**Uncertainty Toolbox**

**Developer’s Manual**

Slimane Aoudjit

Wolfgang Granig

Hubert Zangl

Dailys Arronde Pérez

**Uncertainty Toolbox Version V0.91, 22.05.2019**

# Revision history

|  |  |  |  |
| --- | --- | --- | --- |
| **Description** | **Author** | **Approved** | **Version** |
| Initial version | Granig |  | V0.1 |
| Version 0.9 | Aoudjit |  | V0.9 |
| Version 0.91 | Arronde |  | V0.91 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

# Content

[1. Revision history 2](#_Toc4063150)

[2. Content 3](#_Toc4063151)

[3. Disclaimer 7](#_Toc4063152)

[4. General description 7](#_Toc4063153)

[5. Notation 7](#_Toc4063154)

[6. Uncertainty Class 8](#_Toc4063155)

[6.1. The name Property 8](#_Toc4063156)

[6.2. The value Property 8](#_Toc4063157)

[6.3. The std\_unc Property 8](#_Toc4063158)

[6.4. The dep Property 8](#_Toc4063159)

[6.5. The grad Property 9](#_Toc4063160)

[6.6. The OutDataTypeStr Property 9](#_Toc4063161)

[6.7. The rel\_names Property 9](#_Toc4063162)

[6.8. The rel\_mat Property 9](#_Toc4063163)

[6.9. The r Property 10](#_Toc4063164)

[6.10. The img Property 10](#_Toc4063165)

[7. Properties Handling Methods 11](#_Toc4063166)

[7.1. function obj = set.name(obj,Name) 11](#_Toc4063167)

[7.2. function obj = set.value(obj,Value) 13](#_Toc4063168)

[7.3. function obj = set.std\_unc(obj,Std\_Unc) 14](#_Toc4063169)

[8. Class Constructor: function obj = unc(arg1,arg2,arg3) 15](#_Toc4063170)

[8.1. Global Variables 15](#_Toc4063171)

[8.2. Syntax 1: unc(value,std\_unc,name) 16](#_Toc4063172)

[8.3. Syntax 2: unc(value,std\_unc,name) 16](#_Toc4063173)

[8.3.1. Syntax 2.1: unc(value,std\_unc,name) 17](#_Toc4063174)

[8.4. Syntax 3: unc( unc\_objects , CX , 'cov') 18](#_Toc4063175)

[8.5. Syntax 4: unc( unc\_objects , CX , 'corr') 18](#_Toc4063176)

[8.6. Syntax 5: unc( value , name ) 19](#_Toc4063177)

[8.7. Syntax 6: unc( value , std\_unc ) 20](#_Toc4063178)

[8.8. Syntax 7: unc( value , name ) 20](#_Toc4063179)

[8.9. Syntax 8: unc( value , CX ) 21](#_Toc4063180)

[8.10. Syntax 9: unc( unc\_objects , CX ) 21](#_Toc4063181)

[8.11. Syntax 10: unc( value ) 22](#_Toc4063182)

[8.12. Syntax 11: unc( unc\_object ) 22](#_Toc4063183)

[8.13. Syntax 12: unc( ) 23](#_Toc4063184)

[9. Operator Overloading 23](#_Toc4063185)

[9.1. function obj = plus(obj1,obj2) 23](#_Toc4063186)

[9.2. function obj = minus(obj1,obj2) 25](#_Toc4063187)

[9.3. function obj = mtimes(obj1,obj2) 26](#_Toc4063188)

[9.4. function obj=rdivide(obj1,obj2) 27](#_Toc4063189)

[9.5. function obj=ldivide(obj1,obj2) 27](#_Toc4063190)

[9.6. function obj =mldivide(obj1,obj2) 28](#_Toc4063191)

[9.7. function obj =mrdivide(obj1,obj2) 28](#_Toc4063192)

[9.8. function obj=times(obj1,obj2) 32](#_Toc4063193)

[9.9. function obj=uplus(obj1) 33](#_Toc4063194)

[9.10. function obj=uminus(obj1) 33](#_Toc4063195)

[9.11. function obj = gt(obj1,obj2) 33](#_Toc4063196)

[9.12. function obj = lt(obj1,obj2) 34](#_Toc4063197)

[9.13. function obj = mpower(obj1,obj2) 35](#_Toc4063198)

[9.14. function obj = realpow(obj1,obj2) 36](#_Toc4063199)

[9.15. function obj =power(obj1,obj2) 37](#_Toc4063200)

[10. Function Overloading 39](#_Toc4063201)

[10.1. function disp(obj) 39](#_Toc4063202)

[10.2. function obj = sqrt(obj1) 40](#_Toc4063203)

[10.3. function obj = atan2(obj1,obj2) 40](#_Toc4063204)

[10.4. function obj = prod(obj1) 41](#_Toc4063205)

[10.5. function obj = diag(obj1) 41](#_Toc4063206)

[10.6. function obj = trace(obj1) 42](#_Toc4063207)

[10.7. function obj = ctranspose(obj1) 42](#_Toc4063208)

[10.8. function obj = max(obj1) 43](#_Toc4063209)

[10.9. function obj = min(obj1) 44](#_Toc4063210)

[10.10. function obj = sum(obj1) 44](#_Toc4063211)

[10.11. function obj = cumsum(obj1) 45](#_Toc4063212)

[10.12. function obj = inv(obj1) 46](#_Toc4063213)

[10.13. function [determ]= det(obj) 46](#_Toc4063214)

[10.14. function Z = complex(x,y) 48](#_Toc4063215)

[10.15. function X=real(Z) 49](#_Toc4063216)

[10.16. function Y=imag(Z) 49](#_Toc4063217)

[10.17. function obj=conj(Z) 50](#_Toc4063218)

[10.18. function obj = abs(Z) 50](#_Toc4063219)

[10.19. function obj = angle(Z) 50](#_Toc4063220)

[11. Special Uncertainty Class Methods 51](#_Toc4063221)

[11.1. function value = gmv(obj) 51](#_Toc4063222)

[11.2. function std\_unc = gmu(obj) 51](#_Toc4063223)

[11.3. function names = gmn(obj) 52](#_Toc4063224)

[11.4. function obj = eval\_std\_unc(obj1) 52](#_Toc4063225)

[11.5. function [ obj ] = eval\_obj(str,obj1 ) 54](#_Toc4063226)

[11.6. function disp\_contribution(obj1) 60](#_Toc4063227)

[11.7. function disp\_dep(obj) 61](#_Toc4063228)

[11.8. function DispString = GenerateDispStr(value,std\_unc) 62](#_Toc4063229)

[11.9. function obj = plusscal(obj1,obj2) 65](#_Toc4063230)

[11.10. function obj = minusscal(obj1,obj2) 70](#_Toc4063231)

[11.11. function obj = multscal(obj1,obj2) 73](#_Toc4063232)

[11.12. function obj = divscal(obj1,obj2) 75](#_Toc4063233)

[11.13. function set\_cov(x1,x2,cov) 76](#_Toc4063234)

[11.14. function set\_correl(x1,x2,cor) 78](#_Toc4063235)

[11.15. function [CX,Dep\_ord] = get\_input\_cov\_mat(Y) 81](#_Toc4063236)

[11.16. function [CX] = get\_input\_cor\_mat(Y) 82](#_Toc4063237)

[11.17. function [CX\_cov,Y\_new,Names\_Y\_new]=get\_cov\_mat(X) 83](#_Toc4063238)

[11.18. function [CX\_cor]=get\_cor\_mat(Y) 85](#_Toc4063239)

[11.19. function [CX\_cov] = cor\_into\_cov\_mat(X,CX\_cor) 85](#_Toc4063240)

[11.20. function [CX\_cor] = cov\_into\_cor\_mat(X,CX\_cov) 85](#_Toc4063241)

[11.21. function [K, Coef\_mat]=karhloev( X,CX,arg3 ) 86](#_Toc4063242)

[11.22. function [cov,CY\_cov] = covr(y1,y2) 87](#_Toc4063243)

[11.23. function [cor,CY\_corr] = correl(y1,y2) 91](#_Toc4063244)

[11.24. function [J,Names]= jacobian(obj) 95](#_Toc4063245)

[11.25. function [obj\_names,obj\_array] = get\_ws\_obj() 96](#_Toc4063246)

[11.26. function data = read\_data(arg1,arg2,arg3,arg4,arg5) 97](#_Toc4063247)

[12. References 101](#_Toc4063248)

# Disclaimer

THE SAT/IST/AAU MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

# General description

This uncertainty toolbox was developed in order to enhance the widespread use of uncertainty calculations in measurement science that is directly based on the utilization of software tools. The developed material is currently based on the GUM (Guide to the expression of uncertainty in measurement) method. Additionally, the concept of this toolbox goes beyond the scope of measurement uncertainty quantification demonstrating that it is also useful for system analysis and optimization. [1]

# Notation

MATLAB commands and variables are written in blue color, whereas mathematical equations and variables are written in red color.

# Uncertainty Class

The uncertainty class has the following properties:

## The name Property

It represents the name of the uncertainty object. If x is an uncertainty object, then the *name* property of x can be accessed using the command:

x.name;

It can also be accessed using the method gmn():

gmn(x);

Names must be assigned to uncertainty objects in order to be able to perform covariance and correlation calculations.

## The value Property

It represents the mean value of the uncertainty object. If x is an uncertainty object, then the *value* property of x can be accessed using the command:

x.value;

It can also be accessed using the method gmv():

gmv(x);

## The std\_unc Property

It represents the standard uncertainty of the uncertainty object. If x is an uncertainty object, then the std\_unc property of x can be accessed using the command:

x.std\_unc;

It can also be accessed using the method gmu():

gmu(x);

## The dep Property

It is an array containing all the uncertainty objects on which a given uncertainty object depends. If x is an uncertainty object, then the *dep* property of x can be accessed using the command:

x.dep;

It can also be accessed using the method disp\_dep():

disp\_dep(x);

The method disp\_dep() displays the objects on which x depends as well as the assigned names of those objects.

## The grad Property

It is an array containing all the gradient values of the uncertainty object computed at the mean values. If x is an uncertainty object, then the *grad* property of x can be accessed using the command:

x.grad;

## The OutDataTypeStr Property

It represents the data type of the uncertainty object. If x is an uncertainty object, then the *OutDataTypeStr* property of x can be accessed using the command:

x.OutDataTypeStr;

## The rel\_obj Property

It is an array containing all the uncertainty objects with which a given uncertainty object is correlated. If x is an uncertainty object, then the *rel\_obj* property of x can be accessed using the command:

x.rel\_obj;

## The rel\_mat Property

It is a 2-row matrix which contains covariance and correlation values of the uncertainty object with respect to the uncertainty objects whose names are found in the *rel\_names* property. The first row contains covariances, whereas the second row contains correlation coefficients.

If x is an uncertainty object, then the *rel\_mat* property of x can be accessed using the command:

x.rel\_mat;

If x is correlated with n other uncertainty objects, then:

To access covariance values, use the command:

x.rel\_mat(1,i);

where i=1,2,…,n.

To access correlation values, use the command:

x.rel\_mat(2,i);

where i=1,2,…,n.

## The r Property

It is the real part of the uncertainty object. If x is an uncertainty object, then the *r* property of x can be accessed using the command:

x.r;

It can also be accessed using the method *real():*

real(x)

## The img Property

It is the imaginary part of the uncertainty object. If x is an uncertainty object, then the *img* property of x can be accessed using the command:

x.img;

It can also be accessed using the method *imag():*

imag(x)

# Properties Handling Methods

## function obj = set.name(obj,Name)

This method is used for the proper assignment of the name property to an uncertainty object as well as issue warnings to the user.

Only names of data type character or cell are allowed, otherwise an error message is displayed:

error('Invalid datatype! Name must be character or cell data type.')

All the variables present in MATLAB Workspace as well as their names are stored in the arrays obj\_array and obj\_names respectively using the method obj.get\_ws\_obj():

[obj\_names,obj\_array] = obj.get\_ws\_obj();

1. Case where the assigned name is of type character:

That is:

if isa(Name,'char')

The name is checked against the other existing name in Workspace:

c=0

for i=1:length(obj\_names)

temp = obj\_array{i}.name ;

if strcmp(temp,Name)

c=c+1;

end

end

The variable c is a counter for the number of times the variable name 'Name' is assigned. If the name already exists in Workspace, then a warning message is displayed:

warning(' Variable name ''%s'' is already ''%d'' time(s) assigned!',Name,c);

Then the variable name ‘Name’ is converted from character data type into cell data type:

obj.name = {Name};

1. Case where the assigned name is of type cell:

That is:

if isa(Name,'cell')

The name is checked against the other existing name in Workspace:

c=0;

for i=1:length(obj\_names)

temp = obj\_array{i}.name ;

if strcmp(temp,Name)

c=c+1;

end

end

The variable c is a counter for the number of times the variable name 'Name' is assigned. If c==1, then a warning message is displayed:

warning(' Variable name ''%s'' already exists in Workspace or assigned to another unc object!',Name{1});

If, on the other hand, c > 1, then the following warning message is displayed:

warning(' Variable name ''%s'' is already ''%d'' time(s) assigned to (an)other variable(s)!',Name{1},c);

Since the assigned name is of data type cell, it is directly assigned to the name property:

obj.name = Name;

Only one name is allowed to be assigned to an uncertainty object, otherwise an error message is displayed:

error('One name only is allowed!')

**Remark:**

If the name is assigned inside the class constructor, for example a=unc(1,0.1,’a’), then only case(2) will be executed because character-to-cell conversion is already implemented inside the class constructor.

On the other hand, if the name is assigned using the syntax a.name for example, then both case(1) and case(2) can be executed depending on whether the assigned name is of data type character or cell.

## function obj = set.value(obj,Value)

This method is used for the proper assignment of the value property to an uncertainty object.

If obj is a complex uncertainty object, i.e.:

if imag(Value)~=0

then obj is first split into its real and imaginary parts:

obj\_X=real(obj);

obj\_Y=imag(obj);

The assigned value *Value* is also split into its real and imaginary parts:

Value\_X=real(Value);

Value\_Y=imag(Value);

If *Value* is of data type double and its real and imaginary parts are equal to the real and imaginary parts of obj respectively, i.e.:

if isa(Value,'double') && (obj\_X.value==Value\_X) && (obj\_Y.value==Value\_Y)

then this assignment is accepted:

obj.value = Value;

If not, an error message is displayed:

error('Value property assignment to complex unc objects is not needed.')

In the case of a real uncertainty object, i.e.:

if imag(Value)==0

the value *Value* is assigned only if it is of data type double, otherwise an error message is displayed:

if isa(Value,'double')

obj.value = Value;

else

error('Invalid datatype!Value must be a double!')

end

## function obj = set.std\_unc(obj,Std\_Unc)

This method is used for the proper assignment of the std\_unc property to an uncertainty object.

If obj is a complex uncertainty object, i.e.:

if imag(obj)~=0

then obj is first split into its real and imaginary parts:

obj\_X=real(obj);

obj\_Y=imag(obj);

If *Std\_Unc* is of data type double and is equal to the standard uncertainty of the complex object obj, i.e.:

if isa(Std\_Unc,'double')&&(Std\_Unc == sqrt( obj\_X.std\_unc^2 +obj\_Y.std\_unc^2))

then this assignment is accepted:

obj.std\_unc = Std\_Unc;

If not, an error message is displayed:

error('Standard uncertainty property assignment to complex unc objects is not needed.')

In the case of a real uncertainty object, i.e.:

if imag(obj)==0

the value *Std\_Unc* is assigned only if it is of data type double, otherwise an error message is displayed:

if isa(Std\_Unc,'double')

obj.std\_unc = Std\_Unc;

else

error('Invalid datatype!std\_unc must be a double!')

end

# Class Constructor: function obj = unc(arg1,arg2,arg3)

## Global Variables

Two global variables are used in the Uncertainty Class, namely AUTTOOL and Number\_of\_Digits\_to\_Display:

global AUTTOOL;

global Number\_of\_Digits\_to\_Display;

If any method or function declares as global a particular variable with the same name as AUTTOOL and Number\_of\_Digits\_to\_Display, then it will share a single copy of those variables. Any change of value to the global variables, in any function or method, will be visible to all the functions and methods that declare it as global.

Global variables are initialized to an empty 0x0 array.

In order to ensure that a disclaimer text ( disclaimer.m ) appears only when the Uncertainty Class is run for the first time, the display code:

help disclaimer;

disp(['Press any key to continue...',13])

pause;

is executed only if the global variable AUTTOOL is empty:

if isempty(AUTTOOL)

Then ATTOOL is set to 1 right after the IF statement:

AUTTOOL=1;

Indeed, global variables are initialized in MATLAB by default to an empty 0x0 array. That’s why the isempty() statement is used.

The global variable Number\_of\_Digits\_to\_Display is handled in the same way as AUTTOOL. It is set by default to 2:

Number\_of\_Digits\_to\_Display=2;

Then the following message is displayed:

fprintf(['Using %d digits to report uncertainty (global Number\_of\_Digits\_to\_Display)\n',13],Number\_of\_Digits\_to\_Display);

The uncertainty objects can be then constructed using different syntaxes.

## Syntax 1: unc(value,std\_unc,name)

The input arguments are:

*value*: 1x1 double array corresponding to the property *value*

*std\_unc*: 1x1 double array corresponding to the property *std\_unc*

*name*: 1x1 character array corresponding to the property *name*

The above restrictions on the input arguments are reflected in the conditional statement:

if isa(arg1,'double')&& isa(arg2,'double')&& isa(arg3,'char')&& length(arg1)==1 && length(arg2)==1 && size(arg3,1)==1

Then the properties of the created uncertainty object are assigned accordingly to the input arguments, either explicitly or implicitly. Those properties that are not assigned are set to default values:

obj.value=arg1;

obj.std\_unc=arg2;

obj.name={arg3};

obj.dep=obj;

obj.grad=1;

obj.OutDataTypeStr='double';

obj.rel\_obj={};

obj.rel\_mat=[];

obj.img=0;

obj.r= obj;

The command obj.name={arg3} ensures the conversion of the name property from character data type into cell data type.

## Syntax 2: unc(value,std\_unc,name)

The input arguments are:

*value*: nxm double array corresponding to the property *value*

*std\_unc*: nxm double array corresponding to the property *std\_unc*

*name*: nxm cell array corresponding to the property *name*

The above restrictions on the input arguments are reflected in the conditional statement:

if isa(arg1,'double')&& isa(arg2,'double')&& isa(arg3,'cell') && size(arg1,1)==size(arg2,1) && size(arg1,2)==size(arg2,2) && size(arg1,1)==size(arg3,1) && size(arg1,2)==size(arg3,2)

Then the properties of the created uncertainty object are assigned accordingly to the input arguments, either explicitly or implicitly. Those properties that are not assigned are set to default values:

obj(i,j).value = arg1(i,j);

obj(i,j).std\_unc = arg2(i,j);

obj(i,j).name = arg3(i,j);

obj(i,j).dep=obj(i,j);

obj(i,j).grad=1;

obj(i,j).OutDataTypeStr='double';

obj(i,j).rel\_obj={};

obj(i,j).rel\_mat=[];

obj(i,j).img=0;

obj(i,j).r= obj(i,j);

## 8.3.1. Syntax 2.1: unc(value,std\_unc,name)

In the case it is not required to assign a different name to each element of the array, the input arguments are:

*value*: nxm double array corresponding to the property *value*

*std\_unc*: nxm double array corresponding to the property *std\_unc*

*name*: 1x1 character array corresponding to the property *name.*

The conditional statement and the properties assignment are slightly changed as follows:

if isa(arg1,'double')&& isa(arg2,'double')&& isa(arg3,'char') && size(arg1,1)==size(arg2,1)&& size(arg1,2)==size(arg2,2) && size(arg3,1)== 1

obj(i,j).value = arg1(i,j);

obj(i,j).std\_unc = arg2(i,j);

obj(i,j).name = arg3 ;

obj(i,j).dep=obj(i,j);

obj(i,j).grad=1;

obj(i,j).OutDataTypeStr='double';

obj(i,j).rel\_obj={};

obj(i,j).rel\_mat=[];

obj(i,j).img=0;

obj(i,j).r= obj(i,j);

Note that the same name is assigned to the property name of all the elements in the array.

## Syntax 3: unc( unc\_objects , CX , 'cov')

The input arguments are:

*unc\_objects*: 1xn array of real uncertainty objects

*CX*: nxn double array

*‘cov’*: indicates that *CX* is a covariance matrix

The above restrictions on the input arguments are reflected in the conditional statement:

if isa(arg1,'unc') && size(arg1,1)==1 && isa(arg2, 'double') && size(arg2,1)==size(arg1,2)&& size(arg2,2)==size(arg1,2) && isa(arg3,'char') && isreal(gmv(arg1))

Additionally, the covariance matrix must be symmetric and positive semi-definite (PSD), which is checked as follows:

[~,p] = chol(arg2);

if (all(all(arg2 == transpose(arg2))) && (p==0))

In syntax 3, the input argument arg3 is selected to be ‘cov’.

The value of covariance between two random variables should not be greater than the product of its standard deviations, i.e. |cov(x,y)|≤ \* . This constraint is checked and an error message is displayed when the covariance values are out of range. The covariance and correlation values, as well as the corresponding objects, are then assigned to the rel\_mat and rel\_obj properties of each object respectively:

n=size(arg1,2); % number of uncertainty objects

for i=1:n

k=1; % counter

for j=1:n

if i~=j

%check if the values of covariances are inside the range

if (arg2(i,j) > (arg1(i).std\_unc \* arg1(j).std\_unc)) || ( arg2(i,j) < -(arg1(i).std\_unc \* arg1(j).std\_unc))

error('Incorrect value in entry(%d,%d) of the covariance matrix! \n The covariance between two random variables x and y must satisfy the property |cov(x,y)|%s %sx\*%sy, where %sx\*%sy represent the standard deviation of x and y respectively.',i,j,num2str(char(8804)),num2str(char(963)),

num2str(char(963)),num2str(char(963)),num2str(char(963)));

else

arg1(i).rel\_obj{1,k}=arg1(j);

arg1(i).rel\_mat(1,k)=arg2(i,j);

arg1(i).rel\_mat(2,k)=arg2(i,j)/ sqrt(arg2(i,i) \* arg2(j,j));

k=k+1; % increment counter

end

end

end

end

The method karhloev() is called to perform the Karhunen-Loeve decomposition of the covariance matrix, and obtain a set of uncorrelated uncertainty objects for updating the dependency property. (See Section 11.21)

## Syntax 4: unc( unc\_objects , CX , 'corr')

The input arguments are:

unc\_objects: 1xn array of real uncertainty objects

CX: nxn double array

‘corr’: indicates that CX is a correlation matrix

The implementation for syntax 4 is very similar to syntax 3. In this case, the input argument arg3 is selected to be ‘corr’. The correlation coefficient between two random variables should be between -1 and 1, i.e. |corr(x,y)|≤1. This restriction is evaluated as follows

if ( arg2(i,j) > 1 ) || ( arg2(i,j) < -1 )

displaying an error message when the value is out of range. The covariance and correlation values, as well as the corresponding objects, are then assigned to the rel\_mat and rel\_obj properties of each object respectively:

arg1(i).rel\_obj{1,k}= arg1(j);

arg1(i).rel\_mat(2,k)= arg2(i,j);

arg1(i).rel\_mat(1,k)= arg2(i,j)\*arg1(i).std\_unc\*arg1(j).std\_unc;

## Syntax 5: unc( value , name )

The input arguments are:

*value*: 1x1 double array corresponding to the property *value*

*name*: 1x1 character array corresponding to the property *name*

The above restrictions on the input arguments are reflected in the conditional statement:

if isa(arg1,'double')&& isa(arg2,'char')&& length(arg1)==1 && size(arg2,1)==1

Then the properties of the created uncertainty object are assigned accordingly to the input arguments, either explicitly or implicitly. Those properties that are not assigned are set to default values:

obj.value=arg1;

obj.std\_unc=0;

obj.name=arg2;

obj.dep=obj;

obj.grad=1;

obj.OutDataTypeStr='double';

obj.img=0;

obj.r= obj;

## Syntax 6: unc( value , std\_unc )

The input arguments are:

value: nxm double array corresponding to the property *value*

std\_unc: nxm double array corresponding to the property *std\_unc*

The above restrictions on the input arguments are reflected in the conditional statement:

if isa(arg1,'double')&& isa(arg2, 'double') && size(arg1,1)==size(arg2,1) && size(arg1,2)==size(arg2,2)

Then the properties of the created uncertainty object are assigned accordingly to the input arguments, either explicitly or implicitly. Those properties that are not assigned are set to default values:

obj(i,j).value = arg1(i,j);

obj(i,j).std\_unc = arg2(i,j);

obj(i,j).name = {};

obj(i,j).dep=obj(i,j);

obj(i,j).grad=1;

obj(i,j).OutDataTypeStr='double';

obj(i,j).rel\_obj={};

obj(i,j).rel\_mat=[];

obj(i,j).img=0;

obj(i,j).r= obj(i,j);

## Syntax 7: unc( value , name )

The input arguments are:

value: nxm double array corresponding to the property *value*

name: nxm cell array corresponding to the property *name*

The above restrictions on the input arguments are reflected in the conditional statement:

if isa(arg1,'double')&& isa(arg2, 'cell') && size(arg1,1)==size(arg2,1) && size(arg1,2)==size(arg2,2)

Then the properties of the created uncertainty object are assigned accordingly to the input arguments, either explicitly or implicitly. Those properties that are not assigned are set to default values:

obj(i,j).value = arg1(i,j);

obj(i,j).std\_unc = 0;

obj(i,j).name = arg2(i,j);

obj(i,j).dep=obj(i,j);

obj(i,j).grad=1;

obj(i,j).OutDataTypeStr='double';

obj(i,j).rel\_obj={};

obj(i,j).rel\_mat=[];

obj(i,j).img=0;

obj(i,j).r= obj(i,j);

## Syntax 8: unc( value , CX )

The input arguments are:

value: nxm double array corresponding to the property *value*

CX: (nxm)x(nxm) double array which corresponds to the covariance matrix of the elements of the array *value*.

The above restrictions on the input arguments are reflected in the conditional statement:

if isa(arg1,'double')&& isa(arg2, 'double') && size(arg2,1)==size(arg2,2) && size(arg2,1)==size(arg1,1)\*size(arg1,2)

The covariance matrix must be symmetric and positive semi-definite (PSD), which is checked as follows:

[~,p] = chol(arg2);

if (all(all(arg2 == transpose(arg2))) && (p==0))

Otherwise an error message will be displayed.

error('Covariance matrix must be symmetric and positive semidefinite')

First, a 1-row array arg1\_row is created from the array arg1:

arg1\_row=arg1;

arg1\_row=arg1\_row';

arg1\_row=arg1\_row(:);

Another array diag\_arg2 containing the elements on the diagonal of arg2 (that are in fact variances) is also created:

diag\_arg2= diag(arg2)';

Then the properties of the created uncertainty object are assigned accordingly to the input arguments, either explicitly or implicitly. Those properties that are not assigned are set to default values:

obj(i).value = arg1\_row(i);

obj(i).std\_unc = sqrt(diag\_arg2(c));

obj(i).dep=obj(i);

obj(i).grad=1;

obj(i).OutDataTypeStr='double';

obj(i).img=0;

obj(i).r= obj(i);

Covariance and correlation values are assigned in exactly the same way used in syntax 3.

## Syntax 9: unc( unc\_objects , CX )

The input arguments are:

*unc\_objects*: 1xn array of real uncertainty objects

*CX*: (nxm)x(nxm) double array which corresponds to the covariance matrix of the elements of the array *unc\_objects*.

The above restrictions on the input arguments are reflected in the conditional statement:

if isa(arg1,'unc')&& isa(arg2, 'double') && size(arg2,1)==size(arg1,2) && size(arg2,2)==size(arg1,2)

Covariance and correlation values are assigned in exactly the same way used in syntax 3.

## Syntax 10: unc( value )

The input argument is:

*value*: nxm double array corresponding to the property *value.* That is:

if isa(arg1,'double') && isreal(arg1)

In this case, the value property of the created uncertainty object is assigned accordingly to the input argument. Those properties that are not assigned are set to default values:

obj(i,j).value = arg1(i,j);

obj(i,j).std\_unc = 0;

obj(i,j).name = {};

obj(i,j).dep=[];

obj(i,j).grad=[];

obj(i,j).OutDataTypeStr='double';

obj(i,j).rel\_obj={};

obj(i,j).rel\_mat=[];

obj(i,j).img=0;

obj(i,j).r= obj(i,j);

When the elements of the input array are complex, the function complex() for uncertainty objects is called.

## Syntax 11: unc( unc\_object )

The input argument is:

*unc\_object*: 1x1 array of uncertainty objects. That is:

if isa(arg1,'unc')

In this case, the uncertainty objects contained in the array unc\_object are directly copied into the created array obj of uncertainty objects:

obj = arg1;

## Syntax 12: unc( )

Empty input argument. That is:

if (nargin==0)

The properties of the created uncertainty object are set to default values:

obj.value = 0;

obj.std\_unc = 0;

obj.name = {};

obj.dep=obj;

obj.grad=[];

obj.OutDataTypeStr='double';

obj.rel\_obj={};

obj.rel\_mat=[];

obj.img= [];

obj.r= [];

# Operator Overloading

## function obj = plus(obj1,obj2)

This method implements the overloading of the addition operation defined by the operator .+.

The two objects obj1 and obj2 must have compatible sizes. That is, for every dimension, the dimension sizes of the objects obj1 and obj2 are either the same or one of them is 1.

In the case the dimension size of obj1 is equal to 1, i.e.:

if length(obj1)==1

the method plusscal() is called to implement the addition operation with input arguments (obj1,obj2(i,j)):

for i = 1:size(obj2,1)

for j = 1:size(obj2,2)

obj(i,j)= plusscal(obj1,obj2(i,j));

end

end

In the case the dimension size of obj2 is equal to 1, i.e.:

if length(obj2)==1

the method plusscal() is called to implement the addition operation with input arguments (obj1(i,j),obj2):

for i = 1:size(obj1,1)

for j = 1:size(obj1,2)

obj(i,j)= plusscal(obj1(i,j),obj2);

end

end

In the case the objects obj1 and obj2 have the same dimension size, i.e.:

size(obj1)==size(obj2)

the method plusscal() is called to implement the addition operation with input arguments (obj1(i,j),obj2(i,j)):

for i=1:size(obj1,1)

for j=1:size(obj1,2)

obj(i,j)= plusscal(obj1(i,j),obj2(i,j));

end

end

Similarly as explained before, it is performed the addition operation when one of the inputs is a matrix and the other is a column vector with the same number of rows or a row vector with the same number of columns. For example, if obj1 is a matrix with dimension [m x n] and obj2 is a [m x 1] column vector, the addition operation is implemented by

elseif (iscolumn(obj2) && ismatrix(obj1) && length(obj2)==size(obj1,1))

obj(size(obj1,1),size(obj1,2))=unc;

for i=1:size(obj1,1)

for j=1:size(obj1,2)

obj(i,j)= plusscal(obj2(i),obj1(i,j));

end

end

The addition operation between a row vector and a column vector is also considered. If the objects obj1 and obj2 do not have compatible size, then an error message is displayed:

error('Invalid operation. Matrix dimension must agree')

## function obj = minus(obj1,obj2)

This method implements the overloading of the subtraction operation defined by the operator .-.

The two objects obj1 and obj2 must have compatible sizes. That is, for every dimension, the dimension sizes of the objects obj1 and obj2 are either the same or one of them is 1.

In the case the dimension size of obj1 is equal to 1, i.e.:

if length(obj1)==1

the method minusscal() is called to implement the subtraction operation with input arguments (obj1,obj2(i,j)):

for i = 1:size(obj2,1)

for j = 1:size(obj2,2)

obj(i,j)= minusscal(obj1,obj2(i,j));

end

end

In the case the dimension size of obj2 is equal to 1, i.e.:

if length(obj2)==1

the method minusscal() is called to implement the subtraction operation with input arguments (obj1(i,j),obj2):

for i = 1:size(obj1,1)

for j = 1:size(obj1,2)

obj(i,j)= minusscal(obj1(i,j),obj2);

end

end

In the case the objects obj1 and obj2 have the same dimension size, i.e.:

size(obj1)==size(obj2)

the method minusscal() is called to implement the subtraction operation with input arguments (obj1(i,j),obj2(i,j)):

for i=1:size(obj1,1)

for j=1:size(obj1,2)

obj(i,j)= minusscal(obj1(i,j),obj2(i,j));

end

end

Similarly, the subtraction operation when one of the inputs is a matrix and the other is a column vector with the same number of rows or a row vector with the same number of columns. For example, if obj1 is a [m x 1] column vector and obj2 is a matrix with dimension [m x n], the subtraction operation is implemented by

elseif (iscolumn(obj1) && ismatrix(obj2) && length(obj1)==size(obj2,1))

obj(size(obj2,1),size(obj2,2))=unc;

for i=1:size(obj2,1)

for j=1:size(obj2,2)

obj(i,j)= minusscal(obj1(i),obj2(i,j));

end

end

The subtraction operation between a row vector and a column vector is also considered. If the objects obj1 and obj2 do not have compatible size, then an error message is displayed:

error('Invalid operation. Matrix dimension must agree')

## function obj = mtimes(obj1,obj2)

This method implements the overloading of the matrix multiplication operator for two arrays obj1 and obj2 of uncertainty objects.

If the input argument obj1, instead of being an uncertainty object, is a complex number, that is:

if isa(obj1,'unc')==0 & (~isreal(obj1))

then obj1 is transformed into a complex uncertainty object:

obj1=complex(unc(real(obj1)), unc(imag(obj1)) );

On the other hand, if the input argument obj2, instead of being an uncertainty object, is a complex number, that is:

if isa(obj2,'unc')==0 & imag(obj2)~=0

then obj2 is transformed into a complex uncertainty object:

obj1=complex( unc(real(obj2)) , unc(imag(obj2)) );

If the total number of elements in obj1 or in obj2 is equal to one then multiplication is performed as follows:

if (length(obj1(:))==1)||(length(obj2(:))==1)

obj=obj1.\*obj2;

Otherwise, for matrix multiplication of obj1 and obj2 to be valid, the number of columns of obj1 must be equal to the number of rows of obj2:

size(obj1,2)==size(obj2,1)

In this case, the methods multscal() and plusscal() are called in the algorithm implementation of matrix multiplication using the Row-Column expansion rule:

for i=1:size(obj1,1)

for j=1:size(obj2,2)

for k=1:size(obj2,1)

temp=multscal(obj1(i,k),obj2(k,j));

obj(i,j)=plusscal(obj(i,j),temp);

end

end

end

If the number of columns of obj1 is not equal to the number of rows of obj2, then an error message is displayed:

error('Invalid operation. Matrix dimension must agree')

## function obj=rdivide(obj1,obj2)

This method implements the overloading of the right element-wise division operator (./) for two arrays obj1 and obj2 of uncertainty objects with compatible size.

The arrays obj1 and obj2 can be of the same size, that is:

size(obj1)==size(obj2)

or one of them can contain only one element,

length(obj2(:))==1

The other cases are when one of the inputs is a matrix and the other is a column vector with the same number of rows or a row vector with the same number of columns. Also when obj1 is a column vector and obj2 a row vector or viceversa, i.e.

isrow(obj1) && iscolumn(obj2)

Otherwise an error message is displayed:

error('Matrices must be of equal dimension')

If obj1 and obj2 have the same size, then the elements obj(i,j) can be found by simply dividing obj1(i,j) by obj2(i,j):

obj(i,j)=obj1(i,j)/obj2(i,j);

If the array obj2 is a column vector and array obj1 is a matrix, then the resulting object obj can be found by:

(iscolumn(obj2) && ismatrix(obj1) && length(obj2)==size(obj1,1))

obj=unc(zeros(size(obj1)));

for i=1:size(obj1,1)

for j=1:size(obj1,2)

obj(i,j)=obj1(i,j)/obj2(i);

end

end

## function obj=ldivide(obj1,obj2)

This method implements the overloading of the left element-wise division operator (.\) for two arrays obj1 and obj2 of uncertainty objects with compatible size.

Using the fact that the element-wise operators (./) and (.\) are related to each other by the equation A./B = B.\A, then

obj = rdivide(obj2,obj1);

## function obj =mldivide(obj1,obj2)

This method implements the overloading of matrix left division for two arrays obj1 and obj2 of uncertainty objects.

If both inputs are scalars, then

if isscalar(obj1) && isscalar(obj2)

obj = obj2/obj1;

If obj1 is an scalar and obj2 is an array, then obj1\obj2 is equivalent to the element-wise operation (.\), that is

elseif isscalar(obj1) && (ismatrix(obj2)|| isvector(obj2))

obj = unc(zeros(size(obj2)));

for i=1:size(obj2,1)

for j=1:size(obj2,2)

obj(i,j)= obj2(i,j)/obj1;

end

end

If obj1 is a square matrix and obj2 is a matrix with the same number of rows, then obj is a solution to the equation A\*x = B, where A=obj1 and B=obj2 respectively. This is implemented by first taking the inverse of the array obj1:

obj3=inv(obj1);

and then use the method mtimes() to perform the multiplication of the resulting array obj3 with the array obj2:

obj = mtimes(obj3,obj2);

If obj1 is a rectangular matrix and obj2 is a matrix with the same number of rows, then obj returns a least-squares solution to the system of equation A\*x = B, i.e. x = inv(A’\*A)\*A’\*B

elseif ismatrix(obj1) && (size(obj1,1)== size(obj2,1))

obj = inv(obj1'\*obj1)\*obj1'\*obj2;

## function obj =mrdivide(obj1,obj2)

This method implements the overloading of the matrix right division of two uncertainty objects obj1 and obj2 and returns another uncertainty object obj.

1. Case of the division involving at least one complex uncertainty object:

if (any(imag(obj1)~=0) && any(imag(obj2)~=0)) || (any(imag(obj1)==0) && any(imag(obj2)~=0)) || (any(imag(obj1)~=0) && any(imag(obj2)==0))

The objects obj1 and obj2 are first split into their respective real and imaginary parts:

obj1\_1= unc(real(obj1));

obj2\_1= unc(real(obj2));

obj1\_2=unc(imag(obj1));

obj2\_2=unc(imag(obj2));

Let us consider two complex-valued quantities *z* and *w*:

Where:

The variables *x,* *y*, *u*, and *v* will represent the uncertainty objects *obj1\_1*, *obj1\_2*, *obj2\_1*, and *obj2\_2* respectively. So we will have the following correspondences:

x=obj1\_1.value

x’= obj1\_1.grad

y=obj1\_2.value

y’= obj1\_2.grad

u=obj2\_1.value

u’= obj2\_1.grad

v=obj2\_2.value

v’= obj2\_2.grad

Consider the following condition on the complex quantity *w*:

Taking the ratio of the two complex-valued quantities *z* and *w*:

Since, the complex conjugate is also different from zero ( and

Hence expression (1) can also be written as:

Where and are respectively the real and imaginary parts of the resulting ratio.

Therefore, the *value* property of the real and imaginary parts of obj is obtained by applying equation (2):

a= (obj1\_1.value \* obj2\_1.value + obj1\_2.value \* obj2\_2.value )/( obj2\_1.value^2 + obj2\_2.value^2);

b= (-obj1\_1.value \* obj2\_2.value + obj2\_1.value \* obj1\_2.value )/( obj2\_1.value^2 + obj2\_2.value^2);

obj = complex( unc( a , 0 ) ,unc( b , 0 ) );

The dep and std\_unc properties are obtained in the same way as done in plusscal() method.

The grad property is also computed in the same way as done in plusscal() method with the exception of the Grad array.

The gradient of the real part is:

Hence, the *Grad* array for the real part is given by:

Grad = [ obj1\_1.grad \* obj2\_1.value/(obj2\_1.value^2 + obj2\_2.value^2) ,...

obj1\_2.grad \* obj2\_2.value/(obj2\_1.value^2 + obj2\_2.value^2)...

, a/(obj2\_1.value^2 + obj2\_2.value^2)^2 , b/(obj2\_1.value^2 +obj2\_2.value^2)^2 ];

Where a and b are defined by:

a = obj2\_1.grad \* obj1\_1.value \* (obj2\_1.value^2 + obj2\_2.value^2)-...

(obj1\_1.value \* obj2\_1.value + obj1\_2.value \*obj2\_2.value)\*2\*obj2\_1.value\*obj2\_1.grad;

b = obj2\_2.grad \* obj1\_2.value \* (obj2\_1.value^2 + obj2\_2.value^2)-...

(obj1\_1.value \* obj2\_1.value + obj1\_2.value \*obj2\_2.value)\*2\*obj2\_2.value\*obj2\_2.grad;

The gradient of the imaginary part is:

Hence, the *Grad* array for the imaginary part is given by:

Grad = [ -obj1\_1.grad \* obj2\_2.value /(obj2\_1.value^2 + obj2\_2.value^2)...

,obj1\_2.grad \* obj2\_1.value/(obj2\_1.value^2 + obj2\_2.value^2)...

, b/(obj2\_1.value^2 + obj2\_2.value^2)^2 , a/(obj2\_1.value^2 +obj2\_2.value^2)^2 ];

Where a and b are defined by:

a = -obj2\_2.grad \* obj1\_1.value \* (obj2\_2.value^2 + obj2\_1.value^2)-...

(-obj1\_1.value \* obj2\_2.value + obj1\_2.value \* obj2\_1.value)\*2\*obj2\_2.value\*obj2\_2.grad;

b = obj2\_1.grad \* obj1\_2.value \* (obj2\_2.value^2 + obj2\_1.value^2)-...

(-obj1\_1.value \* obj2\_2.value + obj1\_2.value \* obj2\_1.value)\*2\*obj2\_1.value\*obj2\_1.grad;

Finally, the object obj can be returned:

obj = complex(obj\_X , obj\_Y);

1. Case of the division involving only real uncertainty objects and both are scalar:

if any(imag(obj1)==0)) & any(imag(obj2)==0) & isscalar(obj1) & isscalar(obj2)

In this case, the division is performed as follows:

obj1 / obj2 = obj1 \* (1/obj2)

So, first the object obj2 is inverted by calling the method inv() and the product obj1 \* obj3 is computed by :

obj = obj1\*inv(obj2);

If obj2 is an scalar and obj1 is an array, then obj1/obj2 is equivalent to the element-wise operation (./), then

elseif isscalar(obj2) && (ismatrix(obj1)|| isvector(obj1))

obj = unc(zeros(size(obj1)));

for i=1:size(obj1,1)

for j=1:size(obj1,2)

obj(i,j)= obj1(i,j)\*inv(obj2);

end

end

If obj2 is a square matrix and obj1 is a matrix with the same number of columns, then obj is a solution to the equation x\*A = B, where B=obj1 and A=obj2 respectively. This is implemented by first taking the inverse of the array obj2:

obj3=inv(obj2);

and then use the method mtimes() to perform the multiplication of the resulting array obj3 with the array obj2:

obj = mtimes(obj1,obj3);

If obj2 is a rectangular matrix and obj1 is a matrix with the same number of columns, then obj returns a least-squares solution to the system of equation x\*A = B, i.e. x = B\*A’\*inv(A\*A’)

elseif ismatrix(obj2) &&(size(obj2,2)== size(obj1,2))

obj = obj1\*obj2'\*inv(obj2\*obj2');

When none of the cases before is fulfilled, an error message is displayed

error('Matrix dimension must agree');

## function obj=times(obj1,obj2)

This method implements the overloading of the element-wise multiplication operator called for the syntax ' .\*'.

First, it is checked if the inputs are complex. In the case they are not uncertainty objects, the method complex() is used

if (~isreal(obj1)) && (~isa(obj1,'unc'))

obj1 = complex(unc(real(obj1)),unc(imag(obj1)));

elseif (~isreal(obj2)) && (~isa(obj2,'unc'))

obj2 = complex(unc(real(obj2)),unc(imag(obj2)));

then, it is verified if any of the inputs is not an uncertainty object

elseif (~isa(obj1,'unc'))|| (~isa(obj2,'unc'))

obj1=unc(obj1);

obj2=unc(obj2);

end

The algorithm is implemented in such a way that, object obj1 is always the input with size 1, if one of the inputs is an scalar, then:

if length(obj2(:))==1

obh=obj1;

obj1=obj2;

obj2=obh;

end

In this case, the variable issc is set to logic 1 (i.e. *true* ) to indicate that one of the input arguments ( which is in this case always forced to be obj1) is of size 1:

if length(obj1)==1

issc=true;

end

The elements obj(i,j) are then obtained using the method multscal() called for input arguments obj1 and obj2(i,j):

obj(i,j)= multscal(obj1,obj2(i,j));

On the other hand, if the two arrays obj1 and obj2 are of the same size, then the elements obj(i,j) are obtained using the method multscal() called for input arguments obj1(i,j) and obj2(i,j):

obj(i,j)= multscal(obj1(i,j),obj2(i,j));

The cases when one of the inputs is a matrix and the other is a column vector with the same number of rows or a row vector with the same number of columns, as well as, when both inputs are vectors, is also considered. If the two arrays obj1 and obj2 are not with compatible size, then an error message is displayed:

error('Error using times Matrix dimension must agree')

## function obj=uplus(obj1)

This method implements the overloading of the unary plus operator called for the syntax '+obj1'.

The method times() is called to implement the product obj=1\*obj1:

obj=times(1,obj1);

## function obj=uminus(obj1)

This method implements the overloading of the unary minus operator called for the syntax '-obj1'.

The method times() is called to implement the product obj=-1\*obj1:

obj=times(-1,obj1);

## function obj = gt(obj1,obj2)

This method implements the overloading of the *greater than* operator called for the syntax 'obj1 > obj2'.

The inequality of two uncertainty objects *obj1* and *obj2* given by the *greater than* operator > :

obj1 > obj2

is an uncertainty object obj=unc() where:

: is the probability that a random sample taken from the Gaussian distribution of obj1 is greater than a random sample taken from the Gaussian distribution of obj2.

This can be implemented as shown below:

h=obj1-obj2;

obj.value=double(h.value>0);

obj.std\_unc=normcdf(h.value,0,h.std\_unc);

In the case one of the two terms of the inequality is a constant number, that is:

if (isa(obj1,'unc') && isa(obj2,'double')) || (isa(obj1,'double') && isa(obj2,'unc'))

then if the object obj2 is a constant number, and are found as follows:

obj.value=double(obj1.value>obj2);

obj.std\_unc=1-normcdf(obj2,obj1.value,obj1.std\_unc);

On the other hand, if the object obj1 is a constant number, then and are found as follows:

obj.value=double(obj2.value<obj1);

obj.std\_unc=normcdf(obj1,obj2.value,obj2.std\_unc);

## function obj = lt(obj1,obj2)

This method implements the overloading of the *less than* operator called for the syntax 'obj1 < obj2'.

The inequality of two uncertainty objects *obj1* and *obj2* given by the *less than* operator < :

obj1 < obj2

is an uncertainty object obj=unc() where:

: is the probability that a random sample taken from the Gaussian distribution of obj1 is less than a random sample taken from the Gaussian distribution of obj2.

This can be implemented as shown below:

h=obj1-obj2;

obj.value=double(h.value<0);

obj.std\_unc=normcdf(-h.value,0,h.std\_unc);

In the case one of the two terms of the inequality is a constant number, that is:

if (isa(obj1,'unc') && isa(obj2,'double')) || (isa(obj1,'double') && isa(obj2,'unc'))

then if the object obj2 is a constant number, then and are found as follows:

obj.value=double(obj1.value<obj2);

obj.std\_unc=normcdf(obj2,obj1.value,obj1.std\_unc);

On the other hand, if the object obj1 is a constant number, then and are found as follows:

obj.value=double(obj2.value>obj1);

obj.std\_unc=1-normcdf(obj1,obj2.value,obj2.std\_unc);

## function obj = mpower(obj1,obj2)

This method implements the overloading of matrix power called for the syntax 'obj1 ^ obj2'.

1. The case where obj2 is a scalar and obj1 is a square matrix:

If obj2 is an integer greater than one, then the power is computed by repeated squaring. The general case is implemented as follows:

if isa(obj2,'double') && ((abs(round(obj2)-obj2))<eps\*2)

obj=unc(obj1);

for i=2:int16(obj2)

obj=obj\*obj1;

end

1. If obj2 is a square matrix and obj1 is a scalar, then the resulting object obj can be computed using eigenvalues and eigenvectors. This case is not implemented in the current version of Uncertainty Toolbox. Hence an error message is displayed:

error('Sorry - this operation is currently not supported')

Notice that case where both obj1 and obj2 are matrices is an error.

## function obj = realpow(obj1,obj2)

This method implements the overloading of element-by-element powers.

If the arrays obj1 and obj2 do not have the same size, then an error message is displayed:

error('Error using realpow. Matrix dimensions must agree.')

If both obj1 and obj2 are both matrices of the same size, i.e.:

if ismatrix(obj1) || ismatrix(obj2)

then the elements obj(i,:) are found using the method power() called for input arguments obj1(i,:)and obj2(i,:):

obj(i,:)= power(obj1(i,:),obj2(i,:));

If the resulting value and std\_unc properties are complex numbers, i.e.:

if ~isreal(obj(i).value) || ~isreal(obj(i).std\_unc)

then an error message is displayed:

error('Error using realpow. Realpow produced complex result.')

In the case, object obj1 is of size 1 and obj2 is of size greater than 1, i.e.:

if isscalar(obj1) && ~isscalar(obj2)

then the elements obj(i,:) are found using the method power() called for input arguments obj1 and obj2(i,:):

obj(i,:)=power(obj1,obj2(i,:));

If the resulting value and std\_unc properties are complex numbers, i.e.:

if ~isreal(obj(i).value) || ~isreal(obj(i).std\_unc)

then an error message is displayed:

error('Error using realpow. Realpow produced complex result.')

In the case, object obj2 is of size 1 and obj1 is of size greater than 1, i.e.:

if ~isscalar(obj1) && isscalar(obj2)

then the elements obj(i,:) are found using the method power() called for input arguments obj1(i,:) and obj2:

obj(i,:)= power(obj1(i,:),obj2);

If the resulting value and std\_unc properties are complex numbers, i.e.:

if ~isreal(obj(i).value) || ~isreal(obj(i).std\_unc)

then an error message is displayed:

error('Error using realpow. Realpow produced complex result.')

## function obj = power(obj1,obj2)

This method implements the overloading of the element-wise power operator for two arrays obj1 and obj2 of uncertainty objects defined as obj1.^obj2.

If obj1 and obj2 have the same size or if the length of obj2 is equal to 1, i.e.:

if min(size(obj1) == size(obj2)) || length(obj2)==1

then the element-wise power can be implemented. Otherwise, an error message is displayed:

error('Error using .^ Matrix dimensions must agree.')

In the case of a real uncertainty object obj1, the element-wise power is implemented as follows:

obj = exp(obj2.\* log(obj1));

where the exp() and log() methods are the overloaded versions of the mathematical exponential and logarithmic functions respectively.

In the case of a complex uncertainty object obj1, i.e.:

if imag(obj1)~=0

first, obj1 is split into its real and imaginary parts:

obj2\_X = real(obj2);

obj2\_Y = imag(obj2);

Then, based on de Moivre's formula for complex numbers, the element-wise power is implemented as follows:

obj=exp(obj2\_X .\* log(obj1)) .\* ( cos(obj2\_Y) + 1i \* sin(obj2\_Y));

# Function Overloading

## function disp(obj)

This method is used to display uncertainty objects by calling the static method GenerateDispStr(). The syntax used in a static method call is done by writing the name of the class object followed by a dot then the name of the static method:

obj.GenerateDispStr(obj(i,j).value,obj(i,j).std\_unc)

where obj is an uncertainty object with properties *value* and *std\_unc*.

In the case of a real uncertainty object, i.e.:

if imag(obj(i,j))==0

the object obj is displayed directly with the use of the function fprintf():

fprintf([' \t' , obj.GenerateDispStr(obj(i,j).value,obj(i,j).std\_unc)]);

In the case of a complex uncertainty object, i.e.:

if imag(obj(i,j))~=0

the object obj is first split into its real and imaginary parts:

obj\_X(i,j) = real(obj(i,j));

obj\_Y(i,j)= imag(obj(i,j));

then depending on the sign of the value property of the imaginary part obj\_Y(i,j), an appropriate string is displayed.

If the value property of the imaginary part obj\_Y(i,j)is positive, then the fprintf()function is called with the following arguments:

fprintf([' \t', obj\_X.GenerateDispStr(obj\_X(i,j).value,obj\_X(i,j).std\_unc) ,' + '...

obj\_Y.GenerateDispStr(obj\_Y(i,j).value,obj\_Y(i,j).std\_unc) , ' \* i ']);

On the other hand, if the value property of the imaginary part obj\_Y(i,j)is negative, then the fprintf()function is called with the following arguments:

fprintf([' \t', obj\_X.GenerateDispStr(obj\_X(i,j).value,obj\_X(i,j).std\_unc) ,' - '...

obj\_Y.GenerateDispStr(abs(obj\_Y(i,j).value),obj\_Y(i,j).std\_unc) , ' \* i ']);

## function obj = sqrt(obj1)

This method implements the overloading of the sqrt() function by simply calling the method power(obj1,obj2) with the second input argument equal to 0.5 :

obj = power(obj1,0.5);

## function obj = atan2(obj1,obj2)

This method implements the overloading of the four-quadrant inverse tangent function for two real uncertainty objects obj1 and obj2.

The value property of the resulting object is found by directly using the non-overloaded version of the function atan2():

obj=unc( atan2( gmv(obj1) , gmv(obj2) ), zeros(size(obj1)) );

The grad property of obj is found in exactly the same way used in the method plusscal() with the exception of the Grad array which is given by:

Grad = [ dobj\_1(i,j).\* obj1(i,j).grad , dobj\_2(i,j).\* obj2(i,j).grad ] ;

Where dobj\_1 and dobj\_2 are the partial derivatives of the atan2() function given by:

dobj\_2 = gmv(obj1)./ (gmv(obj1).^2 + gmv(obj2).^2);

dobj\_1 = -gmv(obj2)./ (gmv(obj1).^2 + gmv(obj2).^2);

Once the grad property is known, the std\_unc property of obj can be calculated by calling the method eval\_std\_unc():

obj= eval\_std\_unc(obj);

## function obj = prod(obj1)

This method implements the overloading of the function prod().

It returns the product of the elements of the object obj1.

If obj1 is a vector of uncertainty objects, that is:

if size(obj1,1)==1

then the object obj is a single uncertainty object equal to the product of the elements of obj1:

for i= 1:length(obj1)

obj = obj \* obj1(i);

end

If obj1 is a matrix of uncertainty objects, then the object obj is a row vector of uncertainty objects with the product over each column:

for j=1: size(obj1,2)

obj3=unc(1);

for i=1:size(obj1,1)

obj3 = obj3 \* obj1(i,j);

end

obj(j) = obj3;

end

## function obj = diag(obj1)

This method returns the diagonal elements of the matrix obj1 of uncertainty objects.

If r1 and c1 are the number of rows and columns of obj1 respectively:

[r1,c1] = size(obj1);

Then an error message is displayed in case the matrix obj1 is not square:

if r1 ~= c1

error('Matrix must be Square.')

end

If the matrix obj1 is square, then only the elements obj1(i,j) with equal indices i and j are copied into the object obj:

for i=1:r1

for j=1:c1

if i==j

obj(i) = obj1(i,j);

end

end

end

## function obj = trace(obj1)

This method returns the sum of the diagonal elements of the matrix obj1 of uncertainty objects.

If r1 and c1 are the number of rows and columns of obj1 respectively:

[r1,c1] = size(obj1);

Then an error message is displayed in case the matrix obj1 is not square:

if r1 ~= c1

error('Matrix must be Square.')

end

If the matrix obj1 is square, then the diagonal elements of obj1 are taken using the method diag() and then summed up using the method sum():

obj = sum(diag(obj1));

## function obj = ctranspose(obj1)

This method overloads the complex conjugate transpose function.

The output uncertainty objects obj is obtained in two steps:

First, taking the transpose of each element of obj1 :

for i=1:r1

for j=1:c1

obj2(j,i)=obj1(i,j);

end

end

where [r1,c1] = size(obj1).

Then, taking the complex conjugate of the resulting object:

obj = conj(obj2);

## function obj = max(obj1)

This method takes as input argument an array of uncertainty objects obj1 and returns the uncertainty object obj whose *value* property is the largest.

First, the maximum of the value properties of obj1 is found using the non-overloaded function max() (since the value properties are of type *double*). Then all the value properties of obj1 are compared to the found maximum using the If statement:

if obj1(i).value==max([obj1.value])

Whenever an element of obj1 whose value property is found to be equal to the maximum, it is stored in in the array obj:

idx=0;

for i=1:length(obj1)

if obj1(i).value==max([obj1.value])

idx = idx+1;

obj(idx)=obj1(i);

end

end

Since the array obj might contain several elements that satisfy the above If statement, the unique() function is used to return the non-repeated elements:

obj= unique(obj);

Notice that the unique() function acts only on the value property and not on the std\_unc property.

## function obj = min(obj1)

This method takes as input argument an array of uncertainty objects obj1 and returns the uncertainty object obj whose *value* property is the smallest.

First, the minimum of the *value* properties of obj1 is found using the non-overloaded function min() (since the value properties are of type *double*). Then all the value properties of obj1 are compared to the found minimum using the If statement:

if obj1(i).value==min([obj1.value])

Whenever an element of obj1 whose value property is found to be equal to the minimum, it is stored in in the array obj:

idx=0;

for i=1:length(obj1)

if obj1(i).value==min([obj1.value])

idx = idx+1;

obj(idx)=obj1(i);

end

end

Since the array obj might contain several elements that satisfy the above If statement, the unique() function is used to return the non-repeated elements:

obj= unique(obj);

Notice that the unique() function acts only on the *value* property and not on the std\_unc property.

## function obj = sum(obj1)

This method returns the sum of the elements of the array of uncertainty objects obj1. It is implemented by simply adding up the elements obj1(i,j) where i=1:size(obj1,1) and j=1:size(obj1,2). The result is stored and returned in the uncertainty object obj:

obj=unc(0,0);

for i=1:size(obj1,1)

for j=1:size(obj1,2)

obj = obj +obj1(i,j);

end

end

## function obj = cumsum(obj1)

This method returns the cumulative sum of the elements of the array of uncertainty objects obj1.

First, the array obj1 is transformed into a one-dimensional array that has the following form:

[ 1st row of obj1 , 2nd row of obj1 , …, row of obj1 ]

In order to obtain the above form, the following steps are performed:

1. Find the transpose matrix W of obj1:

W=obj1';

1. Transform W into a one-dimensional column array Q:

Q=W(:);

1. Transpose the Q array in order to get the desired one-dimensional row array:

Q=Q';

Now, the array Q has the desired form:

[ 1st row of obj1 , 2nd row of obj1 , …, row of obj1 ]

Next, the positions of the elements of the array obj1 ,given by obj1(i,j), are mapped into their new positions in the array Q, given by Q(k). The mapping function has the form:

k= j + (i - 1) \* J

where J=size(obj1,2).

Finally, the cumulative sum at the position (i,j) is calculated by summing all the elements of the Q array from k=1 to j + (i - 1) \* J:

for i=1:size(obj1,1)

for j=1:size(obj1,2)

z=unc(0,0);

for k=1:(j+(i-1)\*J)

z = Q(k) + z;

end

obj(i,j)=z;

end

end

## function obj = inv(obj1)

This method returns the inverse matrix of the matrix obj1 of uncertainty objects.

The matrix obj1 must be square, otherwise an error message is displayed:

error('Error using inv. Matrix must be Square')

If the matrix obj1 is singular, i.e.:

if det(gmv(obj1))== 0

an error message is displayed:

error('Matrix is singular')

In the case, the matrix obj1 is rank deficient

if rank(gmv(obj1))<c1

where c1 is the number of columns of obj1, then a warning message is displayed:

error('Matrix is close to singular or badly scaled')

If obj1 is square and not singular, then the matrix inversion algorithm is applied.

## function [determ]= det(obj)

This method returns the determinant of the array obj of uncertainty objects.

The array obj must be a square matrix, otherwise an error message is displayed:

disp('Matrix needs to be square');

If the number of columns of obj is equal to 1, then the first element of obj is returned as the determinant:

determ =obj(1,1);

If obj is a 2x2 matrix, then the resulting determinant is given by:

determ=obj(1,1)\*obj(2,2)-obj(1,2)\*obj(2,1);

For nxn matrices with n > 2, the minors of M = obj are computed and then used for finding the determinant:

determ = determ + M(i,j)\*((-1)^(i+j))\*detM\_ij;

where M(i,j) represents the element of the row i and the column j and detM\_ij is the minor associated to it, which is the determinant of the reduced matrix obtained removing the row i and the column j = 1.

Whenever the size of the resulting square matrix is equal to 2x2, the minor is computed directly using the formula of the determinant for a 2x2 matrix, presented previously. On the other hand, if the resulting matrix is still not yet a 2x2 matrix, then the method det() is called until a 2x2 matrix is obtained. The determinant is computed following the procedure:

for i=1:n

Mi = M;

Mi(:,j) = [];

Mi(i,:) = [];

M\_ij = Mi;

detM\_ij = det(M\_ij);

determ = determ + M(i,j)\*((-1)^(i+j))\*detM\_ij;

M = obj;

end

## function Z = complex(x,y)

This method generates complex uncertainty objects Z out of real uncertainty objects x and y.

If x and y have the same size, i.e.:

if (r1==r2) && (c1==c2)

where [r1,c1]=size(x) and [r2,c2]=size(y), and if x and y are both real uncertainty objects, i.e.:

if (x(i,j).img==0) && (y(i,j).img==0)

then a complex uncertainty object Z(i,j)can be created:

Z(i,j).OutDataTypeStr='complex';

with real and imaginary parts given by:

Z(i,j).r = x(i,j);

Z(i,j).img= y(i,j);

The value and std\_unc properties are then assigned to Z(i,j):

Z(i,j).value= x(i,j).value + 1i \* y(i,j).value;

Z(i,j).std\_unc= sqrt( (x(i,j).std\_unc)^2 + (y(i,j).std\_unc)^2 );

If x and y are not both real uncertainty objects, then an error message is displayed:

error('Input arrays must be real objects.')

If x and y do not have the same size, then an error message is displayed:

error('Input arrays must have the same size.')

## function X=real(Z)

This method returns the real parts of the uncertainty objects contained in the array Z.

It takes the element Z(i,j)and returns its real part Z(i,j).r if this latter is non zero, and returns zero otherwise:

if Z(i,j).r ~=0

X(i,j)= Z(i,j).r;

else

X(i,j)=0;

end

## function Y=imag(Z)

This method returns the imaginary parts of the uncertainty objects contained in the array Z.

It takes the element Z(i,j)and returns its imaginary part Z(i,j).img if this latter is non zero, and returns zero otherwise:

if Z(i,j).img ~=0

Y(i,j)= Z(i,j).img;

else

Y(i,j)=0;

end

If the array Z contains both real and complex uncertainty objects and the first element of Z is a real uncertainty object, i.e.:

if (length(Z)>1) && ( Z(1,1).img == 0 )

then an error occurs arising from conversion from uncertainty class data type into double data type.

In order to handle this issue, the first time an element of Z ,with a non-zero imaginary part, is encountered, it is immediately swapped with the first element of the array (i.e. Z(1,1)). A counter c is then incremented to signal that an error handling routine has been executed. This is done by the following algorithm:

if Z(i,j).img ~=0

k=i;

l=j;

temp=Z(1,1);

Z(1,1)=Z(i,j);

Z(i,j)=temp;

j=size(Z,2); % force to halt the loop

i=size(Z,1); % force to halt the loop

c=1;

end

In the case the error handling routine has been executed, the inverse swapping operation is done:

if (c==1)

temp=Y(1,1);

Y(1,1)=Y(k,l);

Y(k,l)=temp;

end

## function obj=conj(Z)

This method returns the complex conjugate of the uncertainty objects contained in the array Z.

It takes the complex conjugate of each entry (i,j)of the array Z:

obj(i,j)= Z(i,j).real - 1i \* Z(i,j).img;

## function obj = abs(Z)

This method returns the magnitudes of the uncertainty objects contained in the array Z.

It takes the magnitude of each entry (i,j)of the array Z:

obj(i,j)= sqrt(Z(i,j).real^2 + Z(i,j).img^2);

## function obj = angle(Z)

This method returns the phase angles, in radians, of the uncertainty objects contained in the array Z.

It takes the phase angle of each entry (i,j) of the array Z:

obj(i,j)= atan2(Z(i,j).img, Z(i,j).real);

# Special Uncertainty Class Methods

## function value = gmv(obj)

This method returns the *value* property of the array obj of uncertainty objects.

It uses reshape(X,m,n) function that takes an array X as input and returns an mxn matrix whose elements are taken column-wise from X. m\*n must be equal to the number of elements in the array X.

The uncertainty object obj is an mxn matrix where m=1,2,… and n=1,2,….

First, an array of value properties is formed by [obj.value].

Then it is reshaped into a matrix of a size equal to size(obj,1)\*size(obj,2) as follows:

value=reshape([obj.value],size(obj,1),size(obj,2));

In the case of a complex uncertainty object ( i.e. imag(obj)~=0 ), the value property is a complex number:

value = obj\_X.value + 1i \* obj\_Y.value;

where obj\_X.value is the real part of the object obj and obj\_Y.value is the imaginary part of the object obj.

## function std\_unc = gmu(obj)

This method returns the *std\_unc* property of the array obj of uncertainty objects.

It uses reshape(X,m,n) function that takes an array X as input and returns an mxn matrix whose elements are taken column-wise from X. m\*n must be equal to the number of elements in the array X.

The uncertainty object obj is an mxn array where m=1,2,… and n=1,2,….

First, an array of std\_unc properties is formed by [obj.std\_unc].

Then it is reshaped into an array of a size equal to size(obj,1)\*size(obj,2) as follows:

std\_unc = reshape([obj.std\_unc],size(obj,1),size(obj,2));

In the case of a complex uncertainty object (i.e. imag(obj)~=0 ), the std\_unc property of a given complex quantity Z= X + i Y is calculated using the formula:

Hence the command:

std\_unc = sqrt((obj\_X.std\_unc)^2 + (obj\_Y.std\_unc)^2 );

where obj\_X.value is the real part of the object obj and obj\_Y.value is the imaginary part of the object obj.

## function names = gmn(obj)

This method returns the *name* property of the array obj of uncertainty objects.

The output *names* is a cell array of the same size as obj:

names=cell(size(obj));

If an uncertainty object obj(i,j) has no name assigned to it (i.e. isempty(obj(i,j).name returns logic 1), then the output ‘not named’ is returned:

if isempty(obj(i,j).name)

names(i,j)={'not named'};

else

names(i,j)= obj(i,j).name;

end

## function obj = eval\_std\_unc(obj1)

This method is solely used and called by the following methods:

plusscal(), minusscal(), multscal, and divscal().

It evaluates the *std\_unc* property of the uncertainty object *obj1* and assigns it to the returned uncertainty object *obj*.

The algorithm used implements the following mathematical equation for the calculation of the standard uncertainty:

Where:

: Output quantity in the measurement model . In the syntax *obj=eval\_std\_unc(obj1)*, *y* corresponds to the uncertainty object *obj1*. Hence, the variables correspond to the uncertainty objects obj1.dep(i) where i=1,2,…, n.

: Standard uncertainty of the variable . It corresponds to obj1.dep(i).std\_unc.

: The covariance between the variables and . It corresponds to covr( obj1.dep(i) , obj1.dep(j) ).

The partial derivatives are evaluated at the mean values of the variables respectively.

In the general case, where obj1 is an *mxn* array of uncertainty objects, the standard uncertainties correspond to:

obj1(i,k).dep(j).std\_unc

where:

i = 1,2,…,m where m=size(obj1,1) is the number of rows of obj1

k = 1,2,…,n where n=size(obj1,2) is the number of columns of obj1

j = 1,2,…,length(obj1(i).grad) where length(obj1(i).grad) is the number of gradient values of the element of obj1.

The values of the partial derivatives at the mean values are directly taken from the grad properties of obj1 as follows:

obj1(i,k).grad(j)

The algorithm is implemented as shown below:

obj=obj1;

for i=1:size(obj1,1)

for k=1:size(obj1,2)

% ------------------------------------------------

sum1=0;

sum2=0;

for j=1:length(obj1(i).grad)

for p=1:length(obj1(i).grad)

if j~=p

CX=covr( obj1(i,k).dep(j) , obj1(i,k).dep(p) );

sum1 = sum1 + obj1(i,k).grad(j) \* obj1(i,k).grad(p) \* CX;

else

sum2 = sum2 + (obj1(i,k).grad(j).\* …

[obj1(i,k).dep(j).std\_unc]).^2;

end

end

end

obj(i,k).std\_unc = sqrt(sum1 + sum2);

% ------------------------------------------------

end

end

The condition if j~=p corresponds to the condition in the double sum

## function [ obj ] = eval\_obj(str,obj1 )

This method is used in the overloading of common mathematical functions.

A given mathematical function with name str can be evaluated at the uncertainty object obj1 using the command eval\_obj(str,obj1).

To do so, first, the *value* property of the uncertainty object obj is computed by taking the mathematical function, designated by the input argument *str*, of the matrix of *value* properties returned by the method gmv(obj1). For example, if str=’sin’ then:

obj=unc(sin(gmv(obj1)),zeros(size(obj1)));

Now, in order to calculate the *std\_unc* property of obj, the gradient values of the mathematical function designated by *str* are needed.

This is done by first evaluating the gradient of the mathematical function designated by *str* at the mean value of obj1 given by obj1.value.

Computing the gradients requires derivatives computations. The derivatives of the following mathematical functions are explicitly implemented in the Uncertainty Toolbox:

sin, cos, tan, asin, acos, atan, sinh, cosh, tanh, asinh, acosh, atanh, cot, acot, coth, exp, log, and log10.

See table below for the list of implemented functions and their derivatives.

|  |  |  |
| --- | --- | --- |
| **Mathematical Function** | **Description** | **Derivative** |
| exp(x) | Exponential function | exp(x) |
| log(x) | Natural logarithm function |  |
| Log10(x) | Decimal logarithm function |  |
| sin(x) | Sine function | cos(x) |
| cos(x) | Cosine function | -sin(x) |
| tan(x) | Tangent function |  |

|  |  |  |
| --- | --- | --- |
| **Mathematical Function** | **Description** | **Derivative** |
| cot(x) | Cotangent function |  |
| asin(x) | Inverse sine function |  |
| acos(x) | Inverse cosine function |  |
| atan(x) | Inverse tangent function |  |
| acot(x) | Inverse cotangent function |  |
| sinh(x) | Hyperbolic sine function | cosh(x) |
| cosh(x) | Hyperbolic cosine function | sinh(x) |
| tanh(x) | Hyperbolic tangent function |  |
| coth(x) | Hyperbolic cotangent function |  |
| asinh(x) | Inverse hyperbolic sine function |  |
| acosh(x) | Inverse hyperbolic cosine function |  |
| atanh(x) | Inverse hyperbolic tangent function |  |

As an example, if str=’sin’, then the derivative function of sin(x) is cos(x). Therefore,

dobj = cos(gmv(obj1));

The Switch Statement is used as a selection control mechanism to take in the desired mathematical function *str*. This latter must be of class character, and hence it is entered enclosed between single quotes (e.g. ‘cos’).

After evaluating the gradient of the given mathematical function at obj1.value, the grad property of the uncertainty object obj is updated as follows:

The term is calculated inside the switch statement and referred to as dobj. Hence the command:

obj(i,j).grad = dobj(i,j).\*[obj1(i,j).grad];

The dep property of obj is updated by directly copying the dep property of the uncertainty object obj1:

obj(i,j).dep =obj1(i,j).dep;

After updating the grad property (i.e. the gradient values of obj), it is now possible to call the eval\_std\_unc() method in order to compute the std\_unc property of obj:

obj= eval\_std\_unc(obj);

In the case of complex uncertainty objects (i.e. imag(obj1)~=0), the complex object obj1 is split into two real uncertainty objects: its real part( obj1\_X = real(obj1) ) and its imaginary part ( obj1\_Y = imag(obj1) ). The overloading of a complex function f(z) where , amounts then to the overloading of a real function f(x+iy) where . The mathematical expressions of the complex functions implemented in the algorithms used in the Uncertainty Toolbox are listed in the table below.

|  |  |  |
| --- | --- | --- |
| **Complex Function of z= x + i y** | **Mathematical Expression** | **Description** |
| exp(z) |  | Complex exponential |
| log(z) |  | Complex natural logarithm |
| Log10(z) |  | Complex decimal logarithm |
| sin(z) |  | Complex sine |
| cos(z) |  | Complex cosine |
| tan(z) |  | Complex tangent |
| cot(z) |  | Complex cotangent |
| asin(z) |  | Inverse complex sine |
| acos(z) |  | Inverse complex cosine |
| atan(z) |  | Inverse complex tangent |
| acot(z) |  | Inverse complex cotangent |
| sinh(z) |  | Complex hyperbolic sine |
| cosh(z) |  | Complex hyperbolic cosine |

|  |  |  |
| --- | --- | --- |
| Complex Function of z= x + i y | Mathematical Expression | Description |
| tanh(z) |  | Complex hyperbolic tangent |
| coth(z) |  | Complex hyperbolic cotangent |
| asinh(z) |  | Inverse complex hyperbolic sine |
| acosh(z) |  | Inverse complex hyperbolic cosine |
| atanh(z) |  | Inverse complex hyperbolic tangent |

The following mathematical functions are overloaded in the current version of the Uncertainty Toolbox:

function obj=sin(obj1)

function obj=cos(obj1)

function obj = tan(obj1)

function obj=acos(obj1)

function obj=asin(obj1)

function obj=atan(obj1)

function obj = cosh(obj1)

function obj = sinh(obj1)

function obj = tanh(obj1)

function obj = acosh(obj1)

function obj = asinh(obj1)

function obj = atanh(obj1)

function obj = cot(obj1)

function obj = acot(obj1)

function obj = coth(obj1)

function obj = log(obj1)

function obj = log10(obj1)

function obj = exp(obj1)

The same overloading mechanism is used for all the above functions by calling the method eval\_obj(str,obj1). For example, the cosine function overloading is done as follows:

function obj=cos(obj1)

obj = eval\_obj('cos',obj1);

end

## function disp\_contribution(obj1)

Consider an uncertainty object *y* given as a function *f* of some other uncertainty objects x1, x2, …, xn:

y=f(x1, x2, …, xn)

The method *disp\_contribution(y)* displays the contributions of each of the objects x1, x2, …, xn to the uncertainty of y.

If obj1 is defined by the function then the uncertainty contribution of any given variable is given by:

The uncertainty contribution array is then defined as follows:

The above relationship is implemented using the following line of code:

contrib = [obj1.grad].\*[obj1.dep.std\_unc];

The absolute values of the elements of the above array contrib is then

sorted using the sort() function :

[ ValueArray , IndexArray ] = sort(abs(contrib));

Next, the array IndexArray is implemented to include only the indices whose values associated to it are greater than a minimum uncertainty contribution threshold defined as the maximum value times 1e-16:

IndexArray=IndexArray(ValueArray>max(ValueArray)\*1e-16);

The value 1e-16 can be changed as needed.

Finally, the fprintf() function is used to display the uncertainty contributions starting from largest contributor to smallest contributor. The display is done in a special frame as shown in the example of figure 2.

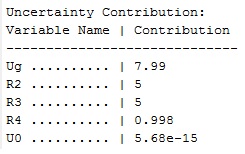


Figure 1: Displaying uncertainty contribution example

## function disp\_dep(obj)

This method displays the uncertainty objects on which the uncertainty object obj depends, as well as their names if assigned.

First, a cell array depend is created with the same dimension of the uncertainty object obj.

depend = cell(size(obj));

It will contain a list with the names and values on which each element of obj depends, considering that obj can be an nxm array.

The presence of assigned names is checked using the isempty() function and the display of the unc objects is done by calling the method disp().

Uncertainty objects without assigned names are displayed as: ‘not named’.

This is implemented by the following algorithm, for each element of obj:

if isempty(obj(i,j).dep(1,k).name)

dep\_name = 'not named';

dep\_value = obj.GenerateDispStr(obj(i,j).dep(1,k).value,…

obj(i,j).dep(1,k).std\_unc);

dep\_inf = [dep\_name,' = ' ,dep\_value];

else

dep\_name = obj(i,j).dep(1,k).name{1,1};

dep\_value = obj.GenerateDispStr(obj(i,j).dep(1,k).value,…

obj(i,j).dep(1,k).std\_unc);

dep\_inf = [dep\_name,' = ' ,dep\_value];

end

In order to display a unique list of dependencies, we evaluate if the current dependent object dep\_inf is repeated. If not, it is added to the cell array compdep of no repeated elements and the display of the unc objects is done by calling the method disp().

if isempty(compdep)| ~strcmp(dep\_inf,compdep)

compdep{end+1} = dep\_inf;

depend{i,j} = char(dep\_inf);

disp(depend{i,j})

fprintf('\n')

end

## function DispString = GenerateDispStr(value,std\_unc)

This is a static method that takes as input arguments the *value* and *std\_unc* properties and returns as output a string DispString. It has two purposes:

1. Set the number of decimal places to be displayed for value and std\_unc properties,
2. and generate a customized display string for the uncertainty objects.

This method is used by disp() and disp\_dep() methods to customize the display of uncertainty objects.

The variable Number\_of\_Digits\_to\_Display is set to be a global variable so that other functions can set the number of digits to be displayed.

The value and standard uncertainty properties to be displayed are given by the following formulas:

Where:

d = Number\_of\_Digits\_to\_Display – 1

And the value of *digit* is given by:

The function round(X) rounds the value of X to the nearest integer, whereas the function floor(X) rounds the value of X to the nearest integer towards minus infinity.

The eval() function is used to return the value of the expression enclosed in single quotation marks. The expression to be evaluated depends on the values of the variables *disp\_std\_unc*, *digit* and *d*.

Figure 3 shows how the display string is generated in the case digit<d in which the following strings are concatenated and then displayed:

disp\_value with (d - digit) decimal places(disp\_std\_unc with (d - digit) decimal places)

Where:

disp\_std\_unc = disp\_std\_unc\*10^(digit-d)

So that the final display string looks like this:

disp\_value(disp\_std\_unc)



Figure 2: Display string in the case digit < d

The case digit < 0 is treated in the same manner, with the exception of the expression of the number of decimal places used for the value property (in this case abs(digit)+d ) and also using the num2str() function for displaying the string std\_unc as shown below:

sprintf('[sprintf(''%%0.0%if'',disp\_value),''('',num2str(disp\_std\_unc),'')'']',abs(digit)+d)

In order to show the result in scientific notation in case there are more than two leading zeros in disp\_value and digit < -4, we use

sprintf('[sprintf(''%%0.0%ie'',disp\_value),''('',num2str(disp\_std\_unc),'')'']',abs(digit)- … abs(floor(log10(abs(value))))+d)

In the case above the expression of the number of decimal places used for the value property is changed to abs(digit)- abs(floor(log10(abs(value)))) +d .

The num2str() function converts the numerical-type variable disp\_std\_unc into a string-type variable.

In the case disp\_std\_unc=0, the following expression is evaluated:

'[num2str(disp\_value),''(0)'']'

The evaluation of the above expression leads to the following display string:

disp\_value(0)

If none of the above mentioned cases is selected then the following string will be displayed:

[num2str(disp\_value),'(',num2str(disp\_std\_unc),')']

Where:

disp\_std\_unc = disp\_std\_unc\*10^(digit-d)

## function obj = plusscal(obj1,obj2)

This is a method called when performing arithmetic operator overloading.

It takes as input arguments two uncertainty objects obj1 and obj2 and returns another uncertainty object obj.

1. Case of the sum of two real uncertainty objects:

if imag(obj1)==0 && imag(obj2)==0

The *value* property of obj is obtained by simply adding the value properties of obj1 and obj2:

obj = unc(obj1.value+obj2.value,0);

The dep property of obj is obtained by taking all the dependent unc class objects of obj1 and obj2 without any repetitions. This is performed using the unique() built-in Matlab function as follows:

[C,IA,IC]=unique([obj1.dep,obj2.dep]);

obj.dep=C;

The grad property of obj is obtained as follows:

First, the length of the grad property array will consist of the elements of C without repetition, so it can be created using the following line of code:

obj.grad=zeros(1,length(IA));

where IA is the array containing the indices of the elements of C in the array [ obj1.dep , obj2.dep ]. An alternative way to create the obj.grad array is:

obj.grad=zeros(1,length( obj.dep ));

Then all the grad property values of obj1 and obj2 are collected in one single array called Grad:

Grad = [obj1.grad,obj2.grad];

The IC array found in the line of code:

[C,IA,IC]=unique([obj1.dep,obj2.dep]);

contains the indices in C of the elements of the array [obj1.dep,obj2.dep]. Therefore, it will be used to locate the positions in the obj.grad array where to place each element of the Grad array. This is done in the following code:

for i=1:length(IC)

obj.grad(IC(i))=obj.grad(IC(i))+Grad(i);

end

where the elements of Grad array that share the same position in obj.grad array are added together.

The mechanism used by the above code is illustrated in figure 1, in which the two uncertainty objects obj1 and obj2 have the following properties:

The process is then carried out in the following way:

Place 23 in position 1 of obj.grad

Place 19 in position 2 of obj.grad

Place 7 in position 3 of obj.grad

Place 39 in position 1 of obj.grad. But since in position 1 is also placed 23, then addition is to be performed: 23 + 39 = 62. So position 3 will contain 62 (this is illustrated in figure 1).

Finally, the gradient property of obj is:

obj.grad = [ 62 19 7]

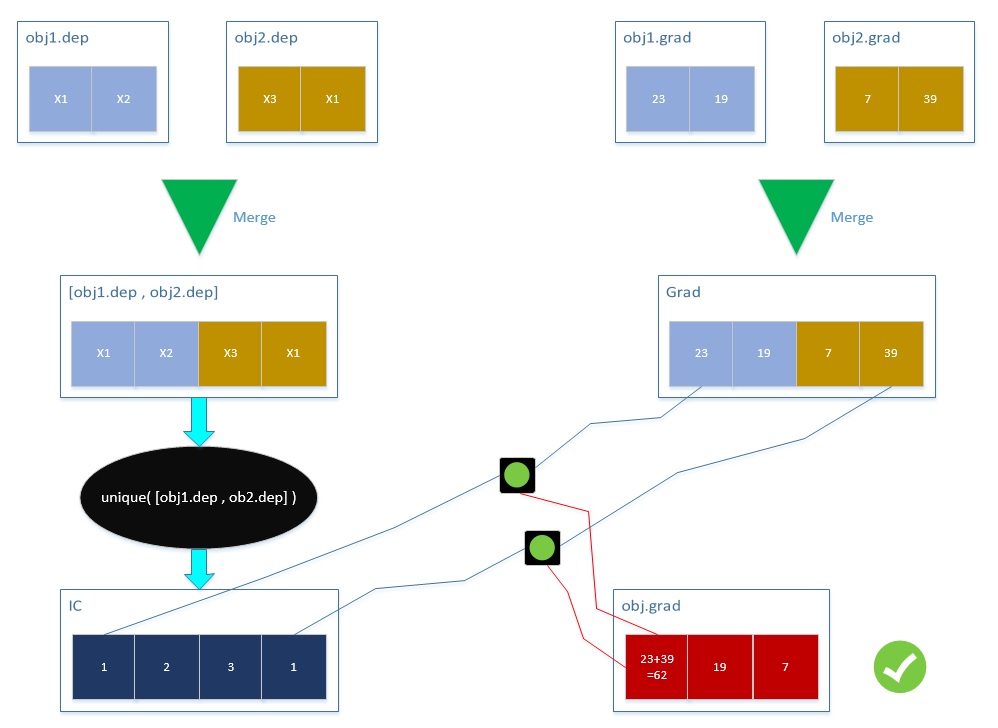


Figure 3: Computation of obj.grad

The last property ,to be computed, of the uncertainty object obj is the std\_unc property. This is done by calling the method eval\_std\_unc(obj):

s = eval\_std\_unc(obj);

obj.std\_unc = s.std\_unc;

1. Case of the sum involving at least one complex uncertainty object:

if (imag(obj1)~=0 && imag(obj2)~=0) || (imag(obj1)==0 && imag(obj2)~=0) || (imag(obj1)~=0 && imag(obj2)==0)

The objects obj1 and obj2 are first split into their respective real and imaginary parts:

obj1\_1= unc(real(obj1));

obj2\_1= unc(real(obj2));

obj1\_2=unc(imag(obj1));

obj2\_2=unc(imag(obj2));

The *value* property of the real part of obj is obtained by simply adding the value properties of obj1\_1 and obj2\_1, whereas the *value* property of the imaginary part of obj is obtained by simply adding the value properties of obj1\_2 and obj2\_2:

obj = complex( unc(obj1\_1.value+obj2\_1.value,0) ,unc(obj1\_2.value+obj2\_2.value,0) );

The object obj, in its turn, is split into its real and imaginary parts:

obj\_X = real(obj);

obj\_Y = imag(obj);

The dep property of the real part of obj is obtained by taking all the uncertainty objects, without any repetitions, on which obj1\_1 and obj2\_1 depend. This is performed using the unique() function as follows:

[C\_X,IA\_X,IC\_X]=unique([obj1\_1.dep,obj2\_1.dep]);

obj\_X.dep=C\_X;

The dep property of the imaginary part of obj is obtained by taking all the uncertainty objects, without any repetitions, on which obj1\_2 and obj2\_2 depend. This is performed using the unique() function as follows:

[C\_Y,IA\_Y,IC\_Y]=unique([obj1\_2.dep,obj2\_2.dep]);

obj\_Y.dep=C\_Y;

The *grad* property is calculated in the same way as done in the case of the sum of two real uncertainty objects with the exception of the *Grad* array which, in the current case, has a different mathematical form.

Let us consider two complex-valued quantities *z* and *w*:

Where:

The variables *x,* *y*, *u*, and *v* will represent the uncertainty objects *obj1\_1*, *obj1\_2*, *obj2\_1*, and *obj2\_2* respectively. So we will have the following correspondences:

x=obj1\_1.value

x’= obj1\_1.grad

y=obj1\_2.value

y’= obj1\_2.grad

u=obj2\_1.value

u’= obj2\_1.grad

v=obj2\_2.value

v’= obj2\_2.grad

Taking the sum of the two complex-valued quantities *z* and *w*:

If we consider and, then the sum can be written as:

The gradient of the real part is given by:

Hence, the *Grad* array for the real part of obj is:

Grad = [ obj1\_1.grad , 0 , obj2\_1.grad , 0 ] ;

To simplify numerical computations, the *Grad* array can be reduced to the following form:

Grad=[ obj1\_1.grad , obj2\_1.grad ] ;

The gradient of the imaginary part is given by:

Hence, the *Grad* array for the imaginary part of obj is:

Grad = [ 0 , obj1\_2.grad , 0 , obj2\_2.grad ] ;

To simplify numerical computations, the *Grad* array can be reduced to the following form:

Grad = [ obj1\_2.grad , obj2\_2.grad ];

The last property ,to be computed, of the uncertainty object obj is the std\_unc property. This is done by calling the method eval\_std\_unc(obj) for both real and imaginary parts of obj.

For the real part:

s\_X = eval\_std\_unc(obj\_X);

obj\_X.std\_unc = s\_X.std\_unc;

For the imaginary part:

s\_Y = eval\_std\_unc(obj\_Y);

obj\_Y.std\_unc = s\_Y.std\_unc;

Finally, the object obj can be returned:

obj = complex(obj\_X , obj\_Y);

## function obj = minusscal(obj1,obj2)

This is a method called when performing arithmetic operator overloading.

It takes as input arguments two unc class objects obj1 and obj2 and returns a unc class object obj.

1. Case of the subtraction of two real uncertainty objects:

if imag(obj1)==0 && imag(obj2)==0

The value property of obj is obtained by simply subtracting the value property of obj2 from the value property of obj1:

obj = unc(obj1.value-obj2.value,0);

The dep and std\_unc properties are obtained in the same way as done in plusscal() method.

The grad property is also computed in the same way as done in plusscal() method with the exception of the Grad array which is formed in the following way:

Grad=[obj1.grad,-obj2.grad];

1. Case of the subtraction involving at least one complex uncertainty object:

if (imag(obj1)~=0 && imag(obj2)~=0) || (imag(obj1)==0 && imag(obj2)~=0) || (imag(obj1)~=0 && imag(obj2)==0)

The objects obj1 and obj2 are first split into their respective real and imaginary parts:

obj1\_1= unc(real(obj1));

obj2\_1= unc(real(obj2));

obj1\_2=unc(imag(obj1));

obj2\_2=unc(imag(obj2));

The *value* property of the real part of obj is obtained by simply subtracting the value property of obj2\_1 from the value property of obj1\_1, whereas the *value* property of the imaginary part of obj is obtained by simply subtracting the value property of obj2\_2 from obj1\_2:

obj = complex( unc(obj1\_1.value-obj2\_1.value,0) , unc(obj1\_2.value-obj2\_2.value,0) );

The dep and std\_unc properties are obtained in the same way as done in plusscal() method.

The grad property is also computed in the same way as done in plusscal() method with the exception of the Grad array.

Let us consider two complex-valued quantities *z* and *w*:

Where:

The variables *x,* *y*, *u*, and *v* will represent the uncertainty objects *obj1\_1*, *obj1\_2*, *obj2\_1*, and *obj2\_2* respectively. So we will have the following correspondences:

x=obj1\_1.value

x’= obj1\_1.grad

y=obj1\_2.value

y’= obj1\_2.grad

u=obj2\_1.value

u’= obj2\_1.grad

v=obj2\_2.value

v’= obj2\_2.grad

If we consider the subtraction operation, then

If we consider and, then the subtraction can be written as

The gradient of the real part is then:

Hence, the *Grad* array for the real part of obj is given by

Grad = [ obj1\_1.grad , 0 , -obj2\_1.grad , 0 ] ;

To simplify numerical computations, the *Grad* array can be reduced to the following form:

Grad = [ obj1\_1.grad , -obj2\_1.grad ] ;

The gradient of the imaginary part is:

Hence, the *Grad* array for the imaginary part of obj is given by

Grad = [ 0 , obj1\_2.grad , 0 , -obj2\_2.grad ] ;

To simplify numerical computations, the *Grad* array can be reduced to the following form

Grad = [ obj1\_2.grad , -obj2\_2.grad ];

Finally, the object obj can be returned:

obj = complex(obj\_X , obj\_Y);

## function obj = multscal(obj1,obj2)

This is a method called when performing arithmetic operator overloading.

It takes as input arguments two uncertainty objects obj1 and obj2 and returns another uncertainty object obj.

1. Case of the multiplication of two real uncertainty objects:

if imag(obj1)==0 && imag(obj2)==0

The *value* property of obj is obtained by simply multiplying the value properties of obj1 and obj2:

obj = unc(obj1.value\*obj2.value,0);

The dep and std\_unc properties are obtained in the same way as done in plusscal() method.

The grad property is also computed in the same way as done in plusscal() method with the exception of the Grad array which is formed in the following way:

Grad=[obj1.grad\*obj2.value,obj2.grad\*obj1.value];

1. Case of the product involving at least one complex uncertainty object:

if (imag(obj1)~=0 && imag(obj2)~=0) || (imag(obj1)==0 && imag(obj2)~=0) || (imag(obj1)~=0 && imag(obj2)==0)

The objects obj1 and obj2 are first split into their respective real and imaginary parts:

obj1\_1= unc(real(obj1));

obj2\_1= unc(real(obj2));

obj1\_2=unc(imag(obj1));

obj2\_2=unc(imag(obj2));

Let us consider two complex-valued quantities *z* and *w*:

Where:

The variables *x,* *y*, *u*, and *v* will represent the uncertainty objects *obj1\_1*, *obj1\_2*, *obj2\_1*, and *obj2\_2* respectively. So we will have the following correspondences:

x=obj1\_1.value

x’= obj1\_1.grad

y=obj1\_2.value

y’= obj1\_2.grad

u=obj2\_1.value

u’= obj2\_1.grad

v=obj2\_2.value

v’= obj2\_2.grad

Taking the product of the two complex-valued quantities *z* and *w*:

Where and are respectively the real and imaginary parts of the resulting product.

Therefore, the *value* property of the real and imaginary parts of obj is obtained by applying equation (1):

obj = complex( unc( obj1\_1.value \* obj2\_1.value - obj1\_2.value \* obj2\_2.value , 0 ) ,...

unc( obj1\_1.value \* obj2\_2.value + obj1\_2.value \* obj2\_1.value , 0 ) );

The dep and std\_unc properties are obtained in the same way as done in plusscal() method.

The grad property is also computed in the same way as done in plusscal() method with the exception of the Grad array.

The gradient of the real part is:

Hence, the *Grad* array of the real part is given by:

Grad = [ obj1\_1.grad \* obj2\_1.value , -obj1\_2.grad \* obj2\_2.value...

, obj2\_1.grad \* obj1\_1.value , -obj2\_2.grad \* obj1\_2.value ];

The gradient of the imaginary part is:

Hence, the *Grad* array for the imaginary part is given by:

Grad = [ obj1\_1.grad \* obj2\_2.value , obj1\_2.grad \* obj2\_1.value...

, obj2\_1.grad \* obj1\_2.value , obj2\_2.grad \* obj1\_1.value ];

Finally, the object obj can be returned:

obj = complex(obj\_X , obj\_Y);

## function obj = divscal(obj1,obj2)

This is a method called when performing arithmetic operator overloading.

It takes as input arguments two uncertainty objects obj1 and obj2 and returns another uncertainty object obj.

1. Case of the quotient of two real uncertainty objects:

if imag(obj1)==0 && imag(obj2)==0

The value property of obj is obtained by simply dividing the value property of obj1 by the value property of obj2:

obj = unc(obj1.value/obj2.value,0);

The dep and std\_unc properties are obtained in the same way as done in plusscal() method.

The grad property is also computed in the same way as done in plusscal() method with the exception of the Grad array which is formed in the following way:

Grad=[ obj1.grad/obj2.value , -obj2.grad\*obj1.value/(obj2.value)^2 ];

1. Case of the product involving at least one complex uncertainty object:

if (imag(obj1)~=0 && imag(obj2)~=0) || (imag(obj1)==0 && imag(obj2)~=0) || (imag(obj1)~=0 && imag(obj2)==0)

In this case, the method mrdivide()is called:

mrdivide(obj1,obj2)

## function set\_cov(x1,x2,cov)

This method is used to set the covariance between two uncertainty objects x1 and x2 to the value cov.

Covariance can be set only between real uncertainty objects, otherwise an error message is displayed:

if (x1.img~=0) || (x2.img~=0)

error('For complex unc objects, covariance must be set for real and imaginary parts.')

end

If x1 and x2 have no names assigned to them, i.e.:

if isempty(x1.name) || isempty(x2.name)

then an error message is displayed:

error('unc objects must have names! Assign a name to a unc object using the name property.');

If the covariance value assigned cov is not a valid value, i.e.:

if ( cov > (x1.std\_unc \* x2.std\_unc) ) || (cov < -(x1.std\_unc \* x2.std\_unc))

then an error message is displayed:

error('Incorrect covariance value!\nCovariance must have a value between %f and %f .', -x1.std\_unc \* x2.std\_unc , x1.std\_unc \* x2.std\_unc );

In the case of input arguments x1 and x2 having the same names, an error message is displayed:

if strcmp(x1.name,x2.name)

error('Covariance can only be set for two unc objects having two different names!');

end

If the object x1 is already correlated with other uncertainty objects, then the implemented algorithm searches if the object x2 is one of them. The index, if found, is stored in the variable indx:

indx = find( strcmp( x2.name , x1.rel\_names ) );

If x1 and x2 have not yet been assigned any covariance value, i.e.:

if isempty(indx)

then the name of x2 is first added into the rel\_names property of x1:

x1.rel\_names( length(x1.rel\_names) + 1 ) = x2.name ;

and the covariance value cov is added into the first row of rel\_mat property of x1:

x1.rel\_mat(1, length(x1.rel\_names)) = cov ;

The correlation coefficient is finally added to the second row of rel\_mat property of x1:

x1.rel\_mat(2, length(x1.rel\_names)) = cov/(x1.std\_unc \* x2.std\_unc) ;

Similarly for the object x2, the name of x1 is first added into the rel\_names property of x2:

x2.rel\_names( length(x2.rel\_names) + 1 ) = x1.name ;

and the covariance value cov is added into the first row of rel\_mat property of x2:

x2.rel\_mat(1, length(x2.rel\_names) ) = cov ;

The correlation coefficient is finally added to the second row of rel\_mat property of x2:

x2.rel\_mat(2, length(x2.rel\_names) ) = cov/(x1.std\_unc \* x2.std\_unc) ;

A warning message is displayed to notice the user about the covariance setting that has just taken place:

warning('%s and %s are now correlated!',x1.name{1},x2.name{1});

If x1 and x2 have already been assigned a covariance value, then the variable indx is used to directly update the rel\_mat property of x1:

x1.rel\_mat( 1 , indx ) = cov;

x1.rel\_mat( 2 , indx ) = cov/(x1.std\_unc \* x2.std\_unc) ;

The implemented algorithm searches then for the index of x1 in the rel\_names property of x2 and stores it in the variable indx2:

indx2 = find( strcmp( x1.name , x2.rel\_names ) );

The variable indx2 is then used to directly update the rel\_mat property of x2:

x2.rel\_mat( 1 , indx2 ) = cov;

x2.rel\_mat( 2 , indx2 ) = cov/(x1.std\_unc \* x2.std\_unc) ;

A warning message is displayed to notice the user about the covariance change that has just taken place:

warning('Covariance value between %s and %s changed!',x1.name{1},x2.name{1});

## function set\_correl(x1,x2,cor)

This method is used to set the correlation coefficient between two uncertainty objects x1 and x2 to the value cor.

Correlation can be set only between real uncertainty objects, otherwise an error message is displayed:

if (x1.img~=0) || (x2.img~=0)

error('For complex unc objects, correlation must be set for real and imaginary parts.')

end

If x1 and x2 have no names assigned to them, i.e.:

if isempty(x1.name) || isempty(x2.name)

then an error message is displayed:

error('unc objects must have names! Assign a name to a unc object using the name property.');

If the correlation coefficient assigned *cor* is not a valid value, i.e.:

if (cor > 1) || (cor < -1)

then an error message is displayed:

error('Correlation value must be between -1 and +1!');

In the case of input arguments x1 and x2 having the same names, an error message is displayed:

if strcmp(x1.name,x2.name)

error('Correlation can only be set for two unc objects having two different names!');

end

If the object x1 is already correlated with other uncertainty objects, then the implemented algorithm searches if the object x2 is one of them. The index, if found, is stored in the variable indx:

indx = find( strcmp( x2.name , x1.rel\_names ) );

If x1 and x2 have not yet been assigned any correlation value, i.e.:

if isempty(indx)

then the name of x2 is first added into the rel\_names property of x1:

x1.rel\_names( length(x1.rel\_names) + 1 ) = x2.name ;

and the correlation value *cor* is added into the second row of rel\_mat property of x1:

x1.rel\_mat(2, length(x1.rel\_names)) = cor ;

The covariance value is finally added to the first row of rel\_mat property of x1:

x1.rel\_mat(1, length(x1.rel\_names)) = cor \* x1.std\_unc \* x2.std\_unc ;

Similarly for the object x2, the name of x1 is first added into the rel\_names property of x2:

x2.rel\_names( length(x2.rel\_names) + 1 ) = x1.name ;

and the correlation value cor is added into the second row of rel\_mat property of x2:

x2.rel\_mat(2, length(x2.rel\_names) ) = cor ;

The covariance value is finally added to the first row of rel\_mat property of x2:

x2.rel\_mat(1, length(x2.rel\_names) ) = cor \* x1.std\_unc \* x2.std\_unc ;

A warning message is displayed to notice the user about the correlation setting that has just taken place:

warning('%s and %s are now correlated!',x1.name{1},x2.name{1});

If x1 and x2 have already been assigned a correlation value, then the variable indx is used to directly update the rel\_mat property of x1:

x1.rel\_mat( 2 , indx ) = cor;

x1.rel\_mat( 1 , indx ) = cor \* x1.std\_unc \* x2.std\_unc ;

The implemented algorithm searches then for the index of x1 in the rel\_names property of x2 and stores it in the variable indx2:

indx2 = find( strcmp( x1.name , x2.rel\_names ) );

The variable indx2 is then used to directly update the rel\_mat property of x2:

x2.rel\_mat( 2 , indx2 ) = cor;

x2.rel\_mat( 1 , indx2 ) = cor \* x1.std\_unc \* x2.std\_unc

A warning message is displayed to notice the user about the correlation change that has just taken place:

warning('Correlation value between %s and %s changed!',x1.name{1},x2.name{1});

## function [CX,Dep\_ord] = get\_input\_cov\_mat(Y)

This method returns the input covariance matrix CX of the output quantity Y.

An ordered non-repeated array Dep\_ord containing the uncertainty objects on which Y depends is also returned.

The uncertainty objects Y(i).dep on which Y depends must have assigned names, otherwise an error message is displayed:

error(sprintf('Variables must have names!\nAssign a name to a variable using the name property.\n'));

If all of them have assigned names, i.e.:

if ~isempty(Y(i).dep(j).name)

then they are stored in the variable var\_names:

var\_names(n)=Y(i).dep(j).name;

The names are then stored non-repeated in the variable C:

[ C , ~ , ~] = unique( var\_names );

All the uncertainty objects on which Y depends are then stored in one single variable Dep:

n=0;

for j=1:length(Y)

for i=1:length(Y(j).dep)

n=n+1;

Dep(n) = Y(j).dep(1,i) ;

end

end

An ordered non-repeated array Dep\_ord can be created from Dep:

for i=1:size(Dep,2)

indx = find ( strcmp(Dep(i).name , C) ) ;

Dep\_ord (indx)= Dep(i) ;

end

The covariance matrix CX will have the same size as Dep\_ord, and so can be initially created:

CX = zeros( length( Dep\_ord) );

The elements on the diagonal of CX (i.e. the elements CX(i,j) where i=j)are the variances:

CX(i,j) = ( Dep\_ord(i).std\_unc )^2 ;

Whereas, on the off-diagonal are placed the covariance values as follows:

If the object Dep\_ord(i) is already correlated with other uncertainty objects, then the implemented algorithm searches if the object Dep\_ord(j) is one of them. The index, if found, is stored in the variable indx:

indx = find( strcmp( Dep\_ord(j).name ,Dep\_ord(i).rel\_names ) );

If the indx is empty, i.e.:

if isempty(indx)

then this means that Dep\_ord(i) and Dep\_ord(j) are not correlated, thus a zero value is assigned to the element CX(i,j). If indx is not empty, then this means that Dep\_ord(i) and Dep\_ord(j) are correlated. The variable indx indicates then where to find the covariance value:

if isempty(indx)

cov = 0;

else

cov = Dep\_ord(i).rel\_mat(1,indx);

end

CX(i,j) = cov ;

## function [CX] = get\_input\_cor\_mat(Y)

This method returns the input correlation matrix of the output quantity Y.

First, the input covariance matrix of Y is obtained using the method get\_input\_cov\_mat():

[CX\_cov,Dep\_ord]=get\_input\_cov\_mat(Y);

The returned outputs CX\_cov and Dep\_ord are then used to convert the input covariance matrix into an input correlation matrix:

CX=cov\_into\_cor\_mat(Dep\_ord,CX\_cov);

## function [CX\_cov,Y\_new,Names\_Y\_new]=get\_cov\_mat(X)

This method returns the covariance matrix *CX\_cov* of the uncertainty objects contained in the array X. Returned also are:

Y\_new: array containing all the real and imaginary parts of the elements of the array X but ordered and non-repeated.

Names\_Y\_new: array containing the names of the objects contained in Y\_new.

First, a new array Y, that contains the separate real and imaginary parts of the elements of X, is created:

k=1;

for i=1:length(X)

if X(i).img~=0

Y(k)=X(i).r;

Y(k+1)=X(i).img;

k=k+2;

else

if X(i).img==0

Y(k)=X(i);

k=k+1;

end

end

end

The uncertainty objects Y(i) must have assigned names, otherwise an error message is displayed:

error(sprintf('Variables must have names!\nAssign a name to a variable using the name property.\n'));

If all of them have assigned names, then they are stored in the variable var\_names:

var\_names(n)=Y(i).name;

The names are then stored non-repeated in the variable C:

[ C , ~ , ~] = unique( var\_names );

An ordered non-repeated array Y\_new can be created from comparing the elements of arrays Y.name and C:

for i=1:length(Y)

indx = find ( strcmp(Y(i).name , C) ) ;

Y\_new (indx)= Y(i) ;

Names\_Y\_new (indx)= Y(i).name ;

end

The covariance matrix CX\_cov will have the same size as C, and so can be initially created:

CX\_cov = zeros( length( C ) );

The elements on the diagonal of CX\_cov (i.e. the elements CX\_cov(i,j) where i=j)are the variances:

CX\_cov(i,j) = ( Y\_new(i).std\_unc )^2 ;

Whereas, on the off-diagonal are placed the covariance values as follows:

If the object Y\_new(i) is already correlated with other uncertainty objects, then the implemented algorithm searches if the object Y\_new(j) is one of them. The index, if found, is stored in the variable indx:

indx = find( strcmp( Y\_new(j).name ,Y\_new(i).rel\_names ) );

If the indx is empty, i.e.:

if isempty(indx)

then this means that Y\_new(i) and Y\_new(j) are not correlated, thus a zero value is assigned to the element CX\_cov(i,j). If indx is not empty, then this means

that Y\_new(i) and Y\_new(j)are correlated. The variable indx indicates then where to find the covariance value:

if isempty(indx)

cov = 0;

else

cov = Y\_new(i).rel\_mat(1,indx);

end

CX\_cov(i,j) = cov ;

The array X must be a 1xn array, i.e.:

size(X,1)==1

Otherwise, an error message is displayed:

error(' Input argument must be a one-row array of uncertainty objects!')

## function [CX\_cor]=get\_cor\_mat(Y)

This method returns the correlation matrix of the uncertainty objects contained in the array Y.

First, the covariance matrix of Y is obtained using the method get\_cov\_mat():

CX\_cov=get\_cov\_mat(Y);

The returned covariance matrix CX\_cov is then converted into a correlation matrix:

CX\_cor=cov\_into\_cor\_mat(Y,CX\_cov);

## function [CX\_cov] = cor\_into\_cov\_mat(X,CX\_cor)

This method transforms a correlation matrix CX\_cor of an array X of uncertainty objects into a covariance matrix CX\_cov.

CX\_cov has the same size as CX\_cor, with the elements CX\_cov(i,j) given by:

CX\_cov(i,j)=CX\_cor(i,j) \* X(i).std\_unc \* X(j).std\_unc ;

## function [CX\_cor] = cov\_into\_cor\_mat(X,CX\_cov)

This method transforms a covariance matrix CX\_cov of an array X of uncertainty objects into a correlation matrix CX\_cor.

CX\_cor has the same size as CX\_cov, with the elements CX\_cor(i,j) given by:

CX\_cor(i,j)=CX\_cov(i,j) / (X(i).std\_unc \* X(j).std\_unc) ;

## function [K, Coef\_mat]=karhloev( X,CX,arg3 )

This method generates the array K of uncorrelated uncertainty objects from the array X of correlated uncertainty objects using Karhunen-Loeve decomposition.

The number of input arguments must be either one or three, that is:

(nargin==1) || (nargin==3)

Otherwise, an error message is displayed:

error('Incorrect number of input arguments!');

If the number of input arguments is one, i.e.:

if nargin==1

then the syntax karhloev(X)is selected which is indicated by setting the variable arg to 1 ( arg=1 ). The covariance matrix CX of X, as well as an array Z containing all the real and imaginary parts of the elements of the array X ,ordered and non-repeated, are obtained using the method get\_cov\_mat():

[CX,Z]=get\_cov\_mat(X);

Since a covariance matrix is used, the variable arg3, representing the third input argument, is set to ‘cov’:

arg3='cov';

The implemented algorithm will operate on the variable with name Z, that’s why in the case of three input arguments, X is renamed to Z:

if nargin==3

Z=X;

end

Only strings ‘cov’ and ‘corr’ are valid for the third input argument arg3. The second argument CX must be a matrix of type double and with size nxn where n is the number of elements in the array Z. The condition on the size of CX can be ignored in the case the number of input arguments is equal to one. The new array Z must be of size 1xn. Hence the overall conditions:

if (strcmp(arg3,'cov') || strcmp(arg3,'corr')) && isa(CX,'double')&& ((size(CX,1)==length(Z) && size(CX,2)==length(Z) ) || (arg==1) ) && size(Z,1)==1

If the above conditions are not satisfied, then an error message is displayed:

error (' Incorrect input arguments!');

If the third input argument is ‘corr’, then the method cor\_into\_cov\_mat() is used to transform the correlation matrix ,given in the second argument, into a covariance matrix:

if strcmp(arg3,'corr')

CX=cor\_into\_cov\_mat(Z,CX);

end

The covariance values contained in CX are then assigned to the uncertainty objects contained in Z:

unc(Z,CX,'cov');

The Eigen-vectors of the covariance matrix CX are obtained using the function eig():

[ V , ~ ] = eig(CX);

The uncorrelated uncertainty objects can then be readily found:

K = V' \* Z';

With a coefficient matrix given by:

Coef\_mat= V;

## function [cov,CY\_cov] = covr(y1,y2)

This method returns the covariance *cov* as well as the output covariance matrix *CY\_cov* of two uncertainty objects y1 and y2.

1. **Case where at least one of the objects y1 or y2 is complex:**

If y1 and y2 are two different complex uncertainty objects, i.e.:

if (y1.img~=0) && (y2.img~=0) && (y1~=y2)

then the real and imaginary parts of y1 and y2 are first stored in the array A:

A=[y1.r y1.img y2.r y2.img];

Finding the covariance between complex uncertainty objects amounts then to finding covariance between the real and imaginary parts contained in the array A. The covariance *cov* and the output covariance matrix CY\_cov are therefore the same 4x4 array entity:

for i=1:4

for j=1:4

cov(i,j)=covr(A(i),A(j));

end

end

CY\_cov=cov;

If y1 and y2 are two equal complex uncertainty objects, i.e.:

if (y1.img~=0) && (y2.img~=0) && (y1==y2)

then the real and imaginary parts of y1 are first stored in the array A:

A=[y1.r y1.img];

Finding the covariance between complex uncertainty objects amounts then to finding covariance between the real and imaginary parts contained in the array A. Since y1 and y2 are equal, the covariance *cov* and the output covariance matrix CY\_cov are the same 2x2 array entity:

for i=1:2

for j=1:2

cov(i,j)=covr(A(i),A(j));

end

end

CY\_cov=cov;

If y1 and y2 are two different objects and one of them is real, i.e.:

if ( ( (y1.img~=0) && (y2.img==0) ) || ( (y1.img==0) && (y2.img~=0) ) ) && (y1~=y2)

then the covariance *cov* and the output covariance matrix *CY\_cov* are the same 1x2 array whose elements are the covariance between the real object and the real and imaginary parts of the complex object:

cov(1)=covr(a,b);

cov(2)=covr(a,c);

where a, b, and c are given by the algorithm:

if y1.img==0

a=y1;

b=y2.r;

c=y2.img;

else

if y2.img==0

a=y2;

b=y1.r;

c=y1.img;

end

end

1. **Case where both of the objects y1 and y2 are real:**

That is:

if (y1.img==0) && (y2.img==0)

If the object y1 is already correlated with other uncertainty objects, then the implemented algorithm searches if the object y2 is one of them. The index, if found, is stored in the variable indx:

indx = find( strcmp( y2.name , y1.rel\_names ) );

If the variable indx is not empty then the covariance between y1 and y2 is taken directly from the rel\_mat property of y1:

cov=y1.rel\_mat(1, indx );

The output covariance matrix CY\_cov is returned empty:

CY\_cov=[];

If y1 and y2 are identical, i.e.:

if strcmp(y1.name,y2.name)

then the covariance cov is equal to the variance of either one of them:

cov=(y1.std\_unc)^2;

The output covariance matrix CY\_cov is returned empty:

CY\_cov=[];

In the case y1 and y2 have been just created in MATLAB Workspace for the first time, i.e.:

if length(y1.grad)<=1 && length(y2.grad)<=1

if the object y1 is correlated with other uncertainty objects, then the implemented algorithm searches if the object y2 is one of them. The index, if found, is stored in the variable indx:

indx = find( strcmp( y2.name ,y1.rel\_names ) );

If the variable indx is empty, i.e.:

if isempty(indx)

then by default the covariance is set to zero:

cov = 0;

and the output covariance matrix CY\_cov is returned empty:

CY\_cov=[];

If, on the other hand, the variable indx is not empty, then the covariance between y1 and y2 is directly taken from the rel\_mat property of y1:

cov = y1.rel\_mat(1,indx);

and the output covariance matrix CY\_cov is returned empty:

CY\_cov=[];

If either y1 or y2 is an output quantity of some measurement model, then the output covariance matrix CY\_cov is found using the Uncertainty Propagation Law:

CY\_cov= J \* CK \* J' ;

First, the uncertainty objects on which y1 and y2 depend are gathered in one array

var\_objects= [ y1.dep y2.dep ];

In exactly the same procedure employed in the method jacobian(), an array X of ordered non-repeated uncertainty objects is generated.

The array X is then decorrelated using the method karhloev():

[K, Coef\_mat]=karhloev( X , CX ,'cov');

Where CX is the input covariance matrix obtained using the method get\_input\_cov\_mat():

[CX]=get\_input\_cov\_mat([y1 y2]);

The returned array K is used to find the input covariance matrix CK associated with the uncorrelated uncertainty objects, whereas the returned array Coef\_mat is used ,along with the gradients arrays Grad1 and Grad2, to find the Jacobian of the new system with uncorrelated inputs:

J= [ Grad1 ; Grad2 ] \* Coef\_mat ;

The covariance between y1 and y2 can then be readily found from the output covariance matrix CY\_cov:

cov = CY\_cov(1,2);

## function [cor,CY\_corr] = correl(y1,y2)

This method returns the correlation coefficient (Pearson's correlation coefficient) *cor* as well as the output correlation matrix *CY\_corr* of two uncertainty objects y1 and y2.

1. **Case where at least one of the objects y1 or y2 is complex:**

If y1 and y2 are two different complex uncertainty objects, i.e.:

if (y1.img~=0) && (y2.img~=0) && (y1~=y2)

then the real and imaginary parts of y1 and y2 are first stored in the array A:

A=[y1.r y1.img y2.r y2.img];

Finding the correlation between complex uncertainty objects amounts then to finding correlation between the real and imaginary parts contained in the array A. The correlation *cor* and the output correlation matrix CY\_corr are therefore the same 4x4 array entity:

for i=1:4

for j=1:4

cor(i,j)=correl(A(i),A(j));

end

end

CY\_corr=cor;

If y1 and y2 are two equal complex uncertainty objects, i.e.:

if (y1.img~=0) && (y2.img~=0) && (y1==y2)

then the real and imaginary parts of y1 are first stored in the array A:

A=[y1.r y1.img];

Finding the correlation between complex uncertainty objects amounts then to finding correlation between the real and imaginary parts contained in the array A. Since y1 and y2 are equal, the correlation *cor* and the output correlation matrix CY\_corr are the same 2x2 array entity:

for i=1:2

for j=1:2

cor(i,j)=correl(A(i),A(j));

end

end

CY\_corr=cor;

If y1 and y2 are two different objects and one of them is real, i.e.:

if ( ( (y1.img~=0) && (y2.img==0) ) || ( (y1.img==0) && (y2.img~=0) ) ) && (y1~=y2)

then the correlation *cor* and the output correlation matrix *CY\_corr* are the same 1x2 array whose elements are the correlations between the real object and the real and imaginary parts of the complex object:

cov(1)=covr(a,b);

cov(2)=covr(a,c);

where a, b, and c are given by the algorithm:

if y1.img==0

a=y1;

b=y2.r;

c=y2.img;

else

if y2.img==0

a=y2;

b=y1.r;

c=y1.img;

end

end

1. **Case where both of the objects y1 and y2 are real:**

That is:

if (y1.img==0) && (y2.img==0)

If the object y1 is already correlated with other uncertainty objects, then the implemented algorithm searches if the object y2 is one of them. The index, if found, is stored in the variable indx:

indx = find( strcmp( y2.name , y1.rel\_names ) );

If the variable indx is not empty then the correlation between y1 and y2 is taken directly from the rel\_mat property of y1:

cor=y1.rel\_mat(2, indx );

The output correlation matrix CY\_corr is returned empty:

CY\_corr=[];

If y1 and y2 are identical, i.e.:

if strcmp(y1.name,y2.name)

then the correlation cor is equal to 1:

cor=1;

The output correlation matrix CY\_corr is returned empty:

CY\_corr=[];

In the case y1 and y2 have been just created in MATLAB Workspace for the first time, i.e.:

if length(y1.grad)<=1 && length(y2.grad)<=1

if the object y1 is correlated with other uncertainty objects, then the implemented algorithm searches if the object y2 is one of them. The index, if found, is stored in the variable indx:

indx = find( strcmp( y2.name ,y1.rel\_names ) );

If the variable indx is empty, i.e.:

if isempty(indx)

then by default the correlation is set to zero:

cor = 0;

and the output correlation matrix CY\_corr is returned empty:

CY\_corr=[];

If, on the other hand, the variable indx is not empty, then the correlation between y1 and y2 is directly taken from the rel\_mat property of y1:

cor = y1.rel\_mat(2,indx);

and the output correlation matrix CY\_corr is returned empty:

CY\_corr=[];

If either y1 or y2 is an output quantity of some measurement model, then the output covariance matrix CY is first found using the method covr():

[~,CY]=covr(y1,y2);

*CY* is then used to find the correlation matrix *CY\_corr*, which is in turn used to return the correlation coefficient *cor*:

cor=CY\_corr(1,2);

## function [J,Names]= jacobian(obj)

This method returns:

1. J: Jacobian matrix of the array obj of uncertainty objects
2. Names: names of the variables with respect to which the

partial derivatives are taken.

If the array obj contains only one element, then the Jacobian is directly taken from the grad property of obj:

J(1,i)= obj.grad(i);

The Names array in this case contains the names of the elements of the dep property of obj:

Names(1,i)=obj.dep(1,i).name;

In case no names are assigned, the string ‘not named’ is returned:

if isempty(obj.dep(1,i).name)

Names(1,i)={'not named'};

If the array obj contains more than one element and is a one-row array, i.e.:

if size(obj,2)>1 && size(obj,1)==1

then the uncertainty objects must have names assigned to them. Performing ordering operation is necessary as well.

If obj is not a one-row array, then an error message is displayed:

error(' Number of rows of the input array must be equal to one.');

The uncertainty objects obj(1,i).dep(1,j) must have assigned names, otherwise an error message is displayed:

error(' unc objects must have names!');

If all of them have assigned names, then they are stored in the variable var\_names:

var\_names(c)=obj(1,i).dep(1,j).name;

The names are then stored non-repeated in the variable C:

[ C , ~ , ~] = unique( var\_names );

The names of the elements obj(1,i) are stored in the variable obj\_Names(i,:):

obj\_Names(i,n)=obj(1,i).dep(1,n).name;

An ordered array Grad(i,j) ,which contains the gradients of the elements obj(1,i), is created in the same way done in the method plusscal().

An ordered array Names, which contains the names of the elements obj(1,i) with respect to which the partial derivatives are taken, is created as well:

for ii=1:size(obj\_Names(i,:),2)

for jj=1:length(C)

indx = find( strcmp(obj\_Names(i,ii),C));

Grad(i,indx) = obj(1,i).grad(ii);

Names(i,indx) = obj(1,i).dep(1,ii).name;

end

end

The Jacobian matrix is then given by J=[ Grad(1,:); Grad(2,:);... ;Grad(n,:)]. Hence the command:

J=Grad;

## function [obj\_names,obj\_array] = get\_ws\_obj()

This method returns all the uncertainty objects existing in MATLAB Workspace.

The returned output is namely:

1. obj\_names: cell array containing their names
2. obj\_array: cell array containing the uncertainty objects themselves

The MATLAB Workspace of a function is separate from the Base Workspace. For

this reason the function evalin() is used to find all the variables in Workspace and store the result in the variable var\_mat:

var\_mat=evalin('base','whos');

The number of uncertainty objects is first counted by comparing the data type of the objects found in Workspace with the data type of uncertainty objects ( which is *unc* ):

c=1; % counter

for i=1:length(var\_mat)

if strcmp(var\_mat(i).class,'unc')

c=c+1; % increment counter

end

end

Then are finally stored in the variable obj\_array. Their respective names are stored in the variable obj\_names:

c=1; % counter

for i=1:length(var\_mat)

if strcmp(var\_mat(i).class,'unc')

obj\_names{1,c}=[var\_mat(i).name];

obj\_array{1,c}=evalin('base',obj\_names{1,c});

c=c+1; % increment counter

end

end

## function data = read\_data(arg1,arg2,arg3,arg4,arg5)

This method is used to generate an array *data* of uncertainty objects out of a data file. Data files can be MICROSOFT Excel, dat or csv files.

The extension of the file is obtained using the function fileparts()and stored in the variable ext:

[~,~,ext] = fileparts(arg1);

If the file extension is xls or xlsx, i.e.:

if strcmp(ext , '.xls') || strcmp(ext , '.xlsx')

then two syntaxes are possible, one with five input arguments, and the other one with four input arguments.

If the syntax with five input arguments is selected, i.e.:

if (nargin==5)

then the corresponding input arguments are:

read\_data(file\_name , sheet\_name , mean\_val , std\_unc , names)

In this case, the contents of the file cells are read and stored into their corresponding variables using the xlsread() function:

[~,var\_names] = xlsread(arg1, arg2 , arg5 );

var\_means = xlsread(arg1, arg2 , arg3 );

var\_stds = xlsread(arg1, arg2 , arg4 );

The array data of uncertainty objects can then be generated:

data=unc(var\_means , var\_stds , var\_names );

If the syntax with four input arguments is selected, i.e.:

if (nargin==4)

then the corresponding input arguments are:

read\_data( file\_name , sheet\_name , range , 'r/c')

Read raw data from a specified range *range* of an Excel sheet *sheet\_name* of an Excel file *file\_name* and generate uncertainty objects either from rows ( if argument r is used) or columns( if argument c is used).

Mean values and standard uncertainties are computed each row (if argument r is used, i.e. if strcmp(arg4,'r') ) using the functions mean() and std() respectively:

for i=1:size(data,1)

var\_means(i)=mean(data(i,:));

var\_stds(i)=std(data(i,:));

end

The array *data* of uncertainty objects is then created accordingly:

data=unc(var\_means , var\_stds );

Mean values and standard uncertainties are computed each column ( if argument c is used. i.e. if strcmp(arg4,'c')) using the functions mean() and std() respectively:

for i=1:size(data,2)

var\_means(i)=mean(data(:,i));

var\_stds(i)=std(data(:,i));

end

The array *data* of uncertainty objects is then created accordingly:

data=unc(var\_means , var\_stds );

Another possible syntax for the four input argument case is:

read\_data( file\_name , sheet\_name , mean\_val , std\_unc )

In this case, the contents of the file cells are read and stored into their corresponding variables using the xlsread() function:

var\_means = xlsread(arg1, arg2 , arg3 );

var\_stds = xlsread(arg1, arg2 , arg4 );

The array data of uncertainty objects can then be generated:

data=unc(var\_means , var\_stds );

If the number of input arguments is neither five nor four, then an error message is displayed:

error('Incorrect number of input arguments!');

If the file extension is dat or csv, i.e.:

if strcmp(ext , '.dat') || strcmp(ext , '.csv')

then the array data of uncertainty objects is generated in exactly the same way as done for the case of xls and xlsx extensions with the exception of:

1. the use of the function xlsread() which is replaced by the function csvread(), and
2. the interpretation of the input arguments and their number. The syntax with five input arguments is :

read\_data( file\_name , R , C , [R C R2 C2] , 'r/c' )

That is: from the file *file\_name*, read only the range specified by [R C R2 C2] where (R,C) is the upper-left corner of the data to be read and (R2,C2) is the lower-right corner.

The syntax with two input arguments is :

read\_data( file\_name , 'r/c' )

That is: read data from the file *file\_name*. Mean values and standard uncertainties are computed each row (if argument r is used) or column ( if argument c is used) and uncertainty objects are created accordingly.

If file extensions other than xls, xlsx, dat, or csv are used, then an error message is displayed:

error('Incorrect file type!');

# References

[1] Hubert Zangl, Klaus Hörmaier, “Educational Aspects of Uncertainty Calculation with Software Tools”

[2] Wolfgang Granig, Slimane Aoudjit, Lisa Marie Faller, Hubert Zangl, “Statistical Modeling of Integrated Sensors for Automotive Applications”

[3] MATLAB R2016b, The MathWorks, Inc., Natick, Massachusetts, United States