symbolic code
SYMPUT	— Sends a variable to the Maple kernel
SYMRUN	— Sends code to the Maple kernel for execution
SYMSTATE	— Controls the Maple kernel
SYMTEST	— Procedure for testing AD code

5.1.2 Maple Commands

These commands are called from within a string sent to the Maple kernel.

SYMDAT	— Assign symbolic names to matrix columns - returns a list
SYMEVAL	— Evaluates a Maple function that expects algebraic inputs
SYMLIST	— Creates a symbolic list
SYMMAT	— Creates a symbolic matrix
SYMSET	— Assign symbolic names to a vector - returns a list
SYMVEC	— Creates a symbolic vector

Chapter 6

Reference - GAUSS commands

These commands extend the GAUSS language to allow for control over the Maple kernel and symbolic manipulation. Symbols require a new data type - a list - and these commands provide support for this data type.

- **Purpose**

Specifies data argument in `symproc`.

- **Format**

SYMARG(*argn*);

- **Inputs**

argn literal, argument number.

- **Remarks**

When using the `symproc` command, the user specifies the input arguments that are required by the proc. Normally, no additional knowledge is required. However, when one of the arguments is data, symproc needs to know this to undertake the correct dimensions of the output of the proc. The argument number of a parameter that is data is supplied in `argn`.

The default value (2) is set in symbolic.dec.

- **Example**

```
library symbolic;
   ...
call symarg(3);
call symproc("myproc","avec,bvec,dta","llf",txt);
```

In this example, the third input argument of myproc is data.

- **See also**

SYMPROC

Purpose

Controls debugging mode of Symbolic Tools.

Format

SYMDEBUG(*mode***);**

Inputs

mode	literal, debug mode (`on, [off]`).

Remarks

In normal operation, code is sent to the Maple kernel, and results are retrieved from the kernel with no output from the kernel unless an error is detected. Turning the debug *mode* to `on` results in the kernel displaying each line of Maple code as it is executed, and displaying any file created with `symproc`.

The default mode is `off`; this is set in symbolic.dec.

Example

```
library symbolic;
call symdebug(on);
```

In this example, the debug mode is enabled.

- **Purpose**

Sends a GAUSS command to the kernel for execution.

- **Format**

rslt = SYMGAUSS (*txt*) **;**

- **Inputs**

txt	string, GAUSS command

- **Outputs**

rslt	GAUSS output

- **Remarks**

The `symgauss` command executes the single GAUSS command embedded in the string *txt* in the Maple kernel, and returns the result. The arguments to the GAUSS command are automatically satisfied from the GAUSS workspace.

This facility permits the user to execute a GAUSS command under Maple, with input and output managed seamlessly. Note that Maple is case sensitive, so the code in *txt* should be lower case. Only GAUSS commands with single returns are permitted.

- **Example**

```
library symbolic;
...
let x[3,3] = 4 2 6 8 5 7 3 8 9;
rslt = symgauss("cond(x)");
"The result is: " rslt;
```

In this example, the GAUSS matrix `x` is used in the `symgauss` command to derive the condition number of `x` using the Maple kernel. The result is returned and displayed by GAUSS.

Purpose

Retrieves a variable from the Maple kernel.

Format

rslt = SYMGET (*name*) **;**

Inputs

name	string, variable name

Outputs

rslt	GAUSS variable

Remarks

The `symget` command retrieves a variable from the Maple kernel, and stores it in the GAUSS workspace. Valid data types are Maple matrices, vectors, scalars and strings. Numeric data is stored as regular GAUSS variables, while a symbolic result is stored as a string. Memory allocation and type recognition are taken care of automatically.

Example

```
library symbolic;
 ...
rslt = symget("llfg");
"The result is: " rslt;
```

In this example, a Maple variable, `llfg`, is retrieved and displayed by GAUSS.

- **Purpose**

Displays the Symbolic Tools online help

- **Format**

SYMHELP ;
SYMHELP (*topic*);

- **Inputs**

topic	literal, Maple topic

- **Remarks**

The `symhelp` command with no arguments displays the online help file. If *topic* is specified, the appropriate Maple help file is displayed in Notepad.

- **Example**

```
library symbolic;
symhelp;
symhelp (linalg);
symhelp codegen;
```

The first call to symhelp brings up the online help facility. The second and third calls show alternative methods of retrieving the Maple help page for `linalg` and `codegen` respectively.

Purpose

Sends a Maple command to the kernel for execution.

Format

rslt = SYMMAPLE (*txt, mode***) ;**

Inputs

txt	string, Maple command
mode	literal, evaluation mode

Outputs

rslt	GAUSS output

Remarks

The `symmaple` command executes the Maple command embedded in the string *txt* in the Maple kernel, and returns the result. The arguments to the Maple command are automatically satisfied from the GAUSS workspace.

mode determines how the arguments are treated. *mode* is set to zero for those cases where the entire argument is treated as a whole, such as the determinant of a matrix. *mode* is set to unity for those cases where the function is evaluated on each element of the argument, such as sin.

This facility permits the user to execute a Maple command under Maple, with input and output managed seamlessly. Note that Maple is case sensitive, so the code in *txt* should be correctly cased for Maple. Only Maple commands with single returns are permitted.

Example

```
library symbolic;
 ...
let x[3,3] = 4 2 6 8 5 7 3 8 9;
rslt = symmaple("ratform(x)",0);
"The result is: " rslt;
```

In this example, the GAUSS matrix `x` is used in the `symmaple` command to derive the rational canonical form (or Frobenius form) of `x` using Maple. The result is returned and displayed by GAUSS.

- **Purpose**

Controls the Symbolic Tools syntax.

- **Format**

SYMMODE (*mode*) **;**

- **Inputs**

mode	literal, operating mode (`[Gauss]`, `Maple`)

- **Remarks**

In the default, GAUSS code is parsed to Maple format prior to being sent to the Maple kernel. Pure Maple code can be also sent to the kernel, in which case no parsing is required. The default mode (GAUSS) is set in symbolic.dec

- **Example**

```
library symbolic;
call symmode(maple);
txt = " with (linalg);
        x:=matrix(2,2,[1,2,3,4]);
        z:=trace(x); ";
call symrun(txt);
```

In this example, parsing is turned off, so the raw code is sent to Maple as is.

- **Purpose**

Controls the Symbolic Tools output buffer.

- **Format**

SYMOUT (*mode*) ;

- **Inputs**

mode	literal, output mode

- **Remarks**

This command provides a diary of the Maple output. *mode* takes the following values:

mode	on	Turns on the output buffer.
	off	Turns off the output buffer.
	reset	Clears and initializes the output buffer.
	view	Displays the output buffer using Microsoft Notepad, and clears the output buffer.

- **Example**

```
library symbolic;
call symout(reset);
   ...
call symout(view);
```

In this example. the output buffer is cleared, some commands are carried out, and the buffer is then viewed using Notepad.

- **Purpose**

Creates and compiles a GAUSS proc derived from symbolic code.

- **Format**

prc = SYMPROC(*pname, inarg, outarg, txt***);**

- **Inputs**

pname	string, proc name
inarg	string, input arguments
outarg	string, output argument
txt	string, symbolic code

- **Outputs**

prc	string, GAUSS proc

- **Remarks**

The `symproc` command creates and compiles a GAUSS proc based on symbolic code evaluated by the Maple kernel. In a trivial example, if one had a function $sin(x)$, and wished to have access to a proc that was the gradient of this function, then that proc would simply return $cos(x)$. `symproc` automates this process. The GAUSS code is parsed, sent to the Maple kernel, and executed. Maple takes the required value (say a gradient), and returns the optimized code to create this gradient. `Symproc` reparses this optimized code to be a GAUSS compatible proc, and compiles the proc. This proc is then accessible for use by the user.

The idea behind writing code for a symbolic process is to compose the code based on a single (symbolic) observation. Symbolic Tools takes care of creating a proc for n observations. Since all symbols will become matrices when the proc is run under GAUSS, all operations will be dot evaluated.

In most cases, a single argument (a parameter vector) is passed to the compiled proc. Symbolic Tools is usually able to ascertain all the information needed from the code itself. An exception is where data is passed to the proc as an argument, in which case one should use the `symarg` command to specify which argument is data.

For the most part, the Maple kernel will evaluate GAUSS commands as Maple code. Note that Maple is case sensitive, so the code in *txt* should be lower case. However, you can force a GAUSS evaluation of a GAUSS command by capitalizing the command.

- **Example**

```
library optmum, symbolic;
optset;
call symstate(reset);
  ... load data y, x1 - x5

proc fct(avec);
 local indx, llf;
 indx = avec[1] + avec[2]*x1 + avec[3]*x2 + avec[4]*x4^avec[5]
      + avec[6]*ln(x5+avec[7]);
 llf = sumc((y-indx)^2);
 retp (llf);
endp;

txt = "
  slist = symset(bpar,a,7);
  indx = a1 + a2*x1 + a3*x2 + a4*x4^a5 + a6*ln(x5+a7);
  llf =  sumc((y-indx)^2);
  llfg = gradp(llf,slist);
  llfh = hessp(llf,slist);
";

proc gradproc; endp;
proc hessproc; endp;

gcode = symproc("gradproc","bpar","llfg",txt);
hcode = symproc("hessproc","bpar","llfh",txt);

__opgdprc = &gradproc;
__ophsprc = &hessproc;
avec0 = ones(7,1);
{x, f, g, retcode} = optmum(&fct,avec0);
```

This example shows how one would estimate a non-linear least squares problem using the GAUSS optmum command. The libraries are specified, and each package is set. `Optmum` requires pointers to procedures that return the function to be minimized, and optionally the gradient and Hessian. The function is specified in `fct`, which requires 7 parameters, and the data `(y, x1-x5)` is in core. The equivalent symbolic GAUSS code is supplied as a string `(txt)`, augmented with the code to generate the gradient and Hessian. GAUSS procs for the gradient and Hessian are generated by symproc, as shown. Note the dummy procs for the gradient and Hessian – these are needed so that GAUSS can compile the program. Note also that the code for these procs is returned in `gcode` and `hcode` respectively, so it is easy to cut and past the code into your program if the model specification does not change - ie. once Symbolic Tools has generated the gradient and Hessian, it doesn't need to be run again. `Optmum` requires that pointers to user supplied gradients and Hessians are placed in `__opgdprc` and `__ophsprc` respectively. A starting value for the parameters is specified, and the optimization is carried out by `Optmum`.

Purpose

Sends a variable to the Maple kernel.

Format

SYMPUT (*var, name*) **;**

Inputs

var	literal, GAUSS variable
name	string, variable name

Remarks

The `symput` command send a GAUSS variable *var* to the Maple kernel, and stores it under the name *vname*. The supported data types are shown below.

Supported	
	Real matrix
	Complex matrix
	Real vector
	Complex vector
	Real scalar
	Complex scalar
	String
Not Supported	
	Character vector
	Sparse matrix
	Array
	String Array
	Structures
	Data set

Example

```
library symbolic;
call symstate(reset);
x = rndu(5,2);
call symput(x,"xmat");
```

In this example, the GAUSS matrix `x` is sent to the Maple kernel and stored under the name `xmat`.

- **Purpose**

Executes GAUSS or Maple code using the Maple kernel,

- **Format**

SYMRUN (*txt*) **;**

- **Inputs**

txt	string, code

- **Remarks**

The `symrun` command executes the code embedded in the string *txt* by the Maple kernel. If `symmode` has been set to GAUSS, the code is first parsed. No output is returned unless an error is trapped, or unless `symdebug` has been set to `ON`. Note that Maple is case sensitive, so the code in *txt* should be lower case.

This facility permits the user to execute GAUSS code under Maple. This can be used to allow for symbolic operations in GAUSS, to allow for grater precision than is available in GAUSS, and to extend the GAUSS language through the use of Maple commands.

- **Example**

```
1.   library symbolic;
      ...
      txt = "x = symmat(2,2,{a,b,c,d}); detx = det(x);" ;
      call symrun(txt);
      detx = symget("detx");
      "The result is: " detx;

2.    txt1 = "
       fx= (c1-x)/(c0-c1*x+c2*(x^2));
      intg = intquad(fx,{x});
      ";
      call symout(on);
      call symrun(txt1);
      call symout(view);
```

3.
```
let a[3,3] = 1 2 3 1 2 3 1 5 6;
call symput(a,"a");
call symrun("ca = charpoly(a,x);");
ca = symget("ca");
"The charpoly of a is " ca;
```

In the first example, the determinant of the symbolic matrix `x` is evaluated by the kernel, and retrieved and displayed by GAUSS. In the second example, a Pearson function is defined, and the symbolic integral is evaluated by the kernel, and then displayed using the viewer. In example 3, the GAUSS language is extended by using the Maple `charpoly` function.

Purpose

Controls the Symbolic Tools environment.

Format

SYMSTATE (*mode*) ;

Inputs

mode	literal, mode `(on, off, reset)`

Remarks

mode can take three values:

`on`	Initialize the Symbolic Tools environment.
`off`	Closes the Symbolic Tools environment.
`reset`	Reinitialize the Symbolic Tools environment.

This command is required to initiate the Symbolic Tools environment. Once loaded, the Maple kernel records all Symbolic Tools activity. The reset mode clears the Maple workspace (like `new` in GAUSS). A session can be started with either mode set to `on` or `reset`.

Example

```
library symbolic;
call symstate(reset);
```

In this example, the Maple kernel is initialized at the beginning of a session.

■ Purpose

Provides a mechanism for testing AD code.

■ Format

proc_f, proc_g, proc_h = SYMTEST (*&fg, &fs, param, dta***);**

■ Inputs

&fg	pointer to GAUSS procedure that returns the function.
&fs	pointer to GAUSS procedure that specifies the symproc call.
param	literal, typical parameter vector required by the function.
dta	literal, typical data matrix required by the function (or 0).

■ Outputs

proc_f	string, symbolic code for the function proc
proc_g	string, symbolic code for the gradient proc
proc_h	string, symbolic code for the hessian proc

■ Remarks

To use automatic differentiation from within an optimization program, such as Maxlik, one needs to define procedures that takes a parameter argument, and returns the gradient and Hessian respectively. The Symbolic tools `symproc` command is used to create these procedures. However, before running the estimation, one wants to make sure that the symbolic procedures are correct. `symtest` does this.

fs is a GAUSS proc that provides symtest with the information needed to test the symbolic code. *fs* returns 5 elements:

1	string	the input argument(s) to the symbolic code eg "bvec , xmat"
2	string	the text of the symbolic code
3	string	the variable name for the function (llf)
4	string	the variable name for the gradient (llfg)
5	string	the variable name for the Hesssian (llfh)

`symtest` evaluates the symbolic function, gradient and Hessian, and prints out the results along with the comparable GAUSS estimates (based on forward differencing) for validation. Strings containing the symbolic code as GAUSS procs for the function, gradient and Hessian are returned.

■ Example

See the example file ADTest.e in the symbolic\samples\AD folder.

Chapter 7

Reference - Maple commands

These commands extend the GAUSS language to allow for symbolic manipulation. Symbols require a new data type - a list - and these commands provide support for this data type. The following commands are only applicable within the Maple kernel, and are used as part of a string that is sent to the Maple kernel.

- **Purpose**

  Assign symbolic names to columns of a matrix.

- **Format**

  slst = SYMDAT (*m, col*)**;**
  slst = SYMDAT (*m, sn, col*)**;**

- **Inputs**

m	literal, matrix name.
sn	literal, symbol
col	numeric, column or range

- **Outputs**

slist	list

- **Remarks**

  The `symdat` command assigns symbols to specified columns of a data matrix. This facility permits the data to be entered as an argument to the procedure created by the `symproc` command. Typically, if a vector x is required within a procedure, then specifying the vector as the symbol "x" will permit GAUSS to pick up the global vector `x` when the procedure is executed. However, MAXLIK requires that both the parameter vector and the data be arguments to procedures that return gradients and Hessians, and `symdat` provides the required functionality.

- **Example**

```
1.  y = symdat(dta,1);

2.  xlist = symdat(dta,x,2:4);
    indx = x1*b1+x2*b2+x3*b3;

3.  xlist = symdat(xmat,v,5);
```

The first example assigns the symbol `y` to the first column of the matrix `dta`. The second example creates three symbols - `x1, x2, x3` - corresponding to columns 2, 3 and 4 of dta respectively. `xlist` is a list of these three symbols - ie xlist = {x1,x2,x3}. The creation of an index is also shown in this example. In the third example, five symbols are created (`v1` through `v5`) corresponding to the the first five columns of `xmat`.

- **See also**

  SYMPROC

■ Purpose

Evaluates a Maple function that expects algebraic inputs.

■ Format

rlst = SYMEVAL (*fn, arg1,arg2..*)**;**

■ Inputs

fn	literal, Maple function.
arg	literal, argument

■ Outputs

slist	result

■ Remarks

The `symeval` command is a utility function that allows the evaluation of a Maple command that normally requires algebraic input. This can always be achieved using the Maple `map, map2` and `zip` commands, but `symeval` makes it easy.

■ Example

```
x = symmat(2,2,{a,b,c,d);
b = symeval(BesselK, 2, x);
```

In this example, we wish to calculate the BesselK of each of the elements of the matrix `x`. BesselK objects to this, since it requires an algebraic input. `symeval` permits the use of matrices for Maple functions that require algebraic input.

- **Purpose**

Creates a symbolic list.

- **Format**

s = **SYMLIST** (*v*);
s = **SYMLIST** (*n, sym*);

- **Inputs**

v	literal, nx1 vector.
n	numeric, number of elements.
sym	literal, symbol

- **Outputs**

s	list

- **Remarks**

A list in Maple is an ordered sequence of expressions or symbols. (A vector is a one dimensional array). The `symlist` command enables the creation of a list, or the conversion of a vector to a list.

- **Example**

```
1.   let v = 1 2 3;
     call symput(v, "v");
     call symrun( "vlst = symlist(v);");

2.   vlst = {v1, v2, v3};

3.   vlst = symlist(3, v);
```

Example 1 shows how a vector can be converted to a list. Example 2 shows how a list can be created from the individual entities. The same operation is carried out in example 3; this format is useful if there are a large number of elements in `vlst`.

- **See also**

SYMSET

- **Purpose**

Creates a symbolic matrix.

- **Format**

s = SYMMAT (*n, m ,vlst*)**;**

- **Inputs**

n	numeric, row dimesnsion.
m	numeric, column dimension.
vlst	literal, symbolic list

- **Outputs**

s	nxm matrix

- **Remarks**

The symmat command creates a symbolic matrix from a list.

- **Example**

```
1.  veclst = {a,b,c,d,e,f};
    m = symmat(2,3,veclst);

2.  m = symmat(2,3,{a,b,c,d,e,f});

3.  v1 = symvec({a,d});
    v2 = symvec({b,e});
    v3 = symvec({c,f});
    m = v1~v2~v3;
```

All three examples are equivalent. m is the 2x3 matrix:

a	b	c
d	e	f

- **Purpose**

Assign symbolic names to a vector.

- **Format**

slst = SYMSET (*v, sn , ord***);**

- **Inputs**

v	literal, vector.
sn	literal, symbol
ord	literal, order or range

- **Outputs**

slst	list

- **Remarks**

The `symset` command creates a symbolic list and assigns each symbol to the corresponding element of a GAUSS vector. *v* is either the name of an argument that is passed to the procedure, or is the name of a global GAUSS vector or matrix.

- **Example**

```
1.  slist = symset(bpar,b,4);
    indx = x1*b1+x2*b2+x3*b3;
    sigma = b4;
       ...
    llfg = gradp(llf,slist);

2.  b1 = bpar[1];
    b2 = bpar[2];
    b3 = bpar[3];
    sigma = bpar[4];
    slist = {b1,b2,b3,sigma);

3.  vlist = symset(bpar,v,7:9);
```

The first example is taken from a Tobit estimation, in which the first three elements of the parameter vector, `bpar`, are the structural coefficients, and the last parameter is the variance. The symset command creates a list of the four parameters in `slist`, as well as associating `b1` through `b4` with the respective elements of `bpar`. `slist` is needed as an argument to `gradp`. Equivalent code is shown in example 2. Example 3 shows how a subset of a parameter vector can be associated - in this case, `v1`, `v2`, and `v3` are associated with `bpar[7]`, `bpar[8]` and `bpar[9]` respectively.

- **Purpose**

Creates a symbolic vector.

- **Format**

s = SYMVEC (*vlst*)**;**

- **Inputs**

vlst	literal, symbolic list

- **Outputs**

s	nx1 vector

- **Remarks**

The `symvec` command creates a vector from a list. In the context of Symbolic Tools , a vector is an nx1 matrix.

- **Example**

```
1.  veclst = {x,y,z};
    v = symvec(veclst);

2.  v = symvec({x,y,z});

3.  v = symmat(3,1,{x,y,z});
```

All three examples are equivalent.

Chapter 8

GAUSS functions

The following list provides information on the functionality of each GAUSS command in a Symbolic Tools context. Note that these functions are case sensitive.

Notes

1. In `symproc` mode, matrix arguments must be passed either as a single parameter, (a scalar symbol), or as a numeric matrix.
2. Does not work in `symproc` mode, since code optimization is not compatible with user specified order.
3. Requires numeric argument.
4. In `symproc` mode, requires integer argument.
5. Only operates in `symproc` mode. Note that matrix arguments must be passed as a single parameter, ie. either a scalar symbol, or the name of an existing GAUSS matrix.

abs	ok	
acf	ok	
arccos	ok	
arcsin	ok	
atan	ok	
atan2	ok	
base10	ok	
besselj	ok	
bessely	ok	
boxcox	ok	
break	ok	
call	ok	
cdfbeta	ok	For arg(2), see notes 3 and 4.
cdfbvn	ok	See note 1.
cdfbvn2	ok	See note 1.
cdfbvn2e	na	Use cdfbvn2.
cdfchic	ok	
cdfchii	ok	See note 1.
cdfchinc	ok	See note 5.
cdffc	ok	For arg(3), see notes 3 and 4.
cdffnc	ok	See note 5.
cdfmvn	ok	See note 5.
cdfgam	ok	
cdfn	ok	
cdfnc	ok	
cdfni	ok	See note 1.
cdfn2	ok	
cdftc	ok	For arg(2), see notes 3 and 4.
cdftci	ok	See note 1
ceil	ok	
chol	ok	
choldn	ok	
cholsol	ok	
cholup	ok	
chrs	ok	
code	ok	
cols	ok	
complex	ok	
cond	ok	See note 1.
conj	ok	
conv	ok	
corrm	ok	
corrvc	ok	

corrx	ok	
cos	ok	
cosh	ok	
counts	ok	See note 1.
countwts	ok	See note 1.
crossprd	ok	
crout	na	Use lu.
croutp	na	Use lu.
cumprodc	ok	
cumsumc	ok	
curve	ok	See note 5.
debug	na	Use symdebug.
delif	ok	
design	ok	See note 1.
det	ok	
detl	na	Use det.
dfft	na	Use fft.
dffti	na	Use ffti.
diag	ok	
diagrv	ok	
digamma	ok	
do	ok	
dummy	ok	See note 1.
dummybr	ok	See note 1.
dummydn	ok	See note 1.
eig	ok	
eigh	na	Use eig.
eighv	na	Use eigv.
eigv	ok	Normalization for eigenvectors may differ from GAUSS.
eqsolve	ok	Solves numerically using `fsolve`.
erf	ok	
erfc	ok	
exp	ok	
eye	ok	
fft	ok	
ffti	ok	
fftm	ok	See note 5.
fftmi	ok	See note 5.
fftn	na	Use fft.
floor	ok	
fmod	ok	
for	na	Use do.
gamma	ok	
gammaii	ok	See note 1.

gradp	ok	Overloaded to provide symbolic differential symbolic: gradp($f(x, y), x, y$); numeric: gradp($f(x, y), x, y, x0$); $x0$ is a 2x1 vector.
hasimag	ok	
hess	ok	See note 1.
hessp	ok	Overloaded to provide symbolic differential. symbolic: hessp($f(x, y), x, y$); numeric: hessp($f(x, y), x, y, x0$); $x0$ is a 2x1 vector.
if	ok	
imag	ok	
indexcat	ok	See note 1.
indnv	ok	See note 1.
intgrat	na	Use intquad.

intquad	ok	Overloaded to provide indefinate integral.
		symbolic: intquad($f(x, y), x, y$);
		numeric: intquad($f(x, y), x, y, x0$);
		1st row of x0 is upper bound, second row is lower bound.
		Upper and lower bounds can be functions. (as in intgrat).

intrsect	ok	3rd parameter is ignored, and is optional.
intsimp	na	Use intquad.
inv	ok	
invpd	ok	
invswp	na	Use pinv.
iscplx	ok	See note 1.
isinfnanmiss	ok	
ismiss	ok	
lag1	ok	
lagn	ok	
let	na	use symvec and symmat.
ln	ok	
lncdfbvn	ok	See note 1.
lncdfbvn2	ok	See note 1.
lncdfmvn	ok	See note 5.
lncdfn	ok	
lncdfn2	ok	
lncdfnc	ok	
lnfact	ok	
lnpdfn	ok	
lnpdfmvn	ok	
lnpdfmvt	ok	
lnpdft	ok	
log	ok	
lower	ok	

lowmat	ok	
lowmat1	ok	
ltrisol	na	Use solpd.
lu	ok	
lusol	na	Use solpd.
machepsilon	ok	
matalloc	ok	
matinit	ok	Uses the equivalent `GAUSS` function in `symproc` mode.
max	ok	The function max(b0,b1) is supplied.
maxc	ok	If argument has more than 2 elements, see note 1.
maxindc	ok	See note 1.
mbesseli	ok	
meanc	ok	
median	ok	See note 1.
minc	ok	If argument has more than 2 elements, see note 1.
minindc	ok	See note 1.
miss	ok	
missex	ok	
missrv	ok	
moment	ok	
new	na	Use symstate(reset).
null	ok	See note 1.
ols	ok	Returns coefficient vector.
olsqr	ok	
olsqr2	ok	
ones	ok	
orth	ok	See note 1.
pacf	ok	
packr	ok	See note 1.
pdfn	ok	
pi	ok	
pinv	ok	
polychar	ok	
polyeval	ok	
polyint	ok	
polymake	ok	
polymat	ok	
polymroot	na	
polymult	ok	
polyroot	ok	
princomp	ok	See note 1.
prodc	ok	

QNewton	na	
QProg	na	
qqr	ok	
qqre	na	Use qqr.
qqrep	na	Use qqr.
qr	ok	
qre	na	Use qr.
qrep	na	Use qr.
qrsol	ok	
qrtsol	ok	
qtyr	ok	
qtyre	na	Use qtyr.
qtyrep	na	Use qtyr.
qyr	ok	
qyre	na	Use qyr.
qyrep	na	Use qyr.
quantile	ok	See note 1.
rank	ok	
rankindx	ok	See note 1.
real	ok	
recode	ok	
recserar	ok	In `symproc` mode, 2nd argument is treated as a scalar.
recsercp	ok	
recserrc	ok	
reshape	ok	
rev	ok	
rfft	na	Use fft.
rffti	na	Use ffti.
rndbeta	ok	See note 1.
rndgam	ok	See note 1.
rndi	ok	
rndKM	na	Use the equivalent rnd function.
rndLC	na	Use the equivalent rnd function.
rndn	ok	
rndnb	ok	See note 1.
rndns	ok	
rndp	ok	See note 1.
rndseed	ok	See note 2.
rndu	ok	See note 1.
rndus	ok	
rndvm	ok	See note 1.
rotater	ok	
round	ok	

rows	ok	
rref	ok	See note 1.
scalinfnanmiss	ok	
scalmiss	ok	
schtoc	na	
schur	ok	See note 3.
selif	ok	
seqa	ok	
seqm	ok	
setdif	ok	See note 1.
shiftr	ok	
sin	ok	
sinh	ok	
solpd	ok	
sortc	ok	See note 1.
sortind	ok	See note 1.
sortindc	na	
sortmc	ok	See note 5.
spline	na	
sqpsolve	na	
sqrt	ok	
stdc	ok	
strindx	ok	
strlen	ok	
strput	ok	
strrindx	ok	
strsect	ok	
submat	ok	
subscat	ok	
substute	ok	
sumc	ok	
svd	ok	
svd1	ok	See note 3.
svd2	ok	See note 3.
svdcusv	na	Use svd2.
sdvs	na	Use svd.
tan	ok	
tanh	ok	
toeplitz	ok	
trigamma	ok	
trimr	ok	
trunc	ok	
type	ok	

typecv	na	Use type.
union	ok	
uniqindx	ok	See note 1.
unique	ok	See note 1.
upmat	ok	
upmat1	ok	
upper	ok	
utrisol	ok	
vals	ok	
varmares	na	
vcm	ok	
vcx	ok	
vec	ok	
vech	ok	
vecr	ok	
xpnd	ok	
zeros	ok	

Index

AD debugging 43

commands
– summary 25
– symarg 28
– symdat 46
– symdebug 29
– symeval 47
– symgauss 30
– symget 31
– symhelp 32
– symlist 48
– symmaple 33
– symmat 49
– symmode 34
– symout 35
– symproc 36
– symput 39
– symrun 40
– symstate 42
– symtest 43

data argument 28
debugging 29

help 32

installation 7

kernel
– execute 30, 33, 40
– initialization 42
– retrieve 31
– send 39

output 35

symbolic
– element 20
– evaluation 47
– list 20, 48
– matrix 22, 46
– proc 36
– vector 21
syntax 34