LUMASS – User Guide

Author: Alexander Herzig
Version: 0.3 (April 2017)

Licence

Acknowledgement

LUMASS development has been partly funded by the New Zealand Ministry of Business, Innovation and Employment's Science and Innovation Group. The development of the open source version of LUMASS was enabled and massively supported by research projects led by Daniel Rutledge (Manaaki Whenua – Landcare Research New Zealand Ltd).

Disclaimer

This document is not an introduction to spatial system dynamics modelling or GIS.

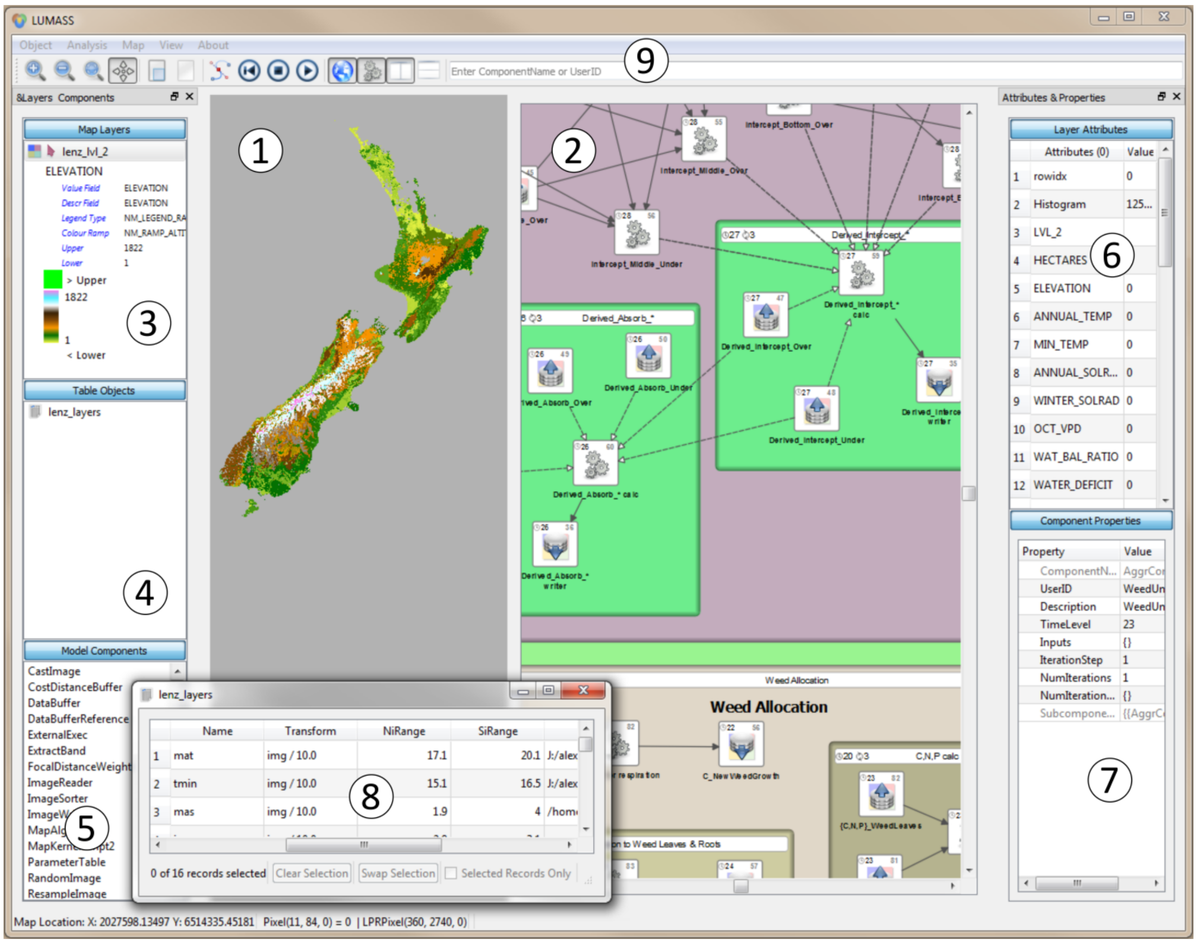
Contents

What is LUMASS?

LUMASS is a **L**and **U**se **MA**nagement **S**upport **S**ystem and is designed to provide support for two high level aspects of land management: i) land use impact assessment and ii) spatial planning. The former aspect is supported by LUMASS' spatial system dynamics modelling framework, whereas the latter is supported by LUMASS' spatial optimisation component (s. SpatialOptimisationHowTo.pdf).

LUMASS is mainly focused on processing and displaying raster data. However, it also provides selected functionality of displaying (polygon) vector data and 3D point clouds. Spatial optimisation scenarios can also be run on polygon vector layers.

Graphical User Interface (GUI)



	Attributes (0)	Value
1	rowidx	0
2	Histogram	125...
3	LVL_2	
4	HECTARES	
5	ELEVATION	0
6	ANNUAL_TEMP	0
7	MIN_TEMP	0
8	ANNUAL_SOLR...	0
9	WINTER_SOLRAD	0
10	OCT_VPD	0
11	WAT_BAL_RATIO	0
12	WATER_DEFICIT	0

Property	Value
ComponentN...	AggrCo
UserID	WeedUn
Description	WeedUn
TimeLevel	23
Inputs	{}
IterationStep	1
NumIterations	1
NumIteration...	{}
Subcompone...	{{AggrC

lenz_layers

	Name	Transform	NiRange	SiRange	
1	mat	img / 10.0	17.1	20.1	J:/alex
2	tmin	img / 10.0	15.1	16.5	J:/alex
3	mas	img / 10.0	1.9	4	/home

0 of 16 records selected   Clear Selection   Swap Selection   Selected Records Only

Map Location: X: 2027598.13497 Y: 6514335.45181 | Pixel(11, 84, 0) = 0 | LPRPixel(360, 2740, 0)

Figure 1 LUMASS graphical user interface

1. **Map View:** 2D/3D Display area for raster and vector layers as well as for point clouds
2. **Model View:** Visual modelling environment
3. **Map Layers:** Table of contents and legend configuration for layers displayed in the map view
4. **Table Objects:** Table of contents for stand-alone tables (s. Table view)
5. **Model Components:** List of available model process components in the modelling environment
6. **Layer Attributes:** List of map layer attribute values for a given point in the map (view)
7. **Component Properties:** Properties of a given model component
8. **Table View:** Table display and processing interface
9. **Menu Bar & Tool Bar**

10. **Notification Area:** (not shown in Fig. 1) displays information, warning, and error messages related to processing and modelling tasks

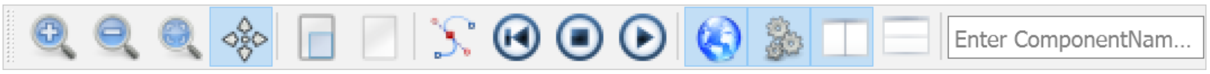


Figure 2 LUMASS tool bar

1. Zoom in map/model view
2. Zoom out map/model view
3. Zoom to map/model content
4. Pan map/model
5. Select features/components
6. Clear selection (features / components)
7. Link model components
8. Reset model
9. Stop model execution
10. Execute model
11. Display/hide map view
12. Display/hide model view
13. Stack main views horizontally
14. Stack main views vertically
15. Find/zoom to model component (by model component name or UserID)

LUMASS provides a typical desktop user interface that embraces the use of drag & drop and context menus. So, if you want to accomplish a certain task and are in doubt of how to do it, try drag and & drop, e.g. to import a table, image, or model into the respective view areas. If you want to perform an action on a particular object, double check whether it provides a context menu (right click) offering object specific actions (e.g. map layer, table column, model component).

The user interface can be adjusted to best suit the current task at hand (i.e. mapping, or modelling). For example, the main views (Figs. 1-1, 1-2) can be individually hidden and displayed (Figs. 2-11, 2-12) adjusted in their size (by using their separating slider), or stacked vertically or horizontally (Figs. 2-13, 2-14). The display areas left and right of the main views (Fig. 1, `Layers & Components` and `Attributes and Properties`), as well as the not displayed `Notifications` area are dockable windows and can be arbitrarily positioned around the centred main views or float on top of the main user interface. Layer attribute tables and stand-alone tables (Fig. 1-8) are displayed in their own top level window independent of the main user interface.

The individual content and property display areas (Figs. 1-3 to 1-7) can be collapsed and unfolded individually by clicking on their respective title button showing their name (e.g. `Table Objects`, Fig. 1-4). The `View` menu provides the view modes `Map View Mode` and `Model View Mode`, which configure the associated display areas for mapping and modelling tasks respectively.

Map display

Layer types

LUMASS supports the display of three different types of spatial layers:

- Raster (2D image) (multi-band and attribute table support)
- Vector (polygon only)
- Point cloud (3D) (experimental)

Data formats

The supported data formats for raster and vector layers are largely defined by the supported formats of the underlying GDAL library used for import and export of 2D raster and vector layers. Additionally, LUMASS supports the internally used VTK PolyData format (*.vtk) for vector layers. On Linux and if compiled with rasdaman support, LUMASS also provides a direct interface to the rasdaman array database for reading and writing multi-dimensional image data (including WCS metadata). Note that LUMASS provides experimental support for processing 3D images using the spatial modelling framework. Point cloud data may be provided as simple ASCII file, containing the comma separated x,y,z coordinates of a single point on an individual line. Please note that 3D mapping (and processing) functionality is still very much in development.

How to map raster layer values

0. In this example we map a multi-band raster layer without attribute table. It can be easily applied to single band raster layers without attribute tables as well.
1. **Navigate to the LUMASS `SampleData/data` folder:** Use your favourite filesystem browser to do this (i.e. Windows File Explorer, Dolphin, Nautilus, etc.).
2. **Load the layer:** Select the `LUMASS_icon_2048.kea` file and drag it into the `Map Layers` or `Map View` area
3. **Toggle layer visibility:** To toggle the layer's visibility, click on the coloured tile icon left of the layer name.
4. **Select the layer**: Select the layer by left clicking its layer name. The layer name should now be highlighted in blue. Note that left clicking a selected layer de-selects it.
5. **Observe mouse pointer position, pixel values and image resolution**: Move the mouse pointer over the map (i.e. the LUMASS logo) and watch the information displayed in the status bar (Fig. 3) at the very bottom of the GUI.

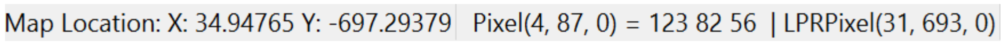

Map Location: X: 34.94765 Y: -697.29379 | Pixel(4, 87, 0) = 123 82 56 | LPRPixel(31, 693, 0)

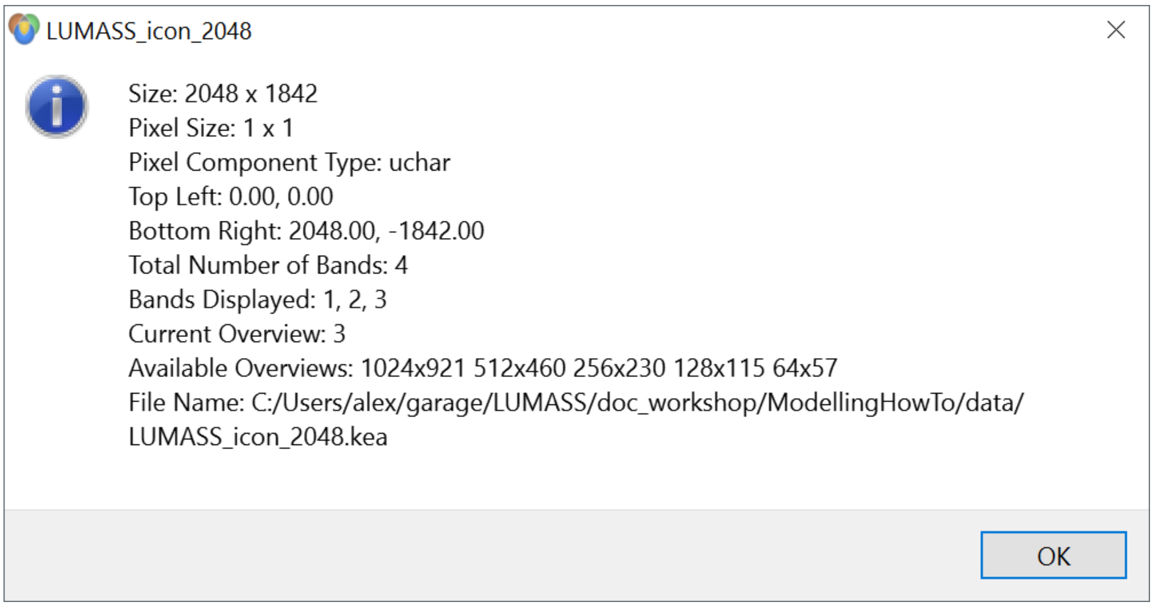
Figure 3 LUMASS status bar information

It displays from left to right the

a. location of the mouse pointer in map coordinates
b. (0-based) pixel index of the currently displayed image pyramid layer (i.e. reduced resolution image) of the image, and the
c. pixel value under the mouse pointer

   Note: In case of a multi-band image, LUMASS displays three bands of the given image as RGB colour image and displays the corresponding values in the status bar as RGB tuple.
d. the (0-based) pixel index of the largest possible region

6. **Alter displayed image resolution by zooming in and out:** Use the mouse wheel or the zoom tool to zoom in or out. Watch the display at the bottom and the displayed pixel index. When the displayed pixel index and the largest possible pixel index are identical, the image is displayed in its maximum resolution.
7. **View image layer metadata:** Select the layer name in the `Map Layers` section (Fig. 1-3). Open the context menu (right click on layer name) and select `Show Image Information`

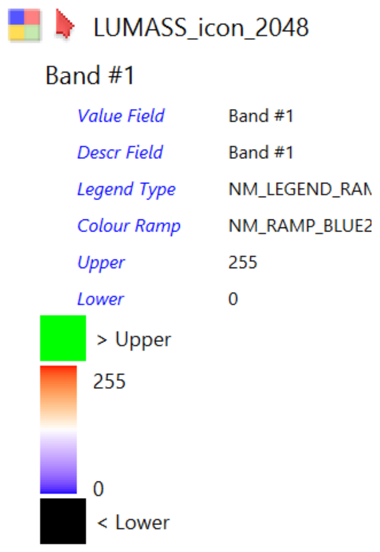


LUMASS_icon_2048

Size: 2048 x 1842
Pixel Size: 1 x 1
Pixel Component Type: uchar
Top Left: 0.00, 0.00
Bottom Right: 2048.00, -1842.00
Total Number of Bands: 4
Bands Displayed: 1, 2, 3
Current Overview: 3
Available Overviews: 1024x921 512x460 256x230 128x115 64x57
File Name: C:/Users/alex/garage/LUMASS/doc_workshop/ModellingHowTo/data/
LUMASS_icon_2048.kea

Figure 4 Image information

8. **Display the layer legend:** Double click on the layer name to display the legend. In case of a multi-band image, LUMASS maps three of the available bands to represent the colours red, green, and blue respectively.
9. **Change the band assignment:** Double click on any of the RGB colour items in the legend and assign a different band to the given colour.
10. **Map the value range of an individual band:** Open the context menu of the layer (right click) and select `Map Band Value Range`. This maps the entire value range of the selected band according to a given colour ramp. It is useful for multi-band images that don't represent colour values but, for example, a time series of an environmental variable, such as temperature.
11. **Adjust the value range mapping:** Double click on the band name (`Band #1`) above the displayed colour ramp to display the legend administration settings.



LUMASS_icon_2048

Band #1

Value Field	Band #1
Descr Field	Band #1
Legend Type	NM_LEGEND_RAN
Colour Ramp	NM_RAMP_BLUE2
Upper	255
Lower	0

Figure 5 Layer legend for a value range map; items displayed in blue font define the layer legend administration settings

e. **Change the mapped band:** Double click on the `Value Field` entry and select a different band or select `RGB` to return to the RGB mapping mode.

f. **Change the value range:** Double click on the `Upper` and `Lower` entries respectively to adjust the mapped value range to 100 to 150. Image Values below 100 are now rendered in black (i.e. the `< Lower` colour) and image values greater than 150 are now rendered in green (i.e. the `> Upper` colour). The image values in the range from 100 to 150 are linearly mapped against the selected colour ramp.

g. **Change the colour ramp and upper and lower colour**: Double click on the `Colour Ramp` entry in the legend administration settings and select a different colour ramp. To change the colours used to display values outside the specified range of 100 to 150, double click the `> Upper` or `< Lower` colour items respectively and select a new colour from the colour dialog. Note that you can also change the alpha channel value for those colours. For example, setting the alpha channel value to `0`, renders any selected colour transparent.

How to map raster layer attributes

0. In this example we focus on the mapping of a raster layer with an associated attribute table.

1. **Navigate to the LUMASS `SampleData/data` folder:** Use your favourite filesystem browser to do this (i.e. Windows File Explorer, Dolphin, Nautilus, etc.).

2. **Load the layer:** Select the `logo_rat.img` file and drag it into the `Map Layers` or `Map View` area respectively.

   Note that this layer was created from the `LUMASS_icon_2048.kea` layer using the `CreateLogoWithRAT.lmx` LUMASS model in the LUMASS `SampleData/models` folder. It extracts the first band (model component `ExtractBand`) from the image and then uses the model component `SumZones` to create an attribute table for the image. `SumZones` treats each set of pixel that share the same (integer) pixel value as an individual zone and creates and attribute table entry (i.e. table record) for it.

3. **Display the layer legend:** Double click on the layer name to unfold the legend.

   Note: Categorical raster layers are displayed by default with a `Unique Value` legend and the mapped attribute defaults to the first integer value attribute (here: `zone_id`). The colours are randomly assigned to each category.

4. **Change the colour of individual (`zone_id`) categories:** Double click on the category `'0'` and select an alpha channel value of `0` and click `OK`. This renders the logo background transparent.

5. **Save a unique value legend:** To save your individually configured unique value legend, right click on the layer name and select `Save Legend …`

   Note: The legend is saved as simple comma separated text file (*.csv) and assigns red, green, blue, and alpha values to each individual category value defined for the given attribute. However, the top row of the legend table assigns a particular colour value (the default is white) for image values which are not explicitly defined in the legend table (i.e. nodata colour). The legend can be loaded to colour arbitrary attributes (Layer Context Menu | `Load Legend …`) as long as they are displayed as unique value map. Also note that once you have loaded a particular legend file (i.e. unique value colour legend), this legend gets reapplied as long as you only change the attribute to be mapped (s. below) and not the legend type (e.g. to value range mapping).

6. **Map a different (integer) attribute:** You can change the current attribute being mapped by either i) using the layer context menu option `Map Unique Values …` or by ii) double clicking the Value Field entry in the layer legend administration settings (Fig. 5). Use option i) to change the mapped attribute to `rowidx`.

7. **Map the value range of a layer attribute:** To change the legend type, right click the layer name to open its context menu and select `Map Value Range` … This maps the currently selected layer attribute using a colour ramp across the attribute's value range from minimum to maximum.
   a. Display the layer legend administration settings.
   b. Double click on the `Value Field` item and select the attribute `count` from the drop down list

   Note: When you move the mouse pointer over the map (don't forget to select the layer), the pixel values now displayed in the status bar represent the mapped attribute values of the currently configured `Value Field` attribute (here: the number pixel in each category).

8. **Display all attribute values for a given point (pixel) in the map:**
   a. Select the layer name in the Map Layers display area (Fig. 1-3)
   b. Move the mouse pointer over the map and click with the left mouse button in the displayed map.

   This opens the `Layer Attributes` display area (Fig. 1-6). It shows a list of all layer attributes and their values for the given point.

9. **Map the value range of the actual pixel values:** Double click on the `Value Field` item and select `Pixel Values` from the drop down list.

   Note: Categorical maps use the actual pixel values stored in the image file to reference particular records in the associated attribute table. In the previous steps (2 to 7) this relationship is used to present a spatial map of the values stored in the attribute table. The `Pixel Values` option enables the mapping of the actual pixel values stored in the image. When you change to pixel values, it might be necessary to re-adjust the value range (i.e. `Upper` and `Lower` values). If you don't know the appropriate value range, you can use the layer's context menu options `Visible Pixel Statistics` or `Whole Image Statistics` to find out the minimum and maximum of the actual pixel values.

10. **Display a summary statistic of a layer attribute:** Map the value range of a layer attribute (s. step 7). Now, right click the layer name to open the context menu and select `Value Field Statistics`.

    Note: The value field statistic refers to the currently configured `Value Field` attribute and is derived from the values stored in the attribute table for this attribute.

Tables

LUMASS supports two types of tables: i) attribute tables for vector and raster layers and ii) stand-alone tables.

Attribute Tables

Raster and vector attribute tables are loaded automatically whenever a raster or vector layer is displayed in the Map View (Fig. 1-1). To view an attribute table, select `Open Attribute Table` from the layer's context menu. Click on the column headers (left mouse button) to sort the table or open the table's context menu using the right mouse button. Depending on whether you are viewing a raster or vector attribute table, LUMASS provides different capabilities especially with regard to querying and processing the table data. Since raster attribute tables are stored in a SQLite database, they provide a richer set of querying and processing options than vector attribute tables.

Stand-Alone Tables

LUMASS also supports the viewing and processing of stand-alone table data. To import tabular data, drag either a i) valid SQLite database, ii) a comma separated text file (*.csv), or iii) an Excel spread sheet (Excel 97 - 2003, *.xls) into the `Map Layers`, `Table Objects`, or `Map View` display areas (Figs. 1-3, 1-4, 1-1) respectively.

Spatial System Dynamics Modelling Framework

Model Structure

LUMASS' spatial modelling framework is built around two core components: i) data and ii) processes. The process components work on their input data to produce output data. Each process component represents a self-sufficient basic (spatial) algorithm that only depends on appropriate input data and a set of parameters. For example, the process component labelled `extract majorcat` (Fig. 6, bottom left corner), extracts higher order catchment identifiers from a given attribute of the input layer's attribute table. The new image data created by this process (i.e. its output), is passed on as input data to the next processing component `createRAT`. This component summarises the aggregated catchment data it receives and creates an attribute table containing a record for each higher order catchment. The received input data together with the newly created attribute table constitutes this component's output data. It is passed on to the `ImageWriter` component, which stores the image together with its associated attribute table as image file on disk. Such a sequence of process components, concatenated by their respective output and input data, is referred to as a processing pipeline.

Stand-alone process components and processing pipelines can be combined to aggregate components. This might be simply done to contain several pipelines contributing to the same higher level process, or to enable the repetitive execution of its child components. For example, the aggregate component `MarkCat` (Fig. 6, top right) is executed three times in succession, which is indicated by the number `3` next to the circular arrow symbol in its title bar. It means that the processing pipeline hosted by the component is executed three times in a row. In each iteration, the `ImageReader` component (cat) reads the same sub-catchment image file including its associated raster attribute table and passes it on to the `SQLProcessor` component (`mark major catchments`). This identifies, for a different catchment in each iteration, all of the catchment's upstream catchments and writes a new higher order catchment identifier for all identified catchments into the raster attribute table.

Aggregate components may be nested inside each other to construct complex hierarchical processing workflows. Together with the capability to repetitively execute components (Looping and Branching), it enables the development of models operating on multiple temporal scales.

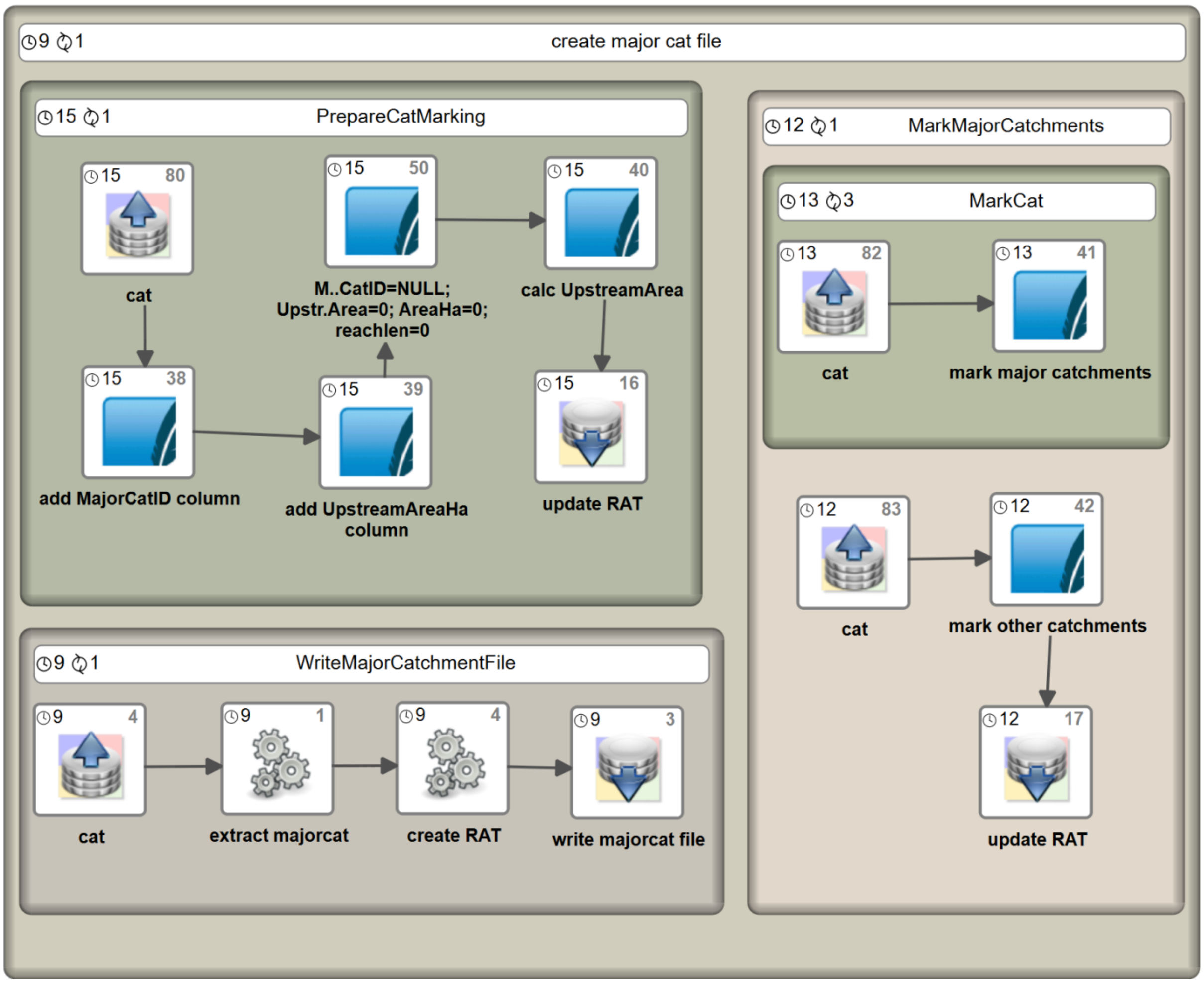

9 1 create major cat file
15 1 PrepareCatMarking
15 80 cat
15 50 M..CatID=NULL; Upstr.Area=0; AreaHa=0; reachlen=0
15 40 calc UpstreamArea
15 38 add MajorCatID column
15 39 add UpstreamAreaHa column
15 16 update RAT
12 1 MarkMajorCatchments
13 3 MarkCat
13 82 cat
13 41 mark major catchments
12 83 cat
12 42 mark other catchments
12 17 update RAT
9 1 WriteMajorCatchmentFile
9 4 cat
9 1 extract majorcat
9 4 create RAT
9 3 write majorcat file

Figure 6 Hierarchical model structure and execution order

Properties and Parameters

In the following sections we will often refer to *parameters* and *properties*. To avoid any confusion, we here define their meaning in the context of the LUMASS modelling framework.

Parameter

In a LUMASS model, a parameter denotes a constant value over a single execution of a process component. It may be a text (string) or numeric value and may reference specific data or a specific mode of computation. For example, a parameter could specify a particular characteristic of an image, such as its number of bands, a particular table, column, or row in a SQL expression, a constant numeric value in a mathematical equation, or a particular mode of computation, e.g. `STRIPPED` versus `TILED` streaming (cf. `ImageWriter`).

Property

A property refers to a particular characteristic of a model component. It ties a model parameter value to a specific model component and represents the technical means by which a model parameter is supplied to an aggregate or process component. Table 1 provides a list of properties shared across model and process components respectively. In addition to these general properties, individual process components are characterised by further properties depending on their specific functionality. Please refer to the Model Component Reference section for a comprehensive overview.

Table 1 General model and process component properties

	Property	Characteristic	Meaning
(Aggregate) Model Component	ComponentName	compulsory, system-defined	Unique model component name; LUMASS ensures that each model component can be uniquely identified by its `ComponentName`
	UserID	optional, user-defined	A non-unique user-defined short name, which is used to refer to a specific model component in map algebra expressions, map kernel scripts, SQL statements, and parameter expressions.
	Description	optional, User-defined	A short user-defined description of the component; it defaults to the `ComponentName`
	TimeLevel	compulsory, user-defined	The user editable time level of the model component; time levels are used to define the control flow of a model
	Inputs	optional, user-defined	A list of the `ComponentNames` of the input components;
	IterationStep	compulsory, user-defined	The start iteration step for the next execution of the component, or the actual iteration step if the component is currently being executed
	NumIterations	compulsory, user-defined	The number of times the component is executed
	NumIterationsExpression	optional, user-defined	A list of parameter expressions to dynamically define the number of times a component is executed (e.g. used for conditional iteration)
Process Component	ProcessName	compulsory system-defined	The non-editable class name of the process object embedded in this model component
	InputNumDimensions	compulsory, user-defined	The number of dimensions of the input component
	NMInputComponentType	compulsory, user-defined	The data type of the input component
	NMOutputComponentType	compulsory, user-defined	The data type of the output component
	InputNumBands	compulsory, user-defined	The number of bands of the input component
	OutputNumBands	compulsory, user-defined	The number of bands of the output component

Control Flow

Execution Sequence

The model component `create major cat file` (Fig. 6) extracts three aggregated higher order catchments from a provided sub-catchment file and writes them together with an associated attribute table into a new image file. The overall functionality is broken down into smaller processing steps, which, executed in the right order, provide the desired result. To control the execution sequence of process and aggregate components, LUMASS uses time levels that are assigned to each individual model component. The time level is shown next to a clock symbol in the top left corner of each component. Execution flows from higher time levels to lower time levels and from higher aggregation levels to lower aggregation levels, i.e. from the outside to the inside. For example, the execution sequence of the `create major cat file` (Fig. 6) component is as follows (with time levels labelled with TL):

TL 9: `create major cat file`
- **TL 15:** `PrepareCatMarking`
  - **TL 15**: Pipeline to prepare the raster attribute table (RAT) (e.g. add columns, etc.)
- **TL 12**: `MarkMajorCatchments`
  - **TL 13**: `MarkCat` *(note: executed 3 times in a row)*
    - **TL 13**: Pipeline to `mark major catchments`
    - **TL 12**: Pipeline to `mark other catchments`
- **TL 9:** `WriteMajorCatchmentFile`
  - **TL 9:** Pipeline `to extract majorcat` and `create RAT`

Process components that are part of a processing pipeline and that share the same host component (i.e. aggregate component), have to sit on the same time level to be properly initialised prior to pipeline execution. Note that, counter intuitively to the overall down-stream execution flow, pipeline execution always starts at the bottom end of each pipeline (pull model) (Johnson et al. 2016). That means the position of a pipeline's bottom end component in the model hierarchy determines when the pipeline is executed. Consequently, the number of iterations of a pipeline-end's host component determines the number of times the whole pipeline is repeated in sequence. This implies that a processing pipeline may reach across different aggregate components as long as it follows the general rule of down-stream execution (and thus data) flow. In other words, individual processing components that are linked into a processing pipeline, may only provide input to other process components that sit either on the same time level and share the same host component or that are positioned down-stream with regard to the overall model hierarchy. Components that are not part of a processing pipeline but share the same host component and time level, may be executed in an arbitrary order.

Time levels are user-defined (Table 1) and do not necessarily have to be strictly sequential, i.e. the sequence may omit individual numbers. For example, the internal execution sequence inside the `create major cat file component` starts at time level `15`, which is followed by time level `13`. The only rule LUMASS enforces is that the minimum time level of child components inside an aggregate component must not be smaller than the time level of the component itself (i.e. their host or parent component). For example, when the hierarchy of a model is changed by moving or cloning components from one component to another, if required, LUMASS automatically adjusts the time levels of the inserted components according to this rule. Additionally, LUMASS provides efficient means to change and adjust time levels for multiple components at a time.

Looping and Branching

Conditional execution in a LUMASS model is possible at two levels: i) the pixel or record level, and ii) the component level. Conditional statements at the pixel level are provided by the `MapAlgebra` and the `MapKernelScript2` components and at the record level by the `SQLProcessor` component. While these enable the conditional computation of individual pixel or table record values, conditional execution at the component level enables runtime control over the execution of aggregated components representing higher level processes. This is implemented by way of the `NumIterationsExpressions` property (Table 1) that enables the dynamic definition of the number of times an aggregated component is executed, which, if the value is zero, is not executed at all. However, if no `NumIterationsExpression` is provided, the number of iterations of an aggregate component is defined statically using its `NumIterations` property (Table 1).

Dynamic Model Parameters

Parameter Lists

To support the representation of dynamic systems as well as enabling the assessment of model sensitivities and uncertainties, LUMASS provides the concept of dynamic parameters by way of i) static parameter lists for individual component properties and ii) parameter expressions. A parameter list provides a set of values (Fig. 7, `1..n`) to a particular component property (Table 1). During an iterative execution (`1..m`) of the component's host component (Fig. 7, `Aggregate Component`), these parameter values are passed on to the process component's property one after another, as long as there are more values on the list. In case the iteration continues beyond the last available parameter value (`m > n`), the last value is re-used for the remainder of the iterations. By default, an iteration starts with `IterationStep = 1`. However, it may be re-configured by the user, e.g. to debug or test a particular modelling step or to statically disable a particular component (`IterationStep > NumIterations`) (in the latter case, the aggregate component is rendered transparent to indicate that the component is disabled).

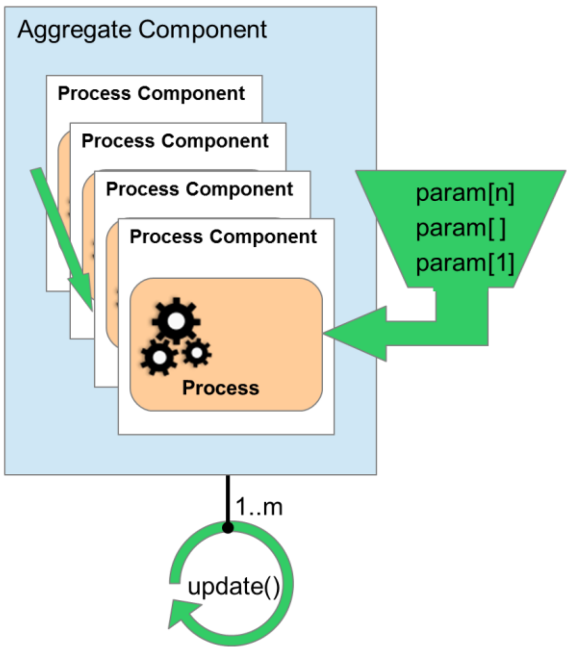


Figure 7 Dynamic processing pipeline

Parameter Expressions

Notation

Parameter expressions extend LUMASS' capabilities to dynamically set model parameters at runtime. They enable

a. access of numeric process and aggregate component property values,
b. retrieval of table values (text and numeric), and the
c. evaluation of mathematical expressions to calculate parameter values.

Parameter expressions to access component property values or to retrieve table values, take the following general form:

```
$[<component>:<property | column>:<index>]$
```

with:

`<component>`: Unique `ComponentName` or `UserID` of the component providing the parameter. In case the given parameter expression references a table value, the component referred to may be an `ImageReader`, a `TableReader`, or a `DataBuffer` or `DataBufferReference`.

Note: Since `UserID`s are non-unique identifiers, here it references the first strictly upstream component matching the given `UserID`. That is, starting at the hierarchy level of the component, whose property is going to be set by the given expression, LUMASS looks for a component matching the specified `UserID`. If no component is found, the search continues at the next higher level, i.e. the component's host component's level, until a matching component is found. Thereby, the search direction is strictly upward, that is child components of any aggregate component on the search path are excluded.

`<property | column>`: Property or table column name

`<index>`: In case the given property name references a list of values (e.g. table column values), `index` refers to the 1-based index of the referenced value.

Note: If the expression references a table column value, the index value actually refers to the table's primary key. If the primary key is 0-based, LUMASS adjusts the user-specified 1-based index automatically to deliver the appropriate result. If the given index cannot be found in the table, model execution is aborted and LUMASS reports an error in the `Notifications` window.

In addition to the full qualified form of a parameter expression (s. above), LUMASS supports a special short-hand notation to refer to the current `IterationStep` of an aggregate component:

```
$[<component> <+ | -> <integer number>]$
```

It comprises the component specification and may optionally be followed by a simple arithmetic expression to add or subtract a whole number to or from the `IterationStep` respectively. Furthermore, LUMASS supports a general more powerful notation to evaluate sophisticated mathematical expressions. It is initiated by the character sequence '`math:`' and is followed by a mathematical general expression:

```
$[math: <mathematical expression>]$
```

with:

`<mathematical expression>`: A mathematical expression as understood by the mathematical function parser muparser.

LUMASS parameter expressions may be nested and may occur anywhere inside a `QString` type (s. Editing Model Parameters) component property value specification, e.g. inside mathematical expressions (`MapAlgebra`), map kernel scripts (`MapKernelScript2`), or SQL expressions (`SQLProcessor`).

Examples

In this section we illustrate and explain the use of sample parameter expressions. The examples are taken from the LUMASS implementation of the DaisyWorld model (Watson and Lovelock, 1983; Neuwirth et al., 2015). To inspect the model drag the LUMASS model file

```
SampleData/models/DaisyWorld/DaisyWorld.lmx
```

into the `Model View` (Fig. 1-2). To locate the individual components, just enter the `ComponentName` or `UserID` into the search bar in the tool bar (Fig. 2-15). Keep an eye on the `Notifications` window (`View | Notifications`) when you search by `UserID`.

Example 1 (ComponentName: `ImageReader1`, UserID: `landscape`, Property: `FileNames`)

```
$[DaisyParams:Path:1]$/$[DaisyParams:LayerName:1]$_0.img
```

In this example two parameter expressions are used to specify the `FileNames` property value of the `ImageReader1` component. Each expression refers to the model component `DaisyParams`, a `DataBuffer` component with the `UserID` 'DaisyParams' that holds an in-memory connection to a SQLite database table. `Path` and `LayerName` represent columns in this table and the index `1` refers to the respective column values stored in the first row of the table. Note that the parameter expressions are tightly integrated into the `FileNames` specification and that each expression is replaced by its actual value before the property value is supplied as a model parameter to the `ImageReader1` process component. In this case, after both parameter expressions have been processed, the property value specification evaluates to:

`C:/Temp/DaisyWorld/landscape_0.img`

Example 2 (ComponentName: `ImageWriter2`, UserID: `N/A`, Property: `FileNames`)

`$[DaisyParams:Path:1]$/$[DaisyParams:LayerName:3]$_$[LifeCycle]$.img`

This parameter expression demonstrates the short-hand notation to refer to the `IterationStep` property of the aggregate component `LifeCycle`. Whereas the first and second parameter expression in the `FileNames` value specification refer to static values in the `DaisyParams` table, the `$[LifeCycle]$` expression refers to the dynamically changing `IterationStep` property value during the repetitive execution of the `LifeCycle` aggregate component. For example, given an `IterationStep` value of `1` and a `NumIterations` value of `100` before the execution of the `LifeCycle` component, the above `FileNames` specification evaluates to

`C:/Temp/DaisyWorld/Age_10.img`

in the 10th iteration of the component (i.e. `IterationStep` = 10).

Example 3 (ComponentName: `AggrComp2`, UserID: `ShrinkWhite`, Property: `NumIterationsExpression`)

```
$[math: rint(
$[DaisyWorldBuffer:$[DaisyParams:ColumnName:8]$:$[LifeCycle]$]$
) < 0 ? 1 : 0]$
```

This example nests regular parameter expressions with the short-hand `IterationStep` notation and a mathematical parameter expression. The outermost expression

`$[math: rint(<expr2>) < 0 ? 1 : 0]$`

rounds a floating point value `<expr2>` to the nearest integer value and returns `1`, if the result is negative and `0` otherwise. `<expr2>` evaluates to a floating point number, which is looked up from the `DaisyWorldBuffer` table. This table contains model state and control data, which are calculated for each iteration of the `LifeCycle` aggregate component. <expr2> specifies where the required data for the particular iteration step is located in the table:

`$[DaisyWorldBuffer:<expr3>:<expr4>]$`

The column name and the row index in `<expr2>` are in turn represented by parameter expressions `<expr3>` and `<expr4>`, which represent a regular parameter expression and an `IterationStep` expression respectively:

```
$[DaisyParams:ColumnName:8]$
$[LifeCycle]$
```

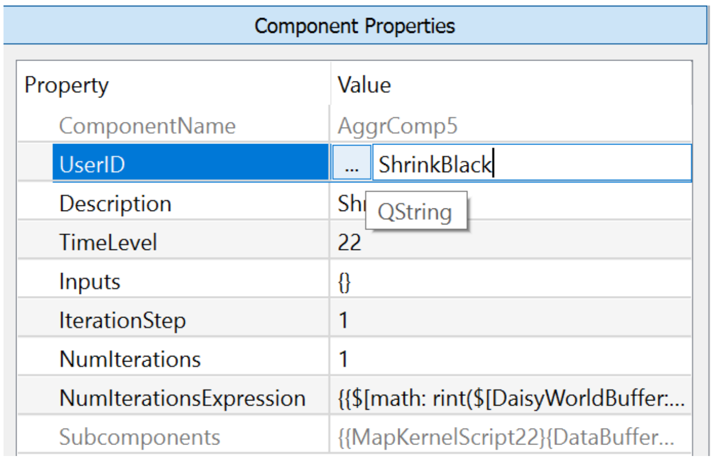
This suggests that for each iteration the data is read from the same column but a different row and that the latter depends on the `IterationStep` of the `LifeCycle` component.
LUMASS successively evaluates nested parameter expressions from the inside to the outside to generate the actual parameter value to be supplied to the model component. The dynamic expression in this example evaluates to either `1` or `0` and specifies the `NumIterationsExpression` property value of the aggregate component `ShrinkWhite`, a child component of the `LifeCycle` aggregate component. Hence, it controls whether `ShrinkWhite` is being executed as part of the actual iteration step of the `LifeCycle` component or not.

Editing Model Parameters

Property values are specified using the `Component Properties` window (Fig. 1-7). Moving the mouse pointer over the `Property Value` section of the window indicates the property type:

- `bool` (True or False)
- `int` (whole number; enumeration)
- `PixelType` (enumeration)
- `QString` (text)
- `QStringList` (list of texts)
- `QList<QStringList>` (list of lists of texts)
- `QList<QList<QStringList> >` (list of lists of lists of texts)

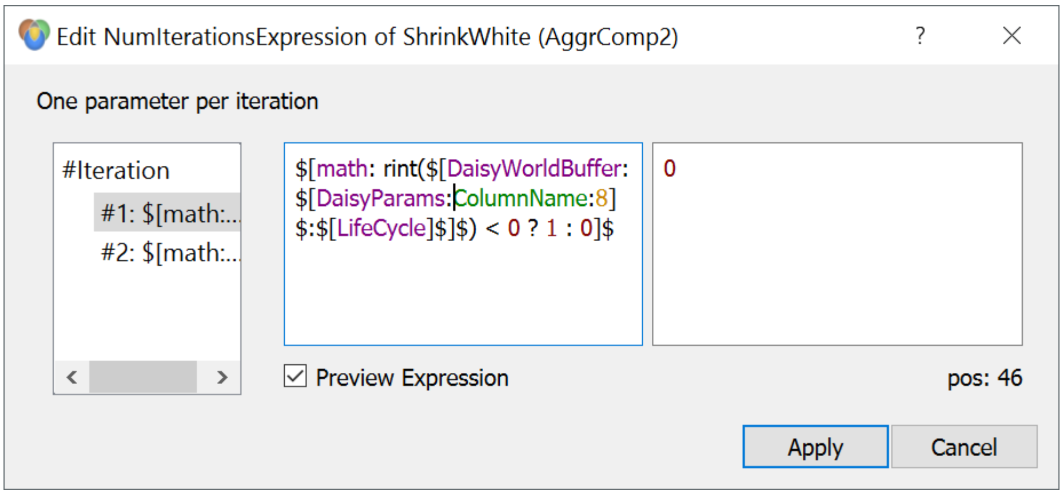
The `bool`, `int`, `PixelType`, and `QString` type properties can be edited by clicking into the `Value` section of the `Component Properties` window to either change or select the appropriate value for the given property respectively. The latter one can also be edited using the parameter editor (Fig. 9, s. below) by clicking the … button at the left hand side of the inline editor for `QString` type properties (Fig. 8).


Component Properties

Property	Value
ComponentName	AggrComp5
UserID	... ShrinkBlack
Description	Shi QString
TimeLevel	22
Inputs	{}
IterationStep	1
NumIterations	1
NumIterationsExpression	{{$[math: rint($[DaisyWorldBuffer:...
Subcomponents	{{MapKernelScript22}{DataBuffer...

Figure 8 `Component Properties` window: Inline editor for `QString` type properties

`bool`, `int`, `PixelType`, and `QString` type properties represent static single values, which cannot change during an iteration sequence. However, properties of type `QString` may also be specified using parameter expressions, which can be used to effectively circumvent this restriction. Properties of type `QStringList`, `QList<QStringList>`, and `QList<QList<QStringList> >` represent parameter lists representing parameter values of increasing dimensionality. Whereas a `QStringList` comprises a single `QString` type property value per iteration step, `QList<QStringList>` represents a list of property values supplied to the model component each iteration step. Consequently `QList<QList<QStringList> >` represents a table of values, which is supplied each iteration step to the model component. It is important to point out that the actual parameter value, though encoded as `QString` type, may actually represent a numeric value or in fact a string of characters (i.e. text). `QString` is only used as a carrier, able to encode all different types of parameter values as well as representing complex nested parameter expressions, which can be evaluated to an actual parameter value of any type. Parameter lists are edited using the parameter editor (Fig. 9), which opens automatically when clicking into the appropriate `Value` section of a parameter list property.


Edit NumIterationsExpression of ShrinkWhite (AggrComp2)
One parameter per iteration
#Iteration
#1: $[math:...
#2: $[math:...
Preview Expression
pos: 46
Apply
Cancel

```
$[math: rint($[DaisyWorldBuffer:
$[DaisyParams:ColumnName:8]
$:$[LifeCycle]$]$) < 0 ? 1 : 0]$
```

```
0
```

Figure 9 Parameter Editor

It facilitates the specification of parameter lists by providing

- context menu and drag & drop driven editing of lists and parameters (Fig. 9, left)
- syntax highlighting and auto-completion for editing parameter expressions (Fig. 9, middle)
- a preview evaluation of parameter expressions for the given model state (Fig. 9, right)

Note that the auto-completion and preview evaluation functionality of the parameter editor depends on the referenced model components (and their underlying data) being available and properly initiated.

Processing Pipeline

Apart from concatenating data objects and processing objects, processing pipelines offers two main benefits:

Sequential processing of large images: The image(s) are split into smaller image regions, which are then processed one by one and reassembled at the end of the pipeline (e.g. an `ImageWriter` component).

Parallel processing: The parallel processing of individual image regions (s. sequential processing) in individual threads. Whether a process component provides multi-threading or not depends on the implemented algorithm. Most LUMASS process components support at least partial parallel processing.

The above processing pipeline benefits are provided by the core libraries underpinning the processing capabilities of the LUMASS modelling framework, i.e. Insight Toolkit (ITK) and Orefo Toolbox (OTB). Please consult the ITK Software Guide (Johnson et al., 2016) for further information.

Model Component Reference

Table 2 provides an overview of the currently available model components within LUMASS (version 0.9.52).

Table 2 Available model components

Component[1]	Function	Inputs[2]	Outputs[2]
CastImage[c,e,f]	Pixel data type conversion	0: Image of type T1	0: Image of type T2
CostDistanceBuffer[b]	Object buffer or cost-distance surface around/to defined objects (i.e. pixel values)	Feature image (file name); cost image (file name)	Buffer or cost distance image (file name)
DataBuffer[a,f]	In-memory image and/or attribute table	0: Process component output	0: Image and/or table
DataBufferReference[a,f]	Reference to DataBuffer	N/A; references DataBuffer by `UserID` or `ComponentName`	0: Image and/or table
ExternalExec[b]	Execution of an external non-interactive programme or script	N/A	N/A
ExtractBand[c,f]	Extract an image band from a multi-band image	0: Multi-band image	0: Single band image
FocalDistanceWeight[c,f]	Focal CA-based weighting algorithm for calculating land-use transition potentials	0: Land-use image	0: Neighbourhood-based influence factor for calculating land-use transition potential
ImageReader[c,f]	Reads an image file from disk or a rasdaman database (3D) including its associated RAT	Image (absolute file name or rasdaman image specification)	0: Image and associated RAT, if available
ImageSorter[b]	Sorts images based on the first input image; also produces an index image containing the 1D unsorted input pixel indices	List of input images (file names)	Sorted images written to disk (indicated by _desc or _asc); index image (_idx) containing unsorted 1D input pixel indices
ImageWriter[c,f]	Writes an image to disk or into a rasdaman database (3D)	0: input image and RAT, if applicable	N/A
MapAlgebra[c,f]	Pixelwise mathematical operation on a set of input images and attribute tables	0..(n-1): single band image of type T, where n equals the number of input images	0..(m-1): single band image of type T, where m equals the number of expressions supplied
MapKernelScript2[c,f]	Script-based pixel processing incl. neighbourhood access (CA-based modelling)	0..(n-1): single band image of type T, where n equals the number of input images	0: single band output image 1: AuxData Table
ParameterTable[a]	A visual and editable stand-alone table in the model view	N/A	N/A

Component[1]	Function	Inputs[2]	Outputs[2]
RandomImage[c,f]	Creation of a range constrained pseudo random image from scratch	N/A	0: single band output image
ResampleImage[c,f]	Resamples an image using different interpolation methods	0: single band input image	0: resampled single band output image
SpatialOptimisation[d,f]	Solves a spatial optimisation problem on input's RAT	0: input image with associated RAT	0: output image with associated RAT
SQLProcessor[c,f]	Performs SQL processing on input tables	0: input image (with associated RAT) or stand-alone table, image is passed through as single output 1..(n-1): input image or stand-alone table	0: input image at index 0
SummarizeZones[c,f]	Calculates a zonal statistic of image 1 within the zones of image 0	0: integer zone image 1: value image	0: integer zone image with RAT summary statistic of input 1 (or, if not available, input 0)
TableReader[d]	Reads a stand-alone *.csv, *.xls or SQLite table from disk; note: if *.csv or *.xls files are supplied it creates an associated SQLite database (*.ldb) and table and opens a connection to the table	Input table (file name)	0: stand-alone table (suitable for SQL processor)
TextLabel[a]	Rich text label component for comments model description	N/A	N/A
UniqueCombination[c,f]	Identifies unique combinations of input pixel (raster union)	0..(n-1): categorical input image	0: unique combination index of inputs with associated RAT showing original pixel value per input layer

[1] a: model component; b: stand-alone process component, cannot be incorporated in pipeline; c: streamable process component; d: non-streamable process component; e: multi-band support; f: 3D support (experimental)
[2] Inputs and outputs are prepended by their 0-based index position they have to be provided at (input) or can be accessed at (output) (e.g. `MapAlgebra:0` refers to the first output and `MapAlgebra:1` to the second output of the `MapAlgebra` component respectively)
RAT: raster attribute table

Please note that the description of component properties in the following sections only refers to the actual type of the parameter value that is supplied to the component during an individual iteration step.

ExternalExec

Overview

`ExternalExec` enables the execution of any non-interactive programme or script installed on the computer running LUMASS or the LUMASS engine. LUMASS executes the specified `Command` and continues its internal processing workflow after the execution of the external programme or script has finished.

Properties

`Command` (`QStringList`): Operating system specific command (text) including any command-line arguments to execute the external programme or script.

`Arguments` (`QStringList`): Do not use.

FocalDistanceWeight

Overview

This filter implements a focal weighting algorithm as described by White et al. (1997, p. 326). The central pixel of a circular neighbourhood of radius `r` is weighted according to the occurrence of certain pixel values within defined distance classes.

Distance Classes

The number of distance classes `n` is defined by the radius of the neighbourhood and can be calculated as follows (note: zero distance is not included):

```
n = ((r*r) / 2.0) + 0.5   (odd radii)
n = ((r*r) / 2.0) + 1.5   (even radii)
```

Weights Matrix

The weights for certain pixel values for each of the distance classes is provided by a `m x n` matrix (`Weights`), where `n` (i.e. number of columns) equals the number of distance classes and `m` (i.e. number of rows) equals the number of pixel values to take into account in the weighting procedure. The `Weights` matrix needs to be ordered with classes representing increasing distance from left to right. The indices of the rows for individual pixel values need to be provided as an accordingly ordered array of values (i.e. `Weights[i][0]` represents the weight of pixel value `Values`[i] for the smallest distance class).

Properties

`RadiusList` (`QStringList`): Radius (integer) of the focal neighbourhood in pixel.

`Weights` (`QList<QList<QStringList> >`): Matrix (table) of weights (float).

`Values` (`QList<QStringList>`): List of pixel values of `NMInputComponentType` type to be accounted for in the weighting algorithm.

ImageReader

Overview

The `ImageReader` component reads images from files stored on a hard drive or from a rasdaman database.

Reading Image Files from Disk

Image files may be stored in any image format that uses regular file names (e.g. `/home/images/myimage.img`) and that is supported by the underlying GDAL library. Sub-datasets, for example stored in a NETCDF file, cannot be directly read by LUMASS. However, they may be extracted beforehand, for example, by using the `ExternalExec` component in conjunction with the gdal_translate command-line application.

Reading Images from a Rasdaman Database

To read an image from a rasdaman database, the `RasConnector` property needs to be set to `True`. Furthermore, the user needs to have access to a rasdaman database instance and the rasgeo client application for rasdaman needs to be setup. Rasdaman images are specified by their collection name and, if applicable, by their unique object identifier (OID). Collection name and OID are separated by a colon ':', e.g. `MyRasdamanCollection:3053`. If the OID is omitted and the given collection comprises more than one image, LUMASS interprets the images in the given collection as a multi-band image.

Reading Raster Attribute Tables (RAT)

If the given image has an associated raster attribute table (RAT), LUMASS automatically reads it together with the image. `ImageReader` provides the RAT in one of two different modes, i.e. a simple in-memory table (`ATTABLE_TYPE_RAM`), or as a SQLite database table (`ATTABLE_TYPE_SQLITE`) (s. property `RATType`). In the latter case, the table name inside the SQLite database is based on the image file's base name. For example the RAT name inside a SQLite database for the image `C:/temp/myimage.img` would be `myimage_1`, where the '`_1`' represents the band number of the image. However, LUMASS currently only supports fetching the RAT of the first band. The different types of tables provide different benefits for different purposes. For example, a SQLite database table is required by the `SQLProcessor` component to perform tree queries or create or join tables. Additionally, SQLite database tables don't have to be kept in main memory. In-memory tables (`ATTABLE_TYPE_RAM`), on the other hand, cannot be used for SQL-based processing, but provide fast access to tabular data, e.g. to the `MapAlgebra` or the `MapKernelScript2` component respectively.

Properties

`FileNames` (`QStringList`): Input image file name (i.e. absolute file path or rasdaman image specification).

`RasConnector (bool)`: If set to `True`, the component expects to read from a rasdaman database and `FileNames` to provide a rasdaman image specification rather than a disk-based file name.

`RATType` (`enumeration`): Defines whether the image's RAT, if applicable, is read into an in-memory RAT (`ATTABLE_TYPE_RAM`) or a SQLite database table (`ATTABLE_TYPE_SQLITE`).

`RGBMode` (`bool`): If set to `True`, the image's internal pixel type is interpreted as 'RGB' and each colour component is of type `NMOutputComponentType`. Note that this is not required to read multi-band images in general, but only if another component requires the internal RGB image pixel type (cf. Johnson et al. 2016, RGB Images).

`BandList` (`QList<QStringList>`): A list of 1-based (integer) band numbers. The parameter may be left empty to read all bands of the image.

ImageWriter

Overview

The `ImageWriter` component writes images and an associated RAT to disk or into a rasdaman database. Support for building overview (pyramid) layers is currently only available for disk-based image files.

Writing an Image File to Disk

The `ImageWriter` uses the GDAL library to write images to the hard drive. Image file names have to be specified as absolute file path.

Writing an Image to a Rasdaman Database

To write an image into a rasdaman database, the `RasConnector` property needs to be set to `True`. Furthermore, the current user needs to have access to the rasdaman database instance and needs to have setup the rasgeo client application for rasdaman. To write a new image into a rasdaman database, the user has to specify the collection name, the image is going to be stored in. If the collection does not exist, it is created upon writing the image. If the collection already exists, LUMASS adds a new image to the collection. If the collection name is specified together with a valid OID, the specified image is going to be updated.

Properties

`FileNames` (`QStringList`): *Disk-based storage:* Absolute file path of the output image to be created (or overwritten) on disk. Note that the file name extension, e.g. '`.img`', defines the GDAL-supported output image type. *Database storage*: Rasdaman image specification.

`RasConnector (bool)`: If set to `True`, `ImageWriter` expects to write to a rasdaman database and `FileNames` to provide a rasdaman image specification rather than a disk-based file name.

`InputTables` (`QStringList`): The name of a component providing the RAT to be stored together with this image. Note that if this parameter is specified, the `WriteTable` parameter is ignored.

`WriteImage` (`bool`): Defines whether the actual image data is going to be written out. Setting this property to `False` avoids unnecessary write operations, e.g. if only the RAT of an image is being processed by the input component (e.g. `SQLProcessor`) and shall be written out.

`WriteTable` (`bool`): If the input image has an associated RAT, this property defines whether it is written out together with the image or not. Note that this property is being ignored, if the `InputTables` property is specified.

`UpdateMode` (`bool`): If set to `True`, the output image is opened in update mode, rather than being overwritten. This mode is required when an input processing component performs a global operation (i.e. all image data is required to be processed to provide a meaningful output) and hence its output data is only valid after all pixel of the image have been processed. For example, the zonal statistics table created by the `SumZones` component is only provided after the processing of the input image is completed. Note that not all image file formats support this mode. Double check the GDAL formats reference for information.

`StreamingMethodType` (`enumeration`): Defines how the input image is split for streamed (i.e. sequential) processing. (`TILED`: set of rectangular regions; `STRIPPED`: set of rows)

`StreamingSize` (`int`): The maximum approximate memory usage (MB) (integer) of the pipeline this writer is connected to.

`PyramidResamplingType` (`enumeration`): The resampling method used to build overview (pyramid) layers. The option `NONE` writes or updates an image without creating overviews.

`RGBMode` (`bool`): If set to `True`, the writer interprets the internal image's pixel type as 'RGB' and the type of each colour component as `NMInputComponentType`. Note that this is not required for writing multi-band images. However, it is required to write an input image with RGB pixel type (cf. Johnson et al. 2016, RGB Images).

MapAlgebra

Overview

The `MapAlgebra` component performs a pixelwise mathematical operation on a series of input images. All input images have to share the same physical space (i.e. origin, extent, and resolution) and pixel type and must be single-band.

Mathematical Expression

The mathematical operations supported by this component are defined by its underlying muparser library. A list of its built-in functions and operators can be found here. In addition to these built-in functions, LUMASS supports the following functions and constants:

```
rand(a,b): returns a uniform pseudo random integer number n in the range a <= n <= b
fmod(a, b): returns the remainder of a/b.
constants: e,log2e,log10e,ln2,ln10,pi,euler
```

Inside a mathematical expression, input images are referred to by the `UserID` of their associated model component (e.g. `ImageReader`, `ExtractBand`, etc.). Attribute data is referred to by concatenating the respective `UserID` and the table column name by a double underscore:

```
<UserID>__<column name>
```

For example, the extract `majorcat` `MapAlgebra` component in the `WriteMajorCatchmentFile` component (Fig. 6, bottom left), uses the following expression to extract the major catchment identifier from the input image (`UserID`: `cat`):

```
cat < 0 ? 0 : cat__MajorCatID
```

This expression sets all output pixel values to `0` if their respective input pixel value `cat` is smaller than `0` (`cat < 0`). If the condition `cat < 0` is false, it assigns the respective attribute table value in column `MajorCatID` of input image `cat` to the output pixel. Please note that a `MapAlgebra` expression does not support the muparser assignment operator. The given expression is evaluated for each individual set of input pixel and its result is automatically assigned to the respective output pixel. Note that expressions may also be nested and use brackets as shown in this example:

```
q < 1 ? (1 - 4 / (pi * sqrt(1 - q^2)) * atan(sqrt((1 - q) / (1
+ q)))) : q == 1 ? (1 - 2 / pi) : 1 - 2 / (pi * sqrt(q^2 - 1))
* ln((1 + sqrt((q-1)/(q+1))) / (1 - sqrt((q-1)/(q+1))))
```

Data Types and Type Casting

Every value used in a muparser expression as well as its return (i.e. output) value after evaluating an expression, are internally represented by the data type `double` (i.e. a 64 bit floating point number). Hence, any input data, e.g. tabular data, must be numeric. All non-numeric table columns of an input RAT are not made available for usage inside a muparser expression. However, all numeric non-`double`-typed input data are casted into a `double` type, which may involve data loss. Similarly, the calculated output pixel value is casted into the `NMInputComponentType` type configured for the `MapAlgebra` component. For output pixel values, `MapAlgebra` keeps track of overflows and underflows as a result of incompatible data types and issues a warning in the LUMASS `Notifications` window (`View | Notifications`).

Multiple Expressions

It is also possible to define more than one mathematical expression for each iteration step (`NumExpressions`). In that case individual expressions must be separated by a comma '`,`'. It means that more than one distinct output image can be produced by an instance of the given `MapAlgebra` component. Each output image can be accessed by its respective output index, defined as the expression number minus 1. For example, the result of expression 2 can be accessed at output index 1 and the result of expression 5 can be accessed at index 4 etc. Note that although all expressions are evaluated at the same time and each time the component is executed, only one image is allocated at a time and populated with the appropriate output data of the requested output image. Hence, the particular `MapAlgebra` component needs to be executed each time a different output image is requested by a downstream component.

Properties

`InputTables` (`QStringList`): Deprecated – do not use. Note that his property is no longer required, since LUMASS automatically fetches RATs associated with input images.

`InputTableVarNames (QList<QList<QStringList> >)`: Deprecated – do not use. Note that his property is no longer required, since LUMASS automatically detects the RAT attributes used in the given mathematical expressions.

`MapExpressions` (`QStringList`): Mathematical expression to be performed on the input images.

`NumExpressions` (`QStringList`): The number of comma separated expressions supplied per iteration.

`UseTableColumnCache` (`bool`): If set to `True`, input RAT attributes are cached prior to the execution of the filter. This option is useful to provide fast access to attribute values during the pixelwise computation. Not that this option should be set to `False` for in-memory RATs (`ATTABLE_TYPE_RAM`).

MapKernelScript2

Overview

This process component executes a small user-defined script to calculate individual output pixel values. Depending on the configured neighbourhood size (`Radius`), individual pixel values in a user-defined `CIRCULAR` or `RECTANGULAR` neighbourhood (`KernelShape`), may be accessed to calculate the output pixel value of the centre pixel. Hence, `MapKernelScript2` facilitates the implementation of arbitrary focal operators, for example to calculate slope or shaded relief maps, or to build cellular automata. The user-defined script represents a sequence of muparser-based expressions (cf. `MapAlgebra`), completed by a semi-colon '`;`'. Additionally, users may use C-style for loops for the repetitive evaluation of a sequence of mathematical expressions (General Kernel Script Syntax).

Neighbourhood (Kernel Window)

`MapKernelScript2` supports a different neighbourhood radius for each image dimension `d`. The kernel window length `len` in pixel for each dimension is (Johnson et al., 2016, Neighborhood Iterators)

```
len = 2 * radius(d) + 1
```

where `radius(d)` is the kernel radius in image dimension `d`. For example, a kernel radius of `1` for a 2D image yields a `3x3` neighbourhood. Individual pixel values in the neighbourhood can be accessed by their index position in the neighbourhood, which is numbered sequentially from left to right and top to bottom (for a 2D image) starting with index `0`:

0	1	2
3	4	5
6	7	8

Accessing Pixel Values

Input (image) pixel values are referenced by their respective image identifier, that is the `UserID` of the particular input component, such as an `ImageReader`'s `UserID`. If a kernel radius of greater than zero is specified for any of the image dimensions, the input pixel values are accessed using the `kwinVal` function

```
v = kwinVal(img, pixIdx, thid, addr);
```

where `v` represents the pixel value of input image `img` at neighbourhood index position `pixIdx`. The `thid` and `addr` parameters are pre-defined constants and have to be supplied for technical reason. For the convenience of the user, the index of the centre pixel is provided as pre-defined constant `centrePixIdx` and saves the user from having to calculate it specifically for different kernel sizes and shapes. Another convenience function is `neigDist`

```
        h = neigDist(pixIdx, addr);
```

where `h` represents the distance in pixel from the centre pixel to the pixel indicated by the neighbourhood index `pixIdx`. `addr` is a pre-defined constant and has to be supplied for technical reasons.

Accessing Table Values

Table values are referenced by their 0-based column and row index in the table. For example the value `t`, in column `colIdx` and row `rowIdx` of the table `mytab` can be retrieved with

```
        t = tabVal(mytab, colIdx, rowIdx, addr);
```

where `t` represents the attribute table value in the (`colIdx + 1`)th column and (`rowIdx + 1`)th row of table `mytab`. In this case `mytab` represents a stand-alone table and is the `UserID` of a `TableReader` input component. In contrast to a stand-alone table, an input image's RAT is referenced by the input component's `UserID` (e.g. `img`) extended by the suffix `_t`, i.e. `img_t`. This is required to distinguish between the image pixel values and the values in its associated RAT since each input component only has one `UserID`. Also note that similar to the `kwinVal` function, the `tabVal` function also requires the pre-defined constant `addr` to be supplied as a function parameter.

Referencing the Output Value

The variable name of the output pixel value is user-defined and has to be specified via property `OutputVarName`.

General Kernel Script Syntax

A kernel script is made up of a set of variable assignments, e.g.

```
        var = a + b;
```

The right hand side (i.e. right of the '=' sign) is represented by a muparser expression, concluded by a semi-colon ';'. See the section Mathematical Expression for a description of LUMASS supported muparser expressions. If the left hand side (e.g. '`var =`') is missing, as common for the test expression in a for-loop header (s. below), an implicit variable is assigned to it. The typing of variables is implicit. Every variable is of type `double` (cf. Data Types and Type Casting). Variable assignments have to be specified explicitly using the '`=`' sign. A side-effect assignment similar to the C-style '`++`' or '`--`' operator is not supported. Also not support are the operators '`+=`', '`-=`', '`/=`', and '`*=`'. The following listing shows a simple sample script including a C-style for-loop.

```
        size=10;
        out=0;
        b=5.7;
        for (i=1; i < size; i = i+1)
        {
                b = b * i;
        }
        out=b;
```

A for loop may not be nested in a muparser expression, as for example

```
        var = a < 0 ?
                for (int myvar=0; myvar < numPix; myvar = myvar+1)
                {
                        out=out+1;
                }
                : 0;
```

However, for loops itself may be nested, e.g.

```
        for (i=0; i < number; i=i+1)
        {
                for (g=4; g >=0; g = g-1)
                {
                        out = i*g;
                }
        }
```

and muparser expressions may be used in the loop header or body.

Reserved Names (Kernel Script Keywords)

The following list represents the reserved names within a kernel script. These names must not be used for user variables:

`numPix` : Number of active pixel in the user-defined neighbourhood
`centrePixIdx` : 1D neighbourhood index of the centre pixel
`addr` : `MapKernelScript2` instance identifier
`thid` : Thread identifier
`kwinVal` : Function to access neighbourhood values by 1D index
`tabVal` : Function to access table values by column and row index
`neigDist` : Function to access distance between centre pixel and user specified pixel
mathematical constants: `e`, `log2e`, `log10e`, `ln2`, `ln10`, `pi`, `euler`

Auxiliary Output Data

In addition to the computed output image `MapKernelScript2` provides auxiliary output data to enable feedback between pixel-level computations and model component control flow. After the component has completed the computation of the output image, it creates a table with a simple summary statistic for each explicitly and implicitly defined kernel script variable. The columns of the table represent the individual variables and the rows of the table represent the summary statistics: (in this order) i) minimum value, ii) maximum value, iii) mean value per image pixel, and iv) sum of variable value across threads. Please note that the kernel script is executed for each individual pixel and that each individual computational thread stores its own instance (and hence value) for each of the explicitly and implicitly defined script variables. The summary statistic for the script variables is calculated over the thread specific values for each variable after the whole image has been completely processed. Hence, the meaning of these summary statistics depends on how the individual variables are used and updated during the pixelwise script execution. For example, one use case could be the identification of the maximum image value, or counting how often a particular pixel value (state) has been changed from one value to another value over the whole image.

Properties

`Radius` (`QList<QStringList>`): A list of integer values specifying the neighbourhood (i.e. kernel) size for each dimension of the image. Hence, the length of the list equals the number of dimensions of the input image.

`KernelScript (QStringList)`: The kernel script to be executed to calculate individual output pixel values.

`InitScript` (`QStringList`): An optional initialisation script, which is executed once before the output pixel values are calculated. Note that this is suitable to initiate global counting variables or to calculate constant parameters that can then be accessed in the pixelwise computation.

`KernelShape` (`enumeration`): If a kernel radius of greater than 0 is specified, this property defines the shape of the neighbourhood.

`OutputVarName` (`QStringList`): Variable name for the output pixel value.

`Nodata` (`QStringList`): Defines the value to be assigned to the output pixel value for a non-neighbourhood-based map script in case of an overflow or underflow result value. (Note that this behaviour is to be extended to neighbourhood-based computations.)

`NumThreads` (`int`): Sets the maximum number of threads to be used by this component for parallel processing.

SQLProcessor

Overview

The `SQLProcessor` enables SQL processing on SQLite database tables. Potential input components, which may provide SQLite-based RAT or stand-alone tables are `ImageReader` (s. `RATType`), `TableReader`, and `DataBuffer` or `DataBufferReference` respectively. The component may be incorporated into an image processing pipeline, as long as its first input (at index 0) represents an image with an associated RAT. This image and its RAT is passed on as the only output (at output index 0). Streaming may occur depending on the downstream components. For example, if an `ImageWriter` is the only downstream component and its `WriteImage` property is set to false, streaming is omitted and the output table is directly passed on to the writer.

Properties

`SQLStatement` (`QStringList`): The SQL statement to be executed on the input table(s).

Important Model Development Guidelines

- Execution order flows from higher to lower time levels and from higher aggregation levels to lower aggregation levels (or from the outside to the inside).
- Processing pipelines
  - Processing components linked into the same processing pipeline and sharing the same host component have to sit on the same time level to ensure proper initialisation.
  - A processing pipeline may reach across aggregate component boundaries as long as all of its components only contribute input data to down-stream components, i.e. components that are positioned on a lower level in the model hierarchy.
- Repetitive execution of model components is controlled via the `NumIterations` or `NumIterationsExpression` property of an aggregate component.
- Conditional execution of model components is realised via the `NumIterationsExpression` property of aggregate components.
- The input and output properties dimension, data type, and number of bands need to be defined explicitly for each individual process component.
- Watch the `Notifications` window for warnings and error messages to help you debug and test your model.

How to create a simple processing pipeline

0. Start LUMASS and select `Model View Mode` from the `View` menu (`View` | `Model View Mode`). This displays the `Table Objects`, `Model Components`, and `Component Properties` display areas and collapses the mapping related display areas.

1. **Add a process component to the model:** Select the `ImageReader` entry in the `Model Components` list and drag it into the `Model View`.

2. **Add the `ImageWriter` component** to the `Model View` using drag and drop.

   Note: If none of the tools in the tool bar is selected (cf. Fig. 2-1 to 2-5 and 2-7), you can reposition the individual components using the mouse. Clicking with the left mouse button on a component shows its properties in the `Component Properties` window. A process component comprises the general model component properties (at the top) and the set of general process component properties at the bottom (below the `ProcessName` property). Refer to Table 1 for a short description of the general component properties. Please note that each process component has an additional set of properties, which define process specific parameters, such as file names.

3. **Link the process components:**
   a. Select the `Link Components` tool (Fig. 2-7) from the tool bar.
   b. Left click on the `ImageReader` component and hold the left mouse button down.
   c. Move the mouse pointer onto the `ImageWriter` component and release the left mouse button.
   d. Deselect the `Link Components` tool.

   Note: Always link components starting at the source (output) and ending at the target (input). Click on the `ImageWriter` component and inspect its `Inputs` property. The `ImageReader` Component is now listed as an input component to the `ImageWriter`.

4. **Define/double check the components' properties:** For the error-free execution of the processing pipeline, it is important to double check the components' properties.

   Note: An important concept of the modelling framework is that it requires an image's data type, dimension, and number of bands (image characteristics) to be explicitly specified. Furthermore, the image characteristics of an output image have to match the image characteristics of an associated input image. If these characteristics do not match, the pipeline will not execute properly and produce an error. For model components, which don't have a different input and output type, such as reader and writer components, define the input and output components identically.

   a. **Define the input file name:** Use your filesystem browser to navigate to `SampleData/data` folder. Select the image file `LUMASS_icon_2048.kea` and drag it onto the `ImageReader` Component. This adds the absolute file name to the `FileNames` property of the `ImageReader` component.

   b. **Define the `ImageReader's` image properties:** Apply the following settings to the component's corresponding input and output properties:

      `ComponentType`: `uchar`
      `NumBands`: `4`

   c. **Define the output file name:** Click on the `ImageWriter` to display its properties in the `Component Properties` window. Click on its `FileNames` property to open the parameter editor (Fig. 9). Point into the left hand area of the dialog below the text `#Iteration`. From the context menu (right mouse click) select `Insert parameter here`. This inserts an editable file name parameter into the parameter list. To facilitate editing the output file name, drag the input file, as defined for the `ImageReader`, into the `Edit parameter` window. This inserts the file's absolute file name into the editor window. Delete the leading sequence of characters ('`file:///`') and change the image file name to `output_exercise1.img`. Click the Apply button to transfer the parameter to the component's `FileNames` property.

Note: LUMASS uses a forward slash '/' as path separator independent of the operating system. Furthermore, the `ImageWriter` component recognises the output image format from the defined file name suffix. Hence, this pipeline can be used to convert images between different formats as long as they are supported by the underlying GDAL library.

d. **Define the ImageWriter's image properties:** Apply the following settings to the component's corresponding input and output properties:

```
ComponentType: uchar
NumBands: 4
```

5. **Save the pipeline as LUMASS model**: Select the ImageReader and ImageWriter components by holding the CTRL key and a left mouse click on each component. Now point the mouse on one of the components and right click with the mouse to open the context menu. Select Save 2 Components As … Select a file name and folder for the model file and save the model.

   Note: LUMASS models are comprised of two different files differentiated by their file name suffix. The *.lmx file saves the actual model as XML representation, whereas the *.lmv file stores binary version of the the visual representation displayed in the `Model View`. While saving or reading a LUMASS model file, only one file needs to be explicitly specified. The corresponding other file is read/saved automatically by LUMASS. To execute a LUMASS model with the lumassengine commandline application, only the *.lmx version of the file is required. However, to edit the file in the LUMASS user interface, both files need to be available.

6. **Display the `Notifications` window**: Select the `Notifications` option from the `View` menu to open the `Notifications` window. It displays information, warnings, and error messages occurring during a model run.
7. **Execute the model:** To execute the model either click on the `Execute Model` button on the tool bar (Fig. 2-10) or open the context menu of the `ImageWriter` component and select `Execute ImageWriter`. If the notifications window does not show any error messages, you should find a newly created image file at its specified output location.

LUMASS Engine

The LUMASS software comprises two applications (executables), a desktop application including a graphical user interface (Fig. 1) and a command-line application, i.e. the lumassengine. Whereas the former application is meant for model development and viewing results, the latter application is solely meant for running either separate spatial optimisation scenarios, or LUMASS models (Spatial System Dynamics Modelling Framework):

```
LUMASS (lumassengine) 0.9.52

Usage: lumassengine --moso <settings file (*.los)> | --model <LUMASS
model file (*.lmx)> [--logfile <file name>]
```

The lumassengine enables the execution of LUMASS models in compute cluster or server environments.

References

Johnson HJ, McCormick MM, Ibanez L, Insight Software Consortium, 2016. The ITK Softeware Guide Book 1: Introduction and Development Guidelines, Fourth Edition, Updated for ITK version 4.9. http://www.itk.org/ItkSoftwareGuide.pdf

Neuwirth C, Peck A, Simonovic SP 2015. Modeling structural change in spatial system dynamics: A DaisyWorld example. Environmental Modelling & Software 65: 30—40.

Watson AJ, Lovelock JE 1983. Biological homeostasis of the global environment: the parable of DaisyWorld. Tellus B 35(4): 284—289.

White R, Engelen G, Uljee I 1997. The use of constrained cellular automata for high-resolution modelling of urban land-use dynamics. Environment and planning 24(3): 323—343.