MATVEC 2.4.0

Tianlin Wang, Rohan L. Fernando and Stephen D. Kachman

November 10, 2024

Acknowledgements

Without generous supports and thoughtful ideas from my academic adviser Prof. Rohan Fernando, there would not have been such a package.

Dr. Ignacy Misztal and I used to have several nice discussions about matvec development. In particular, his encouragement is greatly appreciated.

Contents

1 A Tour of matvec **9**

1.1 Running matvec . . . 9

1.1.1 Interactively . . . 9

1.1.2 From a script . . . 9

1.1.3 From standard input . . . 10

1.2 Special Attention . . . 10

1.3 Start-up File . . . 10

1.4 Creating Objects . . . 10

1.5 Getting Online Help . . . 11

1.6 Scalar Arithmetic . . . 12

1.7 Matrix-Vector Arithmetic . . . 13

1.8 Variables, Expressions, and Statements . . . 15

1.9 Inbreeding Coefficient Computation . . . 16

2 Working on Objects **17**

2.1 Special Object: this . . . 17

2.2 Scalar Object . . . 19

2.3 Vector Object . . . 19

2.4 Matrix Object . . . 20

2.5 String Object . . . 20

2.6 Pedigree Object . . . 21

2.6.1 sample . . . 22

2.6.2 input . . . 22

2.6.3 inbcoef . . . 22

2.6.4 logdet . . . 22

2.6.5 rela . . . 23

2.6.6 reld . . . 23

2.6.7 inv . . . 23

2.6.8 size . . . 23

2.6.9 nbase . . . 23

2.6.10 save . . . 23

2.7 Data Object . . . 23

2.8 Model Object . . . 23

2.9 StatDist Object . . . 23

3 Matrix Object **24**
3.1 Creation . . . 24
3.1.1 Using operators [and] . . . 24
3.1.2 Using object-creating functions . . . 24
3.2 Operator . . . 25
3.2.1 Arithmetic operators . . . 25
3.2.2 Relational operators . . . 27
3.2.3 Logical operators . . . 27
3.3 Manipulation . . . 27
3.3.1 Accessing to elements and sub-matrix . . . 28
3.3.2 Horizontal and vertical concatenation . . . 28
3.3.3 Selecting elements . . . 29
3.3.4 Reshaping and Rotating . . . 29
3.4 Matrix Statistics . . . 31
3.4.1 Descriptive statistics . . . 31
3.4.2 Best linear unbiased prediction (BLUP) . . . 32
3.5 Matrix Algebra . . . 34
3.5.1 Standard c++ mathematical functions . . . 34
3.5.2 Detect all elements being non-zero . . . 34
3.5.3 Detect any element being non-zero . . . 35
3.5.4 Cholesky factorization . . . 35
3.5.5 Condition number . . . 35
3.5.6 Correlation coefficient . . . 36
3.5.7 Covariance . . . 36
3.5.8 Determinant . . . 36
3.5.9 Diagonal . . . 36
3.5.10 Eigenvalues . . . 37
3.5.11 Incomplete gamma . . . 38
3.5.12 Logorithm of gamma . . . 38
3.5.13 Identity . . . 39
3.5.14 Read data from a file . . . 39
3.5.15 Inverse . . . 39
3.5.16 Kronecker product . . . 39
3.5.17 Log determinant . . . 40
3.5.18 LU factorization . . . 40
3.5.19 QR factorization . . . 40
3.5.20 Largest element . . . 41
3.5.21 Mean of elements . . . 42
3.5.22 Smallest element . . . 42
3.5.23 modulo operation . . . 42
3.5.24 norm . . . 43
3.5.25 power . . . 43
3.5.26 rank . . . 43
3.5.27 Reshape elements . . . 43
3.5.28 Resize . . . 44
3.5.29 Rotation . . . 44

3.5.30 Randon number . . . 45
3.5.31 Save data into a file . . . 45
3.5.32 Select elements . . . 45
3.5.33 Round to integers . . . 45
3.5.34 Cast to integers . . . 45
3.5.35 solve . . . 46
3.5.36 sort . . . 46
3.5.37 Standard deviation . . . 46
3.5.38 Sum . . . 47
3.5.39 Sum of least-squares . . . 47
3.5.40 Singular value decomposition . . . 47
3.5.41 Sweep operation . . . 47
3.5.42 Transpose . . . 48
3.5.43 Trace . . . 48
3.5.44 Lower triangular . . . 48
3.5.45 Upper triangular . . . 49
3.5.46 Variance . . . 50
3.5.47 Vectorize vertically . . . 50
3.5.48 Vectorize a lower triangular . . . 50
3.5.49 Zero elements . . . 50
3.5.50 Unit elements . . . 50
3.6 Special Matrices . . . 51

4 Program Flow Control **52**
4.1 If–Endif and If–Else Statements . . . 52
4.2 For Statement . . . 54
4.3 While Statement . . . 55
4.4 Repeat–Until Statement . . . 55

5 File Stream **56**
5.1 Standard File Streams . . . 56
5.1.1 cout . . . 56
5.1.2 cin . . . 56
5.1.3 cerr . . . 56
5.2 User Specified File Stream . . . 57
5.2.1 Creation . . . 57
5.2.2 Example . . . 57
5.3 Member Function . . . 57
5.3.1 open . . . 57
5.3.2 reopen . . . 57
5.3.3 close . . . 57
5.3.4 set . . . 57
5.3.5 out . . . 57
5.3.6 in . . . 58
5.3.7 getline . . . 58
5.3.8 eof . . . 58

5.3.9 endl . . . 58
5.3.10 flush . . . 58
5.3.11 rewind . . . 58

6 Date and Time **59**
6.1 Digital Clock . . . 59
6.1.1 Time . . . 59
6.1.2 Localtime . . . 59
6.1.3 Gmtime . . . 60
6.1.4 Asctime . . . 60
6.1.5 Difftime . . . 60
6.1.6 Mktime . . . 60
6.1.7 Date . . . 60
6.1.8 Ctime . . . 61
6.1.9 Clock . . . 61
6.2 Western Calendar . . . 61
6.2.1 Leap year . . . 61

7 User-Defined Function **62**
7.1 Syntax . . . 62
7.2 Description . . . 62
7.3 Scope . . . 63
7.4 Recursion . . . 63
7.5 Examples . . . 63

8 Plotting **65**
8.1 Creation . . . 65
8.2 Member Function . . . 65
8.2.1 plot . . . 65
8.2.2 plot3D . . . 65
8.2.3 replot . . . 66
8.2.4 set . . . 66
8.2.5 save . . . 67
8.2.6 open . . . 67
8.2.7 close . . . 67
8.3 Examples . . . 67
8.3.1 Example 1 . . . 67
8.3.2 Example 2 . . . 68
8.3.3 Example 3 . . . 68
8.3.4 Example 4 . . . 69

9 Macro Packages **70**
9.1 Demo Package . . . 71
9.1.1 demo() function . . . 71
9.1.2 demo_src() function . . . 73
9.2 Prime Number Package . . . 74

9.2.1 prime(n) 74
9.2.2 prime_next(n) 74
9.2.3 prime_less(n) 74
9.2.4 examples 74
9.3 Special Matrix Package 75
9.3.1 hilb 75
9.3.2 invhilb 75
9.3.3 hankel 75
9.3.4 vander 76
9.3.5 hadamard 76
9.3.6 pascal 76
9.3.7 toeplitz 77
9.3.8 magic 77
9.4 Linear Programming Package 77

10 Mixed Model Analyses **78**
10.1 Best Linear Unbiased Prediction (BLUP) 78
10.1.1 Data 78
10.1.2 Model 79
10.1.3 Obtain BLUP using matrix algebra 79
10.1.4 Obtain BLUP using matvec higher level functions 81
10.1.5 Multi-trait and multi-model example 83
10.2 Linear Estimation 84
10.3 Linear Hypothesis Test 84
10.4 Least Squares Means (lsmeans) 85
10.5 Variance Components Estimation (VCE) 86

11 Segregation and Linkage Analyses **88**
11.1 Genotype Probability Computation 88
11.2 Genetic Mapping 89

12 Statistical Distributions **90**
12.1 Introduction 90
12.2 Continuous Distribution 91
12.2.1 Normal distribution 91
12.2.2 Uniform distribution 94
12.2.3 χ^2 distribution 95
12.2.4 t distribution 97
12.2.5 F distribution 99
12.2.6 Gamma distribution 101
12.2.7 Exponential distribution 102
12.2.8 Beta distribution 104
12.2.9 Log normal distribution 106
12.3 Discrete Distribution 107
12.3.1 Binomial distribution 107
12.3.2 Poisson distribution 109

12.3.3 Geometric distribution 110
12.3.4 Negative binomial distribution 112

13 Exception Handling **114**
13.1 Error Handling 114
13.2 Troubleshooting 114
13.2.1 Lack of memory 114
13.2.2 Intermediate files 114
13.2.3 Bugs 115
13.2.4 EPSILON 115
13.3 Limitation 115

Preface

The object-oriented problem solving and object-oriented programming represent a way of thinking and a methodology for computer programming that are quite different from the traditional approaches supported by structured programming languages. The powerful features of object-oriented programming support the concepts that make computer problem solving a more human-like activity and that increase the re-usability of software code.

C++ provides programmers and problem solvers object-oriented capability without loss of run-time or memory efficiency. In addition, C++ is available in almost every computer systems from PC to mainframe.

matvec, written in C++, was particularly designed for animal breeders. There are, however, lots of general operations and functions for matrix-vector. matvec is an object-oriented, interactive, and interpreted programming language. It provides a comprehensive computing environment where you can solve problems ranging from matrix-vector operation to variance components estimation. It is a powerful tool for teaching and research.

matvec is an easy to learn programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming.The matvecinterpreter and the extensive C++ library are freely available in source form.

This tutorial introduces the reader informally to the basic concepts and features of the matvec language.

Chapter 1

A Tour of matvec

1.1 Running matvec

1.1.1 Interactively

On most computer systems, matvec can be invoked on your terminal by entering the shell command `matvec`. Then matvec will start with an initial message similar to:

```
matvec v2.4.0, Copyright (C) 2000, Matvec team
matvec comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
For details see the COPYLEFT file coming with this software.
type ? for the on-line document
Hollow World!
```

and then a prompt `>` next to the cursor. Now matvec is ready for you to try something out. For instance, type `a=[1,2,3]`.

1.1.2 From a script

If you want to accomplish a little bit complicated job, then it is a better idea to write a script file using matvec language. Suppose your script file named `try.mv` contains the following matvec statements

```
A = [3,2,1; 4,6,5; 7,8,9];
A.det()
```

Then you can type `matvec try.mv` at your computer system prompt to get the determinant of matrix A. You can also redirect the output from matvec to a file using UNIX redirect mechanism. For instance, under C-shell, the command

```
matvec try.me >try.out&
```

will create a file try.out which contains standard output from matvec. The last character & tells the computer to run matvec in background.

1.1.3 From standard input

Try the following command at your computer system prompt:

```
cat try.mv |matvec
```

1.2 Special Attention

There are a few of things to which you have to pay a special attention:

- anything after `#` or `//` to the end of a line is treated as comments, thus ignored by matvec interpreter. For multi-line comments, use `/* ... */`
- an empty line is ignored by matvec interpreter.
- backslash `\` at the end of a line is a continuation operator.
- the best way to stop matvec is command `quit`. In an emergency, you can use Cntl-c to terminate matvec.
- If a statement (expression) is ended with a semicolon (;), then returned value from this statement is not displayed on the screen. In other word, semicolon (;) suppresses printing.
- if `$` is the first character in a line, then anything after `$` is interpreted as the shell command and will be sent to the shell to execute. Almost all shell commands can be accessed and executed within matvec through this way. For instance, `$ls` in UNIX displays the contents of the current working directory.

1.3 Start-up File

Whenever you starts matvec, it tries to read the script file $HOME/.matvecrc where $HOME is your home directory. This optional file is referred to as a start-up file where you can put any matvec statements to initialize some global variables and parameters. Here is an example of $HOME/.matvecrc:

```
PAGER = "less";
this.parameter("OUTPUT_FLAG",1);
this.parameter("OUTPUT_PRECISION",4);
this.parameter("OUTPUT_LINE_WIDTH", 132);
```

1.4 Creating Objects

Example 1:

```
> A = 4
       4
```

It simply creates a scalar object named `A`.

Example 2:

```
> a=1:4

        i = 1         i = 2         i = 3         i = 4
            1             2             3             4
```

It simply creates a vector object named `a`. There are four elements 1, 2, 3, and 4.

Example 3:

```
> A = [1,2,3;4,5,6]

                      Col 1         Col 2         Col 3
        Row 1             1             2             3
        Row 2             4             5             6
> A.mean()

        i = 1         i = 2         i = 3
          2.5           3.5           4.5
```

The first statement creates a matrix object named `A` with two rows and three columns. The second statement returns a vector containing the means of each column of `A`.

Example 4:

```
> s = "Hello, My name is Tianlin Wang"
Hello, My name is Tianlin Wang
```

It simply creates a string object named `s`.

1.5 Getting Online Help

matvec provides the online help. Type the question mark `?` at the prompt, it shows the location for the on-line help.

```
> ?max
              Builtin Friend and Member Function: max
------------------------------------------------------------
Syntax
        max(A)
        max(A,B)
        A.max()
        A.max(B)
```

```
Description
  - If A is a scalar, A.max() returns itself.

  - If A is a vector, A.max() returns an element with the
    largest value.

  - If A is a matrix, A.max() returns a vector whose each
    element is the largest element of each column of A.
    If A is a one-row or one-column matrix, then the returned
    value would be a scalar.

  - If both A and B are scalars A.max(B) returns the largest
    one between A and B.

  - If both A and B are vectors (one of them could be a
    scalar), then  A.max(B) returns a vector of the same
    size as A or B with resulting element(i) being the
    largest element among A(i) and B(i).

  - If both A and B are matrices (one of then could be a
    scalar), then  A.max(B) returns a matrix of the same
    size as A or B with resulting element(i,j) being the
    largest element among A(i,j) and B(i,j).

  - max(A) is the same as A.max().
  - max(A,B) is the same as A.max(B).

See Also
  min, sum, sumsq, mean, variance, Scalar, Vector, Matrix.
-------------------------------------------------------------
```

You can obtain the most powerful online help using `xhelp`. It will bring you the hypertext version of matvec online help resources. This requires a hypertext browser such as netscape or Mosaic installed on your computer system.

1.6 Scalar Arithmetic

matvec is a super calculator. Almost all of arithmetic operations are available. The symbols `+`, `-`, `*`, `/`, `^` represent addition, subtraction, multiplication, division, and power, respectively. For examples,

```
> (2+3)*2^3-8/4
38
> 0/0
NaN
```

Notice that 1) matvec handles the precedence of operations in a usual way, and 2) there is no difference between integer and real number. In fact they are internally stored in memory as a double precision number.

1.7 Matrix-Vector Arithmetic

The simplest way to create a matrix is to type it in at the matvec prompt:

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

                    Col 1        Col 2        Col 3
        Row 1           1            2            3
        Row 2           4            5            6
        Row 3           7            8            9
```

Note that the symbol pair `[]` is one of matvec special operators. matvec is ready to create a matrix object whenever it recognizes this pair. Everything inside the `[]` is the contents of a matrix. Each element is separated with comma(,) and each row is separated with semicolon (;). If there is nothing inside of `[]`, then a null (empty) matrix object is created. Any matrix specific operations and functions can be applied to an empty matrix, though most of them may not make sense.

matvec provides a great number of functions for matrix computation. For example,

```
# let's first create a matrix object named A
> A = [3,2,1; 2,5,3; 1,3,4]

                    Col 1        Col 2        Col 3
        Row 1           3            2            1
        Row 2           2            5            3
        Row 3           1            3            4
# find the largest elements for each columns
> A.max()

        i = 1        i = 2        i = 3
            3            5            4
# find the largest element in matrix A
> A.max().max()
        5
# how about the inverse of A, surely, matrix A is non-singular
> A.inv()
                    Col 1        Col 2        Col 3
        Row 1    0.458333    -0.208333    0.0416667
        Row 2   -0.208333     0.458333    -0.291667
        Row 3   0.0416667    -0.291667     0.458333
```

```
# what about a generalized inverse of matrix A
> A.sweep()
                   Col 1        Col 2        Col 3
        Row 1     0.458333    -0.208333    0.0416667
        Row 2    -0.208333     0.458333    -0.291667
        Row 3    0.0416667    -0.291667     0.458333

# the eigenvalues from MATVEC is not pre-sorted
> A.eigen()
        i = 1        i = 2        i = 3
      2.38677      1.19440      8.41883
```

The singular value decomposition of an m-by-n matrix A is a decomposition

$$A = U^*W^*V'$$

where U is m-by-n with U'*U = I, W is an n-by-n diagonal matrix and V is an n-by-n orthogonal matrix.

Both A.svd() and A.svd(U,V) returns a vector containing diagonals in W, but the latter has U and V which must be declared as matrix objects before calling.

```
> A = [3,2,1; 2,5,3; 1,3,4];
> u=[]; v=[];
> A.svd(u,v)
        i = 1        i = 2        i = 3
      8.41883      2.38677      1.19440

> u
  u =

                   Col 1        Col 2        Col 3
        Row 1    -0.374359     0.815583    -0.441225
        Row 2    -0.725619    0.0386051     0.687013
        Row 3    -0.577350    -0.577350    -0.577350

> v
  v =

                   Col 1        Col 2        Col 3
        Row 1    -0.374359     0.815583    -0.441225
        Row 2    -0.725619    0.0386051     0.687013
        Row 3    -0.577350    -0.577350    -0.577350
```

1.8 Variables, Expressions, and Statements

An identifier consists of letters, underscores, and digits with a leading letter or underscore. Identifiers are used to name keywords, functions, and variables. matvec itself like other language has its own reserved keywords. They are listed here: *and, andif, break, continue, else, for, function, if, for, input, or, protected, quit, return, repeat, until while.*

When you name a variable, consider the following:

- make the name long enough to mean something to you. There is no limit on the length of variable name in matvec.
- make the name easy to read by using the underscore character to separate parts of the name
- never use any of matvec predefined names.

matvec is case-sensitive in the names of variables, functions, and statements. For example, *Matrix* and *matrix* is not the same, the former is a function name, and latter is a variable name.

A single number can be an integer, a decimal fraction, or a number in scientific (exponential) notation. Note that a single number is represented within matvec in double-precision floating point format. Here are examples of single numbers:

3.14
+3.14
314e-2
314E-2
.314e+1
-(-3.14)

The basic structural element of the matvec language is the expression:

- a single number such as 3.14 is an expression,
- a matrix (or vector) is an expression
- an assignment such as *variable = expression* is an expression
- most of operations and functions result in expressions

matvec statements includes expression, assignment, *if-endif*, *if-else*, *while*, *repeat–until*, *for*, *return*, *break*, *continue*, etc. A statement is normally terminated with the carriage return. A statement, however, can be continued to the next line with continuation operator \ followed by a carriage return. On the other hand, several statements can be placed on a single line if separated by either comma or semicolons. If a statement is ended by a semicolon, the printing is suppressed. Most of intermediate expression-statements are not expected to print. Thus, statements should always be ended with semicolons except those you want printing.

1.9 Inbreeding Coefficient Computation

Suppose we have a pedigree

```
A1  .  .
A2  .  .
A3 A1 A2
A4  . A2
A5 A3 A4
```

which is stored in a ASCII file called `try.ped`. matvec provides a very friend interface to compute the inbreeding coefficient for each individual or the average inbreeding of a population or the maximum inbreeding coefficient in a population:

```
/home/tianlin/matvec
  MATVEC v0.10, Mar. 12, 1995, University of Illinois
  type ? (or help, xhelp, xman) for on-line help

# the 1st step is to create a Pedigree object, and input ASCII file
> p = Pedigree();
> p.input("try.ped");

# now, compute the inbreeding coefficient for each individual
> p.inbcoef()

        i = 1         i = 2         i = 3         i = 4         i = 5
            0             0             0             0         0.125
# and the average of inbreeding coefficients for the population
> p.inbcoef().mean()
        0.025

# finally the maximum inbreeding coefficient in the population
> p.inbcoef().max()
        0.125
```

You may immediately complain about the size of pedigree that matvec are dealing with. Try a pedigree with a half million individuals, and compute the same things as you did for five-individual pedigree. Don't be surprised if matvec gives your answer within minutes.

Chapter 2

Working on Objects

matvec is object-oriented. Almost every piece in matvec system is an object. All objects in matvec can be classified into nine classes: *Scalar, Vector, Matrix, SparseMatrix, String, Data, Model, FileStream*, and *StatDist*.

It should always keep in mind that you're using an object-oriented language. Thus, to play with matvec you have to create your own objects at the very beginning.

2.1 Special Object: this

A very special object built in matvec is *this*. It is the object of the current working session. All of the parameters, properties pertaining to the current working session can be changed through this special object. For C++ programmer, keyword *this* is an old friend

If you simply type this, it will display the names of the user-defined variables and functions.

this.clear(a,b,...) deletes the named objects from the current matvec session immediately. The memory associated with the objects are freed. this.clear() deletes all objects (but not the builtins') and all intermediate temporary files (but not ones created from the other matvec sessions) from the matvec trash can. The return value is the number of objects that have been successfully deleted.

this.parameter() returns a string of all possible parameter names.

For example,

```
> this.parameter()
  ans = EPSILON,OUTPUT_FLAG,OUTPUT_PRECISION,PI_VALUE,RAND_SEED,WARNING
```

this.parameter(`"OUTPUT_PRECISION"`, k) sets the output precision for the current matvec working session to k. The default value is 6. this.parameter(`"OUTPUT_PRECISION"`) returns the output precision in the current matvec working session. Note that "precision" is defined to be the number of significant digits in output.

this.parameter("OUTPUT_FLAG", 1) sets the output flag for the current matvec working session to 1. The default value is 0. this.parameter("OUTPUT_FLAG") returns the output flag in the current matvec working session. Note that "flag" is used to control the output style for the matrix and vector.

For example,

```
> A=[1,2;3,4]
 A =
                      Col 1        Col 2
         Row 1            1            2
         Row 2            3            4

> this.parameter("OUTPUT_FLAG", 1);
> A=[1,2;3,4]
  A =
          1            2
          3            4
> this.parameter("OUTPUT_FLAG", 2);
> Vector(1,2,3)
  ans =
          1
          2
          3
```

this.parameter("WARNING", k) sets the warning flag for the current working session to k, which can be either 1 or 0. The default value is 1. this.parameter("WARNING") returns the value of the warning flag in the current working session.

this.parameter("EPSILON", x) sets the epsilon value for the current matvec working session to x. The default value is 1.0e-14. matvec users are strongly discouraged to change the default epsilon value. this.parameter("EPSILON") returns the epsilon value in the current matvec working session.

this.parameter("OUTPUT_LINE_WIDTH", k) sets the line width of output for the current matvec working session to k. The default value is 80. this.parameter("OUTPUT_LINE_WIDTH") returns the line width of output in the current matvec working session.

this.parameter("INPUT_LINE_WIDTH", k) sets the line (record) width for input in the current matvec working session to k. The default value is 1024. this.parameter("INPUT_LINE_WIDTH") returns the line (record) width for input in the current matvec working session.

this.parameter("MAX_ERRORS", k) sets the maxmum number of errors allowed for the current matvec working session to k. The default value 15. this.parameter("MAX_ERRORS") returns the maxmum number of errors allowed for the current matvec working session.

For example,

```
> this.parameter("OUTPUT_PRECISION")
        6
```

```
> 2.0*asin(1.0)
        3.14159
> this.parameter("OUTPUT_PRECISION",15)
> 2.0*asin(1.0)
        3.14159265358979
```

2.2 Scalar Object

A scalar object in matvec is defined as a floating point number (double-precision). For example 1.5, .3, 3, -3e-2, etc.

The simplest way to create an object of class *Scalar* is to type in a floating point number at matvec prompt. The standard arithmetic operators such as `+, -, *, /, ^` are working for scalars. The standard mathematical functions are designed as friend as well as member functions for scalars using the C++ math library (except *max*, *min*, *gammaln*, and *gammainc*). They are listed below:

abs	acos	asin	atan	atan2	ceil	cos
cosh	floor	log	log10	max	min	mod
sin	sinh	sqrt	tan	tanh	gammaln	gammainc

For example,

```
> A = 10;
> A.sin().sin()
     -0.517581
```

Details about the ranges and error conditions for these functions can be found in your math library reference manuals.

A scalar x can be converted to one-element matrix or vector by using [x] and x:x. For example,

```
> x=2.5
        2.5
> [x]

                    Col 1
        Row 1         2.5
> x:x

        i = 1
          2.5
```

2.3 Vector Object

A vector object in matvec is defined as a one-dimensional array with each element being floating point number of double precision. It can be created using the colon operator and be resized at any time for any size using `A.resize(n)` where `n` is the size you want. For example,

```
> A = 1:4

        i = 1        i = 2        i = 3       i = 4
            1            2            3           4

> B = 1:4:2
        i = 1        i = 2
            1            3
```

Another method to create a vector object is to use one of object-creating function `Vector(...)` as shown below:

```
> C = Vector(-5, 5, B)
        i = 1        i = 2        i = 3        i = 4
           -5            5            1            3
```

The i'th element of vector A can be accessed by A(i). Note that vector indexing starts from one rather than zero. The usual algebraic operators such as `+, -, *, /, .+, .-, .* ./,+=, -=, *=, /=` are available. If both A and B are vectors, then `A*B` returns an inner product of vectors A and B, whereas `A.*B` returns a vector with each element being product of corresponding elements in A and B. The element-by-element division can be done by either A/B or A./B.

The relational operators are `==, !=, <, >, <=, >=`. The logical operators are `||(OR), &&(AND)` and `.||, .&&`.

There are a lot of member functions for vectors such as **`max, min, mean, input, mat, diag, sin`**, etc.

A vector (v) can be easily converted to any size of matrix by using v.mat(m,n) as long as there are enough elements to fill.

2.4 Matrix Object

See details in Chapter 3.

2.5 String Object

A string object in matvec is defined as a sequence of characters. It can be created by using double-quotes operator `"`. For example,

```
s = "Hello World";
```

The operators and member functions for string objects are very limited. Indexing and concatenation works as usual. For example

```
> s = "Hello World";
> s.size()
        11
> t = "Greetings From MATVEC";
> s + t
        Hello WorldGreetings From MATVEC
> s + ", " + t
        Hello World, Greetings From MATVEC
> s([1,7])
        HW
```

Some characters are so special that they cannot be included literally in a string object. Instead, they are represented with escape sequences. An escape sequence is a two-character sequence beginning with a backslash (\) followed by a special character. The following is a list of escape sequences implemented in matvec:

escape sequences	meanings
`\\`	a literal backslash (\)
`\"`	a literal double-quote (")
`\n`	a newline
`\t`	a horizontal tab
`\f`	a formfeed
`\b`	a backspace
`\r`	a carriage return
`\a`	an alert
`\v`	a vertical tab

For examples,

```
> "Matvec"
        Matvec
> "\"Mat\tVec\""
        "Mat    Vec"
```

2.6 Pedigree Object

A Pedigree object in matvec is a object which holds a pedigree.

For example,

```
> ped_file_name = RawData {
   A1         .       .
   A2         .       .
   A3        A1      A2
   A4         .      A2
   A5        A4      A3
  }
        .matvec.798672017.0
```

```
> # note that .matvec.798672017.0 is the file-name for
> #          the temporary file
> P = Pedigree();
> P.input(file_name);
> P.inbcoef().max()
        0.125
```

The first three columns in a pedigree file must be "idividual father mother". If not, then you have to specify the coloumn format such as "skip idividual skip father mother". If a line starts with "//", then this line is treaded as a comment line. A empty line in a pedoigree file is ignored.

2.6.1 sample

P.sample(n,ng,s0,d0,imrate,parent,nopo,nofsib,sexratio) generates a arbitrary pedigree of size n with ng generations starting from s0 base sires and d0 base dams. The argument imrate is the immigration rate; argument parent is a switch with 1 meaning no missing parents are allowed; argument nopo is a switch with 1 meaning no mating between parent and its progeny is allowed; argument nofsib is a switch with 1 meaning no mating between full-sibs is allowed; argument sexratio is the sex ratio of new born progeny (male:female). All arguments except the first one are optional with default values: ng=1, s0=1, d0=1, imrate=0.1, parent=1, nopo=1, nofsib=1, sexratio=0.5.

2.6.2 input

P.input(`"<file_name>"`,`"<column_names>"`,`"<pedtype>"`) inputs pedigree from the ASCII file `"<file_name>"`, and `"<column_names>"` must contain the three key words in any order: individual mother father. If `"<column_names>"` is omitted, then matvec uses the default: "individual mother father". The last argument `"<pedtype>"` is optional. The possible pedtypes available are `"raw"`,`"standard"`, `"group"`.

For examples, P.input("skip individual skip father skip skip mother") results in the format "individual father mother". You do not have to worry about the trailing columns. If it is the pedgree file was saved in a previous session using P.save("filename"), then you must specify colfmt="dummy individual father mother sex".

2.6.3 inbcoef

P.inbcoef() returns a vector of inbreeding coefficients for all members in the pedigree.

2.6.4 logdet

P.logdet() returns the natural logarithm of the additive numerator relationship matrix.

2.6.5 rela

P.rela() returns the additive numerator relationship matrix.

2.6.6 reld

P.reld() returns the dominant relationship matrix, assuming no no inbred individuals.

2.6.7 inv

P.inv() returns the inverse of the additive relationship matrix.

2.6.8 size

P.size() returns the total number of individuals in pedigree P.

2.6.9 nbase

The number of base members in pedigree P.

2.6.10 save

P.save(`"<fname>"`) saves a ASCII copy of pedigree

2.7 Data Object

A Data object in matvec is a object which can handle a dataset. Data class in still under development. A data object can be created by D = Data(). Then, use its member function *input* to input data from disk into the object.

2.8 Model Object

A model object can be created by M = Model(). Then, use its member function *equation* to specify the linear model equations. Other characteristics for the model such as random effect, covariate, etc. can be specified through appropriate member functions.

For example,

```
> M = Model();
> M.equation("y = herd animal");
```

2.9 StatDist Object

See details in Chapter 12.

Chapter 3

Matrix Object

Matrix computation is a complex task. matvec provides a comprehensive set of functions and tools, which are powerful and easy to learn, to perform the massive matrix manipulation and sophisticated matrix computation. In particular, matvec can be used to answer *what-if* questions raised by your research colleagues or by yourself.

3.1 Creation

There are several ways to create a matrix object. The three common matrix creation methods are listed below:

3.1.1 Using operators [and]

You can type in all elements row-by-row directly from keyboard with the operator [and]. That is starting with [, separating each elements with comma (,), separating each row with semicolon (;), and ending with]. For example,

```
A = [1.0, 2.0, -2.0;
     3.0, 2.0, sqrt(9.0)];
```

3.1.2 Using object-creating functions

You can use the follwoing functions to create a matrix.

Matrix(m,n)

It creates a matrix object of m rows and n columns. Each element is initialized with 0.0.

```
> Matrix(2,3)
                    Col 1         Col 2         Col 3
       Row 1            0             0             0
```

```
       Row 2               0              0              0
```

ones(m,n)

It creates a matrix object of m rows and n columns. Each element is initialized with 1.0.

```
> ones(2,3)

                   Col 1          Col 2          Col 3
        Row 1          1              1              1
        Row 2          1              1              1
```

zeros(m,n)

It creates a matrix object of m rows and n columns. Each element is initialized with 0.0.

```
> zeros(2,3)

                   Col 1          Col 2          Col 3
        Row 1          0              0              0
        Row 2          0              0              0
```

rand(m,n)

It generates a matrix object of m rows and n columns. Each element is sampled from Uniform(0.0,1.0).

```
rand(2,3)
                   Col 1          Col 2          Col 3
        Row 1     0.865423       0.122624       0.39272
        Row 2     0.199465       0.912348       0.192371
```

identity(m,n)

It creates a matrix object of m rows and n columns. The diaginal elements are 1.0 and off-diaginals are 0.0.

```
> identity(2,3)
                  Col 1          Col 2          Col 3
       Row 1          1              0              0
       Row 2          0              1              0
```

3.2 Operator

3.2.1 Arithmetic operators

-A unary minus (negation)

+A unary plus

!A negation

A' matrix transpose

++A prefix-increment

--A prefix-decrement

A++ post-increment

A-- post-decrement

A = B assignment

A + B does element-by-element addition

A - B does element-by-element subtraction

A * B does matrix multiplication

A / B does matrix right division

A @ B Kronecker tensor product of A and B

A ˆ B power

A .+ B does element-by-element addition

A .- B does element-by-element subtraction

A .* B does element-by-element multiplication

A ./ B does element-by-element division

A .ˆ B does element-by-element power

A += B does short hand addition assignment

A -= B does short hand subtraction assignment

A *= B does short hand multiplication assignment

A /= B does short hand right division assignment

A .+= B does short hand element-by-element addition assignment

A .-= B does short hand element-by-element subtraction assignment

A .*= B does short hand element-by-element multiplication assignment

A ./= B does short hand element-by-element division assignment

3.2.2 Relational operators

There are six relational operators, they operate between two objects.

`==` equal to

`!=` not equal to

`<` less than

`>` greater than

`<=` less then or equal to

`>=` greater than or equal to

These six operators are based on element-by-element comparison. It returns a matrix with resulting element(i,j) = 1.0 if the relationship is true, otherwise resulting element(i,j) = 0.0

3.2.3 Logical operators

There are two sets of logical operators: (`&&`, `||`) and (`.&&`, `.||`).

The `&&` is a short-circuit logical operator AND. A `&&` B returns 1 if both A and B are true (B is evaluated only after A is true), otherwise return 0. If A (or B) is a matrix, then A.all().all() is implicitly applied. If A (or B) is a vector, then A.all() is implicitly applied. Any other types of objects will be determined false.

The `||` is a short-circuit logical operator OR. A `||` B returns 1 if either A and B are true (B is evaluated only after A is false), otherwise return 0. If A (or B) is a matrix, then A.all().all() is implicitly applied. If A (or B) is a vector, then A.all() is implicitly applied. Any other types of objects will be determined false.

The `.&&` and `.||` are the same as `&&` and `||` except that 1) the former is performed element-by-element, 2) the former is non-short-circuit, thus both operands will be evaluated.

For readability, matvec creates synonyms *AND* and *OR* for `&&` and `||`, respectively.

3.3 Manipulation

Matrix manipulation is a tricky work. Basically, there are three kinds of manipulations: 1) accessing elements and sub-matrices, 2) adjoining and stacking, 3) selecting elements which satisfy whatever condition you specify. They are described below in detail.

3.3.1 Accessing to elements and sub-matrix

- accessing an element
  A(i,j) accesses (i,j)'th element with boundary checking
- accessing a row
  A(i,*) gets a copy of the i'th row of matrix A.
- accessing a column
  A(*,j) gets a copy of the j'th column of matrix A.
- accessing sub-matrix
  A(i1:i2, j1:j2) returns a sub-matrix of A with row i1 through i2 and column j1 through j2. A([i1,i2,i3], [j1,j2,j3]) returns a sub-matrix of A with rows i1, i2, and i3 and columns j1, j2, and j3.

For example,

```
> A = [1,2,3;4,5,6]

                     Col 1        Col 2        Col 3
      Row 1              1            2            3
      Row 2              4            5            6
> A(2,3)           # returns element at row 2 and column 3
      6
> A(*,2)           # returns the second column

                     Col 1
      Row 1              2
      Row 2              5
> A(1,*)           # returns the first row

                     Col 1        Col 2        Col 3
      Row 1              1            2            3
> A(2,1:3)         # returns the second row

                     Col 1        Col 2        Col 3
      Row 1              4            5            6
> A(*,[3,2,1])     # the columns are reversed

                     Col 1        Col 2        Col 3
      Row 1              3            2            1
      Row 2              6            5            4
```

3.3.2 Horizontal and vertical concatenation

Adjoining or stacking matrices in matvec are quit convenient. Suppose both A and B are matrix objects. Then `[A, B]` creates a new matrix object resulting from adjoining of A and B. Whereas `[A; B]` creates a new matrix object resulting from stacking of A and B. In fact A and B can be vector or scalar as long as they have appropriate dimensions for adjoining and stacking. For example

```
> A=[1,2;3,4]

              Col 1      Col 2
    Row 1         1          2
    Row 2         3          4

> [A,         [3; 7];
   A.sum(),  A.sum().sum()]

              Col 1      Col 2      Col 3
    Row 1         1          2          3
    Row 2         3          4          7
    Row 3         4          6         10

> ones(3,1),identity(3,3)]

              Col 1      Col 2      Col 3      Col 4
    Row 1         1          1          0          0
    Row 2         1          0          1          0
    Row 3         1          0          0          1
```

3.3.3 Selecting elements

Sometimes, you may want to select elements which satisfy certain condition. Here is an example:

```
> A = [1,2,3;4,5,6]

                    Col 1        Col 2        Col 3
        Row 1           1            2            3
        Row 2           4            5            6
> A.select(A>2)

        i = 1        i = 2        i = 3        i = 4
            3            4            5            6
```

3.3.4 Reshaping and Rotating

- A.reshape(m,n)

- A.rop90(k)
  Return a copy of A with the elements rotated counterclockwise in 90-degree increments.

The second argument is optional, and specifies how many 90-degree rotations are to be applied (the default value is 1). Negative values of k rotate the matrix in a clockwise direction. For example,

```
> A=[1,2;3,4];
> A.rot90(-1)
                    Col 1         Col 2
       Row 1            3             1
       Row 2            4             2
```

rotates the matrix clockwise by 90 degrees. The following are all equivalent statements:

```
A.rot90(-1)
A.rot90(3)
A.rot90(7)
```

- A.t()
  Return the transpose of Matrix A. This function is equivalent to `A'`

- A.fliplr()
  Return a copy of Matrix A after being flipped left-to-right about a vertical axis. For example:

```
> A=[1,2;3,4];
> A.fliplr()
                    Col 1         Col 2
       Row 1            2             1
       Row 2            4             3
```

- A.flipud()
  Return a copy of Matrix A after being flipped upside-down about a horizontal axis. For example:

```
> A=[1,2;3,4];
> A.flipud()
                    Col 1         Col 2
       Row 1            3             4
       Row 2            1             2
```

- A.tril(k) and A.triu(k)
  Return a new matrix formed by extracting the lower (tril) or upper (triu) triangular part of the matrix A, and setting all other elements to zero.

3.4 Matrix Statistics

3.4.1 Descriptive statistics

- A.max()
  Return a vector with the maximum value of each column. `A.max().max()` returns the largest element of matrix A. If A is a one-row or one-column matrix, then the returned value would be a scalar. If both A and B are matrices (one of them could be a scalar), then A.max(B) returns a matrix of the same size as A or B with resulting element(i,j) being the largest element among A(i,j) and B(i,j).

- A.min()
  Return a vector with the minimum value of each column. `A.min().min()` returns the smallest element of matrix A. If A is a one-row or one-column matrix, then the returned value would be a scalar. If both A and B are matrices (one of them could be a scalar), then A.min(B) returns a matrix of the same size as A or B with resulting element(i,j) being the smallest element among A(i,j) and B(i,j).

- A.sum()
  Return a vector with the sum of each column.

- A.mean()
  Return a vector with the mean of each column.

- A.variance()

- A.cov()
  Return a matrix with element (i, j) being the covariance between column i and column j.

- A.corrcoef()
  Return a matrix with element (i, j) being the correlation between column i and column j.

For examples:

```
> A=[5,3,2;3,3,0;2,,0,2];
> A.min()
           i = 1        i = 2        i = 3
               2            0            0

> A.max()
           i = 1        i = 2        i = 3
               5            3            2

> A.sum()
```

```
          i = 1        i = 2        i = 3
             10            6            4

> A.mean()
          i = 1        i = 2        i = 3
        3.33333            2      1.33333

> A.variance()
          i = 1        i = 2        i = 3
        2.33333            3      1.33333

> A.covariance()
                  Col 1        Col 2        Col 3
      Row 1     2.33333            2     0.333333
      Row 2           2            3           -1
      Row 3    0.333333           -1      1.33333

> A.corrcoef()
                  Col 1        Col 2        Col 3
      Row 1           1     0.534522     0.327327
      Row 2    0.534522            1         -0.5
      Row 3    0.327327         -0.5            1

> A=rand(2,2)

                   Col 1        Col 2
       Row 1   0.0100414     0.578454
       Row 2    0.973198     0.792847
> B=rand(2,2)

                   Col 1        Col 2
       Row 1    0.619559     0.927987
       Row 2     0.89682     0.990276
> A.min(B)

                   Col 1        Col 2
       Row 1   0.0100414     0.578454
       Row 2     0.89682     0.792847
```

3.4.2 Best linear unbiased prediction (BLUP)

A single trait animal model is assumed:

$$y_i = \mu + h_i + a_i + e_i$$

First of all, we write down a script using matrix tools and functions provided by matvec. You can use any of you favorable ASCII editor, and save the file as `blup.mv` in your current working directory.

```
A=[1,   0,   0.5, 0,    0.25;
   0,   1,   0.5, 0.5,  0.25;
   0.5, 0.5, 1,   0.25, 0.5;
   0,   0.5, 0.25,1,    0.125;
   0.25,0.25,0.5, 0.125,1];
Ai = A.inv();

X = [1, 1, 0;
     1, 1, 0;
     1, 1, 0;
     1, 0, 1;
     1, 0, 1;
     1, 0, 1];

Z = [0, 1, 0, 0, 0;
     0, 1, 0, 0, 0;
     0, 0, 1, 0, 0;
     0, 0, 0, 1, 0;
     0, 0, 0, 0, 1;
     0, 0, 0, 0, 1];
y = [6;
     8;
     8;
     7;
     9;
    12];

r = 2/1;

MMEq =[X'*X,     X'*Z;
       Z'*X,     Z'*Z + A.inv()*r];

rhs = [X'*y;
       Z'*y];

blup = MMEq.sweep()*rhs
```

Now, at the matvec prompt, simply type

```
> input blup.mv;
  blup=
                    Col 1
       Row 1         9.11
```

```
Row 2        -1.69
Row 3            0
Row 4         0.26
Row 5        -0.26
Row 6         0.26
Row 7        -0.67
Row 8         0.67
```

3.5 Matrix Algebra

3.5.1 Standard c++ mathematical functions

matvecsupports the following 21 standard c++ mathematical functions.

sin	cos	tan	asin	acos
atan	atan2	sinh	cosh	tanh
ceil	floor	exp	abs	log
log10	power	sqrt	mod	erf
erfc				

These functions apply to each element of a matrix.

```
> A = [8,1,6;3,5,7;4,9,2]
  A =
                  Col 1        Col 2        Col 3
      Row 1           8            1            6
      Row 2           3            5            7
      Row 3           4            9            2

> A.sqrt()
  .sqrt =
                  Col 1        Col 2        Col 3
      Row 1     2.82843            1      2.44949
      Row 2     1.73205      2.23607      2.64575
      Row 3           2            3      1.41421
```

3.5.2 Detect all elements being non-zero

A.all() operates over the columns of A, returning a vector of 1's and 0's. If A is a one-row or one-column matrix, then the returned value would be a scalar. For example,

```
> A=[1,0,2;3,4,5]

                  Col 1        Col 2        Col 3
       Row 1          1            0            2
       Row 2          3            4            5
> A.all()
```

```
        i = 1        i = 2        i = 3
            1            0            1
```

3.5.3 Detect any element being non-zero

A.any() operates over the columns of A, returning a vector of 1's and 0's. If A is a one-row or one-column matrix, then the returned value would be a scalar. For example,

```
> A=[1,0,2;3,4,5]

                    Col 1        Col 2        Col 3
        Row 1           1            0            2
        Row 2           3            4            5
> A.any()

        i = 1        i = 2        i = 3
            1            1            1
```

3.5.4 Cholesky factorization

A.chol() returns Cholesky factorization for a real symmetric positive (semi)-definite matrix A, The object A remains intact. This Cholesky decomposition method takes only non-singular part with maximum rank of A into account, everything else are set to zeros. If A is positive definite, then A.chol() returns an lower triangular matrix L so that L*L' = A. For example,

```
> A=[5,3,2;3,3,0;2,0,2]

                    Col 1        Col 2        Col 3
        Row 1           5            3            2
        Row 2           3            3            0
        Row 3           2            0            2
> A.chol()

                    Col 1        Col 2        Col 3
        Row 1     2.23607            0            0
        Row 2     1.34164      1.09545            0
        Row 3    0.894427     -1.09545            0
```

3.5.5 Condition number

A.cond() returns the condition number in norm2, which is the ratio of the largest singular value of A to the smallest. Object A remains intact.

3.5.6 Correlation coefficient

If each row of A and B is an observation and each column is a variable, A.corrcoef(B) returns a matrix whose the (i,j)'th element is the correlation coefficient between the i'th variable in A and the j'th variable in B. A.corrcoef() is identical to A.corrcoef(A), returning a correlation coefficient matrix.

This member function is a user-defined function, it's loaded automatically into memory whenever you launch matvec.

For example,

```
> A = [3,2,1;4,5,6;9,8,7];
> A.corrcoef()

                Col 1       Col 2       Col 3
        Row 1       1    0.933257    0.741935
        Row 2  0.933257        1    0.933257
        Row 3  0.741935  0.933257          1
```

3.5.7 Covariance

A.cov(B) returns a matrix whose element (i,j) is the covariance between column i of A(*,i) and column j of B(*,j).

For example,

```
> A = [1,2; 3,4; 5,6];
> B = [6,5; 2,3;8,6];
> A.cov(B)

                Col 1       Col 2
        Row 1       2           1
        Row 2       1           1
```

3.5.8 Determinant

A.det() returns the determinant of A based on a LU factorization. Object A remains intact.

3.5.9 Diagonal

A.diag(k), where A is a vector, returns a square matrix of order N+abs(k) with the elements of A on the k-th diagonal, where N is the size of vector A. k=0 (default) is the main diagonal, k > 0 is above the main diagonal and k < 0 is below the main diagonal.

A.diag(k), where A is a matrix, returns a vector formed from the elements of the k-th diagonal of A.

A.diag(B,C,D,...), where A,B,C,D,.., are all matrices, returns a block-diagonal matrix with A, B, C, ... on the diagonal

A.diag().diag() returns a diagonal matrix.

For example,

```
> c = [1,2;3,4]

                    Col 1        Col 2
       Row 1            1            2
       Row 2            3            4
> c.diag()

       i = 1        i = 2
           1            4
> c.diag().diag()

                    Col 1        Col 2
       Row 1            1            0
       Row 2            0            4
> c.diag([5])

                    Col 1        Col 2        Col 3
       Row 1            1            2            0
       Row 2            3            4            0
       Row 3            0            0            5
```

3.5.10 Eigenvalues

A.eigen() returns a vector containing eigenvalues , and matrix A remain intact.

A.eigen(U) returns the same as A.eigen(), but with the eigenvectors brought out by matrix U so that A*U = U*D where D is a diagonal matrix with D(i,i)'s being the eigenvalues.

There are two basic algorithms: Jacoba and Householder, the latter is used in this package.

Note that if matrix A is non-symmetric, its eigenvalues/eigenvectors could be complex numbers, which are all ignored with certain warnings in matvec.

For example,

```
> A = [3,2,1; 2,5,3; 1,3,4];
> A.eigen()

       i = 1        i = 2        i = 3
     2.38677       1.1944      8.41883
```

3.5.11 Incomplete gamma

A.gammainc() returns the incomplete gamma function value of A. The incomplete gamma function is defined as

$$\text{gammainc}(x,a) = \frac{\int_0^x t^{(a-1)} e^{-t}\, dt}{\Gamma(a)} (a > 0)$$

3.5.12 Logorithm of gamma

A.gammaln() returns the log of the gamma function value of A. The gamma function is defined as

$$\Gamma(x) = \int_0^\infty t^{(x-1)} e^{-t}\, dt$$

with $\Gamma(1) = 1, \Gamma(1/2) = \sqrt{(\pi)}$, and $\Gamma(x+1) = x\Gamma(x)$.

subsectionMoore-Penrose Pseudoinverse A.pinv(mc) returns a matrix G, a generalized (or pseudo) inverse of A. The matrix G has the same dimensions as A' so that A*G*A = A, G*A*G = G and AG and GA are Hermitian.

The argument mc is the method code, which is optional with default value 1. If mc = 0, then a method based on the singular value decomposition (SVD) is used. If mc = 1, then a method based on Cholesky decomposition is ued. This method works only for symmetric positive semi-definite matrices, and is more efficient than the method based on SVD.

Any singular values less than a tolerance are treated as zero

For example,

```
> X=[1,1,0;
     1,1,0;
     1,1,0;
     1,0,1;
     1,0,1];
> A = X'*X

                      Col 1         Col 2         Col 3
       Row 1              5             3             2
       Row 2              3             3             0
       Row 3              2             0             2
> A.pinv()

                      Col 1         Col 2         Col 3
       Row 1      0.0925926     0.0185185     0.0740741
       Row 2      0.0185185      0.203704     -0.185185
       Row 3      0.0740741     -0.185185      0.259259
> A.sweep()

                      Col 1         Col 2         Col 3
       Row 1            0.5          -0.5             0
```

```
        Row 2          -0.5      0.833333             0
        Row 3             0             0             0
```

3.5.13 Identity

- identity(m,n) creates an identity matrix of order m by n.
- A.identity() resets matrix A to be an identity.
- A.identity(n) resets matrix A to be an identity of order n by n.
- A.identity(m,n) resets matrix A to be an identity of order m by n.

3.5.14 Read data from a file

A.input("<filename>",m,n) inputs an m by n matrix whose elements are input from "<filename>" row-by-row. Each row can occupy several lines, however,each row must start from a new line. Then columns in "<filename>" must be separated with blanks or tabs.

3.5.15 Inverse

A.inv() returns the inverse of A based on a LU factorization. Matrix A must be non-singular, otherwise run-time error will result in. For example,

```
> A = [3,2,1; 2,5,3; 1,3,4]

                     Col 1         Col 2         Col 3
        Row 1            3             2             1
        Row 2            2             5             3
        Row 3            1             3             4
> A.inv()
                     Col 1         Col 2         Col 3
        Row 1     0.458333     -0.208333     0.0416667
        Row 2    -0.208333      0.458333     -0.291667
        Row 3    0.0416667     -0.291667      0.458333
```

3.5.16 Kronecker product

A.kron(B), which is the same as A@B, is the Kronecker tensor product of matrices A and B. The result is a large matrix formed by taking all possible products between the elements of A and those of B. For example, if A is 2 by 3, then A.kron(B) is

```
 [ A(1,1)*B  A(1,2)*B  A(1,3)*B
   A(2,1)*B  A(2,2)*B  A(2,3)*B ]
```

3.5.17 Log determinant

A.logdet() returns the natural log of the g-determinant for a real symmetric positive (semi)definite (psd) matrix. A remains intact. "g" in g-determinant means it takes non-singular part of matrix A with maximum rank into account.

3.5.18 LU factorization

L=A.lu(U,P) computes an LU factorization of a general m-by-n matrix A using partial pivoting with row interchanges. The factorization has the form A = P * L * U where P is a m-by-m permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if m > n), and U is upper triangular (upper trapezoidal if m < n).

There are three forms to call this function

```
L=A.lu();
L=A.lu(U);
L=A.lu(U,P);
```

For example,

```
> A=[1,2;3,4;5,6];
> U=[];P=[];
> L=A.lu(U,P);
> L
  L =
                  Col 1        Col 2
       Row 1          1            0
       Row 2        0.2            1
       Row 3        0.6          0.5
> U
  U =
                  Col 1        Col 2
       Row 1          5            6
       Row 2          0          0.8
> P
  P =
                  Col 1        Col 2        Col 3
       Row 1          0            1            0
       Row 2          0            0            1
       Row 3          1            0            0
```

3.5.19 QR factorization

R=A.qr(Q) computes an QR factorization of a general m-by-n matrix A. The factorization has the form A = QR. When A is a square matrix, then Q is an orthogonal matrix and R is an upper triangular matrix. When m > n, then Q is a m-by-n matrix and R is a n-by-n upper triangular matrix. When m < n, then Q is a m-by-m orthogonal matrix and R is m-by-n upper trapezoidal matrix.

There are two forms to call this function

```
R=A.qr();
R=A.qr(Q);
```

For example,

```
> A=[1,2;3,4];
> Q=[];
> R=A.qr(Q);
> Q
  Q =
                  Col 1        Col 2
       Row 1    -0.316228    -0.948683
       Row 2    -0.948683     0.316228
> R
  R =
                  Col 1        Col 2
       Row 1     -3.16228     -4.42719
       Row 2            0    -0.632456
```

3.5.20 Largest element

A.max() returns a vector whose each element is the largest element of each column of A. If A is a one-row or one-column matrix, then the returned value would be a scalar.

If both A and B are matrices (one of them could be a scalar), then A.max(B) returns a matrix of the same size as A or B with resulting element(i,j) being the largest element among A(i,j) and B(i,j).

For example,

```
> A=rand(2,2)

                   Col 1        Col 2
        Row 1    0.0100414     0.578454
        Row 2     0.973198     0.792847
> B=rand(2,2)

                   Col 1        Col 2
        Row 1     0.619559     0.927987
        Row 2      0.89682     0.990276
> A.max()

        i = 1        i = 2
     0.973198     0.792847
```

```
> A.max(B)

                 Col 1        Col 2
       Row 1   0.619559     0.927987
       Row 2   0.973198     0.990276
```

3.5.21 Mean of elements

A.mean() returns a vector containing average values for each column of A. If A is a one-row or one-column matrix, then the returned value would be a scalar.

3.5.22 Smallest element

A.min() returns a vector whose each element is the smallest element of each column of A. If A is a one-row or one-column matrix, then the returned value would be a scalar.

If both A and B are matrices (one of them could be a scalar), then A.min(B) returns a matrix of the same size as A or B with resulting element(i,j) being the smallest element among A(i,j) and B(i,j).

For example,

```
> A=rand(2,2)

                 Col 1        Col 2
       Row 1  0.0100414     0.578454
       Row 2   0.973198     0.792847
> B=rand(2,2)

                 Col 1        Col 2
       Row 1   0.619559     0.927987
       Row 2    0.89682     0.990276
> A.min()

       i = 1        i = 2
   0.0100414     0.578454
> A.min().min()
       0.0100414
> A.min(B)

                 Col 1        Col 2
       Row 1  0.0100414     0.578454
       Row 2    0.89682     0.792847
```

3.5.23 modulo operation

A.mod(B) returns a matrix with elements being the mod of corresponding elements in A and B. Both A and B must be the same size except B can be a scalar.

3.5.24 norm

For matrix A:

- A.norm() is the largest singular value of A, namely A.svd().max().
- A.norm(2) is the same as A.norm().
- A.norm(1) is the 1-norm of A, the largest column sum, namely A.abs().sum().max().
- A.norm(”inf”) is the infinity norm of A, the largest row sum, namely A.t().abs().sum().max().
- A.norm(”fro”) is the F-norm, defined as (A’*A).diag().sum().sqrt().

For vector V:

- V.norm(p) = (A.abs().ˆp).sum().ˆ(1/p)
- V.norm() = A.norm(2).
- V.norm(”inf”) = A.abs().max().
- V.norm(”-inf”) = A.abs().min().

If A is a scalar, then it will be treated as a single element vector.

3.5.25 power

A.power(B) returns a matrix with elements being the elements in A with the power of the corresponding elements in B. Both A and B must be the same size except B can be a scalar.

A.product() returns a vector containing product values for each column of A. If A is a one-row or one-column matrix, then the returned value would be a scalar.

3.5.26 rank

A.rank() returns rank of an object A based on singular value decomposition.

3.5.27 Reshape elements

A.reshape(m,n) returns a new matrix of m rows and n columns, whose elements are taken from the matrix A row-by-row. If the matrix A doesn’t have enough elements, then the operation cycles back to the beginning of A to get enough values for the new matrix. For example,

```
> A = [1,0,0,0];
> B = A.reshape(3,3)

              Col 1      Col 2      Col 3
    Row 1         1          0          0
    Row 2         0          1          0
    Row 3         0          0          1
> C = A.reshape(3,4)

              Col 1      Col 2      Col 3      Col 4
    Row 1         1          0          0          0
    Row 2         1          0          0          0
    Row 3         1          0          0          0
```

3.5.28 Resize

A.resize(m, n), Matrix A is resized to be m-by-n. The contents in A could be garbage.

3.5.29 Rotation

A.rot90(k) rotate the matrix A counter clockwise k*90 degrees.

This member function is a user-defined function, it's loaded automatically into memory whenever you launch matvec.

For example,

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

                    Col 1        Col 2        Col 3
        Row 1           1            2            3
        Row 2           4            5            6
        Row 3           7            8            9
> A.rot90()

                    Col 1        Col 2        Col 3
        Row 1           3            6            9
        Row 2           2            5            8
        Row 3           1            4            7
> A.rot90(1)

                    Col 1        Col 2        Col 3
        Row 1           3            6            9
        Row 2           2            5            8
        Row 3           1            4            7
```

3.5.30 Randon number

A=rand(m,n) creates a matrix of size m-by-n with each element being a random number from UniformDist(0.0,1.0. A=rand(m,n) creates a matrix of size m-by-n with each element being a random number from UniformDist(0.0,1.0)) A.rand() replaces its elements with random numbers from UniformDist(0.0,1.0). A.sample(m,n) first resizes matrix A to m by n, then replaces its elements with random numbers from UniformDist(0.0,1.0).

sample() is an alias for rand().

3.5.31 Save data into a file

A.save(`"<filename>","<option>`") saves an ASCII copy of the contents of the object A. The second argument `"<option>"` is optional with default value `"noreplace"`; it specifies the output mode with the following possible values: `"out"`, `"app"`, and `"noreplace"`. The mode `"out"` means out by force and overwrite the existing file if it exists. The mode `"app"` causes all output to that file to be appended to the end. The mode `"noreplace"` causes failure if the file does already exist.

3.5.32 Select elements

A.select(B) returns a vector whose element are selected from A(i,j) based on the corresponding element B(i,j) is nonzero or zero.

For example,

```
> A = rand(3,3)
# I want all elements A(i,j) that are >= 0.5
> A.select(A >= 0.5)

       i = 1        i = 2        i = 3        i = 4
    0.865423     0.912348     0.967485     0.641004
```

3.5.33 Round to integers

Return the integer nearest to its oroiginal element. If there are two nearest integers, return the one further away from zero.

```
> round([-2.7,2.7,-2.4,2.4,2.5])
  round =
                  Col 1        Col 2        Col 3        Col 4        Col 5
       Row 1         -3            3           -2            2            3
```

3.5.34 Cast to integers

Return the integer parts of each element.

```
> int([-2.7,2.7,-2.4,2.4,2.5])
  int =
                  Col 1        Col 2        Col 3        Col 4        Col 5
       Row 1         -2            2           -2            2            2
```

3.5.35 solve

A.solve(b, stopval, relax, mxiter) returns the solution (vector or matrix) of the linear equations Ax = b, where b could be a vector or matrix object.

The first argument, which is the right hand side of linear equations, is mandatory. The rest of arguments are optional. The argument stopval is the criterion value to stop the iteration with its default 0.001. When the argument relax = 1.0 (default), the linear equations is solved using Gauss-Seidel iteration; otherwise using the method of successive overrelaxation (SOR). The last argument mxiter is the maximum number of iterations allowed with its default 100,000.

Note that A.solve(b) is much more efficient than A.inv()*b.

3.5.36 sort

If A is a matrix, A.sort() sorts each column of A in an ascending order.

If A is a vector, A.sort() sorts the vector in an ascending order.

If you want to sort it in descending order, the following will do the job:

```
> a = Vector(3,2,4);
> b = a.sort()

       i = 1        i = 2        i = 3
           2            3            4

> b(3:1:-1)         # sort it in descending order
        i = 1        i = 2        i = 3
            4            3            2
```

3.5.37 Standard deviation

A.std() returns a vector containing the standard deviations for each column of A.

This member function is a user-defined function, it's loaded automatically into memory whenever you launch matvec.

For example,

```
> A = [3,2,1;4,5,6;9,8,7];
> A.std()

        i = 1        i = 2        i = 3
      3.21455            3      3.21455
```

3.5.38 Sum

A.sum() returns a vector containing sum values for each column of A. If A is a one-row or one-column matrix, then the returned value would be a scalar.

3.5.39 Sum of least-squares

A.sumsq() returns a vector containing sum square values for each column of A. If A is a one-row or one-column matrix, then the returned value would be a scalar.

3.5.40 Singular value decomposition

singular value decomposition

The singular value decomposition of an m-by-n matrix A is a decomposition

$$A = U*W*V'$$

where U is m-by-n with U'*U = I, W is an n-by-n diagonal matrix and V is an n-by-n orthogonal matrix.

Both A.svd() and A.svd(U,V) returns a vector containing diagonals in W, but the latter has U and V which must be declared as matrix objects before calling.

If A is a scalar, then it will be treated as a 1-by-1 matrix.

For example,

```
> a=[1,2,3;2,9,4;2,1,8];
> u=[]; vt=[];
> a.svd(u,vt)

       i = 1        i = 2        i = 3
     12.0953    0.0538603      6.14008


> u
                    Col 1         Col 2         Col 3
        Row 1   -0.304833     -0.946777      0.103397
        Row 2   -0.770928      0.181542     -0.610502
        Row 3   -0.559237      0.265813      0.785237
> vt
                    Col 1         Col 2         Col 3
        Row 1    -0.24515     -0.966676      0.073756
        Row 2   -0.670281      0.114035     -0.733293
        Row 3   -0.700446      0.229204        0.6759
```

3.5.41 Sweep operation

The sweep operation has four forms:

1. A.sweep()
2. A.sweep(tol)
3. A.sweep(tol,lgdet)
4. A.sweep(tol,lgdet,rank)

It sweeps matrix A. This is best approach to inverse matrix in the sense of memory requirement and speed for a symmetrix matrix. It returns sweeped matrix, log determinant, and the rank of matrix A. By default, tol=this.parameter("EPSILON")*10000;

For example:

```
> A=[5,3,2;3,3,0;2,0,2];
> tol = 2.22045e-12;
> lgdet=[];
> rank=[];
> A.sweep(tol,lgdet,rank)
   .sweep =
                  Col 1         Col 2         Col 3
       Row 1        0.5          -0.5             0
       Row 2       -0.5      0.833333             0
       Row 3          0             0             0
> lgdet
  lgdet =
                  Col 1
       Row 1    1.79176

> rank
  rank =
                  Col 1
       Row 1          2
```

3.5.42 Transpose

A.t() returns the transpose of A, which remains intact. Another way to transpose a matrix is to use matvec operator `'`. Thus A.t() is equivalent to `A'`.

3.5.43 Trace

A.trace(), trace of matrix A, A isn't destroyed. A is not necessary be to square.

3.5.44 Lower triangular

A.tril(k) returns the elements on and below the k-th diagonal of A where k is optional. k=0 (default) is the main diagonal, $k > 0$ is above the main diagonal and $k < 0$ is below the main diagonal.

This member function is a user-defined function, it's loaded automatically into memory whenever you launch matvec.

For example,

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

                    Col 1        Col 2        Col 3
        Row 1           1            2            3
        Row 2           4            5            6
        Row 3           7            8            9
> A.tril()

                    Col 1        Col 2        Col 3
        Row 1           1            0            0
        Row 2           4            5            0
        Row 3           7            8            9
> A.tril(-1)

                    Col 1        Col 2        Col 3
        Row 1           0            0            0
        Row 2           4            0            0
        Row 3           7            8            0
```

3.5.45 Upper triangular

A.triu(k) returns the elements on and above the k-th diagonal of A where k is optional. k=0 (default) is the main diagonal, k $>$ 0 is above the main diagonal and k $<$ 0 is below the main diagonal.

This member function is a user-defined function, it's loaded automatically into memory whenever you launch matvec.

For example,

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

                    Col 1        Col 2        Col 3
        Row 1           1            2            3
        Row 2           4            5            6
        Row 3           7            8            9
> A.triu()

                    Col 1        Col 2        Col 3
        Row 1           1            2            3
        Row 2           0            5            6
        Row 3           0            0            9
> A.triu(1)

                    Col 1        Col 2        Col 3
        Row 1           0            2            3
        Row 2           0            0            6
        Row 3           0            0            0
```

3.5.46 Variance

A.variance() returns a vector containing variance values for each column of A.

3.5.47 Vectorize vertically

A.vec(), where A is a matrix, returns a Vector converting from matrix A column-by-column. For example,

```
> A=[1,2;3,4]

                    Col 1        Col 2
        Row 1           1            2
        Row 2           3            4
> A.vec()

        i = 1        i = 2        i = 3        i = 4
            1            3            2            4
```

3.5.48 Vectorize a lower triangular

A.vech(), where A is a matrix, returns a vector converting from lower triangular part of matrix A column-by-column. For example,

```
> A=[1,2;3,4]

                    Col 1        Col 2
        Row 1           1            2
        Row 2           3            4
> A.vech()

        i = 1        i = 2        i = 3
            1            3            4
```

3.5.49 Zero elements

zeros(m,n) creates a matrix object of m by n, with each element being zero.

A.zeros() resets each element of matrix A to be zero.

3.5.50 Unit elements

ones(m,n) creates a matrix object of m by n, with each element being unit 1.

A.ones() resets each element of matrix A to be unit 1.

```
 A=ones(2,3)
  A =
                   Col 1        Col 2        Col 3
       Row 1           1            1            1
```

```
      Row 2            1            1            1
> A=[1,2,3]
  A =
                  Col 1        Col 2        Col 3
      Row 1            1            2            3
> A.ones()
  .ones =
                  Col 1        Col 2        Col 3
      Row 1            1            1            1
```

3.6 Special Matrices

In matvec, there are eight user-defined functions to create special matrix objects:

hadamard	hankel	hilb	invhilb	pascal
toeplitz	vander	magic		

The first seven are defined in macro package "special_matrix". The last one `magic` is a default macro. See details in Chapter 9, section 9.3.

Chapter 4

Program Flow Control

The program flow-control statements provided by matvec are quite rich: *if*, *while*, *for*, *break*, *continue*, *repeat*, and *null*. They are similar to those in the C/C++ language:

A BLOCK construction is defined as either a single statement or a sequence of statements (not necessary to be in the same line) enclosed with braces. For instance

```
x = 8;
```

is a BLOCK, and

```
{
   a = 6;
   b = sample(3,5);
}
```

is a BLOCK, too.

4.1 If–Endif and If–Else Statements

The *if* statement has two forms:

```
if (expression) BLOCK1 endif
if (expression) BLOCK1 else BLOCK2
```

The expression must evaluate to a scalar, otherwise a run-time error will result.

In the first form of *if* statement, it simply performs a test on the expression in the parenthesis, and if the expression evaluates to a non-zero scalar, the statement(s) in BLOCK will be executed. The *endif* is mandatory to tell matvec interpreter not to expect an else-branch.

In the second form of *if* statement, it performs a test on the expression in the parenthesis, and if the expression is true then executes BLOCK1 otherwise execute BLOCK2.

A nested *if* is an *if* that is the part of a BLOCK belonging to another *if* or *else*. In matvec an *else*-branch always refers to the nearest *if* statement that is within the same block as the *else* and is not already associated with an *if*. For example,

```
if (i) {
  if (j) BLOCK1 endif;
  if (k) BLOCK2 else BLOCK3
}
else {
   BLOCK4
}
```

Note that matvec does not have *elseif* statement, nor does C/C++. matvec, however, allow users to form an *if-else-if* ladder:

```
if (expr)
   BLOCK1
else if (expr)
   BLOCK2
else if (expr)
   BLOCK3
   .
   .
   .
else
     BLOCKn
```

The final statement in this *if-else-if* ladder must be either *else* or *endif*. The expr-conditions are evaluated from top downward. As soon as a true condition is found, the BLOCK associated with it is immediately executed and the rest of the ladder is bypassed. If none of the expr-conditions are true, then BLOCKn is executed if this ladder has a final *else* branch, otherwise no action takes place if this ladder ends with a *endif*. For example,

```
if (A.class() == "Scalar") {
   if (A <  0) {
      retval = -1;
   }
   else if (A == 0) {
      retval = 0;
   }
   else {
      retval = 1;
   }
```

```
}
endif;
```

The `any` and `all` functions may be useful with *if* statement as shown below:

```
if (A.any().any()) {
     "there are non-zero elements in matrix A"
}
else {
   " all elements in matrix A are zero"
}
```

4.2 For Statement

The *for* statement has also two forms:

```
for (initialization; condition; increment) BLOCK
for (i in expression) BLOCK
```

The first form of *for* statement is similar to that in C/C++. In general, `initialization` is generally an assignment statement that is used to set the loop control variable. The `condition` is usually a relational expression that determines when the loop exits. The `increment` defines how the loop control variable changes each time the loop is repeated. You must separate these three sections by semicolons. The *for* loop continues to execute until the condition becomes false. The execution resumes on the statement following *for*. Note that all those three sections can be omitted simultaneously, thus `for (; ;) BLOCK` construction provides user an infinite loop. The *break* statement is the only wat to exit such an infinite loop.

The second form of *for* statement is similar to that in Maple language. The expression must evaluate to a either matrix, or vector, or list, otherwise a run-time error will result. The *for* statement simply executes the BLOCK the N times where N is the size of expression unless it encounters the *break* statement, which will terminate the nearest for–loop execution. The *for* statement can be nested, too. For instance,

```
> A = zeros(3,4);
> for (i=0; i<3; i++) {
     for(j in 1:4) A(i,j) = i+j;
  }
> A
            Col 1         Col 2         Col 3         Col 4
  Row 1   2.00000       3.00000       4.00000       5.00000
  Row 2   3.00000       4.00000       5.00000       6.00000
  Row 3   4.00000       5.00000       6.00000       7.00000
```

4.3 While Statement

The while statement is very similar to that in C/C++.

```
while (expression) BLOCK
```

If the expression evaluates to anything rather than zero, then the BLOCK is executed, otherwise goes to the statement right after the BLOCK.

For example,

```
> i=1; j=0;
> while(i > 10) {
     j += i++;
  }
> j
  45
```

4.4 Repeat–Until Statement

The repeat statement is very similar to that in PASCAL language.

```
repeat BLOCK until (expression)
```

the BLOCK is executed until the expression evaluates to anything rather than zero. Then, it goes to the statement right after the expression.

For example,

```
> i=1; j=0;
> repeat {
    j += i++;
  } until (i >= 10)
> j
  45
```

Chapter 5

File Stream

matvec provides a class of input/out stream, which is a substantially limited C++ I/O stream system. A stream is a logical object associated with a physical device such as disk.

When output is performed, data is not immediately written to the physical device associated with the stream. Instead, data is stored in an internal buffer until the buffer is full. However, data in the buffer can be forced out to disk before the buffer is full by using either endl() or flush(). Note that buffering is used to improve performance. Flushing each line of output using either endl() or flush() decreases efficiency.

5.1 Standard File Streams

There are three standard file streams created automatically each time when matvec is invoked: cout, cin, cerr.

5.1.1 cout

not yet available

5.1.2 cin

not yet available

5.1.3 cerr

not yet available

5.2 User Specified File Stream

5.2.1 Creation

`f = FileStream("<filename>","<mode>")` creates an object of class FileStream and assign to f. The `"<filename>"` can be either relative or absolute, and the `"<mode>"` must be one the following or appropriate combination: `out`, `in`, `app`, `noreplace`.

5.2.2 Example

```
> f = FileStream("try.out","out");
> f.out("I am testing Matvec file stream\n");
> Pi=3.14159;
> f.out("Pi = ", Pi, " 2*Pi = ", 2*Pi,"\n");
> f.close()
> $more try.out
I am testing Matvec file stream
Pi = 3.14159 2*Pi = 6.28319
```

5.3 Member Function

5.3.1 open

f.open(`"<filename>","<mode>"`) opens the file stream f. The `"<filename>"` can be either relative or absolute, and the `"<mode>"` must be one the following or appropriate combination: `out`, `in`, `app`, `noreplace`.

5.3.2 reopen

f.reopen() reopens the file stream for either writing or reading. not yet available

5.3.3 close

f.close() closes the file stream.

5.3.4 set

f.set(`"<flag>"`,n) sets value n for the associated flag: `"width"`, `"precision"`, `"format"`, `"fill"`.

5.3.5 out

f.out(arg1,arg2,...) writes out arg1, arg2, ... to the file stream.

5.3.6 in

f.in(arg1,arg2,...) reads in arg1, arg2, ... from the file stream.
not yet available

5.3.7 getline

f.getline() reads and returns a string line from the file stream.
not yet available

5.3.8 eof

f.eof() returns nonzero when the end of the file stream has been reached; otherwise it returns zero.

5.3.9 endl

f.endl() immediately flushes data in the buffer to its destination: either disk or screen.

5.3.10 flush

f.flush() forces the data in the buffer to be physically written to its destination (either disk or screen) in the same way as endl().

5.3.11 rewind

f.rewind() resets the position indicator to the beginning of the file stream. Note that f must be a read/write FileStream.

Chapter 6

Date and Time

Though I do not know what time and date exactly mean, I did build several conventional functions related to time and date.

6.1 Digital Clock

6.1.1 Time

time() returns the current system calendar time, which is a double precision number. It is a builtin function. For example,

```
> this.parameter("OUTPUT_PRECISION",10);
> time()
        815691299
```

6.1.2 Localtime

localtime() returns a vector of local time with seven components: year, month, day-of-month, day-of-year, hour, minute, and second. Such a vector will be referred to as time vector (tv). Whereas localtime(t) converts the calendar time represented by t into the local time vector. It is a builtin function. For example,

```
> t = time()
        8.15692e+08

> localtime()

      i = 1      i = 2     i = 3     i = 4     i = 5     i = 6
       1995         11         6      3101         4         9

      i = 7
         13
```

```
> localtime(t)

      i = 1     i = 2     i = 3     i = 4     i = 5     i = 6
       1995        11         6       310        14         9

      i = 7
          5
```

6.1.3 Gmtime

gmtime() is the same as localtime() except for Greenwich mean time. So is gmtime(t).

6.1.4 Asctime

asctime(tv) converts the time vector (tv) to a character string. It is a builtin function. For example,

```
> asctime(localtime())
        Sun Nov  6 14:16:34 1995
```

6.1.5 Difftime

difftime(t1,t0) returns the difference in second between two times t1 and t0, where t1 and t0 must be values returned by time(). It is a builtin function. The following script times a loop using time() and difftime():

```
> # timing a loop using time() and difftime()
> t0 = time();
> for (i=0; i<1000; i++) sample(100,100);
> t1 = time();
> difftime(t1,t0)
        41
```

6.1.6 Mktime

mktime(tv) converts the time vector, tv, to system calendar time. It is a builtin function. The following statement returns the system calendar time of the current local time vector:

```
> mktime(localtime())
        8.15697e+08
```

6.1.7 Date

date() is a macro function, equivalent to asctime(localtime()). For example

```
> date()
        Sun Nov  6 14:24:50 1995
```

6.1.8 Ctime

ctime(t) converts the calendar time represented by t to a string. It is a macro function, equivalent to asctime(localtime(t)). For example,

```
> t = time();
> ctime(t)
        Sun Nov  6 14:30:13 1995
```

6.1.9 Clock

clock() is a macro friend function, equivalent to localtime().

6.2 Western Calendar

6.2.1 Leap year

leapyear(year) is a macro friend function. It returns 1 if the argument year is a leap year, otherwise return 0. For example,

```
> # Is 1990 a leap year?
> leapyear(1990)
      0
> # Is 1996 a leap year?
> leapyear(1996)
      1
```

Chapter 7

User-Defined Function

7.1 Syntax

```
function obj.foo(args) { <MATVEC statements> }
function foo(args) { <MATVEC statements> }
protected function obj.foo(args) { <MATVEC statements> }
protected function foo(args) { <MATVEC statements> }
```

7.2 Description

User-defined functions in matvec can be constructed or defined) by the above four forms. The first form is so-called a user-defined member function with respect to object *obj*, whereas the second form is so-called a user-defined friend function. The third and fourth are the same as the first two except that functions are protected. In another word function definition in the last two forms can not be overwritten.

Usually, the definition of a user-defined functions are stored in a disk file. Before calling them, we have to load them into the computer memory using the matvec builtin function `input`:

input(`"<filename>"`)

User-defined functions in matvec are treated as objects, in a manner similar to other objects such as Matrix, Vector. Thus, in principle, users can assign/copy them like ordinary variables, but users should avoid to use this feature.

By default all variables within a function are global except those symbolic arguments in function header and those declared in local statements: local(A,B,C,...);

Note that *local* statement can appear anywhere only within function definition, and multiple *local* statements are also allowed.

Function execution will terminated right after return statement. the returning value of a function is the last expression evaluated in the function execution. The return statement in the function is optional.

7.3 Scope

7.4 Recursion

not working yet so far.

7.5 Examples

```
protected function A.sign()
{
   #
   # Return 1,  if element >= 0
   # Return 0,  if element == 0
   # Return -1, if element < 0
   #
   # Tianlin Wang at UIUC, Fri Dec 30 16:10:38 CST 1994
   #

   local(i, j, retval);
   if (A.class() == "Scalar") {
      if (A >  0)  retval = 1;  endif;
      if (A == 0)  retval = 0;  endif;
      if (A <  0)  retval = -1; endif;
   }
   else if (A.class() == "Vector") {
      retval = Vector();
      retval.resize(A.size());
      for (i in 1:A.size()) {
         if (A(i) >  0) retval(i) = 1;  endif;
         if (A(i) == 0) retval(i) = 0;  endif;
         if (A(i) <  0) retval(i) = -1; endif;
      }
   }
   else if (A.class() == "Matrix") {
      retval = Matrix(A.nrow(),A.ncol());
      for (i in 1:A.nrow()) {
         for (j in 1:A.ncol()) {
            if (A(i,j) >  0) retval(i,j) = 1;  endif;
            if (A(i,j) == 0) retval(i,j) = 0;  endif;
            if (A(i,j) <  0) retval(i,j) = -1; endif;
```

```
         }
      }
   }
   else {
      retval = 0;
      error("sign only works for numeric objects");
   }
   return retval;
}
```

Chapter 8

Plotting

The 2-dimension and 3-dimension plotting within matvec interface are currently using external shell program: *gnuplot*. This means *gnuplot* must be installed in the same machine where matvec resides.

8.1 Creation

p = Plotter() creates an object of class Plotter.

8.2 Member Function

8.2.1 plot

p.plot(`"literal function such as sin(x)"`) plots function y=sin(x) on your X11 terminal.

p.plot(x,y,`"<options>"`) plots vectors x versus y on your X11 terminal. "options" is optional, with possible value: `{title '<title>'}{with <style>}` where `<style>` can be one of the following six styles: lines, point, linespoints, impulses, dots, step, or errorbars.

8.2.2 plot3D

p.plot3D(`"literal function such as sin(x)+cos(y)"`) plots function $z = \sin(x) + \cos(y)$ on your X11 terminal.

p.plot3D(x,y,z,`"<options>"`) plots vectors x, y versus z on your X11 terminal `"<options>"` is optional, with possible value: `{title '<title>'}{with <style>}` where `<style>` can be one of the following six styles: lines, point, linespoints, impulses, dots, step, or errorbars.

8.2.3 replot

p.replot() replot the previous data or function after setting title, labels, etc.

8.2.4 set

p.set(`"key value"`) where key can be one of followings:

- angles `{degree | radians}`
- arrow `<tag> {from sx,sy,sz} {to ex,ey,ez}`
- autoscale `<axes>`
- border
- boxwidth `<width>`
- clabel
- clip `<clip_type>`
- contour `{base | surface | both}`
- data sytyle `<style_choice>`
- dummy `<dummy_var> {,<dummy_var>}`
- format `{<axes>} {'<format_string>'}`
- function style `<style_choice>`
- hidden3d
- key `<x>,<y>{,<z>}`
- label `{<tag>}{'<label_text>'}{at x,y{,z}}{<justification>}`
- logscale `<axes><base>`
- mapping `{cartesian | spherical | cylindrical}`
- offsets `<left>,<right>,<top>,<bottom>`
- output `{'<filename>'}`
- parametric
- polar
- rrange `[{<rmin> : <rmax>}]`
- samples `<samples_1> {,<samples_2>}`
- size `{<xscale>,<yscale>}`
- surface
- terminal `{<terminal_type>}`
- tics `{<direction>}`
- time `{<xoff>} {,<yoff>}`
- title `{'<title_text>'} {<xoff>} {,<yoff>}`
- trange `[{<tmin> : <tmax>}]`
- urange `[{<umin> : <umax>}]`
- view `<rot_x> {,{<rot_z>} {,{<scale>} {,<scale_z>}}}`
- vrange `[{<vmin> : <vmax>}]`
- xlabel `{'<label>'} {<xoff>} {,<yoff>}`
- xrange `[{<xmin> : <xmax>}]`
- xtics `{{<start>,<incr> {,<end>}}|{....}`

xmtics	
xzeroaxis	
ylabel	{'<label>'}{<xoff>}{,<yoff>}
yrange	[{<ymin> : <ymax>}]
ytics	
ymtics	
yzeroaxis	
zero	<expression>
zlabel	{'<label>'} {<xoff>} {,<yoff>}
zrange	[{<zmin> : <zmax>}]
ztics	
zmtics	

8.2.5 save

p.save("<fname>","<format>") save the graph into your disk in the specified format. The second argument is optional with default value "postscript". The common graphic formats are postscript, latex, fig, eepic, emtex, mf, pbm, x11, etc.

8.2.6 open

p.open() open (re-open) a shell for external plotting program

8.2.7 close

p.close() close the graph window on X11 terminal. Note that after closing the graph window, all settings such as title,labels are also cleared.

8.3 Examples

8.3.1 Example 1

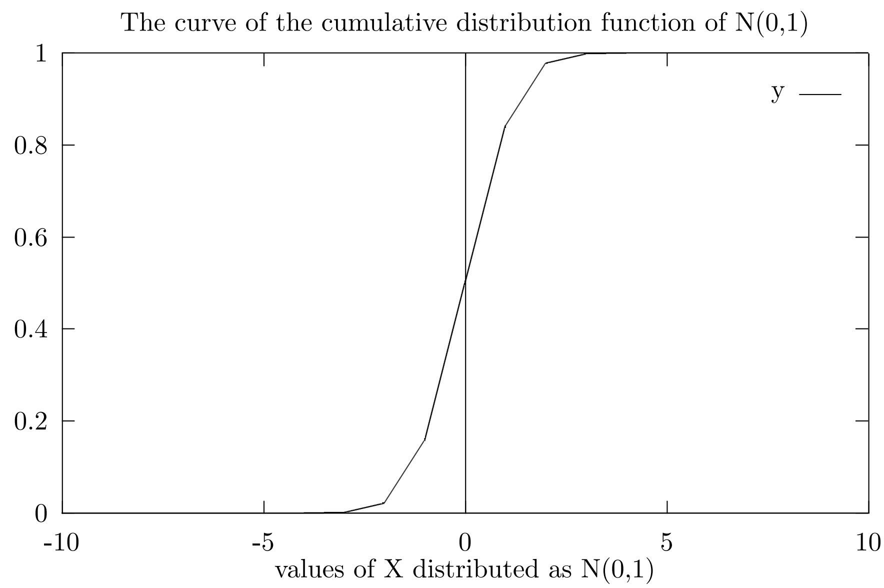
Try the following example

```
> N = StatDist("Normal",0,1);
> x = -10:10;
> y = N.cdf(x);

> p = Plotter();
> p.set("xlabel 'values of X distributed as N(0,1)'");
> p.plot(x,y);
```

The picture from the above matvec script is shown in Fig. 8.1:

Figure 8.1: The curve of the cumulative distribution function of N(0,1)



8.3.2 Example 2

```
> p = Plotter();
> p.plot3D("sin(x)+cos(y)");
> p.save("myfig.ps");
```

8.3.3 Example 3

The following matvec script draw the graph of normal pdf function for two sets of parameters.

```
D1 = StatDist("Normal",0,1);
D2 = StatDist("Normal",2,0.5);

x = -4:4:0.1;
xy1 = [x; D1.pdf(x)]';
xy2 = [x; D2.pdf(x)]';
xy1.save("Normal.dat1",1);
xy2.save("Normal.dat2",1);

curve1 = " 'Normal.dat1' title 'N(0,1)' with lines";
curve2 = " 'Normal.dat2' title 'N(2,0.5)' with lines";
curve = curve1 + "," + curve2;
```

```
p = Plotter();
p.set("nozeroaxis");
p.set("size 4/5.0, 2.5/3.0");
p.set("title 'Normal Probability Density Function (pdf)' ");
p.set("xlabel 'x'");
p.set("ylabel 'pdf(x)'");
p.set("key at -2.0,0.5");

p.plot(curve);
```

The graph is shown on page 92.

8.3.4 Example 4

The following matvec script draw the graph of the pdf of $b(25, 0.15)$.

```
D = StatDist("Binomial",25,0.15);

x = -0.5:19.5;
xy = [x; D.pdf([0:20])]';

xy.save("Binomial.dat",1);

p = Plotter();
p.set("title 'Binomial Probability Density Function (pdf)' ");
p.set("nozeroaxis");
p.set("format y '%3.2f' ");
p.set("xtics 0, 1.0, 13");
p.set("xrange [-0.5 : 12.5]");
p.set("size 4/5.0, 2.5/3.0");
p.set("xlabel 'k'");
p.set("ylabel 'pdf(k)'");

p.plot(" 'Binomial.dat' title 'b(25,0.15)' with steps");
```

The graph is shown on page 108.

Chapter 9

Macro Packages

A macro package is a single plain text file, consisting of a set of user-defined functions to accomplish a specific task. There are two types of packages: pre-loaded and post-loaded. Those pre-loaded macro packages are automatically loaded whenever you start matvec session just like the built-in functions. Those pre-loaded macro packages are stored in the folder `$(MATVEC_HOME)/packages/default/b`.

Before accessing to functions in a post-loaded macro package, you first have to load the package into your computer memory with matvec function

```
package("<package_name>")
```

Each macro package should have an appropriate package name (related to the task) with the extension `.mv`. A package_name is referred to as an absolute name if it contains either at least one slash (`/`) or file extension `.mv`, otherwise it is referred to as a relative name. If a package_name is a relative name, matvec will only search for it in the directory `$(MATVEC_HOME)/packages/contrib`. If a package name is an absolute name, it must contain the full information including pathname and file extension. If you want to load a macro package called `mypackage` in the current working directory, the following trick works:

```
package("./mypackage")
```

However, users are strongly encouraged to name your macro package with file extension `.mv`.

A variety of macro packages are provided with the standard matvec distribution. They usually reside in the MATVEC_HOME/packages/contrib folder. These macro packages are called the contribution packages because most of them are contributed by matvec experienced users. They are the good resources for matvec beginners to learn how to properly and efficiently use matvec language. So take a close look at them.

Any good macro packages written by users can be easily shared by the local group provided that your system administrator is willing to put a copy of your package in the `$(MATVEC_HOME)/packages/contrib` folder.

As a matter of fact, a few of matvec standard functions such as `A.sign()` and `magic(n)` are macros. They are called the matvec default macro packages. The differences between a default macro package and a contribution package are

- the former is automatically loaded into memory whenever matvec is launched.
- the former is residing in the `$(MATVEC_HOME)/packages/default` folder.

9.1 Demo Package

The macro package `demo` is designed for beginners to have a taste of matvec system. It provides two functions: demo() and demo_src(). To use one of these two functions, you first have to load the whole package into computer memory using matvec function `package( "demo" )`.

9.1.1 demo() function

Here is the macro script of demo() function:

```
function demo(program_name)
{
   #
   # run a demo program in $(MATVEC_HOME)/examples/cli/txt
   #
   # Tianlin Wang at UIUC, Sat Jan 28 16:13:05 CST 1995
   #

   local(retval,filename);
   if (program_name.class() == "String") {
      filename = getenv("MATVEC_HOME") + "/examples/cli/txt/";
      filename += program_name;
      retval = input(filename);
   }
   else {
      error("demo(program_name): program_name must be a string");
      retval = 0;
   }
   return retval;
}
```

There are a number of demo examples coming with the matvec standard distribution. Usually, these examples are residing in MATVEC_HOME/examples/cli/txt ' directory, where MATVEC_HOME can be obtained using matvec function getenv("MATVEC_HOME"). These demo examples are listed and briefly described below:

hello: greetings to every body.

gianola: matrix version of BLUP.

try: variance component estimation based on a five-animal pedigree

VG: single trait variance component estimation.

susan: single trait BLUP.

For examples

```
> package("demo");
> demo("try")
       .matvec.799959823.0
       .matvec.799959823.1
 iteration  sigma_1 ..... sigma_e  log_likelihood
    0:            1           2     -6.10457
    1:      1.74667     2.87176     -6.48199
    2:      1.65664     2.91128     -4.73543
    3:      2.23916     2.68895     -5.76399
    4:      2.22348      2.6937     -5.76309
    5:      2.21558     2.69613     -5.76264
    6:      2.21156     2.69737     -5.76241
    7:      2.20951     2.69801     -5.76229
    8:      2.20846     2.69833     -5.76223
    9:      2.20792      2.6985      -5.7622
   10:      2.20765     2.69859     -5.76218
   11:      2.20751     2.69863     -5.76217

          some extra information in the model
 ----------------------------------------------------
 variance for animal =

                   Col 1
       Row 1     2.20751
 residual variance =

                   Col 1
       Row 1     2.69863

 MME dimension   : 8
 non-zeros in MME: 23

       basic statistics for dependent variables
  ---------------------------------------------------
     trait-name        n         mean        std
             y         6      8.33333    2.06559
 ------------------------------------------------------
```

9.1.2 demo_src() function

`demo_src("<program_name>")` is used to displays the source code of demo program `"<program_name>"`; whereas `demo_src()` displays the source code of the demo macro package itself.

Here is the macro script of demo_src() function:

```
function demo_src(program_name)
{
   #
   # list a demo program in $(MATVEC_HOME)/examples/cli/txt
   #
   # Tianlin Wang at UIUC, Sat Jan 28 16:13:05 CST 1995
   #

   local(retval,shellcmd,filename);
   if (program_name.class() == "String") {
      filename = getenv("MATVEC_HOME") + "/examples/cli/txt/";
      filename += program_name;
      shellcmd = PAGER + " " + filename;
      retval = system(shellcmd);
   }
   else {
      error("demo_src(program_name): program_name must be a string");
      retval = 0;
   }
   return retval;
}
```

For examples

```
> package("demo");
> demo_src("try")
######################## try  ######################
data_file_name = RawData{
     A2    1   1.0   6.0
     A2    1   1.0   8.0
     A3    1   1.0    .
     A3    1   1.0   8.0
     A4    .   0.0   5.0
     A4    2   2.0   7.0
     A5    2   2.0   9.0
     A5    2   2.0  12.0
}

ped_file_name = RawData {
   A1         .       .
```

```
   A2         .       .
   A3        A1      A2
   A4         .      A2
   A5         .      A3
}

D = Data();
D.input(data_file_name,"animal$ herd _skip y");

P = Pedigree();
P.input(ped_file_name);

M = Model("y = intercept herd animal");
M.variance("residual",2.0);
M.variance("animal",P,1.0);
M.fitdata(D);

M.vce_emreml();  // correct answer: sigma_e = 2.69863, sigma_a = 2.20751
```

9.2 Prime Number Package

In the prime number package, there are three interesting functions related to prime number.

9.2.1 prime(n)

It returns the first n prime numbers.

9.2.2 prime_next(n)

It returns the prime number next to n. If n is the prime number, then it simply returns n itself.

9.2.3 prime_less(n)

It returns a vector containing all prime numbers less or equal to the argument n.

9.2.4 examples

```
> package("prime");
> prime(4)

        i = 1         i = 2         i = 3         i = 4
            2             3             5             7
> prime_next(4)
```

```
       5
> prime_next(888)
       907
> prime_less(8)

       i = 1        i = 2        i = 3        i = 4
           2            3            5            7
```

9.3 Special Matrix Package

The macro package `special_matrix` contains following special matrices: hilb, invhilb, hankel, vander, hadamard, pascal, and toeplitz. They are contributed by users. The macro package `magic` contains two functions: magic() and magic_odd().

9.3.1 hilb

hilb(n) creates the Hilbert matrix of order n. The i, j element H(i,j) of a Hilbert matrix is defined as H(i,j) = 1 / (i + j - 1). For example,

```
> package("special_matrix");
> hilb(3)

                   Col 1        Col 2        Col 3
       Row 1           1          0.5     0.333333
       Row 2         0.5     0.333333         0.25
       Row 3    0.333333         0.25          0.2
```

9.3.2 invhilb

invhilb(n) returns the inverse of the Hilbert matrix of order n. This is the exat results. See and compare invhilb(n) with hilb(n).inv(). For example,

```
> package("special_matrix");
> invhilb(3)

                   Col 1        Col 2        Col 3
       Row 1           9          -36           30
       Row 2         -36          192         -180
       Row 3          30         -180          180
```

9.3.3 hankel

hankel(C) returns a square Hankel matrix whose first column is C and whose elements are zero below the first anti-diagonal; whereas hankel(C,R) returns a Hankel matrix whose first column is C and whose last row is R. Hankel matrices are symmetric, constant across the anti-diagonals, and have elements H(i,j) = R(i+j-1). For example,

```
> package("special_matrix");
> hankel([1,2,3])

                    Col 1        Col 2        Col 3
        Row 1           1            2            3
        Row 2           2            3            0
        Row 3           3            0            0
> hankel([1,2,3],[3,4,5])

                    Col 1        Col 2        Col 3
        Row 1           1            2            3
        Row 2           2            3            4
        Row 3           3            4            5
```

9.3.4 vander

vander(C) returns the Vandermonde matrix whose second to last column is C. The j-th column of a Vandermonde matrix is given by $A(*, j) = C^(n - j)$. For example,

```
> package("special_matrix");
> vander([1,2,3])

                    Col 1        Col 2        Col 3
        Row 1           1            1            1
        Row 2           4            2            1
        Row 3           9            3            1
```

9.3.5 hadamard

hadamard(k) return the Hadamard matrix of order $n = 2^k$.

```
> package("special_matrix");
> hadamard(2)
  hadamard =
                   Col 1        Col 2        Col 3        Col 4
       Row 1           1            1            1            1
       Row 2           1           -1            1           -1
       Row 3           1            1           -1           -1
       Row 4           1           -1           -1            1
```

9.3.6 pascal

pascal(n) returns the Pascal matrix of order n: a symmetric positive definite matrix with integer entries, made up from Pascal's triangle. For example,

```
> package("special_matrix");
> pascal(3)

                Col 1       Col 2       Col 3
      Row 1         1           1           1
      Row 2         1           2           3
      Row 3         1           3           6
```

9.3.7 toeplitz

toeplitz(C) returns a symmetric (or Hermitian) Toeplitz matrix; whereas toeplitz(C,R) returns a non-symmetric Toeplitz matrix having C as its first column and R as its first row. For example,

```
> package("special_matrix");
> toeplitz([1,2,3])

                Col 1       Col 2       Col 3
      Row 1         1           2           3
      Row 2         2           1           2
      Row 3         3           2           1

> toeplitz([1,2,3],[1,4,5])

                Col 1       Col 2       Col 3
      Row 1         1           4           5
      Row 2         2           1           4
      Row 3         3           2           1
```

9.3.8 magic

A magic matrix of order n contains numbers from 1 to n^2, sum of each row, each column and main diagonal are the same.

```
> A=magic(3)
  A =
               Col 1       Col 2       Col 3
     Row 1         8           1           6
     Row 2         3           5           7
     Row 3         4           9           2
```

9.4 Linear Programming Package

not available yet

Chapter 10

Mixed Model Analyses

10.1 Best Linear Unbiased Prediction (BLUP)

10.1.1 Data

Suppose we have a five-animal pedigree stored in an ASCII file called `try.ped`:

```
// a animal pedigree
// note that unknown parents can also be represented by 0
// individual father mother
   A1         .       .
   A2         .       .
   A3        A1      A2
   A4         .      A2
   A5         .      A3
```

Then, the eight-line records from four animals were obtained, and stored in another ASCII file called `try.dat`:

```
// a small test data
// animal herd  wt    y
     A2    1   1.0   6.0
     A2    1   1.0   8.0
     A3    1   1.0    .
     A3    1   1.0   8.0
     A4    .   0.0   5.0
     A4    2   2.0   7.0
     A5    2   2.0   9.0
     A5    2   2.0  12.0
```

There are several matvec features shown in these two files:

- either the pedigree file or the raw data file can contain comments which is started from the double backslash. This could be very handy when you want to write down a note about your data. Comments and empty lines will be quietly ignored by matvec.

- missing values must be represented by a dot(.) with an exception that unknown parents in the pedigree file can also be represented by 0 (the cost is that we can't use 0 as a legitimate animal identification any more). Missing values can appear anywhere in both files. matvec will automatically and appropriately handle missing values for you.
- animal or herd identification can be either a string or a number.

In the following several sections, we will demonstrate how to analyze the above data using linear model methodology.

10.1.2 Model

A single trait animal model is assumed:

$$y_i = \mu + h_i + a_i + e_i$$

10.1.3 Obtain BLUP using matrix algebra

First of all, we write down a script using matrix tools and functions provided by matvec. You can use any of you favorable ASCII editor, and save the file as `try1` in your current working directory.

```
A = [1.0, 0.0, 0.5, 0.0,  0.25;
     0.0, 1.0, 0.5, 0.5,  0.25;
     0.5, 0.5, 1.0, 0.25, 0.5;
     0.0, 0.5, 0.25,1.0,  0.125;
     0.25,0.25,0.5, 0.125,1.0];   // A = relationship matrix

X = [1.0, 1.0, 0.0;
     1.0, 1.0, 0.0;
     1.0, 1.0, 0.0;
     1.0, 0.0, 1.0;
     1.0, 0.0, 1.0;
     1.0, 0.0, 1.0];

Z = [0.0, 1.0, 0.0, 0.0, 0.0;
     0.0, 1.0, 0.0, 0.0, 0.0;
     0.0, 0.0, 1.0, 0.0, 0.0;
     0.0, 0.0, 0.0, 1.0, 0.0;
     0.0, 0.0, 0.0, 0.0, 1.0;
     0.0, 0.0, 0.0, 0.0, 1.0];

y = [6.0;
```

```
     8.0;
     8.0;
     7.0;
     9.0;
    12.0];

XPX = X'*X;
XPZ = X'*Z;
ZPZ = Z'*Z;

XPy = X'*y;
ZPy = Z'*y;
rhs = [XPy;
       ZPy];

sigma_e = 2.0;
sigma_a = 1.0;
r = sigma_e/sigma_a;
G = A*sigma_a;


MME = [XPX,  XPZ;
       XPZ', ZPZ + A.inv()*r];

blup = MME.sweep()*rhs
```

Now, at the matvec prompt, simply type

```
> input try1
  blup =

                   Col 1
        Row 1      5.51000
        Row 2      1.91000
        Row 3      3.60000
        Row 4     0.260000
        Row 5    -0.260000
        Row 6     0.260000
        Row 7    -0.670000
        Row 8     0.670000
```

The above script written in matvec language is the exactly what Professor Gianola told me to do when I was taking his linear model course. Step by step, I have clearly shown you what I was doing in the above script. Now I will do the same job in an encapsulated but efficient way using matvec higher level functions.

10.1.4 Obtain BLUP using matvec higher level functions

M.blup() returns the blup as a vector if M is a *Model* object. I wrote down the following matvec script and saved it in an ASCII file called `try2`:

```
P = Pedigree();
P.input("try.ped");

D = Data();
D.input("try.dat","animal$ herd _skip y");

M = Model();
M.equation("y = intercept + herd + animal");
M.variance("residual",2);
M.variance("animal",P,1);
M.fitdata(D);

M.blup()
```

There are two matvec features I want to mention here:

- In statement

  ```
  D.input("try.dat","animal$ herd _skip y");
  ```

  the dollar sign (`$`) after animal is used to indicate that this data-column is string column. The `_skip` is one of reserved data-column name meaning to *skip* or ignore the column.

- In statement

  ```
  M.equation("y = intercept + herd + animal");
  ```

  the `intercept` is a reserved model-term meaning an intercept or a grand-mean. The `+` between model terms is optional. The interaction term or the nested term can be expressed in the same way as SAS. For instance, A*B and x(A). The above statement about model can also be re-written as

  ```
  M.equation("y = intercept +",
             "herd + animal");
  ```

  This should give you enough hint about how to break down a model with dozens of terms.

If you add a semicolon(;) after statement `M.blup()`, then BLUP solution would not be displayed on screen.

Now, at the matvec prompt, I type

```
> input try2
    i = 1        i = 2       i = 3        i = 4        i = 5
     7.42          0          1.69          0.26         -0.26

    i = 6        i = 7       i = 8
     0.26        -0.67        0.67
```

If you want a more descriptive output for BLUP, then use statement `M.save("try.out")` right after `M.blup()`. The following could be the possible contents in `"try.out"`:

```
      some extra information in the model
 ------------------------------------------------
 variance for animal =

                    Col 1
       Row 1            1
 residual variance =

                    Col 1
       Row 1            2

 MME dimension   : 8
 non-zeros in MME: 23

    basic statistics for dependent variables
  ------------------------------------------------
   trait-name         n          mean         std
           y          6      8.33333      2.06559
 --------------------------------------------------

          BLUP (BLUE, Posterior_Mean)
   -----------------------------------------------

        Term      Trait  1
   intercept            y

           1         7.42

        Term      Trait  1
        herd            y

           1            0
           2         1.69

        Term      Trait  1
      animal            y
```

```
A1         0.26
A2        -0.26
A3         0.26
A4        -0.67
A5         0.67
```

Note that because of different g-inverse algorithms, the estimates of herds effects seems quite different from the two above programs. However, the value of the estimable function h1-h2 is identical. Of course, the BLUP values for animal additive effects are identical, too.

There are several optional arguments for M.blup() function to let user to choice appropriate solver, maximum iterations allowed, and stop criterion. See blup entry in the reference manual for more details.

10.1.5 Multi-trait and multi-model example

```
P = Pedigree();
Ped.input("large.ped");

D = Data();
D.input("large.dat",
        "animal$ testdate line herd sex weight bf dg_test fce dg_farm");


M = Model("bf      = weight(line) line          animal",
          "dg_test =              line testdate animal",
          " fce    =              line testdate animal",
          "dg_farm = herd sex     line testdate animal");

M.covariate("weight");
M.variance("residual",
            2.8,    10.7,    0.0182,  0.5,
            10.7,   4160.0, -3.64,    1050.0,
            0.0182, -3.64,   0.026,  -1.3,
            0.5,    1050.0, -1.3,     1840.0);

M.variance("animal",P,
            1.2,    7.7,       0.003,  3.32,
            7.7,    2240.0,  -1.68,    470.0,
            0.003, -1.68,     0.014,  -0.72,
            3.32,   470.0,    -0.72,   410.0);
M.fitdata(D);
M.blup();
M.save("mme.out");
```

10.2 Linear Estimation

M.estimate(Kp) returns BLUE(K'b) where `Kp means K'`. For example, the following matvec script is saved in a file called `try3`.

```
P = Pedigree();
P.input("try.ped");

D = Data();
D.input("try.dat","animal$ herd _skip y");

M = Model();
M.equation("y = intercept herd animal");
M.variance("residual",2);
M.variance("animal",P,1);
M.fitdata(D);

Kp = [0,1,-1];
M.estimate(Kp)
```

Now, at the matvec prompt, I type

```
> input try3

        i = 1
        -1.69
```

10.3 Linear Hypothesis Test

M.contrast(Kp,m) is doing a linear hypothesis test that H0: K'b = m. Vector m is optional, the default is a vector of 0's. For example, the following matvec script is saved in a file called `try4`.

```
P = Pedigree();
P.input("try.ped");

D = Data();
D.input("try.dat","animal$ herd _skip y");

M = Model();
M.equation("y  = intercept herd animal");
M.variance("residual",2);
M.variance("animal",P,1);
M.fitdata(D);

Kp = [0,1,-1];
M.contrast(Kp)
```

Now, at the matvec prompt, I type

```
> input try4

           RESULTS FROM CONTRAST(S)
 ----------------------------------------------------------
  Contrast MME_addr    K_coef   Raw_data_code
  ---------------------------------------------
   1         2           1   herd:1
   1         3          -1   herd:2
         estimated value (K'b-M) = -1.69 +- 1.70425
            Prob(|t| > 0.991639) = 0.377506 (p_value)
 ----------------------------------------------------------
       0.377506
```

The last two statements in the above matvec script can be replaced by a single statement:

```
M.contrast("herd",[1,-1])
```

Any composite hypothesis tests require multi-row matrix Kp. For instance, the test H0: h1 = h2 = 0 can be accomplished in matvec as below:

```
M.contrast("herd",[1,0; 0,1])
```

10.4 Least Squares Means (lsmeans)

The least-squares means (lsmeans), also called population marginal means, are the expected values of class or subclass means that you would expect for a balanced design involving the class variable with all covariates at their mean level. A least-squares mean for a given level of a given model-term may not be estimable.

M.lsmeans("<termname-list>") outputs the least-squares means for each terms in termname-list. Here is the example saved in a file `try5`

```
D = Data();
D.input("try.dat","animal$ herd _skip y");

P = Pedigree();
P.input("try.ped");

M = Model();
M.equation("y = intercept  herd  animal");
M.variance("residual",2.0);
M.variance("animal",P,1.0);
M.fitdata(D);

M.lsmeans("herd");
```

Now, at the matvec prompt, I type

```
> input try5
                LEAST SQUARE MEANS
 ------------------------------------------------------------
     herd      lsmean  sqrt(Var(kb))     p_value  estimability
        1        7.42       1.4594   0.00705964
        2        9.11      1.36628   0.00262883
 ------------------------------------------------------------
 note 1: estimability = Max(mme*sweep(mme)*k - k);
         if it is non-zero, lsmean may or may not be estimable.
 note 2: sqrt(Var(kb)) is also called STD ERR (see SAS)
```

10.5 Variance Components Estimation (VCE)

There are two member functions of a model object for variance components estimation: M.vce_emreml() and M.vce_dfreml().

`M.vce_emreml(maxiter,interactive,stopval)` estimates the variance components for random_effects using EM_REML algorithm. Three arguments are optional. The first argument `maxiter` is the maximum number of iterations allowed with the default value 1000; if you are running matvec interactively, it allows you to add more iterations at run-time, thus a recommended `maxiter` value is 20 in order to monitor increment of restricted log-likelihood. The second argument `interactive` is the switch for interactive mode or non-interactive with the default value 1. The third argument `stopval` is the stop criterion with the default value 1.0e-5.

`M.vce_dfreml("df_method",maxiter,stopval)` estimates the variance components for random_effects using DF_REML algorithm. Three arguments are optional. The first argument `"<df_method>"` is the name of derivative-free based maximization maximum algorithm with default name `"powell"`; the second `maxiter` is the maximum number of iterations allowed with default value 500000; the third `stopval` is the stop criterion with default value 1.0e-5.

Both member functions return the BLUP solution which used the latest estimated variance components. Therefore, a semicolon (;) should always be used to end the functions to suppress printing of the length BLUP solution.

For example, the following matvec script is saved in a file called `try6`.

```
D = Data();
D.input("try.dat","animal$ herd _skip y");

P = Pedigree();
P.input("try.ped");

M = Model();
M.equation("y = intercept + herd + animal");
M.variance("residual",2.0);     // initial variance value
```

```
M.variance("animal",P,1.0);     // initial variance value
M.fitdata(D);

M.vce_emreml();   // correct answer: sigma_e = 2.69863,
                  //                 sigma_a = 2.20751
```

Now, at the matvec prompt, I type

```
> input try6
 iteration  sigma_1 ..... sigma_e  log_likelihood
    0:            1            2      -6.10457331
    1:      1.74667      2.87176     -6.481992428
    2:      1.65664      2.91128     -4.735431018
    3:      2.23916      2.68895     -5.763994369
    4:      2.22348       2.6937     -5.763092324
    5:      2.21558      2.69613     -5.762636885
    6:      2.21156      2.69737     -5.762405211
    7:      2.20951      2.69801     -5.762286907
    8:      2.20846      2.69833     -5.762226375
    9:      2.20792       2.6985     -5.762195371
   10:      2.20765      2.69859     -5.762179483
   11:      2.20751      2.69863     -5.762171339

       some extra information in the model
 -------------------------------------------------
 variance for animal =

                   Col 1
       Row 1     2.20751
 residual variance =

                   Col 1
       Row 1     2.69863

 MME dimension   : 8
 non-zeros in MME: 23

      basic statistics for dependent variables
  -------------------------------------------------
  trait-name        n         mean        std
          y         6      8.33333      2.06559
 ----------------------------------------------------
```

Chapter 11

Segregation and Linkage Analyses

11.1 Genotype Probability Computation

It is very difficult, in general, to compute the genotype probabilities for each member of a pedigree with loops . Van Arendonk et al. (1889), Janss et al. (1992), Fernando et al. (1993), and Wang et al. (1995) discussed and proposed an iterative algorithm to compute the genotype probabilities for large and complex livestock pedigrees. The proposed iterative algorithms are based on primary works by Murphy and Mutalik (1969), Elston and Stewart (1971), and Heuch and Li (1972).

The conditional probability that pedigree member i has genotype u_i given all the phenotypes $y_1, \ldots, y_n$ can be computed as

$$\Pr(u_i \mid y_1, \ldots, y_n) = \frac{a_i(u_i) g(y_i \mid u_i) \prod_{j \epsilon S_i} \{p_{ij}(u_i)\}}{\sum_{u_i} a_i(u_i) g(y_i \mid u_i) \prod_{j \epsilon S_i} \{p_{ij}(u_i)\}} \tag{11.1}$$

where $a_i(u_i)$ is the joint probability of phenotypes of members anterior to i and of genotype u_i for i, $g(y_i \mid u_i)$ is the penetrance function, and $\Pi_{j\epsilon S_i}\{p_{ij}(u_i)\}$ is the conditional probability of phenotypes of members posterior to i, given i has genotype u_i.

The anterior probability, $a_i(u_i)$, and posterior probability, $p_{ij}(u_i)$, can be computed as

$$\begin{aligned} a_i(u_i) &= \sum_{u_m} \{a_m(u_m) g(y_m \mid u_m) \prod_{j\epsilon S_m, j\neq f} p_{mj}(u_m) \\ &\quad \times \sum_{u_f} \{a_f(u_f) g(y_f \mid u_f) \prod_{j\epsilon S_f, j\neq m} p_{fj}(u_f) \\ &\quad \times tr(u_i \mid u_m, u_f) \end{aligned} \tag{11.2}$$

$$\times \prod_{j \epsilon C_{mf}, j \neq i} \left[\sum_{u_j} tr(u_j \mid u_m, u_f) g(y_j \mid u_j) \prod_{k \epsilon S_j} p_{jk}(u_j) \right] \} \}$$

and

$$\begin{aligned} p_{ij}(u_i) &= \sum_{u_j} \{ a_j(u_j) g(y_j \mid u_j) \prod_{k \epsilon S_j, k \neq i} p_{jk}(u_j) \\ &\times \prod_{k \epsilon C_{ij}} \left[\sum_{u_k} tr(u_k \mid u_i, u_j) g(y_k \mid u_k) \prod_{l \epsilon S_k} p_{kl}(u_k) \right] \} \end{aligned} \tag{11.3}$$

where $C_{ij}(C_{mf})$ is the set of offspring of parents i and j (m and f).

The step-by-step iterative algorithm is given below:

1. For each member i:
   (a) Initialize anterior values: set anterior values equal to the population genotype frequencies and set the anterior log scaling factor equal to zero,
   (b) Initialize posterior values: set posterior values equal to unity and set the posterior log scaling factor equal to zero, and
   (c) Compute penetrance values and the corresponding log scaling factor.
2. For each connector i:
   (a) For families in which i is an offspring, compute its anterior value $a_i(u_i)$ using current values of required quantities.
   (b) For families in which i is a parent, compute its posterior value through each mate j, p_{ij}, using current values of the required quantities.
3. Repeat step 2 (usually less than 10 times) until each of the anterior and posterior values for each member has converged.

11.2 Genetic Mapping

not completed yet.

Chapter 12

Statistical Distributions

12.1 Introduction

An object of StatDist class can be created as below:

```
D = StatDist("name",parameter-list)
```

where

1. `name` is one of the followings: Uniform, Normal, ChiSquare, t, F, Gamma, Exponential, Beta, DiscreteUniform, Poisson, Binomial, NegativeBinomial, Geometric. Hypergeometric, Logistic, Laplace, InverseGaussian, Cauchy, Weibull, Erlang will be added into the list.

2. `parameter-list` is a list of scalar values required by the corresponding distribution.

Note that some distributions have their non-centrality parameters. The non-centrality parameter is always the last one in the parameter list. It is automatically set to zero if users don't provide it explicitly.

There are nine member functions available now.

pdf D.pdf(x) returns the probability density function values of x which could be a vector or matrix.

cdf D.cdf(x) returns the cumulative distribution function values of x which could be a vector or matrix

mgf D.mgf(t) returns the moment-generating function values of t which could be a vector or matrix.

quantile x = D.quantile(p) is the inverse function of D.cdf(x), where p could be a vector or matrix.

nonct D.nonct(cv,p) returns non-centrality value given the critical value (cv) and p value (cdf). Both cv and p could be either vector or matrix as long as the sizes are the same.

sample D.sample(), D.sample(n), and D.sample(m,n) return a random scalar or a vector of size n or a matrix of size m by n.

parameter D.parameter(k) returns the k'th parameter of the distribution. D.parameter(k,x) re-set the k'th parameter of the distribution by new value x.

mean D.mean() returns the first moment (expected value) of the distribution.

variance D.variance() returns the second central moment (variance) of the distribution.

Here are two examples:

```
> N = StatDist("Normal",0, 10);
> B = StatDist("Binomial",2, 0.1);
```

12.2 Continuous Distribution

The distributions for continuous random variables are Uniform, Normal, ChiSquare, t, F, Gamma, Beta, Exponential, and LogNormal.

12.2.1 Normal distribution

Definition

The random variable X has a *normal distribution* if its probability density function (pdf) is defined by

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp(-\frac{(x-\mu)^2}{2\sigma^2}), \quad -\infty < x < \infty \tag{12.1}$$

where μ (real) and σ^2 (real) are the parameters with their range $-\infty < \mu < \infty, \sigma^2 > 0$. In short, we say $X \sim N(\mu, \sigma^2)$. The normal distribution with $\mu = 0$ and $\sigma^2 = 1$ is known as the standard normal distribution $N(0, 1)$.

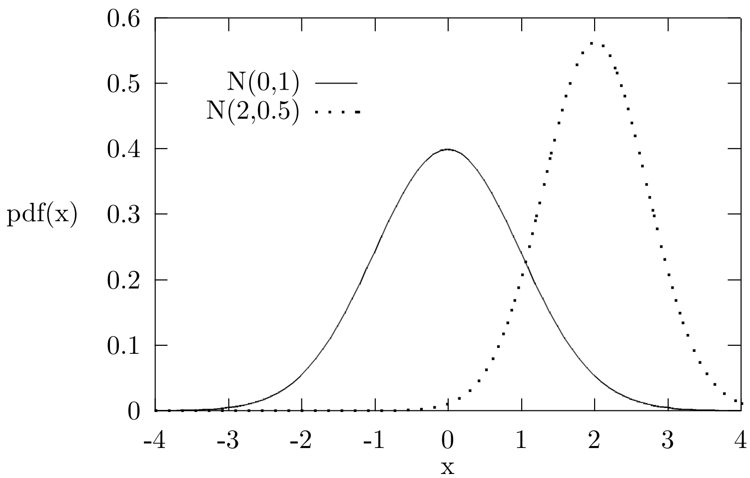
The graph of normal pdf function for two sets of parameters is shown in Fig. 12.1. The matvec script used to draw this graph is on page 68.

The normal distribution is certainly the most important distribution in statistical data analyses since many performance measurements such as milkyield have (approximate) normal distributions.

Properties

1. The normal distribution is scalable. If $Z \sim N(0, 1)$ then $\sigma Z + \mu \sim N(\mu, \sigma^2)$, and conversely if $X \sim N(\mu, \sigma^2)$, then $\frac{X-\mu}{\sigma} \sim N(0, 1)$.

Figure 12.1: Normal Probability Density Function (pdf)



2. the distribution is symmetric
3. its moment generation function (mgf) is

$$M(t) = \exp(\mu t + \sigma^2 t^2/2), \quad -\infty < t < \infty$$

4. the cumulative distribution function (cdf) for $X \sim N(0,1)$ is

$$\begin{aligned} \Phi(x) &= \int_{-\infty}^{x} \frac{1}{\sqrt{2\pi}} \exp(-\frac{w^2}{2})\, dw \\ &= \begin{cases} \frac{1+\mathrm{erf}(x/\sqrt{2})}{2} & x \geq 0 \\ \frac{1-\mathrm{erf}(-x/\sqrt{2})}{2} & x < 0 \end{cases} \end{aligned}$$

5. $\mathrm{E}(X) = \mu, \mathrm{Var}(X) = \sigma^2$.
6. the median $= \mu$.
7. if $X_1, X_2, \ldots, X_n$ is a sample from $N(\mu, \sigma^2)$, let

$$\bar{X} = \frac{\sum_{i=1}^{n} X_i}{n}, \text{ and } s^2 = \frac{\sum_{i=1}^{n} (X_i - \bar{X})^2}{n-1}$$

then $\bar{X} \sim N(\mu, \frac{\sigma^2}{n}), (n-1)\frac{s^2}{\sigma^2} \sim \chi^2(n-1)$, and both are independent.

matvec interface

An object of $N(\mu, \sigma^2)$ can be created by

```
D = StatDist("Normal",mu,sigma2);
D = StatDist("Normal");            // standard normal N(0,1)
```

matvec provided several standard member functions to allow user to access most of properties and functions of $N(\mu, \sigma^2)$:

pdf D.pdf(x) returns the probability density function (pdf) values of x which could be a vector or matrix.

cdf D.cdf(x) returns the cumulative distribution function (cdf) values of x which could be a vector or matrix

mgf D.mgf(t) returns the moment-generating function (mgf) values of t which could be a vector or matrix.

quantile D.quantile(p) is the inverse function of D.cdf(x), where p could be a vector or matrix. That is if p = D.cdf(x), then x = D.quantile(p).

sample D.sample(), D.sample(n), and D.sample(m,n) return a random scalar or a vector of size n or a matrix of size m by n.

parameter D.parameter(1) returns μ and D.parameter(2) returns σ^2.

mean D.mean() returns the expected value of $X \sim N(\mu, \sigma^2)$.

variance D.variance() returns the variance of $X \sim N(\mu, \sigma^2)$.

Examples

```
> D = StatDist("Normal",2, 10)
        NormalDist(2,10)
> D.mean()
        2
> D.pdf([-2,0,2])
                     Col 1        Col 2        Col 3
        Row 1    0.0566858     0.103288     0.126157
> D.cdf([-3.20148,2,7.20148])
                     Col 1        Col 2        Col 3
        Row 1    0.0500001     0.500000     0.950000
> D.quantile([0.05,0.5,0.95])
                     Col 1        Col 2        Col 3
        Row 1     -3.20148      2.00000      7.20148
> D.sample(2,3)
                     Col 1        Col 2        Col 3
        Row 1     -1.26660      2.85846      5.64868
        Row 2      3.51262     -1.97644      3.44860
```

12.2.2 Uniform distribution

Definition

The random variable X has a *uniform distribution* if its probability density function (pdf) is defined by

$$f(x) = \frac{1}{b-a}, \quad a \leq x \geq b \tag{12.2}$$

where a (real) and b (real) are the parameters with their range $-\infty < a < b < \infty$. In short, we say $X \sim U(a,b)$. The uniform distribution with $a = 0$ and $b = 1$ is known as the standard uniform distribution $U(0,1)$. One of interesting example is that if D is an object of StatDist, then D.cdf(x) is distributed as $U(0,1)$.

Properties

1. $\mathrm{E}(X) = \frac{a+b}{2}, \mathrm{Var}(X) = \frac{(b-a)^2}{12}$.
2. its moment generation function is

$$M(t) = \frac{e^{tb} - e^{ta}}{t(b-a)}, \quad t \neq 0; M(0) = 1$$

matvec interface

An object of $U(a,b)$ can be created by

```
D = StatDist("Uniform",a,b);
D = StatDist("Uniform");            // standard uniform U(0,1)
```

matvec provided several standard member functions to allow user to access most of properties and functions of $U(a,b)$:

pdf D.pdf(x) returns the probability density function (pdf) values of x which could be a vector or matrix.

cdf D.cdf(x) returns the cumulative distribution function (cdf) values of x which could be a vector or matrix

mgf D.mgf(t) returns the moment-generating function (mgf) values of t which could be a vector or matrix.

quantile D.quantile(p) is the inverse function of D.cdf(x), where p could be a vector or matrix. That is if p = D.cdf(x), then x = D.quantile(p).

sample D.sample(), D.sample(n), and D.sample(m,n) return a random scalar or a vector of size n or a matrix of size m by n.

parameter D.parameter(1) returns a and D.parameter(2) returns b.

mean D.mean() returns the expected value of $X \sim U(a,b)$.

variance D.variance() returns the variance of $X \sim U(a,b)$.

Examples

```
> D = StatDist("Uniform",1, 4)
        UniformDist(1,4)
> D.variance()
        0.75
> D.pdf([1,2,3,4])
                  Col 1        Col 2        Col 3        Col 4
     Row 1     0.333333     0.333333     0.333333     0.333333
> D.cdf([1,2,3,4])
                 Col 1        Col 2        Col 3        Col 4
     Row 1      0.00000     0.333333     0.666667      1.00000
> D.quantile([0,1/3,2/3,1])
                  Col 1        Col 2        Col 3        Col 4
     Row 1      1.00000      2.00000      3.00000      4.00000
> U.sample(1,3)
                     Col 1        Col 2        Col 3
        Row 1      3.59627      1.36787      2.17816
```

12.2.3 χ^2 distribution

Definition

The random variable X has a non-central χ^2 distribution if its probability density function (pdf) is defined by

$$f(x) = \frac{\exp(-(x+\lambda)/2)}{2^{\frac{1}{2}r}} \sum_{j=0}^{\infty} \frac{x^{r/2+j-1}\lambda^j}{\Gamma(r/2+j)2^{2j}j!}, \quad 0 \leq x < \infty \tag{12.3}$$

where r (integer) and λ (real) are the parameters with their ranges $r \geq 1, \lambda \geq 0$. The parameter r is commonly called the degrees of freedom and λ non-centrality parameter. In short, we say $X \sim \chi^2(r, \lambda)$.

Alternatively, If $Z_1, Z_2, \ldots, Z_r$ are independent and distributed as $N(\delta_i, 1)$ then the random variable

$$X = \sum_{i=1}^{r} Z^2$$

is called the non-central χ^2 distribution with r degrees of freedom and non-centrality parameter $\lambda = \sum_{i=1}^{r} \delta_i$.

When $\lambda = 0$, the p.d.f of $\chi^2(r, \lambda)$ reduces to

$$f(x) = \frac{1}{\Gamma(r/2)2^{r/2}} x^{r/2-1} e^{-x/2}, \tag{12.4}$$

we say $X \sim \chi^2(r)$ which is commonly called (central) χ^2 distribution.

The central χ^2 distribution is the special case of Gamma distribution: $\chi^2(r) = \Gamma(r/2, 2)$

For example, if $X_1, X_2, \ldots, X_n$ is a sample from $N(\mu, \sigma^2)$, then

$$(n-1)\frac{s^2}{\sigma^2} \sim \chi^2(n-1)$$

and is independent of the sample mean $\bar{X}$.

Properties

1. its moment generation function is

$$M(t) = (1-2t)^{-r/2} \exp(\frac{\lambda t}{1-2t}), \quad t < \frac{1}{2}$$

2. $\mathrm{E}(X) = r + \lambda, \mathrm{Var}(X) = 2(r + 2\lambda)$.
3. If $\chi^2(r_1, \lambda_1)$ and $\chi^2(r_2, \lambda_2)$ are independent, then $\chi^2(r_1, \lambda_1) + \chi^2(r_2, \lambda_2) = \chi^2(r_1 + r_2, \lambda_1 + \lambda_2)$

matvec interface

An object of $\chi^2(r, \lambda)$ can be created by

```
D = StatDist("ChiSquare",r,lambda);
D = StatDist("ChiSquare",r);
```

matvec provided several standard member functions to allow user to access most of properties and functions of $\chi^2(r, \lambda)$:

pdf D.pdf(x) returns the probability density function (pdf) values of x which could be a vector or matrix.

cdf D.cdf(x) returns the cumulative distribution function (cdf) values of x which could be a vector or matrix

mgf D.mgf(t) returns the moment-generating function (mgf) values of t which could be a vector or matrix.

quantile D.quantile(p) is the inverse function of D.cdf(x), where p could be a vector or matrix. That is if p = D.cdf(x), then x = D.quantile(p).

nonct D.nonct(cv,p) returns non-centrality value given the critical value (cv) and p value (cdf). Both cv and p could be either vector or matrix as long as the sizes are the same.

sample D.sample(), D.sample(n), and D.sample(m,n) return a random scalar or a vector of size n or a matrix of size m by n.

parameter D.parameter(1) returns r, degree of freedom.

mean D.mean() returns the expected value of $X \sim \chi^2(r, \lambda)$.

variance D.variance() returns the variance of $X \sim \chi^2(r, \lambda)$.

Examples

```
> D = StatDist("ChiSquare",10,0.1)
        ChiSquareDist(10,0.1)
> D.mean()
        10.1
> D..sample(1000).mean()
        10.0323
> D.pdf(5)

 ***ERROR***
  ChiSquareDist::pdf():  not available yet: non-centrality
> D.cdf([0,1,10])
                 Col 1         Col 2         Col 3
     Row 1     0.00000  0.000164396      0.550770
> D.quantile([0,0.000164396,0.550770])
                 Col 1         Col 2         Col 3
     Row 1     0.00000      1.00000      10.0000
> D.nonct(10,0.550770)
        0.100003
> D.sample()
        5.15093
```

12.2.4 t distribution

Definition

The random variable X has a *non-central t distribution* if its probability density function (pdf) is defined by

$$f(x) = \frac{\exp(\delta^2/2)\Gamma((r+1)/2)}{\sqrt{\pi r}\Gamma(r/2)}(\frac{r}{r+x^2})^{(r+1)/2} \quad (12.5)$$

$$\times \sum_{j=0}^{\infty} \frac{\Gamma((r+j+1)/2)}{j!\Gamma((r+1)/2)} \left[\frac{x\delta\sqrt{2}}{\sqrt{r+x^2}}\right]^j, \quad -\infty < x < \infty \quad (12.6)$$

where r (integer) and δ (real) are the parameters with their ranges $r \geq 1, -\infty < \delta < \infty$; r is commonly called the degrees of freedom and δ non-centrality parameter. In short, we say $X \sim t(r, \delta)$.

Alternatively, If $Z \sim N(\delta, 1)$ and $U \sim \chi^2(r)$ are independent then the random variable

$$X = \frac{Z}{\sqrt{U/r}}$$

is called the non-central t distribution with r degrees of freedom and non-centrality parameter δ.

When $\delta = 0$, the p.d.f of $t(r, \delta)$ reduces to

$$f(x) = \frac{\Gamma((r+1)/2)}{\sqrt{\pi r}\Gamma(r/2)} (\frac{r}{r+x^2})^{(r+1)/2} \tag{12.7}$$

we say $X \sim t(r)$ which is commonly called (central) t distribution.

For example, if $X_1, X_2, \ldots, X_n$ is a random sample from $N(\mu, \sigma^2)$, then

$$\frac{\bar{X} - \mu}{\sigma \sqrt{n}} \sim N(0,1), (n-1)\frac{s^2}{\sigma^2} \sim \chi^2(n-1),$$

and both are independent. Thus,

$$\frac{\bar{X} - \mu}{s/\sqrt{n}} \sim t(n-1)$$

Because

$$\frac{\bar{X}}{\sigma \sqrt{n}} \sim N(\mu, 1)$$

Thus,

$$\frac{\bar{X}}{s/\sqrt{n}} \sim t(n-1, \frac{\mu}{\sigma\sqrt{n}})$$

Properties

1. $\mathrm{E}(X) = (r/2)^{1/2}\frac{\Gamma((r-1)/2)}{\Gamma(r/2)}\delta, \mathrm{Var}(X) = \frac{r}{r-2}(1+\delta^2) - \mathrm{E}^2(X)$.

matvec interface

An object of $t(r, \delta)$ can be created by

```
> D = StatDist("t",r,delta)
> D = StatDist("t",r)
```

matvec provided several standard member functions to allow user to access most of properties and functions of $t(r, \delta)$:

pdf D.pdf(x) returns the probability density function (pdf) values of x which could be a vector or matrix.

cdf D.cdf(x) returns the cumulative distribution function (cdf) values of x which could be a vector or matrix

mgf D.mgf(t) returns the moment-generating function (mgf) values of t which could be a vector or matrix.

quantile D.quantile(p) is the inverse function of D.cdf(x), where p could be a vector or matrix. That is if p = D.cdf(x), then x = D.quantile(p).

nonct D.nonct(cv,p) returns non-centrality value given the critical value (cv) and p value (cdf). Both cv and p could be either vector or matrix as long as the sizes are the same.

sample D.sample(), D.sample(n), and D.sample(m,n) return a random scalar or a vector of size n or a matrix of size m by n.

parameter D.parameter(1) returns r, degree of freedom; whereas D.parameter(2) returns δ, non-centrality parameter.

mean D.mean() returns the expected value of $X \sim t(r, \delta)$.

variance D.variance() returns the variance of $X \sim t(r, \delta)$.

Examples

```
> D = StatDist("t",15,2)
        tDist(15,2)
> D.mean()
        2.10746
> D.variance()
        1.32782
> D.pdf(2)

 ***ERROR***
 tDist::pdf(): non-centrality: not available
> D.cdf([-1,0,1,5])
                  Col 1       Col 2       Col 3       Col 4
     Row 1  0.00165624   0.0227501    0.158591    0.986127
> D.quantile([0.00165624,0.0227501, 0.158591,0.986127])
                  Col 1        Col 2       Col 3       Col 4
     Row 1    -1.00000 -0.000122502    0.999999     5.00002
> D.nonct(5,0.986127)
        1.99998
> D.sample(3)
      i = 1        i = 2        i = 3
   0.952105      2.99387     0.707365
```

12.2.5 F distribution

Definition

If $U_1 \sim \chi^2(r_1, \delta)$ and $U_2 \sim \chi^2(r_2)$ are independent then the random variable

$$X = \frac{U_1/r_1}{U_2/r_2}$$

is called the non-central F distribution with $r1$ (integer) and $r2$ (integer) degrees of freedom and non-centrality parameter δ (real), denoted by $F(r_1, r_2, \delta)$ where $r_1, r_2 \geq 1$ and $\delta \geq 0$.

Properties

1. $\mathrm{E}(X) = \frac{r_2(r_1+\delta)}{r_1(r_2-2)}$, for $r_2 > 2$

2. $\mathrm{Var}(X) = 2(r_2/r_1)^2 \frac{(r_1+\delta)^2+(r_1+2\delta)(r_2-2)}{(r_2-2)^2(r_2-4)}$ for $r_2 > 4$

matvec interface

An object of $F(r_2, r_2, \delta)$ can be created by

```
D = StatDist("F",r1,r2,delta);
D = StatDist("F",r1,r2);
```

matvec provided several standard member functions to allow user to access most of properties and functions of $F(r_1, r_2, \delta)$:

pdf D.pdf(x) returns the probability density function (pdf) values of x which could be a vector or matrix.

cdf D.cdf(x) returns the cumulative distribution function (cdf) values of x which could be a vector or matrix

mgf D.mgf(t) returns the moment-generating function (mgf) values of t which could be a vector or matrix.

quantile D.quantile(p) is the inverse function of D.cdf(x), where p could be a vector or matrix. That is if p = D.cdf(x), then x = D.quantile(p).

nonct D.nonct(cv,p) returns non-centrality value given the critical value (cv) and p value (cdf). Both cv and p could be either vector or matrix as long as the sizes are the same.

sample D.sample(), D.sample(n), and D.sample(m,n) return a random scalar or a vector of size n or a matrix of size m by n.

parameter D.parameter(1) returns r_1, D.parameter(2) returns r_2, and D.parameter(3) returns δ.

mean D.mean() returns the expected value.

variance D.variance() returns the variance.

Examples

```
> D = StatDist("F",5,100,0.5)
        FDist(5,100,0.5)
> D.mean()
        1.12245
> D.sample(1000).mean()
        1.11399
```

```
> D.variance()
       0.536452
> D.pdf(2)

 ***ERROR***
 FDist::pdf():  not available yet: non-centrality
> D.cdf([0.1,1,3])
                    Col 1        Col 2        Col 3
       Row 1   0.00642900     0.520762     0.976739
> D.quantile([0.00642900,0.520762,0.976739])
                    Col 1        Col 2        Col 3
       Row 1     0.100000      1.00000      3.00000
> D.mgf(2)

 ***ERROR***
 FDist:mgf(): not available yet
> D.nonct(3,0.95)
       1.626
```

12.2.6 Gamma distribution

Definition

The random variable X has a *gamma distribution* if its probability density function is defined by

$$f(x) = \frac{1}{\Gamma(\alpha)\theta^{\alpha}} x^{\alpha-1} e^{-x/\theta}, \quad 0 \leq x < \infty. \tag{12.8}$$

where α (real) and θ (real) are the parameters with their ranges $\alpha, \theta > 0$. In short, we say $X \sim Gamma(\alpha, \theta)$.

Properties

1. moment generating function

$$M(t) = \frac{1}{(1-\theta t)^{\alpha}}, \quad t < 1/\theta$$

2. $\mathrm{E}(X) = \alpha\theta, \quad \mathrm{Var}(X) = \alpha\theta^2$

matvec interface

An object of $Gamma(\alpha, \theta)$ can be created by

```
D = StatDist("Gamma",alpha,theta);
```

matvec provided several standard member functions to allow user to access most of properties and functions of $Gamma(\alpha, \theta)$:

pdf D.pdf(x) returns the probability density function (pdf) values of x which could be a vector or matrix.

cdf D.cdf(x) returns the cumulative distribution function (cdf) values of x which could be a vector or matrix

mgf D.mgf(t) returns the moment-generating function (mgf) values of t which could be a vector or matrix.

quantile D.quantile(p) is the inverse function of D.cdf(x), where p could be a vector or matrix. That is if p = D.cdf(x), then x = D.quantile(p).

sample D.sample(), D.sample(n), and D.sample(m,n) return a random scalar or a vector of size n or a matrix of size m by n.

parameter D.parameter(1) returns α, D.parameter(2) returns θ.

mean D.mean() returns the expected value.

variance D.variance() returns the variance.

Examples

```
> D = StatDist("Gamma",2,20)
        GammaDist(2,20)
> D.mean()
        40
> D.sample(1000).mean()
        40.3099
> D.pdf([1,10,100])
                     Col 1         Col 2         Col 3
        Row 1   0.00237807     0.0151633    0.00168449
> D.cdf([1,10,100])
                     Col 1         Col 2         Col 3
        Row 1   0.00120910     0.0902040      0.959572
> D.quantile([0.00120910,0.0902040,0.959572])
                     Col 1         Col 2         Col 3
        Row 1     0.999998       10.0000       99.9998
```

12.2.7 Exponential distribution

Definition

The random variable X has a *exponential distribution* if its probability density function is defined by

$$f(x) = \frac{1}{\theta}e^{-x/\theta}, \quad 0 \leq x < \infty. \tag{12.9}$$

where θ (real) is the parameter with its range $\theta > 0$. In short, we say $X \sim E(\theta)$. The exponential distribution with $\theta = 1$ is known as the standard exponential distribution.

Properties

1. moment generating function

$$M(t) = \frac{1}{1-\theta t}, \quad t < \frac{1}{\theta}$$

2. $\mathrm{E}(X) = \theta, \mathrm{Var}(X) = \theta^2$

3. the cumulative distribution function (cdf) is defined

$$F(x) = \begin{cases} 0 & -\infty < x < 0 \\ 1 - e^{-x/\theta} & 0 \leq x < \infty \end{cases}$$

4. the median m is found by solving $F(m) = 0.5$. That is $m = \theta \log(2)$. because $\log(2) < 1$. Thus the median is always less than the mean (θ).

matvec interface

An object of $E(\theta)$ can be created by

```
D = StatDist("Exponential",theta);
D = StatDist("Exponential");
```

matvec provided several standard member functions to allow user to access most of properties and functions of $E(\theta)$:

pdf D.pdf(x) returns the probability density function (pdf) values of x which could be a vector or matrix.

cdf D.cdf(x) returns the cumulative distribution function (cdf) values of x which could be a vector or matrix.

mgf D.mgf(t) returns the moment-generating function (mgf) values of t which could be a vector or matrix.

quantile D.quantile(p) is the inverse function of D.cdf(x), where p could be a vector or matrix. That is if p = D.cdf(x), then x = D.quantile(p).

sample D.sample(), D.sample(n), and D.sample(m,n) return a random scalar or a vector of size n or a matrix of size m by n.

parameter D.parameter(1) returns θ.

mean D.mean() returns the expected value.

variance D.variance() returns the variance.

Examples

```
> D = StatDist("Exponential",10)
        ExponentialDist(10)
> D.sample(1000).mean()
        9.69597
> D.pdf([0,2,10])
                     Col 1         Col 2         Col 3
        Row 1     0.100000    0.0818731    0.0367879
> D.cdf([0,2,10])
                     Col 1         Col 2         Col 3
        Row 1      0.00000     0.181269     0.632121
> D.quantile([0,0.181269,0.632121])
                     Col 1         Col 2         Col 3
        Row 1  5.63450e-07      2.00000      10.0000
```

12.2.8 Beta distribution

Definition

The random variable X has a *non-central beta distribution* if its probability density function is defined by

$$f(x) = ... \tag{12.10}$$

where α (real), β (real) and λ (real) are the parameters with their ranges $\alpha, \beta > 0$ and $\lambda \geq 0$. In short, we say $X \sim Beta(\alpha, \beta, \lambda)$.

If $\lambda = 0$, then its pdf reduces to

$$f(x) = \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} x^{\alpha-1}(1-x)^{\beta-1}, \quad 0 < x < 1 \tag{12.11}$$

we say $X \sim Beta(\alpha, \beta)$ which is commonly called (central) *Beta* distribution.

Alternatively, if $X_1 \sim \chi^2(r_1, \lambda)$ and $X_2 \sim \chi^2(r_2)$ are independent, then the ratio

$$X = \frac{X_1}{X_1 + X_2} \tag{12.12}$$

is defined as a non-central beta distribution $Beta(r_1/2, r_2/2, \lambda)$

Properties

1. for $X \sim Beta(\alpha, \beta), \mathrm{E}(X) = \frac{\alpha}{\alpha+\beta}, \mathrm{Var}(X) = \frac{\alpha\beta}{(\alpha+\beta+1)(\alpha+\beta)^2}$.

matvec interface

An object of $Beta(\alpha, \beta, \lambda)$ can be created by

```
D = StatDist("Beta",alpha,beta,lambda);
D = StatDist("Beta",alpha,beta);
```

matvec provided several standard member functions to allow user to access most of properties and functions of $Beta(\alpha, \beta, \lambda)$:

pdf D.pdf(x) returns the probability density function (pdf) values of x which could be a vector or matrix.

cdf D.cdf(x) returns the cumulative distribution function (cdf) values of x which could be a vector or matrix

mgf D.mgf(t) returns the moment-generating function (mgf) values of t which could be a vector or matrix.

quantile D.quantile(p) is the inverse function of D.cdf(x), where p could be a vector or matrix. That is if p = D.cdf(x), then x = D.quantile(p).

nonct D.nonct(cv,p) returns non-centrality value given the critical value (cv) and p value (cdf). Both cv and p could be either vector or matrix as long as the sizes are the same.

sample D.sample(), D.sample(n), and D.sample(m,n) return a random scalar or a vector of size n or a matrix of size m by n.

parameter D.parameter(1) returns α, D.parameter(2) returns β, and D.parameter(3) returns λ.

mean D.mean() returns the expected value.

variance D.variance() returns the variance.

Examples

```
> C = StatDist("Beta",2,3)
        BetaDist(2,3,0)
> D.mean()
        0.4
> D.sample(1000).mean()
        0.407094
> D.pdf([0.01,0.5,0.9])
                     Col 1        Col 2        Col 3
        Row 1     0.117612      1.50000     0.108000
> D.cdf([0.01,0.5,0.9])
                     Col 1        Col 2        Col 3
        Row 1  0.000592030     0.687500     0.996300
> D.quantile([0.000592030,0.687500,0.996300])
                     Col 1        Col 2        Col 3
        Row 1   0.00999928     0.500001     0.900001
```

12.2.9 Log normal distribution

Definition

The random variable X has a *lognormal distribution* if its probability density function (pdf) is defined by

$$f(x) = [(x-\theta)\sqrt{(2\pi)}\sigma]^{-1} \exp(-\frac{\log(x-\theta)-\mu)^2}{2\sigma^2}), \quad x > \theta \tag{12.13}$$

where μ (real), σ^2 (real) and θ (real) are the parameters with their ranges $-\infty < \mu, \theta < \infty$ and $\sigma^2 > 0$. In short, we say $X \sim lognormal(\mu, \sigma^2, \theta)$.

Alternatively, if $\log(X-\theta) \sim N(\mu, \sigma^2)$, then we say X has a $lognormal(\mu, \sigma^2, \theta)$ distribution.

The lognormal distribution with $\theta = 0$ is known as two-parameter lognormal distribution $lognormal(\mu, \sigma^2)$.

Properties

1. $\mathrm{E}(X) = \exp(\mu + \sigma^2/2) + \theta, \mathrm{Var}(X) = \exp(2\mu + 2\sigma^2) - \exp(2\mu + \sigma^2) + 2\theta^2$.
2. $\mathrm{mode}(X) = \exp(\mu - \sigma^2) + \theta$
3. $\mathrm{median}(X) = \exp(\mu) + \theta$

matvec interface

An object of $lognormal(\mu, \sigma^2, \theta)$ can be created by

```
D = StatDist("LogNormal",mu,sigma2,theta);
D = StatDist("LogNormal",mu,sigma2);
```

matvec provided several standard member functions to allow user to access most of properties and functions of $lognormal(\mu, \sigma^2, \theta)$:

pdf D.pdf(x) returns the probability density function (pdf) values of x which could be a vector or matrix.

cdf D.cdf(x) returns the cumulative distribution function (cdf) values of x which could be a vector or matrix

mgf D.mgf(t) returns the moment-generating function (mgf) values of t which could be a vector or matrix.

quantile D.quantile(p) is the inverse function of D.cdf(x), where p could be a vector or matrix. That is if p = D.cdf(x), then x = D.quantile(p).

sample D.sample(), D.sample(n), and D.sample(m,n) return a random scalar or a vector of size n or a matrix of size m by n.

parameter D.parameter(1) returns μ, D.parameter(2) returns σ^2, and D.parameter(3) returns θ.

mean D.mean() returns the expected value.

variance D.variance() returns the variance.

Examples

```
> D = StatDist("LogNormal",1,3,2);
        LogNormalDist(1,3,2)
> D.mean()
        14.1825
> D.sample(1000).mean()
        14.087
> D.pdf([2.01,10,30])
                     Col 1        Col 2        Col 3
        Row 1     0.122530    0.0237094   0.00332270
> D.cdf([2.01,10,30])
                     Col 1        Col 2        Col 3
        Row 1  0.000605776     0.733429     0.910929
> D.quantile([0.000605776,0.733429,0.910929])
                     Col 1        Col 2        Col 3
        Row 1      2.01000      10.0000      30.0000
```

12.3 Discrete Distribution

The distributions for common discrete random variables are Uniform(discrete), Binomial, Poisson, Geometric, Negative Binomial and Hypergeometric.

12.3.1 Binomial distribution

Definition

The random variable X has a *binomial distribution* if its probability density function (pdf) is defined by

$$f(x) = \frac{n!}{(n-x)!x!} p^x (1-p)^{n-x}, \quad x = 0, 1, 2, \ldots, n \tag{12.14}$$

where p (real) and n (integer) are the parameters with their ranges $0 \le p \le 1$, and $n \ge 1$. In short, we say $X \sim b(n, p)$.

The graph of the pdf of $b(25, 0.15)$ is shown in Fig. 12.2. The matvec script used to draw this graph is on page 68.

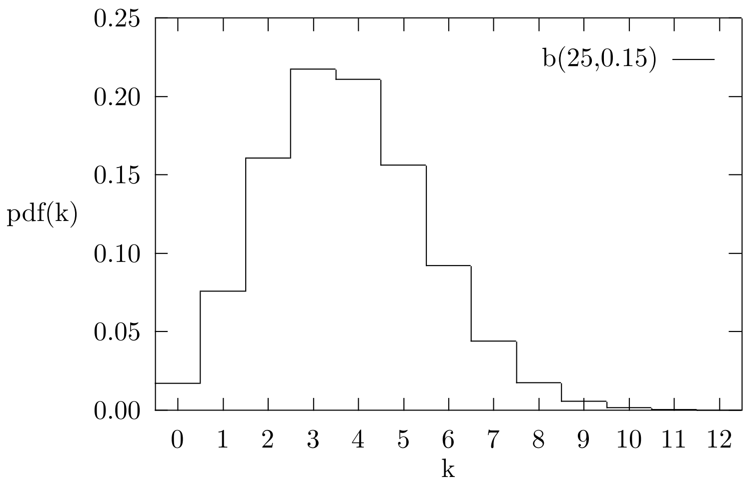
Properties

1. its moment generation function is

$$M(t) = (1 - p + pe^t)^n$$

2. $\mathrm{E}(X) = np, \mathrm{Var}(X) = np(1-p)$.

Figure 12.2: Binomial Probability Density Function (pdf)



matvec interface

An object of $b(n, p)$ can be created by

```
D = StatDist("Binomial",n,p);
```

matvec provided several standard member functions to allow user to access most of properties and functions of $b(n, p)$:

pdf D.pdf(x) returns the probability density function (pdf) values of x which could be a vector or matrix.

cdf D.cdf(x) returns the cumulative distribution function (cdf) values of x which could be a vector or matrix

mgf D.mgf(t) returns the moment-generating function (mgf) values of t which could be a vector or matrix.

quantile D.quantile(p) is the inverse function of D.cdf(x), where p could be a vector or matrix. That is if p = D.cdf(x), then x = D.quantile(p).

sample D.sample(), D.sample(n), and D.sample(m,n) return a random scalar or a vector of size n or a matrix of size m by n.

parameter D.parameter(1) returns n, number of Bernoulli trials; whereas D.parameter(2) returns p, the probability of success in a Bernoulli trial.

mean D.mean() returns the expected value.

variance D.variance() returns the variance.

Examples

```
> D = StatDist("Binomial",4,0.2)
        BinomialDist(4,0.2)
> D.mean()
        0.8
> D.sample(1000).mean()
        0.764
> D.pdf([0,1,4])
                     Col 1        Col 2        Col 3
        Row 1     0.409600     0.409600   0.00160000
> D.cdf([0,1,4])
                     Col 1        Col 2        Col 3
        Row 1     0.409600     0.819200      1.00000
> D.quantile(0.5)

 ***ERROR***
  BinomialDist::quantile(): not available
```

12.3.2 Poisson distribution

Definition

The random variable X has a *Poisson distribution* if its probability density function (pdf) is defined by

$$f(x) = \frac{\lambda^x e^{-\lambda}}{x!}, \quad x = 0, 1, 2, \ldots, \tag{12.15}$$

where λ (real) is the parameter with its range $\lambda > 0$. In short, we say $X \sim P(\lambda)$.

Properties

1. its moment generation function is

$$M(t) = e^{\lambda(e^t - 1)}$$

2. $\mathrm{E}(X) = \lambda, \mathrm{Var}(X) = \lambda$.

matvec interface

An object of $P(\lambda)$ can be created by

```
D = StatDist("Poisson",lambda);
```

matvec provided several standard member functions to allow user to access most of properties and functions of $P(\lambda)$:

pdf D.pdf(x) returns the probability density function (pdf) values of x which could be a vector or matrix.

cdf D.cdf(x) returns the cumulative distribution function (cdf) values of x which could be a vector or matrix

mgf D.mgf(t) returns the moment-generating function (mgf) values of t which could be a vector or matrix.

quantile D.quantile(p) is the inverse function of D.cdf(x), where p could be a vector or matrix. That is if p = D.cdf(x), then x = D.quantile(p).

sample D.sample(), D.sample(n), and D.sample(m,n) return a random scalar or a vector of size n or a matrix of size m by n.

parameter D.parameter(1) returns λ.

mean D.mean() returns the expected value.

variance D.variance() returns the variance.

Examples

```
> D = StatDist("Poisson",10)
       PoissonDist(10)
> D.sample(1000).mean()
       10.199
> D.sample(1000).variance()
       9.98214
> D.pdf([0,10,100])
                   Col 1        Col 2        Col 3
       Row 1  4.53999e-05     0.125110  4.86465e-63
> D.cdf([0,10,100])
                   Col 1        Col 2        Col 3
       Row 1  4.53999e-05     0.583040      1.00000
> D.quantile(0.5)

 ***ERROR***
 PoissonDist::quantile(): not available
```

12.3.3 Geometric distribution

Definition

The random variable X has a *geometric distribution* if its probability density function (pdf) is defined by

$$f(x) = (1-p)^x p, \quad x = 0, 1, 2, \ldots, \tag{12.16}$$

where p (real) is the parameter with its range $0 \leq p \leq 1$. In short, we say $X \sim g(p)$.

For example, let X be the number of failures before the first success in a sequence of Bernoulli trials with probability of success p, then $X \sim g(p)$.

Properties

1. its moment generation function is

$$M(t) = \frac{p}{1-(1-p)e^t}, \quad t < -\ln(1-p)$$

2. $\mathrm{E}(X) = \frac{1-p}{p}, \mathrm{Var}(X) = \frac{1-p}{p^2}$.

matvec interface

An object of $g(p)$ can be created by

```
D = StatDist("Geometric",p);
```

matvec provided several standard member functions to allow user to access most of properties and functions of $g(p)$:

pdf D.pdf(x) returns the probability density function (pdf) values of x which could be a vector or matrix.

cdf D.cdf(x) returns the cumulative distribution function (cdf) values of x which could be a vector or matrix

mgf D.mgf(t) returns the moment-generating function (mgf) values of t which could be a vector or matrix.

quantile D.quantile(p) is the inverse function of D.cdf(x), where p could be a vector or matrix. That is if p = D.cdf(x), then x = D.quantile(p).

sample D.sample(), D.sample(n), and D.sample(m,n) return a random scalar or a vector of size n or a matrix of size m by n.

parameter D.parameter(1) returns p, the probability of success in a Bernoulli trial.

mean D.mean() returns the expected value.

variance D.variance() returns the variance.

Examples

```
> D = StatDist("Geometric",0.2)
        GeometricDist(0.2)
> D.mean()
        4
> D.sample(1000).mean()
        3.751
> D.pdf([1,2,10])
                     Col 1        Col 2        Col 3
        Row 1     0.160000     0.128000    0.0214748
> D.cdf([1,2,10])
```

```
                    Col 1        Col 2        Col 3
       Row 1     0.360000     0.488000     0.914101
> D.quantile(0.5)

 ***ERROR***
  GeometricDist::quantile(): not available
```

12.3.4 Negative binomial distribution

Definition

The random variable X has a *negative binomial distribution* if its probability density function (pdf) is defined by

$$f(x) = \frac{x+n-1)!}{x!(n-1)!} p^n (1-p)^x, \quad x = 0, 1, 2, \ldots, \tag{12.17}$$

where p (real) and n (integer) are the parameters with their ranges $0 \leq p \leq 1$ and $n \geq 1$.

For example, let X be the number of failures before the n'th success in a sequence of Bernoulli trials with probability of success p, then X is distributed as negative binomial distribution with parameters n and p.

Properties

1. its moment generation function is

$$M(t) = \frac{p^r}{[1-(1-p)e^t]^r}, \quad t < -ln(1-p)$$

2. $\mathrm{E}(X) = \frac{r(1-p)}{p}, \mathrm{Var}(X) = \frac{r(1-p)}{p^2}$.

matvec interface

An object of the negative binomial distribution can be created by

```
D = StatDist("NegativeBinomial",n,p);
```

matvec provided several standard member functions to allow user to access most of properties and functions of the negative binomial distribution:

pdf D.pdf(x) returns the probability density function (pdf) values of x which could be a vector or matrix.

cdf D.cdf(x) returns the cumulative distribution function (cdf) values of x which could be a vector or matrix

mgf D.mgf(t) returns the moment-generating function (mgf) values of t which could be a vector or matrix.

quantile D.quantile(p) is the inverse function of D.cdf(x), where p could be a vector or matrix. That is if p = D.cdf(x), then x = D.quantile(p).

sample D.sample(), D.sample(n), and D.sample(m,n) return a random scalar or a vector of size n or a matrix of size m by n.

parameter D.parameter(1) returns n, number of Bernoulli trials; whereas D.parameter(2) returns p, the probability of success in a Bernoulli trial.

mean D.mean() returns the expected value.

variance D.variance() returns the variance.

Examples

```
> D = StatDist("NegativeBinomial",4,0.1)
        NegativeBinomialDist(4,0.1)
> D.mean()
        36
> D.sample(1000).mean()
        36.256
> D.pdf([0,10,100])
                     Col 1        Col 2        Col 3
        Row 1  0.000100000   0.00997220  0.000469741
> D.cdf([0,10,100])
                     Col 1        Col 2        Col 3
        Row 1  0.000100000    0.0441329     0.994276
> D.quantile(0.5)

 ***ERROR***
  NegativeBinomialDist::quantile(): not available
```

Chapter 13

Exception Handling

Exceptions in matvec

13.1 Error Handling

matvec does not handle errors very gracefully. You will be forced to quit matvec if the number of errors reaches 15 in a session.

After numerous errors, you may find that matvec behaves unexpectedly, then you REALLY have to quit the current session immediately and restart it again.

13.2 Troubleshooting

13.2.1 Lack of memory

Because matvec allocate memory dynamically for each object. If you get a message saying "lack of memory", then you have reached the maximum memory in your computer.

13.2.2 Intermediate files

matvec may automatically swap data from the computer memory to disk. These intermediate files are saved in the $(HOME)/.matvec directory with file names something like `798233981.2` where `798233981` is your computer system time when you launched matvec, and `2` indicates that this is the second intermediate file since you launched matvec. It is clear that you are safe to launch multiple matvec sessions within the same directory as long as those sessions started at different time. All of these intermediate files are automatically deleted whenever matvec is normally ended; otherwise you have to manually delete them.

13.2.3 Bugs

- Whenever you get a message similar to "you have probably found a bug!", report to matvec maintainer with your script and data.
- Whenever you suspect a bug, don't report it until you can duplicate the bug.

13.2.4 EPSILON

- if you get a message similar to

  ```
  ***ERROR***
  SparseMatrix seems not positive (semi)definite: -3284690426.413546

  ***ERROR***
  in sdrvd(), nonzero flag 1
  ```

  Then you have to increase EPSILON value (the default is 1.0e-14) using

  ```
  this.parameter("EPSILON",1.0e-13);
  ```

  or even larger.

13.3 Limitation

There are almost no hard-code in matvec implementation. Thus, there are almost no limitations except the following one.

- One data record must be within one line. Multi-line data record is not allowed. This limitation may be removed later.

Index

Index

*, 21
*=, 21
+, 20, 21
++, 21
+=, 21
-, 20, 21
–, 21
-=, 21
.*, 21
.*=, 21
.+, 21
.+=, 21
.-, 21
.-=, 21
./, 21
./=, 21
.^, 21
/, 21
/=, 21
=, 21
==, 22
^, 21
', 21
<=, 22
<, 22
>=, 22
>, 22

all(), 29
anterior probability, 83
any(), 30
asctime, 55
@, 21

beta distribution, 99
    cdf, 100
    mean, 100
    mgf, 100
    parameter, 100
    pdf, 100
    quantile, 100
    sample, 100
    variance, 100
binomial distribution, 102
    cdf, 103
    mean, 103
    mgf, 103
    parameter, 103
    pdf, 103
    quantile, 103
    sample, 103
    variance, 103
BLUP, 73

Chi square distribution, 90
    cdf, 91
    mean, 91
    mgf, 91
    parameter, 91
    pdf, 91
    quantile, 91
    sample, 91
    variance, 91
chol(), 30
Cholesky factorization, 30
clock, 56
cond(), 30
continuation, 10
continuous distribution, 86
contrast, 79
corrcoef(), 31
cov(), 31
ctime, 56

Data, **18**
date, 55
Date and Time, **54**
  asctime, 55
  clock, 56
  ctime, 56
  date, 55
  difftime, 55
  gmtime, 55
  leapyear, 56
  localtime, 54
  mktime, 55
  time, 54
det(), 31
determinant, 35
DF_REML, 81
diag(), 31
difftime, 55
discrete distribution, 102
distribution
  beta, 99
  binomial, 102
  Chi square, 90
  exponential, 97
  F, 94
  gamma, 96
  geometric, 105
  lognormal, 101
  negative binomial, 107
  non-central beta, 99
  non-central Chi square, 90
  non-central F, 94
  non-central t, 92
  normal, 86
  Poisson, 104
  t, 92
  uniform, 89

eigen value and vector, 32
eigen(), 32
eigenvalues, 32
eigenvectors, 32
else, 47
Elston, 83
EM_REML, 81
EPSILON, 13
escape sequence, 16
estimation, 79
Exception, **109**
!, 21
!=, 22
exponential distribution, 97
  cdf, 98
  mean, 98
  mgf, 98
  parameter, 98
  pdf, 98
  quantile, 98
  sample, 98
  variance, 98

F distribution, 94
  cdf, 95
  mean, 95
  mgf, 95
  parameter, 95
  pdf, 95
  quantile, 95
  sample, 95
  variance, 95
Fernando, 83
fliplr(), 25
flipud(), 25
for, 49

gamma distribution, 96
  cdf, 97
  mean, 97
  mgf, 97
  parameter, 97
  pdf, 97
  quantile, 97
  sample, 97
  variance, 97
Gamma function, 33
gammainc(), 33
gammaln(), 33
genotype probability, 83
geometric distribution, 105
  cdf, 106
  mean, 106
  mgf, 106

  - parameter, 106
  - pdf, 106
  - quantile, 106
  - sample, 106
  - variance, 106
- gmtime, 55

- hadamard, 71
- hankel, 70
- Heuch, 83
- hilb, 70
- hypothesis test, 79

- identity(), 20, 34
- if, 47
- input(), 34
- INPUT_LINE_WIDTH, 13
- int(), 40
- inv(), 34
- inverse, 34
- invhilb, 70
- iterative algorithm, 83

- Janss, 83

- kron(), 34
- Kronecker, 34
- Kronecker product, 34

- leapyear, 56
- least-squares means, 80
- Li, 83
- localtime, 54
- logdet(), 35
- lognormal distribution, 101
  - cdf, 101
  - mean, 102
  - mgf, 101
  - parameter, 101
  - pdf, 101
  - quantile, 101
  - sample, 101
  - variance, 102
- loops, 83
- LU factorization, 35
- lu(), 35

- Matrix, **19**
  - *, 21
  - *=, 21
  - +, 20, 21
  - ++, 21
  - +=, 21
  - -, 20, 21
  - –, 21
  - -=, 21
  - .*, 21
  - .*=, 21
  - .+, 21
  - .+=, 21
  - .-, 21
  - .-=, 21
  - ./, 21
  - ./=, 21
  - .^, 21
  - /, 21
  - /=, 21
  - =, 21
  - ^, 21
  - !, 21
  - ', 21
  - @, 21
  - all(), 29
  - any(), 30
  - chol(), 30
  - cond(), 30
  - corrcoef(), 31
  - cov(), 31
  - det(), 31
  - diag(), 31
  - eigen(), 32
  - fliplr(), 25
  - flipud(), 25
  - gammainc(), 33
  - gammaln(), 33
  - hadamard, 71
  - hankel, 70
  - hilb, 70
  - identity(), 20, 34
  - input(), 34
  - int(), 40
  - inv(), 34
  - invhilb, 70

kron(), 34
logdet(), 35
lu(), 35
Matrix(), 19
max(), 36
mean(), 37
min(), 37
mod(), 37
norm(), 38
ones(), 20, 45
pascal, 71
pinv(), 33
power(), 38
qr(), 35
rand(), 20, 40
rank(), 38
reshape(), 24, 38
resize(), 39
rot90(), 24, 39
round(), 40
sample(), 40
save(), 40
select(), 40
solve(), 41
sort(), 41
std(), 41
sum(), 42
sumsq(), 42
svd(), 42
sweep(), 42
t(), 25, 43
toeplitz, 72
trace(), 43
tril(), 25, 43
triu(), 25, 44
vander, 71
variance(), 45
vec(), 45
vech(), 45
zeros(), 20, 45
max(), 36
MAX_ERRORS, 13
mean(), 37
min(), 37
Mixed Model, **73**
BLUP, 73
contrast, 79
estimation, 79
hypothesis test, 79
least-squares means, 80
multi-trait, 78
variance components estimation, 81
mktime, 55
mod(), 37
modulo operation, 37
Moore-Penrose, 33
multi-trait, 78
Murphy, 83
Mutalik, 83

negative binomial distribution, 107
cdf, 107
mean, 108
mgf, 107
parameter, 108
pdf, 107
quantile, 108
sample, 108
variance, 108
non-central beta distribution, 99
nonct, 100
non-central Chi square distribution, 90
nonct, 91
non-central F distribution, 94
nonct, 95
non-central t distribution, 92
nonct, 94
norm(), 38
normal distribution, 86
cdf, 88
mean, 88
mgf, 88
parameter, 88
pdf, 88
quantile, 88
sample, 88
variance, 88

Objects, 12
ones(), 20, 45
OUTPUT_FLAG, 13
OUTPUT_LINE_WIDTH, 13

OUTPUT_PRECISION, 12

pascal, 71
Pedigree, **16**
pinv(), 33
Poisson distribution, 104
- cdf, 105
- mean, 105
- mgf, 105
- parameter, 105
- pdf, 104
- quantile, 105
- sample, 105
- variance, 105

posterior probability, 83
power(), 38
Program Flow Control, **47**
- else, 47
- for, 49
- if, 47
- repeat, 50
- until, 50
- while, 50

QR factorization, 35
qr(), 35

rand(), 20, 40
rank(), 38
REML, 81
repeat, 50
reshape(), 24, 38
resize(), 39
rot90(), 24, 39
rotation, 39
round(), 40

save elements to a file, 40
save(), 40
select(), 40
singular value decomposition, 42
solve(), 41
sort(), 41
special matrix, 46
standard deviation, 41
Statistical Distributions, **85**
- cdf, 85
- mean, 86
- mgf, 85
- nonct, 86
- parameter, 86
- pdf, 85
- quantile, 85
- random number, 86
- sample, 86
- StatDist, 85
- variance, 86

std(), 41
Stewart, 83
String, **15**
- escape sequence, 16

sum(), 42
summation, 42
sumsq(), 42
svd(), 42
sweep operation, 42
sweep(), 42

t distribution, 92
- cdf, 93
- mean, 94
- mgf, 93
- parameter, 94
- pdf, 93
- quantile, 93
- sample, 94
- variance, 94

t(), 25, 43
time, 54
toeplitz, 72
trace(), 43
transpose, 21, 43
tril(), 25, 43
triu(), 25, 44

uniform distribution, 89
- cdf, 89
- mean, 89
- mgf, 89
- parameter, 89
- pdf, 89
- quantile, 89

sample, 89
variance, 89
until, 50

Van Arendonk, 83
vander, 71
variance components estimation, 81
variance(), 45
vec(), 45
vech(), 45
Vector, **14**
==, 22
!=, 22
<=, 22
<, 22
>=, 22
>, 22

Wang, 83
WARNING, 13
while, 50

zeros(), 20, 45