# Ordinary Differential Equation Solver Language (ODESL) Reference Manual

*Rui Chen*

*11/03/2010*

## 1. Introduction

ODESL is a computer language specifically designed to solve ordinary differential equations (ODE's) numerically for initial value problems (IVP's). It provides a subset of MATLAB's capability for manipulating numbers, mathematical expressions, and matrices, and numerically solving IVP's. The syntax of ODESL is also similar to that of MATLAB. A program written in ODESL is first translated to a JAVA program and then compiled to run on Java Virtual Machine (JVM).

## 2. Lexical Conventions

There are six kinds of tokens: identifiers, keywords, numbers, strings, expression operators, and other special symbols. White spaces, tabs, newlines and comments are used to separate tokens, and they are ignored in other situations. At least one white space character is required to separate adjacent identifiers and constants.

### 2.1. Comments

The start of a comment is marked by the character "%" The end of a comment is marked by a newline character. Anything between the "%" and newline characters will be ignored by the compiler.

### 2.2. Identifiers

An identifier is a sequence characters consists of alphabets and digits. The first character must be an alphabet. The underscore "_" character counts as an alphabet it cannot be used as the first character in an identifier. Upper and lower case alphabets are considered different. There is no limit to the length of identifiers.

### 2.3. Reserved Keywords

The following is a list of reserved keywords that cannot be used for other purposes:

| | |
|---|---|
| break | for |
| continue | function |
| elseif | If |
| end | true |
| false | while |

### 2.4. Numbers

A number consists of a sequence of digits, followed by an optional decimal point ".", followed by an optional "e", followed by optionally signed integer. All numbers except those used as indices for "for" loops and matrices are assumed to be double precision integers. Indices are assumed to be integers.

### 2.5. Strings

A string is a sequence of surrounded by double quotes "".

### 2.6. Special Symbols

The following special symbols are used in the language for various purposes such as expression mathematical and logical operations, indicating comments and creating matrices.

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| + | - | * | / | ^ | = | == | < | > | <= | >= | == |
| != | \| | & | ! | ( | ) | [ | ] | , | : | % | ; |

## 3. Types

The types in this ODESL are not explicitly specified in the program. Because an ODESL program will be translated into a JAVA program, the data type of a variable is determined at compile time. The ODESL compiler determines the data type of a variable or a value returned by a function using the following rules: if a variable is assigned a value of "true" or "false", then that variable is a boolean. All numbers are assumed to be double precision floating points except those without a decimal point and "e". Numbers without a decimal point and "e" are assumed to be 32-bit integers. ODESL also has a build-in data type *matrix* which is an m by n matrix. A variable is of type *matrix* if its value is assigned by using a list of numbers enclosed in "[ ]". A user-defined function specified by a *function* keyword is also a build-in data type in ODESL.

## 4. Expressions

The following sections list the expressions and expression operators used in ODESL. The expression operators are listed in order of precedence (highest precedence first). For example, the expression operator in section 4.3 has higher precedence than that in section 4.4. Within each subsection, the operators have the same precedence. Left- or right-associativity of the operators are also specified.

## 4.1. Primary Expressions

### 4.1.1. *Identifier*

An identifier is bounded to an assigned value. The data type of the identifier depends on the value of the right-hand side of the assignment operator.

### 4.1.2. *Numbers*

A number is a constant and is evaluated to itself.

### 4.1.3. *Strings*

A sequence of characters enclosed in double quotes.

### 4.1.4. ( expression )

A parenthesized expression evaluates to the same value as the expression enclosed by the parenthesis.

### 4.1.5. *Access to elements of matrices*

This primary expression consists of a matrix identifier followed by either one index or two indices separated by comma enclosed by parentheses “( )”. Access to elements in a matrix can also be done by enclosing one or two range specifiers (Section 4.1.6) separated by comma in parenthesis.

### 4.1.6. *Access to elements of matrices*

*integer1:step:integer2* is a range specifier denoting integer ranging from *integer1* to *integer2* with a interval equals to *step*. Matrix access can also be used create matrices with maximum size indicated by the indices enclosed in the parentheses.

### 4.1.7. *1D or 2D matrix creation*

A 1-dimensional matrix can be created by enclosing its elements by “[ ]” with the elements separated by “,”. A 2-dimensional matrix can be created by enclosing its elements by “[ ]” with “,” separating elements in the same row and “;” separating elements in the same column.

### 4.1.8. *Function calls*

A function call is a primary expression followed by parentheses enclosing a list of argument separated by commas. However, the number of arguments can be zero. Each argument is an expression.

## 4.2. Mathematical Expressions

Mathematical expressions take primary expressions as operands.

4.2.1. Unary Operator*: + expression*
+ *expression* evaluates to the expression itself
4.2.2. Unary Operation: - *expression*
- *expression* evaluates to the negative value of the expression

4.2.3. Power operator: “^”
*x ^ y: x* to the power of y. x can be a scaler or a matrix. y must be a scaler. If x is a matrix, then each element of the matrix will be raised to the power of y.
The associativity of the power operators is to the right.

4.2.4. Multiplicative operators: “*” and “/”
The following binary operations are possible:
m*n: where m and n are matrices, and the size of m and n are correct for matrix multiplication. This evaluates to the matrix multiplication of m and n.
x*y: where x and y are scalers. This evaluates to x times y.
m*y: where m is a matrix and y is a scaler. This results in a matrix where every element of the matrix is multiplied by y.
x/y: where x and y are scalers. This evaluates to x divided by y.
m/y: where m is a matrix and y is scaler. This results in a matrix hwere every element of the matrix is divided by y.

4.2.5. Additive operators “+” and “-“
Binary operators “+”, “-“ denote addition and subtraction. They can be used on two scalers, two matrices of the same size, or a scaler and a matrix. If both sides of the operators are not scaler or matrix, the scaler will be added to or subtracted from every element of the matrix. These operators group left to right.

## 4.3. Relational Operators

The relational operators listed below group left-to-right. All relational operators yield true if the relation is true, and false if the relation is false;

**4.3.1.** *expression < expression (less than)*
**4.3.2.** *expression>expression (greater than)*
**4.3.3.** *expression<=expression (less than or equal to)*
**4.3.4.** *expression>=expression (greater than or equal to)*

## 4.4. Equality Operators

The equality operators behave the same as relational operators. But they have lower precedence.

**4.4.1.** *expression == expression (equal to)*
**4.4.2.** *expression != expression (not equal to)*

### 4.5. Logical *not* operator: ! *expression*

The ! operator takes relational expressions as operands and it groups left-to-right. The result is the logical negation of the operands

### *4.6.* Logical *and* operator: *expression & expression*

The & operator takes two relational expressions as operands and it groups left-to-right. The result is the logical and of the two operands. This operator is short-circuited.

### *4.7.* Logical *or* operator: *expression* | *expression*

The **|** operators takes two relational expressions and operands and it groups left-to-right. The result is the logical or of the two operands. The operator is short-circuited.

## *5.* Statements

If not otherwise indicated, statements are executed in sequence.

### *5.1.* Expression statement

Most statements are expression statements which are expressions that are terminated by a semicolon ";".

### *5.2.* Statement list

A statement list is a list of one or more statements separated by ";"

### *5.3.* Assignment statement

The format of the assignment statement is: *Identifier = expression.* The following are examples of assignment statements:

```
a=1;
a=1.2e3;
a=[1,1,1;2,2,2;];
```

### 5.4. Conditional statement

The possible formats of the conditional statements are listed in the following sections.

**5.4.1.** ***If-else***

```
if ( logical expression)
        statement list
else
        statement list (optional)
```

```
end
```

if the logical expression after “if” is true, then the first statement list is executed. Otherwise, the second optional statement list under “else” is executed.

### 5.4.2. *If-elseif-else*

```
if (logical expression)
        statement list
elseif (logical expression)
        statement list
        .
        .
else
        statement list (optional)
end
```

if the logical expression after “if” is true, then the first statement list is executed. Following the “if” statement, there can be any number of “elseif” statements. If the logical expression in the “if” statement is false, then the statement list inside the first logical expression after “elseif” that evaluate to true will be executed. If the logical expressions after both “if” and “elseif” statements return false, then the optional statement list under “else” will be executed.

## *5.5. for* statement

This statement is an iterative statement in the following format:

```
for ( identifier = range)
    statement list
end
```

The number of iterations is determined by the range specifier discussed in Section 4.1.6. As long as the value of the identifier is less than or equals to the highest value in the range, the statements in the statement list will be excuted.

## *5.6. while* statement

This statement is an iterative statement in the following format:

```
while (logical expression)
    statement list
end
```

The statement list is executed repeatedly as long as the value of the logical expression remains true. The test takes place before each execution of the statement list.

## *5.7. break* statement

The statement

*break*

causes the termination of the innermost *for* or *while* iterative statement. Control is passed to the statement following the terminated statement.

### ***5.8. continue*** **statement**

The statement:

*continue*

causes the current iteration of the innermost iterative statement to terminate and proceed to the next iteration.

## *6.* Function definition

A function is defined using the following format:

*function [var1, var2, …] = functionName (argument1, argument2, …)*

*statement list*

*end*

A function is created using the *function* keyword followed by a optional list of return variables separated by commas and enclosed in "[ ]", followed by "=", followed by the name of the function, and then a list of arguments separated by commas and enclosed in parentheses. The data type of all the return variables must the same. The list of return variables is essentially a 1D matrix.

## *7.* Build-in functions

### ***7.1.*** *disp(expression)*

This function displays the value of the *expression* on screen.

### ***7.2.*** *save(fileName, expression)*

This function saves the value of the expression to an ASCII file with name *filename*. If the expression evaluates to a single number, the number is stored in the file. If the expression evaluates to a matrix, then the file will contain the matrix with each elements separated by spaces.

### ***7.3.*** *load (fileName)*

This function loads a value of matrix stored in a file with name *fileName*.

### ***7.4.*** *Matrix functions*

#### **7.4.1.** *width(matrix)*

This function returns the width of a matrix.

#### **7.4.2.** *height(matrix)*

This function returns the height of a matrix.

#### **7.4.3.** *zero(m, n)*

This function creates a m by n matrix whose elements all have values equal to zero.

### 7.5. Mathematical functions

The following lists the build-in mathematical functions of ODESL:

| | |
|---|---|
| abs(x) | absolute value of x |
| exp(x) | $e^x$ |
| log(x) | natural log of x |
| sin(x) | sine of x |
| cos(x) | cosine of x |
| tan(x) | tangent of x |
| asin(x) | arc sine of x |
| acos(x) | arce cosine of x |
| atan(x) | arc tangent of |

x can be either double or matrix. y can only be a double or integer. If x is a matrix, the math operation will be applied to each element of the marix.

## 8. Scope rules

ODESL uses static scoping. There are two types of variable scopes: global and local. A global variable is a variable that is assigned a value outside of a function. Global variables can be used anywhere inside a program except when there is a local variable of the same name in a function. Global variables exist until the termination of the program. A local variable is a variable that is assigned a value inside a function. Local variables can only be used inside the function in which it was first assigned a value, and it cannot be used outside of that function. If there is a global variable and a local variable with the same name inside a function, the value of the local variable will be used instead of the global variable.