BASIS – **B**ATCH **A**NALYSI**S** OF **I**MAGE **S**ETS

GUIDE AND INTRODUCTION

CONTENTS

BASIS is a lightweight, functional dataflow framework for Matlab. The purpose of BASIS is the following:

- To provide a graphical framework for defining image processing and machine vision workflows,
- To provide a method for archiving image processing and machine vision workflows with future repeatability in mind,
- To save time when developing workflows by taking over boilerplate code generation
- To efficiently run image processing workflows,
- To shorten the syntax for batch-processing image sets as much as possible.

BASIS VERSION

This is the first release, version 0.1.

BASIS REQUIREMENTS

BASIS has been developed in Matlab 2016b. Graph methods make heavy use of Matlab's digraph class and its methods, therefore it will not work with earlier versions of Matlab.

The basic functionality of BASIS does not require any additional Matlab toolboxes. However, if BASIS is used for image processing and machine vision tasks, most functions will require the Image Processing and Optimization toolboxes.

As of this release, BASIS does not support Deep Learning-related functionality.

THE PURPOSE OF THIS DOCUMENT

This document is an overview of BASIS in terms of its concept, basic idea and overall functionality. This is not a tutorial or user's manual. Tutorials, workflows and tests can be used to learn the use of BASIS. These will be referenced in this document in topics connected to the proper tutorial, workflow or test.

GETTING HELP

All tutorials, tests and workflows can be accessed in BASIS by typing **`tutorials.`**, **`workflows.`** or **`tests.`** and hitting tab in the Matlab command line. This will list the available tutorials, workflows and tests. Workflows and tests run on their own. Tutorials will open Matlab scripts that explain and teach the use of BASIS. Note the dot (.) after the commands.

All BASIS classes and functions are documented by in-code comments. Type **`help <class or function name>`** or **`doc <class or function name>`** or right click on the source file of a class or function and select "view help" to display the in-code help. Opening the source code and reading the in-code comments is also available. Off-line HTML documentation for all classes and functions is also available. The HTML documentation has been created by Doxygen. For this reason, source code displayed there will contain C++ syntax due to the .m to .cpp filter. Nevertheless, in-code comments and code structure will be legible and searchable there. Type **`basis_help`** in the Matlab command prompt to access the Doxygen docs from Matlab's web browser or launch the main.html file from the \docs\html_docs folder.

INSTALLATION

Copy the BASIS folder to a folder on your computer. Start Matlab and navigate to this folder. Type

```
install
```

in the Matlab command line. This installs BASIS. You do not need to install BASIS every time you run Matlab. The folder that you copied BASIS into will become your "basis folder". All paths in this document and other BASIS documents treat the "basis folder" as the root folder. I.e., if your basis folder is **`C:\work\basis`**, then a path **`\data\graphs\graph.gml`** will mean **`C:\work\basis\data\graphs\graph.gml`**. You can get the basis folder by typing **`get_basis_path`** in the Matlab command line. This utility is also used to refer to the basis folder in Graphs.

GETTING STARTED

The quickest way to get to know BASIS and the image processing packages is to open and run the tutorials. Study the .gml files of each tutorial, see them in action and try to understand what is being done. Start with the less complicated Graphs and move to the more complicated ones.

THE GRAPH

The main unit of BASIS is the Graph object. A Graph is a representation of a complex workflow, i.e., a collection of functions that use shared inputs and feed outputs into successive functions. Computation is represented by a Directed Acyclic Graph (DAG) in BASIS. Node objects in Graphs are analogous to graph nodes in DAG's. Nodes are assigned a Pipeline that defines elementary computation. Pipelines can chain simple ProcessingFunctions. The Graph object maintains inputs and outputs of Nodes. When a Node is run, its Pipeline is called on its inputs and the Node output is set equal to the output of the Node's Pipeline. Node Inputs and Outputs contain data as Data objects. Data objects encapsulate any data in a handle, so that no data is copied between Nodes, but merely passed by reference. A Graph is similar to a miniature Matlab workspace with its own variables (Data objects residing in Nodes). The structure of a Graph in terms of DAG adjacency determines an algorithmic workflow that utilizes the internal variables of the Graph.

EXPLORING GRAPHS

Graphs can be explored in three ways: through the command line, by using the .summary() method; by plotting the Graph in Matlab via the .plot() method and by using a dedicated graph editor program.

The first two methods work well when you need a quick overview of your Graph. However, the third method is preferred when you need more detailed information in a better formatted manner. For this, an excellent program is called **yEd**. yEd is free, user-friendly and platform independent. When working with BASIS Graphs in yEd, remember to use the .gml format instead of the default .graphml format.

GRAPH CALLORDER

The CallOrder property of Graph objects stores the order in which Nodes should be run so that Nodes are only run when all their Inputs have been computed. Nodes that must be run before a specific Node X can be run are called Upstream Nodes relative to Node X. Nodes that use Node X's Output are called Downstream Nodes.

GRAPH EFFICIENCY

Graph Nodes only run if they have no data in their Output field. This way, when a Node has already been run once, it will not run again with the same Inputs unless the Output field is reset. This is by design and to ensure that computation is not repeated unnecessarily. Usually, when the data of a Node is updated, all Downstream Nodes are cleared as well, since their Outputs depend on the Output of the updated Node.

BASIS Graphs have functionality for cloning Data between isomorphic Groups of Nodes. Two Groups are isomorphic if the subgraphs defined by their Nodes and connectivity have the same structure and carry out the same computation. When processing image sets, i.e., video data, a usual task is to process frames pairwise. For such cases, BASIS Graphs have the capability to clone Data between Groups so that the same computation is not carried out twice. An example: in a video sequence of frames #1-5, frames #1 and #2 are processed first in Groups 1 and 2, respectively, then frames #2 and #3 and so on. When processing the first pair, the computation done for frame #2 can be reused in the processing of the second pair by cloning Data from Group 2 from processing the first pair to Group 1 when processing the second pair.

- **`tests.run_some_nodes`**
- **`tutorials.clone_group`**
- **`tutorials.nodes_wont_run_twice`**

GRAPH CONSTRUCTION

BASIS Graphs can be constructed in several ways, such as in Matlab's command line via adding Nodes one-by-one, in a Matlab script via defining a list of Nodes and graphically, via using any graphical editor that supports the Graph Markup Language (GML) format. If the graphical method is used, BASIS utilizes a mini script language that can interpret node labels as Pipeline definitions.

When using a GML editor, again, **yEd** is the preferred solution. Remember to always save your designed Graph in .gml format. See the tutorials below and study the example Graphs to get an idea about how to graphically design Graphs using yEd.

The Graph class sets its own properties based on some (supposedly) clever logic. If the underlying DirectedGraph or Nodes property change, a listener automatically recomputes all relevant Graph properties. Class mechanisms are set up so that the Graph class is easy and natural to use.

- **tests.graph_from_gml**
- **tutorials.graph_construction_add_nodes**
- **tutorials.graph_construction_cell_of_nodes**
- **tutorials.graph_construction_from_gml**

BASIS FUNCTIONS

BASIS comes with a set of elementary functions that are normally used in image processing and machine vision workflows. These elementary functions include capability to merge data, convert images, load images, preprocess images and so on. These functions can be used to define ProcessingFunctions and chain them into Pipelines. Functions are categorized in the following categories:

Category	Purpose	Conventions
arithmetic	combine images or other data	takes multiple images, returns less images of the same size
data	read, write and manipulate data	typically operates on tables or structs
detection	detect objects in images	takes an image, returns a binary image, true at detected objects
filtering	filter data	typically operates on tables or structs
matching	match features in different data	typically operates on tables or structs, produces pairs of matched indices
measurement	calculate object properties	takes a binary image and returns a table of measured parameters
preprocessing	preprocess images	takes an image and returns a preprocessed image
misc	various tasks	normally void functions, i.e., they return nothing or return a status flag
techniques	technique-specific tasks	can be any of the above, but pertain to specific techniques

The basic concept of functions is that they are heavily overloaded so that many tasks can be flexibly carried out by using a single function. This way they act as building blocks of Graphs and there is no need to memorize many function names.

It is common that functions follow this input-output convention:

output = function(input, optional input, name-value pairs),

where input is required and normally specifies an image or a detection table, the optional input is a positional argument that can specify e.g., a second image or table and name-value pairs are optional and parametrize the function.

A parameter of a BASIS function should be something that can be written comfortably in a closed form. Complex data should be passed as additional inputs.

In any other sense BASIS functions are simple Matlab functions.

BASIS PIPELINES

BASIS Nodes run Pipelines that are BASIS class objects. Pipelines can run a chain of functions called ProcessingFunctions. Within a Node, there is no variable routing – all ProcessingFunctions operate in a chain, receiving the output of the previous ProcessingFunction. There are several ways to define Pipelines. You can use a Matlab script, the command line, and a mini-language to script Pipelines in .gml files. The mini-language is built upon Matlab's anonymous function and function handle syntax. Pipelines and ProcessingFunctions can be called through their .apply method.

☞ **`tutorials.pipeline_script_language`**

INPUT-OUTPUT ROUTING

One of the main tasks of BASIS Graphs is to route inputs and outputs to functions. Inputs and outputs are routed via the Graph edges. There are a couple important points to remember:

1. Routing is done in a directed, acyclic fashion. There can be no loops in the Graph. The .validate method will not let cyclic Graphs compile.
2. A Node can have multiple inputs. Inputs are passed as **SEPARATE ARGUMENTS** to the Node Pipeline. If you need to zip a number of inputs, use an auxiliary Node running e.g., data.merge.
3. A Node Pipeline can have multiple output arguments. In this case, the **OUTPUTS WILL BE ZIPPED INTO A MATLAB CELL**. A Node only has one Output Data.
4. If a Node Output is routed to multiple Nodes, **THE ENTIRE OUTPUT IS ROUTED** to every successor Node. If you need to split Outputs, use an auxiliary Node running e.g., data.split.
5. **NODE INPUT ROUTING ORDER MATTERS!** Input route order can be set by giving edges integer labels (in a .gml), or modifying the InputNodeIDs property of the Node directly (in a script). When labeling a .gml, you don't need to number every edge, only the ones that have an important order. All unnumbered edges will be passed after the numbered ones in a random order.

THE EVENT-FRAME STRUCTURE MODEL

The Event-Frame data model is a model for batch image data that BASIS uses. Images are assumed to be arranged in Frames and Events. A Frame is a single image. An Event is a group of Frames that correspond to a singular event. In high-speed photography terminology, Events are analogous to trigger, and Frames are analogous to sync signals. An Event can contain a short video sequence, i.e., a number of Frames. A Case is a collection of Events. A Case is therefore an image set.

The following are assumptions that are made in the Event-Frame model:

1. Events are regularly spaced in time. The time separation of Events is called δt_{event}
2. Frames within Events are regularly space in time. The time separation of Frames is called δt_{frame}
3. All Frames are acquired before a new Event starts
4. A single Case contains a collection of Events

READMODE

Cases can have different types depending on their ReadMode property. Two read modes are supported: so-called stepping and rolling modes. It is easiest to understand these two via examples. Suppose we have an image set with images labeled image_1, image_2, image_3, etc. A stepping read mode with 3 Frames per Event would read the images in this order:

Event 1: image_1, image_2, image_3
Event 2: image_4, image_5, image_6
Event 3: image_7, image_8, image_9 …

Whereas a rolling mode would read them like this:

Event 1: image_1, image_2, image_3
Event 2: image_2, image_3, image_4
Event 3: image_3, image_4, image_5

A stepping mode is used when discrete Events are captured and each Event has a number of Frames attributed to them. There is no correlation between Events. A rolling mode is used when we work with a continuous video recording, but we want to extract information from every subsequent n-tuples of images in the set. An example is pairwise matching between frames in a continuous recording.

THE DRIVER – FEEDING A CASE TO A GRAPH

The Driver class allows for feeding image data from Cases to a Graph. The responsibility of the Driver is to feed the correct images to frame Nodes of the Graph. Frame Nodes are identified by their labels: they follow the format 'frame_<frame index>', where frame index is the linear index of the frame Node.

The Driver.run method makes sure that no Graph branches are computed twice with the same data i.e., in rolling mode image sets. In these cases, Driver clones data between isomorphic Node Groups.

There are a number of options of Driver behavior, i.e., frames can be read sequentially, randomly, frames can be skipped, etc.

DATALINK

The DataLink class is a built-in class in BASIS that allows you to store data resulting from a Graph run without having to stop e.g., the **`Driver.run()`** loop.

A DataLink object can be in-memory or on-disk. If in-memory, data will be stored in the Matlab workspace, in the DataLink object. If on-disk, you specify a path to a .mat file on the disk.

If in-memory, your data will end up in the Link property of the DataLink object. The Link points to an object called Store, that is simply a Matlab object with dynamic fields. You can access the data through the **`datalink_.Link.(case name)`** variable, where datalink_ is the name of the DataLink object.

If on-disk, your data will end up in the .mat file. The .mat file will be handled as a Matlab datastore. See **Matlab's documentation** about how to read and write .mat files as datastores. In any case, you access your data similarly to the in-memory case, except in this case, data will be loaded from disk.

☞ **`tests.store_data`**

GRAPH-LEVEL INPUT-OUTPUT

Data will flow in and out of a defined Graph object.

Data flowing in will include loaded parameters, constants and of course the image frames.

Data flowing out can be intermediate results, final results, visualization, etc.

The following will come up frequently:

1. Constants, parameters, etc. that only need to be loaded once, at the beginning of **`driver.run()`**, should be loaded using i.e., the **`data.load_parameter`** function. In accordance with the "nodes only run once" policy, this Node will only run once, i.e., at the beginning of the **`driver.run()`** loop.
2. Image frames can be fed by using the Driver object. See **The Driver – feeding a Case to a Graph** section.
3. Intermediate results can be displayed or visualized using the appropriate functions, e.g., **`misc.show_detection`**.
4. Data that need to be stored, i.e., the final results, should be stored using a DataLink object. A DataLink object must be instantiated in the workspace before the Graph is run. You can do this manually, or you can use the **data.create_datalink** method. See the example workflows to see how this is done. See the **DataLink** section to get more information about the DataLink class.

To get a feel about how to define Graph-level input-output functionality, take a look at any of the complex workflows and look for the Nodes "datalink", "store" and "params_<demo>" where <demo> is the name of the demo.

- ☞ **`workflows.shadow_full_workflow`**
- ☞ **`workflows.streak_full_workflow`**
- ☞ **`\data\graphs\shadow_simple_processing.gml`**
- ☞ **`\data\graphs\streak_simple_processing.gml`**

A STANDARD OPERATING PROCEDURE

The standard operating procedure (SOP) for most practical cases that you wish to process using BASIS utilizes the Case, Graph and Driver classes.

An instance of the Case class stores information of a user case, that is, a collection of images on the disk.

An instance of the Graph class stores information and defines a complete workflow by which a Case is processed. The Graph therefore tells BASIS how to process a particular case; however, a Graph is much more: see **The Graph** for more information. The Graph is also responsible for loading, storing and displaying information.

An instance of the Driver class is responsible for feeding the images of the Case in a particular way to the Graph.

Therefore a basic SOP would look like the following:

1. Acquire images for the particular case.
2. Instantiate a Case object in Matlab that points to the case folder, i.e., to the folder where the acquired images are stored.
3. Instantiate a Graph object in Matlab that defines the workflow by which the Case will be processed.
4. Instantiate a Driver object and feed the Case object and Graph object as input arguments.
5. Use the .run() method of the Driver to process the Case.

After the images have been placed in their corresponding case folder and the Graph is defined in e.g., a .gml file, the entire procedure looks like this:

```
Driver( ...
   Case([folder '\..\..\data\images\streak\stepping']), ...
   Graph([folder \..\..\data\graphs\streak_simple_processing.gml'])).run();
```

Note that calling the .run() method of the Driver object on the same line where it is instantiated results in the object not being stored in the workspace after it runs. The only items indicating that it has run will be outputs it produces or data it stores. In this case, the Graph will display intermediate results and store the calculations in an in-memory DataLink object.

In this case, the case data sits in \data\images\streak\stepping, while the Graph is defined in the .gml file in \data\graphs\streak_simple_processing.gml.

Set up the case folder in accordance with the Event-Frame structure (see the **The Event-Frame structure model** section). Set up the graph in accordance with the **The Graph** section.