Dymore User's Manual
Model parameterization

Contents

1 Introduction **1**
- 1.1 Context . . . 4
- 1.2 Recursive search for design parameters . . . 4
- 1.3 Prefix . . . 5
- 1.4 Vertices . . . 5
- 1.5 Automatic design parameters . . . 5

2 Example: The four bar mechanism on a flexible beam **5**
- 2.1 Definition of the four bar mechanism assembly . . . 6
- 2.2 Definition of the articulated bar assembly . . . 8

3 Design parameters **9**

4 Expression calculator **9**
- 4.1 Example 1: string operations . . . 10
- 4.2 Example 2: vector operations . . . 10
- 4.3 Example 3: vector operations . . . 11
- 4.4 String operators . . . 11
- 4.5 floating point operators . . . 11

1 Introduction

The data blocks appearing in the various input files that define a model describe a physical system with specific properties. It is often the case that a given system must be analyzed for a range of parameter values. For instance, in a typical situation, the answer to the following questions is of interest: how does the response of the system change when the length of a beam increases? or how is the stability of the system affected by changes in the far field flow velocity? Of course, the user could modify the parameters of interest in the input file and run the simulation for a number of values of the parameters. The model parameterization technique described in this section allows a more systematic approach to model parameterization.

Model parameterization involves two interrelated concepts, *assemblies* and *design parameters*.

1. The finite element based multibody approach to the description of complex mechanical systems is based on a building block approach: the system is composed of a number of interconnected basic element, such as beam, revolute joints, etc. In many cases, complex systems consist of a number of **assemblies**, each modeled by a number of basic elements. For instance, a planar mechanism is composed of a number of articulated bars: each articulated bar assembly involves two basic elements, a beam and a revolute joint. To describe such mechanisms, it is convenient to use the articulated bar assembly as a unit that can be manipulated as a whole. Another example is that of a helicopter rotor; it consists of a number of assemblies each comprising several basic elements. The pitch link assembly is composed of beam rigid bodies and several joints. The blade root retention assembly for a fully articulated helicopter involves beam, rigid bodies and several joints. Both pitch link and root retention assemblies should be manipulated as single units rather than a individual basic elements.

   Assemblies are defined in a special type of input files, called "template files," that can be reused for many applications, further increasing the usefulness of the concept of assemblies.

   An assembly can be included multiple times in a model. For instance the articulated bar assembly will be included three times in the definition of a four bar mechanism. Furthermore, the concept of assembly is recursive: a blade assembly involves assemblies for the pitch link, root retention and blade proper; the rotor assembly then involves a number of blade assemblies.

2. To enable the use of an assembly across several applications, it must be possible to modify the geometric and physical properties of the basic elements composing the assembly. In fact, an assembly should solely define the topological configuration of a subsystem, while the geometric and physical properties of its components will be application dependent. The geometric and physical properties of an assembly's components will be defined by means of **design parameters**, see section 3. Design parameters are variables that can appear in the template files defining an assembly and are used to specify the geometric and physical properties of the assembly's components. For instance, rather than defining the coordinates of a point as

   `@COORDINATES { 0.24, 0.12, -0.10}`,

   the following syntax is allowed

   `@COORDINATES { #X1, #X2, #a #b ~+}`.

   `#X1, #X2, #a` and `#b` are design parameters, as indicated by the initial special character of their name, `#`. `~+` is an arithmetic operator; the special initial character `~` introduces arithmetic operator. The symbolic expression `#a #b ~+` simply indicates that the sum of the values of design parameters `#a` and `#b` will be computed; note that the reverse Polish notation is used for arithmetic operations. The syntax used with design parameters is detailed in section 4. Design parameters are assigned values in

a **design parameter file** recognizable by its extension `.dgp`; this file lists all design parameters and their corresponding values, see section 3.

The concepts of assemblies and design parameters, and their interplay, will be illustrated by the block diagram shown in fig. 1. The master file is called `example.dym`; design parameters can appear in this file, and their values are defined in one or more design parameter files that are declared in the process control parameters. For this simple illustration, a single design parameter file, `example.dgp`, is defined; for later reference, this design parameters file is a `Level 0` design parameter file.

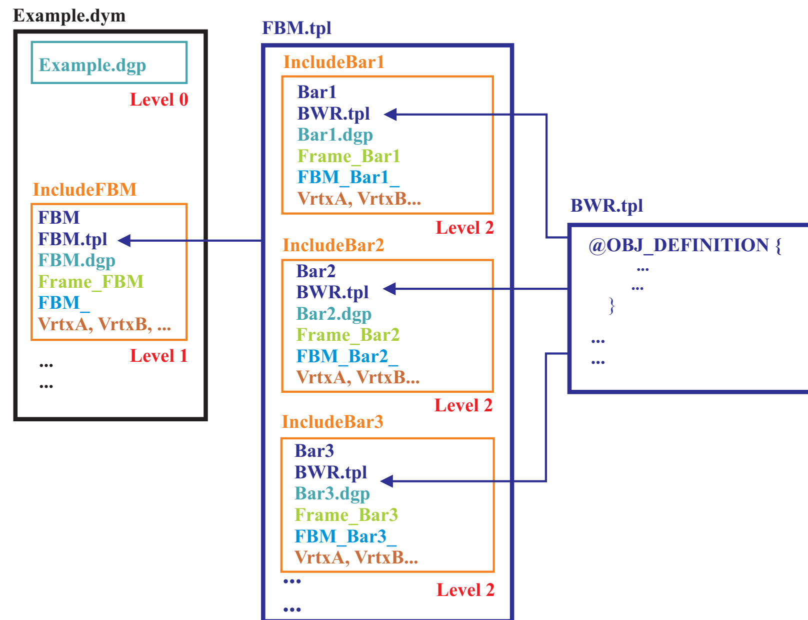

Figure 1: The four bar mechanism mounted on an elastic beam.

In this master file, a single assembly, named `FBM`, is included. The definition of the assembly will be found in file `FBM.tpl`, and the design parameters appearing therein will be listed in file `FBM.dgp`, which is a `Level 1` design parameter file. Three additional parameters are required to define the assembly: a fixed frame, `Frame_FBM`, a prefix, `FBM_`, and a set of connection vertices. All the geometric elements of an assembly should be defined in a fixed frame. This allows a generic assembly to be reused in various geometric configurations. A design parameter named `#FixedFrame` will be created and assigned the value `Frame_FBM`. To identify the elements belonging to an assembly, it is convenient to prefix all their name with a unique string. In this case, the names of all elements defined in the `FBM` assembly will be prefixed with the string `FBM_`. A design parameter named `#PFX` will be created and assigned the value `FBM_`.

The concept of assembly is recursive, *i.e.* an assembly can be defined by a number of assemblies and basic elements. In the present example, assembly `FBM` involves three identical assemblies, named `Bar1`, `Bar2` and `Bar3`. Since these three assemblies are identical, they are defined by the *same template file*, `BWR.tpl`; the properties of each of the assemblies, however, are different since each assembly is assigned a different design parameter file, `Bar1.dgp`, `Bar2.dgp` and `Bar3.dgp`, for assemblies `Bar1`, `Bar2` and `Bar3`, respectively;

each of these design parameter files are `Level 2` design parameter files. Note that identical design parameter files could be assigned to various assemblies; in that case, the geometric and physical properties of the various assemblies will be identical, although each assembly would be defined with respect to its own fixed frame. This option could be used to define a helicopter blade with four identical blades. A single blade assembly in included four times with four identical design parameter files. For the case of a four bladed helicopter with one dissimilar blade, three blade assemblies are included each sharing a single design parameter file, and the dissimilar blade is defined by the same blade assembly, but assigned its own design parameter file.

1.1 Context

Design parameters are defined within a *context*. For instance, when including the template file `BWR.tpl` for the definition of assembly `Bar1`, the values of the design parameters listed in the design parameter file `Bar1.dgp` will be used. On the other hand, when including the same template file `BWR.tpl` for the definition of assembly `Bar2`, the values of the design parameters listed in the design parameter file `Bar2.dgp` will be used. In other words, when including the template file `BWR.tpl` for the definition of assembly `Bar1` the values of the design parameters listed in the design parameter file `Bar1.dgp` *are in context,* whereas those listed in design parameter file `Bar2.dgp` and `Bar3.dgp` *are not in context.*

1.2 Recursive search for design parameters

As mentioned above, the definition of assemblies is recursive. Assembly `FBM` is said to be the *parent assembly* of assemblies `Bar1`, `Bar2` and `Bar3`, whereas assemblies `Bar1`, `Bar2` and `Bar3` are *child assemblies* of assembly `FBM`. The same vocabulary is used for the associated design parameter files. The `Level 0, 1`, and `2` notation used fig. 1 indicates the depth of recursion of the assemblies. Note that design parameter files are always associated with an assembly, except for the `Level 0` design parameter file which is defined in the process control parameters section of the master file and has no associated assembly.

It is possible for a design parameter to be undefined in a given context. For instance, while including the template file `BWR.tpl` for the definition of assembly `Bar1`, it might happen that a particular design parameter is not defined in the design parameter file `Bar1.dgp`. In that case, the missing design parameter could be defined in the parent design parameter file, file `FBM.dgp` in this case. If not defined in that file, it would have to be defined in the root design parameter file, the `Level 0` file, `Example.dgp`. In summary, the search for the definition of design parameters is a recursive process. As soon as the definition of a parameter is found, the search ends. This implies that if a design parameter is defined twice within the recursion path, the definition that is met first will be used, and hence, the second is ignored. If the definition of a design parameter is not found within the recursion path, an error message is printed.

The recursive nature of the search for the definition of design parameters enables a single definition of a design parameter to be shared by multiple assemblies. This is a convenient option to use if identical design parameters are to be assigned identical values in a number

of design parameter files: rather than defining the same design parameter multiple times, it can be defined a single time in the parent design parameter file.

1.3 Prefix

The triple inclusion of the template file `BWR.tpl` in the `FBM` assembly creates a potential conflict because elements with identical names will be repeated three time in the model. This is the reason why it is indispensable to create distinctive names for all the elements of an assembly. This achieved by adding a prefix in front of the names of all elements defined in the assembly.

1.4 Vertices

Vertices are connection points between the various structural elements of the model. Typically, assemblies are connected to the rest of the model at a number of vertices. Hence, the definition of an assembly requires the definition of a list of vertices.

1.5 Automatic design parameters

The definition of an assembly involves the specification of the associated design parameter files that assign values to a number of design parameters. In addition, the definition of an assembly also automatically generates the *default design parameters* listed below.

1. `#PFX`. This default design parameter is assigned the string value of the prefix for the assembly.

2. `#FixedFrame`. This default design parameter is assigned the string value of the fixed frame name for the assembly.

3. `#VertexA`, `#PointA`, `#PointAxyz`. This set of default parameters is associated with the first vertex of the assembly. `#VertexA` is assigned the string value of the first vertex name for the assembly; `#PointA` is assigned the string value of the point associated with the first vertex for the assembly; `#PointAxyz` is assigned the vector value of the coordinates of the point associated with the first vertex for the assembly, *expressed in the assembly's fixed frame*. Corresponding design parameters, `#VertexB`, `#PointB`, `#PointBxyz`, are associated with the second vertex of the assembly, `#VertexC`, `#PointC`, `#PointCxyz`, with the third, and so on.

2 Example: The four bar mechanism on a flexible beam

Consider the four bar mechanics mounted on a flexible beam as shown in fig. 2. The beam elements labeled `Bar1`, `Bar2` and `Bar3`, connected by revolute joints labeled `Rvj1`, `Rvj2`, `Rvj3` and `Rvj4`, form the four bar mechanism, which is connected to the flexible beam at vertices `VertexM` and `VertexT`.

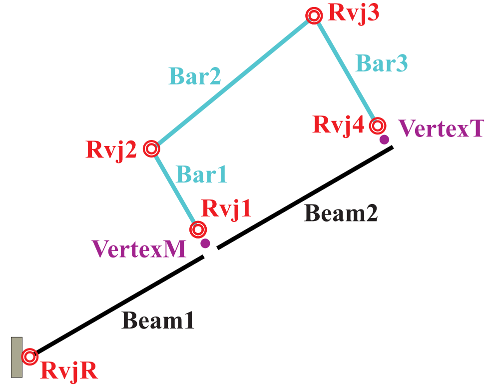

Figure 2: The four bar mechanism mounted on an elastic beam.

The file structure used to describe this example is shown in fig. 1. The master file, `Example.dym`, defines the four bar mechanism and the supporting flexible beam. The description of the supporting flexible beam involves no assemblies, and hence, it will not be discussed here. The four bar mechanism forms an assembly, which is defined in the master file. The template file `FBM.tpl` defines the four bar mechanism, which is itself defined as a combination of three identical articulated bar assemblies, plus a number of basic elements. The template file `BWR.tpl` defines the articulated bar assembly, which is composed of basic elements, and hence will not be discussed here.

Fig. 3 depicts the manner in which the physical mechanism will be represented as a recursion of assemblies. Whereas the flexible beam is represented by a collection of basic elements, two beams and a revolute joint, the four bar mechanism is an assembly that connects to the flexible beam at two vertices, `VertexM` and `VertexT`. Next, the four bar mechanism is now defined by three identical articulated bar assemblies, `Bar1`, `Bar2` and `Bar3`. Note that each articulated bar possesses its own fixed frame, `Frame_Bar1`, `Frame_Bar2` and `Frame_Bar3`, for `Bar1`, `Bar2` and `Bar3`, respectively. Similarly, each bar features its own prefix, `FBM_Bar1_`, `FBM_Bar2_` and `FBM_Bar3_`, for `Bar1`, `Bar2` and `Bar3`, respectively. Finally, each articulated bar has connections to two different vertices: vertices `FBM_Vertex12` and `VertexM` for `Bar1`, vertices `FBM_Vertex12` and `FBM_Vertex23` for `Bar2`, and vertices `FBM_Vertex23` and `VertexT` for `Bar3`. Note that `Bar2` is connected to two vertices that belong to the `FBM` assembly, whereas `Bar1` and `Bar3` are each connected to a vertex belonging to the `FBM` assembly and the other not. The completion of the definition of the four bar mechanism assembly requires additional basic elements, the revolute joint `FBM_RvjT`, and three vertices, `FBM_Vertex12`, `FBM_Vertex23` and `FBM_Vertex34`.

2.1 Definition of the four bar mechanism assembly

Key aspects of the definition of the model depicted in fig. 3, will now be discussed. The following include command appear in the master file, `Example.dym`, and defines the four bar mechanism assembly, named `FBM`.

@INCLUDE_COMMAND {

@INCLUDE_COMMAND_NAME { `IncludeFBM` **} {**

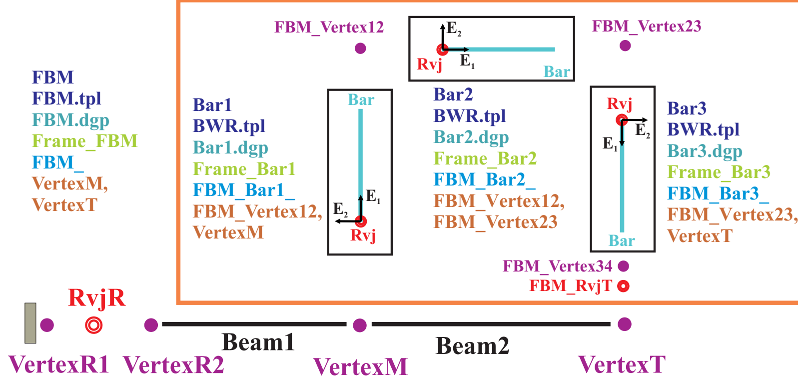

Figure 3: The four bar mechanism mounted on a flexible beam: recursive definition of assemblies.

```
@ACTIVE_COMMAND { YES }
@LIST_OF_FILE_NAMES { template/FBM.tpl,
model/BarProperties.dat, model/ScheduleRotation.dat }
@ASSEMBLY_DEDFINITION {
    @ASSEMBLY_NAME { FBM }
    @DESIGN_PARAMETERS_FILE_NAME_LIST { FBM.dgp }
    @PREFIX_LABEL { FBM_ }
    @FIXED_FRAME_NAME { FrameFBM }
    @CONNECTED_TO_VERTICES { VertexM, VertexT }
}

    }

}
```

Note that this command includes into the model the template file that defines the four bar mechanism assembly, `template/FBM.tpl`, together with additional data files, `model/BarProperties.dat` and `model/ScheduleRotation.dat`. Next, the four bar mechanism assembly, `FBM`, is defined. The prefix for all the elements of the assembly is `FBM_`, and hence, an automatic design parameter will be defined, `#PFX = FBM_`. The four bar mechanism will be defined in a fixed frame, `FrameFBM`; here again, an automatic design parameter will be defined `#FixedFrame = FrameFBM`. Finally, as shown in fig. 2, the four bar mechanism assembly is connected to the flexible beam at two vertices, `VertexM` and `VertexT`. The following automatic design parameters will be defined `#VertexA = VertexM`, `#PointA = PointM` (assuming that `PointM` is associated with `VertexM`), and `#PointAxyz = (0.2, 0.5, -0.7)` (assuming that `(0.2, 0.5, -0.7)` are the coordinates of `PointA` in fixed frame `FrameFBM`). Similarly, the following automatic design parameters will be defined `#VertexB = VertexT`, `#PointB = PointT` (assuming that `PointT` is associated with `VertexT`), and `#PointBxyz = (0.3, -0.4, 0.6)` (assuming that `(0.3, -0.4, 0.6)` are the coordinates of `PointB` in fixed frame `FrameFBM`).

2.2 Definition of the articulated bar assembly

Next, the definition of the four mechanism is discussed. The following include command appear in the template file, FBM.dym.

```
@INCLUDE_COMMAND {
    @INCLUDE_COMMAND_NAME { #PFX <IncludeBar1> ~+ } {
        @ACTIVE_COMMAND { YES }
        @LIST_OF_FILE_NAMES { template/BWR.tpl }
        @ASSEMBLY_DEFINITION {
            @ASSEMBLY_NAME { #PFX <Bar1> ~+ }
            @DESIGN_PARAMETERS_FILE_NAME_LIST { BWR1.dgp }
            @PREFIX_LABEL { #PFX <Bar1_> ~+ }
            @FIXED_FRAME_NAME { #PFX <FrameBar1> ~+ }
            @CONNECTED_TO_VERTICES { #VertexA, #PFX <Vertex12> ~+ }
        }
    }
}
```

This command includes into the model the template file that defines the articulated bar assembly, template/BWR.tpl. The name of the command is specified as #PFX <IncludeBar1> ~+; since the present context is that associated with the FBM assembly, this expression is interpreted as the concatenation of string #PFX = FBM_ (this automatic design parameter was created as part of the definition of the FBM assembly), and string IncludeBar1 (the special characters <...> indicate that IncludeBar1 should be interpreted as a string); the operator ~+ indicates the concatenation operation. In summary, in the present context, #PFX <IncludeBar1> ~+ is interpreted as FBM_IncludeBar1.

Next, the articulated bar assembly is defined. Note that the assembly name, prefix label and fixed frame name are defined as #PFX <Bar1> ~+, #PFX <Bar1_> ~+, and #PFX <FrameBar1> ~+, respectively, and interpreted as FBM_Bar1, FBM_Bar1_, and FBM_FrameBar1, respectively. Two automatic design parameters are defined *within the articulated bar context*: #PFX = FBM_Bar1 and #FixedFrame = FBM_FrameBar1. It is important to understand that *within the four bar mechanism context*, automatic design parameters were defined as #PFX = FBM and #FixedFrame = FrameFBM. Although the names of the design parameters are identical, they are defined in two distinct contexts. The rules of recursive search for design parameters defined earlier enable the management of identically named parameters with distinct values in distinct contexts.

Finally, as shown in fig. 3, the articulated bar assembly, Bar1, is connected to vertices VertexM and FBM_Vertex12. These two vertices are specified as #VertexA, #PFX <Vertex12> ~+, respectively. Since this definition occurs in the context of the four bar mechanism definition, #VertexA, is interpreted as VertexM. Indeed, as discussed above, #VertexA is an automatic design parameter generated by the definition of the four bar mechanism assembly as #VertexA = VertexM. The second vertex definition is interpreted as FBM_Vertex12. Of

course, the following automatic design parameters will be generated `#VertexA = VertexM`, `#PointA = PointM`, and `#PointAxyz` will store the coordinates of `PointM` in fixed frame `FBM_FrameBar1`. These first two design parameters happen to be identical to those defined for the four bar mechanism assembly because the first vertices of both `FBM` and `Bar1` assemblies happen to be identical in this case. Note that the last design parameter, `#PointAxyz`, however, stores the coordinates of `PointM` in fixed frames `FrameFBM` and `FBM_FrameBar1`, in the contexts of assemblies `FBM` and `Bar1`, respectively. Similarly, the following automatic design parameters will be defined `#VertexB = FBM_Vertex12`, `#PointB = FBM_Point12` (assuming that `FBM_Point12` is associated with `FBM_Vertex12`), and `#PointBxyz` stores the coordinates of `FBM_Point12`.

3 Design parameters

Design parameters are listed in design parameter files included in the definition of an assembly. Each design parameter has a unique name that must start with the special character "#" and is associated with a specific value that can be of three different types

1. *String values.* Any sequence of alphanumeric characters. The value of the design parameter is introduced by the keyword *@STRING_VALUE*.

2. *Floating point values.* A single floating point number. The value of the design parameter is introduced by the keyword *@DOUBLE_VALUE*.

3. *Floating point arrays.* An array of three floating point numbers. The value of the design parameter is introduced by the keyword *@VECTOR_VALUE*.

In the example below, design parameters `#Despar1`, `#Despar2` and `#Despar3`, of types string value, floating point value and floating point array, respectively, are assigned the values "`PropertyBar1`," "`5.0E+00`" and "`5.0E+00, 3.5, -6.2e-05`," respectively.

@DESIGN_PARAMETERS_DEFINITION {

> *@DESIGN_PARAMETER_NAME* { `#Despar1` } *@STRING_VALUE* { `PropertyBar1` }
>
> *@DESIGN_PARAMETER_NAME* { `#Despar2` } *@DOUBLE_VALUE* { `5.0E+00` }
>
> *@DESIGN_PARAMETER_NAME* { `#Despar3` } *@VECTOR_VALUE* { `5.0E+00, 3.5, -6.2e-05` }

}

4 Expression calculator

Once design parameters have been defined, they can be used as part of the input stream within the master input file, the data files and the template files defining an assembly. Their values can be combined using the expression calculator. Note the use of the expression calculator uses the reverse Polish notation.

The expression calculator can deal with string or floating point arguments, but the two types of arguments cannot appear in a single expression. The operators that can be used for string and floating point arguments are detailed next.

4.1 Example 1: string operations

1. Assume the following string value design parameters have been defined

   @DESIGN_PARAMETER_NAME {#PFX} @STRING_VALUE {Bar1_}

   @DESIGN_PARAMETER_NAME {#VertexB} @STRING_VALUE {VertexB3}.

2. If the following line appears in the input stream

   @CONNECTED_TO_VERTICES {#PFX <VertexA1> ~+, #VertexB},

3. it will be replaced by

   @CONNECTED_TO_VERTICES { Bar1_VertexA1, VertexB3 }

The expression "#PFX <VertexA1> ~+" evaluates to "Bar1_VertexA1" because the design parameter #PFX is replaced by "Bar1_", <VertexA1> is used to indicate the string "VertexA1," and finally, "~+" is the string concatenation operator. Note the following notational conventions: design parameters start with the symbol #, operators start with the symbol ~, and string constants appear between triangular brackets, <...>.

4.2 Example 2: vector operations

1. Assume the following floating point array design parameters have been defined

   @DESIGN_PARAMETER_NAME {#PointBxyz} @VECTOR_VALUE { 0.0, 0.12, 0.0 },

   @DESIGN_PARAMETER_NAME {#PointCxyz} @VECTOR_VALUE {0.24, 0.15, 0.0},

   @DESIGN_PARAMETER_NAME {#I3} @VECTOR_VALUE {0.0, 0.0, 1.0}.

2. If the following line appears in the input stream

   @ORIENTATION_E2 {#I3 #PointCxyz #PointBxyz ~- ~CROSS ~UNIT},

3. it will be replaced by

   @ORIENTATION_E2 {-0.124, 0.992, 0.0}.

The expression "#I3 #PointCxyz #PointBxyz ~- ~CROSS ~UNIT" is equivalent to the following mathematical statement

$$\frac{\tilde{\imath}_3(\underline{x}_C - \underline{x}_B)}{\|\tilde{\imath}_3(\underline{x}_C - \underline{x}_B)\|} = \frac{\tilde{\imath}_3(0.24, 0.03, 0.0)}{\|\tilde{\imath}_3(0.24, 0.03, 0.0)\|} = \frac{-0.03, 0.24, 0.0)}{\|(-0.03, 0.24, 0.0)\|} = (-0.124, 0.992, 0.0).$$

4.3 Example 3: vector operations

1. Assume the following floating point array design parameters have been defined
   @DESIGN_PARAMETER_NAME `{#I1}` @VECTOR_VALUE { 1.0, 0.0, 0.0 },
   @DESIGN_PARAMETER_NAME `{#I2}` @VECTOR_VALUE { 0.0, 1.0, 0.0 },
   @DESIGN_PARAMETER_NAME `{#BeamAngle}` @DOUBLE_VALUE `{15.0}`,
   @DESIGN_PARAMETER_NAME `{#DegToRad}` @DOUBLE_VALUE `{0.017453292}`,
   @DESIGN_PARAMETER_NAME `{#Beam1Length}` @DOUBLE_VALUE `{2.4}`.
2. If the following line appears in the input stream
   @ORIGIN `{#I1 #BeamAngle #DegToRad ~* ~COS ~*`
   `#I2 #BeamAngle #DegToRad ~* ~SIN ~* ~+ #Beam1Length ~*}`,
3. it will be replaced by
   @ORIGIN `{2.318, 0.621166, 0.0}`.

The expression is equivalent to the following mathematical statement

$$\left[\cos(\frac{\pi\alpha}{180})\bar{\imath}_1 + \sin(\frac{\pi\alpha}{180})\bar{\imath}_2\right] L = (2.318, 0.621166, 0.0),$$

where `#DegToRad` $= \pi/180$, $\pi\alpha/180 = 0.261799$ is the value of `#BeamAngle` in radians, and L is `#Beam1Length`.

4.4 String operators

One single operator can be used for expression involving string arguments: the concatenation operator, `~+`.

4.5 floating point operators

A number of operators can be used for expression involving floating point arguments. They are categorized into **unary** or **binary** operators, which involve a single, or two arguments, respectively. Tables 1 and 2 list the available unary and binary operators, respectively.

Operator	argument a	argument $\underline{a}^T = (a_1, a_2, a_3)$
~ACOS	$\cos^{-1} a$	$(\cos^{-1} a_1, \cos^{-1} a_2, \cos^{-1} a_3)$
~ASIN	$\sin^{-1} a$	$(\sin^{-1} a_1, \sin^{-1} a_2, \sin^{-1} a_3)$
~ATAN	$\tan^{-1} a$	$(\tan^{-1} a_1, \tan^{-1} a_2, \tan^{-1} a_3)$
~COS	$\cos a$	$(\cos a_1, \cos a_2, \cos a_3)$
~NORM	$\|a\|$	$\|\underline{a}\|$
~PUSH	pushes argument in stack	pushes argument in stack
~SIN	$\sin a$	$(\sin a_1, \sin a_2, \sin a_3)$
~TAN	$\tan a$	$(\tan a_1, \tan a_2, \tan a_3)$
~UNIT	N/A	$\underline{a}/\|\underline{a}\|$

Table 1: List of unary operators. All trigonometric functions assume input argument to be in radians. All inverse trigonometric functions return output in radians.

Operator	arguments a, b	arguments $a, \underline{b}$	arguments $\underline{a}, b$	arguments $\underline{a}, \underline{b}$
~CROSS	N/A	N/A	N/A	$\underline{a} \times \underline{b}$
~/	a/b	$\underline{b}/a$	$\underline{a}/b$	$(a_1/b_1, a_2/b_2, a_3/b_3)$
~DOT	N/A	N/A	N/A	$\underline{a} \cdot \underline{b}$
~-	$a - b$	$a\ \underline{1} - \underline{b}$	$\underline{a} - b\ \underline{1}$	$\underline{a} - \underline{b}$
~+	$a + b$	$a\ \underline{1} + \underline{b}$	$\underline{a} + b\ \underline{1}$	$\underline{a} + \underline{b}$
~*	ab	$a\underline{b}$	$b\underline{a}$	$(a_1 b_1, a_2 b_2, a_3 b_3)$

Table 2: List of binary operators. Note the short hand notation $\underline{1}^T = \lfloor 1, 1, 1 \rfloor$.