MATLAB Tutorial

Chapter 7. Data structures and input assertion
7.1. User-defined data structures

Vectors and matrices are not the only means that MATLAB offers for grouping data into a single entity. User defined data structures are also available that enable the programmer to create variable types that mix numbers, strings, and arrays. As an example, let us create a data structure that contains the information for a single student.

We will store the name, status (year and department), the homework and exam grades, and the final class grade.

First, we can define a NameData structure to contain the name. Here, the "." operator, used in the case of Structure.Field tells MATLAB to access the field named "Field" in the structure "Structure".

```
NameData.First = 'John';
NameData.MI = 'J';
NameData.Last = 'Doe';
```

We now create a StudentData structure with a name field.

```
StudentData.Name = NameData;
```

We now initialize the rest of the structure.

```
StudentData.Status = 'ChE grad 1';
StudentData.HW = 10;
StudentData.Exam = linspace(100,100,3);
```

We can now view the contents of the structure

```
StudentData
StudentData.Name
StudentData.Exam
```

We can operate on the elements of a structure.

```
StudentData.Exam(3) = 0;
StudentData.Exam
StudentData.Name.First = 'Jane';
StudentData.Name
```

We can also create arrays of structures

```
num_students = 5;
for i=1:num_students
ClassData(i) = StudentData;
end
ClassData
ClassData(2)
```

Structures can be passed as arguments to functions in the same manner as scalars, vectors, and matrices. In this case, we use the function pass_or_fail listed below.

```
message = pass_or_fail(ClassData(2));
message
```

File pass_or_fail.m

```
function message = pass_or_fail(StudentData)
Exam_avg = mean(StudentData.Exam);

if(Exam_avg >= 70)
message = 'You pass!';
else
message = 'You fail!';
end

return;
```

7.2. Input assertion routines

Good programming style dictates the practice of defensive programming, that is, anticipating and detecting possible errors before they cause a run-time error that results in a halt to the program execution or a crash. This allows one to save the current data to the disk or take corrective action to avoid a catastrophic failure. One common source of errors can be avoided by having each subroutine make sure that the data that it has been fed through its argument list is of the approriate type, e.g. argument 1 should be a real, positive, scalar integer and argument 2 should be a real, non-negative column vector of length N. The following m-files are useful for automating this checking process, and a scalar input function is provided to allow the robust entry of data from the keyboard.

assert_scalar.m

```
function [iflag_assert,message] = assert_scalar( ...
i_error,value,name,func_name, ...
check_real,check_sign,check_int,i_error);
```

This m-file contains logical checks to assert than an input value is a type of scalar number. This function is passed the value and name of the variable, the name of the function making the assertion, and four integer flags that have the following usage :

i_error : controls what to do if test fails
if i_error is non-zero, then use error()
MATLAB command to stop execution, otherwise just return the appropriate negative number.
if i_error > 1, then dump current state to dump_error.mat before calling error().

check_real : check to examine whether input number is real or not. See table after function header for set values of these case flags
check_real = i_real (make sure that input is real)
check_real = i_imag (make sure that input is purely imaginary)
any other value of check_real (esp. 0) results in no check

check_real
i_real = 1;
i_imag = -1;

check_sign : check to examine sign of input value see table after function header for set values
of these case flags
check_sign = i_pos (make sure input is positive)
check_sign = i_nonneg (make sure input is non-negative)
check_sign = i_neg (make sure input is negative)
check_sign = i_nonpos (make sure input is non-positive)
check_sign = i_nonzero (make sure input is non-zero)
check_sign = i_zero (make sure input is zero)

any other value of check_sign (esp. 0) results in no check

check_sign
i_pos = 1;
i_nonneg = 2;
i_neg = -1;
i_nonpos = -2;
i_nonzero = 3;
i_zero = -3;

check_int : check to see if input is an integer
if = 1, then check to make sure input is an integer
any other value, perform no check

Kenneth Beers
Massachusetts Institute of Technology
Department of Chemical Engineering

7/2/2001

Version as of 7/21/2001

```
function [iflag_assert,message] = assert_scalar( ...
i_error,value,name,func_name, ...
check_real,check_sign,check_int);

iflag_assert = 0;
message = 'false';
```

First, set case values of check integer flags.
check_real
```
i_real = 1;
i_imag = -1;
```
check_sign
```
i_pos = 1;
i_nonneg = 2;
i_neg = -1;
i_nonpos = -2;
i_nonzero = 3;
i_zero = -3;
```

Check to make sure input is numerical and not a string.
```
if(~isnumeric(value))
message = [ func_name, ': ', ...
name, ' is not numeric'];
iflag_assert = -1;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end
```

Check to see if it is a scalar.

```
if(max(size(value)) ~= 1)
message = [ func_name, ': ', ...
name, ' is not scalar'];
iflag_assert = -2;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end
```

Then, check to see if it is real.

```
switch check_real;

case {i_real}
if(~isreal(value))
message = [ func_name, ': ', ...
name, ' is not real'];
iflag_assert = -3;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end

case {i_imag}
if(real(value))
message = [ func_name, ': ', ...
name, ' is not imaginary'];
iflag_assert = -3;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end
end
```

Next, check sign.

```
switch check_sign;

case {i_pos}
if(value <= 0)
message = [ func_name, ': ', ...
name, ' is not positive'];
iflag_assert = -4;
```

```
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end

case {i_nonneg}
if(value < 0)
message = [ func_name, ': ', ...
name, ' is not non-negative'];
iflag_assert = -4;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end

case {i_neg}
if(value >= 0)
message = [ func_name, ': ', ...
name, ' is not negative'];
iflag_assert = -4;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end

case {i_nonpos}
if(value > 0)
message = [ func_name, ': ', ...
name, ' is not non-positive'];
iflag_assert = -4;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end

case {i_nonzero}
if(value == 0)
message = [ func_name, ': ', ...
```

```
name, ' is not non-zero'];
iflag_assert = -4;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end

case {i_zero}
if(value ~= 0)
message = [ func_name, ': ', ...
name, ' is not zero'];
iflag_assert = -4;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end
end
```

Finally, check to make sure it is an integer.

```
if(check_int == 1)
if(round(value) ~= value)
message = [ func_name, ': ', ...
name, ' is not an integer'];
iflag_assert = -5;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end
end
```

set flag for succesful passing of all checks

```
iflag_assert = 1;
message = 'true';

return;
```

assert_vector.m

function [iflag_assert, message] = ...
assert_vector(...
i_error,value,name,func_name,num_dim, ...

check_real,check_sign,check_int,check_column);

This m-file contains logical checks to assert than an input value is a vector of a given type.
This function is passed the value and name of the variable, the name of the function making the
assertion, the dimension that the vector is supposed to be, and five integer flags that have the following usage :

i_error : controls what to do if test fails
if i_error is non-zero, then use error()
MATLAB command to stop execution, otherwise
just return the appropriate negative number.
if i_error > 1, create file dump_error.mat
before calling error()

check_real : check to examine whether input is real
see table after function header for set
values of these case flags
check_real = i_real (make sure that input is real)
check_real = i_imag (make sure that input
is purely imaginary)
any other value of check_real (esp. 0)
results in no check

check_real
i_real = 1;
i_imag = -1;

check_sign : check to examine sign of input see table after function header for set
values of these case flags
check_sign = i_pos (make sure input is positive)
check_sign = i_nonneg (make sure input is non-negative)
check_sign = i_neg (make sure input is negative)
check_sign = i_nonpos (make sure input is non-positive)
check_sign = i_nonzero (make sure input is non-zero)
check_sign = i_zero (make sure input is zero)
any other value of check_sign (esp. 0)
results in no check

check_sign
i_pos = 1;
i_nonneg = 2;
i_neg = -1;
i_nonpos = -2;
i_nonzero = 3;
i_zero = -3;

check_int : check to see if input is an integer
if = 1, then check to make sure input is an integer
any other value, perform no check

check_column : check to see if input is a column or row vector
check_column = i_column (make sure input is
column vector)
check_column = i_row (make sure input is row vector)
any other value, perform no check

check_column
i_column = 1;

i_row = -1;

if the dimension num_dim is set to zero, no check as to the dimension of the vector is made.

Kenneth Beers
Massachusetts Institute of Technology
Department of Chemical Engineering

7/2/2001

Version as of 7/21/2001

function [iflag_assert,message] = ...
assert_vector(...
i_error,value,name,func_name,num_dim, ...
check_real,check_sign,check_int,check_column);

First, set case values of check integer flags.
check_real
i_real = 1;
i_imag = -1;
check_sign
i_pos = 1;
i_nonneg = 2;
i_neg = -1;
i_nonpos = -2;
i_nonzero = 3;
i_zero = -3;
check_column
i_column = 1;
i_row = -1;

iflag_assert = 0;
message = 'false';

Check to make sure input is numerical and not a string.
if(~isnumeric(value))
message = [func_name, ': ', ...
name, 'is not numeric'];
iflag_assert = -1;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end

Check to see if it is a vector of the proper length.
num_rows = size(value,1);
num_columns = size(value,2);
if it is a multidimensional array
if(length(size(value)) > 2)
message = [func_name, ': ', ...
name, 'has too many subscripts'];

```
iflag_assert = -2;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end
```

if both the number of rows and number of columns are not equal to 1, then value is a matrix instead of a vector.

```
if(and((num_rows ~= 1),(num_columns ~= 1)))
message = [ func_name, ': ', ...
name, 'is not a vector'];
iflag_assert = -2;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end
```

if the dimension of the vector is incorrect

```
if(num_dim ~= 0)
if(length(value) ~= num_dim)
message = [ func_name, ': ', ...
name, 'is not of the proper length'];
iflag_assert = -2;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end
end
```

check to make sure that the vector is of the correct type (e.g. column)

```
switch check_column;

case {i_column}
```

check to make sure that it is a column vector

```
if(num_columns > 1)
message = [ func_name, ': ', ...
name, 'is not a column vector'];
iflag_assert = -2;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
```

```
else
return;
end
end

case {i_row}
if(num_rows > 1)
message = [ func_name, ': ', ...
name, 'is not a row vector'];
iflag_assert = -2;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end
end
```

Then, check to see if all elements are of the proper complex type.

```
switch check_real;

case {i_real}
```

if any element of value is not real

```
if(any(~isreal(value)))
message = [ func_name, ': ', ...
name, ' is not real'];
iflag_assert = -3;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end

case {i_imag}
```

if any element of value is not purely imaginary

```
if(any(real(value)))
message = [ func_name, ': ', ...
name, ' is not imaginary'];
iflag_assert = -3;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end
end
```

Next, check sign.

```
switch check_sign;

case {i_pos}
if any element of value is not positive
if(any(value <= 0))
message = [ func_name, ': ', ...
name, ' is not positive'];
iflag_assert = -4;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end

case {i_nonneg}
if any element of value is negative
if(any(value < 0))
message = [ func_name, ': ', ...
name, ' is not non-negative'];
iflag_assert = -4;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end

case {i_neg}
if any element of value is not negative
if(any(value >= 0))
message = [ func_name, ': ', ...
name, ' is not negative'];
iflag_assert = -4;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end

case {i_nonpos}
if any element of value is positive
if(any(value > 0))
message = [ func_name, ': ', ...
name, ' is not non-positive'];
iflag_assert = -4;
if(i_error ~= 0)
```

```
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end

case {i_nonzero}
```
if any element of value is zero
```
if(any(value == 0))
message = [ func_name, ': ', ...
name, 'is not non-zero'];
iflag_assert = -4;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end

case {i_zero}
```
if any element of value is non-zero
```
if(any(value ~= 0))
message = [ func_name, ': ', ...
name, ' is not zero'];
iflag_assert = -4;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end
end
```

Finally, check to make sure it is an integer.

```
if(check_int == 1)
if(any(round(value) ~= value))
message = [ func_name, ': ', ...
name, ' is not an integer'];
iflag_assert = -5;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end
end
```

set flag for succesful passing of all checks

```
iflag_assert = 1;
message = 'true';

return;
```

assert_matrix.m

```
function [iflag_assert,message] = assert_matrix( ...
i_error,value,name,func_name, ...
num_rows,num_columns, ...
check_real,check_sign,check_int);
```

This m-file contains logical checks to assert than an input value is a matrix of a given type. This function is passed the value and name of the variable, the name of the function making the assertion, the dimension that the matrix is supposed to be, and four integer flags that have the following usage :

i_error : controls what to do if test fails
if i_error is non-zero, then use error()
MATLAB command to stop execution, otherwise just return the appropriate negative number.
if i_error > 1, create file dump_error.mat
before calling error()

check_real : check to examine whether input is real see table after function header for set values of these case flags
check_real = i_real (make sure that input is real)
check_real = i_imag (make sure that input is
purely imaginary)
any other value of check_real (esp. 0)
results in no check

check_real
i_real = 1;
i_imag = -1;

check_sign : check to examine sign of input
see table after function header for set
values of these case flags
check_sign = i_pos (make sure input is positive)
check_sign = i_nonneg (make sure input is non-negative)
check_sign = i_neg (make sure input is negative)
check_sign = i_nonpos (make sure input is non-positive)
check_sign = i_nonzero (make sure input is non-zero)
check_sign = i_zero (make sure input is zero)
any other value of check_sign (esp. 0)
results in no check

check_sign
i_pos = 1;
i_nonneg = 2;
i_neg = -1;
i_nonpos = -2;
i_nonzero = 3;
i_zero = -3;

check_int : check to see if input value is an integer

if = 1, then check to make sure input is an integer
any other value, perform no check

if the dimensions num_rows or num_columns
are set to zero, no check as to that dimension of the matrix is made.

Kenneth Beers
Massachusetts Institute of Technology
Department of Chemical Engineering

7/2/2001

Version as of 7/21/2001

```
function [iflag_assert,message] = assert_matrix( ...
i_error,value,name,func_name, ...
num_rows,num_columns, ...
check_real,check_sign,check_int);
```

First, set case values of check integer flags.
check_real
```
i_real = 1;
i_imag = -1;
check_sign
i_pos = 1;
i_nonneg = 2;
i_neg = -1;
i_nonpos = -2;
i_nonzero = 3;
i_zero = -3;

iflag_assert = 0;
message = 'false';
```

Check to make sure input is numerical and not a string.
```
if(~isnumeric(value))
message = [ func_name, ': ', ...
name, ' is not numeric'];
iflag_assert = -1;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end
```

Check to see if it is a matrix of the proper length.
if it is a multidimensional array
```
if(length(size(value)) > 2)
message = [ func_name, ': ', ...
name, ' has too many subscripts'];
iflag_assert = -2;
if(i_error ~= 0)
if(i_error > 1)
```

```
save dump_error.mat;
end
error(message);
else
return;
end
end
```

check that value has the proper number of rows

```
if(num_rows ~= 0)
if(size(value,1) ~= num_rows)
message = [ func_name, ': ', ...
name, ' has the wrong number of rows'];
iflag_assert = -2;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end
end
```

check that value has the proper number of columns

```
if(num_columns ~= 0)
if(size(value,2) ~= num_columns)
message = [ func_name, ': ', ...
name, ' has the wrong number of columns'];
iflag_assert = -2;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end
end
```

Then, check to see if all elements are of the proper complex type.

```
switch check_real;

case {i_real}
```

if any element of value is not real

```
if(any(~isreal(value)))
message = [ func_name, ': ', ...
name, ' is not real'];
iflag_assert = -3;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
```

```
end
end

case {i_imag}
if any element of value is not purely imaginary
if(any(real(value)))
message = [ func_name, ': ', ...
name, ' is not imaginary'];
iflag_assert = -3;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end
end

Next, check sign.
switch check_sign;

case {i_pos}
if any element of value is not positive
if(any(value <= 0))
message = [ func_name, ': ', ...
name, ' is not positive'];
iflag_assert = -4;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end

case {i_nonneg}
if any element of value is negative
if(any(value < 0))
message = [ func_name, ': ', ...
name, ' is not non-negative'];
iflag_assert = -4;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end

case {i_neg}
if any element of value is not negative
```

```
if(any(value >= 0))
message = [ func_name, ': ', ...
name, ' is not negative'];
iflag_assert = -4;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end

case {i_nonpos}
if any element of value is positive
if(any(value > 0))
message = [ func_name, ': ', ...
name, ' is not non-positive'];
iflag_assert = -4;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end

case {i_nonzero}
if any element of value is zero
if(any(value == 0))
message = [ func_name, ': ', ...
name, 'is not non-zero'];
iflag_assert = -4;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end

case {i_zero}
if any element of value is non-zero
if(any(value ~= 0))
message = [ func_name, ': ', ...
name, ' is not zero'];
iflag_assert = -4;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
```

```
return;
end
end
end
```

Finally, check to make sure it is an integer.

```
if(check_int == 1)
if(any(round(value) ~= value))
message = [ func_name, ': ', ...
name, ' is not an integer'];
iflag_assert = -5;
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end
end
```

set flag for succesful passing of all checks

```
iflag_assert = 1;
message = 'true';

return;
```

assert_structure.m

function [iflag_assert,message] = assert_structure(...
i_error,Struct,struct_name,func_name,StructType);

This MATLAB m-file performs assertions on a data structure. It makes use of assert_scalar, assert_vector, and assert_matrix for the fields.

INPUT :
=======
i_error controls what to do if test fails
if i_error is non-zero, then use error()
MATLAB command to stop execution, otherwise just return the appropriate negative number.
if i_error > 1, then dump current state to dump_error.mat before calling error().
Struct This is the structure to be checked
struct_name the name of the structure
func_name the name of the function making the assertion
StructType this is a structure that contains the typing data for each field.
.num_fields is the total number of fields
Then, for i = 1,2, ..., StructType.num_fields, we have :
.field(i).name the name of the field
.field(i).is_numeric if non-zero, then field is numeric
.field(i).num_rows # of rows in field
.field(i).num_columns # of columns in field
.field(i).check_real value of check_real passed to assertion
.field(i).check_sign value of check_sign passed to assertion
.field(i).check_int value of check_int passed to assertion

OUTPUT :
=======
iflag_assert an integer flag telling of outcome message a message passed that describes the result of making the assertion

Kenneth Beers
Massachusetts Institute of Technology
Department of Chemical Engineering

7/2/2001

Version as of 7/25/2001

```
function [iflag_assert,message] = assert_structure(...
i_error,Struct,struct_name,func_name,StructType);

iflag_assert = 0;
message = 'false';

first, check to make sure Struct is a structure
if(~isstruct(Struct))
iflag_assert = -1;
message = [func_name, ': ', struct_name, ...
' is not a structure'];
if(i_error ~= 0)
if(i_error > 1);
save dump_error.mat;
end
error(message);
else
return;
end
end

Now, for each field, perform the required assertion.
for ifield = 1:StructType.num_fields

set shortcut to current field type
FieldType = StructType.field(ifield);

check if it exists in Struct
if(~isfield(Struct,FieldType.name))
iflag_assert = -2;
message = [func_name, ': ', struct_name, ...
' does not contain ', FieldType.name];
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end

extract value of field
value = getfield(Struct,FieldType.name);
```

if the field is supposed to be numeric
```
if(FieldType.is_numeric ~= 0)
```

check to make sure field is numeric
```
if(~isnumeric(value))
iflag_assert = -3;
message = [func_name, ': ', ...
struct_name, '.', FieldType.name, ...
' is not numeric'];
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
end
```

decide which assertion statement to use based on array dimension of field value
If both num_rows and num_columns are set equal to zero, then no check of the dimension of this field is made.
```
if(and((FieldType.num_rows == 0), ...
(FieldType.num_columns == 0)))

message = [func_name, ': ', ...
struct_name,'.',FieldType.name, ...
' is not checked for dimension'];
if(i_error ~= 0)
disp(message);
end
```

else, peform check of dimension to make sure it is a scalar, vector, or matrix (i.e. a two dimensional array).
```
else
```

check that is is not a multidimensional array
```
if(length(size(value)) > 2)
iflag_assert = -4;
message = [func_name, ': ', ...
struct_name,'.',FieldType.name, ...
' is multidimensional array'];
if(i_error ~= 0)
if(i_error > 1)
save dump_error.mat;
end
error(message);
else
return;
end
```

else if scalar
```
elseif(and((FieldType.num_rows == 1), ...
(FieldType.num_columns == 1)))
assert_scalar(i_error,value, ...
[struct_name,'.',FieldType.name], ...
```

```
func_name,FieldType.check_real, ...
FieldType.check_sign,FieldType.check_int);
```

else if a column vector

```
elseif (and((FieldType.num_rows > 1), ...
(FieldType.num_columns == 1)))
dim = FieldType.num_rows;
check_column = 1;
assert_vector(i_error,value, ...
[struct_name,'.',FieldType.name], ...
func_name,dim,FieldType.check_real, ...
FieldType.check_sign,FieldType.check_int, ...
check_column);
```

else if a row vector

```
elseif (and((FieldType.num_rows == 1), ...
(FieldType.num_columns > 1)))
dim = FieldType.num_columns;
check_column = -1;
assert_vector(i_error,value, ...
[struct_name,'.',FieldType.name], ...
func_name,dim,FieldType.check_real, ...
FieldType.check_sign,FieldType.check_int, ...
check_column);
```

otherwise, a matrix

```
else
assert_matrix(i_error,value, ...
[struct_name,'.',FieldType.name], ...
func_name, ...
FieldType.num_rows,FieldType.num_columns, ...
FieldType.check_real,FieldType.check_sign, ...
FieldType.check_int);
```

end selection of assertion routine
end if perform check of dimension
end if (FieldType.is_numeric ~= 0)
end for loop over fields

set return results for succesful assertion

```
iflag_assert = 1;
message = 'true';
```

```
return;
```

get_input_scalar.m

function value = get_input_scalar(prompt, ...
check_real,check_sign,check_int);

This MATLAB m-file gets from the user an input scalar value of the appropriate type. It asks for input over and over again until a correctly typed input value is entered.

Kenneth Beers
Massachusetts Institute of Technology
Department of Chemical Engineering

7/2/2001

Version as of 7/25/2001

```
function value = get_input_scalar(prompt, ...
check_real,check_sign,check_int);

func_name = 'get_input_scalar';
name = 'trial_value';

input_OK = 0;

while (input_OK ~= 1)
trial_value = input(prompt);
[iflag_assert,message] = ...
assert_scalar(0,trial_value, ...
name,func_name, ...
check_real,check_sign,check_int);
if(iflag_assert == 1)
input_OK = 1;
value = trial_value;
else
disp(message);
end
end

return;
```