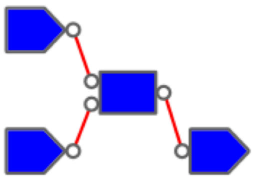
Dataflow programming method



Dataflow programming is a style of programming that models computation as the flow of data through interconnected nodes and is processed by them according to their internal algorithm. This makes it easier for developers to visualize and understand complex computations since they can trace all operations back to input values.

Introduction

This document describes the Proavatar's *dataflow programming*[1] method which can be used to implement mathematical algorithms that process timestamped sensor data and generate specific output values that are to be reported or displayed to a user.

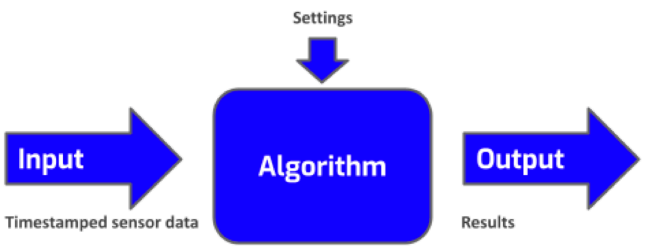


Figure 1: Basic set-up of a sensor data processing algorithm.

When such algorithms are part of a user-facing application, for example on a smartphone, hardcoded algorithms would require an update for each little change to the algorithm. In contrast, with Proavatar's dataflow programming method a *diagram* can be specified as a JSON text file that can be easily shared, exchanged, stored and accessed in online databases.

In such a diagram, the algorithm is implemented by specifying the inputs, outputs and the sequence of interconnected functions to calculate the outputs from the inputs. Whilst theoretically these dataflow diagrams can be written in a text-editor, Proavatar offers graphical tools to visualize dataflow diagrams and construct them in a simple and intuitive manner.

[1] Wikipedia - Dataflow programming

Algorithm architecture

Consider an algorithm that takes one or more sensor data streams as *input* and calculates some kind of *output*. These outputs can then be used by a connected *reporting system* (the executing application), which may include a graphical user interface (GUI).

Then an algorithm is constructed - or implemented - by creating a *dataflow diagram* that uses *input streams* and a number of interconnected *functions* to calculate the value of one or more *variable outputs* of specific types (variable types are discussed in the next chapter). Any settings of the diagram are implemented by using *constants* and these can be changed by the executing application.

The architecture of an algorithm is given in figure 2.

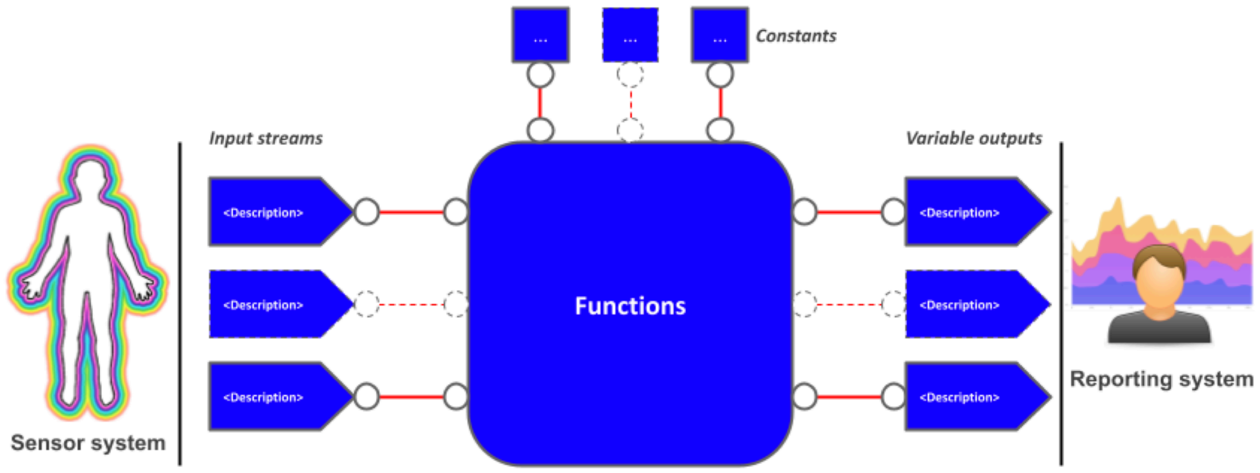


Figure 2: Algorithm architecture.

In the following sections the components of a dataflow diagram are discussed.

Timestamped variables

A dataflow diagram uses several timestamped variables. These variables can contain basic values, compound data types or arrays.

Basic values

In table 1 the variable types are defined that can be used in a dataflow diagram to store the basic values commonly available in most programming languages. The symbol is used to identify the value in the diagram and discussed further in (see "*Timestamped variables in the diagram*").

Table 1: Dataflow basic values.

Name	Description	Symbol
Float[2]	Floating point value, used for angles and other values.	**f**
Integer	Integer value, used for indices and numbers.	**i**
Boolean	Logical value, i.e. *true* or *false*.	**b**
String	A piece of text.	**$**

Compound data types

Since the dataflow programming is specifically suited for processing timestamped sensor data, it is very useful to also have compound data types[3]. The ones that are available in the dataflow programming method are defined and given in table 2.

Table 2: Dataflow compound data types.

Name	Description	Symbol
Vector	A 3D position where the 3 coordinates x, y and z are floats..	**v**
Orientation	An orientation, expressed as a quaternion with the imaginary coordinates x, y, z and real value w as floats.	**q**
Sequence	An array of timestamped floats, i.e. $[(t_0, f_0), \dots, (t_{N-1}, f_{N-1})]$.	**s**

Arrays

Next to the specified types, it is also possible to create *arrays*. These are also timestamped variables but contain a collection of values or compound data. There is an array for each type.

Conforming to most programming languages, in a dataflow diagram an array is indicated by placing the symbol of the variable between square brackets, e.g. `[f]`. `[i]`, etc.

[2] A float is a data type composed of a number that is not an integer, because it includes a fraction represented in decimal format.

[3] Composite data type - Wikipedia

The elements in an array can be accessed (via functions) via a zero-based index, meaning that the first element has index 0.

Timestamped variable in diagrams

The symbol of a timestamped variable is used to indicate the type of a connector of a node in a diagram. An example is given in figure 3 where a function node (discussed further in "*Functions*") is shown with two input connectors and one output connector.

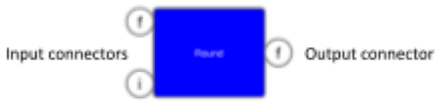


Figure 3: Typed connectors.

A timestamped variable has a number of relevant properties which are given in table 3 and that are used during the execution of the diagram (discussed in "*Diagram execution*").

Table 3: Relevant properties of a timestamped variable.

Property	Description
`value`	The actual value for which the type is defined by the type of the variable itself as give in table 1.
`updated`	Boolean indicating whether the value was updated.
`checked`	Boolean indicating whether the timestamped variable can be evaluated as its updated state will not change during the ongoing execution.
`timestamp`	Timestamp at which the variable was last updated.

Nodes

All the elements in a diagram are called *nodes*. A node has one or more connectors, depending on the type. The possible node types are given in table 4.

Table 4: Node types.

Type	Description	Input connectors	Output connector
Function	Generates a certain output by performing an internal calculation using its inputs.	Multiple	✓
Constant	Stores a constant value.	x	✓
Input stream	Generates an output when it is updated by an external process.	x	✓
Variable output	Store the result of the execution of the dataflow diagram.	1	x
Container	A node that can be constructed by the diagram editor to assist in visually simplifying the diagram and can contain a number of internal function nodes, constant nodes or nested containers.	Multiple	✓

For visualization purposes, a node in the diagram can be given a specific color. The following options are available:

Color	Hexadecimal code
blue	#0000FF
red	#FF0000
black	#000000
brown	#A52A2A
cyan	#00FFFF
gray	#808080

Color	Hexadecimal code
green	#008000
magenta	#FF00FF
orange	#FFA500
purple	#800080
yellow	#FFFF00
White	#FFFFFF

Also, for all nodes, except function nodes, a 'description' can be edited which appears in the body of the node when displayed. For function nodes the description specifies the internal algorithm that it implements.

Functions

The core components of a dataflow diagram are *function nodes*. There is a large variety of function nodes which can be selected from a list and have one or more inputs (depending on the type of function) and a *single* output. In figure 4 an example of such a function node is given with two input connectors and one output connector.

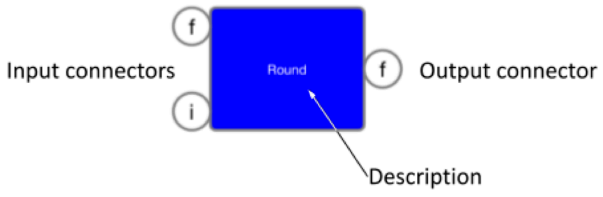


Figure 4: Example of a function node.

The inputs and output are of specific variable types. For example, the function node in figure 4 implements the functionality to round a float (f) to a specified number of decimal places, specified by the integer (i) input.

Since the concept of a data 'flow' is used, in general a function node calculates (i.e. *updates*) its output when all inputs are updated. In the concept of dataflow programming, the dataflow 'assumes' sequential updates of the inputs. Therefore, the function node will 'wait' for the updates of all inputs. When the output is calculated, the inputs are set back to their "not updated" state. This process is illustrated in figure 5.

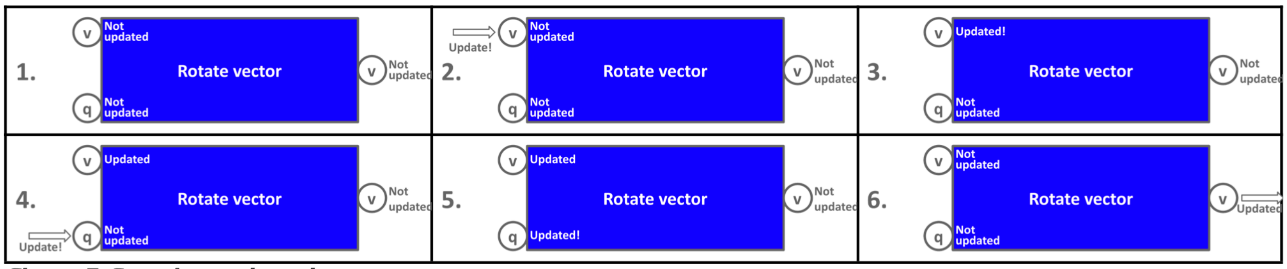


Figure 5: Function node update process.

Note that when one of the input updates a number of times consecutively, these updates are not used (i.e. lost) unless the other input(s) are also updated. Also, the timestamp of the generated output will always have the timestamp of the last updated input value.

There are also function nodes that use the value of the first input to determine how the other inputs are to be used. A simple example is the "*Gate*" function as illustrated in figure 6.

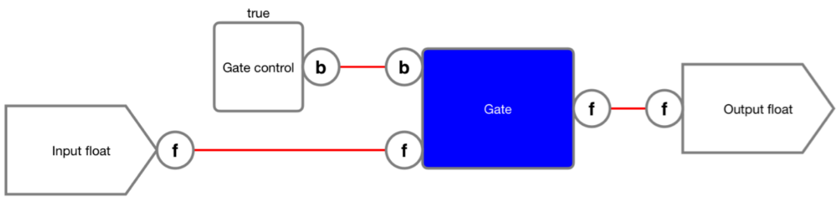


Figure 6: "Gate" function.

The "*Gate*" function will forward an updated second input only when the first input is true.

Generic functions

In the dataflow programming method presented here, *strict typing* is used to prevent type mismatch errors. As such for most functions the connectors have a specific fixed type. However, strictly this would mean that for each specific operation that can be performed on several types – for example addition – separate functions would be required. This is obviously impractical. Therefore, the concept of *generic functions* is introduced that have *generic* connectors for which the type is not specified yet prior to connecting it.

There are the following generic indicators.

Table: Generic indicators.

Name	Description	Indicator
Generic variable	An untyped timestamped variable that can be a value, compound data type or an array.	*****
Generic value	An untyped timestamped value or compound data type, not being an array.	**?**
Generic array	An array for which the type is not yet specified.	**[?]**

As an example, consider the function node in figure 7 which performs an addition of two values. The type will be specified as soon as one of the connectors is connected (also see "*Connections*").

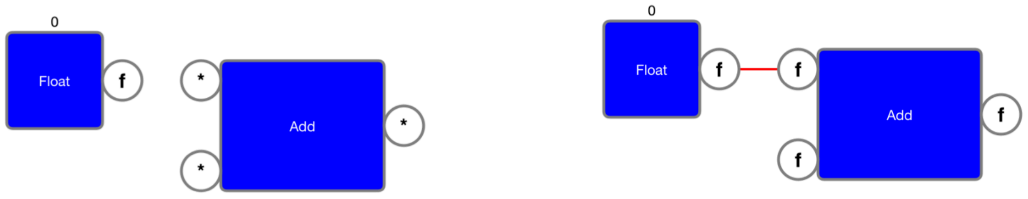


Figure 7: Example of a function with generic variable inputs that are specified when connected.

The connectors do not all have to belong to the same category. For example, in figure 8 two other examples are given of generic functions.

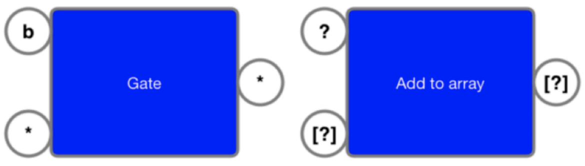


Figure 8: Generic functions.

Constants

A *constant* node can hold a fixed value of a certain variable type which is specified when the dataflow diagram is designed. In figure 9 an example of a constant node is given.

In the visualization of a constant node, the *value* is displayed above the node except when the constant represents an array because that would be impractical to display as a string. The description "*Float*" in the example in figure 9 can be changed, for example to describe the purpose of the constant.

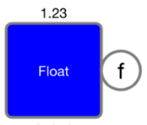


Figure 9: Constant node.

The output of a constant node can be connected to the input of one or more function nodes. When this is done, a function node will obviously not 'wait' for that input to be updated but calculate its output as soon as all the inputs are updated that are not connected to a constant node.

Typical usages of constant nodes could be the specification of axes, size of buffers, indication of indices or other 'settings' of the algorithm. In figure 10 an example is given where a vector constant is used to determine the direction of the z-axis as a result of the rotation of the left upper leg (made available via an input stream which node type is discussed in "*Input streams*").

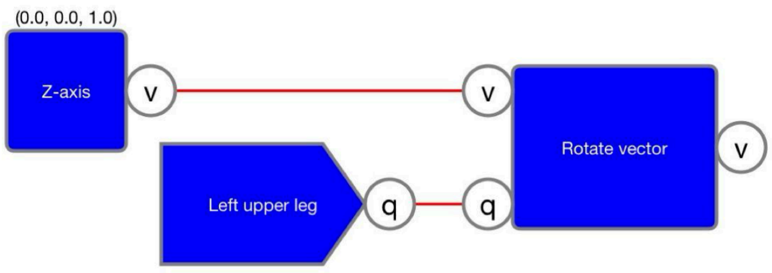


Figure 10: Example of a vector constant.

While the output of a constant node is a timestamped variable – and can be handled accordingly – the timestamp is obviously not defined and as such is set to `NaN`[4] ("not-a-number").

[4] NaN - Wikipedia

Input streams

An *input stream* node offers a certain data stream on its output. An example of input stream nodes is given in figure 11 where the input stream node with the *name* "*Left upper leg*" supplies a timestamped orientation (q) and the one with "*Right knee*" supplies a position (v).

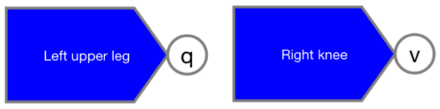


Figure 11: Input stream nodes.

Because the input stream node is the beginning of the dataflow diagram, it has no inputs. Instead, the input stream node is updated by the application whenever new data is available, e.g. at each timestep.

There is an input stream node for all variable types.

Note that in the diagram it is possible and allowed to have any number of similar input stream nodes (i.e. with the same description and same type) which are then updated with the same value by the application. This can be used as a convenient way of using the same input at several locations in the diagram.

By default, the value that is supplied to the input stream node by the application is also offered as an output of the diagram (see "*Variable outputs*"). This is graphically indicated by the extra line printed at the right of the node as illustrated in figure 12. When forwarding is disabled, like is the case in the two input stream nodes in figure 11, the line is not shown.

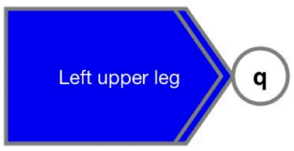


Figure 12: Input stream node with forwarding enabled.

Variable outputs

To be able to present the results of the execution of a dataflow diagram (also see "*Diagram execution*"), a *variable output* node is defined. There is a variable output node for each variable type.

In figure 13 the setup of a variable output node is shown. To be able to identify the value, the description "*Float*" can be changed.

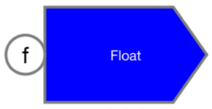


Figure 13: Variable output node.

When the input of a variable output node is updated, the connected system is informed with the description and the value. It is then up to the implementation of the connected system to use (or ignore) the value.

Note that a variable output node can also be of great help when debugging algorithms by supplying intermediate values, e.g. much like a debug statement in normal coding practice.

The description of a variable output node must be unique in the diagram and no other variable output node must exist with the same description, even if the types are different.

To indicate that the description of a variable output node is *not* unique, a red line is printed at the right side of the node as illustrated in figure 14.

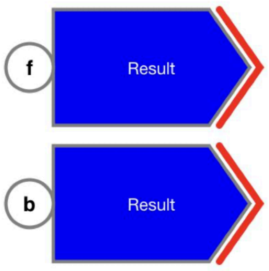


Figure 14: Error line indicating that the description of the variable output node is not unique.

Connections

To implement the functionality of creating a chain of executed function operations, i.e. to let data 'flow', a connection can be created between the output of a node and the inputs of one or more other nodes.

Creating a connection

To implement the functionality of creating a chain of executed function operations, i.e. to let data 'flow', the output of a node can be connected to the input of another node in case of matching variable types or either one being a generic connector. An example of interconnected function nodes is given in figure 15.

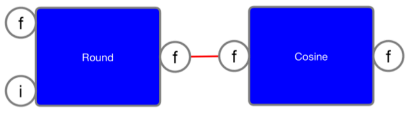


Figure 15: Example of interconnected function nodes where the data 'flows' from left to right.

To create a connection an output and input connector must be selected that are of the same type or either one is a generic connector. For example, in figure 16 the last connection will be created when the float input connector of the variable output node is selected. An input can only be connected to one output, but an output can be connected to any number of inputs.

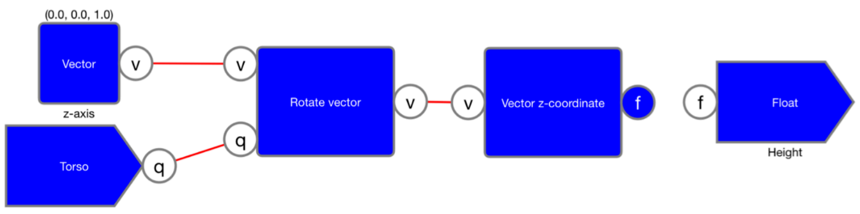


Figure 16: Connecting nodes.

Unsupported connections

In "*Generic functions*" the concept was introduced in which one or more connectors of specific functions don't have the type of one or more of their connectors of a predefined type. However, not all generic functions support all types. For example, the "*Divide*" function does not support booleans and strings, or their arrays since performing a division on a boolean or string would have no meaning. For a function that has generic connectors it is possible to connect these types but the connector will be colored red to indicate that the type is not supported. This is illustrated in figure 17.

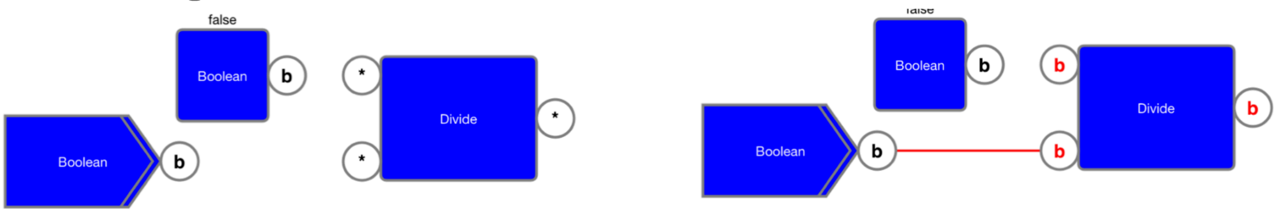


Figure 17: Connecting unsupported types to a generic function.

Note that the function will not output any updates when the inputs are updated in this case.

Containers

Potentially, dataflow diagrams can become quite large. This makes it difficult to quickly assess the implemented functionality. It would be helpful if certain functionality which is implemented in a set of interconnected nodes could be collapsed into a 'container'. Just like a regular function, such a container would have a single output and one or more inputs.

Containers can be created by selecting a number of interconnected nodes, being functions, constants or other container nodes.

The set of selected nodes that must become a container must comply with the following requirements:

- The set does not contain any input stream or variable output nodes.
- There is only one node for which its output is not connected to any other nodes or has connections to nodes outside the set of selected nodes.

In the diagram in figure 19 an example is given of a set of selected nodes (indicated by the red border) that comply with these requirements.

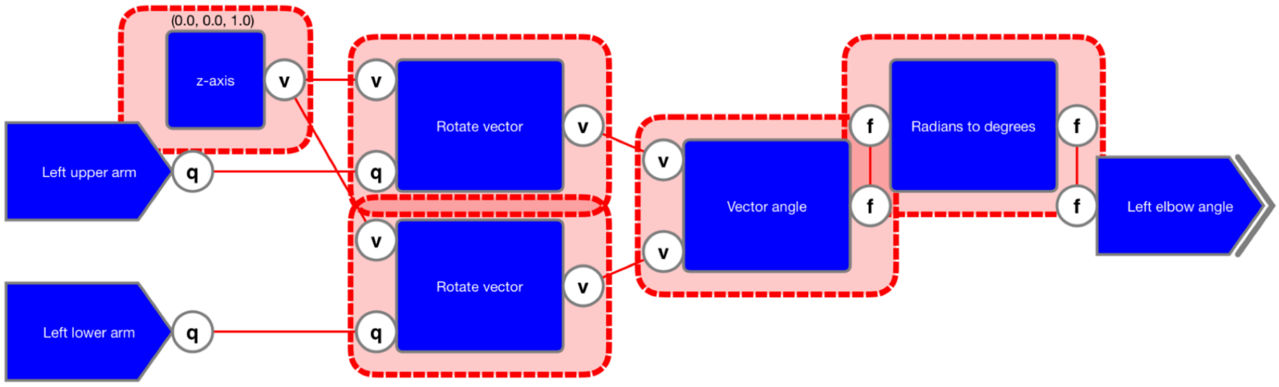


Figure 19: Set of nodes that can be collapsed into a container.

The set of selected nodes can be collapsed to a container. The default description "*[Container]*" can be changed to something more descriptive.

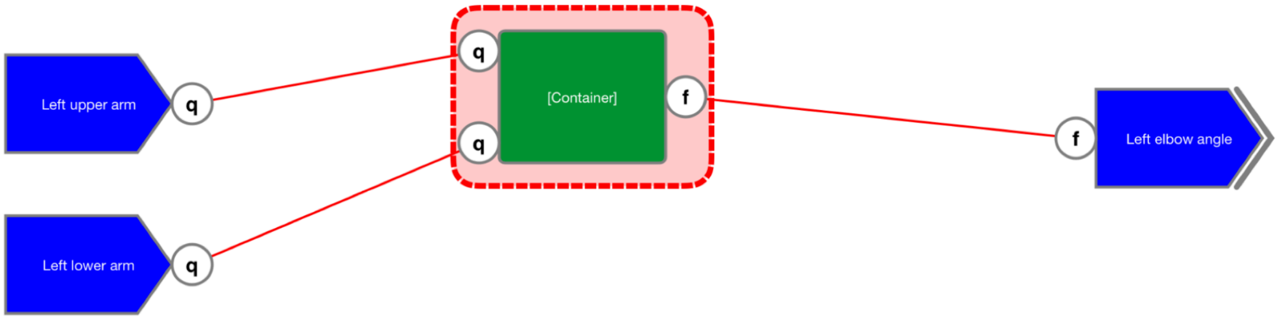


Figure 20: Newly created container node.

A container is only for visual simplification of the diagram. It has no other effect on the data processing. As such, the container still contains the interconnected nodes and can be expanded again if so required.

Sub-diagrams

Another method to get a better overview of the algorithm is to be able to divide the diagram in sub-diagrams that can be displayed separately as individual diagrams.

To do this, a sub-diagram can be specified with a name describing the functionality implemented and the nodes that are contained in it.

Similarly to containers, sub-diagrams only serve visualization purposes and have no other effect on the data processing.

In the editor, a sub-diagram can be created and filled with nodes. A specific sub-diagram can be shown by selecting a tab with the name of the sub-diagram.

Diagram local loops

The application is responsible for setting the values of the input stream nodes and having the diagram update the variable output nodes. However a special situation occurs when variable output nodes are specified that have the same description and variable type as one or more of the input stream nodes. An example of such a diagram is shown in figure 21.

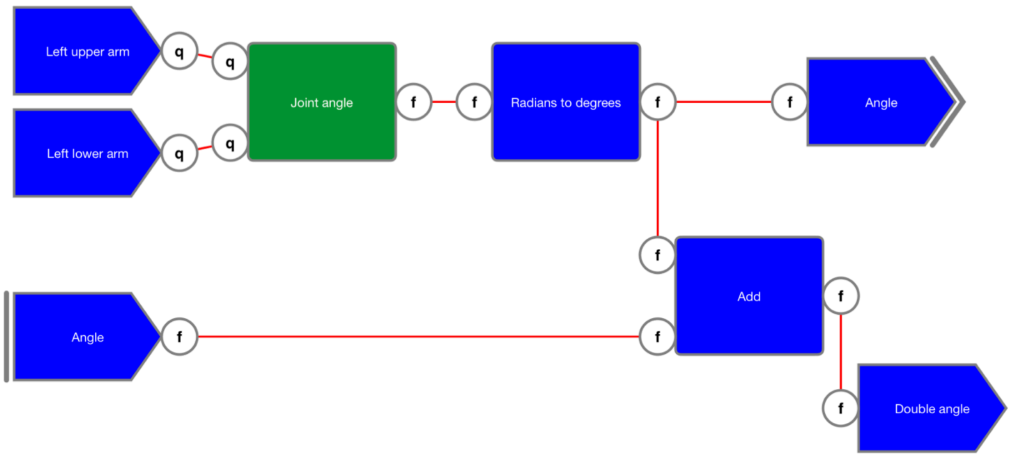


Figure 21: Diagram with an internal looped variable.

In the diagram in figure 21 the external sensor system could update the input stream nodes with the descriptions "*Left upper arm*" and "*Left lower arm*" first and when the execution of a diagram results in an update of the variable output node with the description "*Angle*", the input stream node with the same description and type is updated as well.

As can be seen in figure 21, for a variable output node the loop is indicated with the extra line printed to at the right of the node while at the input stream node the extra line is printed at the left. Note that, for a looped input stream node, the forwarding (see "[Input streams]") is always *disabled* as it can be seen as an internal variable of the diagram.

When there are several same input stream nodes with the same type and description as an variable output node, all receive the same value. As such it makes sense to have a convenience function that asks the user whether the description of the connected input stream nodes needs to be changed when the description of a looped variable output node is changed.

In most cases where the update of a variable output node is used to update one or more input stream nodes it would not make much sense to offer the update of the variable output node also as output of the diagram to the application. Therefore, a variable output node can be hidden, which is indicated by the ✖ printed at the right of the node as illustrated in figure 22.

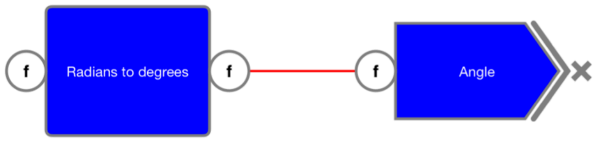


Figure 22: Hidden variable output node.

A more detailed description on how diagram loops are used during the execution of a diagram is given in the next chapter.

Diagram execution

To determine the output of a dataflow diagram at a certain timestamp, the diagram must be executed. This chapter describes this process because it is helpful for application developers to have a basic understanding of how this is done.

The execution of a dataflow diagram at a certain timestamp is started by the external application updating those input streams for which new data is available at a certain timestamp. This will update all the connected inputs.

The next step that is performed is updating the outputs of the constant nodes, which in turn will also update all the connected inputs.

Next, a loop is started that iterates over a list of function nodes and performs the following actions:

1. It checks if all inputs of the current node are checked.
2. If all inputs are not checked, it sets the 'allChecked' variable to false and continues to the next iteration of the for loop.
3. If all inputs are checked, it checks if all relevant inputs of the current node are updated.
4. If all relevant inputs are updated, it sets the 'updated' variable to true and runs the internal algorithm of the current node.
5. After the for-loop has completed its iterations, it checks the values of the 'updated' and 'allChecked' variables. If either of these variables is false, it repeats the loop. If both variables are true, it ends the loop.

This process can be written as pseudo-code:

```
updated = true
allChecked = false
WHILE updated OR NOT allChecked
    updated = false
    allChecked = true
    FOR node IN nodes["Function"]
        # check if all inputs of the function node are checked
        inputsChecked = checkInputs(node)
        IF NOT inputsChecked THEN
            allChecked = false
            CONTINUE
        END IF
        # check if all relevant inputs of the function node are updated
        relevantInputsUpdated = checkRelevantInputs(node)
        IF relevantInputsUpdated THEN
            updated = true
            # run the function node's internal algorithm
            result = runAlgorithm(node)
        END IF
    END FOR
END WHILE
```

Note that running the function node's internal algorithm uses the input values. Therefore it is necessary to *check* the input values first before using them. This is to support special function nodes that do not wait for all their inputs to be updated. For these function nodes, checking the input values ensures that its current value is up-to-date.

Depending on the result of running the function node's internal algorithm, the 'value' property of the timestamped variable associated with the output of the node could be set and when that happens, the generic part is called in which the 'updated' property of the node's output is set to true. It then enters a for-loop that iterates over the node's inputs. For each input, it sets the 'updated' property to false and updates the 'timestamp' property of the node's output to the maximum of its current value and the 'timestamp' property of the input.

After the first for-loop has completed its iterations, another for-loop is started that iterates over the node's connected inputs. For each connected input, it sets the 'value' and 'timestamp' properties to the corresponding properties of the node's output and sets the 'updated' property to true.

These actions can be specified by the following pseudo-code:

```
node.output.updated = true
node.output.timestamp = 0
FOR input IN node.inputs
    input.updated = false
    node.output.timestamp = MAX( node.output.timestamp, input.timestamp )
END FOR
FOR input IN node.connectedInputs
    input.value = node.output.value
    input.timestamp = node.output.timestamp
    input.updated = true
END FOR
```

The process will continue until there are no more updates. At that point it is checked whether the input of any of the variable outputs was updated. If this is the case, the execution process is repeated, but now with the values of the updated variable outputs as input.

This allows for the implementation of diagram loops as discussed in "*Diagram loops*".

To prevent infinite loops, variable outputs will only be used once in this way.

Dataflow diagram file format

A dataflow diagram is stored in a file with the ‘dfd’ extension which is formatted as a JSON file. It has the following sections:

Section	Description	Type
`nodes`	Array of generic node information.	`[NodeInfo]`
`properties`	Array of properties that can differ per node class.	`[PropertyInfo]`
`connections`	Connections between nodes.	`[ConnectionInfo]`
`positions`	Positions of the nodes.	`[PositionInfo]`
`containers`	Array of the information of the defined containers.	`[ContainerInfo]`
`manualLines`	Optional array of strings containing the lines describing for example the purpose of this diagram.	`[String]`
`subdiagrams`	Optional list of sub-diagrams.	`[SubdiagramInfo]`

Node that the information contained in the “*positions*” and “*containers*” section is only required for visualization but not for execution.

The custom types specified above are described in the following sections.

NodeInfo

Parameter	Description	Type
`id`	Id of the node, unique within the file.	`Int`
`classString`	The class of the node, i.e. “*Input stream*”, “*Constant*”, “*Function*” or “*Variable output*”.	`String`
`typeString`	The type of the node within the class, for example “*Add*”	`String`
`outputType`	Optional parameter used to indicate the variable type of the output for function nodes with generic connectors.	`String?`

PositionInfo

Parameter	Description	Type
`id`	Id of the node for which this property must be set.	`Int`
`x`	X-coordinate of the position of the node.	`Float`
`y`	Y-coordinate of the position of the node,	`Float`

ContainerInfo

Parameter	Description	Type
`description`	Description given to the container.	`String`
`nodeIds`	Array of ids of the contained nodes.	`[Int]`
`color`	Optional parameter specifying the color.	`String?`
`x`	X-coordinate of the position of the container.	`Float`
`y`	Y-coordinate of the position of the container,	`Float`
`containers`	List of nested containers.	`[ContainerInfo]`

ConnectionInfo

Parameter	Description	Type
`outputNodeId`	Id of the node for which the output connector is connected.	`Int`
`inputNodeId`	Id of the node for which one of the input connectors is connected.	`Int`
`inputIndex`	Index of the input connector that is connected.	`Int`

PropertyInfo

Parameter	Description	Type
`id`	Id of the node for which this property must be set.	`Int`
`propertyString`	Identifier of the property.	`String`
`valueString`	Textual representation of the value.	`String`

The following `propertyStrings` are defined:

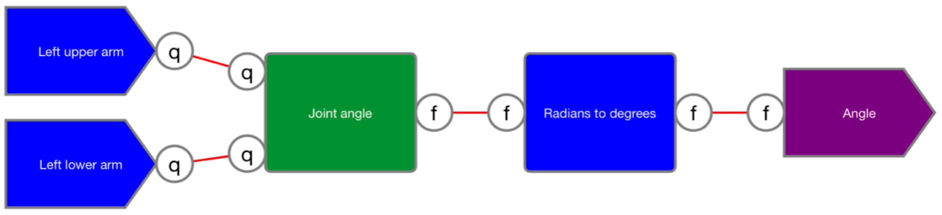
- "*OutputValue*" = specifies the value of a constant.
- "*Color*" = specifies the name of the color.
- "*Description*" = specifies the text as displayed in the body of the node.
- "*Forwarding*" = specifies whether the input stream node forwarding is enabled.
- "*Hidden*" = specifies whether the variable output node update is hidden as output.

SubdiagramInfo

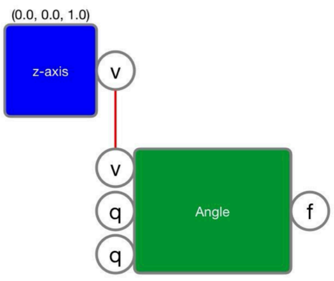
Parameter	Description	Type
`description`	Description given to the container.	`String`
`nodeIds`	Array of ids of the contained nodes.	`[Int]`

Example

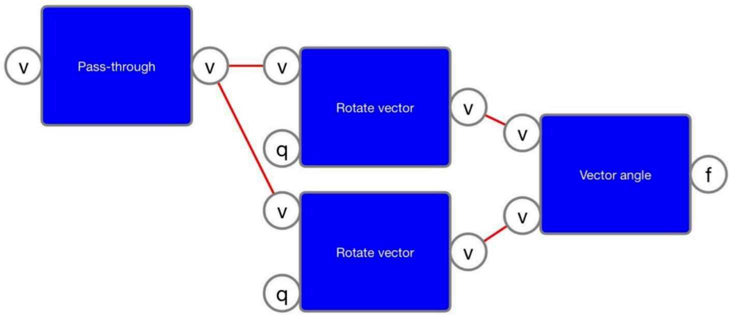
Consider the following diagram.



The contents of the "*Joint angle*" container is given below.



The contents of the "*Angle*" container is given below.



JSON

The corresponding diagram file formatted as a JSON string is as follows.

```
{"connections":[{"inputIndex":0,"outputNodeId":1,"inputNodeId":3},{"inputIndex":1,"outputNodeId":2,"inputNodeId":3},{"inputIndex":0,"outputNod
eId":3,"inputNodeId":5},{"inputIndex":0,"outputNodeId":4,"inputNodeId":1},{"inputIndex":0,"outputNodeId":4,"inputNodeId":2},{"inputIndex":0,"o
utputNodeId":5,"inputNodeId":0},{"inputIndex":1,"outputNodeId":6,"inputNodeId":1},{"inputIndex":1,"outputNodeId":7,"inputNodeId":2},{"inputInd
ex":0,"outputNodeId":8,"inputNodeId":4}],"properties":[{"id":0,"propertyString":"Description","valueString":"Angle"},{"id":0,"propertyString":
"Color","valueString":"purple"},{"id":6,"propertyString":"Description","valueString":"Left upper
arm"},{"id":7,"propertyString":"Description","valueString":"Left lower
arm"},{"id":8,"propertyString":"Description","valueString":"z-axis"},{"id":8,"propertyString":"OutputValue","valueString":"(0.0, 0.0,
1.0)"}],"containers":[{"nodeIds":[8],"x":430,"y":210,"containers":[{"nodeIds":[1,2,3,4],"x":765,"y":545,"containers":[],"description":"Angle"}
],"description":"Joint angle"}],"manualLines":["This is an example of a dataflow diagram with nested
containers."],"positions":[{"id":0,"x":940,"y":210},{"id":1,"x":980,"y":520},{"id":2,"x":980,"y":660},{"id":3,"x":1220,"y":590},{"id":4,"x":73
0,"y":490},{"id":5,"x":690,"y":210},{"id":6,"x":140,"y":150},{"id":7,"x":140,"y":260},{"id":8,"x":615,"y":415}],"nodes":[{"id":0,"classString"
:"Variable output","typeString":"Float"},{"id":1,"classString":"Function","typeString":"Rotate
vector"},{"id":2,"classString":"Function","typeString":"Rotate vector"},{"id":3,"classString":"Function","typeString":"Vector
angle"},{"id":4,"classString":"Function","typeString":"Pass-through","outputType":"Vector"},{"id":5,"classString":"Function","typeString":"Rad
ians to degrees"},{"id":6,"classString":"Input stream","typeString":"Orientation"},{"id":7,"classString":"Input
stream","typeString":"Orientation"},{"id":8,"classString":"Constant","typeString":"Vector"}]}
```