SAND2023-07158C

Sandia National Laboratories

https://github.com/sandialabs/Fugu

Clone, install and follow-along!
All code for the tutorial is available in the examples folder.

Building Scalable, Composable Spiking Neural Algorithms with Fugu

ICONS Tutorials
August 1st 2023

Team:
Brad Aimone
Sylvain Bernard
Suma Cardwell
Frances Chance
Yang Ho
Ingrid Lane*
Zubin Kane
Michael Krygier
Srideep Musuvathy
Leah Reeder*
Fred Rothganger
Corinne Teeter
Craig Vineyard
Felix Wang

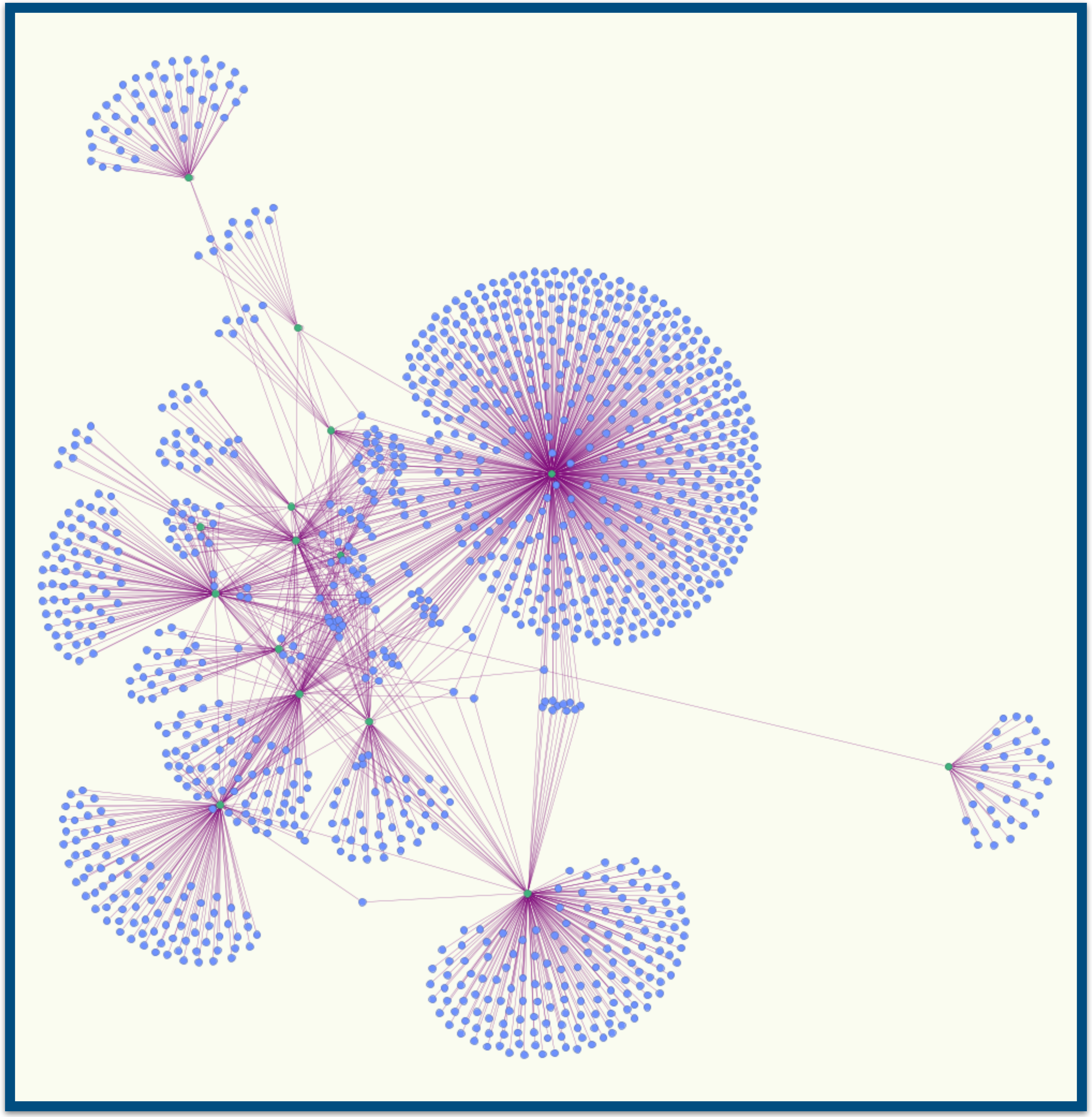
Presented by Srideep Musuvathy[†]
William Severa and Craig Vineyard

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc. for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. SAND2023-07158C

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Agenda

- The Why and How of Fugu
- Workflow
- Building a Scaffold
- Building a Brick
- Backend Overview



Introduction, Concepts and Key Design Ideas

The Why and How of Fugu

Fugu was created out of need to map applications to Neuromorphic hardware

First Prototype	First Distribution to Partners	SpiNNaker Backend	Loihi Backend	Open Source v0.1
Sept 2018	**April 2019**	**July 2019**	**February 2020**	**June 2022**

https://github.com/sandialabs/Fugu

- Fugu aims to help support the field by providing standardization and expand accessibility by lowering barriers of entry
- Some Fugu-related publications:
  - Aimone, James B., William Severa, and Craig M. Vineyard. "Composing neural algorithms with Fugu." Proceedings of the International Conference on Neuromorphic Systems. 2019.
  - Reeder, Leah Evelyn, James Bradley Aimone, and William Mark Severa. The Future of Computing: Integrating Scientific Computation on Neuromorphic Systems. No. SAND-2019-14547R. Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2020.
  - Vineyard, Craig, et al. "Neural Mini-Apps as a Tool for Neuromorphic Computing Insight." Neuro-Inspired Computational Elements Conference. 2022.
  - Aimone, James B., et al. "Spiking Neural Streaming Binary Arithmetic." 2021 International Conference on Rebooting Computing (ICRC). IEEE, 2021.
  - Aimone, James, et al. "A review of non-cognitive applications for neuromorphic computing." Neuromorphic Computing and Engineering (2022).

Motivations

- Hardware Independence - Write once, run in several places

  “You wrote LCA code for Loihi; I want to run it on SpiNNaker.”

- Composition - Pieces work together with one-another

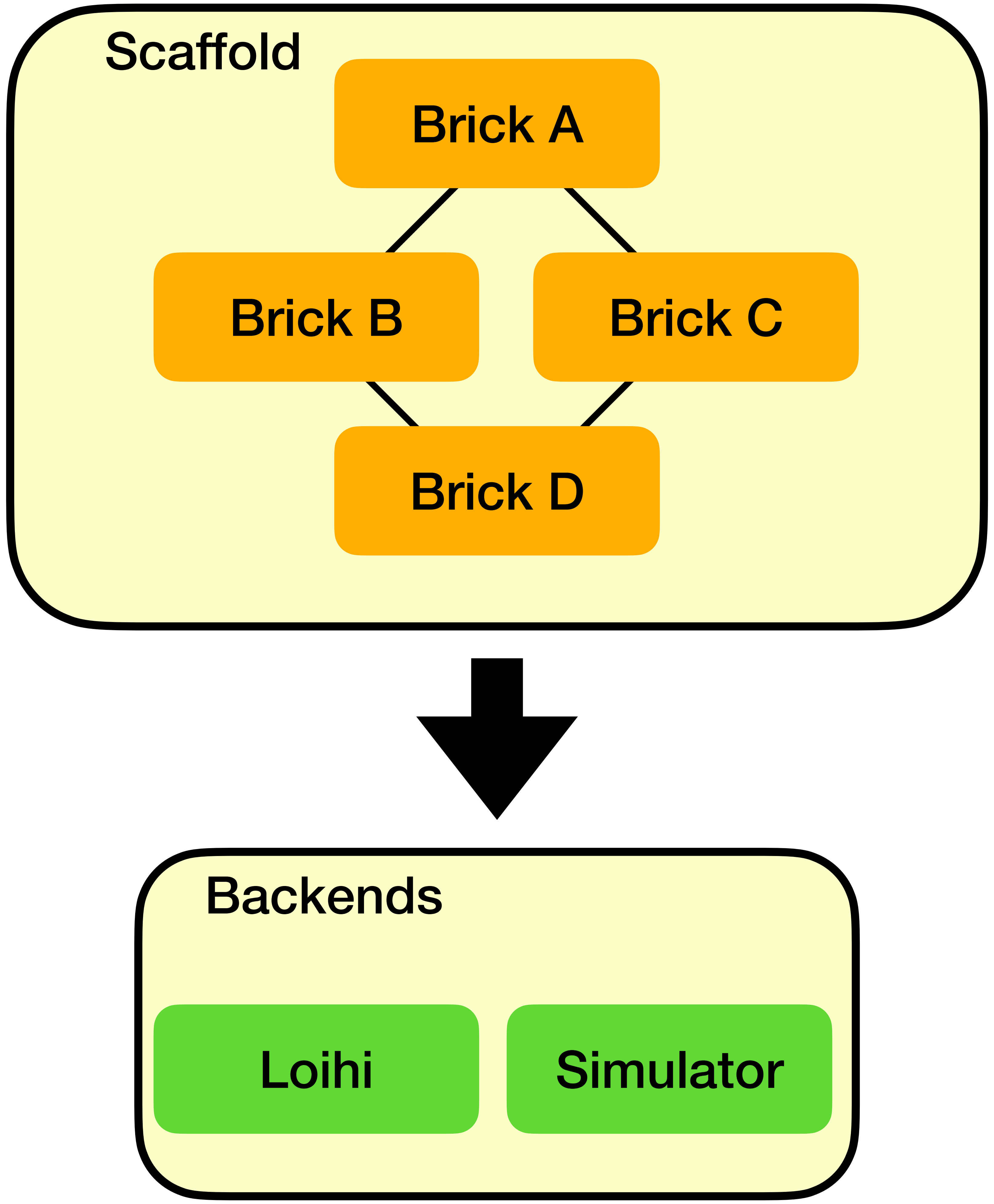
  “I want to use your Localization method with my new idea for Navigation.”

- Scalability - Grows to your problem size

  “I found a CNN trained for 28x28, but my data is 32x32.”

Key Classes

- Bricks
  - Roughly represents a function
  - Should be standalone but also not decomposable
  - Contain code to generate a network
- Scaffold
  - Represents an application
  - Composed of bricks linked together in a graph
- Backends
  - Responsible for conversion to platform-specific versions
  - Responsible for hardware-specific oddities



User Cases

- Users will define an application at the level of functions/Bricks by building a Scaffold
  - Bricks (and Scaffolds) use spikes for both inputs and outputs
  - Users should not worry about how the network itself is built or how/where it runs
- Brick Builders will write code that instructs how a brick will be built
  - Bricks must know which neurons need to exist and how they connect
  - Bricks do not need to know where they run or what they’re connected to
- Backend Developers will write code that converts a generic graph to a HW graph

More About Bricks

- Bricks should be standalone and represent ‘one job’
- Bricks have some defining characteristics, and all bricks should exhibit these qualities
- Bricks should build their network (neurons, synapses) procedurally
- Bricks should maintain compatibility by using standard input/output codings

Workflow and Features

Example Classifier

(Not all these bricks exist)

In this example, we have data $x_i \in X$ and a pre-trained MLP.

We want to classify x_i after some preprocessing using the pre-trained MLP.

Operations are

1. Preprocessing: $f(x) = \dfrac{x - 108}{255}$
2. Apply MLP to $f(x)$


User defines a scaffold
Scaffold
Input
Subtract 108
Divide by 256
MLP
Output
Scaffold is built
Scaffold.lay_bricks()
Scaffold
Input
Subtract 108
Divide by 256
MLP
Output
Dimensionality
Scaffold.graph
Neurons
Synapses
Rough Example of Code
#User defines a scaffold
scaffold = Scaffold()
scaffold.add_brick(Vector_Input(array), 'input')
scaffold.add_brick(Subtract(108))
scaffold.add_brick(Divide(256))
scaffold.add_brick(MLP(model_file), output=True)
scaffold.lay_bricks() #Scaffold is built
backend = loihi_Backend() #User defines a backend
backend.compile(scaffold) #User compiles Scaffold
#User runs a Scaffold
output_spikes = backend.run(time steps)
Loihi HW
User compiles Scaffold
with a Backend
User defines a backend
Loihi Backend
Backend
Parameters
Loihi Backend
Backend
Parameters
HW
Graph
Output
Spikes
User runs a scaffold
via a backend

What is supported?

Neurons and Synapses

- For best compatibility, we choose a very simple neuron model:

$$\hat{x}_{i,t+1} = x_{i,t} + I_i + W_i \cdot S_{t-1}$$

$$S_{t,i} = \begin{cases} 1 \text{ if } \hat{x}_{i,t} > T_i \text{ and } a < p_i \\ 0 \text{ otherwise} \end{cases}$$

$$x_{i,t+1} = \begin{cases} (1 - m_i) \cdot \hat{x}_{i,t} & \text{if } S_{t,i} = 0 \\ 0 & \text{otherwise} \end{cases}$$

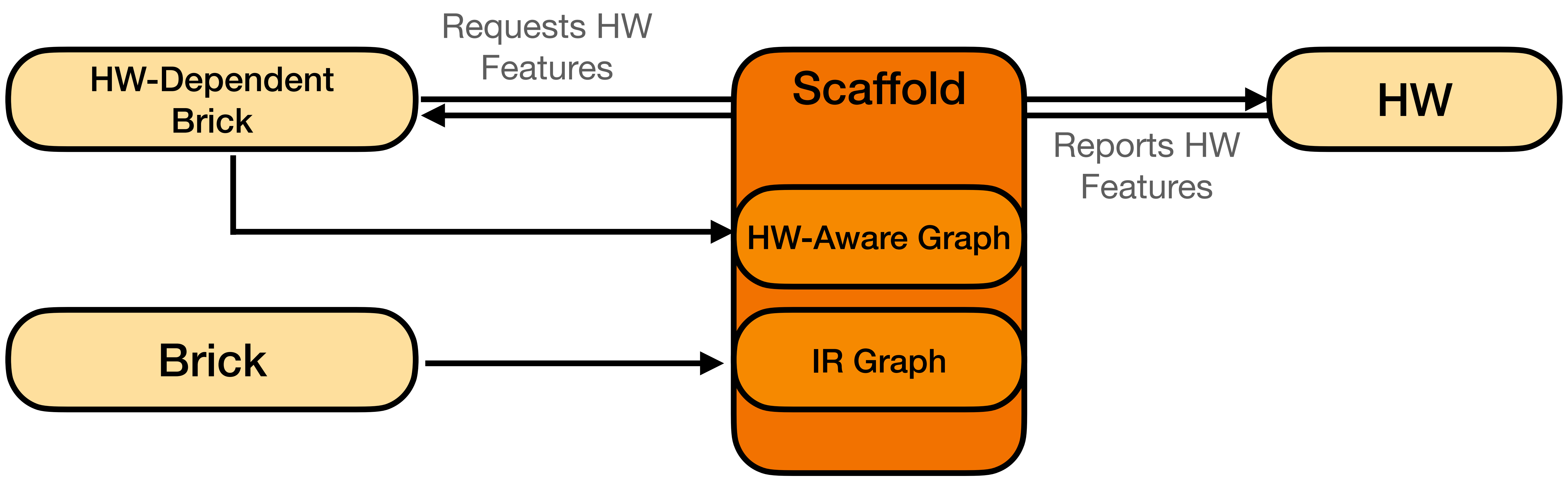
where x_i is the potential of neuron i, I_i is an injection current, W_i is a weight matrix, S is a spike history, T_i is the threshold, p_i is the spike probability, m_i is a decay constant* and a is a random uniform draw

- Synapses have a weight and an integer delay.
- Spikes persist for a single tilmestep and increase the post-synaptic potential by the weight value.

* Correct at the time of presentation. Fred is looking to update this and will notify if there are changes.

What about learning?

- For now, Fugu is designed for non-learning or offline learning algorithms
- A learned component can be fed to a brick at instantiation
- In the future, we would like to support learning in the same manner as other hardware-specific features
- As features become common, they can be moved into the fugu base neuron



Schematic for HW-specific features

How to Interact with Fugu

We are always looking for collaborators!

https://github.com/sandialabs/Fugu

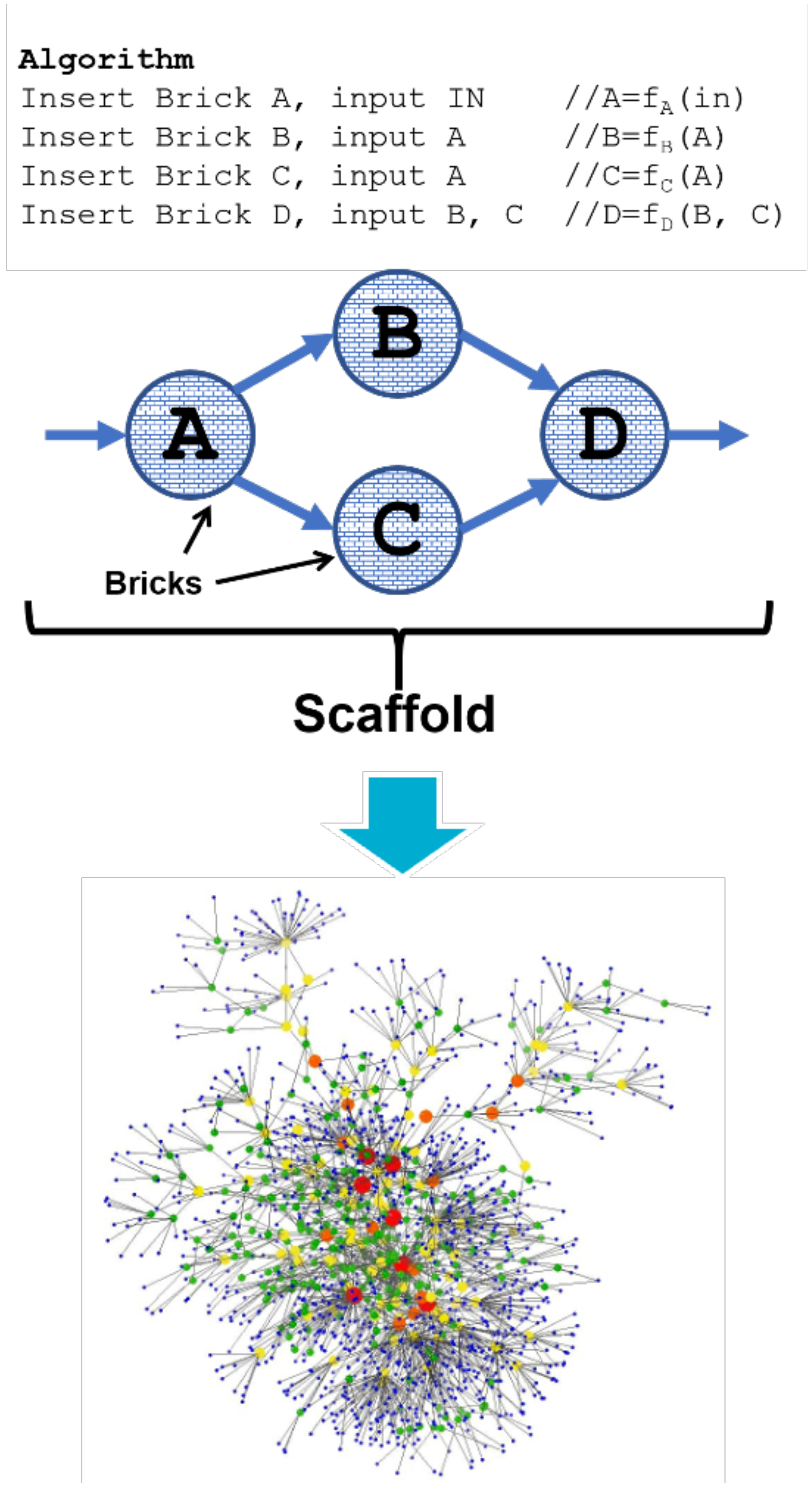
If you use Fugu for research, please cite our ICONS paper: Aimone, Severa, Vineyard. *Composing neural algorithms with Fugu*, 2019.

- End User - Works with bricks, scaffolds and backends
  - Should **clone** Fugu repository and import fugu.
  - New code should go in its own project-specific repository
- Brick/Backend Builder - Creates new Bricks/Backends for End User
  - If the code is generally applicable, create a feature **branch** from Fugu, write code, **merge request.** Recommended to e-mail wg-fugu@sandia.gov to coordinate and collaborate first.
  - If the code is project-specific or sensitive, create your own repository and inherit from Brick / Backend
- Core Fugu - Modifies Core parts of Fugu
  - Create a feature **branch** from Fugu, create code, **merge request.** Recommended to e-mail wg-fugu@sandia.gov to coordinate and collaborate first.
  - Large suggestions/collaborations will require discussions with wg-fugu@sandia.gov

Building a Basic Scaffold

Scaffolds

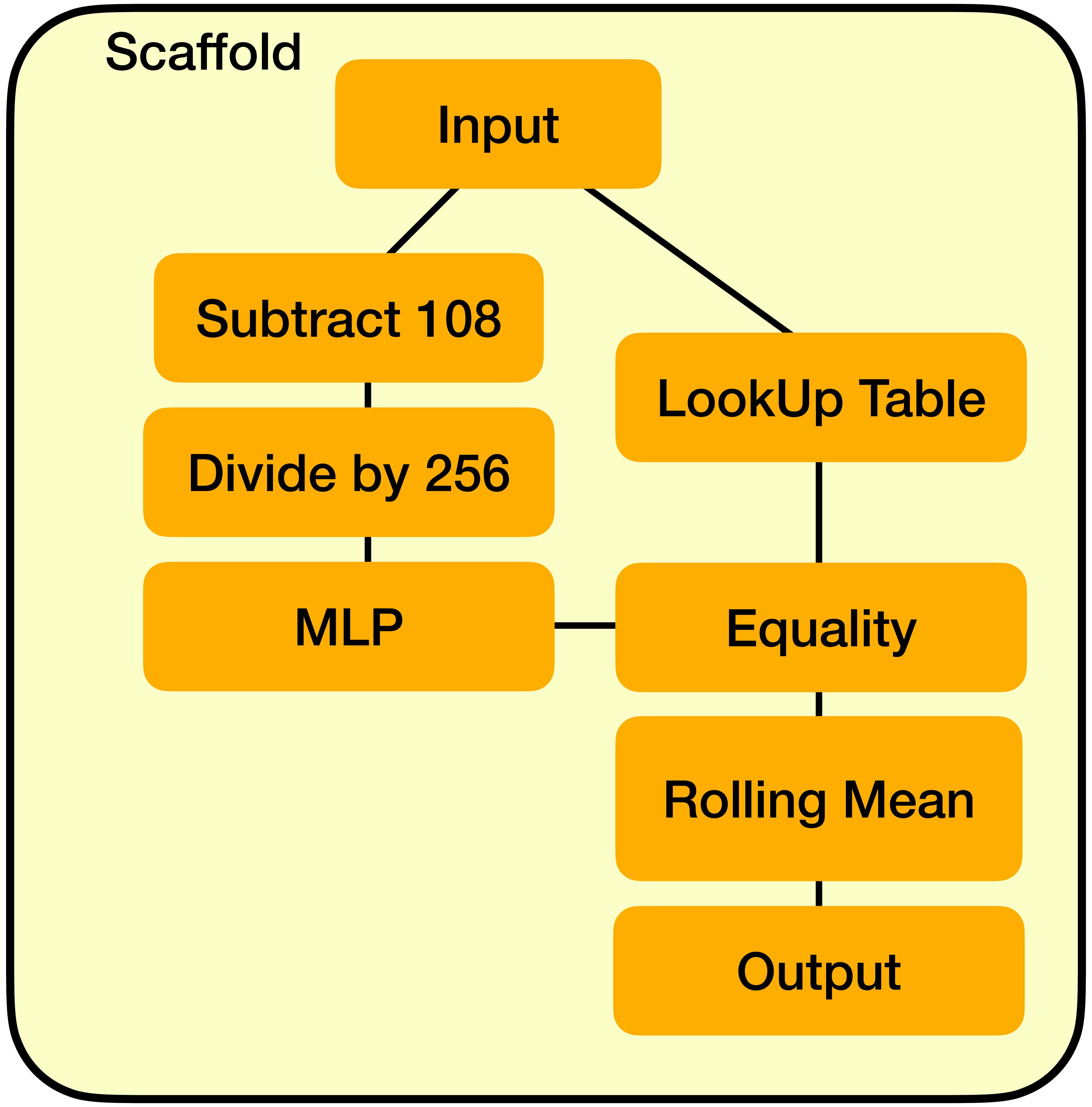
- The main entry point for using Fugu
- Should represent an application
- Could represent an algorithm
    - But Bricks also use an algorithm, so that’s a little confusing
- **Spikes in, spikes out**



Necessary Components

I.e. what to worry about

- The computation we want to do
  - Scaffolds are directed acyclic graphs
  - Think of each node as a function that operates on data
- Input spikes and dimensionality
  - Scaffolds accept an input spike raster (numpy array)
  - Future bricks could handle spike generation automatically



Built-in conversions to/from spikes soon!

Fibonacci Example

`examples/FibonacciTutorial.ipynb`

Setup

This notebook shows how Fugu can be used to generate more complex arithmetic circuits from basic streaming arithmetic functions such as addition. The goal of this notebook is to show how more complex arithmetic functions can be simply composed from Fugu bricks.

Step 0: Setup
First, we need to import Fugu and other relevant libraries. Here, we will include the adder bricks and basic setup.

```
import networkx as nx
import numpy as np
import fugu
from fugu import Scaffold, Brick
from fugu.bricks import Vector_Input
from fugu.backends import snn_Backend
from fugu.bricks import streaming_adder, temporal_shift
```

```
#A function to plot spike rasters:
def plot_spike_raster(scaffold, results):
    import matplotlib.pyplot as plot
    num_elements=scaffold.graph.number_of_nodes()
    print('Number of neurons: ', num_elements)
    results.plot.scatter(x='time', y='neuron_number', title="Spike Raster")
    plot.show()

#A function to compute the value from LEIT coded neurons
def compute_value(result):
    for i in range(0,8):
        add_element=12+9*i
        last_adder_begin=np.sum(result.query('neuron_number=='+str(add_element)+'-6')['time'])
        query_str=str(last_adder_begin)+' <= time and neuron_number ==' +str(add_element-1)
        f10=np.sum(2**(result.query(query_str)['time']-last_adder_begin))
        print('Fibonacci ' + str(i+3) + ' '+ str(f10) +' at neuron ' + str(add_element-1))
```

Brute Force Circuit

The goal of this circuit is to implement the Fibonacci sequence by brute force; if we want to go 10 layers, we will have 10 adders.

First we instantiate an empty scaffold, and we prep some input data.

The input data might seem strange; The examples in this notebook describe circuits that use inputs which are encoded using a little-endian-in-time (LEIT) coding scheme. LEIT coding is simple - think of a binary description of a number (19 = 10011), flip it around so the least significant bit is first (19 => 11001), and then have the input neuron spike at the first, second, and fifth timesteps.

```
scaffold = Scaffold()

#Input values
F_1=[0,0]
F_2=[1,0]
shift_length_total = 2
```

Brute Force Circuit

Specifying Input Bricks

We'll now add input bricks to represent the first and second values. We add a brick to a scaffold like this:

```
scaffold.add_brick(brick_function, input_nodes=[], metadata=None, name=None, output=False)
```

- `brick_function` is the brick itself.
- `input_nodes` is a list of inputs nodes. In this case, they are inputs so we have the special case of 'input', or equivalently ['input'].
- `metadata` is used to include extra information about a node. Previously this had some functionality, but now is mostly just for taking notes and is usually unused.
- `ouput` set to `True` if this is an output for the network. Generally, only output bricks are recorded.

We use `Vector_Input` as our input brick type. This type of input brick is useful for loading a numpy array (or anything that can be cast to a numpy array) as a spike train.

Creating a `Vector_Input` looks like this:

```
Vector_Input(spikes, time_dimension = False, coding='Undefined', batchable=True,
name='VectorInput')
```

```
scaffold.add_brick(Vector_Input(np.array([F_1]), coding='binary-L', name='F1', time_dimension=True), 'input') #0
scaffold.add_brick(Vector_Input(np.array([F_2]), coding='binary-L', name='F2', time_dimension=True), 'input') #1
```

Brute Force Circuit

More addition

Let's repeat the process to build more sums. We could've (and should've) use a for loop for this.

One thing to note, bricks transfer spikes to the next brick as soon as possible.

In this example, that matters because (for example): `F1 + F2 = F3` (via brick `add_12_`)

`F2 + F3 = F4` (via brick `add_23_`)

But `F2` will send its spikes to `add_23_` as soon as they are available. But, `add_12_` takes time to compute. It takes precisely 2 timesteps. So, we add an additional brick, `temporal_shift` to delay the spikes from `F2`. This way all the information arrives at `add_23_` at the same time.

```
# The second adder adds a time-delayed version (2 timesteps) of F2 and F3.  This output is F4
scaffold.add_brick(temporal_shift(name='shift_2_', shift_length=shift_length_total), [(1,0)], output=True) #3
scaffold.add_brick(streaming_adder(name='add_23_'), [(2,0), (3,0)], output=True)

# The third adder adds a time-delayed version of F3 and F4.  This output is F5
scaffold.add_brick(temporal_shift(name='shift_3_', shift_length=shift_length_total), [(2,0)], output=True) #5
scaffold.add_brick(streaming_adder(name='add_34_'), [(4,0), (5,0)], output=True)

# The fourth adder adds a time-delayed version of F4 and F5.  This output is F6
scaffold.add_brick(temporal_shift(name='shift_4_', shift_length=shift_length_total), [(4,0)], output=True) #7
scaffold.add_brick(streaming_adder(name='add_45_'), [(6,0), (7,0)], output=True)
```

Building a Basic Brick

How Do Bricks Communicate?

- Bricks use a predefined set of codings to represent input/output values
  - We need better definition of what a coding means

Coding types are soon to be redefined and defined more concretely and more formally!

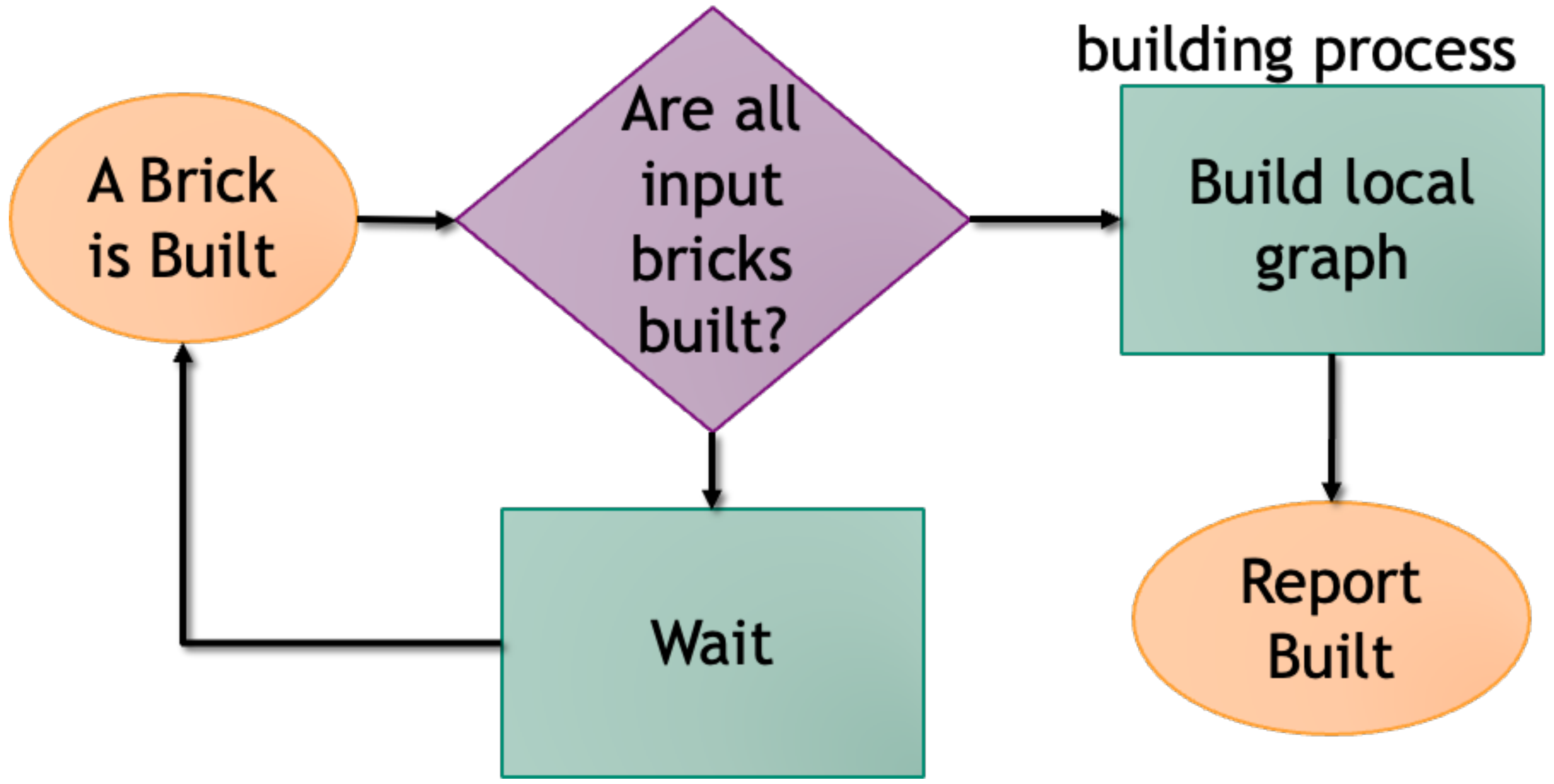
- Bricks also define 'control neurons' which are used to send extra signals such as
  - When a brick should start processing
  - When a brick has finished processing

Name	Description
unary-B	Unary coding, large values first
unary-L	Unary, small values first
binary-B	Binary, large values first
binary-L	Binary, small values first
temporal-B	Temporal, large values first
temporal-L	Temporal, small values first
Raster	Grid-like array
Population	# active represents value
Rate	Rate coded neurons
Undefined	Neurons without a coding
Current	Used for pre-threshold computation

How are Bricks built?

- Bricks are built by iterating over the Scaffold.circuit
- Each brick is responsible for
  - Building a portion, i.e. its part, of the neuron graph
  - Providing indexed output neurons
  - Connecting *Control Nodes*
- Each brick is provided
  - Indexed input neurons from incoming bricks
  - Incoming *Control Nodes*
  - Any Brick-specific parameters

Control Flow for a Brick's building process



80-20 Example

`examples/EightTwentyTutorial.ipynb`

80-20 Example

Networks with 80 percent excitatory and 20 percent inhibitory connectivity are common and can be used, for example, as a liquid in a reservoir computing method (e.g. LSM). This tutorial builds a quick bricks for such a network.

To try to avoid confusion and overloading of terms, we'll refer to neurons within our 80-20 network to be 'liquid neurons' or 'neurons in the liquid,' etc. First, some imports:

```
import numpy as np
np.random.seed(0)
import networkx as nx
import fugu
from fugu import Scaffold, Brick
from fugu.bricks import Vector_Input
from fugu.backends import snn_Backend

def plot_spike_raster(scaffold, results):
    import matplotlib.pyplot as plot
    num_elements=scaffold.graph.number_of_nodes()
    print('Number of neurons: ', num_elements)
    results.plot.scatter(x='time', y='neuron_number', title="Spike Raster")
    plot.show()

def plot_scaffold_graph(scaffold):
    edge_weights = [scaffold.graph.edges[v]['weight'] for v in scaffold.graph.edges()]
    nx.draw_networkx(scaffold.graph,
                     with_labels = False,
                     pos = nx.spring_layout(scaffold.graph),
                     edge_color = edge_weights,
                     node_size = 100)
```

80-20 Example

Inherit from Brick

- All bricks should inherit from the `Brick` class.
- Bricks that are listed as input bricks should instead inherit from `InputBrick`, which is beyond the scope of this tutorial.
- The construction of most brick types is similar; constructing a brick that takes input coding "current" (see below) is a bit different and is beyond the scope of this tutorial.

The `Brick` class provides the framework for the a scaffold to build a neural graph. Subclasses of `Brick` should provide the actual code that will generate the nodes and edges on a graph. The graph construction should take place within the `build` method. Let's look at the definition of the parent class `Brick`.

80-20 Example

The first line `class Brick(ABC)` defines the abstract class of `Brick`. Brick objects inherit from `ABC` which just means that `Brick` is an abstract class that cannot be instantiated on its own; only subclasses may be instantiated.

The `__init__` method contains standard instantiation code. All bricks are expected to have a member property `self.name` that is unique to the brick. The uniqueness needs to be determined by the scaffold, not by the brick.

The property `self.is_built` is a boolean that is True if the brick has been built (added to the graph).

The property `self.supported_codings` is a list of input codings (strings) that the brick supports. Since you have the full use of python when you are defining your brick, you can support multiple coding types completely transparent to the user. A full list of coding types can be found at `fugu.input_coding_types`.

```python
class Brick(ABC):
    def __init__(self):
        self.name = "Empty Brick"
        self.supported_codings = []

    @abstractmethod
    def build(self, graph,
                    metadata,
                    complete_node,
                    input_lists,
                    input_codings):
        pass
```

Better and more formal coding definitions are being developed!

80-20 Example

The method `build` will be called by the scaffold when the graph is to be built. Arguments are:

- graph: The graph object that we are building onto.
- metadata: A dictionary of shapes and parameters. This will likely be modified in future implementations, so don't rely on it.
- control_nodes: A *dict* of *lists* of nodes that transmit a control information. The most common is `control_nodes['complete']` which carries a list of 'finished' spikes from input bricks. If your brick has one input, then this will be a list of a single node. The only other currently used key is `control_nodes['begin']` which is used for temporally coded bricks (and outside the scope of this tutorial)
- input_lists: A *list of lists* of nodes that correspond to input neurons. The outermost list contains a list of neurons, one for each input on the scaffold.
- input_codings: A *list* of input coding types. The list contains one coding type per input on the scaffold.

Each brick is responsible for throwing the appropriate errors/warnings if the inputs are not compatible with the brick.

```python
class Brick(ABC):
    def __init__(self):
        self.name = "Empty Brick"
        self.supported_codings = []

    @abstractmethod
    def build(self, graph,
                    metadata,
                    complete_node,
                    input_lists,
                    input_codings):
        pass
```

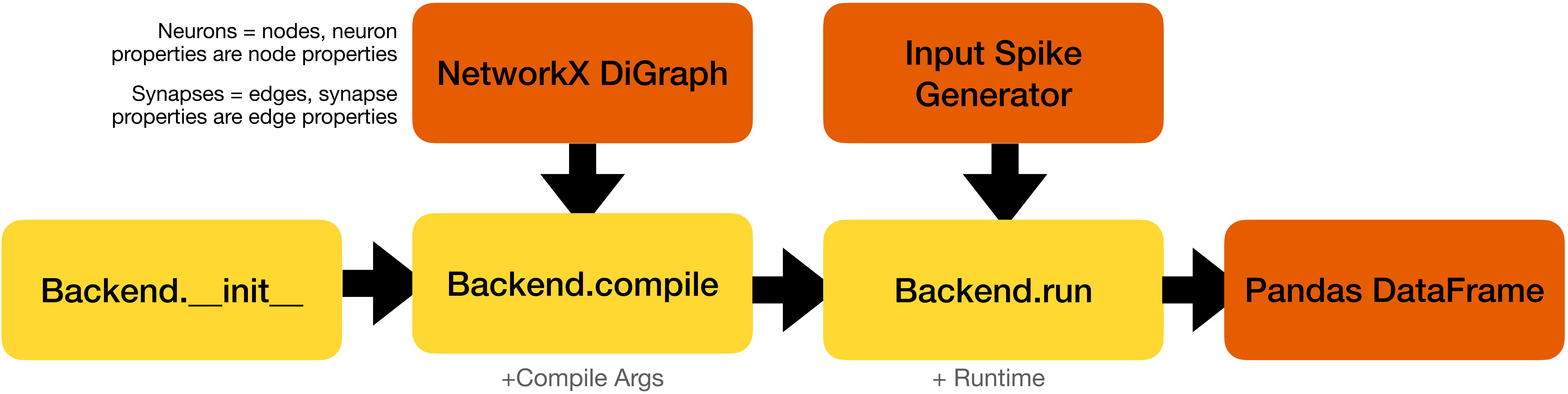
Brick-to-brick communication is being redesigned!

Introduction to Backend Design

Why do we need backends?

- Backends provide an interface with an execution platform (Hardware or Simulator)
- This allows Fugu networks to run on multiple platforms
- Decoupling network design and network execution is an incredibly useful idea!
- From the user perspective:
  - You should not need to worry about hardware details
  - Anything you write will work on new hardware once a backend exists
- From the HW perspective:
  - Experts in the HW are the ones that are interfacing with HW
  - Anything previously written in Fugu is available as soon as you write a backend

Backend Lifetime



- Backend is an abstract class
- Instantiates the backend and any HW initialization or environment variables

- Converts generic IR to platform-specific representation
- Backend owns a platform-specific copy of the graph

- Executes or simulates the network graph

- Return should be a sparse DataFrame where each record is a spike (time, neuron)

Benefits and Challenges for Backend Design

Benefits

- Experts in a particular execution platform (hardware) write platform-specific code
- Common hardware limitations can be abstracted from algorithm design
- Platform-specific optimizations and best practices are easy to use
- Backend designers do not need to worry about spiking algorithms

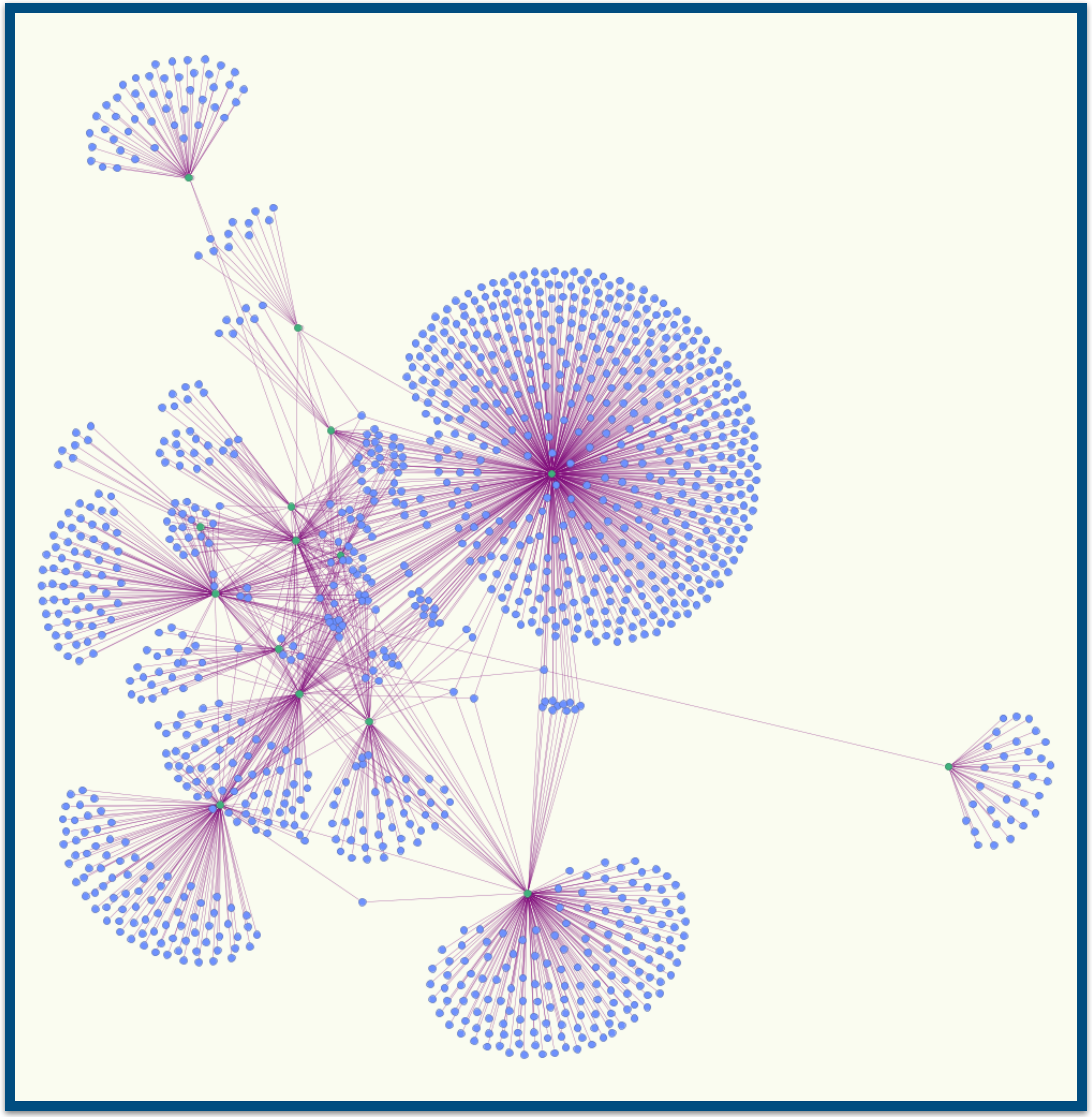
Challenges

- Backend must support general SNN (arbitrary graph, specific neuron model)
- Dynamic range in weights, potentials may be large
- Currently no clear backend testing regimen
- Network graph must fit entirely in memory

Guidelines for (ranges and precision) for neuron and synapse properties! Backend testing program!

Wrap up

- Fugu Motivation
- Workflow
- Scaffold
- Brick
- Backend Overview



August 3rd16:45-17:00 Presentation:

Neuromorphic Population Evaluation using the Fugu Framework.

William Severa, Suma Cardwell, Michael Krygier, Fredrick Rothganger and Craig Vineyard.