COSIVINA: A Matlab Toolbox to Compose, Simulate, and Visualize Neurodynamic Architectures

Sebastian Schneegans
Institut für Neuroinformatik
Ruhr-Universität Bochum

Version 1.1, March 2013

Contents

1 Introduction

1.1 Overview of the Framework

The COSIVINA toolbox is intended as a tool to quickly create Dynamic Neural Field (DNF) architectures in Matlab and simulate their activation time course either in an interactive mode with a GUI, or in an offline mode. The software was developed at the Institut für Neuroinformatik at the Ruhr-Universität Bochum, where it is used both in teaching the concepts of DNFs and in research. It is published under the Simplified BSD License.

The basic idea for the framework is to divide a DNF architecture into individual *elements* that can be implemented as objects of different classes. Assume for example we have a simple one-dimensional dynamic neural field, which has lateral interactions (typically local excitiation and surround inhibition) and receives several inputs, often modeled as Gaussian functions. We can then divide this architecture up into the following elements: The dynamic neural field itself (the distribution of activation that changes according to a dynamic field equation); the lateral interaction kernel that is convolved with the field; and several Gaussian stimuli. These elements are connected to each other: The stimuli feed into the neural field, the element for the lateral interactions receives input from the field and feeds back to it. Larger architectures containing many neural fields that interact with each other through mutual projections or other operations can be segmented in the same way.

The dynamics of the architecture (the change of its state over time) is simulated using the Euler method: The rate of change for each dynamic element is determined for equidistant time steps and assumed fixed for the duration of the step. The segmentation into elements is done in such a way that the operation to be performed in each step can be computed for each element separately, based only on the element's own state, its direct *inputs* and the element's *parameters*.

In the framework, an architecture is constructed by adding elements and specifying their parameters and their connections to each other. The architecture can be run (i.e. the change of its state over time be simulated) either in individual steps or for a fixed duration. The states of of all element can then be accessed, analyzed or plotted, their parameters or states changed, and the simulation continued thereafter. Moreover, the framework offers tools to create graphical user interfaces (GUIs) that can be used to run the architecture while visualizing the states (like activation patterns or outputs) of some or all of its elements online. These GUIs allow online changes of parameters via control elements (sliders, buttons, and edit fields). The settings of an architecture (its elements, their parameters and the connections between them) can be stored in a parameter file, and the architecture can be created again by loading from this file. The framework uses the JSON format (see www.json.org) to store parameter files, and utilizes the Open Source toolbox JSONlab for saving and loading these files.

As a typical usage of this framework for research in the field of behavioral/neural modeling, we envision something like the following: You start with an idea for a DNF model that performs a certain task. You implement that model in the DNF framework, e.g. by writing a script that creates the required elements and their connectivity, using the provided classes. Then you create a GUI for this architecture, which shows you the activations of all fields and maybe the results of some other operations that go on in each step of the simulation. Add some controls to change inputs to the system and key parameters, and try it out. You may have to do some debugging of the connectivity before the simulation actually runs; the framework provides some tools to assist you here. Then find out whether the architecture behaves the way you expected, at least qualitatively. You may have to change or add some elements or connections before it looks good. In the next step, to get the actual

behavior that you want out of the system, tuning of the parameters will almost certainly be necessary. The standard GUI allows you to access all parameters of the elements in your architecture, so that you can change their behavior while the simulation is running. You may also add more controls to have a more direct way to change some key properties. Store the parameter settings that are promising in parameter files as you work on the model. At some point you may want to start running standardized trials with the model, perhaps with some fixed sequence of inputs. You can then move to running the model in an offline mode (without the GUI, though perhaps still with some visualization or analysis as the simulation runs). Just load the architecture from the parameter files and write a script that sets up the desired time course and the necessary analysis of the results. And from here, you can switch between GUI and the offline mode to test the model, analyze its behavior, improve and expand it.

1.2 About this Document

The goal of this document is to enable the users of the COSIVINA toolbox to quickly build DNF architectures, design GUIs for them, and run their models in the way that is most appropriate for their requirements. This document does not provide a general introduction to Dynamic Neural Field Theory, its applications or the underlying mathematics (see http://www.robotics-school.org/ for introductory lectures and materials).

The document is structured in three major parts: The first describes the key components of the framework. The second one provides a reference of the available classes, specifically the elements that can be used to construct an architecture and the controls and visualizations for the design of a custom GUI. The third part describes in detail the construction of some example architectures and GUIs and their use for tuning and testing a model. You may jump directly to this last part to get a quick idea of how to use this framework.

1.3 Object-Oriented Programming and Terminology

Working with the DNF framework does not require detailed knowledge of object-oriented programming, and providing such knowledge would be beyond the scope of this document (see the Matlab help for this). We will, however, give a very brief overview of the concepts and terminology of object-oriented programming in Matlab to facilitate understanding of our framework. The central idea of object-oriented programming is to structure the program code (or parts of it) into *classes*, which combine data structures and functions that act on them. A class in Matlab can be written as a single m-file that contains a class definition, which specifies the name of the class and its relationship to other classes. This class definition is typically followed by a list of persistent variables of this class, called *properties*. A class can furthermore define functions, which typically act on these properties. These functions are called *methods* of the class.

To use the class and its methods in scripts or other functions, it is typically necessary to instantiate it – that is, to create an *object* of that class. This can be done by calling a *constructor* function for the class, which has the same name as the class itself. The constructor call returns a concrete variable, which contains all the properties of the class (analogous to the fields in a struct). One can access these properties and call the methods of the class via the dot-notation (in the form `objectName.propertyName` or `objectName.methodName(...)`). Some properties may not be accessible (or not accessible for writing) except via methods of the class. It is possible and common to instantiate multiple objects of the same class. These objects then have the same structure and offer the same methods, but the content of their internal variables may be entirely different.

The DNF framework strongly relies on so called *handle classes*: When a handle class is created, it is instantiated in memory and a handle to it is returned as variable in the workspace. This handle allows access to the class's properties and methods via the dot-notation in the same way as in non-handle classes. However, when copying this variable, only the handle to the existing object is copied, not the object itself. This is similar to graphics handles in Matlab, which can be copied without multiplying the graphics object. The handle class object itself is destroyed when all handles referring to it are removed from the workspace.

We will furthermore use some fixed terms below that are specific to our framework (and not to be confused with general object-oriented programming terminology). In particular, we will refer to the *parameters* and *components* of an element. These two terms refer to different types of properties of the element classes, that are distinguished by their role for the behavior of the element, not by their implementation.

1.4 Preparing the Framework for Use

To use the framework, download the compressed Matlab sources and unpack them in a folder of your choice. Then add the subfolders `base`, `controls`, `elements`, `examples` (optional), `mathTools`, and `visualizations` to your Matlab path. You can do so manually via the entry `Set Path ...` in the Matlab `File` menu (choose `Add with subfolders`, then save the settings for future Matlab sessions), or call the function `setpath` in the COSIVINA base directory.

The full functionality of COSIVINA is supported by Matlab R2011a and later. The framework can also be used with earlier versions (back to at least R2009a), but then requires a small adjustment to run: In the file `base/Element.m`, replace the class definition

```
classdef Element < matlab.mixin.Copyable
```

with

```
classdef Element < handle
```

(both forms are prepared in the file, just comment/uncomment the appropriate line). With this change, the `copy` functions of the `Element` and `Simulator` classes are no longer functional. Except for this, the framework remains fully functional, including the creation and use of GUIs for interactive simulations.

In order to save and load architecture settings to/from configuration files, you additionally need the Open Source toolbox JSONlab. You can obtain it from

```
http://sourceforge.net/projects/iso2mesh/files/jsonlab/
```

The current version of COSIVINA has been tested with JSONlab versions 0.9.0 and 0.9.1. The location of this toolbox also needs be added to the Matlab path. The `setpath` function will do so if a folder `jsonlab` exists either as a subfolder in the COSIVINA base folder or on the same level as the COSIVINA base folder itself.

To test the toolbox, call one of the scripts from the `examples` folder. For instance, call `launcherOneLayerField.m` from the Matlab command line (the `launcher...` files create a DNF architecture and accompanying GUI and then run that GUI). This should open a GUI window with a running DNF model, in which you can change field parameters and input settings via sliders. Press the `Quit` button to close the simulation. You can also run the example scripts, which are explained in detail in the last part of this document.

Part I
Structure of the Framework

2 Element Class

Different types of elements (like neural fields, lateral interactions, and projections between fields) are implemented as different classes in the DNF framework. These classes are all derived from the abstract superclass `Element`, which defines a common structure for all elements.

2.1 Common Properties

2.1.1 parameters

When we talk about the *parameters* of a class, we refer to a certain subset of its properties (class variables) which together fully define the behavior of an element. Each element class contains as one (constant) property a struct named `parameters`. This struct contains the names of all parameters of the element together with information about whether they may be changed during a simulation. It is important to note that this struct does not contain the parameter values themselves. Instead, it only provides the information which parameters the element has and how they behave, which is important e.g. to automatically generate GUI panels that allow manipulation of parameters online. The parameter values themselves are stored in individual properties of the element class.

For instance, for the class `NeuralField`, the `parameters` struct has the following form:

```
>> NeuralField.parameters
ans =
    size: 0
     tau: 1
       h: 1
    beta: 1
```

This tells us that the elements of this class have four parameters, namely `size`, `tau`, `h` and `beta`. The integers following the parameter names inform us whether the parameter may be changed online, as described in Section 2.4. If we create an object of this class, we get a list of its properties:

```
>> nf = NeuralField
ans =
  NeuralField handle
  Properties:
                parameters: [1x1 struct]
                components: {'input'  'activation'  'output'  'h'}
    defaultOutputComponent: 'output'
                      size: [1 1]
                       tau: 10
                         h: -5
                      beta: 4
                     input: []
                activation: []
```

```
                output: []
                 label: ''
               nInputs: 0
          inputElements: {}
        inputComponents: {}
```

We see that the four parameters listed above now appear as properties of the object, among others. Their values are the actual parameter values (e.g. `h = -5`), in this case the default values assigned to them by the class constructor.

2.1.2 components

A second subset of properties is called *components*. These vectors or matrices contain the state of the element during the simulation (for example, the current activation distribution of a neural field) or the results of the computation performed by an element in every simulation step (for example, the convolution of an input with an interaction kernel). These components can serve as inputs for other elements or be plotted in a GUI's visualizations. Analogously to the parameters, there is a single constant property `components` in each class. We can view it without actually creating an object of the class:

```
>> NeuralField.components
ans =
  'input'  'activation'  'output'  'h'
```

This property lists all of the class's components as a cell of strings.

As shown in this listing, the `NeuralField` class has the components `input`, `activation`, `output`, and `h`. (Note that a single property can serve as both a parameter and a component, as is the case here for the resting level `h`.) The components are typically created as empty matrices in the element constructor and then brought to their correct size during the initialization (see below).

In addition to the `component` property, each element furthermore has a property `defaultOutputComponent`. This string gives the name of the property that is used as the default when projecting from that element to other elements. For most elements, including the `NeuralField` class, the name of this component is simply `output`:

```
>> NeuralField.defaultOutputComponent
ans =
  output
```

2.1.3 label

Each element that is added to the architecture in a `Simulator` object must have a unique label. A valid label can be any non-empty character string. The label is used to refer to this element in the simulator, e.g. when specifying the connectivity between elements or accessing an element for analysis or parameter changes.

2.2 Methods

2.2.1 init

The `init` method initializes the element. After initialization, all components of the elements are created as vectors or matrices of the correct size (filled either with zeros or with appropriate values), so that they can serve as valid inputs for other elements. The `init` method

furthermore performs all preparatory computations and may prepare internal data structures for the `step` method to be executed. For certain elements, it also opens a connection to hardware or interfaces to other programs.

2.2.2 step

The method `step(time, deltaT)` performs one simulation step (Euler step) at a simulation time given by the first argument and of a duration given by the second argument. It fetches input from other elements and updates the element's components as applicable. Note: The arguments `time` and `deltaT` are in practice only used by a small subset of element classes, and ignored in the step functions of other classes.

2.2.3 close

The `close` method disconnects the element from hardware or external interfaces. In pure simulation settings, use of this method is generally not necessary. In these cases, calling this method has no effect.

2.2.4 copy

The `copy` method creates a shallow copy of an element. The method is inherited from `matlab.mixin.Copyable`, the superclass of `Element`. Since all element classes are handle classes, a simple assignment does not copy the element, but only creates a new handle to an existing element. Consider the following example:

```
h1 = NeuralField('field u', 100, 10, -5);
h2 = h1;
h1.tau = 20;
h2.tau
ans =
    20
```

Here, the handles `h1` and `h2` refer to the same underlying element. When a parameter is changed via handle `h1`, the change will also appear when the element is accessed via handle `h2`. In contrast, the following code creates an actual copy of the element:

```
h1 = NeuralField('field u', 100, 10, -5);
h2 = h1.copy();
h1.tau = 20;
h2.tau
ans =
    10
```

The two handles now point to different elements, so a parameter change via one handle will not change the element accessed by the other handle.

2.2.5 addInput

The method `addInput(inputHandle, inputComponent, optArg)` is used to manually add an input to an element. The connections in an architecture are in practice stored in such a way that each element has a list of elements that it receives inputs from, and a list of the components of these elements that serve as inputs. When creating an architecture, the

connections between the elements are typically defined via the `addElement` function of the `Simulator` class (which calls this method internally), such that direct calls of the `addInput` method are not necessary in most cases.

2.2.6 getParameterList

This method returns a list of the element's parameter names as a cell array of strings.

2.3 Note on Size Parameters

At this time, the framework effectively supports one- and two-dimensional DNFs and connecting elements. Most elements have a parameter `size` that indicates the size of the field or other structure represented by the element. These sizes are described by a two-element vector, where the first element indicates the number of rows, the second the number of columns of a matrix (following Matlab conventions). One-dimensional data structures are generally stored as row vectors, and if a scalar is given as a size parameter, it is interpreted as the number of columns in a row vector (so a value `n` given as a size argument is interpreted as `[1, n]`).

2.4 Notes on Parameter Changes and Initialization

The elements of the DNF architecture are implemented in a way that attempts to optimize their run time efficiency in longer simulations. This often requires some auxiliary computations that take place before the `step` function is called to make the `step` function itself run faster (e.g. the computation of a convolution kernel for lateral interactions). These computations are performed during the initialization of an element. As a side effect, changes of an element's parameters do not always have a direct effect on the computation in the `step` function, because some parameter values are only used in the auxiliary computations. Checking for a change in these parameters during each step would be inefficient, therefore a manual re-initialization may be necessary.

The information which parameters changes require a re-initialization of the element to take effect is stored in the element classes (it is used to automatically initialize elements when changes are made via a GUI). You can find this information by viewing the m-file for the class or by inspecting the `parameters` struct of a class, for example:

```
GaussStimulus1D.parameters
ans =
          size: 0
         sigma: 2
     amplitude: 2
      position: 2
      circular: 2
    normalized: 2
```

This yields a list of the element's parameters, and for each parameter a number indicating its changeability status. (Note that these are not the parameter values, and that the parameters can be listed without having a concrete object of that class.) The meaning of these integer values is fixed in the constant class `ParameterStatus`: 0 for fixed, 1 for changeable, 2 for initialization required. Fixed parameters cannot be changed via a GUI, and changing them manually during a simulation may require special care (for instance, changing the size of one element may not be possible without also adjusting the size of other elements). For

parameters listed as changeable, changes in parameter value will take effect immediately without requiring initialization. When changing parameters that are listed as requiring initialization, the element may show inconsistent behavior if the `step` function is called again without prior re-initialization of the element.

When re-initializing connective elements during a simulation, it is advisable to manually call that element's `step` function once after the `init` function. The reason is the following: The initialization generally sets the output of the element to a matrix of zeros. This output may then be used in the simulator step by other elements before it is updated to its proper values. While the system will behave correctly again in the next step, this may for instance cause a brief lapse of lateral interactions that can change the system's behavior in an undesirable way.

For stimulus elements, calling the `step` function is generally not necessary, since here the components are already filled with proper values during initialization (which is not possible for elements that receive inputs from other elements and these in turn are not yet initialized). For dynamic elements (neural fields and memory traces), all parameters can be changed without reinitialization. This allows parameter changes during an ongoing simulation without resetting the state of the elements.

3 Simulator Class

The `Simulator` class holds all elements of an architecture, manages the connections between them, allows to run the whole architecture as a coupled dynamic system and provides easy access to individual components. It furthermore contains methods to save an architecture to or load it from a configuration file. Below we describe the functions for creating, expanding, and using a `Simulator` object.

3.1 Methods for Creating and Expanding the Simulator

3.1.1 Constructor

A new `Simulator` object can be created by calling the constructor without any arguments, e.g. `sim = Simulator()`. Optional arguments can be provided in the form of parameter name / value pairs (where the parameter name is given as a character string). The parameters `'tZero'` (the starting time of the simulator, default is 0) and `'deltaT'` (the time step, default is 1) can be set, or an architecture can be loaded from a configuration file in JSON format by specifying the parameter `'file'` and the filename, or it can be loaded from a parameter structure by giving the parameter `'struct'` and the struct.

Examples

`sim = Simulator();` creates an empty simulator object with default settings

`sim = Simulator('deltaT', 0.1, 'tZero', 500);` creates an empty simulator object with a time step of 0.1 and a start time of 500

`sim = Simulator('file', 'threeLayerArchitecture.json');` creates a simulator object by loading the architecture and parameter settings from the specified JSON-file

`sim = Simulator('struct', threeLayerArchtecture);` creates a simulator object by loading the architecture and parameters from the struct `threeLayerArchtecture`; the struct must be a valid parameter struct, created e.g. by loading from a parameter file using the JSONlab toolbox

3.1.2 addElement

This method provides the key capacity of the `Simulator` class for constructing architectures, by a adding a new element to the `Simulator` object. The function is called in the form

```
sim.addElement(elementHandle, inputLabels, inputComponents,
  targetLabels, componentsForTargets)
```

The element handle is typically created by calling the constructor of the element that is to be added and providing the necessary parameters for the element. The four following arguments are all optional, and describe the connectivity of the new element to the existing elements of the architecture. They can each be either a single string or a cell array of strings, or be an empty matrix if no connections are to be specified.

- `inputLabels` - the labels of those existing elements in the architecture from which the newly added element receives inputs
- `inputComponents` - the components of the elements specified in `inputLabels` that are to be used as inputs for the new element; if the new element receives multiple inputs, both `inputLabels` and `inputComponents` should be cell arrays of strings, with each pair of entries from the two cells specifying one input; the whole argument `inputComponent` or individual elements of the cell array may be replaced by an empty array `[]`, in which case the default output component of the specified input elements is used
- `targetLabels` - the labels of those elements in the architecture that are to receive input from the new element
- `componentsForTarget` - the components of the new element that should serve as inputs for the elements specified in `targetLabels`; analogous to the `inputLabels` and `inputComponents` arguments, multiple targets may be specified through cell arrays of strings, and `componentsForTarget` may be omitted if the default output component of the new element is to be used

3.1.3 copy

Creates a copy of the simulator object and all of its elements. Since the class `Simulator` is a handle class, a simple assignment does not copy the object, but only creates a new handle to the existing object (see 2.2.4 for detailed explanation). The copy method can be used to create different branches of a simulation, which can be run independently to directly compare how a model that is in a certain state behaves under different conditions.

For instance, you may have an architecture with a complex working memory representation, that takes some time to be built. You may then, after forming this representation, create several copies of the simulator object to test how the model behaves when different new inputs are applied. This way, you do not need to form the working memory representation repeatedly, and can be sure that you always start with the exact same state when comparing different conditions.

3.2 Methods for Running the Simulator

3.2.1 run

The method runs the simulator (i.e. performs simulation steps) until a specified time step is reached. It can initialize the simulator and close it after finishing if requested. It is called as

`sim.run(tMax, initialize, close)`, where the boolean arguments `intialize` and `close` are both optional.

If the simulator is not yet initialized or if an initialization is explicitly requested by setting the argument `initialize` to `true`, the `init` method is called (see below), which initializes all elements and sets the time to `tZero`. Otherwise, the simulation continues from the current state of the simulator. This means that a single continuous run may be replaced by multiple successive calls of `run` (with increasing values for `tMax`), allowing e.g. to perform analysis of the simulator's state or change settings between the individual runs.

If the argument `close` is supplied and its value is `true`, all elements are closed after completion of the run by calling their `close` method (see `close` method below).

3.2.2 init

The `init` method initializes all elements in the simulator object (by calling each element's `init` method), and sets the simulator time property `t` to the value `tZero`. It takes no arguments and returns the initialized simulator object. If the simulator was already initialized, all information about its previous state is lost.

3.2.3 step

The `step` method performs a single simulation step (updating of the dynamical system's state according to the Euler method) by calling each element's `step` function, and increments the simulator's time property `t` by `deltaT`. It takes no arguments and returns the initialized simulator object.

Note: The elements are processed (i.e. their `step` functions called) in the order they were added to the simulator. The order may make a difference for the exact behavior of the system.

3.2.4 close

The `close` method terminates all external connections by calling each element's `close` method. Note: The `close` method is used primarily to disconnect elements that provide interfaces to hardware. It does not affect most other elements, and is typically not required in pure simulation settings.

3.3 Element Access Methods

3.3.1 isElement

The method call `sim.isElement(s)` checks whether the string `s` is the label of any element in the object `sim`, and returns a boolean value.

3.3.2 getElement

The method call `sim.getElement(s)` returns a handle to the element with label `s` if such an element exists in the simulator object, and an empty matrix otherwise.

3.3.3 getComponent

The method call `sim.getComponent(s, c)` with string arguments `s` and `c` returns the component with name `c` of element with label `s` if such exists. Otherwise it returns an empty matrix and throws a warning.

3.4 Methods to Assist in Debugging

3.4.1 tryInit

The method `tryInit` performs the same operations as `init`, but provides additional information if the initialization of any element fails. In that case, the label of the element that caused an error and its properties are displayed. If no errors occur, the method performs a valid initialization of the simulator object.

3.4.2 tryStep

The method `tryStep` performs the same operations as `step`, but provides additional information if an error occurs in the `step` function of any element. In that case, the element label, its properties, and information about its inputs are displayed to facilitate debugging (a frequent cause of erros is mismatch between the sizes of connected elements). For actually running the simulations, the `step` method should be preferred since it is slightly faster than `tryStep`.

4 GUI

The graphical user interface of the framework makes it possible to change parameter values while a simulation is running, so that the behavior of an architecture can be observed and adjusted online. Currently there is one type of GUI implemented in the `StandardGUI` class. Each `StandardGUI` object is connected to a single `Simulator` object. A specific GUI is built up by arranging *visualizations* and *controls*: The visualizations are graphical elements like plots or images that show selected components of the architecture elements. Controls can be buttons, sliders, dropdown menus and other objects that are connected to selected parameters of the architecture elements. Certain controls can globally affect the simulation that is run in the GUI (e.g. pausing or ending the simulation). As a default interface to access and change any (non-fixed) parameters, the `StandardGUI` offers a parameter panel (implemented in the separate class `ParameterPanel`). In this panel, an element from the architecture can be selected via a dropdown menu. All of its parameters are then displayed and their values can be changed via edit fields.

4.1 GUI Layout

The `StandardGUI` has one main figure window on which controls and visualizations can be arranged freely. To make the arrangement of items more convenient, two spatial grids over this window can be defined: The *visualizations grid* and the *controls grid*. When a new `StandardGUI` object is created, the positions and sizes of these grids can be specified. The grid position in the main window must be given in relative coordinates (as four-element vector, as used e.g. to place plots in Matlab). The grid size is given by a two-element vector, determining the number of cells vertically and horizontally. An example is shown in Figure 4.1. The locations of controls and visualizations can then be given as positions in the respective grid, and a size within the grid can be specified if they should span multiple grid cells. Alternatively, all positions can also specified manually as relative positions in the figure.

A position within the grid is described by a two-element vector `[v, h]` (vertical and horizontal position), with `[1, 1]` being the top left cell of the grid. While the typical usage

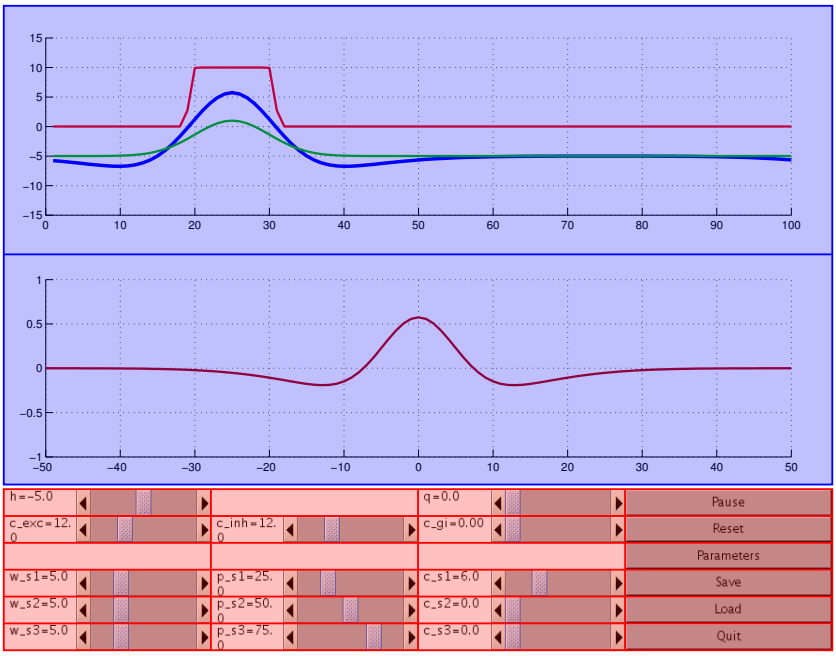

Figure 1: Layout of the GUI in `launcherField1l.m` using visualizations and controls grids. The 2×1 visualizations grid (blue) and the 6×4 controls grid (red) are overlaid over the actual GUI.

is to describe positions by integer values in this grid, non-integer real values and values that fall outside of the specified grid are permissible.

For the visualizations grid, there is an additional parameter `visGridPadding` that determines a space left between the actual visualization and the border of the grid cell. It is given relative to the size of the figure window.

Note that the grids for controls and visualizations may also overlap. An example can be seen in the file `launcherCoupling.m`, where the controls grid covers the whole window (but no controls are actually placed in the area occupied by visualizations).

4.2 Parameter Panel

The parameter panel can be opened in a separate figure window via an appropriate button in the main window. By default, the parameter panel generates a list of all elements in the connected `Simulator` object and shows them in a dropdown menu. For the selected elements, it shows all of its parameters with an edit field containing the parameter values. The values may be changed for non-fixed element parameters, with the changes taking effect after the `Apply` button at the bottom of the panel is clicked. (Note: Changes made for one element are lost if the selection in the dropdown menu is changed without clicking `Apply` first.)

The element list for the parameter panel can be customized by specifying a list of entries and a list of the corresponding elements or element groups in the `StandardGUI` constructor. This customization supports three functions:

- to list the elements in the parameter panel with more descriptive labels than are

actually used in the architecture;

- to eliminate elements from the parameter panel's dropdown menu, e.g. to make the list more concise by removing entries that do not have any changeable parameters or that should not be changed by the user;
- to group elements together, such that changes made via the parameter panel affect a group of elements rather than a single element.

The customization is performed via two parameters `elementGroups` and `elementsInGroups`. The first is a cell array of strings, with each string specifying one entry in the parameter panel's dropdown menu (you may use descriptive labels here, the entries do not have to match the actual element labels). The second parameter `elementsInGroups` specifies one or more elements (via their labels) that are to be associated with the corresponding entry in `elementGroups`. It is either a cell array of strings (if each menu entry is associated with only one element) or a cell array of cell arrays of strings (you may also mix string and cell array of string entries).

4.3 Methods

4.3.1 Constructor

A new `StandardGUI` object is created by calling the class constructor in the form

```
StandardGUI(simulatorHandle, figurePosition, pauseDuration, ...
      visGridPosition, visGridSize, visGridPadding, ...
      controlGridPosition, controlGridSize, ...
      elementGroups, elementsInGroups)
```

All arguments except `figurePosition` are optional.

- `simulatorHandle` - handle to the `Simulator` object that is to be run by the GUI; if the argument is 0, the GUI will be created without being connected to a specific simulator; it must then be connected later using the `connect` method or by specifying a simulator handle when calling `run`
- `figurePosition` - position of the GUI's main figure window as a four element vector `[posX, posY, width, height]`
- `pauseDuration` - duration of pause for every simulation step (default = 0.1, should be set lower for computationally costly simulations)
- `visGridPosition` - position of the visualizations grid in the GUI window in the format `[posX, posY, width, height]`, in normalized coordinates (relative to figure size)
- `visGridSize` - grid size of the visualizations grid as a two-element vector `[rows, cols]`
- `visGridPadding` - padding of the visualizations within the grid cells (relative to figure size), as scalar or as vector `[padHor, padVert]`
- `controlGridPosition` - position and size of the controls grid in the main figure window in relative coordinates (four element vector)
- `controlGridSize` - grid size of the controls grid (two-element vector)

- `elementGroups` - dropdown menu entries for the parameter panel (cell array of strings)
- `elementsInGroups` - labels of elements associated with dropdown menu entries (cell array of strings or cell array of cell array of strings)

4.3.2 addVisualization

The method call

```
gui.addVisualization(visualization, positionInGrid, sizeInGrid)
```

adds the visualization specified in the first argument to the GUI. If the second argument is specified (as a two-element vector `[row, col]`), the visualization will be placed at this position in the GUI's visualization grid. If the third argument is specified (also a two-element vector), the visualization will span the specified number of grid cells vertically towards the bottom and horizontally to the right; otherwise, it will cover one grid cell. Alternatively, the position of the visualization can be specified directly in the visualization object, without making use of the GUI's visualizations grid.

4.3.3 addControl

The method call

```
gui.addControl(control, positionInGrid, sizeInGrid)
```

adds the control specified in the first argument to the GUI. Position information can be specified analogously to the `addVisualization` method.

4.3.4 connect

The method call

```
gui.connect(simulatorHandle)
```

sets the simulator object that is connected with the GUI (that is, which is run in the GUI). The method can be used if no simulator was specified in the constructor call for the GUI, or if the GUI should be connected to a different simulator than originally specified. An error is thrown if the new simulator does not contain the elements, parameters, and components that are specified in the GUI's controls and visualizations.

4.3.5 run

The method call

```
gui.run(tMax, initializeSimulation, simulatorHandle)
```

runs the simulation in the GUI until it reaches simulation time `tMax`. If not specified, `tMax` is set to infinity by default, meaning that the simulation will run until stopped manually from the GUI. The optional argument `intializeSimulation` specifies whether the connected `Simulator` object should be initialized upon start of the GUI. By default, the `Simulator` object is only initialized if it was not initialized before. If the `Simulator` object is already initialized, it will continue at its current state. The optional argument `simulatorHandle` can be used to specify the simulator object that will be run in the GUI. This can be used if no simulator was specified when creating the GUI object, or when the GUI should be run

with a different simulator object than originally specified (e.g. a new copy of the original simulator).

Note: If you quit the GUI via a GUI control, or if the GUI stops after reaching a specified simulation time `tMax`, the simulator object remains in the state that it last had in the GUI. If you call the `run` method again afterwards without explicitly requesting an initialization, you can continue the simulation from this point.

Part II
Class Reference

5 Elements

5.1 Dynamic Elements

5.1.1 NeuralField

Description A dynamic neural field of any size and dimensionality, or a set of dynamic nodes. In each step, the field activation $u(x)$ is updated according to the Amari equation, using the sum of all inputs $I(x)$, and the field output $f(u(x))$ is computed from the activation via a sigmoid function:

$$\tau \dot{u}(x) = -u(x) + h + I(x) \tag{1}$$

$$f(u(x)) = \frac{1}{1 + \exp(\beta u(x))} \tag{2}$$

Lateral interactions in the field have to be implemented by separate connective elements that receive input from the field and project back to it.

Parameters

- `size` - size of the field (determines the size of the three components listed below)
- `tau` - time constant of the field
- `h` - field resting level
- `beta` - steepness parameter of the sigmoid output function

Components

- `activation` - field activation
- `output` - field output (sigmoid of the activation)

(Note: The component `input` that existed for this element in version 1.0 has been removed to improve performance.)

Inputs The `NeuralField` may have an arbitrary number of inputs, which have to be either scalar or have the same size as the field.

5.1.2 MemoryTrace

Description A memory trace for a dynamic neural field of any size and dimensionality, or for a set of dynamic nodes. The memory trace is typically operated with a significantly higher time constant (i.e. slower dynamics) than a field. It typically receives the output of a neural field as input I. If the input at any point exceeds a specified threshold θ, the memory trace is updated according to the rule

$$\dot{m}(x) = \begin{cases} \frac{1}{\tau_{build}}(-m(x) + I(x)), & \text{if } I(x) > \theta \\ \frac{1}{\tau_{decay}}(-m(x)), & \text{else} \end{cases} \tag{3}$$

If the input is below θ everywhere, the memory trace does not change.

Parameters

- `size` - size of the memory trace
- `tauBuild` - time constant for the accumulation of the memory trace
- `tauDecay` - time constant for the decay of the memory trace
- `threshold` - threshold applied to the input to determine whether and where a memory trace should accumulate

Components

- `output` - memory trace
- `activeRegions` - regions in the current input that exceed the threshold (binary vector/matrix of the same size as the input)

Inputs The `MemoryTrace` must have exactly one input whose size matches the element's `size` parameter.

5.1.3 DynamicVariable

Description A matrix of dynamic variables. The summed input is interpreted as a rate of change, which is scaled with a time constant to change the state of the dynamic variables. The dynamic variables behave like a field or set of dynamic nodes without a resting level, that is in the absence of external input they simply maintain their state.

Parameters

- `size` - size of the matrix of dynamic variables
- `tau` - time constant
- `initialState` - state of the dynamic variables at initialization

Components

- `state` - current state of the dynamic variables
- `initialState` - state of the dynamic variables at initialization

Inputs The `DynamicVariable` may have an arbitrary number of inputs, which have to be either scalar or match the element's `size` parameter.

5.1.4 SingleNodeDynamics

Note: This class is intended for easy visualization and analysis of a node's dynamic behavior (in particular for creating plots of rate of change vs. possible activation value, showing attractors and repellors). If you want only the basic behavior of a discrete dynamic node, use the `NeuralField` class instead!

Description The class creates a single dynamic node with a sigmoid (logistic) output function, tunable self-excitation and Gaussian noise:

$$\tau \dot{u} = -u + h + c_{exc} f(u) + I + q\xi \tag{4}$$

Here, τ is a time constant, u is the node activation, h its resting level, c_{exc} the strength of self-excitation (may be negative to create self-inhibition), f is the sigmoid output function, I is the sum of external inputs, q is the noise level and ξ random variable following a normal distribution.

In addition, the class computes the rate of change $\dot{u}$ – given the current external input – for a range of possible activation values u of the node, and determines approximate attractor and repellor states within this range (zero crossings of the rate of change). The components provided by this class can be used in particular for plotting the node dynamics using the `XYPlot` visualization (see `launcherTwoNeuronSimulator` for an example of this).

Parameters

- `tau` - time constant of the node dynamics
- `h` - resting level of node activation
- `beta` - steepness of sigmoid (logistic) output function
- `selfExcitation` - strength of self-excitation
- `noiseLevel` - strength of random Gaussian noise in node activation
- `range` - two-element vector specifying the range of activation values for which the rate of change is computed
- `resolution` - resolution (or actually, step size) with which the given range is sampled for computation of rate of change values

Components

- `input` - sum of all external inputs to the node
- `activation` - current activation value
- `output` - current output of the node (sigmoid of the activation)
- `h` - resting level of node activation
- `rateOfChange` - rate of change for the node's activation in the current step
- `samplingPoints` - vector of activation values (specified by parameters `range` and `resolution`) for which the rate of change is computed
- `sampledRatesOfChange` - vector of rates of change, computed at each sampling point (given the current external input to the node)
- `attractorStates` - vector of approximate activation values within `range` at which attractor states exist (vector may be empty and change size as parameters or inputs change)

- `attractorRatesOfChange` - vector of rates of change at the attractor states (same size as `attractorStates`, all values should be close to zero, provided for plotting attractors)
- `repellorStates` - vector of approximate activation values within `range` at which repellor states exist (vector may be empty and change size as parameters or inputs change)
- `repellorRatesOfChange` - vector of rates of change at the repellor states (same size as `repellorStates`, all values should be close to zero, provided for plotting attractors)

Inputs The `SingleNodeDynamics` can have an arbitrary number of scalar inputs.

5.2 Interaction Kernels

5.2.1 GaussKernel1D

Description Convolves an input with a Gaussian interaction kernel along one dimension (horizontally). Typically this element is used for one-dimensional inputs, but it may also be used with two-dimensional inputs if convolution only along the horizontal dimension is desired (otherwise use GaussKernel2D).

Parameters

- `size` - size of the input and output of the element
- `sigma` - width parameter of the Gaussian kernel
- `amplitude` - amplitude of the Gaussian kernel
- `circular` - flag indicating whether the convolution is performed in a circular fashion (i.e., values at the left end of the input can affect the output at the right end and vice versa); default value is true
- `normalized` - flag indicating whether the Gaussian kernel is normalized before scaling it with the amplitude (for a normalized kernel, the amplitude equals the integral over the kernel; without normalization, the amplitude equals the value of the kernel at the center); default value is true
- `cutoffFactor` - sets the cutoff factor for the kernel (to make computation faster, the kernel function is cut off at a certain multiple of sigma, ignoring those parts of the kernel where interactions strengths are very small); advised range is 3 to 5 (5 being the default); set to `inf` to always use full kernel

Components

- `output` - result of the input's convolution with the interaction kernel
- `kernel` - interaction kernel (incorporating the `amplitude` parameter)

Inputs The `GaussKernel1D` must have exactly one input whose size matches the element's `size` parameter.

5.2.2 GaussKernel2D

Description Convolves a two-dimensional input with a two-dimensional Gaussian interaction kernel. The convolution is internally performed separately along the two dimensions for computational efficiency.

Parameters

- `size` - size of the input and output of the element
- `sigmaY` - width parameter of the Gaussian kernel for the vertical axis
- `sigmaX` - width parameter of the Gaussian kernel for the horizontal axis
- `amplitude` - amplitude of the Gaussian kernel
- `circularY` - flag indicating whether the vertical convolution is performed in a circular fashion (i.e., values at the left end of the input can affect the output at the right end and vice versa); default value is true
- `circularX` - flag indicating whether the horizontal convolution is performed in a circular fashion; default value is true
- `normalized` - flag indicating whether the Gaussian kernel is normalized before scaling it with the amplitude; default value is true
- `cutoffFactor` - sets the cutoff factor for the kernel (see `GaussKernel1D`); default value is 5

Components

- `output` - the result of the input's convolution with the interaction kernel
- `kernelX` - the horizontal component of the interaction kernel (incorporating the `amplitude` parameter)
- `kernelY` - the vertical component of the interaction kernel (incorporating the `amplitude` parameter)

Inputs The `GaussKernel2D` must have exactly one input whose size matches the element's `size` parameter.

5.2.3 MexicanHatKernel1D

Description Convolves the input with a sum of two Gaussian interaction kernels along one dimension (horizontally). The element can be used in particular to implement an interaction pattern of local excitation and surround inhibition ("mexican hat"). Typically this element is used for one-dimensional inputs, but it may also be used with two-dimensional inputs if convolution only along the horizontal dimension is desired (otherwise use two separate instances of GaussKernel2D).

Parameters

- `size` - size of the input and output of the element
- `sigmaExc` - width parameter of the excitatory kernel component
- `amplitudeExc` - amplitude of the excitatory kernel component
- `sigmaInh` - width parameter of the inhibitory kernel component
- `amplitudeInh` - amplitude of the inhibitory kernel component (the inhibitory component is subtracted from the excitatory one, so a positive value of `amplitudeInh` corresponds to actual inhibition)
- `normalized` - flag indicating whether each kernel component is normalized before scaling it with the amplitude; default value is true
- `cutoffFactor` - sets the cutoff factor for the kernel (see `GaussKernel1D`); default value is 5

Inputs The `MexicanHatKernel1D` must have exactly one input whose size matches the element's `size` parameter.

5.2.4 LateralInteractions1D

Description Combines a Mexican-hat style convolution kernel with a global interaction component (in which the sum of the input is computed, scaled with a constant factor and globally added to the output). This class offers a compact form to add the full lateral interactions typically used in DNFs to a model. If no global interactions are needed, `MexicanHatKernel1D` should be used instead.

Parameters

- `size` - size of the input and output of the element
- `sigmaExc` - width parameter of the excitatory kernel component
- `amplitudeExc` - amplitude of the excitatory kernel component
- `sigmaInh` - width parameter of the inhibitory kernel component
- `amplitudeInh` - amplitude of the inhibitory kernel component (the inhibitory component is subtracted from the excitatory one, so a positive value of `amplitudeInh` corresponds to actual inhibition)
- `amplitudeGlobal` - amplitude of the global kernel component (a negative value must be given to create global inhibition)
- `normalized` - flag indicating whether each local kernel component is normalized before scaling it with the amplitude
- `cutoffFactor` - sets the cutoff factor for the kernel (see `GaussKernel1D`); default value is 5

Components

- `output` - result of the input's convolution with the interaction kernel
- `kernel` - local interaction kernel (sum of scaled excitatory and inhibitory components)
- `amplitudeGlobal` - global interaction weight (scalar value)

Inputs The `LateralInteractions1D` element must have exactly one input whose size matches the element's `size` parameter.

5.2.5 WeightMatrix

Description Connective element that computes its output O by multiplying its input I (a row vector) with a weight matrix:

$$O = I \cdot W \tag{5}$$

The weight matrix should be specified in the constructor call and determines the size of the expected input and the produced output. For and input of size $[1, N]$ and an desired output of size $[1, M]$, the weight matrix must have the size $[N, M]$.

Parameters

- `size` - size of the output
- `weights` - weight matrix

Components

- `output` - result of the matrix multiplication (a row vector)

Inputs The `WeightMatrix` element must have exactly one input, a row vector whose length matches the number of rows in the weight matix.

5.3 Dimensional Reduction and Expansion

5.3.1 SumDimension

Description Computes the sum over one or two dimensions of the input.

Parameters

- `sumDimensions` - dimensions over which the sum is formed (1 for vertical, 2 for horizontal, [1, 2] for both)
- `size` - size of the element's output
- `amplitude` - scalar value that is multiplied with the formed sum

Components

- `output` - result of the summation and scaling; if the result of the summing operation is a one-dimensional vector, its shape (row or column vector) is controlled by the parameter `size`.

Inputs The `SumDimension` element must have exactly one input. After summing along the specified dimensions, its size must match the element's `size` parameter.

5.3.2 SumAllDimensions

Description Computes separate sums for the horizontal and vertical dimension as well as the full sum (over both dimensions) for a two-dimensional input.

Parameters

- `size` - size of the element's input (from which the output sizes are derived)

Components

- `horizontalSum` - sum of the input over the horizontal (second) dimension, transposed to yield a row vector (with size $[1, n]$ for an input of size $[n, m]$)
- `verticalSum` - sum of the input over the vertical (second) dimension (with size $[1, m]$ for an input of size $[n, m]$)
- `fullSum` - sum of the input over both dimensions (yielding a scalar value)

Inputs The `SumAllDimensions` element must have exactly one input whose size matches the element's `size` parameter.

5.3.3 ExpandDimension2D

Description Expands a one-dimensional input along a specified axis to form a two-dimensional array.

Parameters

- `expandDimension` - dimension along which the input is to be expanded (1 for vertical, 2 for horizontal; the input is always transposed in such a way that it runs along the dimension that is expanded)
- `size` - size of the element's output (after expansion)

Components

- `output` - result of the expansion of the input

Inputs The `ExpandDimension2D` element must have exactly one input. After expanding along the specified dimension, its size must match the element's `size` parameter. The orientation of the input is not relevant.

5.3.4 DiagonalSum

Description Computes sum of a square array along the second diagonal.

Parameters

- `inputSize` - size of the two-dimensional input (can be either a scalar value or a two-element vector with equal entries)
- `amplitude` - scalar value that is multiplied with the formed sum

Components

- `output` - the scaled diagonal sum of the input as a row vector (for an input array of size $[n, n]$, the size of the output is $[1, 2n - 1]$)

Inputs The `DiagonalSum` element must have exactly one input, a square two-dimensional array whose size matches the element's `size` parameter.

5.3.5 DiagonalExpansion

Description Expands a one dimensional input diagonally into a square matrix. The input must have an odd number of columns, such that the center of the input is expanded along the main diagonal of the resulting square matrix.

Parameters

- `inputSize` - size of the one-dimensional input (must be row vector and the number of columns must be odd)
- `amplitude` - scalar value that is multiplied with the input before expansion

Components

- `output` - the scaled square matrix resulting from the expansion (for an input vector of size $[1, 2n - 1]$, the output has size $[n, n]$)

5.3.6 ScalarToGaussian

Description Creates a one-dimensional Gaussian output pattern centered on a scalar input value. The input value can be scaled and an offset added to map the value range of the scalar to the desired range in the output (the full output has the range `[1, size(2)]`). Note: The difference to the element `GaussStimulus1D` is that `ScalarToGaussian` receives an input and is updated in every step, whereas in `GaussStimulus1D` the position of the Gaussian is determined by a parameter and the output is only updated if that parameter is changed.

Parameters

- `size` - size of the output
- `inputScale` - scale for the input value; the center of the Gaussian is determined as $position = inputScale \cdot input + inputOffset$
- `inputOffset` - constant offset for the input value, see above
- `sigma` - width parameter of the Gaussian

- `amplitude` - strength of the Gaussian
- `circular` - flag indicating whether the output is defined in a circular fashion (i.e., if the Gaussian is centered near the left end of the output, it can flow over into the right end and vice versa); default value is true
- `normalized` - flag indicating whether the Gaussian output is normalized before scaling it with the amplitude (for a normalized Gaussian, the integral is one if the amplitude is one; without normalization, the value at the center is one if the amplitude is one); default value is false

Components

- `output` - the Gaussian pattern
- `position` - the center of the Gaussian (the scaled input with offset added)

Inputs The `ScalarToGaussian` element takes exactly one input, a scalar value that determines the center of the Gaussian.

5.4 Basic Mathematical Operations

5.4.1 ScaleInput

Description Scales an input with a constant factor.

Parameters

- `size` - size of the input and output of the element
- `amplitude` - scalar value with which the input is scaled

Components

- `output` - result of the scaling (input multiplied with `amplitude` parameter)

Inputs The `ScaleInput` element must have exactly one input whose size matches the element's `size` parameter.

5.4.2 SumInputs

Description Computes the sum of multiple inputs.

Parameters

- `size` - size non-scalar inputs and output of the element

Components

- `output` - sum of all inputs

Inputs The `SumInputs` element may have an arbitrary number of inputs, which have to be either scalar or whose size has to match the element's `size` parameter.

5.4.3 ShiftInput

Description Shifts an input array by a specified value along the horizontal and/or vertical axis.

Parameters

- `size` - size of the input and output of the element
- `shiftValue` - a two-element integer vector specifying the value by which the input is shifted along the first (vertical) and second (horizontal) dimension
- `amplitude` - scalar value that is multiplied with the shifted input
- `circular` - flag indicating whether the shift should be performed circularly
- `fillValue` - value with which those parts of the output array are filled that are not occupied by the shifted input (for non-circular shifts)

Components

- `output` - the result of shift and scaling operations

Inputs The `ShiftInput` element must have exactly one input whose size matches the element's `size` parameter.

5.4.4 PointwiseProduct

Description Performs a pointwise multiplication between two inputs of equal size (with arbitrary dimensionality).

Parameters

- `size` - size of the two inputs and the output

Components

- `output` - pointwise product of the inputs

Inputs The `PointwiseProduct` element takes exactly two inputs, whose size must match the element's `size` parameter.

5.4.5 Convolution

Description Performs a convolution between two one- or two-dimensional inputs. Either input can be flipped horizontally and vertically (i.e. rotated by 180°) to effectively perform a correlation.

Parameters

- `size` - size of the result of the convolution operation
- `flipInputs` - integer value coding which of the inputs should be flipped; in the constructor, two separate boolean arguments are required to indicate separately whether the first and/or the second input should be flipped before the convolution
- `shape` - shape of the convolution, as in the Matlab function `conv2`; can be 'full', 'same', or 'valid'

Components

- `output` - the result of the convolution

Inputs The `Convolution` element takes exactly two inputs, and the size of the matrix resulting from the convolution between them must match the `size` parameter of the element.

5.4.6 Interpolation1D

Description Computes the output by interpolating from the input at specified positions, using the Matlab function `interp1`.

Parameters

- `size` - size of the output of the element
- `interpolationPoints` - vector containing the positions at which the input vector should be interpolated
- `method` - interpolation method (see Matlab documentation on `interp1` for further information)
- `extrapValue` - value with which the input is padded for extrapolation (if interpolation points are outside of the range $[1, n]$, n being the length of the input)

Components

- `output` - the result of the interpolation

Inputs The `Interpolation1D` element must have exactly one one-dimensional input.

5.5 Output Functions

5.5.1 Sigmoid

Description Computes the sigmoid (logistic function) of an input of arbitrary size and dimensionality. The output O is computed from the input I at every position as

$$O(x) = \frac{1}{1 + \exp(\beta(I(x) - \theta))} \tag{6}$$

with threshold θ and steepness parameter β. (Note: A logistic output function with threshold 0 is also implemented in the `NeuralField` class, so no separate element is required to compute the field output.)

Parameters

- `size` - size of the input and output of the element
- `beta` - steepness parameter of the logistic function
- `theta` - threshold for the logistic function

Components

- `output` - the sigmoid of the input

Inputs The `Sigmoid` element must have exactly one input whose size matches the element's `size` parameter.

5.5.2 HalfWaveRectification

Description Element that applies a positive half-wave rectification to the input x:

$$O(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{else} \end{cases} \tag{7}$$

This can be used as an alternative, non-saturating output function for neural fields or nodes.

Parameters

- `size` - size of the input and output of the element

Components

- `output` - the half-wave rectified input

Inputs The `HalfWaveRectification` element takes exactly one input whose size matches the element's `size` parameter.

5.6 Stimuli

Stimuli do not take any inputs from other elements.

5.6.1 BoostStimulus

Description A scalar stimulus that can be used as a homogenous boost of a neural field.

Parameters

- `amplitude` - value of the scalar stimulus

Components

- `output` - that same scalar value

5.6.2 GaussStimulus1D

Description A one-dimensional Gaussian.

Parameters

- `size` - size of the output
- `sigma` - width parameter of the Gaussian function
- `amplitude` - strength of the Gaussian stimulus
- `position` - position of the stimulus center
- `circular` - flag indicating whether the stimulus is defined in a circular fashion (i.e., if the stimulus is centered near the left end of the output, it can flow over into the right end and vice versa); default value is true
- `normalized` - flag indicating whether the Gaussian stimulus is normalized before scaling it with the amplitude (for a normalized stimulus, the integral is one if the amplitude is one; without normalization, the value at the center is one if the amplitude is one); default value is false

Components

- `output` - the Gaussian stimulus

5.6.3 GaussStimulus2D

Description A two-dimensional Gaussian.

Parameters

- `size` - size of the output
- `sigmaY` - width parameter of the Gaussian function along the vertical axis
- `sigmaX` - width parameter of the Gaussian function along the horizontal axis
- `amplitude` - strength of the Gaussian stimulus
- `positionY` - vertical position of the stimulus center
- `positionX` - horizontal position of the stimulus center
- `circularY` - flag indicating whether the stimulus is defined in a circular fashion in the vertical dimension (i.e., if the stimulus is center near the top end of the output, it can flow over into the bottom end and vice versa); default value is true
- `circularX` - flag indicating whether the stimulus is defined in a circular fashion in the horizontal dimension; default value is true
- `normalized` - flag indicating whether the Gaussian stimulus is normalized before scaling it with the amplitude (for a normalized stimulus, the integral is one if the amplitude is one; without normalization, the value at the center is one if the amplitude is one); default value is false

Components

- `output` - the Gaussian stimulus

5.6.4 CustomStimulus

Description A freely defined stimulus pattern of arbitrary dimensionality. The full stimulus pattern is specified directly as a parameter. (Note that this can lead to large parameter files as the whole pattern needs to be stored.)

Parameters

- `size` - size of the output
- `pattern` - the full stimulus pattern (matching the `size` parameter)

Components

- `output` - the custom stimulus pattern

5.6.5 NormalNoise

Description Random noise drawn from a normal distribution in every time step.

Parameters

- `size` - size of the generated output
- `amplitude` - scalar value with which the noise is scaled

Components

- `output` - the generated matrix of scaled random values

5.7 History

5.7.1 History

Description Element that stores its input at specified times. A vector of simulation times $[t_1, ..., t_K]$ must be specified. The input to the element at those times is then stored in a $K \times N$ matrix if the input is a vector of size N, or in a $N \times M \times K$ matrix if the input is a matrix of size $N \times M$. (Note: No input is stored if no Euler step occurs exactly at a specified time t_i, e.g. if $t_i = 31.5$ and the step size of the simulation is 1, with steps occurring at $t = 31$ and $t = 32$.)

Parameters

- `size` - size of input (at one time step)
- `storingTimes` - vector of simulation times at which the input is stored

Components

- `output` - matrix of stored inputs (filled with NaN before inputs are stored)

Inputs The `History` element must have exactly one input whose size matches the element's `size` parameter.

5.7.2 RunningHistory

Description A continuously updated history of the input to this element over the recent time steps. At specified intervals, the input to this element is stored into a matrix (same format as in `History`) as first row/layer, while the content of all other rows/layers get pushed back. The oldest stored inputs are lost once the history matrix is full.

Parameters

- `size` - size of input (at one time step)
- `timeSlots` - number of time steps that are stored
- `interval` - interval between storing times (in simulation time; a new input is stored if simulation time modulo interval is zero)

Components

- `output` - matrix of stored inputs (filled with NaN before inputs are stored)

Inputs The `RunningHistory` element must have exactly one input whose size matches the element's `size` parameter.

5.8 Image Acquisition and Processing

5.8.1 CameraGrabber

Description Retrieves a new image from a connected camera in every step. Note: This class is specific for the cameras used at the INI and requires additional mex files to provide the actual link to the hardware.

Parameters

- `device` - device number for the connected camera (typically 0 or 1)
- `size` - size of the produced image; the image taken from the camera is resized if the sizes do not match

Components

- `image` - RGB image of size `[size(1), size(2), 3]`

5.8.2 ImageLoader

Description Loads images from file, allows switching between different images.

Parameters

- `fileNames` - cell array of file names; in the constructor a separate string `filePath` can be provided as common path to the image files
- `size` - size of output image; loaded images are resized if their size does not match this value
- `currentSelection` - index of the currently selected image file

Components

- `image` - RGB image of size `[size(1), size(2), 3]`

5.8.3 ColorExtraction

Description Extracts regions of salient color information from an RGB image. The steps of the color extraction are as follows:

- the region of interest is horizontally resized to match the specified horizontal output size (the vertical size of the region of interest remains unchanged), and transformed into HSV color space
- pixels are determined that exceed a specified saturation and value threshold, and classified according to their hue value into a number discrete colors
- the pixels are summed up vertically to determine for each horizontal position the number of salient pixels of each color

The result is a matrix of size `[nColors, XN]`, where each row yields the distribution of salient color pixels over the horizontal axis of the region of interest.

Parameters

- `roi` - region of interest in the image, within which regions of salient color are extracted; in the constructor, this region is defined by two arguments `imageRangeX` and `imageRangeY`, giving the horizontal and vertical extent of the roi as a two-element vector in pixels of the input image
- `size` - size of the produced, with `size = [nColors, NX]`
- `hueToIndexMap` - matrix specifying which ranges of hue values are mapped onto which discrete colors; the matrix has the form

  ```
  [hueMin_1, hueMax_1, colorIndex_1; ...
   hueMin_2, hueMax_2, colorIndex_2; ...]
  ```

  where each `[hueMin, hueMax]` pair defines a range in hue values, and `colorIndex` is the row in the output matrix to which this color range is counted; multiple ranges of hue values may be mapped to a single `colorIndex`; the total range of hue values is [0, 1]
- `saturationThreshold` - saturation threshold in HSV color space for a pixel to be counted toward any of the colors; the total range of saturation values is [0, 1]
- `valueThreshold` - value threshold in HSV color space a for pixel to be counted toward any of the colors; the total range of value values is [0, 1]

Components

- `output` - matrix of salient color distributions

Inputs The `ColorExtraction` element takes as input one RGB image (matrix of size [M, N, 3]). The specified `roi` has to be within the bounds of this image.

5.9 Motor control

5.9.1 AttractorDynamics

Description Forms a one-dimensional dynamical system from a space-coded input and determines the rate of change for a given state of the system. The dynamical system is defined by scaling shifted sigmoid functions with the space-coded input, producing attractors in regions of high input.

Parameters

- `size` - size of the one-dimensional space-coded input
- `amplitude` - amplitude or scaling factor for the output

Components

- `phiDot` - rate of change for the given state of the dynamical system
- `phiDotAll` - vector of the same size of the input, giving the rate of change for all possible states of the dynamical system

Inputs The `AttractorDynamics` element takes exactly two inputs: The first is a one-dimensional space code whose size matches the element's `size` parameter, the second is a scalar value that specifies the current state of the dynamical system.

5.9.2 DynamicRobotController

Description Connects the DNF architecture to an E-Puck robot (requires additional files for the interface) and allows dynamic control of the robot orientation by specifying its rate of change.

Parameters

- `minWheelVelocity` - minimal velocity for both wheels (velocity is set to zero if absolute value of derived from the orientation dynamics is lower than this value); avoids poor odometry due to erratic robot movements
- `maxWheelVelocity` - maximum velocity for both wheels

Components

- `position` - robot position relative to its starting position (determined from odometry)
- `orientation` - robot orientation, given in the range $[-\pi, \pi)$

Inputs The `DynamicRobotController` takes exactly one input that defines the rate of change for the robot orientation (in rad per second).

6 Controls

6.1 ParameterSlider

The `ParameterSlider` control creates a slider with an accompanying text field in the GUI. The slider is connected to one or more parameters (belonging to a single element or different elements). The parameter value is changed whenever the slider is moved. The range of parameter values covered by the slider as well as a scaling factor for the conversion from slider position to parameter value can be specified.

The control handle is obtained via the constructor call

```
ParameterSlider(controlLabel, elementLabels, parameterNames, ...
  sliderRange, valueFormat, scalingFactor, toolTip, position)
```

with the following arguments:

- `controlLabel` - label for the control displayed in the text field next to the slider
- `elementLabels` - string or cell array of strings specifying the labels of elements controlled by this slider
- `parameterNames` - string or cell array of strings specifying the names of the element parameters controlled by this slider; arguments `elementLabels` and `parameterNames` must have the same size, with each pair of entries fully specifying one controlled parameter
- `sliderRange` - two-element vector giving the range of the slider
- `valueFormat` - string specifying the format of the parameter value displayed next to the slider (optional, see the Matlab documentation of the `fprintf` function on construction of that string)
- `scalingFactor` - scalar value specifying a conversion factor from the element's parameter value to the slider position (optional)
- `tooltip` - tooltip displayed when hovering over the control with the mouse (optional)
- `position` - position of the control in the GUI figure window in relative coordinates (optional, is overwritten when specifying a grid position in the GUI's `addControl` function)

The slider position is initialized to reflect the value of the first connected parameter and is adjusted when this parameter value is changed via any other control (within the range of the slider).

Example

```
h = ParameterSlider('h_u', 'field u', 'h', [-10, 0], '%0.1f', 1,
  'resting level of field u');
```

6.2 ParameterSwitchButton

The `ParameterSwitchButton` control creates a labeled toggle button (i.e. the button toggles between pressed and not pressed when clicked). This control can switch the values of one or more parameter between two predefined values. Note: The state of the button is not adjusted when the controlled parameters are changed via another control or the parameter panel. The button's state does therefore not necessarily reflect the current parameter values.

The control handle is obtained via the constructor call

```
ParameterSwitchButton(controlLabel, elementLabels, parameterNames,
  offValues, onValues, toolTip, pressedOnInit, position)
```

with the following unique arguments (see above for a description of the other arguments):

- `offValues` - scalar or vector specifying for every connected element parameter the value it should take while the button is not pressed
- `onValues` - scalar or vector specifying for every connected element parameter the value it should take while the button is pressed
- `pressedOnInit` - specifies whether the button should be in the pressed or not pressed state on initialization of the GUI (default is false)

Example

```
h = ParameterSlider('h_u', 'field u', 'h', [-10, 0], '%0.1f', 1,
  'resting level of field u');
```

6.3 ParameterDropdownSelector

The `ParameterDropdownSelector` control creates a dropdown menu with an accompanying text field in the GUI. The menu allows to change the values of one or more parameters between a number of presets.

The control handle is obtained via the constructor call

```
ParameterDropdownSelector(controlLabel, elementLabels, ...
  parameterNames, dropdownValues, dropdownLabels, ...
  initialSelection, toolTip, position)
```

with the following unique arguments (see above for a description of the other arguments):

- `dropdownValues` - a numerical array or a cell array of numerical arrays specifying the parameter values associated with each menu entry; if the control is connected to a single element parameter, this argument should be an array with one valid parameter value for each item in the dropdown menu; if multiple parameters are connected, it should be a cell array of such arrays
- `dropdownLabels` - cell array of strings specifying the menu items in the dropdown menu (optional, if not specified the `dropdownValues` for the first connected parameter are used as labels)
- `initialSelection` - integer specifying the initial selection in the dropdown menu (optional, default is 1)

Examples

```
h = ParameterDropdownSelector('p_sA', 'stimulus A', 'position', ...
  [25, 50, 75], {'left', 'center', 'right'}, 2, ...
  'position of stimulus A');
h = ParameterDropdownSelector('d_s', {'stimulus A', 'stimulus B'},...
  {'position', 'position'}, {[40, 45, 48], [60, 55, 52]}, ...
  {'far', 'close', 'very close'}, 1, 'distance between stimuli A and B');
```

6.4 GlobalControlButton

The `GlobalControlButton` control creates a button in the GUI that connects to a property of any specified object, typically the GUI itself. For the `StandardGUI`, it can connect to one of the following boolean properties: `pauseSimulation`, `quitSimulation`, `resetSimulation`, `saveParameters`, `loadParameters`, and `paramPanelRequest`. The values of these flags are checked in every cycle of the GUI and appropriate operations will be performed.

The control handle is obtained via the constructor call

```
GlobalControlButton(controlLabel, controlledObject, ...
  propertyName, onValue, offValue, resetAfterPress, ...
  toolTip, position)
```

with the following unique arguments (see above for a description of the other arguments):

- `controlledObject` - handle to the controlled object
- `propertyName` - property name that is controlled by the button
- `onValue` - value that the controlled property should have while the button is pressed
- `offValue` - value that the controlled property should have while the button is not pressed
- `resetAfterPress` - determines the behavior of the button: for `false` it acts as a toggle button, for `true` as a push button; in the latter case, the controlled property will always be set to the `onValue` when the button is clicked

Examples

```
h = addControl(GlobalControlButton('Pause', gui, 'pauseSimulation', ...
  true, false, false, 'pause simulation');
```

6.5 PresetSelector

The `PresetSelector` control loads full parameter settings for the model from one of a set of predefined parameter files. The control consists of a dropdown menu to select a parameter file (which may be listed with a descriptive label) and a confirmation button. When the button is pressed, the parameter file connected to the currently selected entry in the dropdown menu is loaded.

Note: Loading from a parameter file will re-initialize the simulation. The control handle is obtained via the constructor call

```
PresetSelector(controlLabel, controlledObject, filePath, presetFiles,
  presetLabels, toolTip, position)
```

with the following unique arguments (see above for a description of the other arguments):

- `controlledObject` - the object that performs the loading operation, which is always the GUI that the control is part of
- `filePath` - string specifying a common relative or absolute path for all parameter files (may be empty if files are located in different folders)
- `presetFiles` - cell array of strings containing the file names (or complete paths) for the parameter files
- `presetLabels` - cell array of strings containing a label for each parameter file to be shown in the dropdown menu (optional, by default the filenames are used as labels)

Example

```
h = PresetSelector('Select', gui, 'presetsOneLayerField/', ...
  {'preset_stabilized.json', 'preset_selection.json', 'preset_memory.json'}, ...
  {'stabilized', 'selection', 'memory'}, ...
  'Load pre-defined parameter settings');
```

7 Visualizations

7.1 MultiPlot

The `MultiPlot` visualization creates a set of axes with one or more plots in it, oriented either vertically or horizontally.

The visualization handle is obtained via the constructor call

```
MultiPlot(plotElements, plotComponents, scales, orientation, ...
  axesProperties, plotProperties, title, xlabel, ylabel, position)
```

with the following arguments:

- `plotElements` - string or cell array of strings (with one entry for each plot) listing the labels of the elements whose components should be plotted
- `plotComponents` - string or cell array of strings (with one entry for each plot) listing the component names that should be plotted for the specified elements; one pair of entries from `plotElements` and `plotComponents` fully specifies the source data for one plot
- `scales` - scalar or numeric vector specifying a scaling factor for each plot (optional, by default all scaling factors are 1)
- `orientation` - string specifying the orientation of the plot, should be either `horizontal` (default) or `vertical`; for horizontal plots, the component is used as `YData` of the plot, for vertical plots it is used as `XData`; the data for the respective other axis is fixed and can be set in the `plotProperties` argument
- `axesProperties` - cell array containing a list of valid axes settings (as property/value pairs) that can be applied to the axes handle via the `set` function (optional, see Matlab documentation on `axes` for further information)

- `plotProperties` - cell array of cell arrays containing lists of valid lineseries settings (as property/value pairs or as a single string specifying the line style) that can be applied to the plot handles via the `set` function (see Matlab documentation on the `plot` function for further information); the outer cell array must contain one inner cell array for every plot (optional)
- `title` - string specifying an axes title (optional)
- `xlabel` - string specifying an x-axis label (optional)
- `ylabel` - string specifying a y-axis label (optional)
- `position` - position of the control in the GUI figure window in relative coordinates (optional, is overwritten when specifying a grid position in the GUI's `addVisualization` function)

It is also possible to create a `MultiPlot` object without specifying what is to be plotted (giving empty matricies for arguments in the constructor call that refer to individual plots), and then add plots individually using the `addPlot` function. However, this can only be done *before* the visualization is added to the GUI. The sequence of function calls then has the following form:

```
hMP = MultiPlot([], [], [], 'horizontal', {...});
hMP.addPlot(...);
hMP.addPlot(...);
gui.addVisualization(hMP, ...);
```

The method call for adding plots has the form

```
hMP.addPlot(plotElement, plotComponent, plotProperties, scale)
```

with arguments

- `plotElement` - label of the element whose component should be plotted
- `plotComponent` - name of the element component that should be plotted
- `plotProperties` - cell array containing a list of valid lineseries settings (as property/value pairs or as a single string specifying the line style) that can be applied to the plot handle via the `set` function (optional, see Matlab documentation on the `plot` function for further information)
- `scale` - scaling factor for the plot

Example

```
h = MultiPlot({'field u', 'field u', 'stimulus A'}, ...
  {'activation', 'output', 'output'}, [1, 10, 1], 'horizontal', ...
  {'YLim', [-10, 10]}, { {'b-', 'LineWidth', 2}, {'r-'}, {'g--'} }, ...
  'perceptual field', 'feature value', 'activation');
```

7.2 XYPlot

The `XYPlot` visualization creates a set of axes with one or more plots in it. The difference to `MultiPlot` is that here, both the `XData` and `YData` of each individual plot can be specified as either the component of some element, or as a fixed vector. This makes it possible to plot two components against each other.

The visualization handle is obtained via the constructor call

```
XYPlot(plotElementsX, plotComponentsOrDataX, ...
  plotElementsY, plotComponentsOrDataY, ...
  axesProperties, plotProperties, title, xlabel, ylabel, position)
```

with the following arguments:

- `plotElementsX` - cell array with one entry for each plot: either the label of the element (as string) whose component should be used as `XData` for the plot, or an empty array `[]` if the `XData` is to be a fixed vector (if only a single plot is created, the cell array may be omitted in this and the following arguments)
- `plotComponentsOrDataX` - cell array with one entry for each plot: either the component name (as string) that should be plotted for the specified element, or the fixed vector that should be used as `XData` for that plot
- `plotElementsY` - cell array specifying the source elements for the plots' `YData`, analogous to the parameter `plotElementsX`
- `plotComponentsOrDataY` - cell array specifying the element components or fixed vectors to be used as `YData`, analogous to the parameter `plotComponentsOrDataY`
- `axesProperties` - cell array containing a list of valid axes settings (as property/value pairs) that can be applied to the axes handle via the `set` function (optional, see Matlab documentation on `axes` for further information)
- `plotProperties` - cell array of cell arrays containing lists of valid lineseries settings (as property/value pairs or as a single string specifying the line style) that can be applied to the plot handles via the `set` function (see Matlab documentation on the `plot` function for further information); the outer cell array must contain one inner cell array for every plot (optional)
- `title` - string specifying an axes title (optional)
- `xlabel` - string specifying an x-axis label (optional)
- `ylabel` - string specifying a y-axis label (optional)
- `position` - position of the control in the GUI figure window in relative coordinates (optional, is overwritten when specifying a grid position in the GUI's `addVisualization` function)

As in `MultiPlot`, it is also possible to first create the visualization without specifying the sources and properties of the individual plots (placing empty matrices for the arguments in the constructor), and then to add plots individually through the function `addPlot`:

```
hXYP = XYPlot({}, {}, {}, {}, axesProperties, {});
hXYP.addPlot(plotElementX, plotComponentOrDataX, ...
  plotElementY, plotComponentOrDataY, plotProperties)
```

Again, plots can only added *before* the visualization object itself is added to the GUI.

Example

```
h = XYPlot('node u', 'activation', 'node v', 'activation', ...
  {'XLim', [-10, 10], 'YLim', [-10, 10], 'Box', 'on'}, ...
  { {'bo', 'MarkerSize', 5} }, ...
  'phase plot', 'activation u', 'activation v');
```

7.3 SlicePlot

The `SlicePlot` visualization plots one-dimensional slices (rows or columns) taken at specified positions from one or several two-dimensional input matries.

The visualization handle is obtained via the constructor call

```
SlicePlot(plotElements, plotComponents, plotSlices, ...
  sliceOrientations, scales, plotOrientation, ...
  axesProperties, plotProperties, ...
  title, xlabel, ylabel, position)
```

with the following unique arguments (see above for a description of the other arguments):

- `plotElements` - string or cell array of strings listing the labels of the elements from which slices should be plotted
- `plotComponents` - string or cell array of strings listing the component names from which slices should be plotted for the specified elements; one pair of entries from `plotElements` and `plotComponents` fully specifies the source data for one set of slice plots `plotSlices` - integer vector or cell of integer vector, specifying for each entry in `plotElements` a set of indices of the rows or columns that should be plotted `sliceOrientations` - string or cell array of strings with each entry either 'horizontal' or 'vertical', specifying the slice orientation (rows or columns) for each entry in `plotElements`
- `scales` - scalar or numeric vector specifying a scaling factor for each set of slices (optional, by default all scaling factors are 1)
- `plotOrientation` - string specifying the orientation of all plots, should be either `horizontal` (default) or `vertical`; for horizontal plots, the slices are used as `YData` of the plot, for vertical plots they are used as `XData`; the data for the respective other axis is fixed and can be set in the `plotProperties` argument

Example

```
h = SlicePlot('field u', 'activation', [25, 50, 75], 'horizontal', ...
   1, 'horizontal', {'YLim', [-10, 10]}, {{'r-'}, {'g-'}, {'b-'}}, ...
  'three slices through field u', 'field position', 'activation');
```

7.4 ScaledImage

The `ScaledImage` visualization plots two-dimensional data using the Matlab `imagesc` function. The visualization handle is obtained via the constructor call

```
ScaledImage(imageElement, imageComponent, imageRange, axesProperties, ...
  imageProperties, title, xlabel, ylabel, position)
```

with the following unique arguments (see above for a description of the other arguments):

- `imageElement` - label of the element whose component should be visualized
- `imageComponent` - name of the element component that should be plotted
- `imageRange` - two-element vector specifying the range of the image's color code
- `imageProperties` - cell array containing a list of valid image object settings (as property/value pairs) that can be applied to the image handle via the `set` function (optional, see Matlab documentation on the `image` function for further information)

Example

```
h = ScaledImage('field u', 'activation', [-10, 10], ...
  {'YDir', 'normal'}, {}, 'perceptual field', 'position', 'color');
```

7.5 RGBImage

The `RGBImage` visualization interprets a `3xMxN` matrix as an RGB image and displays it via the Matlab `image` function. The visualization handle is obtained via the constructor call

```
RGBImage(RGBImage(imageElement, imageComponent, axesProperties, ...
  imageProperties, title, xlabel, ylabel, position))
```

(see above for a description of the arguments).

Example

```
h = RGBImage('camera grabber', 'output', {'YDir', 'normal'}, {}, ...
  'camera image');
```

7.6 SurfacePlot

The `SurfacePlot` visualization diplays two-dimensional data either as a mesh or a surface plot. The visualization handle is obtained via the constructor call

```
SurfacePlot(plotElement, plotComponent, zLim, plotType, ...
  axesProperties, plotProperties, ...
  title, xlabel, ylabel, zlabel, position)
```

with the following unique arguments (see above for a description of the other arguments):

- `zLim` - range of the plot's axes in the z-dimension
- `plotType` - either 'mesh' or 'surface' (default)
- `plotProperties` - cell array containing a list of valid surface object settings (as property/value pairs) that can be applied to the surface handle via the `set` function (optional, see Matlab documentation on the `surface` function for further information)
- `zlabel` - string specifying a z-axis label (optional)

Example

```
h = SurfacePlot('field u', 'activation', [-10, 10], {}, {}, ...
  'perceptual field', 'color', 'position', 'activation');
```

7.7 KernelPlot

The `KernelPlot` visualization is used to plot an interaction kernel. It can combine different kernels and global interaction strengths into a single correctly aligned plot: Local interaction kernels are all centered at zero along the x-axis, capped or padded with zeros as necessary on both sides to match the specified `kernelRange` for the plot, and added up. Global components of interaction kernels are added globally to the plot. The visualization handle is obtained via the constructor call

```
KernelPlot(plotElements, plotComponents, kernelTypes, plotRange, ...
  axesProperties, plotProperties, title, xlabel, ylabel, position)
```

with the following unique arguments (see above for a description of the other arguments):

- `plotElements` - string or cell array of strings specifying the elements that contribute to the plotted interaction kernel
- `plotComponents` - string or cell array of strings (with one entry for each plot) listing the component names for the specified elements; one pair of entries from `plotElements` and `plotComponents` fully specifies one contribution to the plotted interaction kernel
- `kernelTypes` - cell array of strings, with one entry of either `'local'` or `'global'` for each entry in `plotElements`
- `plotRange` - scalar value determining to which range (positively and negatively from zero) the kernel should be plotted
- `plotProperties` - cell array containing a list of valid lineseries settings (as property/value pairs or as a single string specifying the line style) that can be applied to the plot handle via the `set` function (optional, see Matlab documentation on the `plot` function for further information)

Example

```
h = KernelPlot({'u -> u', 'u -> u'}, {'kernel', 'amplitudeGlobal'}, ...
  {'local', 'global'}, 50, {'YLim', [-1, 1]}, {'r-', 'LineWidth', 2}, ...
  'kernel plot', 'distance in feature space', 'interaction weight');
```

Part III
Examples

8 Example A: Building and Running a Simple DNF Architecture

In this section we illustrate how a DNF architecture can be built in a Matlab script and how it can be simulated in the offline mode (without a GUI). You find the complete code for this example in the COSIVINA folder under examples/exampleA.m.

(Note: It is in principle also possible to create a full architecture by directly editing a parameter file in JSON format, and then loading the settings from that file. If you intend to do the latter, we suggest that you study the parameter file of an existing architecture. The JSON format is readable and the storage of the elements in the file largely self-explanatory, but strict adherence to this format is required to be able to load it from Matlab.)

8.1 Creating the Architecture

As a first step, we create an empty `Simulator` object by calling the class constructor. The default settings will be adequate in most cases, so the object (called `sim` throughout the example) can be created by calling

```
sim = Simulator();
```

When we show the properties of the object (by typing its name in the Matlab command line), we will see something like this:

```
sim =
  Simulator handle
  Properties:
        nElements: 0
         elements: {}
    elementLabels: {}
      initialized: 0
           deltaT: 1
            tZero: 0
                t: 0
```

The object does not contain any elements is not initialized.

We can now successively add elements to the `Simulator` object. We will begin with a one-dimensional neural field as first element:

```
sim.addElement(NeuralField('field u', 100, 10, -5, 4));
```

The `addElement` method of the `Simulator` class is used to add the new element to the empty architecture. The call of the constructor `NeuralField` returns a handle to a newly created element with the given parameters. When you want to add an element and are unsure about the required parameters and their order in the constructor call, you can use the command window help for all elements (as well as controls and visualizations), for instance:

```
>> help NeuralField
  NeuralField (COSIVINA toolbox)
    Creates a dynamic neural field (or set of discrete dynamic nodes) of
    arbitrary dimensionality with sigmoid (logistic) output function. The
    field activation is updated according to the Amari equation.

  Constructor call:
  NeuralField(label, size, tau, h, beta)
    label - element label
    size - field size
    tau - time constant (default = 10)
    h - resting level (default = -5)
    beta - steepness of sigmoid output function (default = 4)
```

In our example, we give this element the label `field u`, by which it can later be referenced to set up connections or create visualizations of its components. The next argument to the `NeuralField` constructor specifies its size. The scalar argument `100` is interpreted as specifying a one-dimensional field (which is always stored as a row vector). We could also have given the argument as `[1, 100]` to obtain the same result. The other arguments of the constructor specify further parameters of the field (see the specifications of the this element class above). Some or all of these may be omitted if they match the default values (as they do here) and the actual values are to be set e.g. via the GUI.

Note that we can also perform the above operations in two separate steps:

```
hFieldU = NeuralField('field u', 100, 10, -5, 4);
sim.addElement(hFieldU);
```

Here, we first called the constructor method and obtained an explicit handle `hFieldU`, and then added the element to the simulator via this handle. The handle can then later be used to access the element's properties (but a handle can also be obtained at any time from the `Simulator` object through the `getElement` method). We will in the following examples generally use the more compact form of the function call shown above.

The neural field we have added does not have any lateral interactions, we have to add these as a separate element:

```
sim.addElement( ...
  LateralInteractions1D('u -> u', 100, 4, 15, 10, 15, 0), ...
  'field u', 'output', 'field u', 'output');
```

Again, we have created the element itself by calling its constructor. We give the new element the descriptive label `u -> u`. Since this element is to be connected to `field u`, their sizes must match (several element types exist to couple elements of different sizes or dimensionality if necessary). This is specified in the second argument to the constructor. The following arguments determine the interaction strengths and widths. The element handle returned from the constructor is the first argument to the function `addElement`. Here, additional arguments are added to set up the connectivity of the new element. These arguments specify, in the order they are given: The new element should receive input from `field u` (2nd argument), namely the component `output` of this element (3rd); and it should itself project back to the element `field u` (4th), which receives the component `output` of the new element as input (5th).

Note that the arguments specifying the connectivity are all character strings. Existing elements in the architecture can be addressed via their label (this is why a unique label must be given to every element in an architecture). Components are addressed via their name (as a character string). This matches the name of the property in the element object (for instance, every element of class `NeuralField` has a property `output`).

Since `output` is the name of the default output component of both the `NeuralField` and the `LateralInteractions1D` classes (as well as many others), we can simplify the above call to one of these forms:

```
sim.addElement( ...
  LateralInteractions1D('u -> u', 100, 4, 15, 10, 15, 0), ...
  'field u', [], 'field u', []);
% or
sim.addElement( ...
  LateralInteractions1D('u -> u', 100, 4, 15, 10, 15, 0), ...
  'field u', [], 'field u');
```

We now add two Gaussian stimuli to the architecture:

```
sim.addElement(GaussStimulus1D('stim A', 100, 5, 6, 25), ...
  [], [], 'field u');
sim.addElement(GaussStimulus1D('stim B', 100, 5, 8, 75), ...
  [], [], 'field u');
```

Stimuli do not receive any inputs, therefore the second and third arguments remain empty. Both stimuli should project to the existing element `field u` in the architecture, so that is the fourth argument. One could make it explicit that `field u` should receive the component `output` of the two stimuli as input, but since this is the default anyway, we omit the optional 5th argument. Note that we have to create two separate objects of the GaussStimulus1D class to add them to the simulator object. It would not be possible, for instance, to create one object of the GaussStimulus1D class and obtain the handle for it outside of the `addElement` function call, and then call the `addElement` function twice with this same object handle.

These are all elements we will use in the first architecture. When we show the properties of the object `sim` again, we will see the following:

```
sim =
  Simulator handle
  Properties:
        nElements: 4
         elements: {[1x1 NeuralField]  [1x1 LateralInteractions1D]
                    [1x1 GaussStimulus1D]  [1x1 GaussStimulus1D]}
    elementLabels: {'field u'  'u -> u'  'stim A'  'stim B'}
      initialized: 0
           deltaT: 1
            tZero: 0
                t: 0
```

Four elements exist, but the object is still not initialized.

8.2 Initializing and Running the Simulation

The simplest way to run the simulation we have set up is to call the `run` method of the `Simulator` class. We will describe this below, but first, we will show how to perform the required operations individually. First, the simulator object has to be initialized. If we have set up a new architecture, it may be helpful to use the `tryInit` method, which will give us detailed information if anything goes wrong during initialization:

```
sim.tryInit();
```

This should run through without any problems in our example, and the simulator object is now initialized. We can continue to check our architecture by performing a single simulation step with the `tryStep()` function:

```
sim.tryStep();
```

If there are any problems in the architecture – such as size mismatches between connected elements – this function will throw an error and inform us about which element caused the problem. It also performs an actual step of the simulation. It is mostly sufficient to perform just one of these trial steps, since most errors should appear immediately when running the simulation. Again, there should be no problems in the example we give here.

If we have used the architecture before (or are confident that everything is correct), we can omit the previous steps and proceed directly to the actual simulation. Still, we have to initialize it first:

```
sim.init();
```

All elements are now initialized, and all the components have been created as matrices of the correct size. Some of these are still filled with zeros (e.g. in the connective elements), some already have meaningful content (true for the dynamic elements and stimuli). For instance, we can plot one of the Gaussian stimuli in the architecture:

```
plot(sim.getComponent('stim A', 'output'));
```

We used the `getComponent` method here to directly access the component `output` of the element labeled `stim A`, and should obtain the plot of a Gaussian curve. In the same way, we can show the activation of the neural field:

```
plot(sim.getComponent('field u', 'activation'));
```

Currently, the newly initialized field is at its resting level everywhere. It will change under the influence of the stimuli when we run the simulation.

We can go through simulation steps manually using the `step` function, for instance in this form:

```
for i = 1 : 10
  sim.step();
end
```

If we plot the neural field activation again in the same way as before, we will see how its activation has increased locally around the centers of the two stimuli. The steps also counted up the internal timer of the simulator object:

```
>> sim.t
ans =
    10
```

If we perform another 10 steps and repeat the plot of the field activation, we can see that supra-threshold activation peaks have formed and observe the effect of the lateral interactions in the form of depressed activation around the peaks. For formal analysis, we can also store these activation patterns at different times during the simulation.

In addition to plotting or storing the state of the system during the simulation, we can also change the properties of elements. For instance, we can now turn off one of the stimuli by setting its `amplitude` parameter to zero. We can do this via a handle for that element that we obtain through the `getElement` method. For the change to take effect, it is necessary in this case to re-initialize the element (because the stimulus is not automatically computed in each step for efficiency reasons).

```
hStimB = sim.getElement('stim B');
hStimB.amplitude = 0;
hStimB.init();
```

We can plot the output of the stimulus again to confirm that it is now flat, and can then continue to run the simulation with further calls of the step function. (To find out which parameter changes require a re-initialization, see Section 2.4.)

We can simplify the code for performing the simulation by utilizing the `run` method of the `Simulator` class. The operations described above (from the initialization on) can then be replaced with the following piece of code:

```
sim.run(10, true);

hStimB = sim.getElement('stim B');
hStimB.amplitude = 0;
hStimB.init();

sim.run(20, false);
```

The first call of `run` initializes the simulator (explicitly requested by setting the second argument to `true`) and runs it until it reaches simulation time `t = 10`. In this case, this is equivalent to the ten steps that we explicitly performed in a loop above. Note, however, that it could be a different number of steps depending on the simulator's `tZero` and `deltaT` parameters. The parameter `tZero` determines the value to which the simulation time is set during initialization. The parameter `deltaT` controls the temporal sampling: For instance, if we set `deltaT = 0.1`, the call `sim.run(10, true)` will perform one hundred steps, but each will only create a smaller change in the field activation. The final state of the system will be qualitatively the same, with some numerical differences.

In the second call of `run` in the piece of code above, we specify that the simulation should now run until it reaches simulation time 20. The second argument `false` indicates that the simulation should not be re-initialized but continue from its previous state. We could also omit this second argument here, since the `run` command does by default not perform an initialization if the simulator is already initialized.

9 Example B: Building an Architecture with Two-Dimensional Fields

In this example we will describe a second simple architecture, now including two-dimensional fields and coupling between fields of different dimensionality. We will describe in detail how lateral interactions are set up in two-dimensional fields and how projections between one-dimensional and two-dimensional fields can be implemented.

9.1 Lateral Interactions in Two-Dimensional Fields

Mathematically, the lateral interactions in two-dimensional field can be described as a straightforward extension of the one-dimensional case: The interaction kernel (typically a Gaussian or difference of Gaussians) is convolved with the field output. However, since this operation is computationally costly, it is highly desirable to optimize it as far as possible. One particular method that we use is *linear separation* of the kernel: If the interaction kernel can be described as a product of two one-dimensional functions, the costly two-dimensional convolution can be replaced by two one-dimensional convolutions. This is possible for Gaussian interaction kernels, but not for difference-of-Gaussian kernels (which cannot directly be described as a product of one-dimensional functions). While in the one-dimensional case lateral interactions with a Mexican-hat shaped kernel can be computed most efficiently by a single convolution, for two dimensions it is more efficient to compute the excitatory and inhibitory part separately. This is reflected in the framework, where only pure Gaussian interaction kernels are provided for the two-dimensional case and Mexican-hat style interactions have to be constructed manually.

We create an architecture and add a two-dimensional field with the following commands:

```
sim = Simulator();
```

```
sim.addElement(NeuralField('field u', [100, 150], 10, -5, 4));
```

The class `NeuralField` is the same as the one used for one-dimensional fields, the dimensionality is determined by the `size` parameter, which is set to `[100, 150]` here. We add two-dimensional Gaussian stimuli (that have the same size as the field and feed into it) so that we can create some localized activation in the field:

```
sim.addElement(GaussStimulus2D('stim u1', [100, 150], 5, 5, 8, 30, 50), ...
  [], [], 'field u');
sim.addElement(GaussStimulus2D('stim u2', [100, 150], 5, 5, 8, 70, 100), ...
  [], [], 'field u');
```

For the lateral interactions, we first add a Gaussian excitatory component:

```
sim.addElement(GaussKernel2D('u -> u (exc)', [100, 150], 5, 5, 20), ...
  'field u', 'output', 'field u', 'output');
```

The parameters of the `GaussKernel2D` constructor specify that it is adjusted for a input of size `[100, 100]`, that the sigma parameter of the Gaussian should be 5 in both dimensions, and that it has an amplitude of 20 (additional optional parameters of the constructor were omitted). The following arguments to the outer function `addElement` specify that the kernel receives input from `field u` and projects back to it, using the component `output` for both projections. The Gaussian inhibitory component of the lateral interactions is added in the same way, but with a negative amplitude:

```
sim.addElement(GaussKernel2D('u -> u (inh)', [100, 150], 10, 10, -20), ...
  'field u', 'output', 'field u', 'output');
```

This kernel has a sigma parameter of 10 for both dimensions and an amplitude of -20. Local projections between two-dimensional fields can be implemented in an analogous fashion using the `GaussKernel2D` class.

If we also wish to include global interactions in the two-dimensional field, there are two element classes available for that: The `sumDimension` class computes the sum over an input along one or more specified dimensions. The `sumAllDimensions` class computes sums over all possible dimensions of a two-dimensional input, yielding three output components: The horizontal sum, the vertical sum, and the full sum (the former two are vectors, the last is a scalar value). Since the computation of sums over large matrices can also be computationally costly, it is advisable to only compute the sums that are actually needed. In this example architecture, we will later use the sum of `field u` along the vertical dimension as input to a one-dimensional field, and we need the sum over both dimensions for global lateral interactions within `field u`. We can most efficiently compute all we need using two `sumDimension` elements: The first one sums over the vertical dimension, the second one computes the sum of the resulting vector over the horizontal dimension:

```
sim.addElement(SumDimension('sum u (vert)', 1, [1, 150], 1.0), ...
  'field u', 'output');
sim.addElement(SumDimension('u -> u (global)', 2, [1, 1], -0.05), ...
  'sum u (vert)', 'output', 'field u', 'output');
```

We call the first new element `sum u (vert)`. The arguments of this constructor specify that the sum should be formed along dimension 1 of the input, that it creates an output of size `[1, 100]`, and that the results should be scaled with a factor of `1.0` (so effectively not be scaled at all). We do not scale the sum at this point because we want to use for

two different interactions, which will both introduce their own scaling. This `SumDimension` element receives input from `field u`, but does for now not project to any other elements.

The second sum is labeled `u -> u (global)`; even though it does not technically form a direct connection from `field u` onto itself, it is effectively providing the global component of the lateral interactions. It computes the sum over the second dimension of its input (which is the output of `sum u (vert)`), with the result being a scalar (size `[1, 1]`). This sum is scaled with a factor of -0.05, specifying the weight of the inhibitory global interactions.

9.2 Projections between Fields of Different Dimensionality

To describe the typical mechanisms for projections between one- and two-dimensional fields, we first add a one-dimensional field to the architecture. We assume that this field is defined over the same space that forms the second (horizontal) dimension of `field u`. We call this field `field w`:

```
sim.addElement(NeuralField('field w', [1, 150], 10, -5, 4));
```

We further include lateral interactions for this field and a Gaussian stimulus:

```
sim.addElement(...
  LateralInteractions1D('w -> w', [1, 150], 5, 15, 12.5, 15, 0, true), ...
  'field w', 'output', 'field w', 'output');
sim.addElement(GaussStimulus1D('stim w1', [1, 150], 5, 3, 50, true), ...
  [], [], 'field w', 'output');
```

We now want to implement a projection from `field u` to `field w`. Since the vertical dimension of `field u` has no correspondence in `field w`, `field u` is summed over this dimension first, yielding a one-dimensional output. We already implemented this above by adding element `sum u (vert)`. The actual projection is now mediated by another Gaussian interaction kernel (reflecting synaptic spread generally found in biological neural systems):

```
sim.addElement(GaussKernel1D('u -> w', [1, 150], 5, 0.5), ...
  'sum u (vert)', 'output', 'field w', 'output');
```

The interaction kernel is one-dimensional, with the size parameter matched to `field w`. This is the case because the output of `field u` is already reduced to one dimension by the summing operation before the interaction kernel is applied. The kernel sigma is set to five, the amplitude of the projection to 0.25. Note that the amplitude for such projections from higher- to lower-dimensional fields should typically be chosen rather small, because the summing operation will yield local amplitudes much larger than the output of a one-dimensional field.

The reverse projection is likewise mediated by a Gaussian interaction kernel. In this case, we begin with the convolution, and then expand the result:

```
sim.addElement(GaussKernel1D('w -> u', [1, 150], 5, 5), ...
  'field w', 'output');
sim.addElement(ExpandDimension2D('expand w -> u', 1, [100, 150]), ...
  'w -> u', 'output', 'field u', 'output');
```

We again use a one-dimensional convolution kernel with size `[1, 150]`, which receives input from `field w`. This convolution does initially not project to any other elements, the projection to `field u` is only added in the next step. In this step, an `ExpandDimension2D` element

is added to scale the result of the convolution up to the size of `field u`. The element, labeled `expand w -> u`, receives as input the `output` component of `w -> u`, and expands it along dimension 1 to a size of `[100, 150]`. To do that, it vertically fills a matrix of the specified size with copies of the input vector. For expansion along the second dimension, the `ExpandDimension2D` would also rotate the input appropriately.

In the script for this example, we then run the simulator and plot the activations at two points in time. The simulator effectively performs a selection decision between two stimuli in `field u`, which is biased by subthreshold input to `field w` through the bidirectional coupling between the fields.

10 Example C: Creating and Using a GUI

The GUIs in this framework are created for a specific architecture, and linked to a specific `Simulator` object. We will in this example create a simple GUI for the small architecture described in Example A. To do so, we first create a `StandardGUI` element, with general GUI settings specified in the constructor. Then we add a visualization of the field activation and several control elements to the GUI.

10.1 Creating the GUI Object

The `StandardGUI` object is created via a constructor call, with arguments specifying global parameters of the GUI. We assume that a `Simulator` object with elements as described in Example A exists in the workspace.

```
gui = StandardGUI(sim, [50, 50, 700, 500], 0.05, ...
  [0.0, 1/3, 1.0, 2/3], [1, 1], 0.1, ...
  [0.0, 0.0, 1.0, 1/3], [5, 3]);
```

The first argument `sim` is a handle to the `Simulator` object that is connected to this GUI. The second argument specifies the figure size and position for the GUI main window on start up (it can be freely resized while it is running). The third argument specifies the duration of the pause that is introduced after every simulation step. The value used here is reasonable for relatively small architectures, where the pause makes it simpler to follow the evolution of the field activation. For larger architectures and field sizes, this value should be set to zero, since the computation time slows down the simulation sufficiently for viewing it (or even more than that).

The arguments in the next line specify the set-up of the visualization grid of the GUI: We want it to occupy the upper two-thirds of the window, and contain only a single grid cell (for a single visualization). The last argument in this line specifies a padding around the visualization. The arguments in the last line analogously specify the settings for the controls grid: In this example, the grid should cover the bottom third of the GUI main window, with 5×3 cells for individual control elements. No padding is provided for the control elements. There are two more optional arguments for the constructor call which allow customization of the parameter panel, but we do not use them in this example.

GUI parameters may be changed after the constructor call. For instance, if we find the pause duration to be inadequate, we can change the behavior of an existing GUI in the following form:

```
gui.pauseDuration = 0.025;
```

Note that changes to the visualization or control grid settings do not affect graphical elements that have already been added.

10.2 Adding Visualizations

The visualizations display one or more element components and are typically updated after each simulation step. In this example, we include only a single visualization, a plot of the field activation, output, and stimuli, using the `MultiPlot` visualization class. We add the visualization with the following command:

```
gui.addVisualization(MultiPlot(...
  {'field u', 'field u', 'stim A', 'stim B'}, ...
  {'activation', 'output', 'output', 'output'}, ...
  [1, 10, 1, 1], 'horizontal', ...
  {'YLim', [-10, 10], 'Box', 'on'}, ...
  {{'b', 'LineWidth', 2}, {'r'}, {'g'}, {'g'}}, ...
  'field u', 'field position', 'activation/ouput/input'), ...
  [1, 1]);
```

The outer method call `gui.addVisualization(...)` is the generic function for adding a visualization element to the GUI, with a constrctor call for the `MultiPlot` object yielding the first argument. This constructor call takes several arguments to fully specify the plots and the axes setup. The first two arguments are cell arrays of strings that together specify *what* is plotted: Each pair of entries from these cells specifies the source for one plot, with the first cell array containing the element labels, the second the component names. Here, we want to plot: The activation of `field u`, the output of `field u`, and the outputs of both `stim A` and `stim B`. The third argument of the constructor specifies the scales for these plots. We scale up the field output tenfold to make changes in this component more visible (since it only ranges from zero to one). The next argument specifies that the plot should be oriented horizontally.

Next, we can specify settings for the axes. These are the same axes properties that can also be provided in the `axes` call in Matlab, given as a list of property/value pairs in a cell array. In this case, we set the limits of the y-axis to the range `[-10, 10]`, and specify that the axes should be enclosed by a box. In a similar way, we can set the properties for each plot in the following argument. Here, we use a cell array of cell arrays, with one inner cell array for each individual plot. Within each inner cell array, we can specify line type and color with a single string (as documented in the Matlab `plot` function), and/or list property/value pairs for the plot. In this example, we want the line for the field activation (the first plot) to be blue and a bit thicker, the other plots to be in red and green, respectively. Finally, in the last three arguments to the constructor call, we specify axes labels and a title for the visualization.

There is then one more argument to the outer function: The vector `[1, 1]` specifies the position of this visualization within the grid. We could also provide another argument here specifying a size of the visualization in the grid. For instance, the GUI in the file `launcherCoupling.m` uses a 4×4 visualizations grid, in which two plots of one-dimensional fields are placed (one horizontal, one vertical), as well as an image for a two-dimensional field. The arrangement of these visualizations, shown in Figure 10.2, is created as follow:

```
gui.addVisualization(MultiPlot(...), [4, 2], [1, 3]);
gui.addVisualization(MultiPlot(...), [1, 1], [3, 1]);
gui.addVisualization(ScaledImage(...), [1, 2], [3, 3]);
```

If one does not want to use the visualizations grid, it is always possible to add an explicit position argument directly to the constructor call of the visualization element instead. The position arguments in the call of the `addVisualization` method should then be omitted.

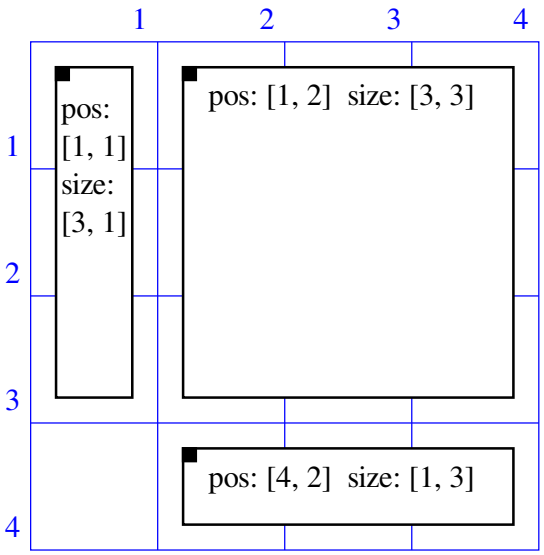

Figure 2: Arrangement plots (for coupled one-dimensional and two-dimensional fields) in a 4×4 visualizations grid.

Since the `MultiPlot` constructor call can become cluttered if a large number of plots is created, an alternative method is provided to add plots successively. The call above can be replaced by the following operations:

```
hMP = MultiPlot([], [], [], 'horizontal', ...
  {'YLim', [-10, 10], 'Box', 'on'}, {}, ...
  'field u', 'field position', 'activation/ouput/input');
hMP.addPlot('field u', 'activation', 1, {'b', 'LineWidth', 2});
hMP.addPlot('field u', 'activation', 1, {'r'});
hMP.addPlot('stim A', 'activation', 1, {'g'});
hMP.addPlot('stim B', 'activation', 1, {'g'});
gui.addVisualization(hMP, [1, 1]);
```

Here, the arguments in the constructor call specifying the individual plots and their sources remain empty, and the plots are then added individually. Note that the visualization element may only be added to a GUI once it is complete.

10.3 Adding Controls to Change Element Parameters

For tuning the model, it is often convenient to change parameters while the simulation is running and directly observe the effects. One way to that is via the parameter panel, which allows access to the elements of all parameters. For settings that are likely to be changed frequently during the online simulation, an even more direct way to access them is desirable. To this end, graphical control elements can be added to the GUI. These controls can be arranged in a control grid analogously to the visualizations grid.

We can add a slider to control the resting level of the neural field in the example architectures using the `addControl` method of the `StandardGUI` class and the controls class `ParameterSlider`:

```
gui.addControl(ParameterSlider('h', 'field u', 'h', [-10, 0],...
  '%0.1f', 1, 'resting level of field u'), ...
  [1, 1]);
```

The constructor of the `ParameterSlider` class takes as first argument a descriptive label that is displayed next to the slider. The label can be chosen freely, and does not need to match any class or parameter name (it should however be relatively short). The second argument is the label of the element accessed by this slider, and the third one the name of the controlled parameter in the element. Here, we want to control the parameter `h` (the resting level) of the element labeled `field u`. The element label must exist in the `Simulator` object linked to the GUI, and the element class must have a parameter of the specified name. The following arguments specify slider range, number format for the display of the parameter value, scaling factor and tool tip (see Controls Reference for details).

We can add sliders for the lateral interaction strengths in the same way. These controls are all connected to different parameters of the element `u -> u`:

```
gui.addControl(ParameterSlider('c_exc', 'u -> u', 'amplitudeExc', ...
  [0, 40], '%0.1f', 1, 'strength of lateral excitation'), [2, 1]);
gui.addControl(ParameterSlider('c_inh', 'u -> u', 'amplitudeInh', ...
  [0, 40], '%0.1f', 1, 'strength of lateral inhibition'), [2, 2]);
gui.addControl(ParameterSlider('c_gi', 'u -> u', 'amplitudeGlobal', ...
  [0, 1], '%0.1f', -1, 'strength of global inhibition'), [3, 1]);
```

The sliders are placed in consecutive locations in the grid via the second argument of the `addControl` method. Note that in the last line, we created a global inhibition slider (labeled `c_gi`) with a scaling factor of -1. This means that when we move that slider to the right (to more positive values), the controlled parameter `amplitudeGlobal` becomes more negative.

We furthermore add controls for the stimulus settings, with each one slider controlling stimulus position and one controlling stimulus strength for each of the elements `stim A` and `stim B`:

```
gui.addControl(ParameterSlider('p_s1', 'stim A', 'position', ...
  [0, 100], '%0.1f', 1, 'position of stimulus 1'), [4, 1]);
gui.addControl(ParameterSlider('c_s1', 'stim A', 'amplitude', ...
  [0, 20], '%0.1f', 1, 'stength of stimulus 1'), [4, 2]);

gui.addControl(ParameterSlider('p_s2', 'stim B', 'position', ...
  [0, 100], '%0.1f', 1, 'position of stimulus 2'), [5, 1]);
gui.addControl(ParameterSlider('c_s2', 'stim B', 'amplitude', ...
  [0, 20], '%0.1f', 1, 'stength of stimulus 2'), [5, 2]);
```

For the position sliders, we adjusted the slider range to the size of field u, such that the stimuli can be centered at any position in the field.

A single slider may also control multiple parameters. For instance, we can set up a slider to adjust both stimulus strengths simultaneously with the following command:

```
gui.addControl(ParameterSlider('c_s', {'stim A', 'stim B'}, ...
  {'amplitude', 'amplitude'}, [0, 20], '%0.1f', 1, ...
  'stength of both stimuli'), [6, 1]);
```

The controlled elements and parameters are here given as cell arrays of strings, with each pair of corresponding strings from the two cell arrays fully specifying one parameter controlled by the slider. We may also want to simply turn stimuli on and off (with a specific strength for the on state). In this case, we can use the `ParameterSwitchButton`. Here, we have to specify the desired values for the controlled parameters for the pressed on non-pressed state of the button:

```
gui.addControl(ParameterSwitchButton('stimuli on', ...
  {'stim A', 'stim B'}, {'amplitude', 'amplitude'}, ...
  [0, 0], [6, 6]), [6, 2]);
```

If the last two control elements are implemented in addition to the individual sliders for the stimulus strength, there is a possibility of conflicts between the controls: For instance, if the stimulus strengths are set to different values by the individual sliders, the slider position of the combined slider cannot reflect this. In the framework, this will not lead to errors in the simulation, but there can be inconsistencies between displayed values on the controls and the actual parameter values. The behavior of the GUI is as follows: The last control element that is activated (button clicked, slider moved, etc.) sets all associated parameters to values specified by this control, other control elements remain unchanged. Similar situations can also occur if parameters are changed in the parameter panel (although sliders are adjusted to reflect new values in this case as far as this is possible).

10.4 Adding Global Controls

Special control elements to globally control the simulation can be added in the same way as the controls to change element parameters. The `StandardGUI` supports the following control mechanism:

- pause the simulation
- reset the simulation (by re-initializing all elements and resetting the simulation time)
- open and close the parameter panel
- save parameters to file
- load parameters from file
- quit the simulation

These functions can be performed by setting control flags in the `StandardGUI` object via a `GlobalControlButton`. The controls for these standard functions can be created in the GUI as follows:

```
gui.addControl(GlobalControlButton('Pause', gui, ...
  'pauseSimulation', true, false, false, 'pause simulation'), ...
  [1, 3]);
gui.addControl(GlobalControlButton('Reset', gui, ...
  'resetSimulation', true, false, true, 'reset simulation'), ...
  [2, 3]);
gui.addControl(GlobalControlButton('Parameters', gui, ...
  'paramPanelRequest', true, false, false, 'open parameter panel'), ...
 [3, 3]);
```

```
gui.addControl(GlobalControlButton('Save', gui, ...
  'saveParameters', true, false, true, 'save parameter settings'), ...
  [4, 3]);
gui.addControl(GlobalControlButton('Load', gui, ...
  'loadParameters', true, false, true, 'load parameter settings'), ...
  [5, 3]);
gui.addControl(GlobalControlButton('Quit', gui, ...
  'quitSimulation', true, false, false, 'quit simulation'), ...
  [6, 3]);
```

The constructor calls for these buttons are generally independent of the type of the GUI and the simulation, and can just be copied and rearranged in the desired fashion for all GUIs.

An additional form of global control element is implemented in the `PresetSelector` class. This control creates a dropdown menu from which one of several prepared parameter settings can be selected. It invokes the same operation as the `Load` button used above, but instead of opening a file selection dialog uses one of a set of predefined files. An example for its use can be seen in the file `launcherField1l_preset`.

10.5 Running a Simulation in the GUI

The simulation can be run within the GUI by calling

```
gui.run();
```

This initializes the GUI, iterates the steps of the associated `Simulator` object, updates the visualizations and manages all interface operations. When the GUI window opens, the simulation immediately and continuously runs: Simulation steps are performed and all elements are continuously updated. This may not be obvious if the model is in a stable state, but it is nonetheless taking place. In the example described here, peaks should evolve in `field u` under the influence of external inputs and then remain stable.

You can now use the sliders to change parameters (these changes take effect immediately) and use the global control buttons to affect the behavior of the GUI: You can pause and unpause the simulation (which will, again, only be obvious while the activation pattern in the field is actually changing), or reset it (which will cause the peaks to form anew). Note: If you want to keep the GUI open in the background but are not currently using it, it is advisable to pause it to reduce CPU usage.

You can open the parameter panel, which gives you access to all elements of the architecture and all of their parameters (although some may not be changed in the GUI). To adjust specific parameters via this panel, select the element in the dropdown menu, then set the parameter value in the edit field (be sure to use Matlab-compatible number formats). The changes take effect once you click the `Apply` button at the bottom. Changes are lost if you close the panel or switch the selected element without clicking `Apply` first.

The current model parameters can be saved or a stored parameter set be loaded via the appropriate buttons. A file selection dialog will open to choose a file to save to or load from (this requires the JSONlab toolbox to be present in the search path). Note that only the parameters are stored or loaded in this procedure, not the full state of the model (e.g. activation patterns of the fields are not saved). Upon loading from a configuration file, the simulation is re-initialized. It is possible to load from a file whose model architecture does not fully match the one of the current simulation (e.g. after changing some part of the architecture). In this case, the model architecture of the current simulation remains

the same, and only the parameters of elements that have a matching counterpart in the configuration file are changed.

By default, the simulation runs in the GUI until the GUI is manually terminated (preferably via the `Quit` button). The simulation time can be limited by supplying an optional first argument in the function call, for instance

```
gui.run(1000);
```

Note that a reset also sets the simulation time back to its initial value, such that it can keep the GUI from terminating.

When the GUI is terminated (either by pressing `Quit` or because it reached its time limit), the `Simulator` object in the workspace will contain the final state of the simulation in the GUI. (Note: This is not guaranteed if the GUI is terminated by other means, like closing the window or pressing `Ctrl-c`, but should still work in most cases). We may at this point for instance inspect individual elements by commands like

```
plot(sim.getComponent('u -> u', 'output'));
```

We can also save the complete `Simulator` object to a `mat` file via the Matlab `save` command:

```
save simulatorFile.mat sim
```

Unlike the configuration files, this file will then preserve the full state of the simulation.

We can now continue the simulation in the GUI at the same point at which is was terminated with the same function call as before:

```
gui.run();
```

The optional second argument of this function allows manual control over re-initialization of the simulation. The call

```
gui.run(inf, true);
```

forces a re-initialization. By default, the simulation is only initialized if it had not been initialized before. (The first argument is set to infinity here to suppress a time limit.)

Finally, it is possible to create a copy of the `Simulator` object and then continue the simulation in the GUI with this copy:

```
sim2 = sim.copy();
gui.run(inf, false, sim2);
```

In the call of the `run` function, we specify the new `Simulator` object as third argument. There are now two independent `Simulator` objects in the workspace, which will likely have different states after `sim2` has been used in the GUI. When the GUI is terminated again, it is possible to go back to the state at which the copy was made by running the GUI once more with the original `Simulator` object:

```
gui.run(inf, false, sim);
```

It is also possible to create more copies, for instance to try out the behavior of the model under different inputs when starting from the same state.

11 General Hints for Efficient DNF Architectures

11.1 Connecting Fields of Different Dimensionality

When connecting a higher-dimensional and lower-dimensional field, all computationally costly operations – such as convolutions – should be performed in the lower dimensional space whenever possible. This means:

- When projecting from a 2D field to a 1D field, one should first compute the sum over the 2D field's output, and then apply the convolution (or other operation) to this sum.
- When projecting from a 1D field to a 2D field, one should first perform the convolution on the output of the 1D field, and then expand the result of this convolution to two dimensions.

11.2 Scaling and Re-using Operations

Many connective elements have an `amplitude` parameters that scales the output of that element (this is the case for all elements that are expected to be used for direct connections between fields). If you need to have the output of an element without `amplitude` parameter scaled, you can do so by feeding it to an element of class `ScaleInput`.

The `ScaleInput` element may also be employed for re-using other elements in multiple projections: Assume for instance that you have a two-dimensional field that should have an excitatory projection to itself and to one other two-dimensional field, both mediated by Gaussian kernels. If the Gaussian for the two projections should have the same width, it is considerably more efficient to perform the convolution only once, and then scale the result twice for the different projections. The amplitude of the Gaussian kernel itself should then be set to 1.0, so that the amplitudes of the `ScaleInput` elements directly reflect the amplitude of the respective projection.

11.3 Order of Elements

The `step` function of the elements in a `Simulator` object is called in the order that the elements were added to the object. This order may affect the exact behavior of the whole dynamical system (although the numerical differences in each step will be small for reasonable step sizes). We suggest the following standard for element order:

- all external stimuli
- all connective elements between the stimuli and the fields (if connective elements are set up in a chain, the elements of that chain should be added in descending order)
- all dynamic elements (neural fields and memory traces)
- all connective elements between fields (in descending order for chains of connective elements)