Converting MATLAB to Idea

2 October 2013

Address:	Gedae, Inc. 1247 N. Church Street, Suite 5 Moorestown, NJ 08057
Telephone:	(856) 231-4458
FAX:	(856) 231-1403
Internet:	www.gedae.com

Contents

Introduction

This document describes how to convert functions and programs written in MATLAB into the Idea language.

Range Variables

The most obvious difference between MATLAB expressions and Idea expressions is the heavy use of range variable in Idea expressions that involve vectors, matrices or higher dimensionality variables. For example to add two matrices in MATLAB and then select a subregion of the matrix we write:

```
C = A + B;
D = C(:,2:5);
```

While to do the same in Idea we write

```
C[r][c] = A[r][c] + B[r][c];
range n = 4;
D[r][n] = C[r][n+1];
```

Where r and c are previously defined range **variables and we define n to select a subrange of** columns of C**.** The following sections describe how to use Idea range variables and translate different MATLAB expressions into Idea expressions using them.

MATLAB vs. Idea Ranges as used to Index Values

Both Gedae and MATLAB use ranges of values to define the size of a vector or matrix and to index the vector or matrix. A range of integer values in MATLAB can be defined as either `(start:stop)` or `(start:step:stop)`. For example the range `(1:5)` is the vector 1 2 3 4 5. `(1:2:7)` is the vector 1 3 5 7.

In Gedae a range can be defined as:

```
range n = N;
```

The range is not a vector but is considered to take on all the values 0,1,…N-1. Idea ranges are thus somewhat simpler than MATLAB ranges in that they always start at 0 and always step by 1. To make an Idea range work like the most general MATLAB range the expression

```
step*n+start
```

can be used to generate the same values as the MATLAB range

```
(start:step:stop)
```

Idea Dimensions are 0 Based.

Because Idea ranges are often used to index matrices and vectors they begin with a value of 0. This is because Idea dimensions are 0 based as opposed to MATLAB dimensions which are 1 based. So for example the MATLAB expression

```
A = B(3: 6);
```

Which sets `A` to be a 4 element vector with values `B(3),B(4),B(5),B(6)`. This expression for `A` in MATLAB is equivalent to the Idea expression.

```
range n = 4  // takes on the values 0,1,2,3
A[n] = B[n + 2];
```

Which sets A to a 4 element vector with values `B[2],B[3],B[4],B[5]`. Note that since Idea uses 0 based dimensions `B[2]` is the third value of vector `B`.

Defining Range Variables

The most direct way to define a range is to declare it as:

```
range n = <some integer expression>
```

A simple example would be

```
range n = 16;
```

A second way a range can be defined is in a function input argument: For example we can write a matrix multiplication function in Idea as:

```
out mmult(float A[i][j],float B[j][k]) {
  out[i][k] += A[i][j]*B[j][k];
}
```

In the above example the ranges i, j, and k are brought in as part of the function argument list.
A third way that ranges can be defined is as the output of function call. For example, the real discrete Fourier transform produces an output vector half the size of the input vector.
given we have already defined:

```
float x[n] = …
```

Then

```
y[m] = rdft(x);
```

And `m` is a new range variable with a size half that of n.
Note that it is not required that the range variable m be previously undefined. But if it is defined elsewhere (explicitly, as a function argument range or as the output from another function) then the

values of each must all be provably equal. This equality of multiple range definitions is enforced by the Idea compiler.

Getting the size of a range

To get the size of a range the unary `#` operator is used. So for example if we have:

```
range n = 16;
```

then `#n` has the value `16`.
In MATLAB to get the dimensions of a matrix `A` the size function is used. So to get the number of rows and columns of A in MATLAB we use:

```
R = size(A,1);
C = size(A,2);
```

To do the same thing in Idea the values of the ranges in the matrix declaration are used. So if a matrix A is defined in Idea as:

```
A[r][c] = func(…); // func is any function defined by a .k or .i file
```

then the number of rows and columns are:

```
R = #r;
C = #c;
```

Algebraic and Functional Expressions

Idea has two basic types of expression which we call algebraic and functional. An algebraic expression is one whose right-hand-side (rhs) contains any number of unary, binary or collapsing operators and whose functions are all valid scalar functions. For example:

```
out[i] = abs(a[i] + b[i])/#i;
```

is an algebraic expression because the function abs is a scalar function. A functional expression has a rhs that consists of a single function call and its arguments can be the name of vectors or matrices or any scalar expression. For example if we have a variable `x[n]` defined then:

```
y[m] = rdft(x);
```

is a functional expression because the rdft function does not have all scalar arguments. If the functional expression is a multiple output function then the multiple outputs on the lhs are given in a comma separated list. For example the Idea meshgrid function that is equivalent to the MATLAB meshgrid function is:

```
rows[r] = …
cols[c] = …
mrows[t],mcols[t] = meshgrid(rows,cols);
```

Algebraic expressions require all non-scalar terms to have a full set of dimension indices where the indices can be range variables or integer values. Functional expressions require all non-scalar arguments to be passed as the name of the nonscalar term without dimensions. Scalar arguments to functional expressions can still be algebraic expressions. For example the following are incorrect:

```
y[m] = rdft(x[n]);
y[m]= rdft(-x);
y[m] = rdft(x+2);
```

because x is not passed as just the name of the vector variable x. But the find function that takes a vector as its first argument and the max number of values to find as its second argument can be called as:

```
x[n] = …
y[m] = find(x,#n/2-1);
```

Here the scalar second argument can be an arbitrary algebraic expression.

Algebraic expressions require all range variables used on the lhs and rhs of the expression to be previously defined. The range variables used on the lhs of the declaration define the size of the data being produced. If the expression uses the = operator (noncollapsing) then any range variables used on rhs of the expression must have been part of the definition of the lhs of the expression. Functional expressions by contrast do not require the dimensions used on the lhs of the expression to be previously defined but instead define the expression itself defines the dimensions. However, if the dimensions are previously defined the Idea compiler will insure that the definitions are consistent.
The limitation on functional expressions means that some MATLAB expressions that can be written as one line must be written as multiple Idea lines. For example the MATLAB program:

```
A(1:32) = 1.0;
B(1:32) = 4.0;
C = fft(A+B);
```

Is written in Idea as:

```
range n = 32;
A[n] = 1.0;
B[n] = 4.0;
AB[n] = A[n]+B[n];// algebraic expression
C[n] = dft(AB);   // functional expression
                  // dft is the Idea equivalent of fft
```

Interpreting Algebraic Expressions Containing Ranges

An algebraic expression that uses ranges is evaluated for all possible values of the range. For example:

```
range r = 3;
range c = 2;
```

```
A[r][c] = (r+1)*c;
```

then A[r][c] is a 3x2 matrix and is evaluated for the values r = 0..2 and c = 0..1. This definition of A is equivalent to the MATLAB expression:

```
A = (1:3)'*(0:1);
```

Matrix Element Selection

In MATLAB to select subcomponents of a matrix the following notation works:

```
A = B(2:5,3:8);
```

to do the same thing in Idea

```
range r = 4;
range c = 6;
A[r][c] = B[r+1][c+2];
```

Setting a subset of elements of a matrix

In MATLAB to set a subset of elements in a matrix `A` that was previously defined to the values of a matrix `B`:

```
A(2:5,3:8) = B
```

In Idea each variable must have a unique name. As a result a matrix such as `A` cannot be modified as above but instead we create a new matrix `A1` that is the modified version of `A`. So to implement the above expression in Idea if `A` is defined as `A[n][m] = ..` then the user can set `A1` as:

```
A1[n][m] = set(A,B,2,3);
```

Where `A1` is the same as `A` except for the values of `A` that are: `A[r+2][c+3]= B[r][c];`
The following table shows a complete list describing how to set subsets of elements in a matrix `A` or a vector `v`.

MATLAB	Idea
`A(r0:r1,c0:c1) = B;`	`A1[n][m] = set(A,B,r0,c0);`
`A(r0,c0) = 2;`	`A1[n][m] = set(A,2,r0,c0);`
`A(r0,:) = v`	`A1[n][m] = rset(A,v,r0);`
`A(:,c0) = v;`	`A1[n][m] = cset(A,v,c0);`
`A(r0,c0:c1) = v;`	`A1[n][m] = rsetsub(A,v,r0,c0)`
`A(r0:r1,c0) = v;`	`A1[n][m] = csetsub(A,v,r0,c0)`
`v(n0) = 2;`	`v1[n] = set(v,2,n0);`
`v(n0:n1) = v2`	`v1[n] = set(v,v2,n0);`
`v = vB(vC);`	`v[n] = vB[vC[n]];`
`v(vB) = vC;`	`v1[n] = scatter(v,vB,vC);`

Idea Matrices are Stored in Row Major Order

MATLAB matrices are stored in column major order while Idea matrices are stored in row major order. This distinction becomes important when traversing a MATLAB matrix by a single index. So for example the MATLAB Matrix

```
A = (1:3)'*(2:4)

A =

    2    3    4
    4    6    8
    6    9   12
```

Then

```
B = A(1:9)

B =

    2    4    6    3    6    9    4    8   12
```

The following is the equivalent code in Idea:

```
range r = 3;
range c = 3;
range n = #r*#c;
A[r][c] = (r+1)*(c+2)
B[n] = A[n%#r][n/#r];
```

The conversion of A to the vector B basically is a matrix transpose of the ordering. This can be seen in the following equivalent Idea translations:

```
C[c][r] = A[r][c]; // matrix transpose
v[r](c) = C[c][r]  // reinterpretation of matrix as #c vector tokens
s(r) =v[r];        // reinterpretation of vector as #r scalar tokens
B[n] = s(n);       // reinterpretation of #n scalar tokens as a vector
```

or

```
C[c][r] = A[r][c];  // matrix transpose
B[n] = m_v1(C);  // using the built in zero copy m_v1 function.
```

Once the matrix is transposed no additional computation is needed by either of the above two approaches of converting the matrix elements to a vector.

Operators

Idea operators extend C operators

Idea operators are identical to C operators with the extension of the unary size operator # that takes the size of a range and the power operator `. For example:

```
range n = 4;
y = 3;
z = y`#n;
```

and z is 3 raised to the 4^{th} power.

Idea collapsing operators

Idea provides a set of collapsing operators. The most common is the += operator. For example

```
range i = 6;
S += v[i];
```

Then

```
S = v[0]+v[1]+v[2]+v[3]+v[4]+v[5];
```

In general for operator <op>

```
S <op>= v[i];
```

is the same as:

```
S = v[0] <op> v[1] <op> v[2] <op> v[3] <op> v[4] <op> v[5];
```

The collapsing operator collapses any indices on the rhs of an expression not seen on the lhs of the expression. For example matrix multiplication can be expressed as:

```
a[i][j] = …
b[j][k] = …
c[i][k] += a[i][j]*b[j][k];
```

Family and timeindices can also be used in collapsing operators. For example an FIR filter can be expressed as:

```
y += C[i]*x(-i);
```

Summing across multiple families(which could be mapped to multiple processors) can be expressed as:

```
y[i] += [f]x[i];
```

Some other simple examples are:

```
sum += m[i][j]; // matrix sum
dot += x[i] * y[i]; // dot product
```

```
prod += m[i][j] * n[k][j]; // matrix multiply by transpose
dil[i][j] ||= im[i+i1][j+j1]; // bitmap image dilation kernel
erd[i][j] &&= im[i+i1][j+j1]; // bitmap image erosion kernel
```

The following collapsing operators are provided, illustrated with a scalar output and a vector input:

Collapsing operator	Description
`y += x[i]`	sum of elements
`y *= x[i]`	product of elements
`y |= x[i]`	bitwise or of elements
`y &= x[i]`	bitwise and of elements
`y ||= x[i]`	logical or of elements
`y &&= x[i]`	logical and of elements
`y >?= x[i]`	maximum of all elements
`y <?= x[i]`	minimum of all elements

Translating MATLAB operators into Idea operators

MATLAB	Idea	Idea Functional Expression
`C = A*B`	`C[i][j] += A[i][k]*B[k][j]`	`C[i][j] = mmult(A,B)`
`C = A+B`	`C[i][j]= A[i][j]+B[i][j]`	
`C = A-B`	`C[i][j]= A[i][j]-B[i][j]`	
`C = A.*B`	`C[i][j] = A[i][j]*B[i][j]`	
`C = A./B`	`C[i][j] = A[i][j]/B[i][j]`	
`C = A.\B`	`C[i][j] = B[i][j]/A[i][j];`	
`C = A.^2`	`C[i][j] = A[i][j]`2`	`C[i][j]= pow(A[i][j],2)`
`C = A.'`	`C[i][j] = A[j][i]`	
`C = A'`	`C[i][j] = conj(A[j][i])`	
`C = xor(A,B)`	`C[i][j] = A[i][j]^B[i][j]`	
`v2 = v/A`		`v2[i] = solve(A,v)`
`v2= A\v`		`v2[i] = solve(A,v)` no distinction in Idea between row and column vectors
`C = B/A`		`C[i][j] = solve(A,B)`
`C = A^-1`		`C[i][j] = inv(A)`
`C = any(A)`	`C[j] ||= A[i][j]`	
`C = all(A)`	`C[j] &&= A[i][j]`	

Logical Indexing

In MATLAB we can gather elements of a vector or matrix using logical indexing. For example

```
A = [(0:5),(0:4)]
A =
   0   1   2   3   4   5   0   1   2   3   4

B = A(A>3)
B =
   4   5   4
```

The above can be done using a find/gather in Idea as:

```
range n = 11;
```

```
A[n] = n>5 ? n-5 : n;
C[n] = A[n] > 3;
Indx[m] = find(C);
B[m] = A[Indx[m]];
```

Simarly we can do a find/gather on a matrix as:

```
range r = R;
range c = C;
A[r][c] = …
C[r][c] = A[r][c] > 3;
Rindx[m],Cindx[m] = find(C);
B[m] = A[Rindx[m],Cindx[m]);
```

Declaring Variables

A typical variable declaration in Idea is:

```
float A[r][b] = <valid idea expression>
```

When a variable is declared its dimensions, if any, must always be declared as simple range variables. In this case A is declared to be a matrix with range variables r and c. The following however is currently illegal

```
A[r+3][c*2] = <valid expression>
```

as all lhs indices must be simple. Family and time indices must also be simple. So the following are legal:

```
range n = 16;
range f = 4;
[f]A[n] = B[f+1];
i(n) = n*2;
```

The type of the declaration is optional when it can be unambiguously derived from the rhs of the expression and when no explicit type conversion is desired.

A main difference between MATLAB and Idea is that a variable name can only be used once on the lhs of an expression. So for example the MATLAB expressions:

```
A(1:4) = (1:4);
A(2) = 8;
```

Must be expressed in Idea as:

```
range r = 4;
A1[r] = r+1;
A[r] = set(A1,8,2);
```

Loops

The below illustrates a typical for loop in MATLAB that builds up an array element by element. We introduce a function `func` that operates on a matrix `B`, a vector `V`, and the index `i` as a means of creating an algorithm that is most conveniently defined using a for loop.

```
A = zeros(N);
V = a vector of size N
B = a matrix
for i = 1:N
 ai = func(B,V(i),i);
 A[i] = ai;
end
```

Translating MATLAB Loops into Idea Loop

The corresponding Gedae code to implement the above for loop is:

```
B[r][c] = a matrix;
V[n] = a vector of size N;
for (i=0, float A[n] = create(N); i<N; i=i+1) {
  ai = func(B,V[i],i);
  A[n] = set(A,ai,i);
}
```

A few comments:

- the for loop syntax is the same as in C.
- In C the update of i would typically be done as `i++`. Since the `++` operator is not currently supported in Idea it must be done as `i =i+1;`
- The `create(N)` function creates an uninitialized array `A` and a range variable n of size `N`.
- The `create(N)` function can be used to create a vector of any datatype. Therefore, the type of `A` in the `create(N)` function must be declared.
- The set function which was already covered sets the ith value to `i+2` – we set it to `i+2` instead of the MATLAB `i+1` because idea has 0 based array indexing while MATLAB has 1 base.
- The variable name `A` appears both in the for loop initializer and in the body of the for loop. This is an exception to the rule that all lhs variable names must be unique in Idea
- Variables initialized in the for loop initialization section, such as `A`, can appear in both the rhs and lhs of the set function call.

The Dataflow (or Streaming) Concept

In Gedae dataflow is a proper feature of the language. (Note: that MATLAB has step functions and uses a while loop to implement streaming. We make no attempt to define the equivalence between MATLAB and Gedae streams.) Streams are a feature of a data item. If a data item is a stream then the compiler recognizes that there will be multiple tokens driven by some external source and that the compiler has to implement software to process all tokens from the source. For example, if the function foo() reads data from an external interface then the expression x = foo(); will execute whenever data is available on

the interface until the program is terminated. There is no need to implement software to process the stream. The compiler understands the stream concept and implements the necessary software.

While the possible sources of a stream of tokens is quite broad, most sources are either driven by an external interface (e.g. AD/DA converter or message interface) or by decomposition of a token into a stream of subtokens, followed by processing the subtoken and then a recomposition into a full token. One of the most common examples of internal streams is decomposition of a matrix into a stream of vectors where each vector is a row of data. The vector is processed in some way, for example, a filter, and the results are then reassembled into a matrix. There are many other examples, such as finding ROIs in an image and then populating a database of tracks based on the ROIs. The point of this latter example is to emphasize that the streaming of data can be quite dynamic.

Translating MATLAB Loops into Idea Dataflow

The same code that could be implemented using a for loop in MATLAB and as a for loop in Idea can also be written using the Idea dataflow notation as:

```
B[r][c] = some matrix;
V[n] = a vector of size N;
Bn[r][c](n) = B[r][c];
i(n) = n;
v(n) = V[n];
ai= func(Bn,v,i);
A[n] = ai(n);
```

A few comments:

- `Bn` is a stream of `N` tokens that repeats the value of token `B`. We say we are holding the value of `B` `N` times.
- `v(n) = V[n]` converts the `N` values of 1 vector token of size `N` into `N` scalar tokens.
- `i` is a stream of `N` integer tokens taking on the values `0..N-1`
- `func` is a user defined function that is assumed to produce one token out for one token in on each argument.
- Notice arguments `Bn`, `v` and `i` to `func` are all at a rate `N` times higher than `B` or `V`.
- `ai = func(B,i)` would not give the same results because the dataflow between `B` and `I` are not balanced. The expression would get a new matrix `B` for each of the `N` values of `i`.
- `A[n] = ai(n)` collects groups of `#n = N` tokens of `ai` into a vector `A` of size `N`.

Typically when constructing a matrix or vector an index at a time the Idea dataflow notation is often simpler and easier to parallelize though either method will work.

Do/while and while loops

Idea also supports the do/while and while syntax supported in C though it has some differences having to do with initialization of loop variables. The do/while syntax to implement the for loop is

```
do (i=0, float A[n] = create(N)) {
```

```
  ai = func(B,V[i],i);
  A[n] = set(A,ai,i);
  i = i+1;
} while (i<N);
```

What is different is the comma-separated initializer list. Similarly a while implementation of the loop is:

```
while (i=0, float A[n] = create(N); i<N) {
  ai = func(B,V[i],i);
  A[n] = set(A,ai,i);
  i = i+1;
}
```

Choosing between for loop and dataflow implementations of loops

The question arises when should a MATLAB loop be implemented using Idea data flow and when as an Idea loop.

A situation like

```
x[c](r) = A[r][c]  // produce r tokens for each matrix A containing the rows of A.
y[c] = fft(x);
z[c] = y[c]*Coeff[c];
w[c] = ifft(z);
B[r][c] = w[c](r);   // collect r tokens of w into a new matrix B.
```

could be done with a loop but is better done as above. If done as a loop we must run the fft and a granularity of 1 but here we can run the fft - under user compiler option control - at either a granularity of #r, 1 or something in between.

But when you have many inputs - some of them being replicated - looping can be a great simplification on the dataflow version. For example suppose variable `a1...an` are defined outside the loop then.

```
a1 = ..
...
an = ..
for (x = 0, i=0; i<N; i = i+1) {
   x = func(x,a1,...an)
}
```

would require if done in dataflow a new set of variable say `b1..bn` defined as

```
range i = N;
a1 = ..
...
an = .
b1(i) = a1;
...
bn(i) = an;
x = func(x(-1),b1,...bn);
```

The point is that the replication is done automatically for you when you use an Idea loop. It is often not obvious in a large example what tokens must be replicated as the tokens that need to be replicated may be spread over multiple lines in the body of the loop.

Also our example for loop implements a simple counting loop. However loops in Idea can have an indefinite length. For example autofocus algorithms might iterate on an image until some criteria that the results are "good enough" is reached. Such a construct can be implemented in dataflow but is quite awkward.

In summary use dataflow where there are only a few streams that need to be replicated and where the MATLAB loop is known to execute a predetermined number of times. Use Idea loops otherwise.

Conditionals

Conditionals in MATLAB can easily be translated to conditionals in Idea. C style if and switch statements are supported:

The if Statement

An example if/else statement is:

```
stream float x = …
stream float a1 = …
if (x > 1.0) {
   b = 1+a1;
   y = x-b;
  } else {
   y = x-1;
  }
}
z = y+1;
q = b+2;
```

Some things to note are:

- Unlike C the if and else bodies must be surrounded by `{}` even if a single line
- A token is consumed off of the `a1` input even for every `y` token produced even though it only appears in the if clause
- The variable name `y` can be repeated in both the if and else clause and if so one `y` token is produced for every input token consumed
- The variable named `b` is only produced when the if clause is invoked. So `z` and `q` have different data rates. An expression like `w = y+b` outside the if/else statement would be caught at compile time as a dataflow error.

An if clause does not require an else – for example:

```
if (x > 0) {
  y = x;
}
```

produces a stream y that only has the tokens on stream x that are positive.

The switch Statement

An Idea switch statement is illustrated below:

```
stream float x = …
stream float a1 = …
stream int mode = …

switch (mode) {
case 0: {
  b = 1+a1;
  y = x-b;
}
case 1: {
  y = x-1;
}
case 2: {
  b = 1-a1;
  y = x-b;
}
default: {
  y = x-3;
}
}
```

Some things to note are:

- Unlike C a break statement is not allowed or required at the end of a case body
- Unlike C even single line case bodies must be surrounded by {}
- A token is consumed off of the a1 input even for every y token produced even though it only appears in the case 0 clause
- The variable name y can be repeated in all the cases and the optional default clause and if so one y token is produced for every input token consumed
- The variable named b is only produced when the case 0 or case 3 clause is invoked. So z and q have different data rates. An expression like w = y+b outside the switch statement would be caught at compile time as a dataflow error.
- If a default clause is not provided then y is assumed to be at a different - lower – data rate than x as it is not guaranteed to be produced. In that case an expression outside the switch statement w = y+x would produce a dataflow error at compile time.

Translating MATLAB Function into Idea

When translating MATLAB into Idea the developer needs to know if the equivalent Idea function is already provided and if not how to create new equivalent functions in Idea.

Defining Functions in Idea

The main difference between declaring functions in Idea and MATLAB is that Idea requires the type and dimensionality of all function arguments to be declared in the prototype. So for example a multiple output MATLAB function declared as

```
[A,B] = func(C,D,E) {
   A = ..
   B = ..
}
```

In Idea this function is declared as

```
A,B func(float C[r][c], int D, complex E[n]) {
   A[r][c] = ..  // type derived from rhs of expression
   int B[n] = .. // type set to int and may force type conversion
}
```

As can be seen all Idea function arguments must declare both their type and dimensionality. The `A` and `B` outputs of the Idea function – as all lhs expressions – must declare their range variables in the body of the function. However, declaring the output type is optional if the type can be derived implicitly from the rhs of the expression.

Discussion

MATLAB Features not Directly Supported in Idea

Recursion

Recursive functions cannot be written in Idea. Unbounded runtime recursion is not expected to ever be possible in Idea though a recursion of a fixed depth recursion could be implemented at compile time.

Data Dependent Casting

The expression `sqrt(-1.0)` while legal in MATLAB is illegal in Idea. Because the type of the argument to sqrt is of type float it is assumed the output is of type float. However `complex I = -1;  z = sqrt(I)` does work in Idea.

No Variable Length Arguments

Functions in Idea currently have fixed argument lists with no possibility of setting default values. An additional syntax to set default argument values and use these values if the calling argument list is truncated would be a possible extension to Idea

Utilization of Unindexed Arrays in Algebraic Expressions

Currently an expression such as

```
A = B+C;
```

in MATLAB must be implemented in Idea as

```
A[r][c] = B[r][c] + C[r][c];
```

The evolution of the Idea language will be guided by external forces. It may well include the adoption of some matrix algebra syntax to reduce the use of indices.

Importing external libraries and source code into Idea Kernels

It is straight forward to incorporate external libraries and source code into Gedae using Idea kernels. Kernels are a leaf node in a Gedae application. The kernel can either be explicitly added to a block diagram by name or it can be referenced in an Idea expression based on an equation registered in the Equation field of a kernel. An example of an absolute value kernel is:

```
Name: sf_abs
Equation: out = abs(in);
Type: stream
Comment: "Absolute Value"
Input: {
  stream float in;
}
Output: {
  inplace stream float out={in};
}
Include: {
#include <e_vabs.h>
}
Exec: {
  e_vabs(in,1,out,1,size(in));  /* vector absolute value */
}
```

The equation field registers this kernel with the compiler for use in Idea expressions that call an abs() function. There are equivalent kernels for other token and data types. It is convenient to think of a kernel as defining a function but with a much richer semantics for inputs, outputs and methods. The Exec field in this example is an execution method that processes stream inputs and create stream outputs. An input or output can be a stream or a parameter, and a stream input or output can be held for N executions. There can be a Local field with persistent (state) data and Reset methods to reset the state to initial values at the appropriate time during the processing. The E functions (e_ prefix) showing in this example are Gedae's C code functions and libraries of optimized code for each target platform. The compiler substitutes the optimized versions for the C code versions where appropriate. The document "Kernel Reference Manual.docx" explains kernels in detail.

MATLAB functions provided in Idea

Gedae has a significant and growing library of functions that can be used by the compiler to implement Idea code. Many of these functions are core MATLAB functions. Gedae is currently adding more functions to the standard library. Some of these are similar to MATLAB toolkit functions. One thing to keep in mind is that all functions provided in Gedae's library are free and open source.

Standard C Library functions

Gedae provides implementations of the standard C library functions overloaded to handle all the basic Idea data types. These functions cover many of the same functions provided by MATLAB.

MATLAB Toolbox Functions

Gedae Inc. may at its discretion or under contract implement MATLAB toolbox functions required by a developer.

Regression testing – comparing MATLAB output to Idea output

Gedae provides .m functions to use in MATLAB or Octave to create Gedae data probe files. These files can be used with Gedae probe tools to verify the conversion of the MATLAB code into the Idea language. For more information see the document “