VIRTUAL COMPUTING

Hyper

An Object Oriented Scripting Language

Reference Manual

Shaheen Hoque
8/6/2017

Developed by Shaheen Hoque, Hyper is a general purpose object oriented interpreted scripting language with an emphasis on technical computation. Hyper combines powerful object oriented programming (OOP) with an easy to use interpretive environment.

Contents

Table of Contents

1 Introduction

Hyper is a general-purpose object oriented interpreted scripting language with an emphasis on technical computation. Hyper combines powerful object-oriented programming (OOP) with an easy to use interpretive environment. Hyper can be used in two different modes: interactive and batch script. In the interactive mode, the user can type a command or a statement in a console and get the output immediately. In the batch script mode, the user can write a script and run the script by typing the name of the script file in the consol. The interactive mode has some shell-like commands (similar to Linux/Unix/DOS) which provide utility functions. The syntax of Hyper is similar to the syntax of Java, with some exceptions. In addition to the scripting, programs can also be written in Java, compiled and imported into the Hyper's interpretive environment. The imported classes in the compiled java binary can be accessed as easily as accessing the classes written in Hyper scripting language. In fact, any pre-compiled java classes can be accessed this way, including all the public classes in the Java API. Hyper is fully integrated with the java APIs.

Hyper is a full fledge object oriented language. The object-oriented features such as **encapsulation**, **inheritance**, and **polymorphism** are implemented using classes and objects. It features **try catch** and **finally**. It also features **strong typing** with default parameter values and multiple parameters return from a function call. Hyper supports a rich set of data types related to mathematics and other technical computation, such as **Complex**, **Matrix**, **Polynomial** and several others, in addition to the data types found in a typical programming language. Hyper features a large library of functions, such as **Linear Algebra**, **Probability** and **Statistics**, in addition to the basic math functions.

This document is a reference manual for the Hyper language. It's not intended to be a tutorial. A separate tutorial will be produced at a later time. Section 2 and 3 describes shell commands and utility functions. Section 4 describes the syntax of Hyper language. Section 5 describes the class libraries. Appendices A-G provides detail documentations of the function call signatures of the class libraries.

2 Interactive Commands

The following commands can be used in the interactive mode:

Command	Argument	Description
`cd`	**`<path>`**	Changes default directory Path can be specified in the similar fashion as in DOS or Unix. ".." implies path for the directory above the current directory.

clear	**<filter>**	Clears variable(s) from the workspace Filter can be keywords, "all", "var", "fcn", "lib", and "screen" or partial variable name with single (?) or multiple (*) wild cards. "all" implies everything in the workspace. "var" implies only variables. "fcn" implies functions. "lib" implies imported Java library. "screen" implies the console output screen.
ls	**<filter>**	Prints in a table format the list of the files in the current working directory. Filter can be partial file name with single (?) or multiple (*) wild cards used in a similar fashion to DOS or Unix. If no filter is used, the entire list will be printed.
lsd	**<filter>** **<sort key>**	Prints in a detailed format the list of the files in the current working directory. Detailed format consists of 4 columns: Name, Type, Size, and Date Modified. Filter usage is the same as in "ls" command. An optional second argument can be used as a sort key. An optional Plus("+") or minus ("-") character can be used before the sort key to sort in ascending or descending order. The default sort order is ascending. The following sort keys can be used: "name", "type", "size", and "date". The default sort key is "name".
pwd	**none**	Print the working directory
ws	**<filter>**	Prints in a table format the list of variables in the current workspace. Filter can be keywords, "all", "var", "fcn" and "lib" or partial variable name with single (?) or multiple (*) wild cards. "all" implies everything in the workspace. "var" implies only variables. "fcn" implies only functions. "lib" implies only imported Java library. If no filter is used, only the variables will be printed. (Same as using the "var" filter)
wsd	**<filter>**	Prints in a detailed format the list of variables in the current workspace.

		Detailed format consists of 2 columns: Name and Type. Filter usage is the same as in "ws" command.
wsv	**<filter>**	Prints in a detailed format the list of variables in the current workspace. Detailed format consists of 3 columns: Name and Type, and Value. Filter usage is the same as in "ws" command.

Text between the angle brackets (**<>**) are provided by the user. Although these commands cannot be accessed from the script, but there are equivalent functions that can be called from the script to obtain similar results. These functions are called Utility functions. For example, to change directory in the interactive mode, the user can type the following command:

```
cd ..\abc
```

But when writing a script, the user needs to call the following function:

```
changedir("..\abc");
```

More details of the Utility functions are described in the next section.

3 Utility Functions

Utility functions for Hyper are listed alphabetically in the table below:

Call Signature	Description	Return Type
changedir(<path>)	Changes the working directory according to **<path>**	**void**
clearws(<var list>)	Clears the variables in the **<var list>** from the work space.	**void**
exec(<Script name>)	Executes the named script.	**void**
getworkingdir()	Returns the path of the current working directory.	**string**
listdir(<filter>)	Returns a filtered list of the files in the current working directory.	**list**
maxdigits(<arg>)	Sets the maximum number of digits after the decimal point corresponding to the absolute value of the parameter <arg>. <arg> can be a long value, or a double value. If double value used, the fractional part is ignored. If no argument or more than 1 argument used, the current maximum number of digits is displayed.	**integer**
notation(<arg>)	Sets the output notation in one of the following formats: REGULAR, SCIENTIFIC, or ENGINEERING. The output format can be specified by using the parameter **<arg>**. **<arg>** can be a string, an integer value, or a real value. "regular", reg", 1, 1.0 corresponds to the notation REGULAR. "scientific", "sci", 2, 2.0 corresponds to the notation SCIENTIFIC. "regular",	**string**

	reg”, 1, 1.0 corresponds to the notation REGULAR. “engineering”, “eng”, 3, 3.0 corresponds to the notation ENGINEERING. String arguments are case insensitive. If no argument or more than 1 argument used, the current notation is displayed.	
`print(<arg>)`	Prints the argument. The argument can be a string or a variable.	**`void`**

4 Syntax

4.1 Lexical Conventions

4.1.1 Comments:

Comments : "//"

Two forward slashes used to comment out the texts after the double slashes in a given line. This is similar to the comments used in Java and C++ languages.

Example:

```
// This is a comment.
```

4.1.2 Reserved Words:

RESERVED WORDS:

“break”
"catch"
"class"
“default”
"else"
"finally"
"for"
"function"
"if"
"import"
"in"
“loop”
“switch”
"return"

"throw"
"try"
"while"

4.1.3 Literals:

4.1.3.1 Decimal Literal:

DECIMAL_LITERAL: ["1"-"9"] (["0"-"9"])*

HEX_LITERAL: "0" ["x","X"] (["0"-"9","a"-"f","A"-"F"])+

OCTAL_LITERAL: "0" (["0"-"7"])*

FLOATING_POINT_LITERAL:

(["0"-"9"])+ "." (["0"-"9"])* (<EXPONENT>)?

| "." (["0"-"9"])+ (<EXPONENT>)?

| (["0"-"9"])+ <EXPONENT>

EXPONENT: ["e","E"] (["+","-"])? (["0"-"9"])+

Example:

```
-1.602E-19
```

4.1.3.2 Character Literal:

CHARACTER_LITERAL:

```
""""
(   (~[""","\\","\n","\r"])
  | ("\\"
      ( ["n","t","b","r","f","v","\\",""","\""]
      | ["0"-"7"] ( ["0"-"7"] )?
      | ["x","X"] ["0"-"9","a"-"f","A"-"F"] (["0"-"9","a"-"f","A"-"F"])?
      )
    )
)
""""
```

Examples:

```
'c'
'5'
```

4.1.3.3 String Literal:

STRING_LITERAL:

```
"\""
(   (~["\"","\\","\n","\r"])
  | ("\\"
      ( ["n","t","b","r","f","v","\\","'","\""]
      | ["0"-"7"] ( ["0"-"7"] )?
      | ["x","X"] ["0"-"9","a"-"f","A"-"F"] (["0"-"9","a"-"f","A"-"F"])?
      )
    )
)*
"\""
```

Example:

```
"This is a String"
"This is a String with a carriage return \n"
"This is a String with a back slash \\"
```

4.1.4 Identifier:

IDENTIFIER: <LETTER> (<LETTER>|["0"-"9"])*

Example:

```
Abc_123
```

LETTER: ["_","a"-"z","A"-"Z"]

DIGIT: ["0"-"9"] >

Separators: [",", ";", "(" , ")", "{", "}", "[", "]", ".", "::",]

4.1.5 Operators:

Operators:

Symbol	Operation
"="	Assignment
">"	Greater than
"<"	Less than
"!"	Not
"=="	Equal
"<="	Less than or equal
">="	Greater than or equal
"!="	Not equal
"||"	Or
"&&"	And
"+"	Plus
"-"	Minus
"*"	Multiply
"/"	Divide
"^"	Exponent
"%"	Remainder
":"	Range

4.2 Data Types

Hyper supports the following data types:

- Boolean
- Character
- String
- Numeric
  - Integer
  - Real
  - Imaginary
  - Complex
  - Quaternion
  - Polynomial
  - Rational
  - Range
  - Vector
    - Real Vector
    - Complex Vector
    - Quaternion Vector
    - Polynomial Vector
    - Rational Vector

- Matrix
  - Real Matrix
  - Complex Matrix
  - Quaternion Matrix
  - Polynomial Matrix
  - Rational Matrix
- Array
- Table
- Class

Variables representing the data types have attributes that can be used to obtain various properties of a variable. All data types share some common attributes, and some data types have special attributes. These attributes can be accessed using dot notation as below:

```
<another var> = <var>.<attribute>;
```

or

```
<another var> = <var>.<attribute>(<param_1>,  …, <param_n”);
```

The common attributes are given in the table below:

Call Signature	Description	Return Type
`type`	Returns the name of the type of the variable.	`string`
`isA(type)`	Returns TRUE if the parameter **`type`** is the type of the variable, and returns FALSE otherwise.	`boolean`

4.2.1 Boolean

boolean: “True” | “False”

A variable, **`<var>`**, can be defined as **boolean** by following ways:

```
<var> = boolean();
```

or

```
<var> = <boolean value>;
```

Examples:

```
b = boolean(1); -> b = true.
```

```
b = boolean(0); -> b = false.
b = True;
b = false;
```

4.2.2 Character

Character: CHARACTER_LITERAL

A variable, **`<var>`**, can be defined as **character** by following ways:

```
<var> = char("<value");
```

or

```
<var> = '<value>';
```

Examples:

```
c = char("a");
c = 'a';
c = '5';
```

4.2.3 String

String: STRING_LITERAL

A variable, **`<var>`**, can be defined as **String** by following ways:

```
<var>= String(<value>);
```

or

```
<var>= "<Value>";
```

Examples:

```
str = String("This is a String");
str = "This is a String";
```

Attributes for String type are given in the table below:

Call Signature	Description	Return Type
`length`	Returns the **length**.	`integer`
`index`	Returns the **index** of a character.	`integer`

`rindex`	Returns the **right index** of a character.	`integer`
`substring`	Returns a **sub string**.	`string`
`toLowerCase`	Returns the same string with all **lower-case** characters.	`string`
`toUpperCase`	Returns the same string with all **upper-case** characters.	`string`
`startsWith`	Tests if this string starts with the specified prefix.	`boolean`
`endsWith`	Tests if this string ends with the specified suffix.	`boolean`
`split`	Returns a list with elements obtained from splitting the string.	`arrayList`

4.2.4 Integer

The **integer** data type is a 32-bit signed two's complement integer, which has a minimum value of -2^{31} and a maximum value of $2^{31}-1$.

integer : DECIMAL_LITERAL

A variable, **`<var>`**, can be defined as long by following ways:

```
<var> = integer(<value>);
```

or

```
<var> = <integer value>;
```

Examples:

```
i = integer(3.2); -> i = 3.
i = 10;
```

4.2.5 Real

The **real** data type is a double-precision 64-bit IEEE 754 floating point.

real : FLOATING_POINT_LITERAL

A variable, **`<var>`**, can be defined as double by following ways:

```
<var> = real(<value>);
```

or

```
<var> = <real value>;
```

Examples:

```
r = real(2); -> r = 2.0.
r = 3.14;
r = 9.11E-31;
```

4.2.6 Imaginary

A variable, `<var>`, can be defined as imaginary by following ways:

```
<var>= imaginary(<value>);
```

or

```
<var>= <value>i;
```

`<value>` can be either integer or double;

Examples:

```
imag = imaginary(2.5); -> imag = 2.5i
```

or

```
imag = imaginary(2); -> imag = 2.0i
```

or

```
imag = 2.5i;
```

or

```
imag = 2i; -> imag = 2.0i
```

4.2.7 Complex

A variable, `<var>`, can be defined as complex by following ways:

```
<var>= complex(<value>,<value>)
```

or

```
<var> = <value> ± <value>i;
```

<value> can be either integer or double;

Examples:

```
com = complex(2.5, -3); -> com = 2.5 - 3.0i
```

or

```
com = 2.5 + 3i; -> com = 2.5 + 3.0i
```

Attributes for Complex type are given in the table below:

Call Signature	Description	Return Type
`real`	Returns the **real** part.	`real`
`imaginary`	Returns the **imaginary** part.	`real`
`magnitude`	Returns the **magnitude**.	`real`
`angle`	Returns the **phase angle**.	`real`
`conjugate`	Returns the **complex conjugate**.	`complex`

4.2.8 Quaternion

A variable, **<var>**, can be defined as quaternion by following ways:

```
<var> = quaternion(<value>,< value >,< value >,< value >)
```

or

```
<var> = <value> ± <value>i ± <value>j ± <value>k;
```

or

```
<var> = <value>i ± <value>j ± <value>k;
```

or

```
<var> = <value> ± <value>i ± <value>j;
```

or

```
<var> = <value> ± <value>i ± <value>k;
```

or

```
<var> = <value> ± <value>j ± <value>k;
```

or

```
<var> = <value> ± <value>j;
```

or

```
<var> = <value> ± <value>k;
```

or

```
<var> = <value>i ± <value>j;
```

or

```
<var> = <value>i ± <value>k;
```

or

```
<var> = <value>j ± <value>k;
```

<value> can be either integer or double;

Examples:

```
quat = quaternion (2.5, -3, 4, -1.2); -> quat = 2.5 - 3.0i + 4j -1.2k.
```

or

```
quat = 2.5 - 3.0i + 4j -1.2k;
```

Some components may be skipped as shown below:

```
quat = -3.0i + 4j -1.2k; -> quat = 0.0 - 3.0i + 4.0j -1.2k.

quat = 2.5 - 3.0i + 4j; -> quat = 2.5 - 3.0i + 4.0j + 0.0k.

quat = 2.5 - 3.0i + -1.2k; -> quat = 2.5 - 3.0i + 0.0j -1.2k.

quat = 2.5 + 4j -1.2k; -> quat = 2.5 + 0.0i + 4.0j -1.2k.

quat = 2.5 + 4j; -> quat = 2.5 + 0.0i + 4.0j + 0.0k.

quat = 2.5 -1.2k; -> quat = 2.5 + 0.0i + 0.0j -1.2k.

quat = -3.0i + 4j; -> quat = 0.0 - 3.0i + 4.0j + 0.0k.

quat = -3.0i -1.2k; -> quat = 0.0 - 3.0i + 0.0j -1.2k.

quat = 4j -1.2k; -> quat = 0.0 + 0.0i + 4.0j -1.2k.
```

Attributes for Quaternion type are given in the table below:

Call Signature	Description	Return Type
`scalar`	Returns the **scalar** part.	`real`
`vector`	Returns the **vector** part.	`realVector`
`i`	Returns the **i component of the vector** part.	`real`
`j`	Returns the **j component of the vector** part.	`real`
`k`	Returns the **k component of the vector** part.	`real`
`norm`	Returns the **norm**.	`real`
`unit`	Returns the **unit quaternion**.	`quaternion`
`conjugate`	Returns the **complex conjugate**.	`quaternion`
`reciprocal`	Returns the **reciprocal quaternion**.	`quaternion`

4.2.9 Polynomial

The type Polynomial represents a mathematical polynomial. A variable, **`<var>`**, can be defined as polynomial by following ways:

```
<var>= polynomial(<coefficient>,<coefficient>, … ,<coefficient>)
```

or

```
<var>= #<coefficient>, <coefficient>, … , <coefficient>#;
```

<**`coefficient`**> can be either long or double, but the coefficients of the polynomial are always double;

Examples:

A polynomial can be created using a constructor:

```
poly = polynomial(1.5, 2,1e-3)
```

will result in: $poly = 1.5x^2 + 2x + 0.003$

or using the polynomial operator, "#".

```
poly = #1.5, 2, 1e-3#;
```

will result in: $poly = 1.5x^2 + 2x + 0.003$

or

```
poly = #1.5, 2, 1e-3, 4.5#;
```

will result in: $poly = 1.5x^3 + 2x^2 + 0.003x + 4.5$

Attributes for Polynomial type are given in the table below:

Call Signature	Description	Return Type
`degree`	Returns the **degree** of the polynomial.	`integer`
`eval(integer x)`	Returns the **value** of the polynomial for the parameter **`x`**.	`real`
`eval(real x)`	Returns the **value** of the polynomial for the parameter **`x`**.	`real`
`eval(imaginary x)`	Returns the **value** of the polynomial for the parameter **`x`**.	`complex`
`coefficients`	Returns all the coefficients.	`realVector`
`coefficient(integer n)`	Returns coefficient corresponding to the power **`n`**.	`real`
`coefficient(real n)`	Returns coefficient corresponding to the power **`n`**.	`real`

4.2.10 Rational

The type Rational represents a ratio of two polynomials. A variable, **`<var>,`** can be defined as rational using a constructor:

```
<var>= rational(<polynomial_value>, <polynomial_value>)
```

or using division operator, "/".

```
<var>= <polynomial_value>/<polynomial_value>;
```

Examples:

```
poly1 = #1, 2, 3#;
poly2 = #4, 5, 6, 7#;
rat = polynomial(poly1,poly2);
```

will result in: $rat = \frac{x^2+2x+3}{4x^3+5x^2+6x+7}$

The same result can be achieved by

```
rat = poly1/poly2;
```

Attributes for Rational type are given in the table below:

Call Signature	Description	Return Type
`num`	Returns the **numerator**.	`polynomial`
`den`	Returns the **denominator**.	`polynomial`
`eval(integer x)`	Returns the **value** of the polynomial for the parameter **`x`**.	`real`
`eval(real x)`	Returns the **value** of the polynomial for the parameter **`x`**.	`real`
`eval(imaginary x)`	Returns the **value** of the polynomial for the parameter **`x`**.	`complex`

4.2.11 Range

The type range represents a sequence of equally spaced integers. A variable, **`<var>`**, can be defined as a range by following ways:

```
<var>= range(<first>, <last>, <increment>);
```

or

```
<var>= range(<first>, <last>);
```

In this method, the increment is assumed to be 1.

or

```
<var> = range(<last>)
```

In this method, the first value of the sequence is assumed to be 0, and the increment is assumed to be 1.

Example 1:

```
rng = range(10, 30, 5);
```

The statement above will produce the following sequence:

10, 15, 20, 25, 30

Example 2:

```
rng = range(10, 15);
```

The statement above will produce the following sequence:

10, 11, 12, 13, 14, 15

Example 3:

```
rng = range(5);
```

The statement above will produce the following sequence:

0, 1, 2, 3, 4, 5

Attributes for Quaternion type are given in the table below:

Call Signature	Description	Return Type
`length`	Returns the **length** of the range.	`integer`
`first`	Returns the **first element** of the range.	`integer`
`last`	Returns the **last element** of the range.	`integer`
`incr`	Returns the **increment** value.	`integer`

4.2.12 Vector

The type Vector represents a mathematical vector, as defined in Linear Algebra. There are five subtypes of Vector: Real Vector, Complex Vector, Quaternion Vector, Polynomial Vector, and Rational Vector.

Common attributes for all Vector type are given in the table below:

Call Signature	Description	Return Type
`length`	Returns the **length** of the vector.	`integer`
`transpose`	Returns the **transpose** of the vector.	`matrix`

4.2.12.1 Real Vector

The elements of a Real Vector are of type double.

A variable, `<var>`, can be defined as a vector by two different methods:

First method:

```
<var> = [<element>, <element>, … , <element>];
```

Example:

```
vec = [1.5, 2, 1e-3];
```

`<element>` can be either integer or real, but the vector elements are always real.

Second Method:

```
<var> = realVector(<element>,<element>, … ,<element>)
```

Example:

```
vec = realVector(1.5, 2, … ,1e-3)
```

Both methods will produce the following vector:

$$vec = \begin{bmatrix} 1.5 \\ 2.0 \\ 0.003 \end{bmatrix}$$

Attributes for Vector type are given in the table below:

Call Signature	Description	Return Type
`norm`	Returns the **norm** of the vector.	`real`

4.2.12.2 Complex Vector

The elements of a Complex Vector are of type complex. A variable, **`<var>`**, can be defined as a complex vector by three different methods:

First Method:

```
<var> = [<element>, <element>, … , <element>];
```

If at least one **`<element>`** is of type Complex or Imaginary and all other **`<element>`**s are of type Integer or Real, a Complex Vector will be produced using the parameters. Elements of type Real are converted to complex numbers with zero imaginary parts, and elements of type Imaginary are converted to complex numbers with zero real parts.

Example:

```
vec = [1.5 + 3i, 2 + 4.5i, 1e-3 + 1e-2i];
```

Second method:

```
<var> = complexVector(<element>, <element>, … , <element>)
```

If at least one parameter is of type Complex or Imaginary and all other parameters are of type Integer or Real, a Complex Vector will be produced using the parameters. The **`<element>`**s are of type complex. Elements of type Real are converted to complex numbers with zero imaginary parts and elements of type Imaginary are converted to complex numbers with zero real parts.

Third method:

```
<var>= complexVector(<element1>,<element2>)
```

If `<element1>,<element2>` are vector of real values and have equal lengths, a Complex Vector will be produced whose from the `<element1>` and `<element2>`. `<element1>` will provide the real components and the `<element2>` will provide the imaginary components.

Attributes for Complex Vector type are given in the table below:

Call Signature	Description	Return Type
`real`	Returns the **real** part.	`realVector`
`imaginary`	Returns the **imaginary** part.	`realVector`
`magnitude`	Returns the **magnitude**.	`realVector`
`angle`	Returns the **phase angle**.	`realVector`
`conjugate`	Returns the **complex conjugate**.	`complexVector`

4.2.12.3 Quaternion Vector

The elements of a Quaternion Vector are of type quaternion. A variable, **`<var>`**, can be defined as a quaternion vector by three different methods:

First Method:

```
<var>= [<element>, <element>, … , <element>];
```

If at least one `<element>` is of type Quaternion and rests of the `<element>`s are of type Long, Double, Imaginary, or Complex, a Quaternion Vector will be produced using the parameters. Elements of type other than Quaternion are converted to Quaternion.

Example:

```
vec = [1.5 + 3i + 2j + 1k, 2 + 4.5i + 1e-3j + 1e-2k];
```

Second method:

```
<var>= quaternion_realVector(<element>, <element>, … , <element>)
```

If at least one parameter is of type Quaternion and rests of the parameters are of type Long, Double, Imaginary, or Complex, a Quaternion Vector will be produced using the parameters. Elements of type other than Quaternion are converted to Quaternion.

Third method:

```
<var> = quaternionVector(<element1>,<element2>,<element3>,<element4>)
```

If **<element1>,<element2>,<element3>,<element4>** are vectors of real values and equal lengths, a Quaternion Vector will be produced from the **<element1>, <element2>, <element3>, and <element4>. <element1>** will provide the scalar components and the **<element2>, <element3>, and <element4>** will provide the vector components.

Attributes for Quaternion Vector type are given in the table below:

Call Signature	Description	Return Type
`scalar`	Returns the **scalar** part.	`realVector`
`i`	Returns the **i component of the vector** part.	`realVector`
`j`	Returns the **j component of the vector** part.	`realVector`
`k`	Returns the **k component of the vector** part.	`realVector`
`norm`	Returns the **norm**.	`realVector`
`unit`	Returns the **unit quaternion**.	`quaternionVector`
`conjugate`	Returns the **complex conjugate**.	`quaternionMatrix`
`reciprocal`	Returns the **reciprocal quaternion**.	`quaternionVector`

4.2.12.4 Polynomial Vector

The elements of a Polynomial Vector are of type polynomial. A variable, **<var>**, can be defined as a polynomial vector by two different methods:

First Method:

```
<var> = [<element>, <element>, … , <element>];
```

If all the **<element>**s are of type polynomial, a polynomial vector will be created.

Example:

```
vec = [#1,2,3# , #4,5,6#];
```

Second method:

```
<var> = polynomialVector(<element>, <element>, … , <element>)
```

If all the parameters are of type Polynomial, a Polynomial Vector will be produced using the parameters.

Attributes for Polynomial Vector type are given in the table below:

Call Signature	Description	Return Type
`degree`	Returns the **degree** of the polynomial.	`realVector`
`eval(integer x)`	Returns the **value** of the polynomial for the parameter **x**.	`realVector`
`eval(real x)`	Returns the **value** of the polynomial for the parameter **x**.	`realVector`
`eval(imaginary x)`	Returns the **value** of the polynomial for the parameter **x**.	`complexVector`

4.2.12.5 Rational Vector

The elements of a Rational Vector are of type rational. A variable, **`var`**, can be defined as a polynomial vector by two different methods:

First Method:

```
<var> = [<element>, <element>, … , <element>];
```

The **`<element>`**s are of type rational.

Example:

```
vec = [#1,2,3# / #4,5,6#, #11,12,13# / #14,15,16#];
```

Second method:

```
<var> = rationalVector(<element>, <element>, … , <element>)
```

If all the parameters are of type Rational, a Rational Vector will be produced using the parameters.

Attributes for Rational Vector type are given in the table below:

Call Signature	Description	Return Type
`num`	Returns the **numerator**.	`polynomialVector`

`den`	Returns the **denominator**.	`polynomialVector`
`eval(integer x)`	Returns the **value** of the polynomial for the parameter **`x`**.	`realVector`
`eval(real x)`	Returns the **value** of the polynomial for the parameter **`x`**.	`realVector`
`eval(imaginary x)`	Returns the **value** of the polynomial for the parameter **`x`**.	`complexVector`

4.2.13 Matrix

The type Matrix represents a mathematical matrix, as defined in Linear Algebra. There are five subtypes of Matrix: Real Matrix, Complex Matrix, Quaternion Matrix, Polynomial Matrix, and Rational Matrix.

Accessing matrix elements:

The element at row *i* and column *j* of the matrix can be accessed by following way:

```
mat[i,j];
```

Contiguous elements from row r_1 to r_2 and from column c_1 to c_2 can be accessed by following way:

```
mat[r1:r2, c1:c2]
```

Common attributes for all Matrix type are given in the table below:

Call Signature	Description	Return Type
`rows`	Returns the **number of rows** of the matrix.	`integer`
`columns`	Returns the **number of columns** of the matrix.	`integer`
`transpose`	Returns the **transpose** of the vector.	`Matrix`
`isSquare`	Returns TRUE if the matrix is square, and returns FALSE otherwise.	`boolean`
`isSymmetric`	Returns TRUE if the matrix is symmetric, and returns FALSE otherwise.	`boolean`

4.2.13.1 Real Matrix

The elements of a Real Matrix are of type Real. A variable, **`<var>`**, can be defined as Real Matrix by three different methods.

First method:

```
<var> =      | <element>, <element>, … , <element> |,
             | <element>, <element>, … , <element> |,
                              ⋮
             | <element>, <element>, … , <element> |, (3)
```

<element>s can be of type either Integer or Real, but the matrix elements types are always Real.

Example:

The matrix from the second method can be created by the following way:

```
mat = |1.5,  4.7, 7.3 |,
      |2.0,  5.0, 8.22|,
      |1e-3, 6.0, 9.0 |;
```

Will result in the following the matrix:

$$mat = \begin{bmatrix} 1.5 & 4.7 & 7.3 \\ 2.0 & 5.0 & 8.22 \\ 0.003 & 6.0 & 9.0 \end{bmatrix}$$

Second Method:

```
<var> = realMatrix(<numRow>,<numColumn>
<element>,<element>, … ,<element>) (1)
```

<numRow> and **<numColumn>** are number or rows and the number columns in the matrix. The total number of elements (TNOE) equals to **<numRow>** times **< numColumn>**.

If there are no other parameters, all the elements in the matrix will be initialized to zeros. If a third parameter, **<element>**, of type Integer or Real is provided, all the elements in the matrix will be initialized to the value of the third parameter. If more than one **<element>** are provided, these **<element>**s will be used in an attempt to fill the matrix. If the number of **<element>**s (NOE) is equal to TNOE, the matrix will be filled exactly. If NOE is less than TNOE, the rest of the elements will be set to zero. If the NOE is greater than TNOE, the excess **<element>**s will be ignored.

Example:

```
mat = realMatrix(2, 3, 1, 2.0, 3.5, 4, 5, 6)
```

Will result in the following matrix:

$$mat = \begin{bmatrix} 1.0 & 2.0 & 3.5 \\ 4.0 & 5.0 & 6.0 \end{bmatrix}$$

Third Method:

```
<var> = realMatrix(<realVector_value>,< realVector _value>
, … ,< realVector _value>) (2)
```

<realVector_value>s are of type Vector. All **<realVector_value>**s must have the same length. Vectors are represented as columns of the matrix. Any number of **<realVector_value>**s can be provided.

Example:

```
Vec1 = realVector(1.5, 2, … ,1e-3)
Vec2 = [4.7, 5, 6];
Vec3 = [7.3, 8.22, 9.0];
mat = realMatrix(vec1, vec2, vec3)
```

Will result in the following the matrix:

$$mat = \begin{bmatrix} 1.5 & 4.7 & 7.3 \\ 2.0 & 5.0 & 8.22 \\ 0.003 & 6.0 & 9.0 \end{bmatrix}$$

Accessing matrix elements:

Examples:

```
mat[2,3] -> 8.22
```

`mat[2:3, 2:3] ->` $\begin{bmatrix} 5.0 & 8.22 \\ 6.0 & 9.22 \end{bmatrix}$

The number of rows of a matrix can be obtained as follows:

```
mat.numrows -> 3
```

Similarly, the number of columns of a matrix can be obtained as follows:

```
mat.numcolumns -> 3
```

For the matrix form the examples of methods 2 and 3, the values would be 3 and 3.

4.2.13.2 Complex Matrix

The elements of a Complex Matrix are of type complex. A variable, **<var>**, can be defined as Complex Matrix by four different methods.

First method:

```
<var> =     | <element>, <element>, … , <element> |,
            | <element>, <element>, … , <element> |,
                              :
            | <element>, <element>, … , <element> |
```

<element>s can be of type Integer, Real, Imaginary or Complex, but if at least one of the **<element>**s is of type Complex, all the matrix elements will be of type Complex.

Example:

The matrix from the second method can be created by the following way:

```
mat = |1.5+i,  4.7+1.5i,      7.3 |,
      |    2,        5i, 8.22+0.3i|,
      |1e-3i,      6+2i,        9 |;
```

Will result in the following the matrix:

$$mat = \begin{bmatrix} 1.5+i & 4.7+1.5i & 7.3 \\ 2.0 & 5.0i & 8.22+0.3i \\ 0.003i & 6.0+2i & 9.0 \end{bmatrix}$$

Second method:

```
<var> = complexMatrix (<numRow>,<numColumn>
<element>,<element>, … ,<element>) (1)
```

<numRow> and **<numColumn>** are number or rows and the number columns in the matrix. The total number of elements (TNOE) equals to **<numRow>** times **< numColumn>**.

If there are no other parameters, all the elements in the matrix will be initialized to complex numbers, whose real and imaginary parts are zeros. If a third parameter, **<element>**, of type

Integer, Real, Imaginary or Complex is provided; all the elements in the matrix will be initialized to the value of the third parameter. If more than one **<element>** are provided, these **<element>**s will be used in an attempt to fill the matrix. If the number of **<element>**s (NOE) is equal to TNOE, the matrix will be filled exactly. If NOE is less than TNOE, the rest of the elements will be set to complex numbers, whose real and imaginary parts are zeros. If the NOE is greater than TNOE, the excess **<element>**s will be ignored.

Example:

```
mat = complexMatrix (2, 3, 1+2i, 2.0+4i, 0.5i, 4+i, 5+2.5i, 6)
```

Will result in the following matrix:

$$mat = \begin{bmatrix} 1.0+2i & 2.0+4i & 0.5i \\ 4.0+i & 5.0+2.5i & 6.0 \end{bmatrix}$$

Third Method:

```
<var> = complexMatrix (<complexVector>,<complexVector>
, … ,<complexVector>) (2)
```

<complexVector>s are of type Complex Vector. All **<complexVector>**s must have the same length. Vectors are represented as columns of the matrix. Any number of **<complexVector>**s can be provided.

Example:

```
Vec1 = complexVector(1.5+i, 2, … ,1e-3+0.1i)
Vec2 = [4.7+1.5i, 5i, 6+2i];
Vec3 = [7.3, 8.22+0.3i, 9i];
mat = complexMatrix (vec1, vec2, vec3)
```

Will result in the following the matrix:

$$mat = \begin{bmatrix} 1.5+i & 4.7+1.5i & 7.3 \\ 2.0 & 5.0i & 8.22+0.3i \\ 0.003i & 6.0+2i & 9.0i \end{bmatrix}$$

Fourth Method:

```
<varR> =     | <real_element>, < real_element>, … , < real_element>  |,
             | < real_element>, < real_element>, … , < real_element> |,

                                 ⋮
```

```
                | < real_element>, < real_element>, … , < real_element> |


<varI> =        | <real_element>, < real_element>, … , < real_element>  |,
                | < real_element>, < real_element>, … , < real_element> |,
                                          ⋮
                | < real_element>, < real_element>, … , < real_element> |


<var> = complexMatrix (<varR>, <varI>);
```

Example:

```
matR = |1.5, 4.7, 7.3 |,
       |  2,    0, 8.22|,
       |  0,    6,    9 |;


matI = |   1, 1.5,    0|,
       |   0,    5, 0.3|,
       |1e-3,    2,    0|;


mat = complexMatrix (matR, matI);
```

Will result in the following the matrix:

$$mat = \begin{bmatrix} 1.5 + i & 4.7 + 1.5i & 7.3 \\ 2.0 & 5.0i & 8.22 + 0.3i \\ 0.003i & 6.0 + 2i & 9.0i \end{bmatrix}$$

Attributes for Complex type are given in the table below:

Call Signature	Description	Return Type
`real`	Returns the **real** part.	`realMatrix`
`imaginary`	Returns the **imaginary** part.	`realMatrix`
`magnitude`	Returns the **magnitude**.	`realMatrix`
`angle`	Returns the **phase angle**.	`realMatrix`
`conjugate`	Returns the **complex conjugate**.	`complexMatrix`

4.2.13.3 Quaternion Matrix

The elements of a Quaternion Matrix are of type quaternion. A variable, **<var>**, can be defined as Quaternion Matrix by four different methods.

First method:

```
<var> =     | <element>, <element>, … , <element> |,
            | <element>, <element>, … , <element> |,
                              ⋮
            | <element>, <element>, … , <element> | (3)
```

<element>s can be of type Long, Double, Imaginary, Complex or Quaternion, but if at least one of the **<element>**s is of type Quaternion, all the matrix elements will be of type Quaternion.

Example:

The matrix from the second method can be created by the following way:

```
mat = |1.5+i+2j+3.5k,  4.7+1.5i,      7.3 |,
      |            2,        5i, 8.22+0.3i|,
      |        1e-3i,      6+2i,      9+k |;
```

Will result in the following the matrix:

$$mat = \begin{bmatrix} 1.5 + i + 2j + 3.5k & 4.7 + 1.5i & 7.3 \\ 2.0 & 5.0i & 8.22 + 0.3i \\ 0.003i & 6.0 + 2i & 9.0 + 0i + 0j + k \end{bmatrix}$$

Second method:

```
<var> = quaternionMatrix(<numRow>,<numColumn>
<element>,<element>, … ,<element>) (1)
```

<numRow> and **<numColumn>** are number or rows and the number columns in the matrix. The total number of elements (TNOE) equals to **<numRow>** times **< numColumn>**.

If there are no other parameters, all the elements in the matrix will be initialized to complex numbers, whose real and imaginary parts are zeros. If a third parameter, **<element>**, of type Integer, Real, Imaginary, Complex or Quaternion is provided; all the elements in the matrix will be initialized to the value of the third parameter. If more than one **<element>** are provided, these **<element>**s will be used in an attempt to fill the matrix. If the number of **<element>**s (NOE) is equal to TNOE, the matrix will be filled exactly. If NOE is less than TNOE, the rest of the elements will be set to quaternions, whose scalar and vector parts are zeros. If the NOE is greater than TNOE, the excess **<element>**s will be ignored.

Example:

```
mat = quaternionMatrix(2, 3, 1+2i+3j+4k, 2.0+4j, 0.5i, 4+3k, 5+2.5i, 6)
```

Will result in the following matrix:

$$mat = \begin{bmatrix} 1.0 + 2i + 3j + 4k & 2.0 + 0i + 4j + 0k & 0.5i \\ 4.0 + 0i + 0j + 3k & 5.0 + 2.5i & 6.0 \end{bmatrix}$$

Third Method:

```
<var> = quaternionMatrix(<quaternion_realVector>,< quaternion_realVector>
, … ,< quaternion_realVector>) (2)
```

<complexVector>s are of type vector defined previously. All **<complexVector>**s must have the same length. Vectors are represented as columns of the matrix. Any number of **<complexVector>**s can be provided.

Example:

```
Vec1 = quaternion_realVector(1.5+i+2j+3.5k, 2, … ,1e-3+0.1i)
Vec2 = [4.7+1.5i, 5i, 6+2i];
Vec3 = [7.3, 8.22+0.3i, 9+k];
mat = quaternionMatrix(vec1, vec2, vec3)
```

Will result in the following the matrix:

$$mat = \begin{bmatrix} 1.5 + i + 2j + 3.5k & 4.7 + 1.5i & 7.3 \\ 2.0 & 5.0i & 8.22 + 0.3i \\ 0.003i & 6.0 + 2i & 9.0 + 0i + 0j + k \end{bmatrix}$$

Fourth Method:

```
<varS> =    | <real_element>, < real_element>, … , < real_element>  |,
            | < real_element>, < real_element>, … , < real_element> |,
                                   ⋮
            | < real_element>, < real_element>, … , < real_element> |
```

```
<varI> =     | <real_element>, < real_element>, … , < real_element>  |,
             | < real_element>, < real_element>, … , < real_element> |,
                                      ⋮
             | < real_element>, < real_element>, … , < real_element> |


<varJ> =     | <real_element>, < real_element>, … , < real_element>  |,
             | < real_element>, < real_element>, … , < real_element> |,
                                      ⋮
             | < real_element>, < real_element>, … , < real_element> |

<varK>  =    | <real_element>, < real_element>, … , < real_element>  |,
             | < real_element>, < real_element>, … , < real_element> |,
                                      ⋮
             | < real_element>, < real_element>, … , < real_element> |


<var> = quaternionMatrix(<varS>, <varI>, <varJ>, <varK>);
```

Example:

```
matS = |1.5, 4.7, 7.3 |,
       |  2,    0, 8.22|,
       |  0,    6,    9 |;


matI = |   1, 1.5,    0|,
       |   0,    5, 0.3|,
       |1e-3,    2,    0|;


matJ = | 2, 0, 0|,
       | 0, 0, 0|,
       | 0, 0,  0|;

matK = |3.5, 0, 0|,
       |  0, 0, 0|,
       |  0, 0, 1|;
```

```
mat = quaternionMatrix(matS, matI, matj, matK);
```

Will result in the following the matrix:

$$mat = \begin{bmatrix} 1.5 + i + 2j + 3.5k & 4.7 + 1.5i & 7.3 \\ 2.0 & 5.0i & 8.22 + 0.3i \\ 0.003i & 6.0 + 2i & 9.0 + 0i + 0j + k \end{bmatrix}$$

Attributes for Quaternion Matrix type are given in the table below:

Call Signature	Description	Return Type
`scalar`	Returns the **scalar** part.	`realMatrix`
`i`	Returns the **i component of the vector** part.	`realMatrix`
`j`	Returns the **j component of the vector** part.	`realMatrix`
`k`	Returns the **k component of the vector** part.	`realMatrix`
`norm`	Returns the **norm**.	`realMatrix`
`unit`	Returns the **unit quaternion**.	`quaternionMatrix`
`conjugate`	Returns the **complex conjugate**.	`quaternionMatrix`
`reciprocal`	Returns the **reciprocal quaternion**.	`quaternionMatrix`

4.2.13.4 Polynomial Matrix

The elements of a Polynomial Matrix are of type polynomial. A variable, **`<var>`**, can be defined as Polynomial Matrix by three different methods.

First method:

```
<var> =      | <element>, <element>, … , <element> |,
             | <element>, <element>, … , <element> |,
                              ⋮
             | <element>, <element>, … , <element> |
```

If all the **`<element>`**s are of type polynomial, a polynomial matrix will be created.

Example:

```
mat = |  #1,2#,   #3,4#|,
      |#1,2,3#, #5,6,7#|
```

Will result in the following the matrix:

$$mat = \begin{bmatrix} x+2 & 3x+4 \\ x^2+2x+3 & 5x^2+6x+7 \end{bmatrix}$$

Second Method:

```
<var> = polynomialMatrix(<numRow>,<numColumn>
<element>,<element>, … ,<element>) (1)
```

<numRow> and **<numColumn>** are number or rows and the number columns in the matrix. The total number of elements (TNOE) equals to **<numRow>** times **< numColumn>**.

If there are no other parameters, all the elements in the matrix will be initialized to polynomials with a single coefficient of zero value. If a third parameter, **<element>**, of type Polynomial is provided, all the elements in the matrix will be initialized to the value of the third parameter. If more than one **<element>** are provided, these **<element>**s will be used in an attempt to fill the matrix. If the number of **<element>**s (NOE) is equal to TNOE, the matrix will be filled exactly. If NOE is less than TNOE, the rest of the elements will be set to polynomials with a single coefficient of zero value. If the NOE is greater than TNOE, the excess **<element>**s will be ignored.

Example:

```
mat = polynomialMatrix(2, 3, #1#, #1,2,3#, #1,2,3,4#, #1,2,3,4,5#,
#1,2,3,4,5,6#, #1,2,3,4,5,6,7#)
```

Will result in the following matrix:

$$mat = \begin{bmatrix} 1 & x+2 & x^2+2x+3 \\ x^3+2x^2+3x+4 & x^4+2x^3+3x^2+4x+5 & x^5+2x^4+3x^3+4x^2+5x+6 \end{bmatrix}$$

Third Method:

```
<var> = polynomialMatrix
(<polynomial_realVector>,<polynomial_realVector>
, … ,<polynomial_realVector>) (2)
```

< polynomial_realVector>s are of type Polynomial Vector. All **<polynomial_realVector>**s must have the same length. Vectors are represented as columns of the matrix. Any number of **<polynomial_realVector>**s can be provided.

Example:

```
Vec1 = realVector(#1,2#, #1,2,3#)
Vec2 = [#1,2,3#, #1,2,3,4#];
mat = polynomialMatrix(vec1, vec2)
```

Will result in the following the matrix:

$$mat = \begin{bmatrix} x+2 & x^2+2x+3 \\ x^2+2x+3 & x^3+2x^2+3x+4 \end{bmatrix}$$

Attributes for Polynomial Matrix type are given in the table below:

Call Signature	Description	Return Type
`degree`	Returns the **degree** of the polynomial.	`realMatrix`
`eval(integer x)`	Returns the **value** of the polynomial for the parameter **`x`**.	`realMatrix`
`eval(real x)`	Returns the **value** of the polynomial for the parameter **`x`**.	`realMatrix`
`eval(imaginary x)`	Returns the **value** of the polynomial for the parameter **`x`**.	`complexMatrix`

4.2.13.5 Rational Matrix

The elements of a Rational Matrix are of type rational. A variable, **`<var>`**, can be defined as Rational Matrix by four different methods.

First method:

```
<var> =     | <element>, <element>, … , <element> |,
            | <element>, <element>, … , <element> |,
                              ⋮
            | <element>, <element>, … , <element> |, (3)
```

If all the **`<element>`**s are of type Rational, a rational matrix will be created.

Example:

The matrix from the second method can be created by the following way:

```
mat = |#1,2#/#1,2,3#,          #3,4#/#5,6,7#|,
      |#11,12#/#11,21,13#, #13,14#/#15,16,17#|
```

Will result in the following the matrix:

$$mat = \begin{bmatrix} \frac{x+2}{x^2+2x+3} & \frac{3x+4}{5x^2+6x+7} \\ \frac{11x+12}{11x^2+12x+3} & \frac{13x+14}{15x^2+16x+17} \end{bmatrix}$$

Second Method:

```
<var>= rational_matrix(<numRow>,<numColumn>
<element>,<element>, … ,<element>) (1)
```

`<numRow>` and **`<numColumn>`** are number or rows and the number columns in the matrix. The total number of elements (TNOE) equals to **`<numRow>`** times **`< numColumn>`**.

If there are no other parameters, all the elements in the matrix will be initialized rationals with the numerators and the denominators consist of polynomial with a single coefficient of zero value. If a third parameter, **`<element>`**, of type Rational is provided, all the elements in the matrix will be initialized to the value of the third parameter. If more than one **`<element>`** are provided, these **`<element>`**s will be used in an attempt to fill the matrix. If the number of **`<element>`**s (NOE) is equal to TNOE, the matrix will be filled exactly. If NOE is less than TNOE, the rest of the elements will be set to rationals with the numerators and the denominators consist of polynomial with a single coefficient of zero value. If the NOE is greater than TNOE, the excess **`<element>`**s will be ignored.

Example:

```
mat = rational_matrix(2, 2, #1#/#1,2,3#, #1,2,3#/#1,2,3,4#,
#11#/#11,12,13#, #11,12,13#/#11,12,13,14#,)
```

Will result in the following matrix:

$$mat = \begin{bmatrix} \frac{1}{x+2} & \frac{x+2}{x^2+2x+3} \\ \frac{11}{11x+12} & \frac{11x+12}{11x^2+12x+13} \end{bmatrix}$$

Third Method:

```
<var>= rational_matrix (<rational_realVector>,<rational_realVector>
, … ,<rational_realVector>) (2)
```

<rational_realVector>s are of type Polynomial Vector. All **<polynomial_realVector>**s must have the same length. Vectors are represented as columns of the matrix. Any number of **<rational_realVector>**s can be provided.

Example:

```
Vec1 = rational_realVector(#1#/#1,2#, #11#/#11,12#)
Vec2 = [#1,2#/#1,2,3#, #11,12#/#11,12,13#];
mat = rational_matrix(vec1, vec2)
```

Will result in the following the matrix:

$$mat = \begin{bmatrix} \frac{1}{x+2} & \frac{x+2}{x^2+2x+3} \\ \frac{11}{11x+12} & \frac{11x+12}{11x^2+12x+13} \end{bmatrix}$$

Fourth Method:

```
<var> = rational_matrix
(<numerator_polynomialMatrix>,<denominarot_polynomialMatrix>)
```

All the elements in the **<numerator_polynomialMatrix>** and **<denominarot_polynomialMatrix** must be of type Polynomial.

Example:

```
num = | #1#,   #1,2#|,
      |#11#, #11,12#|;

den = |  #1,2#,    #1,1,3#|,
      |#11,12#, #11,12,13#|;

mat = rational_matrix(num,den)
```

Will result in the following the matrix:

$$mat = \begin{bmatrix} \frac{1}{x+2} & \frac{x+2}{x^2+2x+3} \\ \frac{11}{11x+12} & \frac{11x+12}{11x^2+12x+13} \end{bmatrix}$$

Attributes for Rational Matrix type are given in the table below:

Call Signature	Description	Return Type
`num`	Returns the **numerator**.	`polynomialMatrix`
`den`	Returns the **denominator**.	`polynomialMatrix`
`eval(integer x)`	Returns the **value** of the polynomial for the parameter **x**.	`realMatrix`
`eval(real x)`	Returns the **value** of the polynomial for the parameter **x**.	`realMatrix`
`eval(imaginary x)`	Returns the **value** of the polynomial for the parameter **x**.	`complexMatrix`

4.2.14 Array

A variable, **`<var>`**, can be defined as an **array** by following ways:

```
<var> = {<element>, < element >, …, < element >};
```

`<element>` can be of any type, and several types can be mixed. Elements of an array can be accessed by specifying indices of the elements in an index operator ([]). The indices start at 1. Consecutive elements can be accessed by using a range operator (:) with the fist and the last indices of interest.

```
a[<index>] -> element value
a[<index_start>:<index_end>]-> element values.
```

Values of specific elements can be assigned by using indices and range operator in the similar manner as in accessing element values.

```
a[<index>] = element value
a[<index_start>:<index_end>] = element values.
```

Examples:

Array definitions:

```
a = {1, 2.5, 3.0, true, ‘c’, “abc”,…};
```

Accessing element values

```
a[1] -> 1
a[4] -> true.
a[1:3] -> 1, 2.5, 3.0
```

Assigning element values:

```
a[2] = 2.5;
```

4.2.15 Table

Tables are two dimensional arrays.

Examples:

```
tbl = {{'a', "abc"},{2, 3.5}};
```

4.3 Statements

4.3.1 Expression

Logical expressions can be created by combining any of the logical operators.

Example:

```
l = (a > b || c < d) && (e != 0 && f==1)
```

Algebraic expressions can be created by combining any of mathematical operators or functions. A list of available functions will be presented later in the document.

Example:

```
x = (a*b) + c*d^2 – e/f + abs(y);
```

The mathematical operators can operate on multiple data types. The table below provides a list of the valid operators, their descriptions, and the valid left and right operands types for each operator.

Operator	Operation	Left Operand Data Types	Right Operand Data Types
"+"	Plus	Integer, real, imaginary, complex, quaternion, polynomial, rational, real vector, real matrix	Integer, real, imaginary, complex, quaternion, polynomial, rational, real vector, real matrix
"-"	Minus	Integer, real, imaginary, complex, quaternion, polynomial, rational, real vector, real matrix	Integer, real, imaginary, complex, quaternion, polynomial, rational, real vector, real matrix
"*"	Multiply	Integer, real, imaginary, complex, quaternion, polynomial, rational, real vector, real matrix	Integer, real, imaginary, complex, quaternion, polynomial, rational, real vector, real matrix
"/"	Divide	Integer, real, imaginary, complex, quaternion, polynomial, rational,	Integer, real

		real vector, real matrix complex, quaternion, polynomial	
"^"	Exponent	Integer, real, imaginary, complex, quaternion, polynomial, rational, real vector, real matrix	Integer, real
"%"	Remainder	Integer, real	Integer, real
":"	Range	Integer, real	Integer, real

The Range operator (:) can be used to access elements of an **array**, **vector** or a **matrix**. A range of numbers from *a* to *b* with increments of 1 is expressed as *a:b*. And a range of numbers from *a* to *b* with increments of *c* is expressed as *a:c:b*.

Examples:

```
1:10 -> 1, 2, 3, 4, 5 ,6, 7, 8, 9, 10
1:0.5:3.5 -> 1.0, 1.5, 2.0, 2.5, 3.0, 3.5
```

4.3.2 Control Flow Statements

4.3.2.1 If Statement

The syntax for creating **if** statement is:

```
if (<logical expression>)
{
   <Statements>
}
```

or

```
if (<logical expression>)
{
   <Statements>
}
else
{
   <Statements>
}
```

or

```
if (<logical expression>)
{
   <Statements>
}
else if (<logical expression>)
{
   <Statements>
```

```
}

   .
   .
   .
else
{
   <Statements>
}
```

Example:

```
a = 10;
b = 5;
if (a < b)
{
   print("a less than b")
}
else if (a > b)
{
   print("a greater than b")
}
else
{
  print("a equal to b")
}
```

4.3.2.2 Switch Statement

The syntax for creating **switch** statement is:

```
switch ( variable_to_test )
{
   case value
   {
      <statements>
      break;
   }
   case value
   {
      <statements>
      break;
   }

   .
   .
   .
   default
   {
      <statements>
   }
}
```

Example:

```
a = 2;

switch(a)
{
   case 1
   {
      b=2
      break
   }
   case 2
   {
      b=3
      break
   }
   case 3
   {
      b=4;
      break
   }
   default
   {
      b=5;
   }
}
```

4.3.2.3 For Loop

The syntax for creating **for** loop is:

```
for <index> in <list>
{
   <statements>
}
```

Examples:

```
week = {"Monday", "Tuesday", "Wednesday", "Thursday", "Friday",
"Saturday", "Sunday"};

for i in week
{
   print(i);
}

for i in -5.0:0.5:5.0
{
   print(i);
}

for i in 1:numOfPoints_x
{
   for j in 1:numOfPoints_y
   {
      R = sqrt(x[i]^2 + y[j]^2) + 1.0e-12;
      z[i,j] = sin(R)/(R*0.1) + 1.0;
   }
```

```
}
```

4.3.2.4 Loop

The **loop** statement can be used two different ways:

1. Infinite loop with a break statement inside the loop.
2. A loop with a termination condition.

The syntax for creating the infinite loop is:

```
loop
{
   <statements>
   break;
}
```

If the **break** statement not used the loop will run forever.

The syntax for creating loop is:

```
loop
{
 <statements>
}
while(<logical expression>)
```

Example:

```
i = 0;
loop
{
   print(i);
   I = i+1;
   if (i==10) break;
}
```

Example:

```
i = 0;
loop
{
   print(i);
   I = i+1;
}
while (i < 10);
```

4.3.2.5 While Loop

The syntax for creating **while** loop is:

```
while(<logical expression>)
{
   <statements>
}
```

Example:

```
i = 0;
while (i < 10)
{
   print(i);
   i = i+1;
}
```

4.3.3 Try Catch Finally Statements

The syntax for **`try/catch/finally`** statement has three different forms. The simplest one consists of a single **`try`** and a **`catch`** block, as below:

```
try
{
   <Statements>
}
catch ( Exception )
{
   <Statements>
}
```

Multiple **`catch`** blocks can be added to a single **`try`** block, as below:

```
try
{
   <Statements>
}
catch ( <Exception> )
{
   <Statements>
}

   .
   .
   .
catch ( <Exception> )
{
   <Statements>
}
```

An optional **`finally`** block can be added:

```
try
{
   <Statements>
}
catch ( <Exception> )
{
   <Statements>
}

    .
    .
    .
catch ( <Exception> )
{
   <Statements>
}
finally
{
   <Statements>
}
```

4.4 Functions

4.4.1 Declaration

The syntax for function declaration is:

```
function <identifier> (<type> <identifier>, … ,
<type> <identifier>)
{
   statements
}
```

or

```
function <identifier> (<type> <identifier>, … ,
<type> <identifier>)
{
   statements
   return_statement
}
```

or

```
function <identifier> (<type> <identifier>, … ,
<type> <identifier> = <default_value>, …)
{
   statements
}
```

or

```
function <identifier> (<type> <identifier>, … ,
```

```
<type> <identifier> = <default_value>, …)
{
   statements
   return_statement
}
```

The symbol (…) in the parameter list indicates optional repetition of the pattern. The parameter list can optionally have default values. If values for these parameters are not provided during a function call, the default values are used. Hyper supports function overloading. Two or more functions can have the same function name if they have different parameters. In this case, the function whose parameters match the calling parameters will be executed.

Local variables can be defined anywhere in the function. If a local variable is defined using a name that is also a name of a global variable, the local variable will shadow the global variable.

The return statement is optional. If return statement is not used, the function behaves a procedure. Hyper functions can return multiple values. To return multiple values, the return values need to be put in an array and return the array.

Examples:

```
function display(real x)
{
   print(x);
}

function abc(integer x)
{
   return x*2;
}

function abc(integer x, integer y)
{
   return x*y;
}

function abc(real x, real y=2.0)
{
   return x/y;
}

function abc(real a, real b, real c)
{
   return {a*2, b*2, c*2};
}
```

4.4.2 Call

The syntax for function call is:

```
<function identifier>(<value>, <value>,…<value>);
```

or

```
<variable> = <function identifier>(<value>, <value>,…<value>);
```

or

```
(<variable>, <variable>, … ,<variable>)
 = <function identifier>(<value>, <value>,…<value>);
```

Examples:

```
t_int = abc(2);
print(t_integer);

t_int_int = abc(7,2);
print(t_int_int);

t_real = abc(4.0);
print (t_real);

t_no_default = abc(27.0,6.0);
print(t_no_default);

t_default = abc(27.0);
print(t_default);

inv_m1 = inv(m1);

(eigvec_m1, eigval_m1) = eigen(m1);
```

The last example shows multiple return values.

4.5 Class

A class is declared in the following way:

```
class <identifier>
{
 class statements;
}
```

or

```
class <identifier>
```

```
(<inherited class identifier>, … , <inherited class identifier>)
{
 class statements;
}
```

Examples:

```
class Point2D
{
   a = 10;
   this.b = 5;

   function Point2D()
   {
   }

   function Point2D(real x)
   {
      this.x = x;
   }

   function Point2D(real x, real y)
   {
      this.x = x;
      this.y = y;
   }

   function setA()
   {
      Point2D.a=15;
   }

      function setBA(real b1, real a1)
      {
         this.b = b1;
         Point2D.a = a1;
      }

}
```

The keyword **`this`** indicates that it is an object variable whose value can differ for each instance of the object. If a variable is declared without the **`this`** keyword, the variable will be a class variable, and its value will be the same in all instances of the class.

4.6 Import

4.6.1 Import Script

```
import <module_name>, … , <module_name>,
```

4.6.2 Import Java

```
import_java_class(<java_class_path>, … , < java_class_path >)
```

Examples:

```
lib_math = import_java_class("java.lang.Math")
lib_basic = import_java_class("library.Basic")
```

5 Class Libraries

Related Hyper library functions are grouped together and implemented using Java classes. One of the advantages using classes is that a primitive operation can be performed ones and multiple higher-level operations can be done subsequently without re invoking the primitive operation. For example, in linear algebra, to compute determinant, trace, inverse, or a solution, a LUP decomposition of a matrix need to be performed. Once a matrix is decomposed, determinant, trace, inverse, or a solution can be computed without re performing the decomposition. Another advantage of using class is that one can have multiple instances of the same class with different attributes. For example, we can have two different instances of the class Integrator or ODE Solver with two different step sizes or integration schemes. In some cases, a library consists of several classes. In these cases, there is a class that contains the constituent classes. The following outline illustrates the organizational structure of the libraries.

1. General Math
2. Linear Algebra
    a. Utility
    b. Linear Equations
    c. Linear Least Square
    d. Eigen
    e. Singular Value
3. Zero Min Max
    a. Root Finder
    b. Optimization
4. Analysis
    a. Differentiation
    b. Integration
    c. Ordinary Differential Equation
5. Estimation
    a. Interpolation
    b. Curve Fit
6. Stochastic
    a. Statistics
        i. Histogram
    b. Probability
        i. Uniform Distribution
        ii. Triangular Distribution
        iii. Normal Distribution
        iv. Log Normal Distribution
        v. Student Distribution

vi. Gamma Distribution
vii. Chi-Squared Distribution
viii. Exponential Distribution
ix. Laplace Distribution
x. Beta Distribution
xi. Fisher Snedecor Distribution
xii. Fisher Tippett Distribution
xiii. Weibull Distribution
xiv. Cauchy Distribution
xv. Histogrammed Distribution
xvi. All Distribution

7. Frequency Domain
   a. FFT

The functions in the libraries are accessed by first importing the library then calling a function in the library using a dot notation. A library can be imported using a command of the following format:

```
<pointer>  = import_java_class(<class path>")
```

A function in the library can be called using the following format:

```
<var> = <pointer>.<function_name>(<param>, <param>, … <param>,)
```

Example:

The determinant of a matrix can be computed the following way:

```
linEqn = import_java_class("library.linear_algebra.LinearEquations");
det = lib_la.determinant(mat);
```

where, **`linEqn`** is the pointer to the library, **`library.linear_algebra.LinearEquations`** is the class path, **`determinant`** is the name of the function, **`mat`** is a variable of type Real Matrix, and **`det`** is variable of type Real.

The library class paths and the function call signature, descriptions, and return types are documented in Appendices A – H. Descriptions of the libraries are presented in the sections below.

5.1 General Math

The General Math library contains functions that are normally found with any programming language such as Java. The functions in this library include absolutes value function, trigonometric and hyperbolic functions etc. A complete list of the functions can be found in Appendix A. Some of these functions are overloaded for various types such as Integer, Real, Vector, and Matrix.

5.2 Linear Algebra

The Linear Algebra library composed of five classes: All, Linear Equations class, Linear Least Square class, Singular Value class, and Eigen class. Each of these classes is described below.

5.2.1 Utility

The Utility class contains methods for creating various types of vectors and matrices, accessing their attributes, and performing operations specific to vectors and matrices. All the functions contained in the Utility class are listed in the Appendix B.

5.2.2 Linear Equations

The Linear Equations class contains methods for solving linear equations of number equal to the number of unknowns. These equations are transformed in the following form:

$$\boldsymbol{Ax} = \boldsymbol{b}$$

Where $\boldsymbol{A}$ is a square matrix of size nxn, represents n equations and n unknowns. $\boldsymbol{x}$ is a column vector of n unknowns, and $\boldsymbol{b}$ is a column vector of n inhomogeneous terms. $\boldsymbol{A}$ is called the coefficient matrix. Gaussian Elimination method is used to factorize the matrix $\boldsymbol{A}$ into three matrices: Lower Triangular (L), Upper Triangular (U), and Permutation (P). Hence, it's called LUP factorization. The matrix $\boldsymbol{A}$ can be LUP factorized by calling the function **`decompose`**. Once $\boldsymbol{A}$ is LUP factorized, various other operations such as computing determinant and inverse can be performed by calling functions such as **`determinant`** and **`inverse`**. All the functions contained in the Linear Equation class are listed in the Appendix B. The example below shows how to use this class.

Example:

```
linEqn = import_java_class("library.linear_algebra.LinearEquations")
```

The statement above imports the Linear Equation class and assigns to the pointer **`linEqn`**.

```
A = |3.0, -0.1, -0.2|,
    |0.1,    7, -0.3|,
    |0.3, -0.2, 10.0|
```

The statement above creates a 3x3 real matrix and assigns to the variable **`A`**.

```
b = [7.85, -19.3, 71.4];
```

The statement above creates a column vector and assigns to the variable **`b`**.

```
linEqn.decomposeLUP(A);
```

The statement above performs LUP factorization of the matrix **`A`**.

```
(l, u ,p) = linEqn.lup()
```

The statement above assigns the computed L, U, and P matrices to the variables **`l`**, **`u`**, and **`p`**.

Alternately,

```
l = linEqn.lower()
```

The statement above assigns the computed L matrix to the variable **`l`**.

```
u = linEqn.upper()
```

The statement above assigns the computed U matrix to the variable **`u`**.

```
p = linEqn.permutation()
```

The statement above assigns the computed P matrix to the variable **`p`**.

```
det  = linEqn.determinant();
```

The statement above computes the determinant of the matrix **`A`** and assigns it to the variable **`det`**.

```
tr  = linEqn.trace();
```

The statement above computes the trace of the matrix `A` and assigns it to the variable **`tr`**.

```
inv = linEqn.inverse();
```

The statement above command above computes the inverse matrix of **`A`** and assigns it to the variable **`inv`**.

```
solb = linEqn.solve(b);
```

The statement above computes the solution corresponding the inhomogeneous terms column **`b`** and assigns it to the variable **`solb`**. Note that to compute solution for another inhomogeneous terms column, say **c**, the coefficient matrix **A** does not need to be factorized again. The solution can be obtained simply using the following statement:

```
solc = linEqn.solve(c);
```

The solutions for both inhomogeneous terms columns can also be computed using a single call by combining **c** and **d** into a matrix (**bc**) and passing this matrix as a parameter as in the following statements:

```
bc = realMatrix(b,c);
solbc = linEqn.solve(bc);
```

Note: For each of the function call above, the matrix **A** can be passed in as a parameter. In this case, **A** will be factorized for each function call.

5.2.3 Linear Least Squares

The Linear Least Squares class contains methods for solving linear equations, where the number of equations are not equal to the number of unknowns.

When the number of equations is greater than the number of unknowns, it is called an overdetermined system. Linear least square is the problem of approximately solving an overdetermined system of linear equations, where the best approximation is defined as that which minimizes the sum of squared differences between the data values and their corresponding modeled values. The approach is called "linear" least squares since the assumed function is linear in the parameters to be estimated.

Let $\boldsymbol{Ax} = \boldsymbol{b}$ be an overdetermined linear equation. Where $\boldsymbol{A}$ is an *mxn* matrix with $m > n$. If $\boldsymbol{b}$ is not in the column space of $\boldsymbol{A}$, the system is inconsistent and the equation cannot be solved for $\boldsymbol{x}$. In this case, a least-squares solution can be found by minimizing

$$\|\boldsymbol{Ax} - \boldsymbol{b}\| = \left(\sum_{i=1}^{m}\left(\sum_{j=1}^{n} a_{ij}x_j - b_i\right)^2\right)^{1/2}$$

$\boldsymbol{r} = \boldsymbol{Ax} - \boldsymbol{b}$ is called the residual or error. $\boldsymbol{x}$ with the smallest residual norm $\|\boldsymbol{r}\|$ is called the least-squares solution, which is equivalent to minimizing $\|\boldsymbol{Ax} - \boldsymbol{b}\|^2$. All the functions contained in the Linear Least Squares class are listed in the Appendix B.

Example:

```
linLS = import_java_class("library.linear_algebra.LinearLeastSquare");
```

The statement above imports the Linear Equation class and assigns to the pointer **linLS**.

```
A = |-2, 1|,
    |-1, 1|,
    | 1, 1|,
    | 2, 1|,
    | 1, 2|;
```

The statement above creates a 5x2 real matrix and assigns to the variable **A**.

```
b = [0, 1, 2, 2, 3];
```

The statement above creates a column vector and assigns to the variable **b**.

```
linLS.decomposeQR(A);
```

The statement above performs QR factorization of the matrix **A**.

```
(Q, R) = linLS.qr()
```

The statement above assigns the computed Q, and R matrices to the variables **Q**, and **R**.

Alternately,

```
Q = linLS.getQ()
```

The statement above assigns the computed Q matrix to the variable **Q**.

```
R = linLS.getR()
```

The statement above assigns the computed R matrix to the variable **R**.

```
inv = linLS.inverseLS();
```

The statement above computes the least squares inverse matrix of **A** and assigns it to the variable **inv**.

```
x = lib_lls.solveLS(b);
```

The statement above computes the least squares solution corresponding the column **b** and assigns it to the variable **x**.

5.2.4 Eigen

The Eigen class contains methods for computing eigenvalues and eigenvectors of a real or complex square matrix.

An eigenvector or characteristic vector of a square matrix is a vector that only changes its magnitude (length), but does not change its direction under the associated linear transformation. In other words—if $\boldsymbol{x}$ is a vector that is not zero, then it is an eigenvector of a square matrix $\boldsymbol{A}$ if $\boldsymbol{Ax}$ is a scalar multiple of $\boldsymbol{x}$. This condition could be written as the equation. An eigenvector is a nonzero vector that satisfies the equation

$$\boldsymbol{Ax} = \lambda \boldsymbol{x}$$

Where $\boldsymbol{A}$ is an *nxn* square matrix, scalar λ is called the eigenvalue of $\boldsymbol{A}$ and $\boldsymbol{x}$ is called the eigenvector of A corresponding to λ. Eigenvalues and eigenvectors are also called proper roots and proper vectors(“eigen” is German for the word “own” or “proper”) or characteristic roots and characteristic vectors or latent roots and latent vectors. Geometrically, the equation $\boldsymbol{Ax} = \lambda \boldsymbol{x}$ implies that $\boldsymbol{Ax}$ *and* $\boldsymbol{x}$ are parallel. An eigenvector corresponding to a real, nonzero eigenvalue points in a direction that is stretched by the transformation and the eigenvalue is the factor by which it is stretched. If the eigenvalue is negative, the direction is reversed. Eigenvalues and eigenvectors can be either real or complex. All the functions contained in the Eigen class are listed in the Appendix B.

Example 1:

```
eig  = import_java_class("library.linear_algebra.Eigen");
```

The statement above imports the Eigen class and assigns to the pointer **`eig`**.

```
A = |3.0, -0.1, -0.2|,
    |0.1,    7, -0.3|,
    |0.3, -0.2, 10.0|;
```

The statement above creates a real matrix and assigns to the variable `A`.

```
(eigval, eigvec) = lib_eigen.eigen(A);
```

The statement above computes eigenvalues and eigen vectors of a real matrix **`A`** and assigns to the variables **`eigval`** and **`eigvec`** respectively.

Example 2:

```
C = | 1+3i, 2+1i, 3+2i, 1+i |,
    | 3+4i, 1+2i, 2+1i, 4+3i|,
```

```
| 2+3i, 1+5i, 3+1i, 5+2i|,
| 1+2i, 3+1i, 1+4i, 5+3i|;
```

The statement above creates a complex matrix and assigns to the variable **C**.

```
(eigval, eigvec) = lib_eigen.eigen(C);
```

The statement above computes eigenvalues and eigen vectors of a complex matrix **C** and assigns to the variables **eigval** and **eigvec** respectively.

5.2.5 Singular Value

The Singular Value class contains methods to perform singular value decomposition, compute pseudo inverse, and solve over determined and under determined system of linear equations.

Given a complex matrix A having m rows and n columns, the matrix product $U \Sigma V^*$ (* denotes conjugate transpose) is a singular value decomposition for a given matrix A if

- U and V, respectively, have orthonormal columns.
- Σ has nonnegative elements on its principal diagonal and zeros elsewhere.
- $A = U \Sigma V^*$.

Let p and q be the number of rows and columns of Σ. U is $m \times p$, $p \le m$, and V is $n \times q$ with $q \le n$.

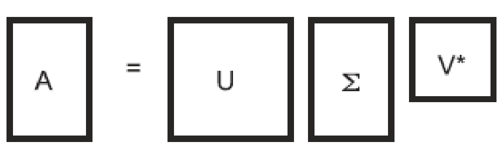
There are three standard forms of the SVD. All have the ith diagonal value of Σ denoted σ_i and ordered as follows: $\sigma_1 \ge \sigma_2 \ge \cdots \ge \sigma_k$, and r is the index such that $\sigma_r > 0$ and either $k = r$ or $\sigma_r+1 = 0$.

1. $p = m$ and $q = n$. The matrix Σ is $m \times n$ and has the same dimensions as A.

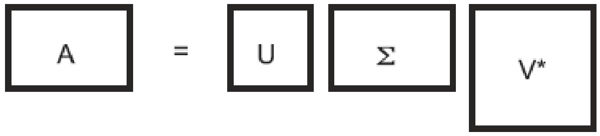
2. $p = q = min\{m, n\}$.The matrix Σ is square.

3. If $p = q = r$, the matrix Σ is square. This form is called a reduced SVD and denoted by $\hat{U}\hat{\Sigma}\hat{V}^*$

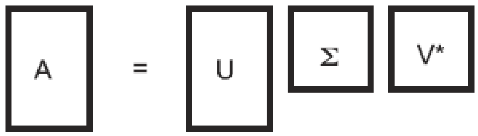
The three standard forms are graphically illustrated



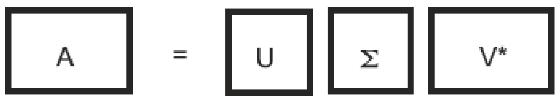
The first form of the singular value decomposition where m < n.



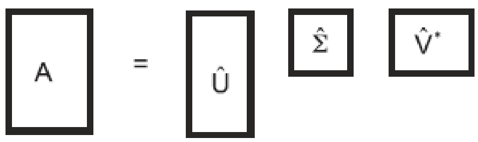
The second form of the singular value decomposition where m ≥ n.



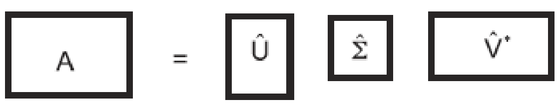
The second form of the singular value decomposition where m < n.



The first form of the singular value decomposition where m ≥ n.



The third form of the singular value decomposition where r ≤ n ≤ m.



The third form of the singular value decomposition where r ≤ m < n.

In the first standard form of the SVD, U and V are unitary. If A is real, then U and V (in addition to Σ) can be chosen real in any of the forms of the SVD. The singular value decomposition $U \Sigma V^*$ is not unique. If $U \Sigma V^*$ is a singular value decomposition, so is $(-U)\, \Sigma\, (-V^*)$. The singular values may be arranged in any order if the columns of singular vectors in U and V are reordered correspondingly. All the functions contained in the Singular Value class are listed in the Appendix B.

Example:

```
lib_svd = import_java_class("library.linear_algebra.SingularValue");
```

The statement above imports the Singular Value class and assigns to the pointer **`lib_svd`**.

```
A1 = |22, 10,  2,   3,  7|,
     |14,  7, 10,   0,  8|,
     |-1, 13, -1, -11,  3|,
     |-3, -2, 13,  -2,  4|,
     | 9,  8,  1,  -2,  4|,
     | 9,  1, -7,   5, -1|,
     | 2, -6,  6,   5,  1|,
     | 4,  5,  0,  -2,  2|;
```

The statement above creates a real matrix and assigns to the variable **`A1`**.

```
(u,s,v) = lib_svd.svd(A1);
```

The statement above first factorizes the matrix `A1` then assigns the computed `U`, `S`, and `V` matrices to the variables **`u`**, **`s`**, and **`v`**.

```
A2 = lib_svd.pseudoinverse();
```

The statement above computes pseudo inverse of the matrix **`A1`** from the already factorized matrix u and assigns to the variable **`A2`**.

```
b = [-1, 2, 1, 4, 0, -3, 1, 0];
```

The statement above creates a real vector and assigns to the variable **`b`**.

```
x = lib_svd.solveSVD(b) ;
```

The statement above computes the over determined solution of **`A1`** corresponding to the column **`b`** and assigns it to the variable **`x`**.

5.3 Zero Min Max

The Zero Min Max library contains methods to find zeros, maxima, and minima of a function. The Zero Max Min library contains three classes Root Finder, Optimizer, and All. The All class contains functions from the Root Finder and Optimizer classes.

5.3.1 Root Finder

The objective of a root finder is to compute solutions of the equation

$$f(x) = 0$$

Two different methods are available to find roots of a function: Bisection method and Newton method. All the functions contained in the Root Finder class are listed in the Appendix C.

Example:

```
rf  = import_java_class("library.function_eval.RootFinder")
```

The statement above imports the Root class and assigns to the pointer **`rf`**.

```
function eqn(real x)
{
  y = x^3+x-1;
  return y;
}
```

The code fragment above sets up an equation whose root is to be determined.

```
result = rf.bisection("eqn(real)", 0.0, 1.0, 0.5e-7);
```

The statement above computes a root using the Bisection method with the bracketing values of 0.0 and 1.0 and a relative precision value of 0.5e-7.

```
result = rf.newton("eqn(real)", 0.0, 0.5e-7);
```

The statement above computes a root using the Newton method with the initial guess of 0.0 and a relative precision value of 0.5e-7.

Alternatively, the same result can be achieved by first setting the root finder method using one of the following statements:

```
setBisection();
```

or

```
setNewton();
```

Then setting up an equation using the following statement:

```
setFunction("eqn(real x)");
```

Then setting the relative precision using the following statement:

```
setPrecision(0.5e-7);
```

After this the multiple attempts to find a root can be made with different bracketing values (for Bisection method) or initial guesses (for Newton method) using one of the following statements:

```
bisection(0.0, 1.0);
```

or

```
newton(0.0);
```

5.3.2 Optimizer

An optimization problem can be represented in the following way:

> *Given:* a function $f : A \longrightarrow \mathbf{R}$ from some set A to the real numbers
>
> *Sought:* an element x_0 in A such that $f(x_0) \leq f(x)$ for all x in A ("minimization") or such that $f(x_0) \geq f(x)$ for all x in A ("maximization").

optimization problems are usually stated in terms of minimization. Generally, unless both the objective function and the feasible region are convex in a minimization problem, there may be several local minima. A *local minimum* x^* is defined as a point for which there exists some $\delta > 0$ so that for all x such that

$$\|X - x^*\| \leq \delta$$

the expression

$$f(x^*) \leq f(x)$$

holds; that is to say, on some region around x^* all of the function values are greater than or equal to the value at that point. Local maxima are defined similarly. Two different optimization strategies are available: Powell (also known as hill climbing) and Simplex.

All the functions contained in the Optimizer class are listed in the Appendix C.

Example:

```
opt = import_java_class("library.function_eval.Optimizer")
```

The statement above imports the Optimizer class and assigns to the pointer `opt`.

```
function banana(realVector x)
```

```
{
  return 100*(x[2] - x[1]*x[1])*(x[2] - x[1]*x[1])
         +(1 - x[1])*(1 - x[1]);
}
```

The code fragment above defines the Rosenbrock's banana function whose optimum is to be determined.

```
result = opt.simplex("banana(realVector x)", [10, -10]);
```

The statement above computes the optimum of the Rosenbrock's banana function using the Simplex method. The first parameter is the signature of the function, and the second parameter is a vector whose elements are the initial values.

```
 result = opt.powell("banana(realVector x)", [10, -10]);
```

The statement above computes the optimum of the same function using the Powell (Hill Climbing) method.

```
print(result);
```

The statement above prints the computed values.

Alternately, the same results can be obtained by first setting the optimizer using the following statement:

```
opt.setOptimizer("simplex");
```

or

```
opt.setOptimizer("powell");
```

Then setting an optimization strategy as:

```
opt.setStrategy("max");
```

or

```
opt.setStrategy("min");
```

Then setting the function as:

```
opt.setFunction("banana(realVector x)");
```

Then setting the initial values as:

```
opt.setGuess([10, -10]);
```

Then computing the optimum values as:

```
result = opt.optimize();
```

5.4 Analysis

The Analysis library contains methods to compute definite integrals and derivatives of a function and solutions of ordinary differential equations. The Analysis library contains four classes: Differentiator, Integrator, ODE Solver, and All. The All class contains functions from the Root Finder and Optimizer classes. All the functions contained in the Analysis library are listed in the Appendix D.

5.4.1 Differentiator

The definition of a derivative

$$f'(x) = \lim_{x \to 0} \frac{f(x+h) - f(x)}{h}$$

Assuming the limit exists; i.e. the function is differentiable. The derivative of a function at $\boldsymbol{x}$ can be approximated by

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

Where h is a very small positive number.

The above formula is called Forward Difference method. The Differentiator class uses a more accurate formula called the Centered Difference method which is given as:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

In vector calculus, the Jacobian matrix is the matrix of all first-order partial derivatives of a vector-valued function. The Jacobian matrix J of $\boldsymbol{f}$ is an $m \times n$ matrix, usually defined and arranged as follows:

$$\boldsymbol{J} = \frac{d\boldsymbol{f}}{d\boldsymbol{x}} = \begin{bmatrix} \frac{\partial \boldsymbol{f}}{\partial x_1} & \cdots & \frac{\partial \boldsymbol{f}}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

or, component-wise:

$$J_{i,j} = \frac{\partial f_i}{\partial x_j}$$

The Differentiator class contains functions to compute derivative of a function at a given point, derivative of a vector with respect to another vector, derivative of a polynomial, and Jacobian matrix of a system of functions.

All the functions contained in the Differentiator class are listed in the Appendix D.

Example 1:

```
dif = import_java_class("library.analysis.Differentiator");
```

The statement above imports the Differentiator class and assigns to the pointer **`dif`**.

```
function func(real x)
{
   return -0.1*x^4 -0.15*x^3 - 0.5*x^2 - 0.25*x + 1.2;
}
```

The code fragment above defines the function to be differentiated.

```
result = dif.dydx("func(real x)",0.5, 1e-6);
```

The statement above computes the derivative of the function at 0.5. The first parameter is the signature of the function, the second parameter is the value at which the derivative is computed, and the third parameter is the step size.

```
print(result);
```

The statement above prints the computed values.

Example 2:

```
x = -10:0.25:10;
```

The statement above creates a vector whose elements range from -10.0 to 10.0 in increments of 0.25 and assigns it to variable **`x`**.

```
y = x1^2;
```

The statement above creates a vector whose elements are squares of the elements of the vector **x** and assigns it to variable **y**.

```
dydx = dif.dydx(x, y);
```

The statement above differentiates vector y with respect to vector **x** and assigns to variable **dydx**.

```
print(dydx);
```

The statement above print the variable **dydx**.

Example 3:

```
poly = #1,2,3#;
```

The statement above creates the following polynomial:

$$x^2 + 5x + 3$$

```
dpoly = dif.derivative(poly);
```

The statement above computes the derivative of the polynomial.

```
print(dpoly);
```

The statement above print the following output:

```
dpoly = 2X + 5
```

Example 4:

```
function func2(realVector x)
{
   y = zeros(3);
   y[1] = x[1]*x[1]*x[1] + x[2];
   y[2] = x[2]*10.0 + x[2]*x[1]*x[1];
   y[3] = x[1]*x[2];
   return y;
}
x = [2, 1];
```

The code fragment above defines a system of functions whose Jacobian matrix is to be computed at point $x = (2,1)$. The point **x** is specified using a vector.

```
jcob = dif.jacobian("func2(realVector x)", x);
```

The statement above computes the Jacobian matrix at point **x** and assigns it to a variable **jcob**.

```
print(jcob);
```

The statement above prints the computed Jacobian matrix.

5.4.2 Integration

Given a function f of a real variable *x* and an interval *[a, b]* of the real line, the definite integral

$$\int_a^b f(x)dx$$

is defined informally as the signed area of the region in the xy-plane that is bounded by the graph of f, the x-axis and the vertical lines x = a and x = b. The area above the x-axis adds to the total and that below the x-axis subtracts from the total.

A double integral is defined as

$$\iint_\Omega g(x,y)dxdy$$

where Ω is a triangle with vertices (x_i, y_i), (x_j, y_j), and (x_k, y_k) and g is real valued.

There are two types of integrators in the Integrator class: definite integrators and polynomial integrators. Several integration schemes are available for definite integrators: Simpson, Simpson Richardson, Romberg, Quadrature, and Tricube. Tricube is a double integrator, and all others are line integrators.

All the functions contained in the Integrator class are listed in the Appendix D.

Example 1:

```
integ = import_java_class("library.analysis.Integrator");
```

The statement above imports the Integrator class and assigns to the pointer **integ**.

```
function integrand(real x)
{
   y = (10*exp(-x)*sin(2*PI*x))^2;
   return y;
}
```

The code fragment above defines the function to be integrated.

```
integ.setFunction("integrand(real)");
```

The statement above sets up the integrand.

```
result = integ.simpson(0.0, 0.5);
```

The statement above performs integration from 0.0 to 0.5 using Simpson method and assigns the integrated value to the variable **result**.

```
result = integ.simpsonRichardson (0.0, 0.5);
```

The statement above performs integration from 0.0 to 0.5 using Simpson Richardson method and assigns the integrated value to the variable **result**.

```
result = integ.romberg (0.0, 0.5);
```

The statement above performs integration from 0.0 to 0.5 using Romberg method and assigns the integrated value to the variable **result**.

```
result = integ.quadrature (0.0, 0.5);
```

The statement above performs integration from 0.0 to 0.5 using Quadrature method and assigns the integrated value to the variable **result**.

Alternatively, the same results can be achieved by the following statements:

```
result = integ.simpson("integrand(real)", 0.0, 0.5)
result = integ.simpsonRichardson("integrand(real)", 0.0, 0.5)
result = integ.romberg("integrand(real)", 0.0, 0.5)
result = integ.quadrature("integrand(real)", 0.0, 0.5)
```

Example 2:

Perform the following integration

$$\int (x + 5)\, dx$$

Solution:

```
poly = #1, 5#;
```

The statement above creates the following polynomial:

$$poly = x + 5$$

and assigns the polynomial to the variable **poly**.

```
ipoly = integ.integral(poly, 3.0);
```

The statement above integrates the polynomial, adds a constant value of 3.0 to the integration and assigns the integrated polynomial to the variable **ipoy**. The integrated polynomial is

$$0.5x^2 + 5x + 3$$

Example 3:

Evaluate the double integral

$$\iint\limits_{\Omega} cos(x)cos(y)dxdy$$

over the triangle Ω in the *x-y* plane with vertices (0.0), $(0.0, \pi/2)$, $(\pi/2\,, \pi/2)$.

Solution:

```
function real_integrand(real x, real y)
{
   y = cos(x)*cos(y);
   return y;
}
```

The code fragment above defines the function to be double integrated.

```
integ.setFunction("integrand4(real, real)");
```

The statement above sets up the **real_integrand** function for integration.

```
result = integ.tricub(0.0,0.0, 0.0,PI/2,PI/2,PI/2, 1e-6);
```

The statement above performs a double integration over the triangle Ω in the *x-y* plane with vertices (0.0), $(0.0, \pi/2)$, $(\pi/2\,, \pi/2)$ within an absolute error value of 1e-6 and assigns the integrated value to the variable **result**.

5.4.3 Ordinary Differential Equation

A classical ordinary differential equation (ODE) is a functional relation of the form

$$F\left(t, x, x^{(1)}, \cdots, x^{(k)}\right) = 0$$

For unknown function $x \in C^k(J), J \subseteq \mathbb{R}$ and derivatives

$$x^{(j)}(t) = \frac{d^j(t)}{dt^j}, \quad j \in \mathbb{N}_o$$

where t is the independent variable and x the depended variable. The highest derivative appearing in F is called the order of the differential equation. A solution of the ODE is a function $\phi \in C^k(I)$, where $I \subseteq J$ is an interval such that

$$F\left(t, \phi(t), \phi^1(t) \cdots \phi^k(t)\right) = 0 \qquad \text{for all } t \in I$$

The ODE Solver class contains two solver methods: Euler and Runge Kutta 4.

All the functions contained in ODE Solver class are listed in the Appendix D.

Example:

Solve the following ODE

$$\frac{d^2y}{dt^2} = -32$$

for the initial condition $y(0) = 0$ and $\frac{dy}{dt}(0) = 100$.

Solution:

```
solver = import_java_class("library.analysis.ODE_Solver")
```

The statement above imports the Integrator class and assigns to the pointer **integ**.

```
function odefcn(real x)
{
  return -32.0;
}
```

The code fragment above defines the ODE.

```
ic = [0.0, 100.0];
```

The statement above sets up the initial condition.

```
ye = solver.euler("odefcn(real x)", ic, 0.0, 7.0, 0.01);
```

The statement above solves the ODE for the initial condition from 0.0 to 7.0 with the step size of 0.01 using the Euler method and assigns the solution to the variable **ye**.

```
yr = solver.rk4("odefcn(real x)", ic, 0.0, 7.0, 0.01);
```

The statement above solves the ODE for the initial condition from 0.0 to 7.0 with the step size of 0.01 using the Runge Kutta 4 method and assigns the solution to the variable **`yr`**.

5.5 Estimation

The Estimation library contains methods for curve fits and interpolation. The Estimation library consists of four classes: Interpolator, Linear Regression, Polynomial Least Square, and All. All the functions in the Estimation library are listed in Appendix E.

5.5.1 Interpolation

Interpolation is a method of constructing new data points within the range of a discrete set of known data points.

Linear interpolation formula is given by

$$f_1(x) = f(x_0) + \frac{f(x_1) - f(x_0)}{x_1 - x_0}(x - x_0)$$

Lagrange interpolation polynomial is given by

$$P_n(x) = \sum_{i=0}^{n} \frac{\prod_{j \neq i}(x - x_j)}{\prod_{j \neq i}(x_i - x_j)} y_i$$

Newton interpolation formula is given by

$$P_n(x) = \alpha_0 + (x - x_0) \cdot \Big[\alpha_1 + (x - x_1) \cdot [\cdots [\alpha_{n-1} + \alpha_n \cdot (x - x_1)]]\Big]$$

Neville's algorithm is given by

$$\begin{cases} \Delta_{j,i+j}^{\text{left}}(x) = \dfrac{x_i - x}{x_j - x_{i+j+1}} \left[\Delta_{j+1,i}^{\text{left}}(x) - \Delta_{j,i}^{\text{right}}(x)\right] \\[2ex] \Delta_{j,i+j}^{\text{right}}(x) = \dfrac{x_{i+j+1} - x}{x_j - x_{i+j+1}} \left[\Delta_{j+1,i}^{\text{left}}(x) - \Delta_{j,i}^{\text{right}}(x)\right] \end{cases}$$

where

$$\begin{cases} \Delta_{j,i}^{\text{left}}(x) = P_j^i(x) - P_j^{i-1}(x) \\[2ex] \Delta_{j,i}^{\text{right}}(x) = P_j^i(x) - P_{j+1}^{i-1}(x) \end{cases}$$

where

$$P_j^i(x) = \frac{(x - x_{i+j})P_j^{i-1}(x) + (x - x_{i+j})P_{j+1}^{i-1}(x)}{x - x_{i+j}}$$

The expression for cubic spline is given by

$$P_{i(x)} = y_{i-1}A_i(x) + y_iB_i(x) + y''_{i-1}C_i(x) + y_i''D_i(x)$$

where

$$\begin{cases} A_i(x) = \dfrac{x_i - x}{x_i - x_{i-1}} \\ \\ B_i(x) = \dfrac{x - x_{i-1}}{x_i - x_{i-1}} \end{cases}$$

$$y''_{i-1} = \left.\frac{d^2P(x)}{dx^2}\right|_{x=x_{i-1}}$$

$$y_i'' = \left.\frac{d^2P(x)}{dx^2}\right|_{x=x_i}$$

$$\begin{cases} C_i(x) = \dfrac{[A_i(x)^2 - 1]}{6}(x_i - x_{i-1})^2 \\ \\ D_i(x) = \dfrac{[B_i(x)^2 - 1]}{6}(x_i - x_{i-1})^2 \end{cases}$$

$$\frac{dP_i(x)}{dx} = \frac{dP_{i+1}(x)}{dx}$$

The Interpolator class contains Linear, Lagrange, Newton, Neville, and Spline interpolation methods. All the functions contained in the Interpolator class are listed in Appendix E.

Example 1:

Given:

$$x = (1900, 1910, 1920, 1930, 1940, 1950, 1960, 1970, 1980, 1990)$$

and

$$y = (75.995, 91.972, 105.711, 123.203, 131.669, 150.697, 179.323, 203.212, 226.505, 249.633)$$

find the y value corresponding to $x = 1975$.

Solution:

```
interpol = import_java_class("library.estimation.Interpolator");
```

The statement above imports the Interpolator class and assigns to the pointer **interpol**.

```
x = 1900.0:10:1990;
```

The statement above creates a vector whose elements are from 1900 to 1990 in the increments of 10 and assigns the vector to the variable **x**.

```
y = [75.995, 91.972, 105.711, 123.203, 131.669, 150.697, 179.323, 203.212,
226.505, 249.633];
```

The statement above creates a vector with the given data and assigns the vector to the variable **y**.

```
lin = interpol.linear(t,p,1975.0);
```

The statement above interpolates **y** for **x** value of 1975 using Liner interpolation and assigns the interpolated value to the variable **lin**.

```
sp = interpol.spline(t,p,1975.0);
```

The statement above interpolates **y** for **x** value of 1975 using Spline interpolation and assigns the interpolated value to the variable **sp**.

```
lag = interpol.lagrange(t,p,1975.0);
```

The statement above interpolates **y** for **x** value of 1975 using Lagrange interpolation and assigns the interpolated value to the variable **lag**.

```
nwt = interpol.newton(t,p,1975.0);
```

The statement above interpolates **y** for **x** value of 1975 using Newton interpolation and assigns the interpolated value to the variable **nwt**.

```
nev = interpol.neville(t,p,1975.0);
```

The statement above interpolates **y** for **x** value of 1975 using Neville interpolation and assigns the interpolated value to the variable **nev**.

After setting up an interpolator, the interpolator can be repeatedly used to interpolate for different x values.

Example 2:

For the x and y values from Example 1, compute interpolated values for x = 1945, x = 1963, x = 1978, and x = 1987 using Linear interpolation method.

Solution:

```
interpol.setLinear(t,p);
```

The statement above sets up the interpolator to use Linear method.

```
l_1945 = interpol.interpolate(1945.0);
```

The statement above interpolates **y** for **x** value of 1945 and assigns the interpolated value to the variable **l_1945**.

```
l_1963 = interpol.interpolate(1963.0);
```

The statement above interpolates **y** for **x** value of 1963 and assigns the interpolated value to the variable **l_1963**.

```
l_1978 = interpol.interpolate(1978.0);
```

The statement above interpolates **y** for **x** value of 1978 and assigns the interpolated value to the variable **l_1978**.

```
l_1987 = interpol.interpolate(1987.0);
```

The statement above interpolates **y** for **x** value of 1987 and assigns the interpolated value to the variable **l_1987**.

5.5.2 Linear Regression

Regression analysis estimates the conditional expectation of the dependent variable given the independent variables – that is, the average value of the dependent variable when the independent variables are fixed. In regression analysis, it is also of interest to characterize the variation of the dependent variable around the regression function which can be described by a probability distribution.

The method of *least-square fit* is a standard approach in regression analysis to the approximate solution of overdetermined systems, i.e., sets of equations in which there are more equations than unknowns. The least-square estimation is obtained by minimizing function $s(\boldsymbol{p})$ is given as

$$s(\boldsymbol{p}) = \sum_{i=1}^{N} \frac{[y - F(x, \boldsymbol{p})]^2}{\sigma_i^2}$$

with respect to the parameter $\boldsymbol{p}$. "Least squares" means that the overall solution minimizes the sum of the squares of the errors made in the results of every single equation. Parameters of a functional dependence of the variable y are determined from the observable quantities x.

A *linear regression* is a least-square fit with a linear function of single variable. Linear regression attempts to model the relationship between two variables by fitting a linear equation to observed data. One variable is considered to be an explanatory variable, and the other is considered to be a dependent variable. A numerical measure of association between two variables is the correlation coefficient, which is a value between -1 and 1 indicating the strength of the association of the observed data for the two variables. A linear regression line has an equation of the form

$$y = a + bx$$

where x is the explanatory variable and y is the dependent variable. The slope of the line is b, and a is the intercept (the value of $\boldsymbol{y}$ when $\boldsymbol{x}$ = 0).

All the functions contained in the Least-Square Fit class are listed in Appendix E.

Example:

Fit a straight line to the x and y values in the Table below:

x	y
1	0.5
2	2.5
3	2.0
4	4.0
5	3.5
6	6.0
7	5.5

Solution:

```
lr = import_java_class("library.estimation.LinearRegression");
```

The statement above imports the Linear Regression class and assigns to the pointer **`lr`**.

```
x1 = 1.0:7.0;
```

The statement above creates a vector whose elements are from 1.0 to 7.0 with increments of 1.0.

```
y1 = [0.5, 2.5, 2.0, 4.0, 3.5, 6.0, 5.5];
```

The statement above creates a vector with the given elements.

```
p1 = lr.linearRegression(x1, y1);
```

The statement above computes a polynomial using linear regression and assigns the polynomial to the variable **`p1`**.

```
r1 = lr.getCorrelationCoefficient();
```

The statement above computes the correlation coefficient of the linear regression and assigns the coefficient to the variable **`r1`**.

5.5.3 Polynomial Regression

Polynomial regression are statistical methods for estimating an underlying polynomial that describes observations. In a polynomial fit, the fit function is a polynomial of degree *m*. The parameters are numbered starting from zero. The number of free parameters is *m+1*.

Approximating a function $Z(t)$with a polynomial

$$\hat{z}(t) = \sum_{i=0}^{m+1} a_i t^i$$

where hat (^) denotes the estimate. Polynomial regression models are usually fit using the method of least squares.

All the functions contained in the Polynomial Regression class are listed in Appendix E.

Example:

Fit a second-order polynomial to the data in the table below:

x	y
0	2.1
1	7.7
2	13.6
3	27.2
4	40.9
5	61.1

Solution:

```
pr = import_java_class("library.estimation.PolynomialRegression");
```

The statement above imports the Polynomial Regression class and assigns to the pointer **`pr`**.

```
x1 = 0.0:5.0;
```

The statement above creates a vector whose elements are from 0.0 to 5.0 with increments of 1.0.

```
y1 = [2.1, 7.7, 13.6, 27.2, 40.9, 61.1];
```

The statement above creates a vector with the given elements.

```
p1 = pr.polynomialLSFit(x1, y1, 2);
```

The statement above computes the coefficient of a polynomial using the least square method and assigns the corresponding polynomial to the variable **`p1`**.

5.6 Stochastic

The Stochastic library contains methods related to statistics and probability. The Stochastic library contains the following classes:

1. Histogram
2. Probability Distribution

The Stochastic library is documented in Appendix F.

5.6.1 Statistics

Given a random variable whose values are a set of data points, $x_1, x_2, \ldots, x_n$, the *k*th order moment of the set is defined as

$$M_k = \frac{1}{n}\sum_{i=1}^{n} x_i^2$$

The moment of the first order is the mean or the average defined as

$$\bar{x} = M_1 = \frac{1}{n}\sum_{i=1}^{n} x_i$$

The central moment of *k*th order is defined by

$$m_k = \frac{1}{n}\sum_{i=1}^{n} (x_i - \bar{x})^k$$

The variance of a set is defined by

$$var = \frac{1}{n-1}\sum_{i=1}^{n} (x_i - \bar{x})^2$$

The standard deviation is defined by

$$\sigma = \sqrt{var} = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n} (x_i - \bar{x})^2}$$

The *skewness* is defined by

$$skew = \frac{1}{(n-1)(n-2)}\sum_{i=1}^{n} (x_i - \bar{x})^3$$

The *kurtosis* is defined by

$$kurtosis = \frac{n+1}{(n-1)(n-2)(n-3)}\sum_{i=1}^{n} \left(\frac{x_i - \bar{x}}{s}\right)^4 - \frac{3(n-1)^2}{(n-2)(n-3)}$$

5.6.1.1 Histogram:

A mathematical **histogram** which is a function that counts the number of observations that fall into each of the disjoint categories (known as *bins*). To construct a histogram, the first step is to divide the entire range of values into a series of intervals—and then count how many values fall into each interval. The bins are usually specified as consecutive, non-overlapping intervals of a variable. The bins (intervals) must be adjacent, and are usually equal size. A histogram is defined by three main parameters: x_{min}, the minimum of all values accumulated into the histogram; w, the bin width; and n, the number of bins. The ith bin of a histogram is the interval $[x_{min} + (i-1)w, x_{min} + iw)$. The bin contents of a histogram is the number of times a value falls within each bin interval. The bin width is computed as

$$w = \frac{x_{max} - x_{min}}{n},$$

where $x_{max}$is the maximum accumulated values.

The Histogram class implements a mathematical histogram. All the functions contained in the Histogram class are listed in Appendix F.

Example:

Divide the data in the table below into 5 equal length intervals between 140 and 190 cm and create a histogram.

162	168	177	147
189	171	173	168
178	184	165	173
179	166	168	165

Compute the following attributes from the histogram:

count, bin width, minimum, maximum, average, standard deviation, skewness, and kurtosis.

Solution:

```
hist = import_java_class("library.stochastic.Histogram");
```

The statement above imports the Histogram class and assigns it to the pointer **`hist`**.

```
hist.setHistogram(140.0, 190.0, 5);
```

The above statement sets up the histogram with 5 bins and interval between 140.0 and 190.0.

```
data = [162, 168, 177, 147, 189, 171, 173, 168, 178, 184, 165, 173, 179,
166, 168, 165];
```

The above statement creates a vector with the given data and assigns it to the variable **`data`**.

```
hist.processData(data);
```

The above statement creates a mathematical using the **`data`**.

```
c = hist.count();
```

The above statement counts the number of data and assigns the result in the variable **`c`**.

```
w = hist.binWidth();
```

The above statement computes the bin width and assigns the result in the variable **`w`**.

```
min = hist.minimum();
```

The above statement computes the minimum value of the data and assigns the result in the variable **`min`**.

```
max = hist.maximum();
```

The above statement computes the maximum value of the data and assigns the result in the variable **`max`**.

```
ave = hist.average();
```

The above statement computes the average value of the data and assigns the result in the variable **`ave`**.

```
sd = hist.standardDeviation();
```

The above statement computes the standard deviation of the data and assigns the result in the variable **`sd`**.

```
skew = hist.skewness();
```

The above statement computes the skewness of the data and assigns the result in the variable **`skew`**.

```
k = hist.kurtosis();
```

The above statement computes the kurtosis of the data and assigns the result in the variable **`k`**.

5.6.2 Probability

A probability density function defines the probability of finding a continuous random variable within an infinitesimal interval. Formally, if X is a continuous random variable, then it has a probability density function $f(x)$, and therefore its probability of falling into a given interval, say $[a, b]$ is given by the integral

$$Prob[a \leq X \leq b] = \int_a^b f(x)dx$$

A continuous cumulative distribution function is defined as

$$F(x) = \mu(-\infty, x] = \int_{-\infty}^{x} f(t)dt$$

The moment of kth order for a probability density function $f(x)$ is defined by

$$M_k = \int x^k f(x)dx$$

The mean or average of the distribution is

$$\mu = M_1 = \int xf(x)dx$$

The central moment of the kth order defined by

$$m_k = \int (x - \mu)^k f(x)dx$$

The *skewness* is defined by

$$skew = \frac{\int x^3 f(x)dx}{\sigma^3}$$

The *kurtosis* is defined by

$$kurtosis = \frac{\int x^4 f(x)dx}{\sigma^4} - 3$$

5.6.2.1 Probability Distributions

The Stochastic library contains the following probability distributions:

1. Uniform Distribution
2. Triangular Distribution
3. Normal Distribution
4. Log Normal Distribution
5. Student Distribution
6. Gamma Distribution
7. Chi-Squared Distribution
8. Exponential Distribution
9. Laplace Distribution
10. Beta Distribution
11. Fisher Snedecor Distribution
12. Fisher Tippett Distribution
13. Weibull Distribution
14. Cauchy Distribution
15. Histogrammed Distribution
16. All Distribution

5.6.2.2 Uniform Distribution

The continuous **uniform distribution** or rectangular distribution is a family of symmetric probability distributions such that for each member of the family, all intervals of the same length on the distribution's support are equally probable. The support is defined by the two parameters, a and b, which are its minimum and maximum values.

Properties of the uniform distribution is given in the table below:

Property	Value
Notation	$\mathcal{U}(a, b)$
Parameters	$-\infty < a < b < +\infty$
Support	$x \in [a, b]$
Probability density function	$\begin{cases} \frac{1}{b-a} & \text{for } a \le x \le b \\ 0 & \text{otherwise} \end{cases}$
Distribution function	$\begin{cases} 0 & \text{for } x < a \\ \frac{x-a}{b-a} & \text{for } a \le x < b \\ 1 & \text{for } x > b \end{cases}$
Mean	$\frac{1}{2}(a+b)$
Median	$\frac{1}{2}(a+b)$
Mode	Any value in (a, b)
Variance	$\frac{1}{12}(b-a)^2$
Skewness	0
Kurtosis	$-\frac{6}{5}$

Example:

Generate a uniformly distributed random number between -1 and 1.

Solution:

```
dist = import_java_class("library.stochastic.ProbabilityDistribution");
```

The statement above imports the Probability Distribution class and assigns it to the pointer **`dist`**.

```
r = dist.uniform(-1.0,1.0);
```

The above statement computes a random number based on a uniform distribution ranged between of -1.0 to 1.0 and assigns the random value to the variable **`r`**.

Alternatively, the same result can be obtained using the following statements.

```
dist.setUniform(-1.0, 1.0);
r = dist.random();
```

5.6.2.3 Triangular Distribution

A **triangular distribution** is a continuous probability distribution with a probability density function shaped like a triangle. It is defined by three values: the minimum value a, the maximum value b,and the peak value c, where $a < b$ and $a \leq c \leq b$.

Properties of the triangular distribution is given in the table below:

Property	Value
Parameters	a: $a \in (-\infty, +\infty)$ b: $a < b$ c: $a \leq c \leq b$
Support	$[a, b]$
Probability density function (PDF)	$\begin{cases} 0 & \text{for } x < a \\ \frac{2(x-a)}{(b-a)(c-a)} & \text{for } a \leq x \leq c \\ \frac{2}{b-a} & \text{for } x = c \\ \frac{2(b-x)}{(b-a)(b-c)} & \text{for } c \leq x \leq b \\ 0 & \text{for } x > b \end{cases}$
Cumulative Distribution function (CDF)	$\begin{cases} 0 & \text{for } x \leq a \\ \frac{(x-a)2}{(b-a)(c-a)} & \text{for } a < x \leq c \\ 1 - \frac{(b-x)^2}{(b-a)(b-c)} & \text{for } c < x < b \\ \frac{2(b-x)}{(b-a)(b-c)} & \text{for } c \leq x \leq b \\ 1 & \text{for } x \geq b \end{cases}$
Mean	$\frac{a+b+c}{3}$
Median	$\begin{cases} a + \sqrt{\frac{(b-a)(c-a)}{2}} & \text{for } c \geq \frac{a+b}{2} \\ b - \sqrt{\frac{(b-a)(c-a)}{2}} & \text{for } c \geq \frac{a+b}{2} \end{cases}$
Mode	c
Variance	$\frac{a^2 + b^2 + c^2 - ab - ac - bc}{18}$

Skewness	$\frac{\sqrt{2}(a+b-2c)(2a-b-c)(a-2b+c)}{5(a^2+b^2+c^2-ab-ac-bc)^{\frac{3}{2}}}$
Kurtosis	$-3/5$

Example:

Generate a random number using a triangular distribution with minimum value = 1, maximum value = 8 and peak value = 3.

Solution:

```
dist = import_java_class("library.stochastic.ProbabilityDistribution");
```

The statement above imports the Probability Distribution class and assigns it to the pointer **`dist`**.

```
r = dist.triangular(1.0, 8.0, 3.0);
```

The above statement computes a random number based on a triangular distribution with minimum value = 1, maximum value = 8 and peak value = 3 and assigns the random value to the variable **`r`**.

Alternatively, the same result can be obtained using the following statements.

```
dist.setTriangular(1.0, 8.0, 3.0);
r = dist.random();
```

5.6.2.4 Normal Distribution

The **normal distribution** is the most important probability distribution. Normal distributions are symmetric and have bell-shaped density curves with a single peak. Most other distributions tend towards the normal distribution when some of their parameters become large. In normal distribution, two quantities must be specified: the mean μ, where the peak of the density occurs, and the standard deviation σ, which indicates the spread of the bell curve.

All normal density curves satisfy the following property which is often referred to as the Empirical Rule.

68% of the observations fall within 1 standard deviation of the mean, that is, between $\mu - \sigma$ and $\mu + \sigma$.

95% of the observations fall within 2 standard deviations of the mean, that is, between $\mu - 2\sigma$ and $\mu + 2\sigma$.

99.7% of the observations fall within 3 standard deviations of the mean, that is, between $\mu - 3\sigma$ and $\mu + 3\sigma$.

Thus, for a normal distribution, almost all values lie within 3 standard deviations of the mean.

Properties of the normal distribution is given in the table below:

Property	Value
Notation	$\mathcal{N}(\mu, \sigma)$
Parameters	$\mu \in \mathbb{R}$ $0 < \sigma^2 < +\infty$
Support	$x \in \mathbb{R}$
Probability density function (PDF)	$\frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{(x-\mu)^2}{2\sigma^2}}$
Cumulative Distribution function (CDF)	$\frac{1}{2}\left[1 + \operatorname{erf}\left(\frac{x-\mu}{\sigma\sqrt{2}}\right)\right]$
Mean	μ
Median	μ
Mode	μ
Variance	σ^2
Skewness	0
Kurtosis	0

Example:

Generate a random number using a normal distribution with mean value = 0, standard deviation = 0.25.

Solution:

```
dist = import_java_class("library.stochastic.ProbabilityDistribution");
```

The statement above imports the Probability Distribution class and assigns it to the pointer **dist**.

```
r = dist.normal(0.0, 0.25);
```

The above statement computes a random number based on a triangular distribution with mean value = 0, standard deviation = 0.25.

Alternatively, the same result can be obtained using the following statements.

```
dist.setNormal(0.0, 0.25);
r = dist.random();
```

5.6.2.5 Log Normal Distribution

A **log-normal (or lognormal) distribution** is a continuous probability distribution of a random variable whose logarithm is normally distributed. Thus, if the random variable X is log-normally distributed, then $Y = \ln(X)$ has a normal distribution. Likewise, if Y has a normal distribution, then $X = \exp(Y)$ has a log-normal distribution. A random variable which is log-normally distributed takes only positive real values. The distribution is occasionally referred to as the **Galton distribution** or **Galton's distribution**, after Francis Galton.

A log-normal process is the statistical realization of the multiplicative product of many independent random variables, each of which is positive. This is justified by considering the central limit theorem in the log domain. The log-normal distribution is the maximum entropy probability distribution for a random variate X for which the mean and variance of $\ln(X)$ are specified.

Properties of the normal distribution is given in the table below:

Property	Value
Notation	$\ln \mathcal{N}(\mu, \sigma^2)$
Parameters	$\mu \in \mathbb{R}$ -- location $0 < \sigma^2 < +\infty$ -- scale
Support	$x \in (0, +\infty)$
Probability density function (PDF)	$\frac{1}{x\sigma\sqrt{2\pi}} e^{-\frac{(\ln x - \mu)^2}{2\sigma^2}}$
Cumulative Distribution function (CDF)	$\frac{1}{2}\left[1 + \mathrm{erf}\left(\frac{\ln x - \mu}{\sqrt{2}\sigma}\right)\right]$
Mean	$e^{\mu + \sigma^2/2}$
Median	e^{μ}
Mode	$e^{\mu - \sigma^2}$
Variance	$(e^{\sigma^2} - 1)e^{2\mu + \sigma^2}$
Skewness	$(e^{\sigma^2} + 2)\sqrt{e^{\sigma^2} - 1}$
Kurtosis	$e^{4\sigma^2} + 2e^{3\sigma^2} + 3e^{2\sigma^2} - 6$

Example:

Example for the normal distribution can be used for the log-normal distribution.

5.6.2.6 Student's T Distribution

Student's *t*-distribution (or simply the ***t*-distribution**) is any member of a family of continuous probability distributions that arises when estimating the mean of a normally distributed population in situations where the sample size is small and population standard deviation is unknown developed by William Sealy Gosset under the pseudonym *Student*. Whereas a normal distribution describes a full population, *t*-distributions describe samples drawn from a full population; accordingly, the *t*-distribution for each sample size is different, and the larger the sample, the more the distribution resembles a normal distribution.

The *t*-distribution plays a role in a number of widely used statistical analyses, including the Student's *t*-test for assessing the statistical significance of the difference between two sample means, the construction of confidence intervals for the difference between two population means, and in linear regression analysis. The Student's *t*-distribution also arises in the Bayesian analysis of data from a normal family.

The *t*-distribution is symmetric and bell-shaped, like the normal distribution, but has heavier tails, meaning that it is more prone to producing values that fall far from its mean. This makes it useful for understanding the statistical behavior of certain types of ratios of random quantities, in which variation in the denominator is amplified and may produce outlying values when the denominator of the ratio falls close to zero. The Student's *t*-distribution is a special case of the generalized hyperbolic distribution.

Properties of the Student's t-distribution distribution is given in the table below:

Property	Value
Parameters	n (a positive integer)
Support	$(-\infty, +\infty)$
Probability density function (PDF)	$\frac{1}{\sqrt{n}B\left(\frac{n}{2},\frac{1}{2}\right)}\left(1+\frac{t^2}{n}\right)^{-\frac{n+1}{2}}$
Cumulative Distribution function (CDF)	$\begin{cases} \frac{1+B\left(\frac{n}{n+x^2};\frac{n}{2},\frac{1}{2}\right)}{2} & \text{for } x \geq 0 \\ \frac{1-B\left(\frac{n}{n+x^2};\frac{n}{2},\frac{1}{2}\right)}{2} & \text{for } < 0 \end{cases}$
Mean	0
Variance	$\frac{n}{n-2}$ for $n > 0$ Undefined otherwise
Skewness	0
Kurtosis	$\frac{6}{n-4}$ for $n > 4$ Undefined otherwise

Example:

Generate a random number using a student’s t-distribution with degrees-of-freedom = 8.

Solution:

```
dist = import_java_class("library.stochastic.ProbabilityDistribution");
```

The statement above imports the Probability Distribution class and assigns it to the pointer **`dist`**.

```
r = dist.student(8.0);
```

The above statement computes a random number based on a triangular distribution with degrees-of-freedom = 8.

Alternatively, the same result can be obtained using the following statements.

```
dist.setStudent(8);
r = dist.random();
```

5.6.2.7 Gamma Distribution

The **gamma distribution** is a two-parameter family of continuous probability distributions. The common exponential distribution and chi-squared distribution are special cases of the gamma distribution. There are three different parameterizations in common use:

1. With a shape parameter k and a scale parameter θ.
2. With a shape parameter $\alpha = k$ and an inverse scale parameter $\beta = 1/\theta$, called a rate parameter.
3. With a shape parameter k and a mean parameter $\mu = k/\beta$.

In each of these three forms, both parameters are positive real numbers.

Properties of the normal distribution is given in the table below:

Property	Value
Parameters	$k > 0$ shape $\alpha > 0$ shape $\theta > 0$ scale $\beta > 0$ scale
Support	$x \in (0, +\infty)$
Probability density function (PDF)	$\frac{x^{\alpha-1}}{\beta^{\alpha}\Gamma(\alpha)} e^{-\frac{x}{\beta}}$
Cumulative Distribution function (CDF)	$\left(\frac{x}{\beta}, \alpha\right)$
Mean	$\alpha\beta$
Variance	$\alpha\beta^2$
Skewness	$\frac{2}{\sqrt{\alpha}}$
Kurtosis	$\frac{6}{\alpha}$

Example:

Generate a random number using a gamma distribution with shape value = 9.56, scale value = 38.94.

Solution:

```
dist = import_java_class("library.stochastic.ProbabilityDistribution");
```

The statement above imports the Probability Distribution class and assigns it to the pointer **dist**.

```
r = dist.gamma(9.56, 38.94);
```

The above statement computes a random number based on a gamma distribution with shape value = 9.56, scale value = 38.94.

Alternatively, the same result can be obtained using the following statements.

```
dist.setGamma(9.56, 38.94);
r = dist.random();
```

5.6.2.8 Chi-Squared Distribution

The **chi-squared distribution** (also **chi-square** or ***χ***²**-distribution**) with *k* degrees of freedom is the distribution of a sum of the squares of *k* independent standard normal random variables. It is a special case of the gamma distribution.

Properties of the Chi-Squared distribution is given in the table below:

Property	Value
Notation	$\chi^2(k)$ or χ_k^2
Parameters	$k \in \mathbb{N} > 0$ (known as "degrees-of-freedom")
Support	$x \in [0, +\infty)$
Probability density function (PDF)	$\frac{1}{2^{\frac{k}{2}}\Gamma\left(\frac{k}{2}\right)} x^{\frac{k}{2}-1} e^{-\frac{x}{2}}$
Cumulative Distribution function (CDF)	$\frac{1}{\Gamma\left(\frac{k}{2}\right)} \gamma\left(\frac{k}{2}, \frac{x}{2}\right)$
Mean	k
Median	$\approx k\left(1 - \frac{2}{9k}\right)^3$
Mode	$\max\{k-2, 0\}$
Variance	$2k$
Skewness	$\sqrt{8/k}$
Kurtosis	$\frac{12}{k}$

Example:

Example for the student's t-distribution can be used for the chi-squared distribution.

5.6.2.9 Exponential Distribution

The **exponential distribution** (a.k.a. **negative exponential distribution**) is the probability distribution that describes the time between events in a Poisson process, i.e. a process in which events occur continuously and independently at a constant average rate. It is a specific case of the gamma distribution. It is the continuous analogue of the geometric distribution, and it has the key property of being memory less.

Properties of the exponential distribution is given in the table below:

Property	Value
Notation	
Parameters	$\lambda > 0$ rate, or inverse scale
Support	$x \in [0, +\infty)$
Probability density function (PDF)	$\lambda e^{-\lambda x}$
Cumulative Distribution function (CDF)	$1 - e^{-\lambda x}$
Mean	$\lambda^{-1} (= \beta)$
Median	$\lambda^{-1} \ln(2)$
Mode	0
Variance	6
Skewness	$\sqrt{8/k}$
Kurtosis	$\frac{12}{k}$

Example:

Generate a random number using an exponential distribution with rate = 0.5.

Solution:

```
dist = import_java_class("library.stochastic.ProbabilityDistribution");
```

The statement above imports the Probability Distribution class and assigns it to the pointer **`dist`**.

```
r = dist.exponential(0.5);
```

The above statement computes a random number based on an exponential distribution with rate = 0.5.

Alternatively, the same result can be obtained using the following statements.

```
dist.setExponential(0.5);
r = dist.random();
```

5.6.2.10 Laplace Distribution

The **Laplace distribution** is a continuous probability distribution named after Pierre-Simon Laplace. It is also sometimes called the *double exponential distribution*, because it can be thought of as two exponential distributions (with an additional location parameter) spliced together back-to-back, although the term 'double exponential distribution' is also sometimes used to refer to the Gumbel distribution. The difference between two independent identically distributed exponential random variables is governed by a Laplace distribution. Increments of Laplace motion or a variance gamma process evaluated over the time scale also have a Laplace distribution.

Properties of the Laplace distribution is given in the table below:

Property	Value
Notation	
Parameters	μ location (real) $b > 0$ scale (real)
Support	$x \in (-\infty, +\infty)$
Probability density function (PDF)	$\frac{1}{2b}\exp\left(-\frac{\lvert x-\mu\rvert}{b}\right)$
Cumulative Distribution function (CDF)	$\begin{cases} \frac{1}{2}\exp\left(-\frac{x-\mu}{b}\right) & \text{if } x < \mu \\ 1-\frac{1}{2}\exp\left(-\frac{x-\mu}{b}\right) & \text{if } x \geq \mu \end{cases}$
Mean	μ
Median	μ
Mode	μ
Variance	$2b^2$
Skewness	0
Kurtosis	3

Example:

Generate a random number using a Laplace distribution with location = 0.0 and scale = 1.0.

Solution:

```
dist = import_java_class("library.stochastic.ProbabilityDistribution");
```

The statement above imports the Probability Distribution class and assigns it to the pointer **`dist`**.

```
r = dist.laplace(0.0, 1.0);
```

The above statement computes a random number based on a Laplace distribution with location = 0.0 and scale = 1.0.

Alternatively, the same result can be obtained using the following statements.

```
dist.setLaplace(0.0, 1.0);
r = dist.random();
```

5.6.2.11 Beta Distribution

The **Beta distribution** is a family of continuous probability distributions defined on the interval [0, 1] parameterized by two positive shape parameters, denoted by α and β, that appear as exponents of the random variable and control the shape of the distribution.

Properties of the Beta distribution is given in the table below:

Property	Value
Notation	$Beta(\alpha, \beta)$
Parameters	$\alpha > 0$ shape (real) $\beta > 0$ shape (real)
Support	$x \in (0,1)$
Probability density function (PDF)	$\frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha,\beta)}$
Cumulative Distribution function (CDF)	$I_x(\alpha,\beta)$
Mean	$E[X] = \frac{\alpha}{\alpha+\beta}$ $E[lnX] = \psi(\alpha) - \psi(\alpha+\beta)$
Median	$I_{\frac{1}{2}}^{[-1]}(\alpha,\beta)$ (in general) $\approx \frac{\alpha - \frac{1}{3}}{\alpha+\beta-\frac{2}{3}}$ for $\alpha, \beta > 1$
Mode	$\frac{\alpha-1}{\alpha+\beta-2}$ for $\alpha, \beta > 1$
Variance	$var[x] = \frac{\alpha\beta}{(\alpha+\beta)^2(\alpha+\beta+1)}$ $var[lnX] = \psi_1(\alpha) - \psi_1(\alpha+\beta)$
Skewness	$\frac{2(\beta-\alpha)\sqrt{\alpha+\beta+2}}{(\alpha+\beta+2)\sqrt{\alpha\beta}}$
Kurtosis	$\frac{6[(\alpha-\beta)^2(\alpha+\beta+1) - \alpha\beta(\alpha+\beta+2)]}{\alpha\beta(\alpha+\beta+2)(\alpha+\beta+3)}$

Example:

Generate a random number using a beta distribution with $\alpha = 2.0$ and $\beta = 3.0$.

Solution:

```
dist = import_java_class("library.stochastic.ProbabilityDistribution");
```

The statement above imports the Probability Distribution class and assigns it to the pointer **dist**.

```
r = dist.beta(2.0, 3.0);
```

The above statement computes a random number based on a beta distribution with $\alpha = 2.0$ and $\beta = 3.0$.

Alternatively, the same result can be obtained using the following statements.

```
dist.setBeta(2.0, 3.0);
r = dist.random();
```

5.6.2.12 Fisher-Snedecor Distribution

The **Fisher–Snedecor distribution** (after Ronald Fisher and George W. Snedecor) , also known as **Snedecor's *F* distribution** or the ***F*-distribution** is a continuous probability distribution.

The *F*-distribution arises frequently as the null distribution of a test statistic, most notably in the analysis of variance.

Properties of the Fisher–Snedecor distribution is given in the table below:

Property	Value
Parameters	$n, d > 0$ deg of freedom
Support	$[0, +\infty)$
Probability density function (PDF)	$\dfrac{\sqrt{\dfrac{(nx)^n d^d}{(nx+d)^n + d}}}{xB\left(\frac{n}{2}, \frac{d}{2}\right)}$
Cumulative Distribution function (CDF)	$F(x) = I_{\frac{nx}{nx+d}}\left(\frac{n}{2}, \frac{d}{2}\right)$
Mean	$\frac{d}{d-2}$ for $d > 2$ Undefined otherwise
Mode	$\frac{n-2}{n} \frac{d}{d+2}$ for $d > 2$
Variance	$\frac{2d^2(n+d-2)}{n(d-n)^2(d-4)}$ for $d > 4$ Undefined otherwise
Skewness	$\frac{(2n+d-2)\sqrt{8(d-4)}}{(d-6)\sqrt{n(n+d-2)}}$ for $d > 6$ Undefined otherwise

Kurtosis	$3 + 12\frac{n(5d-22)(n+d-2)+(d-4)(d-2)^2}{n(d-6)(d-8)(n+d-2)}$ for $d > 8$ Undefined otherwise

Where, B is the Beta function defined in terms of Gamma function (Γ) as

$$B(n, d) = \frac{\Gamma(n)\Gamma(d)}{\Gamma(n + d)}$$

Example:

Generate a random number using a Fisher–Snedecor distribution with degrees-of-freedom, n = 10 and degrees-of-freedom, d = 15.

Solution:

```
dist = import_java_class("library.stochastic.ProbabilityDistribution");
```

The statement above imports the Probability Distribution class and assigns it to the pointer **`dist`**.

```
r = dist.beta(10, 15);
```

The above statement computes a random number based on a Fisher–Snedecor distribution with degrees-of-freedom, n = 10 and degrees-of-freedom, d = 15.

Alternatively, the same result can be obtained using the following statements.

```
dist.setFisherSnedecor(10, 15);
r = dist.random();
```

5.6.2.13 Fisher Tippett Distribution

the **Fisher–Tippett distribution**, named after Ronald Fisher and L. H. C. Tippett, also known as **generalized extreme value** (**GEV**) **distribution** is a family of continuous probability distributions developed within extreme value theory to combine the Gumbel, Fréchet and Weibull families also known as type I, II and III extreme value distributions. By the extreme value theorem the GEV distribution is the only possible limit distribution of properly normalized maxima of a sequence of independent and identically distributed random variables.

Properties of the Fisher–Tippett distribution is given in the table below:

Property	Value
Notation	$GEV(\mu, \sigma, \xi)$
Parameters	$\mu \in \mathbb{R}$ -- location, $\sigma > 0$ -- scale, $\xi \in \mathbb{R}$ -- shape.
Support	$x \in [\mu - \sigma/\xi \ , +\infty)$ when $\xi > 0$ $x \in (-\infty, +\infty)$ when $\xi = 0$ $x \in (-\infty, \mu - \ \sigma/\xi\ )$ when $\xi < 0$
Probability density function (PDF)	$\frac{1}{\sigma} t(x)^{\xi+1} e^{-t(x)}$, where $t(x) = \begin{cases} \left(1 + \left(\frac{x-\mu}{\sigma}\right)\xi\right)^{-\frac{1}{\xi}} & \text{if } \xi \neq 0 \\ e^{-(x-\mu)/\sigma} & \text{if } \xi = 0 \end{cases}$
Cumulative Distribution function (CDF)	$e^{-t(x)}$, for $x \in range$
Mean	$\begin{cases} \mu + \sigma \frac{\Gamma(1-\xi) - 1}{\xi} & \text{if } \xi \neq 0, \xi = 1, \\ \mu + \sigma\gamma & \text{if } \xi = 0, \\ \infty & \text{if } \xi \geq 1, \end{cases}$ Where γ is Euler's constant.
Median	$\begin{cases} \mu + \sigma \frac{(ln2)^{-\xi} - 1}{\xi} & \text{if } \xi \neq 0, \\ \mu - \sigma ln\ ln2 & \text{if } \xi \neq 0. \end{cases}$
Mode	$\begin{cases} \mu + \sigma \frac{(1+\xi)^{-\xi} - 1}{\xi} & \text{if } \xi \neq 0, \\ \mu & \text{if } \xi \neq 0. \end{cases}$
Variance	$\begin{cases} \sigma^2 (g_2 - g_1^2)/\xi^2 & \text{if } \xi \neq 0, \xi = 1, \\ \sigma^2 \frac{\pi^2}{6} & \text{if } \xi = 0, \\ \infty & \text{if } \xi \geq \frac{1}{2}. \end{cases}$ where $g_k = \Gamma(1 - k\xi)$

Skewness	$\begin{cases} \dfrac{g_3 - 3g_1g_2 + 2g_1}{(g_2 - g_1^2)^{\frac{3}{2}}} & \text{if } \xi > 0, \\ -\dfrac{g_3 - 3g_1g_2 + 2g_1^3}{(g_2 - g_1^2)^{\frac{3}{2}}} & \text{if } \xi < 0, \\ \dfrac{12\sqrt{6}\zeta(3)}{\pi^3} & \text{if } \xi = 0. \end{cases}$ where $\zeta(x)$ is Riemann zeta function
Kurtosis	$\begin{cases} \dfrac{g_4 - 4g_1g_3 + 6g_2g_1^2 - 3g_1^4}{(g_2 - g_1^2)^2} & \text{if } \xi \neq 0, \xi = \frac{1}{4}, \\ \dfrac{12}{56} & \text{if } \xi = 0, \\ \infty & \text{if } \xi \geq \frac{1}{4}. \end{cases}$

Example:

Generate a random number using a Fisher–Tippett distribution with mean = 0.0 and standard deviation = 1.0.

Solution:

```
dist = import_java_class("library.stochastic.ProbabilityDistribution");
```

The statement above imports the Probability Distribution class and assigns it to the pointer **`dist`**.

```
r = dist.fisherTippett(0.0, 1.0);
```

The above statement computes a random number based on a Fisher–Tippett distribution with mean = 0.0 and standard deviation = 1.0.

Alternatively, the same result can be obtained using the following statements.

```
dist.setFisherTippett(0.0, 1.0);
r = dist.random();
```

5.6.2.14 Weibull Distribution

the **Weibull distribution** is a continuous probability distribution. It is named after Swedish mathematician Waloddi Weibull, who described it in detail in 1951, although it was first identified by Fréchet (1927) and first applied by Rosin & Rammler (1933) to describe a particle size distribution.

Properties of the Weibull distribution is given in the table below:

Property	Value
Notation	
Parameters	$\lambda \in (-\infty, +\infty)$ -- scale $k \in (-\infty, +\infty)$ -- shape
Support	$x \in [0, +\infty)$
Probability density function (PDF)	$\begin{cases} \frac{k}{\lambda}\left(\frac{x}{\lambda}\right)^{k-1} e^{-(x/\lambda)^k} & \text{for } x \geq 0 \\ 0 & \text{for } x < 0 \end{cases}$
Cumulative Distribution function (CDF)	$\begin{cases} 1 - e^{-(x/\lambda)^k} & \text{for } x \geq 0 \\ 0 & \text{for } x < 0 \end{cases}$
Mean	$\lambda\Gamma(1 + 1/k)$
Median	$\lambda(\ln(2))^{1/k}$
Mode	$\begin{cases} \lambda\left(\frac{k-1}{k}\right)^{\frac{1}{k}} & \text{for } k > 1 \\ 0 & \text{for } k = 1 \end{cases}$
Variance	$\lambda^2\left[\Gamma\left(1 + \frac{2}{k}\right) - \left(\Gamma\left(1 + \frac{1}{k}\right)\right)^2\right]$
Skewness	$\frac{\Gamma(1 + 3/k)\lambda^3 - 3\mu\sigma^2 - \mu^3}{\sigma^3}$
Kurtosis	

Example:

Generate a random number using a Weibull distribution with shape = 1.0 and scale = 2.0.

Solution:

```
dist = import_java_class("library.stochastic.ProbabilityDistribution");
```

The statement above imports the Probability Distribution class and assigns it to the pointer **`dist`**.

```
r = dist.weibull(1.0, 2.0);
```

The above statement computes a random number based on a Weibull distribution with shape = 1.0 and scale = 2.0.

Alternatively, the same result can be obtained using the following statements.

```
dist. setWeibull(1.0, 2.0);
r = dist.random();
```

5.6.2.15 Cauchy Distribution

The **Cauchy distribution**, named after Augustin Cauchy, is a continuous probability distribution. It is also known, especially among physicists, as the **Lorentz distribution** (after Hendrik Lorentz), **Cauchy–Lorentz distribution**, **Lorentz(ian) function**, or **Breit–Wigner distribution**. The simplest Cauchy distribution is called the **standard Cauchy distribution**. It is the distribution of a random variable that is the ratio of two independent standard normal variables and has the probability density function.

Properties of the Cauchy distribution is given in the table below:

Property	Value
Parameters	x_0 location (real) $\Upsilon > 0$ scale (real)
Support	$(-\infty, +\infty)$
Probability density function (PDF)	$\frac{1}{\pi\gamma\left[1+\left(\frac{x-x_0}{\gamma}\right)^2\right]}$
Cumulative Distribution function (CDF)	$\frac{1}{\pi}\arctan\left(\frac{x-x_0}{\gamma}\right)+\frac{1}{2}$
Mean	Undefined
Median	x_0
Mode	x_0
Variance	Undefined
Skewness	Undefined
Kurtosis	Undefined

Example:

Generate a random number using a Cauchy distribution with location = 0.0 and scale = 1.0.

Solution:

```
dist = import_java_class("library.stochastic.ProbabilityDistribution");
```

The statement above imports the Probability Distribution class and assigns it to the pointer **`dist`**.

```
r = dist.cauchy(0.0, 1.0);
```

The above statement computes a random number based on a Cauchy distribution with middle = 0.0 and width = 1.0.

Alternatively, the same result can be obtained using the following statements.

```
dist. setCauchy(0.0, 1.0);
r = dist.random();
```

5.6.2.16 *Histogrammed Distribution*

5.7 Frequency Domain

The Frequency Domain library contains methods that transform time domain data into frequency domain data and vice versa. The Frequency Domain library contains one class, FFT. The Frequency Domain library is documented in Appendix G.

5.7.1 FFT

The **Fourier transform** decomposes a *signal* (a time domain function) into the frequencies that make up the signal. The Fourier transform of a function of time itself is a complex-valued function of frequency, whose absolute value represents the amount of that frequency present in the original function, and whose complex argument is the phase offset of the basic sinusoid in that frequency. The Fourier transform is called the *frequency domain representation* of the original signal. The equation for Fourier transform is

$$G(f) = \int_{-\infty}^{\infty} g(t) e^{-i2\pi f t}\, dt$$

The equation for inverse Fourier transform is

$$g(t) = \int_{-\infty}^{\infty} G(f) e^{i2\pi f t}\, df$$

A **fast Fourier transform** (**FFT**) algorithm computes the discrete Fourier transform (DFT) of a sequence, or its inverse. Fourier analysis converts a signal from its original domain (often time or space) to a representation in the frequency domain and vice versa. An FFT reduces the number of operations of computing the DFT from $O(n^2)$ to $O(n \log n)$, where n is the data size. An FFT is much faster than DFT at evaluating the DFT definition directly, but produces exactly the same result.

Let $x_0, \cdots, x_{N-1}$be complex numbers. The DFT is defined by the formula

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi k\frac{n}{N}} \qquad\qquad k = 0, \dots, N-1$$

All the functions contained in the FFT class are listed in Appendix G.

Example:

Given the signal $x = \cos\left(\frac{2\pi n}{10}\right)$ where $n = 0, 1, \dots, 28, 29$.

Perform FFT for 30, 64, 128, and 256 samples.

Solution:

```
fd = import_java_class("library.frequency_domain.FFT");
```

The statement above imports the FFT class and assigns it to the pointer **`fd`**.

```
X30 = fd.fft(x).magnitude;
```

The statement above performs FFT for 30 (number of data points in x) samples and assigns the output to the variable .

```
X64 = fd.fft(x, 64).magnitude;
```

The statement above performs FFT for 30 samples and assigns the output to the variable **`X64`**.

```
X128 = fd.fft(x, 128).magnitude;
```

The statement above performs FFT for 30 samples and assigns the output to the variable **`X128`**.

```
X256 = fd.fft(x, 256).magnitude;
```

The statement above performs FFT for 30 samples and assigns the output to the variable **`X256`**.

Appendix A -- Library: General Math

A.1 Constants:

Identifier	Description	Type
`E`	The `real` value that is closer than any other to *e*, the base of the natural logarithms.	**`real`**
`PI`	The `real` value that is closer than any other to *pi*, the ratio of the circumference of a circle to its diameter.	**`real`**

A.2 Functions

Call Signature	Description	Return Type
`abs(real a)`	Returns the absolute value of a `real` value.	**`real`**
`abs(integer a)`	Returns the absolute value of a `integer` value.	**`integer`**
`abs(realVector a)`	Returns a vector whose elements are the absolute values of the elements of the input vector a.	**`realVector`**
`abs(realMatrix a)`	Returns a matrix whose elements are the absolute values of the elements of the input matrix a.	**`realMatrix`**
`acos(real a)`	Returns the arc cosine of a value; the returned angle is in the range 0.0 through *pi*.	**`real`**
`acos(integer a)`	Returns the arc cosine of a value; the returned angle is in the range 0.0 through *pi*.	**`real`**
`acos(realVector a)`	Returns a vector whose elements are the arc cosines of the elements of the input vector a; the returned angles are in the range 0.0 through *pi*.	**`realVector`**
`acos(realMatrix a)`	Returns a matrix whose elements are the arc cosines of the elements of the input matrix a; the returned angles are in the range 0.0 through *pi*.	**`realMatrix`**
`asin(real a)`	Returns the arc sine of a value; the returned angle is in the range -*pi*/2 through *pi*/2.	**`real`**
`asin(integer a)`	Returns the arc sine of a value; the returned angle is in the range -*pi*/2 through *pi*/2.	**`real`**
`asin(realVector a)`	Returns a vector whose elements are the arc sines of the elements of the input vector a; the returned angle is in the range -*pi*/2 through *pi*/2.	**`realVector`**
`asin(realMatrix a)`	Returns a matrix whose elements are the arc sines of the elements of the input matrix a; the returned angle is in the range -*pi*/2 through *pi*/2.	**`realMatrix`**
`atan(real a)`	Returns the arc tangent of a value; the returned angle is in the range -*pi*/2 through *pi*/2.	**`real`**
`atan(integer a)`	Returns the arc tangent of a value; the returned angle is in the range -*pi*/2 through *pi*/2.	**`real`**
`atan(realVector a)`	Returns a vector whose elements are the arc tangents of	**`realVector`**

	the elements of the input vector a; the returned angle is in the range -*pi*/2 through *pi*/2.	
`atan(realMatrix a)`	Returns a matrix whose elements are the arc tangents of the elements of the input matrix a; the returned angle is in the range -*pi*/2 through *pi*/2.	**`realMatrix`**
`atan2(integer y, integer x)`	Returns the angle *theta* from the conversion of rectangular coordinates (`x`, `y`) to polar coordinates (r, *theta*).	**`real`**
`atan2(integer y, real x)`	Returns the angle *theta* from the conversion of rectangular coordinates (`x`, `y`) to polar coordinates (r, *theta*).	**`real`**
`atan2(real y, integer x)`	Returns the angle *theta* from the conversion of rectangular coordinates (`x`, `y`) to polar coordinates (r, *theta*).	**`real`**
`atan2(real y, real x)`	Returns the angle *theta* from the conversion of rectangular coordinates (`x`, `y`) to polar coordinates (r, *theta*).	**`real`**
`cbrt(integer a)`	Returns the cube root of a `integer` value.	**`real`**
`cbrt(real a)`	Returns the cube root of a `real` value.	**`real`**
`ceil(real a)`	Returns the smallest (closest to negative infinity) `real` value that is greater than or equal to the argument and is equal to a mathematical integer.	**`real`**
`copySign(real magnitude, real sign)`	Returns the first floating-point argument with the sign of the second floating-point argument.	**`real`**
`cos(real a)`	Returns the trigonometric cosine of an angle.	**`real`**
`cos(integer a)`	Returns the trigonometric cosine of an angle.	**`real`**
`cos(realVector a)`	Returns a vector whose elements are the trigonometric cosines of the elements of the input vector a.	**`realVector`**
`cos(realMatrix a)`	Returns a matrix whose elements are the trigonometric cosines of the elements of the input matrix a.	**`realMatrix`**
`cosh(integer x)`	Returns the hyperbolic cosine of a `integer` value.	**`real`**
`cosh(real x)`	Returns the hyperbolic cosine of a `real` value.	**`real`**
`cosh(realVector x)`	Returns a vector whose elements are the hyperbolic cosines of the elements of the input vector x.	**`realVector`**
`cosh(realMatrix x)`	Returns a matrix whose elements are the hyperbolic cosines of the elements of the input matrix x.	**`realMatrix`**
`exp(integer a)`	Returns Euler's number *e* raised to the power of a `real` value.	**`real`**
`exp(real a)`	Returns Euler's number *e* raised to the power of a `real` value.	**`real`**
`expm1(real x)`	Returns e^x -1.	**`real`**
`floor(real a)`	Returns the largest (closest to positive infinity) `real` value that is less than or equal to the argument and is equal to a mathematical integer.	**`real`**
`getExponent(real d)`	Returns the unbiased exponent used in the representation of a `real`.	**`int`**
`hypot(real x, real y)`	Returns sqrt(x^2 +y^2) without intermediate overflow or underflow.	**`real`**
`IEEEremainder(integer f1,`	Computes the remainder operation on two arguments as	**`real`**

integer f2)	prescribed by the IEEE 754 standard.	
IEEEremainder(integer f1, real f2)	Computes the remainder operation on two arguments as prescribed by the IEEE 754 standard.	**real**
IEEEremainder(real f1, integer f2)	Computes the remainder operation on two arguments as prescribed by the IEEE 754 standard.	**real**
IEEEremainder(real f1, real f2)	Computes the remainder operation on two arguments as prescribed by the IEEE 754 standard.	**real**
log(integer a)	Returns the natural logarithm (base *e*) of a `integer` value.	**real**
log(real a)	Returns the natural logarithm (base *e*) of a `real` value.	**real**
log(realVector a)	Returns the natural logarithm (base *e*) of a `realVector` value.	**realVector**
log(realMatrix a)	Returns the natural logarithm (base *e*) of a `realMatrix` value.	**realMatrix**
logb(integer a, integer b)	Returns the base `b` logarithm of a `integer` value.	**real**
logb(integer a, real b)	Returns the base `b` logarithm of a `integer` value.	**real**
logb(real a, integer b)	Returns the base `b` logarithm of a `real` value.	**real**
logb(real a, real b)	Returns the base `b` logarithm of a `real` value.	**real**
logb(realVector a, integer b)	Returns the base `b` logarithm of a `realVector` value.	**realVector**
logb(realVector a, real b)	Returns the base `b` logarithm of a `realVector` value.	**realVector**
logb(realMatrix a, integer b)	Returns the base `b` logarithm of a `realMatrix` value.	**realMatrix**
logb(realMatrix a, real b)	Returns the base `b` logarithm of a `realMatrix` value.	**realMatrix**
Log2(integer a)	Returns the base 2 logarithm of a `integer` value.	**real**
log2(real a)	Returns the base 2 logarithm of a `real` value.	**real**
log2(realVector a)	Returns the base 2 logarithm of a `realVector` value.	**realVector**
log2(realMatrix a)	Returns the base 2 logarithm of a `realMatrix` value.	**realMatrix**
log10(integer a)	Returns the base 10 logarithm of a `integer` value.	**real**
log10(real a)	Returns the base 10 logarithm of a `real` value.	**real**
log10(realVector a)	Returns the base 10 logarithm of a `realVector` value.	**realVector**
log10(realMatrix a)	Returns the base 10 logarithm of a `realMatrix` value.	**realMatrix**
log1p(real x)	Returns the natural logarithm of the sum of the argument and 1.	**real**
max(integer a, integer b)	Returns the greater of two `integer` values.	**integer**
max(integer a, real b)	Returns the greater of two values.	**real**
max(real a, integer b)	Returns the greater of two values.	**real**
max(real a, real b)	Returns the greater of two `real` values.	**real**

min(integer a, integer b)	Returns the smaller of two `integer` values.	**integer**
min(integer a, real b)	Returns the smaller of two values.	**real**
min(real a, integer b)	Returns the smaller of two values.	**real**
min(real a, real b)	Returns the smaller of two `real` values.	**real**
nextAfter(real start, real direction)	Returns the floating-point number adjacent to the first argument in the direction of the second argument.	**real**
nextUp(real d)	Returns the floating-point value adjacent to `d` in the direction of positive infinity.	**real**
rint(real a)	Returns the `real` value that is closest in value to the argument and is equal to a mathematical integer.	**real**
round(real a)	Returns the closest `integer` to the argument, with ties rounding up.	**integer**
scalb(real d, int scaleFactor)	Return $d \times 2^{scaleFactor}$ rounded as if performed by a single correctly rounded floating-point multiply to a member of the double value set.	**real**
signum(real d)	Returns the signum function of the argument; zero if the argument is zero, 1.0 if the argument is greater than zero, -1.0 if the argument is less than zero.	**real**
sin(real a)	Returns the trigonometric sine of an angle.	**real**
sin(integer a)	Returns the trigonometric sine of an angle.	**real**
sin(realVector a)	Returns a vector whose elements are the trigonometric sines of the elements of the input vector a.	**realVector**
sin(realMatrix a)	Returns a matrix whose elements are the trigonometric sines of the elements of the input matrix a.	**realMatrix**
sinh(integer x)	Returns the hyperbolic sine of a `integer` value.	**real**
sinh(real x)	Returns the hyperbolic sine of a `real` value.	**real**
sinh(realVector x)	Returns a vector whose elements are the hyperbolic sines of the elements of the input vector a.	**realVector**
sinh(realMatrix x)	Returns a matrix whose elements are the hyperbolic sines of the elements of the input matrix a.	**realMatrix**
sqrt(integer a)	Returns the correctly rounded positive square root of a `integer` value.	**real**
sqrt(real a)	Returns the correctly rounded positive square root of a `real` value.	**real**
tan(real a)	Returns the trigonometric tangent of an angle.	**real**
tan(integer a)	Returns the trigonometric tangent of an angle.	**real**
tan(realVector a)	Returns a vector whose elements are the trigonometric tangents of the elements of the input vector a.	**realVector**
tan(realMatrix a)	Returns a matrix whose elements are the trigonometric tangents of the elements of the input matrix a.	**realMatrix**
tanh(integer x)	Returns the hyperbolic tangent of a `integer` value.	**real**
tanh(real x)	Returns the hyperbolic tangent of a `real` value.	**real**
tanh(realVector x)	Returns a vector whose elements are the hyperbolic tangents of the elements of the input vector a.	**realVector**

`tanh(real x)`	Returns a matrix whose elements are the hyperbolic tangents of the elements of the input matrix a.	**`realMatrix`**
`toDegrees(integer angrad)`	Converts an angle measured in radians to an approximately equivalent angle measured in degrees.	**`real`**
`toDegrees(real angrad)`	Converts an angle measured in radians to an approximately equivalent angle measured in degrees.	**`real`**
`toRadians(integer angdeg)`	Converts an angle measured in degrees to an approximately equivalent angle measured in radians.	**`real`**
`toRadians(real angdeg)`	Converts an angle measured in degrees to an approximately equivalent angle measured in radians.	**`real`**
`ulp(real d)`	Returns the size of an ulp of the argument.	**`real`**

A.2.1 toRadians

Signatures:

```
toRadians(integer   angdeg)
toRadians(real angdeg)
```

Description:

Converts an angle measured in degrees to an approximately equivalent angle measured in radians. The conversion from degrees to radians is generally inexact. Argument of type long converted to a double value.

Parameters:

- `angdeg` - an angle, in degrees

Returns:

- The measurement of the angle `angdeg` in radians. The return type is double.

A.2.2 toDegrees

Signature:

```
toDegrees(integer   angrad)
toDegrees(real angrad)
```

Description:

Converts an angle measured in radians to an approximately equivalent angle measured in degrees. The conversion from radians to degrees is generally inexact; users should *not* expect `cos(toRadians(90.0))` to exactly equal `0.0`. Argument of type long converted to a double value.

Parameters:

- `angrad` - an angle, in radians

Returns:

The measurement of the angle `a`

A.2.3 sin

Signatures:

```
sin(integer   a)
sin(real a)
sin(realVector a)
sin(realMatrix a)
```

Description:

Returns the trigonometric sine of an angle.

Special cases:

- If the argument is NaN or an infinity, then the result is NaN.
- If the argument is zero, then the result is a zero with the same sign as the argument.

The computed result must be within 1 ulp of the exact result. Results must be semi-monotonic. Argument of type long converted to a double value.

Parameters:

- `a` - an angle, in radians for type long and double.
- `a` – a vector whose elements are angles, in radians for type vector.
- `a` – a matrix whose elements are angles, in radians for type matrix.

Returns:

- The sine of the argument, for input type long or double.
- A vector whose elements are the sines of the elements of the vector argument, for the input type vector.
- A matrix whose elements are the sines of the elements of the matrix argument, for the input type matrix.

A.2.4 cos

Signatures:

```
cos(integer   a)
cos(real a)
cos(realVector a)
cos(realMatrix a)
```

Description:

Returns the trigonometric cosine of an angle.

Special cases:

- If the argument is NaN or an infinity, then the result is NaN.

The computed result must be within 1 ulp of the exact result. Results must be semi-monotonic. Argument of type long converted to a double value.

Parameters:

- `a` - an angle, in radians for type long and double.
- `a` – a vector whose elements are angles, in radians for type vector.
- `a` – a matrix whose elements are angles, in radians for type matrix.

Returns:

- The cosine of the argument, for input type long or double.
- A vector whose elements are the cosines of the elements of the vector argument, for the input type vector.
- A matrix whose elements are the cosines of the elements of the matrix argument, for the input type matrix.

A.2.5 tan

Signatures:

```
tan(integer   a)
tan(real a)
tan(realVector a)
tan(realMatrix a)
```

Description:

Returns the trigonometric tangent of an angle.

Special cases:

- If the argument is NaN or an infinity, then the result is NaN.
- If the argument is zero, then the result is a zero with the same sign as the argument.

The computed result must be within 1 ulp of the exact result. Results must be semi-monotonic. Argument of type long converted to a double value.

Parameters:

- `a` - an angle, in radians for type long and double.
- `a` – a vector whose elements are angles, in radians for type vector.
- `a` – a matrix whose elements are angles, in radians for type matrix.

Returns:

- The tangent of the argument, for input type long or double.
- A vector whose elements are the tangent of the elements of the vector argument, for the input type vector.
- A matrix whose elements are the tangent of the elements of the matrix argument, for the input type matrix.

A.2.6 asin

Signatures:

```
asin(integer   a)
asin(real a)
asin(realVector a)
asin(realMatrix a)
```

Description:

Returns the arc sine of a value; the returned angle is in the range -*pi*/2 through *pi*/2.

Special cases:

- If the argument is NaN or its absolute value is greater than 1, then the result is NaN.
- If the argument is zero, then the result is a zero with the same sign as the argument.

The computed result must be within 1 ulp of the exact result. Results must be semi-monotonic. Argument of type long converted to a double value.

Parameters:

- `a` - the value whose arc sine is to be returned, for type long and double.
- `a` – a vector whose elements are the values whose arc sine is to be returned, for type vector.
- `a` – a matrix whose elements are the values whose arc sine is to be returned, for type matrix.

Returns:

- The arc sine of the argument, for input type long or double.
- A vector whose elements are the arc sine of the elements of the vector argument, for the input type vector.
- A matrix whose elements are the arc sine of the elements of the matrix argument, for the input type matrix.

A.2.7 acos

Signatures:

```
acos(integer   a)
acos(real a)
acos(realVector a)
acos(realMatrix a)
```

Description:

Returns the arc cosine of a value; the returned angle is in the range 0.0 through *pi*.

Special case:

- If the argument is NaN or its absolute value is greater than 1, then the result is NaN.

The computed result must be within 1 ulp of the exact result. Results must be semi-monotonic. Argument of type long converted to a double value.

Parameters:

- `a` - the value whose arc cosine is to be returned, for type long and double.
- `a` – a vector whose elements are the values whose arc cosine is to be returned, for type vector.
- `a` – a matrix whose elements are the values whose arc cosine is to be returned, for type matrix.

Returns:

- The arc cosine of the argument, for input type long or double.
- A vector whose elements are the arc cosine of the elements of the vector argument, for the input type vector.
- A matrix whose elements are the arc cosine of the elements of the matrix argument, for the input type matrix.

A.2.8 atan

Signatures:

```
atan(integer   a)
atan(real a)
atan(realVector a)
atan(realMatrix a)
```

Description:

Returns the arc tangent of a value; the returned angle is in the range -*pi*/2 through *pi*/2.

Special cases:

- If the argument is NaN, then the result is NaN.
- If the argument is zero, then the result is a zero with the same sign as the argument.

The computed result must be within 1 ulp of the exact result. Results must be semi-monotonic. Argument of type long converted to a double value.

Parameters:

- `a` - the value whose arc tangent is to be returned, for type long and double.
- `a` – a vector whose elements are the values whose arc tangent is to be returned, for type vector.
- `a` – a matrix whose elements are the values whose arc tangent is to be returned, for type matrix.

Returns:

- The arc tangent of the argument, for input type long or double.
- A vector whose elements are the arc tangent of the elements of the vector argument, for the input type vector.
- A matrix whose elements are the arc tangent of the elements of the matrix argument, for the input type matrix.

A.2.9 atan2

Signatures:

```
atan2(integer   y, integer   x)
atan2(integer   y, real x)
atan2(real y, integer   x)
atan2(real y, real x)
```

Description:

Returns the angle *theta* from the conversion of rectangular coordinates (`x`, `y`) to polar coordinates (r, *theta*). This method computes the phase *theta* by computing an arc tangent of `y/x` in the range of -*pi* to *pi*.

Special cases:

- If either argument is NaN, then the result is NaN.
- If the first argument is positive zero and the second argument is positive, or the first argument is positive and finite and the second argument is positive infinity, then the result is positive zero.
- If the first argument is negative zero and the second argument is positive, or the first argument is negative and finite and the second argument is positive infinity, then the result is negative zero.

- If the first argument is positive zero and the second argument is negative, or the first argument is positive and finite and the second argument is negative infinity, then the result is the `real` value closest to *pi*.
- If the first argument is negative zero and the second argument is negative, or the first argument is negative and finite and the second argument is negative infinity, then the result is the `real` value closest to -*pi*.
- If the first argument is positive and the second argument is positive zero or negative zero, or the first argument is positive infinity and the second argument is finite, then the result is the `real` value closest to *pi*/2.
- If the first argument is negative and the second argument is positive zero or negative zero, or the first argument is negative infinity and the second argument is finite, then the result is the `real` value closest to -*pi*/2.
- If both arguments are positive infinity, then the result is the `real` value closest to *pi*/4.
- If the first argument is positive infinity and the second argument is negative infinity, then the result is the `real` value closest to 3**pi*/4.
- If the first argument is negative infinity and the second argument is positive infinity, then the result is the `real` value closest to -*pi*/4.
- If both arguments are negative infinity, then the result is the `real` value closest to -3**pi*/4.

The computed result must be within 2 ulps of the exact result. Results must be semi-monotonic. Arguments of type long converted to double values.

Parameters:

- `y` - the ordinate coordinate
- `x` - the abscissa coordinate

Returns:

- the *theta* component of the point (*r*, *theta*) in polar coordinates that corresponds to the point (*x*, *y*) in Cartesian coordinates. The return type is double.

A.2.10 sinh

Signatures:

```
sinh(integer   x)
sinh(real x)
sinh(realVector x)
sinh(realMatrix x)
```

Description:

Returns the hyperbolic sine of a `real` value. The hyperbolic sine of x is defined to be $(e^x - e^{-x})/2$ where e is Euler's number.

Special cases:

- If the argument is NaN, then the result is NaN.
- If the argument is infinite, then the result is an infinity with the same sign as the argument.
- If the argument is zero, then the result is a zero with the same sign as the argument.

The computed result must be within 2.5 ulps of the exact result.

Parameters:

- `x` - The number whose hyperbolic sine is to be returned for type long and double.
- `x` – a vector whose elements are the numbers whose hyperbolic sine is to be returned for type vector.
- `x` – a matrix whose elements are the numbers whose hyperbolic sine is to be returned for type matrix.

Returns:

- The hyperbolic sine of `x`, for input type long or double.
- A vector whose elements are the hyperbolic sine of `x` of the elements of the vector argument, for the input type vector.
- A matrix whose elements are the hyperbolic sine of `x` of the elements of the matrix argument, for the input type matrix.

A.2.11 cosh

Signatures:

```
cosh(integer   x)
cosh(real x)
cosh(realVector x)
cosh(realMatrix x)
```

Description:

Returns the hyperbolic cosine of a `real` value. The hyperbolic cosine of x is defined to be $(e^x + e^{-x})/2$ where e is Euler's number.

Special cases:

- If the argument is NaN, then the result is NaN.
- If the argument is infinite, then the result is positive infinity.
- If the argument is zero, then the result is `1.0`.

The computed result must be within 2.5 ulps of the exact result.

Parameters:

- `x` - The number whose hyperbolic cosine is to be returned for type long and double.
- `x` – a vector whose elements are the numbers whose hyperbolic cosine is to be returned for type vector.
- `x` – a matrix whose elements are the numbers whose hyperbolic cosine is to be returned for type matrix.

Returns:

- The hyperbolic cosine of `x` , for input type long or double.
- A vector whose elements are the hyperbolic cosine of `x` of the elements of the vector argument, for the input type vector.
- A matrix whose elements are the hyperbolic cosine of `x` of the elements of the matrix argument, for the input type matrix.

A.2.12 tanh

Signatures:

```
tanh(integer   x)
tanh(real x)
tanh(realVector x)
tanh(realMatrix x)
```

Description:

Returns the hyperbolic tangent of a `real` value. The hyperbolic tangent of x is defined to be $(e^x - e^{-x})/(e^x + e^{-x})$, in other words, $\sinh(x)/\cosh(x)$. Note that the absolute value of the exact tanh is always less than 1.

Special cases:

- If the argument is NaN, then the result is NaN.
- If the argument is zero, then the result is a zero with the same sign as the argument.

- If the argument is positive infinity, then the result is `+1.0`.
- If the argument is negative infinity, then the result is `-1.0`.

The computed result must be within 2.5 ulps of the exact result. The result of `tanh` for any finite input must have an absolute value less than or equal to 1. Note that once the exact result of tanh is within 1/2 of an ulp of the limit value of ±1, correctly signed `±1.0` should be returned.

Parameters:

- `x` - The number whose hyperbolic tangent is to be returned for type long and double.
- `x` – a vector whose elements are the numbers whose hyperbolic tangent is to be returned for type vector.
- `x` – a matrix whose elements are the numbers whose hyperbolic tangent is to be returned for type matrix.

Returns:

- The hyperbolic tangent of `x` , for input type long or double.
- A vector whose elements are the hyperbolic tangent of `x` of the elements of the vector argument, for the input type vector.
- A matrix whose elements are the hyperbolic tangent of `x` of the elements of the matrix argument, for the input type matrix.

- `ngrad` in degrees. The return type is double.

A.2.13 exp

Signatures:

```
exp(integer   a)
exp(real a)
```

Description:

Returns Euler's number *e* raised to the power of a `real` value.

Special cases:

- If the argument is NaN, the result is NaN.
- If the argument is positive infinity, then the result is positive infinity.

- If the argument is negative infinity, then the result is positive zero.

The computed result must be within 1 ulp of the exact result. Results must be semi-monotonic. Argument of type long converted to a double value.

Parameters:

- `a` - the exponent to raise *e* to.

Returns:

- The value e^{a}, where *e* is the base of the natural logarithms. The return type is double.

A.2.14 log

Signatures:

```
log(integer   a)
log(real a)
log(realVector a)
log(realMatrix a)
```

Description:

Returns the natural logarithm (base *e*) of a long, double, vector, or matrix value.

Special cases:

- If the argument is NaN or less than zero, then the result is NaN.
- If the argument is positive infinity, then the result is positive infinity.
- If the argument is positive zero or negative zero, then the result is negative infinity.

The computed result must be within 1 ulp of the exact result. Results must be semi-monotonic. Argument of type long converted to a double value.

Parameters:

- `a` - a value

Returns:

- the value ln `a`, the natural logarithm of `a`. The return type is double, vector, or matrix.

A.2.15 logb

Signatures:

```
logb(integer   a, integer   b)
logb(integer   a, real b)
logb(real a, integer   b)
logb(real a, real b)
logb(realVector a, integer   b)
logb(realVector a, real b)
logb(realMatrix a, integer   b)
logb(realMatrix a, real b)
```

Description:

Returns the base `b` logarithm of a long, double, vector, or matrix value.

Special cases:

- If the argument is NaN or less than zero, then the result is NaN.
- If the argument is positive infinity, then the result is positive infinity.
- If the argument is positive zero or negative zero, then the result is negative infinity.
- If the argument is equal to 10^n for integer n, then the result is n.

The computed result must be within 1 ulp of the exact result. Results must be semi-monotonic. Argument of type long converted to a double value.

Parameters:

- `a` - a value

Returns:

- the base `b` logarithm of `a`. The return type is double, vector, or matrix.

A.2.16 log2

Signatures:

```
log2(integer   a)
log2(real a)
log2(realVector a)
log2(realMatrix a)
```

Description:

Returns the base 10 logarithm of a long, double, vector, or matrix value.

Special cases:

- If the argument is NaN or less than zero, then the result is NaN.
- If the argument is positive infinity, then the result is positive infinity.
- If the argument is positive zero or negative zero, then the result is negative infinity.
- If the argument is equal to 10^n for integer *n*, then the result is *n*.

The computed result must be within 1 ulp of the exact result. Results must be semi-monotonic. Argument of type long converted to a double value.

Parameters:

- `a` - a value

Returns:

- the base 2 logarithm of `a`. The return type is double, vector, or matrix.

A.2.17 log10

Signatures:

```
log10(integer   a)
log10(real a)
log10(realVector a)
log10(realMatrix a)
```

Description:

Returns the base 10 logarithm of a `real` value.

Special cases:

- If the argument is NaN or less than zero, then the result is NaN.
- If the argument is positive infinity, then the result is positive infinity.
- If the argument is positive zero or negative zero, then the result is negative infinity.
- If the argument is equal to 10^n for integer *n*, then the result is *n*.

The computed result must be within 1 ulp of the exact result. Results must be semi-monotonic. Argument of type long converted to a double value.

Parameters:

- `a` - a value

Returns:

- the base 10 logarithm of `a`. The return type is double, vector, or matrix.

A.2.18 sqrt

Signature:

```
sqrt(integer   a)
sqrt(real a)
```

Description:

Returns the correctly rounded positive square root of a `real` value.

Special cases:

- If the argument is NaN or less than zero, then the result is NaN.
- If the argument is positive infinity, then the result is positive infinity.
- If the argument is positive zero or negative zero, then the result is the same as the argument.

Otherwise, the result is the `real` value closest to the true mathematical square root of the argument value. Argument of type long converted to a double value.

Parameters:

- `a` - a value.

Returns:

- the positive square root of `a`. If the argument is NaN or less than zero, the result is NaN. The return type is double.

A.2.19 cbrt

Signature:

```
cbrt(integer   a)
cbrt(real a)
```

Description:

Returns the cube root of a `real` value. For positive finite `x`, `cbrt(-x) == -cbrt(x)`; that is, the cube root of a negative value is the negative of the cube root of that value's magnitude.

Special cases:

- If the argument is NaN, then the result is NaN.
- If the argument is infinite, then the result is an infinity with the same sign as the argument.
- If the argument is zero, then the result is a zero with the same sign as the argument.

The computed result must be within 1 ulp of the exact result. Argument of type long converted to a double value.

Parameters:

- `a` - a value.

Returns:

- the cube root of `a`. The return type is double.

A.2.20 IEEEremainder

Signature:

```
IEEEremainder(integer   f1, integer   f2)
IEEEremainder(integer   f1, real f2)
IEEEremainder(real f1, integer   f2)
IEEEremainder(real f1, real f2)
```

Description:

Computes the remainder operation on two arguments as prescribed by the IEEE 754 standard. The remainder value is mathematically equal to `f1 - f2` × *n*, where *n* is the mathematical integer closest to the exact mathematical value of the quotient `f1/f2`, and if two mathematical integers are equally close to `f1/f2`, then *n* is the integer that is even. If the remainder is zero, its sign is the same as the sign of the first argument.

Special cases:

- If either argument is NaN, or the first argument is infinite, or the second argument is positive zero or negative zero, then the result is NaN.
- If the first argument is finite and the second argument is infinite, then the result is the same as the first argument.

Arguments of type long converted to double values.

Parameters:

- `f1` - the dividend.
- `f2` - the divisor.

Returns:

- the remainder when `f1` is divided by `f2`. The return type is double.

A.2.21 ceil

Signature:

```
ceil(real a)
```

Description:

Returns the smallest (closest to negative infinity) double value that is greater than or equal to the argument and is equal to a mathematical integer.

Special cases:

- If the argument value is already equal to a mathematical integer, then the result is the same as the argument.
- If the argument is NaN or an infinity or positive zero or negative zero, then the result is the same as the argument.
- If the argument value is less than zero but greater than -1.0, then the result is negative zero.

Note that the value of `ceil(x)` is exactly the value of `-floor(-x)`.

Parameters:

- `a` - a value.

Returns:

- the smallest (closest to negative infinity) floating-point value that is greater than or equal to the argument and is equal to a mathematical integer.

A.2.22 floor

Signature:

```
floor(real a)
```

Description:

Returns the largest (closest to positive infinity) `real` value that is less than or equal to the argument and is equal to a mathematical integer.

Special cases:

- If the argument value is already equal to a mathematical integer, then the result is the same as the argument.
- If the argument is NaN or an infinity or positive zero or negative zero, then the result is the same as the argument.

Parameters:

- `a` - a value.

Returns:

- the largest (closest to positive infinity) floating-point value that less than or equal to the argument and is equal to a mathematical integer.

A.2.23 rint

Signature:

```
rint(real a)
```

Description:

Returns the `real` value that is closest in value to the argument and is equal to a mathematical integer. If two `real` values that are mathematical integers are equally close, the result is the integer value that is even.

Special cases:

- If the argument value is already equal to a mathematical integer, then the result is the same as the argument.
- If the argument is NaN or an infinity or positive zero or negative zero, then the result is the same as the argument.

Parameters:

- `a` - a `real` value.

Returns:

- the closest floating-point value to `a` that is equal to a mathematical integer.

A.2.24 round

Signature:

`round(real a)`

Description:

Returns the closest `integer` to the argument, with ties rounding up.

Special cases:

- If the argument is NaN, the result is 0.
- If the argument is negative infinity or any value less than or equal to the value of the minimum long value, the result is equal to the value of the minimum long value.
- If the argument is positive infinity or any value greater than or equal to the value of the maximum long value, the result is equal to the value of the maximum long value.

Parameters:

- `a` - a floating-point value to be rounded to a `integer`.

Returns:

- the value of the argument rounded to the nearest `integer` value.

A.2.25 abs

Signature:

```
abs(integer  a)
abs(real a)
abs(realVector a)
abs(realMatrix a)
```

Description:

Returns the absolute value of the argument. If the argument is not negative, the argument is returned. If the argument is negative, the negation of the argument is returned.

Special cases:

- If the argument is positive zero or negative zero, the result is positive zero.
- If the argument is infinite, the result is positive infinity.
- If the argument is NaN, the result is NaN.

Parameters:

- `a` - the argument whose absolute value is to be determined for type long and double.
- `a` – a vector whose elements are the values whose absolute values is to be determined for type vector.
- `a` – a matrix whose elements are the values whose absolute values is to be determined for type matrix.

Returns:

- The absolute value of the argument, for input type long or double.
- A vector whose elements are the absolute value of the elements of the vector argument, for the input type vector.
- A matrix whose elements are the absolute value of the elements of the matrix argument, for the input type matrix.

A.2.26 max

Signature:

```
max(integer  a, integer b)
max(integer  a, real b)
max(real a, integer  b)
```

```
max(real a, real b)
```

Description:

Returns the greater of two values. That is, the result is the argument closer to positive infinity. If the arguments have the same value, the result is that same value. If either value is NaN, then the result is NaN. Unlike the numerical comparison operators, this method considers negative zero to be strictly smaller than positive zero. If one argument is positive zero and the other negative zero, the result is positive zero.

Parameters:

- `a` - an argument.
- `b` - another argument.

Returns:

- the larger of `a` and `b`. The return type is long if both arguments are of long type. Otherwise, the return type is double.

A.2.27 min

Signature:

```
min(integer   a, integer   b)
min(integer   a, real b)
min(real a, integer   b)
min(real a, real b)
```

Description:

Returns the smaller of two values. That is, the result is the value closer to negative infinity. If the arguments have the same value, the result is that same value. If either value is NaN, then the result is NaN. Unlike the numerical comparison operators, this method considers negative zero to be strictly smaller than positive zero. If one argument is positive zero and the other is negative zero, the result is negative zero.

Parameters:

- `a` - an argument.
- `b` - another argument.

Returns:

- the smaller of `a` and `b`. The return type is long if both arguments are of long type. Otherwise, the return type is double.

A.2.28 ulp

Signature:

`ulp(real d)`

Description:

Returns the size of an ulp of the argument. An ulp of a `real` value is the positive distance between this floating-point value and the `real` value next larger in magnitude. Note that for non-NaN *x*, `ulp(-x) == ulp(x)`.

Special Cases:

- If the argument is NaN, then the result is NaN.
- If the argument is positive or negative infinity, then the result is positive infinity.
- If the argument is positive or negative zero, then the result is the minimum double value.
- If the argument is ±(the maximum double value), then the result is equal to 2^{971}.

Parameters:

- `d` - the floating-point value whose ulp is to be returned

Returns:

- the size of an ulp of the argument

A.2.29 signum

Signature:

`signum(real d)`

Description:

Returns the signum function of the argument; zero if the argument is zero, 1.0 if the argument is greater than zero, -1.0 if the argument is less than zero.

Special Cases:

- If the argument is NaN, then the result is NaN.
- If the argument is positive zero or negative zero, then the result is the same as the argument.

Parameters:

- `d` - the floating-point value whose signum is to be returned

Returns:

- the signum function of the argument

A.2.30 hypot

Signature:

`hypot(real x, real y)`

Description:

Returns sqrt(x^2 +y^2) without intermediate overflow or underflow.

Special cases:

- If either argument is infinite, then the result is positive infinity.
- If either argument is NaN and neither argument is infinite, then the result is NaN.

The computed result must be within 1 ulp of the exact result. If one parameter is held constant, the results must be semi-monotonic in the other parameter.

Parameters:

- `x` - a value
- `y` - a value

Returns:

- sqrt(x^2 +y^2) without intermediate overflow or underflow

A.2.31 expm1

Signature:

`expm1(real x)`

Description:

Returns e^x -1. Note that for values of x near 0, the exact sum of `expm1(x)` + 1 is much closer to the true result of e^x than `exp(x)`.

Special cases:

- If the argument is NaN, the result is NaN.
- If the argument is positive infinity, then the result is positive infinity.
- If the argument is negative infinity, then the result is -1.0.
- If the argument is zero, then the result is a zero with the same sign as the argument.

The computed result must be within 1 ulp of the exact result. Results must be semi-monotonic. The result of `expm1` for any finite input must be greater than or equal to `-1.0`. Note that once the exact result of e^x - 1 is within 1/2 ulp of the limit value -1, `-1.0` should be returned.

Parameters:

- `x` - the exponent to raise e to in the computation of e^x -1.

Returns:

- the value e^x - 1.

A.2.32 log1p

`log1p(real x)`

Description:

Returns the natural logarithm of the sum of the argument and 1. Note that for small values `x`, the result of `log1p(x)` is much closer to the true result of ln(1 + `x`) than the floating-point evaluation of `log(1.0+x)`.

Special cases:

- If the argument is NaN or less than -1, then the result is NaN.
- If the argument is positive infinity, then the result is positive infinity.
- If the argument is negative one, then the result is negative infinity.
- If the argument is zero, then the result is a zero with the same sign as the argument.

The computed result must be within 1 ulp of the exact result. Results must be semi-monotonic.

Parameters:

- `x` - a value

Returns:

- the value ln(`x` + 1), the natural log of `x` + 1

A.2.33 copySign

Signature:

`copySign(real magnitude, real sign)`

Description:

Returns the first floating-point argument with the sign of the second floating-point argument. Note that unlike the StrictMath.copySign method, this method does not require NaN `sign` arguments to be treated as positive values; implementations are permitted to treat some NaN arguments as positive and other NaN arguments as negative to allow greater performance.

Parameters:

- `magnitude` - the parameter providing the magnitude of the result
- `sign` - the parameter providing the sign of the result

Returns:

- a value with the magnitude of `magnitude` and the sign of `sign`.

A.2.34 getExponent

Signature:

`getExponent(real d)`

Description:

Returns the unbiased exponent used in the representation of a `real`. Special cases:

- If the argument is NaN or infinite, then the result is `Real.MAX_EXPONENT` + 1.
- If the argument is zero or subnormal, then the result is `Double.MIN_EXPONENT` -1.

Parameters:

- `d` - a `real` value

Returns:

- the unbiased exponent of the argument

A.2.35 nextAfter

Signature:

`nextAfter(real start, real direction)`

Description:

Returns the floating-point number adjacent to the first argument in the direction of the second argument. If both arguments compare as equal the second argument is returned.

Special cases:

- If either argument is a NaN, then NaN is returned.
- If both arguments are signed zeros, `direction` is returned unchanged (as implied by the requirement of returning the second argument if the arguments compare as equal).
- If `start` is ±(minimum double value) and `direction` has a value such that the result should have a smaller magnitude, then a zero with the same sign as `start` is returned.
- If `start` is infinite and `direction` has a value such that the result should have a smaller magnitude, the maximum double **value** with the same sign as `start` is returned.
- If `start` is equal to ± (the maximum long value) and `direction` has a value such that the result should have a larger magnitude, an infinity with same sign as `start` is returned.

Parameters:

- `start` - starting floating-point value
- `direction` - value indicating which of `start`'s neighbors or `start` should be returned

Returns:

- The floating-point number adjacent to `start` in the direction of `direction`.

A.2.36 nextUp

Signature:

`nextUp(real d)`

Description:

Returns the floating-point value adjacent to `d` in the direction of positive infinity.

Special Cases:

- If the argument is NaN, the result is NaN.
- If the argument is positive infinity, the result is positive infinity.
- If the argument is zero, the result is the minimum double value

Parameters:

- `d` - starting floating-point value

Returns:

- The adjacent floating-point value closer to positive infinity.

A.2.37 scalb

Signature:

`scalb(real d, int scaleFactor)`

Description:

Return $d \times 2^{\mathtt{scaleFactor}}$ rounded as if performed by a single correctly rounded floating-point multiply to a member of the double value set. See the Java Language Specification for a discussion of floating-point value sets.

Special cases:

- If the first argument is NaN, NaN is returned.

- If the first argument is infinite, then an infinity of the same sign is returned.
- If the first argument is zero, then a zero of the same sign is returned.

Parameters:

- `d` - number to be scaled by a power of two.
- `scaleFactor` - power of 2 used to scale `d`

Returns:

- $d \times 2^{scaleFactor}$

Appendix B -- Library: Math2

B.1 Classes

Class Name	Class Path
Math2	`library.Math2`

B.2 Functions

Call Signature	Description	Return Type
`diagonal(integer size, real value)`	Returns a diagonal real matrix of size **`size`** x **`size`** and the elements of the diagonal set to the values of the parameter **`value`**.	**`realMatrix`**
`diagonal(real size, real value)`	Returns a diagonal real matrix of size **`size`** x **`size`** and the elements of the diagonal set to the values of the parameter **`value`**.	**`realMatrix`**
`diagonal(real size, integer value)`	Returns a diagonal real matrix of size **`size`** x **`size`** and the elements of the diagonal set to the values of the parameter **`value`**.	**`realMatrix`**
`diagonal(real size, real value)`	Returns a diagonal real matrix of size **`size`** x **`size`** and the elements of the diagonal set to the values of the parameter **`value`**.	**`realMatrix`**
`divElemByElem(realVector left, realVector right)`	Returns a real vector whose elements are ratios of the corresponding elements of the parameters **`left`** and **`right`**.	**`realVector`**
`divElemByElem(realMatrix left, realMatrix right)`	Returns a real matrix whose elements are ratios of the corresponding elements of the parameters **`left`** and **`right`**.	**`realMatrix`**
`identity(integer size)`	Returns a diagonal real matrix of size **`size`** x **`size`** and the elements of the diagonal set to 1.0.	**`realMatrix`**
`identity(real size)`	Returns a diagonal real matrix of size **`size`** x **`size`** and the elements of the diagonal set to 1.0.	**`realMatrix`**
`multElemByElem(realVector left, realVector right)`	Returns a real vector whose elements are products of the corresponding elements of the parameters **`left`** and **`right`**.	**`realVector`**
`multElemByElem(realMatrix left, realMatrix right)`	Returns a real matrix whose elements are products of the corresponding elements of the parameters **`left`** and **`right`**.	**`realMatrix`**
`ones(integer length)`	Returns a real vector of length **`length`** and all the elements set to 1.0.	**`realVector`**
`ones(real length)`	Returns a real vector of length **`length`** and all the elements set to 1.0.	**`realVector`**

`ones(integer row, integer col)`	Returns a real matrix of size **row** x **col** and all the elements set to 1.0.	`realMatrix`
`ones(real row, integer col)`	Returns a real matrix of size **row** x **col** and all the elements set to 1.0.	`realMatrix`
`ones(integer row, real col)`	Returns a real matrix of size **row** x **col** and all the elements set to 1.0.	`realMatrix`
`ones(real row, real col)`	Returns a real matrix of size **row** x **col** and all the elements set to 1.0.	`realMatrix`
`transpose(realMatrix a)`	Returns transpose of **a**.	`realMatrix`
`zeros(integer length)`	Returns a real vector of length **length** and all the elements set to 0.0.	`realVector`
`zeros(real length)`	Returns a real vector of length **length** and all the elements set to 0.0.	`realVector`
`zeros(integer row, integer col)`	Returns a real matrix of size **row** x **col** and all the elements set to 0.0.	`realMatrix`
`zeros(real row, integer col)`	Returns a real matrix of size **row** x **col** and all the elements set to 0.0.	`realMatrix`
`zeros(integer row, real col)`	Returns a real matrix of size **row** x **col** and all the elements set to 0.0.	`realMatrix`
`zeros(real row, real col)`	Returns a real matrix of size **row** x **col** and all the elements set to 0.0.	`realMatrix`

Appendix C -- Library: Linear Algebra

The Linear Algebra library composed of six classes: All, Utility, Linear Equations class, Linear Least Square class, Singular Value class, and Eigen class.

C.1 Classes

The table below lists the classes and their paths.

Class Name	Class Path
All	`library.linear_algebra.All`
Linear Equations	`library.linear_algebra.LinearEquations`
Linear Least Square	`library.linear_algebra.LinearLeastSquare`
Eigen	`library.linear_algebra.Eigen`
Singular Value	`library.linear_algebra.SingularValue`

C.2 Functions

The table below lists in alphabetical order the functions in the Utility class.

Call Signature	Description	Return Type
`arrayToVec(array ar, boolean real, boolean imag, boolean comp)`	Returns a real or complex vector produced from the array input **`ar`**. The boolean parameters **`real, imag`** and **`comp`** indicates if the input **`ar`** contains real, imaginary or complex elements.	**`realVector`** or **`complexVector`**
`arrayToMat(array ar, boolean real, boolean comp)`	Returns a real or complex matrix produced from the array input **`ar`**. The boolean parameters **`real`** and **`comp`** indicates if the input **`ar`** contains real or complex elements.	**`realMatrix`** or **`complexMatrix`**
`diagonal(integer size, real value)`	Returns a diagonal real matrix of size **`size`** x **`size`** and the elements of the diagonal set to the values of the parameter **`value`**.	**`realMatrix`**
`diagonal(real size, real value)`	Returns a diagonal real matrix of size **`size`** x **`size`** and the elements of the diagonal set to the values of the parameter **`value`**.	**`realMatrix`**
`diagonal(real size, integer value)`	Returns a diagonal real matrix of size **`size`** x **`size`** and the elements of the diagonal set to the values of the parameter **`value`**.	**`realMatrix`**
`diagonal(real size, real value)`	Returns a diagonal real matrix of size **`size`** x **`size`** and the elements of the diagonal set to the values of the parameter **`value`**.	**`realMatrix`**
`divElemByElem(realVector left, realVector right)`	Returns a real vector whose elements are ratios of the corresponding elements of the parameters **`left`** and **`right`**.	**`realVector`**
`divElemByElem(realMatrix left, realMatrix right)`	Returns a real matrix whose elements are ratios of the corresponding elements of the parameters **`left`** and **`right`**.	**`realMatrix`**
`findNonZero(realVector v)`	Returns an array of long whose elements are indices of the non-zero elements of the parameter **`v`**.	**`array of integers`**
`identity(integer size)`	Returns a diagonal real matrix of size **`size`** x **`size`** and the elements of the diagonal set to 1.0.	**`realMatrix`**
`identity(real size)`	Returns a diagonal real matrix of size **`size`** x **`size`** and the elements of the diagonal set to 1.0.	**`realMatrix`**
`isSquare(realMatrix a)`	Returns TRUE if the matrix in parameter **`a`** is square. Otherwise, returns FALSE.	**`boolean`**
`isSymmetric(realMatrix a)`	Returns TRUE if the matrix in parameter **`a`** is symmetric. Otherwise, returns FALSE.	**`boolean`**
`locate(realVector v, real d)`	Returns the matched index of the parameter **`d`** in the parameter vector **`v`**.	**`integer`**
`multElemByElem(realVector left, realVector right)`	Returns a real vector whose elements are products of the corresponding elements of the parameters **`left`** and **`right`**.	**`realVector`**
`multElemByElem(realMatrix`	Returns a real matrix whose elements are	**`realMatrix`**

left, realMatrix right)	products of the corresponding elements of the parameters **left** and **right**.	
ones(integer length)	Returns a real vector of length **length** and all the elements set to 1.0.	**realVector**
ones(real length)	Returns a real vector of length **length** and all the elements set to 1.0.	**realVector**
ones(integer row, integer col)	Returns a real matrix of size **row** X **col** and all the elements set to 1.0.	**realMatrix**
ones(real row, integer col)	Returns a real matrix of size **row** X **col** and all the elements set to 1.0.	**realMatrix**
ones(integer row, real col)	Returns a real matrix of size **row** X **col** and all the elements set to 1.0.	**realMatrix**
ones(real row, real col)	Returns a real matrix of size **row** X **col** and all the elements set to 1.0.	**realMatrix**
transpose(realMatrix a)	Returns transpose of **a**.	**realMatrix**
zeros(integer length)	Returns a real vector of length **length** and all the elements set to 0.0.	**realVector**
zeros(real length)	Returns a real vector of length **length** and all the elements set to 0.0.	**realVector**
zeros(integer row, integer col)	Returns a real matrix of size **row** X **col** and all the elements set to 0.0.	**realMatrix**
zeros(real row, integer col)	Returns a real matrix of size **row** X **col** and all the elements set to 0.0.	**realMatrix**
zeros(integer row, real col)	Returns a real matrix of size **row** X **col** and all the elements set to 0.0.	**realMatrix**
zeros(real row, real col)	Returns a real matrix of size **row** X **col** and all the elements set to 0.0.	**realMatrix**

The table below lists in alphabetical order the functions in the Linear Equations class.

Call Signature	Description	Return Type
decomposeLUP(realMatrix A)	Decomposes matrix A using LUP factorization.	**void**
determinant()	Returns determinant of a matrix if the matrix is already been LUP factorized.	**real**
determinant(realMatrix A)	Returns determinant after LUP factorizing matrix A.	**real**
inverse()	Returns inverse of a matrix if the matrix is already been LUP factorized.	**realMatrix**
inverse(realMatrix A)	Returns inverse after LUP factorizing matrix A.	**realMatrix**
lower()	Returns the lower triangular matrix if the matrix is already been LUP factorized.	**realMatrix**
lup()	Returns the lower triangular matrix, the upper triangular matrix, and the permutation matrix if	**array**

	the matrix is already been LUP factorized.	
`lup(realMatrix A)`	Returns the lower triangular matrix, the upper triangular matrix, and the permutation matrix after LUP factorizing matrix `A`.	**`array`**
`permutation()`	Returns the permutation matrix if the matrix is already been LUP factorized.	**`realMatrix`**
`solve(realMatrix A, realVector b)`	Returns the solution vector x of the equation **Ax** = **b** after LUP factorizing matrix `A`.	**`realVector`**
`solve(realMatrix A, realMatrix B)`	Returns the solution matrix X of the equation **AX** = **B** after LUP factorizing matrix `A`.	**`realMatrix`**
`solve(realVector b)`	Returns the solution vector x of the equation **Ax** = **b** if the matrix is already been LUP factorized.	**`realVector`**
`solve(realMatrix B)`	Returns the solution matrix X of the equation **AX** = **B** if the matrix is already been LUP factorized.	**`realMatrix`**
`trace()`	Returns trace of a matrix if the matrix is already been LUP factorized.	**`real`**
`trace(realMatrix mat)`	Returns trace of a matrix after LUP factorizing matrix `A`..	**`real`**
`upper()`	Returns the upper triangular matrix if the matrix is already been LUP factorized.	**`realMatrix`**

The table below lists in alphabetical order the functions in the Linear Least Square class.

Call Signature	Description	Return Type
`decomposeQR(realMatrix A)`	Decomposes matrix `A` using QR factorization.	**`void`**
`inverseLS()`	Returns inverse of a matrix if the matrix is already been QR factorized.	**`realMatrix`**
`getAid()`	Returns the diagonal elements of the upper triangular matrix produced during factorization.	**`realVector`**
`getP()`	Returns P matrix.	**`realMatrix`**
`getQ()`	Returns Q matrix.	**`realMatrix`**
`getE()`	Returns R matrix.	**`realMatrix`**
`getMaxEuclidNorm()`	Returns the maximum of the Euclidean norms of the columns of the given matrix.	**`real`**
`getTolerance()`	Returns the relative tolerance used for calculating diagonal elements of the upper triangular matrix.	**`real`**
`inverseLS(realMatrix A)`	Returns inverse after QR factorizing matrix `A`.	**`realMatrix`**
`solveLS(realVector b)`	Returns the solution vector x of the equation **Ax** = **b** if the matrix is already been QR factorized.	**`realVector`**
`setTolerance(real tolerance)`		**`void`**
`solveLS(realMatrix A, realVector b)`	Returns the solution vector x of the equation **Ax** = **b** after QR factorizing matrix `A`.	**`realVector`**

`solveLS(realMatrix A, integer n, realVector b)`	Returns the solution vector x of the equation **Ax** = **b** after QR factorizing matrix `A` with the orthogonal matrix Q of order `n`.	`realVector`
`qr()`	Returns orthogonal matrices QR of a matrix if the matrix is already been QR factorized.	`array`
`qr(realMatrix A)`	Returns orthogonal matrices Q, R after factorizing matrix `A`.	`array`
`qrE(realMatrix A)`	Returns orthogonal matrices Q, R after producing an "economy-size" decomposition matrix `A`.	`array`

The table below lists in alphabetical order the functions in the Eigen class.

Call Signature	Description	Return Type
`eigen(realMatrix A)`	Returns real or complex eigen values and corresponding real or complex eigen vectors of a real *nxn* matrix `A`.	`array (of realMatrix or complexMatrix)`
`eigen(complexMatrix  A)`	Returns real or complex eigen values and corresponding real or complex eigen vectors of a complex *nxn* matrix `A`.	`array (of realMatrix or complexMatrix)`
`eigenValue()`	Returns real or complex eigen values which already been computed by calling the function `eigen(matrix A)` or `eigen(complexMatrix  A)`.	`realMatrix or complexMatrix`
`eigenReal_vector()`	Returns real or complex eigen vectors which already been computed by calling the function `eigen(matrix A)` or `eigen(complexMatrix  A)`.	`realMatrix or complexMatrix`

The table below lists in alphabetical order the functions for the Singular Value class.

Call Signature	Description	Return Type
`decomposeSVD(realMatrix A)`	Decomposes real matrix `A` using SVD factorization.	`void`
`decomposeSVD(complexMatrix A)`	Decomposes complex matrix `A` using SVD factorization.	`void`
`getMinNonNegSingularValue()`	Returns the minimum non-negative singular value.	`real`
`getU()`	Returns an m × m real or complex unitary matrix U.	`realMatrix or complexMatrix`
`getS()`	Returns an m × n rectangular diagonal matrix S with non-negative real numbers	`realMatrix or`

	on the diagonal.	**complexMatrix**
getV()	Returns an n × n real or complex unitary matrix	**realMatrix or complexMatrix**
pseudoinverse()	Returns a pseudo inverse of a real matrix if the matrix is already been SVD factorized.	**realMatrix**
pseudoinverse(realMatrix A)	Returns a pseudo inverse of a real matrix after SVD factorizing matrix `A`.	**realMatrix**
rank()	Returns the rank of a matrix if the matrix is already been SVD factorized.	**integer**
rank(realMatrix A)	Returns the rank of a real matrix after SVD factorizing matrix `A`.	**integer**
rank(complexMatrix A)	Returns the rank of a complex matrix after SVD factorizing matrix `A`.	**integer**
setMinNonNegSingularValue(real value)	Sets the minimum non-negative singular value.	**void**
solvesvd(realVector b)	Returns the solution vector x of the equation **Ax** = **b** if the matrix is already been SVD factorized.	**realVector**
solvesvd(realMatrix A, realVector b)	Returns the solution vector x of the equation **Ax** = **b** after SVD factorizing matrix `A`.	**realVector**
svd()	Returns an orthogonal m × m real or complex unitary matrix U, an m × n rectangular diagonal matrix S with non-negative real numbers on the diagonal, and an orthogonal n × m real or complex unitary matrix V of a matrix if the matrix is already been SVD factorized.	**array (of realMatrix or complexMatrix)**
svd(realMatrix A)	Returns an orthogonal m × m real matrix U, an m × n rectangular diagonal matrix S with non-negative real numbers on the diagonal, and an orthogonal n × m real matrix V of a real matrix `A` after SVD factorizing matrix `A`.	**array (of realMatrix or complexMatrix)**
svd(complexMatrix A)	Returns an orthogonal m × m complex unitary matrix U, diagonal matrix S, and an orthogonal n × m real or complex unitary matrix V of a complex matrix `A` after SVD factorizing matrix `A`.	**array (of realMatrix or complexMatrix)**

Appendix D -- Library: Zero Min Max

The Zero Min Max library contains two classes: RootFinder and Optimization.

D.1 Classes

The table below lists the classes and their paths.

Class Name	Class Path
All	`library.zero_min_max_eval.All`
RootFinder	`library.zero_min_max.RootFinder`
Optimizer	`library.zero_min_max_eval.Optimizer`

D.2 Functions

The table below lists in alphabetical order the functions in the Root Finder class.

Call Signature	Description	Return Type
`bisection(String fcnName, real x1, real x2, real prec, int maxItarations)`	Returns zero crossing of a function using the Bisection method. Terminates when `maxItarations` reached.	`real`
`bisection(String fcnName, real x1, real x2)`	Returns zero crossing of a function using the Bisection method.	`real`
`bisection(String fcnName, real x1, real x2, real prec)`	Returns zero crossing of a function using the Bisection method.	`real`
`bisection(real x1, real x2)`	Returns zero crossing of a function using Bisection method. Valid after setting root finding method to Bisection.	`real`
`bisection(real x1, real x2, real prec)`	Returns zero crossing of a function using Bisection method. Valid after setting root finding method to Bisection.	`real`
`getIterations()`	Return number of iterations used to find root.	`integer`
`getMaxIterations()`	Return the maximum number of iterations will be used to find root.	`integer`
`getPrecision()`	Returns relative precision used to determine convergence.	`real`
`newton(String fcnName, real start, real prec)`	Returns zero crossing of a function using the Newton method.	`real`
`newton(String fcnName, real start, real prec, int maxItarations)`	Returns zero crossing of a function using the Newton method. Terminates when `maxItarations` reached.	`real`
`newton(String fcnName, real start)`	Returns zero crossing of a function using the Newton method.	`real`

`newton(real start)`	Returns zero crossing of a function using the Newton method. Valid after setting root finding method to Newton.	`real`
`newton(real start, real prec)`	Returns zero crossing of a function using the Newton method. Valid after setting root finding method to Newton.	`real`
`roots(HYP_PolynomialValue poly)`	Returns roots of a polynomial.	`realVector` or `complexVector`
`setMaxIterations(integer maxItarations)`	Sets the maximum number of iterations to be used.	`void`
`setPrecision(real prec)`	Sets the relative precision to be used.	`void`
`setFunction(String fcnName)`	Sets the function who's roots will searched.	`void`
`setBisection(String fcnName)`	Sets the method to be Bisection for a new function.	`void`
`setBisection()`	Sets the method to be Bisection for an existing function.	`void`
`setNewton(String fcnName)`	Sets the method to be Newton for a new function.	`void`
`newton(real start)`	Sets the method to be Newton for an existing function.	`void`

The table below lists in alphabetical order the functions in the Optimization class.

Call Signature	Description	Return Type
`optimize()`	Performs optimization.	`realVector`
`powell(String fcnName, realVector guess)`	Sets the function to be optimized, sets the initial guess values and performs optimization using hill climbing method.	`realVector`
`powell(String fcnName, realVector guess, integer maxIter)`	Sets the function to be optimized, sets the initial guess values and performs optimization using hill climbing method. Terminates when `maxIter` reached.	`realVector`
`simplex(String fcnName, realVector guess)`	Sets the function to be optimized, sets the initial guess values and performs optimization using Simplex method	`realVector`
`simplex(String fcnName, realVector guess, integer maxIter)`	Sets the function to be optimized, sets the initial guess values and performs optimization using Simplex method. Terminates when `maxIter` reached.	`realVector`
`setFunction(String fcnName)`	Sets function to be optimized.	`void`
`setGuess(realVector guess)`	Sets the initial guess values.	`void`
`setOptimizer(String optName)`	Sets the optimization method.	`void`

setStrategy(String strategyName)	Sets optimization strategy (minimize or maximize)	**void**

Appendix E -- Library: Analysis

The Analysis library can be used to compute numerical derivatives and numerical integral of functions and to get solutions for ordinary differential equations (ODE). The Analysis library contains four classes: All, Differentiator, Integrator, and ODE Solver.

E.1 Classes

The table below lists the classes and their paths.

Class Name	Class Path
All	**`library.analysis.All`**
Differentiator	**`library.analysis.Differentiator`**
Integrator	**`library.analysis.Integrator`**
HYP_ODE	**`library.analysis.HYP_ODE`**
ODE_Solver	**`library.analysis.ODE_Solver`**

E.2 Functions

The table below lists in alphabetical order the functions in the Differentiator class.

Call Signature	Description	Return Type
`derivative(polynomial poly)`	Returns the derivative of a polynomial.	**`polynomial`**
`dydx(String fcnSignature, real x, real stepSize)`	Returns the approximate derivative of a new function at `x`.	**`real`**
`dydx(String fcnSignature, real x)`	Returns the approximate derivative of an existing function at `x`.	**`real`**
`dydx(realVector X, realVector Y)`	Returns an approximate differentiation of `Y` with respect to `X`.	**`realVector`**
`jacobian(String fcnSignature, realVector x)`	Returns an approximate partial derivative at `x`.	**`realMatrix`**
`setStepSize(real stepSize)`	Sets the step size for differentiation or integration.	**`void`**

The table below lists in alphabetical order the functions in the Integrator class.

Call Signature	Description	Return Type
`integral(polynomial poly, real constant)`	Returns the integral of a polynomial.	`polynomial`
`integral(polynomial poly, integer constant)`	Returns the integral of a polynomial.	`polynomial`
`integral(polynomial poly)`	Returns the integral of a polynomial.	`polynomial`
`quadrature(real a, real b)`	Returns approximate integral of a function from `a` to `b` using Quadrature method.	`real`
`quadrature(String fcnSignature, real a, real b)`	Returns approximate integral of a function from `a` to `b` using Quadrature method.	`real`
`romberg(real a, real b)`	Returns approximate integral of a function from `a` to `b` using Romberg method.	`real`
`romberg(String fcnSignature, real a, real b)`	Returns approximate integral of a function from `a` to `b` using Romberg method.	`real`
`setFunction(String fcnSignature)`	Sets the integrand function.	`void`
`simpson(real a, real b)`	Returns approximate integral of a function from `a` to `b` using Simpson method.	`real`
`simpson(String fcnSignature, real a, real b)`	Returns approximate integral of a function from `a` to `b` using Simpson method.	`real`
`simpsonRichardson(real a, real b)`	Returns approximate integral of a function from `a` to `b` using Simpson-Richardson method.	`real`
`simpsonRichardson(String fcnSignature, real a, real b)`	Returns approximate integral of a function from `a` to `b` using Simpson-Richardson method.	`real`
`trapeze(double from, double to)`	Returns approximate integral of a function from `from` to `to` using Simpson-Richardson method.	`real`
`trapeze(String fcnSignature, double from, double to)`	Returns approximate integral of a function from `from` to `to` using Simpson-Richardson method.	`real`
`tricub(real xi, real yi, real xj, real yj, real xk, real yk, real acc)`	Returns approximate definite double integral of a function over the triangular domain with vertices (`xi,yi`), (`xj,yj`), and (`xk,yk`) using Tricube method.	`real`
`tricub(String fcnSignature, real xi, real yi, real xj, real yj, real xk, real yk, real acc)`	Returns approximate definite double integral of a function over the triangular domain with vertices (`xi,yi`), (`xj,yj`), and (`xk,yk`) using Tricube method.	`real`

The table below lists in alphabetical order the functions in the HYP_ODE class.

Call Signature	Description	Return

		Type
`setDiffFcn(String odeSignature)`	Sets up the differential equation function.	`void`
`computeDerivative()`	Computes derivative for ODE	`real`

The table below lists in alphabetical order the functions in the ODE Solver class.

Call Signature	Description	Return Type
`euler(String odeSignature, real start, real stop, real stepSize, realVector initVec)`	Returns solution of an ordinary differential equation.	`realMatrix`
`setStepSize(real stepSize)`	Sets the step size for differentiation or integration.	`void`

Appendix F -- Library: Estimation

The Estimation library can be used to compute interpolation, polynomial least square fit, and linear regression. The Estimation library contains four classes: All, Interpolator, PolynomialLeastSquare, and LinearRegression.

F.1Classes

The table below lists the libraries and their class paths.

Class Name	Class Path
All	**`library.estimation.All`**
Interpolator	**`library.estimation.Interpolator`**
Linear Regression	**`library.estimation.LinearRegression`**
Polynomial Regression	**`library.estimation.PolynomialRegression`**

F.2 Functions

The All class contains functions of the other three classes in the Estimation library. When more than one class have functions of the same name, the function names are modified in the All class. The table below list these name changes.

All	Polynomial Regression	Linear Regression
`getErrorMatrix()`	**`getPolyErrorMAtrix()`**	**`getErrorMatrix()`**

The table below lists in alphabetical order the functions in the Interpolator class.

Call Signature	Description	Return Type
`interpolate(real a)`	Returns interpolated value corresponding to **a**, for independent vector **x**, dependent vector **y**, and the interpolator already been set.	**`real`**
`lagrange(realVector x, realVector y, real a)`	Returns interpolated value corresponding to **a**, for independent vector **x** and dependent vector **y**, using Lagrange Interpolator.	**`real`**
`linear(realVector x, realVector y, real a)`	Returns interpolated value corresponding to **a**, for independent vector **x** and dependent vector **y**, using Linear Interpolator.	**`real`**
`linear(realVector x, realVector y, real a, integer index)`	Returns interpolated value corresponding to `a`, for independent vector **x** and dependent vector **y**, using Linear Interpolator and pre-computed index	**`real`**

	for the independent vector.	
`neville(realVector x, realVector y, real a)`	Returns interpolated value corresponding to **a**, for independent vector **x** and dependent vector **y**, using Neville Interpolator.	**real**
`newton(realVector x, realVector y, real number)`	Returns interpolated value corresponding to **a**, for independent vector **x** and dependent vector **y**, using Newton Interpolator.	**real**
`resetCoefficients()`	If the interpolator is set to Newton Interpolator, Resets coefficients.	**void**
`setLagrange(realVector x, realVector y)`	Sets to interpolator to the Lagrange Interpolator for vector **x** and vector **y**.	**void**
`setLinear(realVector x, realVector y)`	Sets to interpolator to the Linear Interpolator for vector **x** and vector **y**.	**void**
`setNeville(realVector x, realVector y)`	Sets to interpolator to the Neville Interpolator for vector **x** and vector **y**.	**void**
`setNewton(realVector x, realVector y)`	Sets to interpolator to the Newton Interpolator for vector **x** and vector **y**.	**void**
`setSpline(realVector x, realVector y)`	Sets to interpolator to the Spline Interpolator for vector **x** and vector **y**.	**void**
`spline(realVector x, realVector y, real a)`	Returns interpolated value corresponding to **a**, for independent vector **x** and dependent vector **y**, using Spline Interpolator.	**real**
`spline (realVector x, realMatrix y, real a)`	Returns 2-dimensional interpolated value corresponding to **a**, for independent vector **x** and dependent matrix **y**, using Spline Interpolator.	**real**
`valueAndError(real a)`	If the interpolator is set to Neville Interpolator, returns the interpolated value and error corresponding to **a**, for independent vector **x** and dependent vector **y** already been set.	**realVector**

The table below lists in alphabetical order the functions in the Linear Regression class.

Call Signature	Description	Return Type
`getCorrelationCoefficient()`	Returns correlation coefficient.	**real**
`getErrorMatrix()`	Returns error matrix.	**realMatrix**
`getIntercept()`	Returns intercept value.	**real**
`getPolynomial()`	Returns polynomial	**polynomial**
`getSlope()`	Returns slope value.	**real**
`linearRegression(realVector vecX, Real_vector vecY)`	Returns a polynomial corresponding to independent vector x and dependent vector y, estimated using Linear Regression.	**polynomial**

The table below lists in alphabetical order the functions in the Polynomial Regression class.

Call Signature	Description	Return Type
`getErrorMatrix()`	Returns error matrix.	**`realMatrix`**
`polynomialLSFit (realVector vecX, realVector vecY, integer n)`	Returns least square estimated polynomial.	**`polynomial`**
`polyError(real x)`	Return error value.	**`real`**

Appendix G -- Library: Stochastic

The Stochastic library can be used for probability and statistical computations. Stochastic library contains sixteen classes.

G.1 Classes

The table below lists the classes in the Stochastic library and their paths.

Class Name	Class Path
Histogram	`library.stochastic.Histogram`
BetaDistribution	`library.stochastic.BetaDistribution`
CauchyDistribution	`library.stochastic.CauchyDistribution`
ChiSquareDistribution	`library.stochastic.ChiSquareDistribution`
ExponentialDistribution	`library.stochastic.ExponentialDistribution`
FisherSnedecorDistribution	`library.stochastic.FisherSnedecorDistribution`
FisherTippettDistribution	`library.stochastic.FisherTippettDistribution`
GammaDistribution	`library.stochastic.GammaDistribution`
HistogrammedDistribution	`library.stochastic.HistogrammedDistribution`
LaplaceDistribution	`library.stochastic.LaplaceDistribution`
LogNormalDistribution	`library.stochastic.LogNormalDistribution`
NormalDistribution	`library.stochastic.NormalDistribution`
ProbabilityDistribution	`library.stochastic.ProbabilityDistribution`
StudentDistribution	`library.stochastic.StudentDistribution`
TriangularDistribution	`library.stochastic.TriangularDistribution`
UniformDistribution	`library.stochastic.UniformDistribution`
WeibullDistribution	`library.stochastic.WeibullDistribution`

G.2 Functions

The table below lists in alphabetical order the functions for the Histogram class.

Call Signature	Description	Return Type
`average()`		`real`
`average(realVector vec)`		`real`
`binContent(real x)`		`real`
`binIndex(real x)`		`integer`
`binParameters(real x)`		`realVector`
`binWidth()`		`real`

`count()`		`integer`
`countsBetween(real x, real y)`		`real`
`countsUpto(real x)`		`real`
`dimension()`		`real`
`errorOnAverage()`		`real`
`kurtosis()`		`real`
`maximum()`		`real`
`minimum()`		`real`
`overflow()`		`integer`
`processData(realVector vec)`		`void`
`range()`		`realVector`
`reset()`		`void`
`setGrowthAllowed()`		`void`
`setIntegerBinWidth()`		`void`
`size()`		`integer`
`skewness()`		`real`
`standardDeviation()`		`real`
`totalCount()`		`integer`
`underflow()`		`integer`
`variance()`		`real`
`xValueAt(integer index)`		`real`
`yValueAt(integer index)`		`real`

The table below lists in alphabetical order the functions common to all the probability distributions.

Call Signature	Description	Return Type
`approximateValueAndGradient( real x)`	Returns an approximation of the gradient.	`realVector`
`average()`	Returns the average of the distribution.	`real`
`distributionName()`	Returns the name of the distribution.	`String`
`distributionValue(real x)`	Returns the probability of finding a random variable smaller than or equal to **`x`**.	`real`
`distributionValue(real x1, real x2)`	Returns the probability of finding a random variable between **`x1`** and **`x2`**.	`real`
`eval(real x)`	Returns probability density function	`real`
`inverseDistributionValue (real x)`	Returns the value for which the distribution function is equal to **`x`**.	`real`
`kurtosis()`	Returns kurtosis of the distribution.	`real`
`parameters()`	Returns parameters for the selected distribution.	`realVector`
`random()`	Returns a random number according to the set	`real`

	distribution.	
`random(integer l)`	Returns real vector of length `l` whose elements are random numbers according to the set distribution.	`realVector`
`random(real l)`	Returns real vector of length `l` whose elements are random numbers according to the set distribution.	`realVector`
`random(integer m, integer n)`	Returns real matrix of size `mXn` whose elements are random numbers according to the set distribution.	`realMatrix`
`random(integer m, real n)`	Returns real matrix of size `mXn` whose elements are random numbers according to the set distribution.	`realMatrix`
`random(real m, integer n)`	Returns real matrix of size `mXn` whose elements are random numbers according to the set distribution.	`realMatrix`
`random(real m, real n)`	Returns real matrix of size `mXn` whose elements are random numbers according to the set distribution.	`realMatrix`
`setHistogram(Histogram histo)`	Sets the parameters of the distribution according from the histogram `histo`.	`void`
`setParameters(realVector params)`	Sets the parameters of the distribution according to `params`.	`void`
`setSeed(integer seed)`	Sets the seed of the random number generator.	`void`
`skewness()`	Returns skewness for the distribution.	`real`
`standardDeviation()`	Returns standards deviation of the distribution from variance.	`real`
`valueAndGradient(real x)`	Returns the value and the gradient of the distribution with respect to the parameters.	`realVector`
`variance()`	Returns the variance of the distribution.	`real`

The table below lists in alphabetical order the functions pertinent to the Cauchy distribution.

Call Signature	Description	Return Type
`setCenter(real center)`	Sets center value for Cauchy Distribution. Valid for Cauchy Distribution only.	`void`
`setWidth(real width)`	Sets beta value for Cauchy Distribution. Valid for Cauchy Distribution only.	`void`

The table below lists in alphabetical order the functions for the Chi-Squared distribution.

Call Signature	Description	Return Type
`confidenceLevel(real x)`	Set the confidence level to x. Only valid for Chi Square, Fisher Snedecor, and Student distributions.	`real`
`setDegreesOfFreedom(int n)`	Valid only for Chi Square Distribution.	`void`

The table below lists in alphabetical order the functions pertinent to the Fisher Snedecor distribution.

Call Signature	Description	Return Type
`defineParameters (integer n1, integer n2)`	Define parameters for FisherSnedecor distribution.	`void`
`confidenceLevel(real x)`	Set the confidence level to x. Only valid for Chi Square, Fisher Snedecor, and Student distributions.	`real`

The table below lists in alphabetical order the functions pertinent to the Fisher Tippett distribution.

Call Signature	Description	Return Type
`defineParameters(real center, real scale)`	Define parameters for Fisher Tippett distribution.	`void`

The table below lists in alphabetical order the functions pertinent to the Gamma distribution.

Call Signature	Description	Return Type
`defineParameters(real shape, real scale)`	Define parameters for Gamma distribution.	`void`

The table below lists in alphabetical order the functions pertinent to the Laplace distribution.

Call Signature	Description	Return Type
`defineParameters(real center, real scale)`	Define parameters for Laplace distribution.	`void`

The table below lists in alphabetical order the functions pertinent to the Normal distribution.

Call Signature	Description	Return Type
`errorFunction(real x)`	Returns error function for the Normal distribution.	`real`
`eval(real x)`	Returns probability density function	`real`
`evalNormal(real x)`	Returns the density function for a (0,1) Normal distribution evaluated at x.	`real`
`setAverage( real average)`	Set the average value for the Normal Distribution. Valid only for the Normal Distribution.	`void`
`setParameters(realVector params)`	Set parameters	`void`
`setStandardDeviation( real standardDeviation)`	Set the standard deviation value for the Normal Distribution. Valid only for the Normal Distribution.	`void`

The table below lists in alphabetical order the functions pertinent to the Student distribution.

Call Signature	Description	Return Type
`confidenceLevel(real x)`	Set the confidence level to x. Only valid for Chi Square, Fisher Snedecor, and Student distributions.	`real`
`defineParameters(integer n)`	Define parameters for Student distribution.	`void`
`eval(real x)`	Returns probability density function	`real`

The table below lists in alphabetical order the functions pertinent to the Uniform distribution.

Call Signature	Description	Return Type
`eval(real x)`	Returns probability density function	`real`
`setLimits(real low, real high)`	Sets the lower and upper limits for the Uniform distribution.	`void`

The table below lists in alphabetical order the functions pertinent to the Weibull distribution.

Call Signature	Description	Return Type
`defineParameters(real shape, real scale)`	Define parameters for Weibull distribution.	`void`

The Probability Distribution class combines all the separate distributions in one single class. Any of the distributions can be used from the Probability Distribution class. The Probability Distribution class contains all the methods listed in the previous tables plus some extra functions. The table below lists in alphabetical order the functions special to the Probability Distribution class.

Call Signature	Description	Return Type
`average()`	Returns average value	`real`
`beta(real shape1, real shape2)`	Returns a random number according to Beta Distribution with shape1 set to `shape1` and `shape2` set to `shape2`.	`real`
`cauchy(double location, double scale)`	Returns a random number according to Cauchy distribution.	`real`
`chiSquare(integer dof)`	Returns a random number according to Chi Square Distribution with degrees-of-freedom set to `dof`.	`real`
`confidenceLevel(real x)`	Return confidence level	`real`
`defineParameters(double shapeOrCenter, double scale)`	Defines parameters for Gamma, Fisher Tippett, or Laplace distribution.	`void`
`defineParameters(integer n)`	Defines parameter for Student distribution	`void`
`defineParameters(integer n1, integer n2)`	Defines parameter for Fisher Snedecor distribution	`void`
`distributionName()`	Returns name of the distribution	`string`
`distributionValue(real x)`	Returns the value of the distribution.	`real`
`distributionValue(real x1, real x2)`	Returns the difference between the values of the distribution due to `x1` and `x2`.	`real`
`eval(real x)`	Evaluates Uniform, Gamma, or Student distribution for the value of x	`real`
`exponential(real rate)`	Returns a random number according to Exponential Distribution with rate set `rate`.	`real`
`fisherSnedecor(integer dof1, integer dof2)`	Returns a random number according to Fisher Snedecor Distribution with the first degrees-of-freedom set to `dof1` and the second degrees-of-freedom set to `dof2`.	`real`

fisherTippett(real center, real scale)	Returns a random number according to Fisher Tippett Distribution with the center set to `center` and the scale set to `scale`.	**real**
gamma(real shape1, real scale)	Returns a random number according to Gamma Distribution with the shape1 set to `shape1` and the scale set to `scale`.	**real**
inverseDistributionValue(real x)	Returns the inverse value	**real**
kurtosis()	Return kurtosis.	**real**
laplace(real center, real scale)	Returns a random number according to Laplace Distribution with the center set to `center` and the scale set to `scale`.	**real**
logNormal()	Returns a random number according to Log Normal Distribution with mean set to 0.0 and standard deviation set to 1.0.	**real**
logNormal(real mean, real stdDev)	Returns a random number according to Log Normal Distribution with mean set to `mean` and standard deviation set to `stdDev`.	**real**
Normal()	Returns a random number according to Normal Distribution with mean set to 0.0 and standard deviation set to 1.0	**real**
normal(real mean, real stdDev)	Returns a random number according to Normal Distribution with mean set to `mean` and standard deviation set to `stdDev`.	**real**
parameters()	Returns the parameters of the distribution.	**realVector**
random()	Returns a random value for the distribution that has already been set.	**real**
setAverage(real average)	Sets average for ormal distribution.	**void**
setBeta(real shape1, real shape2)	Sets the distribution to Beta Distribution with shape1 set to `shape1` and `shape2` set to `shape2`.	**void**
setBeta(Histogram histo)	Sets the distribution to Beta Distribution from a histogram.	**void**
setCauchy(real center, real width)	Sets the distribution to Cauchy Distribution with center set to `center` and scale set to `scale`.	**void**
setCauchy(Histogram histo)	Sets the distribution to Cauchy Distribution from a histogram.	**void**
setCenter(real center)	Sets center for the Cauchy Distribution.	**void**
setChiSquare(Histogram histo)	Sets the distribution to Chi Square Distribution from a histogram.	**void**
setChiSquare(integer dof)	Sets the distribution to Chi Square Distribution with degrees-of-freedom set to `dof`.	**void**
setDegreesOfFreedom(integer n)	Sets degrees-of-freedom for Chi Square Distribution	**void**
setExponential(real rate)	Sets the distribution to Exponent Distribution	**void**

	with rate set to `rate`.	
setExponential(Histogram histo)	Sets the distribution to Exponent Distribution from a histogram.	**void**
setFisherSnedecor(Histogram histo)	Sets the distribution to Fisher Snedecor Distribution from a histogram.	**void**
setFisherSnedecor(integer dof1, integer dof2)	Sets the distribution to Fisher Snedecor Distribution with the first degrees-of-freedom set to `dof1` and the second degrees-of-freedom set to `dof2`.	**void**
setFisherTippett(real center, real scale)	Sets the distribution to Fisher Tippett Distribution with center set to `center` and scale set to `scale`.	**void**
setFisherTippett(Histogram histo)	Sets the distribution to Fisher Tippett Distribution from a histogram.	**void**
setGamma(real shape1, real scale)	Sets the distribution to Gamma Distribution with shape set to `shape1` and scale set to `scale`.	**void**
setGamma(Histogram histo)	Sets the distribution to Gamma Distribution from a histogram.	**void**
setHistogram(Histogram histo)	Sets histogram for the distribution	**void**
setLaplace(real center, real scale)	Sets the distribution to Laplace Distribution with center set to `center` and scale set to `scale`.	**void**
setLaplace(Histogram histo)	Sets the distribution to Laplace Distribution from a histogram.	**void**
setLogNormal()	Sets the distribution to Log Normal Distribution with mean set to 0.0 and standard deviation set to 1.0.	**void**
setLogNormal(real mean, real stdDev)	Sets the distribution to Log Normal Distribution with mean set to `mean` and standard deviation set to `stdDev`.	**void**
setLogNormal(real mean, real stdDev)	Sets the distribution to Log Normal Distribution from a histogram.	**void**
setNormal(Histogram histo)	Sets the distribution to Normal Distribution with mean set to 0.0 and standard deviation set to 1.0.	**void**
setNormal(real mean, real stdDev)	Sets the distribution to Normal Distribution with mean set to `mean` and standard deviation set to `stdDev`.	**void**
setNormal(Histogram histo)	Sets the distribution to Normal Distribution from a histogram.	**void**
setParameters(realVector params)	Sets parameters for the distribution	**void**
setSeed(integer seed)	Sets seed for the distribution.	**void**
setStandardDeviation(real standardDeviation)	Sets standard deviation for the Normal distribution	**void**
setStudent(Histogram histo)	Sets the distribution to Student Distribution	**void**

	from a histogram.	
setStudent(integer dof)	Sets the distribution to Student Distribution with degrees-of-freedom set to `dof`.	**void**
setTriangular(real low, real high, real peak)	Sets the distribution to Triangular Distribution with the low set to `low`, high set to `high` and the peak set to `peak`.	**void**
setTriangular(Histogram histo)	Sets the distribution to Triangular Distribution from a histogram.	**void**
setUniform()	Sets the distribution to Uniform Distribution. Generated random numbers will be between -1.0 and 1.0.	**void**
setUniform(real a, real b)	Sets the distribution to Uniform Distribution with lower and upper limits of the generated random number set to `a` and `b`.	**void**
setUniform(Histogram histo)	Sets the distribution to Uniform Distribution from a histogram.	**void**
setWeibull(real shape1, real scale)	Sets the distribution to Weibull Distribution with shape set to `shape1` and scale set to `scale`.	**void**
setWeibull(Histogram histo)	Sets the distribution to Weibull Distribution from a histogram.	**void**
setWidth(real width)	Set width to Cauchy Distribution	**void**
skewness()	Return skewness.	**real**
standardDeviation()	Return standard deviation,	**real**
student(int dof)	Returns a random number according to Student Distribution with degrees-of-freedom set to `dof`.	**real**
triangular(real low, real high, real peak)	Returns a random number according to Triangular Distribution with the low set to `low`, high set to `high` and the peak set to `peak`.	**real**
uniform()	Returns a random number according to Uniform Distribution with the range set to between -1.0 and 1.0.	**real**
uniform(real a, real b)	Returns a random number according to Uniform Distribution with the with range set to between `a` and `b`.	**real**
valueAndGradient(double x)	Returns value and gradient from the distribution..	**realVector**
variance()	Returns variance from the distribution.	**real**
weibull(real shape1, real scale)	Returns a random number according to Weibull Distribution with the shape1 set to `shape1` and the scale set to `scale`.	**real**

Appendix H -- Library: Frequency Domain

The Frequency Domain library contains one class, FFT.

H.1 Classes

The table below lists the classes and their paths.

Class Name	Class Path
FFT	**`library.frequency_domain.FFT`**

H.2 Functions

The table below lists in alphabetical order the functions in the FFT class.

Call Signature	Description	Return Type
`fft(realVector real)`	Returns a complex vector whose elements have been Fourier transformed from a real vector. The length of the output vector is the same as the length of the input vector.	**`complexVector`**
`fft(table real)`	Returns a complex vector whose elements have been Fourier transformed from a real vector. The length of the output vector is the same as the length of the input vector.	**`complexVector`**
`fft(realVector real, realVector imag)`	Returns a complex vector whose elements have been transformed from a complex vector whose real and imaginary parts are given in **`real`** and **`imag`**. The length of the output vector is the same as the length of the input vector.	**`complexVector`**
`fft(complexVector vec)`	Returns a complex vector whose elements have been transformed from a complex vector. The length of the output vector is the same as the length of the input vector.	**`complexVector`**
`fft (table real, table imag)`	Returns a complex vector whose elements have been transformed from a complex vector whose real and imaginary parts are given in **`real`** and **`imag`**. The length of the output vector is the same as the length of the input vector.	**`complexVector`**
`fft(realVector real, integer n)`	Returns a complex vector whose elements have been Fourier transformed for n data points from a real vector. The length of the output vector is **`n`**.	**`complexVector`**
`fft(table real, integer n)`	Returns a complex vector whose elements have been Fourier transformed for n data points from a real vector. The length of the output vector is **`n`**.	**`complexVector`**
`fft(realVector real,`	Returns a complex vector whose elements have	**`complexVector`**

realVector imag, integer n)	been Fourier transformed for n data points from a complex vector whose real and imaginary parts are given in **real** and **imag**. The length of the output vector is **n**.	
fft(complexVector vec, integer n)	Returns a complex vector whose elements have been Fourier transformed for n data points from a complex vector. The length of the output vector is **n**.	**complexVector**
fft (table real, table imag, integer n)	Returns a complex vector whose elements have been Fourier transformed for n data points from a complex vector whose real and imaginary parts are given in **real** and **imag**. The length of the output vector is **n**.	**complexVector**