Contents

1 Tutorial: Driven Filter Circuit 1
2 Tutorial: Building Modular Enzymatic Reactions 5
3 How-To Guides 11
4 Discussion 15
5 API Reference 17
6 BondGraphTools 27
Bibliography 29
Python Module Index 31
Index 33
Goal: Build and simulate a simple passive filter using basic bond graph modelling techniques.

Difficulty: Beginner.

Requirement: BondGraphTools, jupyter.

How to follow: Enter each block of code in consecutive cells in a jupyter notebook.

In part 1 of this tutorial we will demonstrate how to build and connect models using BondGraphTools by constructing a simple passive filter. Part 2 introduces control sources, and provides examples of how one can perform parameter sweeps or input comparisons.

### 1.1 Part 1: Basic Use

First, import BondGraphTools and create a new model with the name “RC”:

```python
import BondGraphTools as bgt
model = bgt.new(name="RC")
```

Now create a new generalised linear resistor (‘R’ component), a generalised linear capacitor (“C” component) with resistance and capacitance both set to 1, and an equal effort (“0” junction) conservation law through which these components share energy.:

```python
C = bgt.new("C", value=1)
R = bgt.new("R", value=1)
zero_law = bgt.new("0")
```

Add the newly created components to the model:

```python
bgt.add(model, R, C, zero_law)
```

Once the components are added to the model, connect the components and the law together. (Note the first argument is the tail of the energy bond, the second is the head):
Draw the model to make sure everything is wired up:

```python
bgt.draw(model)
```

which produces a sketch of the network topology.

To demonstrate that the isolated system is behaving correctly, we simulate from the initial where the C component has $x_0=1$ and run the simulation over the time interval $(0,5)$. This results in a vector $t$ of time of time points and a corresponding vector $x$ of data points which can then be plotted against each other with `matplotlib`

```python
timespan = [0, 5]
x0 = [1]
t, x = simulate(model, timespan=timespan, x0=x0)
from matplotlib.pyplot import plot
fig = plot(t,x)
```

### 1.2 Part 2: Control

We wish to see how this filter responds to input. Add flow source by creating a new Sf component, adding to the model, and connecting it to the common voltage law:

```python
Sf = bgt.new('Sf')
bgt.add(model, Sf)
bgt.connect(Sf, zero_law)
```

The model should now look something like this

```python
draw(model)
```

The model also now has associated with it a control variable $u_0$. Control variables can be listed via the attribute `model.control_vars` and we can observe the constitutive relations, which give the implicit equations of motion for the system in sympy form:

```python
model.constitutive_relations
# returns \[dx_0 - u_0 + x_0\]
```

where $x_0$ and $dx_0$ are the state variable and it’s derivative. One can identify where that state variable came from via:

```python
model.state_vars
# returns {'x_0': (C: C1, 'q_0')}
```

Here $C$: $C1$ is a reference to the $C$ object itself.

### 1.3 Part 3: Simulations

We will now run various simulations.
Firstly, we simulate with constant effort by passing the control law $u_0=2$ to the solver and plotting the results:

```python
import numpy as np
import matplotlib.pyplot as plt
from BondGraphTools import Model

# Simulation settings
timespan = [0, 5]
x0 = [1]
t, x = simulate(model, timespan=timespan, x0=x0, control_vars={'u_0':2})
plot(t,x)
```

Time dependent control laws can be specified as string. In this case we consider the response to a $\pi^{-1}$Hz sine wave:

```python
import numpy as np
import matplotlib.pyplot as plt
from BondGraphTools import Model

# Simulation settings
timespan = [0, 5]
x0 = [1]
t, x = simulate(model, timespan=timespan, x0=x0, control_vars={'u_0':'sin(2*t)'}
plot(t,x)
```

One can also consider the impulse response of by applying a step function input to the control law:

```python
step_fn = 't < 1 ? 1 : 0' # if t < 0 then 1 else 0
import numpy as np
import matplotlib.pyplot as plt
from BondGraphTools import Model

# Simulation settings
timespan = [0, 5]
x0 = [1]
t, x = simulate(model, timespan=timespan, x0=x0, control_vars={'u_0':step_fn})
plot(t,x)
```

Finally we run a sequence of simulations where a new control law is generated based on the loop iteration.

```python
import numpy as np
import matplotlib.pyplot as plt
from BondGraphTools import Model

fig = plt.figure()
for i in range(4):
    func_text = "cos({i}t)".format(i=i)
t_i, x_i = simulate(model, timespan=timespan, x0=x0, control_vars={'u_0':func_text})
plot(t_i,x_i)
```
2.1 Overview

Goal: To build and simulate a simple reaction A = B, and exploit modularity to substitute in different reactions structures.

Difficulty: Intermediate

Requirement: BondGraphTools, jupyter.

How to follow: Enter each block of code in consecutive cells in a jupyter notebook.

2.2 Part 1: Building a basic chemical reaction

Begin by opening a new jupyter notebook, importing the toolkit and creating a new model:

```python
import BondGraphTools as bgt
model = bgt.new(name="Reaction")
```

Next, add two chemical species, two common effort junctions and a reaction:

```python
A_store = bgt.new("Ce", name="A", library="BioChem", value={'k':10})
B_store= bgt.new("Ce", name="B", library="BioChem", value={'k':1})
A_junction = bgt.new("0")
B_junction = bgt.new("0")
reaction = bgt.new("Re", library="BioChem")

bgt.add(model, A_store, B_store,A_junction, B_junction, reaction)
```

Then wire the network up:
bgt.connect(A_store, A_junction)
bgt.connect(A_junction, reaction)
bgt.connect(reaction, B_junction)
bgt.connect(B_junction, B_store)

Set the pressure and temperature to one (ie; parameters are normalised) and set the conductance of the reaction component to None so as to treat it as control variable:

```python
for param_index, (component, parameter_name) in model.params.items():
    if parameter_name in ('T', 'R'):
        bgt.set_param(model, param_index, 1)
    elif component is reaction:
        bgt.set_param(model, param_index, None)
```

Draw the model to inspect the network topology:

```python
bgt.draw(model)
```

One can go ahead and run simulations on this for example, by varying the reaction rate inside a loop and plotting the results:

```python
import matplotlib.pyplot as plt
x0 = {'x_0':1, 'x_1':1}
t_span = [0,5]
fig = plt.figure()
ax = plt.gca()
ax.set_title("One Step Reaction")
for c, kappa in [('r', 0.1), ('b', 1), ('g', 10)]:
    t, x = bgt.simulate(model, x0=x0, timespan=t_span, control_vars={'u_0':kappa})
    plt.plot(t,x[:,0], c+'-')
    plt.plot(t,x[:,1], c)
```

### 2.3 Part 2: Modularity and enzyme catalysed reactions.

We wish the replace the one step reaction above with an enzyme catalysed reaction. Our first step will be to make a factory function which takes an enzyme name as an argument and produces an enzyme catalysed reaction model. This process is nearly identical to the steps followed in part 1, however we now wrap the procedure in a function so we can reuse it later. Define a factory function to produce models of enzyme catalysed reactions:

```python
def enzyme_catalysed_reaction(name):
    
    This function produces a bond graph model of an basic enzyme catalysed reaction of the from `S + E = E + P` where the substrate and product are exposed as external ports.

    Args:
        name (str): The name of the enzyme

    Returns:
        `BondGraph`: The resulting model
```

(continues on next page)
Use this function to build a new enzyme catalysed reactions, and draw it to make sure the topology is correct:

```python
E1 = enzyme_catalysed_reaction("E1")
bgt.draw(E1)
```

In order to replace the reaction, with the newly built `E1`, first remove all the bonds connecting the original reaction:

```python
for bond in model.bonds:
    if reaction is bond.head.component or reaction is bond.tail.component:
        bgt.disconnect(bond.tail, bond.head)
```

Then remove the old reaction and add `E1`:

```
2.3. Part 2: Modularity and enzyme catalysed reactions.
```
bgt.remove(model, reaction)
bgt.add(model, E1)

Complete the substitution by connecting the substrate to ‘A’ and the product to ‘B’. Draw the model to verify the substitution is complete.

```python
substrate_port, = (port for port in E1.ports if port.name == "S")
product_port, = (port for port in E1.ports if port.name == "P")
bgt.connect(A_junction, substrate_port)
bgt.connect(product_port, B_junction)
bgt.draw(model)
```

Inspect the models constitutive relations, state variables and control vars by:

```python
# State Variables
print(model.state_vars)
# outputs {'x_0': (C: A, 'q_0'), 'x_1': (C: B, 'q_0'), 'x_2': (BG: E1, 'x_0')}

# Control Variables
print(model.control_vars)
# outputs {'u_0': (BG: E1, 'u_0')}

print(model.constitutive_relations)
# outputs [dx_0 + 10*u_0*x_0*x_2 - u_0*x_1*x_2,
  # dx_1 - 10 *u_0*x_0*x_2 + u_0*x_1*x_2,
  # dx_2]
```

Here we can see that the $x_2$ co-ordinate of the model points to the $x_0$ co-ordinate of the enzyme reaction, which we know to be the state of the enzyme component $C:E$. Observe that the appearance of $dx_2$ alone in the constitutive relations implies that $x_2$, the enzyme quantity, is conserved.

### 2.4 Part 3: Exploiting Modularity to reaction chains

We will now use the above function to build a reaction chain. That is, we think the correct model of $A=B$ is $A = A1 = A2 = A3 = B$. Create a new model to represent the reaction chain and add the substrate and product ports:

```python
chain = bgt.new(name="3 Step Chain")
substrate = bgt.new("SS", name='S')
product = bgt.new("SS", name='P')
substrate_law = bgt.new("0")
product_law = bgt.new("0")
bgt.add(chain, substrate, product, substrate_law, product_law)
bgt.connect(substrate, substrate_law)
bgt.connect(product_law, product)
bgt.expose(substrate)
bgt.expose(product)
```

Now, add the first step in the linear chain of reactions, and connect it to the substrate law.:
reaction_step = enzyme_catalysed_reaction('E1')

bgt.add(reaction_step)
substrate_port, = (port for port in reaction_step.ports if port.name == "S")
bgt.connect(substrate_law, substrate_port)

Iteratively add each segment of the linear chain, by finding the product of the last reaction, connecting that to a newly created intermediary $A_i$, which is then connected to the substrate of the next catalysed reaction:

```python
for i in range(1, 4):
    last_product_port, = (port for port in reaction_step.ports if port.name == "P")
    step_law = bgt.new("0")
    step_ce = bgt.new("Ce", library="BioChem", name=f"A{i}" , value={"R":R,"T":T, "k":1})
    reaction_step = enzyme_catalysed_reaction(f"E{i}")
    bgt.add(chain, step_ce, step_law, reaction_step)
    substrate_port, = (port for port in reaction_step.ports if port.name == "S")
    bgt.connect(last_product_port, step_law)
    bgt.connect(step_law, step_ce)
    bgt.connect(step_law, substrate_port)
```

Draw the chain to make sure everything is connected:

```python
bgt.draw(chain)
```

Observe that the constitutive relations:

```python
print(chain.constitutive_relations)
```

for this chain component is clearly a function of two efforts, and two flows, in addition to the internal state variables, and control variables.

We can now return to our model, and swap out the $E1$ for the 3 step chain:

```python
for bond in model.bonds:
    if E1 is bond.head.component or E1 is bond.tail.component:
        bgt.disconnect(bond.tail, bond.head)

bgt.remove(model, E1)
bgt.add(model, chain)

substrate_port, = (port for port in chain.ports if port.name == "S")
product_port, = (port for port in chain.ports if port.name == "P")

bgt.connect(A_junction, substrate_port)
bgt.connect(product_port, B_junction)
```

Observing `bgt.draw(model)`, the network topology of the model has not changed. The difference is noticeable when the constitutive relations are produced.
print(model.constitutive_relations)
# [dx_0 + 10*u_0*x_0*x_2 - u_0*x_2*x_3,
# dx_1 + u_3*x_1*x_8 - u_3*x_7*x_8,
# dx_2,
# dx_3 - 10*u_0*x_0*x_2 + u_0*x_2*x_3 + u_1*x_3*x_4 - u_1*x_4*x_5,
# dx_4,
# dx_5 - u_1*x_3*x_4 + u_1*x_4*x_5 + u_2*x_5*x_6 - u_2*x_6*x_7,
# dx_6,
# dx_7 - u_2*x_5*x_6 + u_2*x_6*x_7 - u_3*x_1*x_8 + u_3*x_7*x_8,
# dx_8]

Where the model co-ordinates are given by:

print(model.state_vars)
# {'x_0': (C: A, 'q_0')},
# 'x_1': (C: B, 'q_0'),
# 'x_2': (BG: 3 Step Chain, 'x_0'),
# 'x_3': (BG: 3 Step Chain, 'x_1'),
# 'x_4': (BG: 3 Step Chain, 'x_2'),
# 'x_5': (BG: 3 Step Chain, 'x_3'),
# 'x_6': (BG: 3 Step Chain, 'x_4'),
# 'x_7': (BG: 3 Step Chain, 'x_5'),
# 'x_8': (BG: 3 Step Chain, 'x_6')}
3.1 Installing BondGraphTools on Linux

A working installation of both Python and Julia is required to use all of BondGraphTools features. In this guide we walk step-by-step through the recommended method of installation.

Commands in this section beginning with $ are entered at the terminal prompt. Commands beginning with >>> are enter at the Python interpreter prompt.

We assume Ubuntu 16.04 or greater. BondGraphTools _should_ work on all distributions but it is only tested and supported on Ubuntu 16.04 and up.

3.1.1 Step 1: Install Python 3.6 or greater

Ubuntu 17.10 or Greater

Skip this step as Python 3.6 is already installed.

Ubuntu 14.04 Only

A newer version of the C++ standard library is required. First, download it a working directory with:

```
$ wget -q -O libstdc++6 http://security.ubuntu.com/ubuntu/pool/main/g/gcc-5/
    --libstdc++6_5.4.0-6ubuntu1~16.04.10_amd64.deb
```

then install using:

```
$ sudo dpkg --force-all -i libstdc++6
```
Ubuntu 14.04 and 16.04

Add the ‘deadsnakes’ repository to the Ubuntu package manager:

```
$ sudo add-apt-repository ppa:deadsnakes/ppa
```

Ubuntu 14.04 to 16.10

Update the package indices and install Python 3.6:

```
$ sudo apt update
$ sudo apt install python36 python36-pip
```

### 3.1.2 Step 2: Install Julia

We must now install Julia 0.6. Our examples follow the Julia platform instructions [https://julialang.org/downloads/platform.html#generic-binaries](https://julialang.org/downloads/platform.html#generic-binaries).

1. Download Julia 0.6.4 generic binaries from [https://julialang.org/downloads/oldreleases.html](https://julialang.org/downloads/oldreleases.html). For example using curl to save the archive to your downloads:

   ```
   $ curl -L https://julialang-s3.julialang.org/bin/linux/x64/0.6/julia-0.6.4-linux-
   x86_64.tar.gz > ~/Downloads/julia-0.6.4-linux-x86_64.tar.gz
   ```

2. Extract the archive `.tar.gz` to the desired directory. For example, here we first create the target directory, then extract the tarball:

   ```
   $ sudo mkdir /usr/local/opt/julia
   $ sudo tar xf ~/Downloads/julia-0.6.4-linux-x86_64.tar.gz --directory /usr/local/
   opt/julia --strip-components=1
   ```

3. Add julia to the PATH, for example by adding a symbolic link to the local bin folder:

   ```
   $ sudo ln -s /usr/local/opt/julia/bin/julia /usr/local/bin/julia
   ```

4. Verify that this has worked by typing `julia -v`, which should result in `julia version 0.6.4`.

### 3.1.3 Step 3: Install BondGraphTools

Use python to install BondGraphTools with:

```
$ pip3 install BondGraphTools
```

or alternatively:

```
$ python3.6 -m pip install BondGraphTools
```

It is optional but highly recommended to install the Jupyter packages via:

```
$ pip3 install Jupyter
```
3.1.4 Step 4: Julia dependencies.

BondGraphTools will automatically download and install Julia dependencies the first time a user attempts to run a simulation.

This can be manually triggered via the following commands in the python interpreter:

```
>>> from bondgraphtools.config import config
>>> config.install_dependencies()
```

3.2 Installing BondGraphTools on Mac OSX

A working installation of both Python and Julia is required to use all of BondGraphTools features. In this guide we walk step-by-step through the recommended method of installation.

3.2.1 Step 1: Instal Homebrew and Python

First we must install python 3. We recommend doing this via Homebrew.

1. Open a terminal instance.
2. Paste the following command into the terminal instance:

```
$ /usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```
3. Once Homebrew has finished, install python by typing the following command into the terminal:

```
$ brew install python
```

Python 3.7 is now installed.

3.2.2 Step 2: Install Julia

We must now install Julia 0.6.

1. Download Julia 0.6.4 from https://julialang.org/downloads/oldreleases.html
2. Once the download is complete open the file. It will contain a link to th Applications folder, and the Julia-0.6 package. Drag the julia package into the applications folder. Once this is done, julia has been installed and you can remove the downloaded file.
3. Add julia to your profile path by typing the following command into the terminal:

```
$ echo 'export PATH=/Applications/Julia-0.6.app/Contents/Resources/julia/bin:/$PATH' >> ~/.bash_profile
```

3.2.3 Step 3: Install BondGraphTools

1. Install BondGraphTools from PyPI by typing the following command into the terminal:

```
$ pip3 install BondGraphTools
```
2. Install the recommended additional packages (jupyter) by typing the following command into the terminal:

$ pip3 install jupyter

BondGraphTools is now installed.

### 3.2.4 Step 4: Julia dependencies.

BondGraphTools will automatically download and install Julia dependencies the first time a user attempts to run a simulation.

This can be manually triggered via the following command:

```python
>>> from bondgraphtools.config import config
>>> config.install_dependencies()
```

### 3.3 Troubleshooting

#### 3.3.1 Julia Errors

If you encounter julia errors, you may need to rebuild pycall, or re-install your julia libraries.

1. To rebuild pycall and related libraries enter the following commands in your python interpreter:

```python
from bondgraphtools.config import config
config.install_dependencies(rebuilt=True)
```

2. If that fails, you may need to reset your julia library. **Warning** this will delete any additional libraries you may have installed. This can be done by removing the directory ‘~/julia/lib’ in Linux/MacOSX, or ‘C:\users\<username>\julia\lib’ in windows.
CHAPTER 4

Discussion
CHAPTER 5

API Reference

5.1 BondGraphTools

class BondGraphTools.BondGraph(name, components=None, **kwargs)
  Representation of a bond graph model.

  basis_vectors
  Basis vectors for the state space (X), port space (J), and control space (U) from an external point of view.
  For the state space dictionaries are of the form:

```python
X = {
    sympy.Symbol('x_i'): (object, var)
}
```

We assume the object is a subclass of BondGraphBase and the var refers to the variable name in the objects local co-ordinate system and may be a string or a sympy.Symbol.

For the port space, dictionaries are of the form:

```python
J = {
    (sympy.Symbol(e_i), sympy.Symbol(f_i)): Port(obj, idx)
}
```

where Port is an instance of Port. Finally for the control variables we have:

```python
U = {
    sympy.Symbol(u_i): (object, var)
}
```

Where object and var are specified as per the state space.

bonds
  The list of connections between internal components
components = None
The components, instances of BondGraphBase, that make up this model

control_vars
A dict of all control variables in the form:

```python
{
    "u_0": (component, control_var)
}
```

internal_ports
A list of the ports internal to this model

map_port (label, ef)
Exposes a pair of effort and flow variables as an external port:
:param label: The label to assign to this port.
:param ef: The internal effort and flow variables.

params
A dictionary of parameters for this model in the form:

```python
i: (component, param_name)
```

state_vars
A dict of all state variables in the form:

```python
{
    "x_0": (component, state_var)
}
```

Where "x_0" is the model state variable, and state_var is the corresponding state variable of component

system_model (control_vars=None)
Produces a symbolic model of the system in reduced form.

In many cases it is useful to have a full description of the system in symbolic form, and not just a list of constitutive relations.

Returns (coordinates, mappings, linear_op, nonlinear_op, constraints)
This method generates:

- The model coordinate system (list) $x$
- A mapping (dict) between the model coordinates and the component coordinates
- A linear operator (sympy.Matrix) $L$
- A nonlinear operator (sympy.Matrix) $F$
- A list of constraints (sympy.Matrix) $G$

The coordinates are of the form

$$x = (dx_0, dx_1, \ldots, e_0, f_0, e_1, f_1, \ldots, x_0, x_1, \ldots, u_0, u_1, \ldots)$$

So that the system obeys the differential-algebraic equation

$$Lx + F(x) = 0 \quad G(x) = 0$$

See also:

BondGraph.basis_vectors

view = None
Graphical Layout of internal components
**BondGraphTools.new**

`creates a new Bond Graph from a library component.`

**Parameters**

- **component** *(str or obj)* – The type of component to create. If a string is specified, the component will be created from the appropriate library. If an existing bond graph is given, the bond graph will be cloned.
- **name** *(str)* – The name for the new component
- **library** *(str)* – The library from which to find this component (if specified by string) *(component)*
- **value** –

**Returns:** instance of *BondGraph*

**Raises:** NotImplementedError

**BondGraphTools.add** *(model, *args)*

Add the specified component(s) to the model

**BondGraphTools.swap** *(old_component, new_component)*

Replaces the old component with a new component. Components must be of compatible classes; one port cannot replace an n-port, for example. The old component will be completely removed from the system model.

**Parameters**

- **old_component** – The component to be replaced. Must already be in the model.
- **new_component** – The substitute component which must not be in the model

**Raises** InvalidPortException, InvalidComponentException

**BondGraphTools.remove** *(model, component)*

Removes the specified components from the Bond Graph model.

**BondGraphTools.connect** *(source, destination)*

Connects two components or ports.

Defines a power bond between the source and destination ports such that the bond tail is at the source, and the bond head is at the destination. We assume that either the source and/or destination is or has a free port.

**Parameters**

- **source** *(Port or BondGraphBase)* – The tail of the power bond
- **destination** *(Port or BondGraphBase)* – The head of the power bond

**Raises** InvalidPortException, InvalidComponentException

**See also:**

disconnect()  

**BondGraphTools.disconnect** *(target, other)*

Disconnects the flow of energy between the two components or ports. If there is no connection, this method does nothing.

**Parameters**

- **target** *(Port, BondGraphBase)* –
- **other** *(Port, BondGraphBase)* –

**Raises** InvalidComponentException

---

5.1. BondGraphTools
See also:
connect()

BondGraphTools.expose(component, label=None)
Exposes the component as port on the parent.

If the target component is not a SS component, it is replaced with a new SS component. A new external port is added to the parent model, and connected to the SS component.

Parameters

- component – The component to expose.
- label – The label to assign to the external port

Raises: InvalidComponentException

BondGraphTools.set_param(component, param, value)
Sets the specified parameter to a particular value.

Parameters

- component (BondGraphBase) – The particular component.
- param – The parameter to set
- value – The value to assign it to, may be None

BondGraphTools.simulate(system, timespan, x0, dx0=None, dt=0.1, control_vars=None)
Simulate the system dynamics.

This method integrates the dynamics of the system over the specified interval of time, starting at the specified initial state.

The solver used is a differential-algebraic integrator which respects conservation laws and algebraic constraints. It is expected that the initial state satisfies the systems inherent algebraic constrains; inconsistent initial conditions will raise exceptions.

The initial values of derivatives can be specified and the solver will ensure they are consistent with the initial state, or change them if they are not.

Currently, control variables can take the form of numbers or a strings and are assigned via a dictionary or list.

Permissible strings:

- numerical constants such as 1.0, pi
- time t
- state variables; for example x_0
- arithmetic operators such as +, -, *, /, as well as ^ (power operator), % (remainder)
- elementary math functions such as sin, exp, log
- ternary if; for example t < 0 ? 0 : 1 which implements the Heaviside

:recipes:`BondGraph`: The system to simulate :recipes:`timespan`: A pair (list or tuple) containing the start and end points

of the simulation.

Parameters

- x0 – The initial conditions of the system.
• \texttt{dx0} (Optional) – The initial rates of change of the system. The default value (\texttt{None}) indicates that the system should be initialised from the state variable initial conditions.

• \texttt{dt} – The time step between reported (not integrated) values.

• \texttt{control\_vars} – A \texttt{dict, list or tuple} specifying the values of the control variables.

\textbf{Returns}\ numpy array of timesteps \(x\): numpy array of state values

\textbf{Return type}\ \texttt{t}

\textbf{Raises} ModelException, SolverException

\texttt{BondGraphTools.draw(system)}

Produces a network layout of the system.

\textbf{Parameters}\ \texttt{system} – The system to visualise

\textbf{Returns}\ matplotlib.Plot

### 5.2 BondGraphTools.atomic

This module contains class definitions for atomic components; those which cannot be decomposed into other components.

\texttt{class BondGraphTools.atomic.Component (metamodel, constitutive\_relations, state\_vars=None, params=None, **kwargs)}

Bases: \texttt{BondGraphTools.base.BondGraphBase, BondGraphTools.port\_managers.PortManager}

Atomic bond graph components are those defined by constitutive relations.

\textbf{basis\_vectors} See \texttt{BondGraphBase.basis\_vectors}

\textbf{constitutive\_relations} See \texttt{BondGraphBase}

\textbf{control\_vars} See \texttt{BondGraphBase}

\textbf{params} See \texttt{BondGraphBase}

\texttt{set\_param (param, value)}

Warning: Scheduled to be deprecated

\textbf{state\_vars} See \texttt{BondGraphBase}

### 5.3 BondGraphTools.base

This module contains the base classes for bond graph models and connections

\texttt{class BondGraphTools.base.Bond}

Bases: \texttt{BondGraphTools.base.Bond}

A \texttt{namedtuple} that stores a connection between two ports. Head and tail are specified to determine orientation
head
The ‘harpoon’ end of the power bond and direction of positive $f$

tail
The non-harpoon end, and direction of negative $f$

class BondGraphTools.base.BondGraphBase (name= None, parent= None, metamodel= None, **kwargs)

Bases: object

Base class definition for all bond graphs.

parent
name
metamodel
template
uri
root
basis_vectors

Parameters

• name – Assumed to be unique
• parent –
• metadata (dict) –

class BondGraphTools.base.Port (component, index)

Bases: object

Basic object for representing ports; Looks and behaves like a namedtupple: component, index = Port

component = None
(PortManager) The component that this port is attached to

index = None
(int) The numerical index of this port

is_connected = None
(bool) True if this port is plugged in.

5.4 BondGraphTools.component_manager

This module takes care of the loading and management of component libraries. This library will automatically load all factory libraries upon import. Additionally libraries can be added via load_library().

Component Libraries are expected to be in json format. The structure must be:

```json
{
    "id": The unique library id (str),
    "description": The description of this library
    "components": {
        "component_id": component dictionary,
        ...
    }
}
```
Each Component dictionary must be of the form:

```
{

}
```

`BondGraphTools.component_manager.find(component, restrict_to=None, find_all=False, ensure_unique=False)`

Finds the specified component.

**Parameters**

- `component` – The component id to find.
- `restrict_to` – list or set of library id’s to be that the search should be restricted to.
- `find_all` – False if the function should return only the first instance of the component, True if the function should return all such instances
- `ensure_unique` – If true, this assumes that the component id must be unique across libraries, and hence will raise an exception if this is assumption is violated.

**Returns:** the library id, or a list of library_id in which this component can be found.

**Raises**

- `NotImplementedError` - if the component is not found.
- `ValueError` - if the component is assume to be unique, but is not

`BondGraphTools.component_manager.get_component(component, library='base')`

Fetches the component data for the specified component :param component: The id of the specific component :param library: The id of the library to which the component belongs

**Returns** dict - the component dictionary

`BondGraphTools.component_manager.get_components_list(library)`

Fetches a list of components available in the given library.

**Parameters** library – The library id of the library to query

**Returns** list of (component id, description) tuples

`BondGraphTools.component_manager.get_library_list()`

Fetches a list of the libraries available for use.

**Returns** list of (library id, description) tuples

`BondGraphTools.component_manager.load_library(filename)`

Loads the library specified by the filename and makes it available for use.

**Parameters** filename – The (absolute) filename of the library to be loaded.

**Returns** True if the library was successfully loaded.

**Return type** bool
5.5 BondGraphTools.exceptions

Exceptions and errors for BondGraphTools

```
exception BondGraphTools.exceptions.InvalidComponentException
    Exception for when trying to use a model that can’t be found, or is of the wrong type

exception BondGraphTools.exceptions.InvalidPortException
    Exception for trying to access a port that is in use, or does not exist

exception BondGraphTools.exceptions.ModelException
    Exception for inconsistent or invalid models when running simulations

exception BondGraphTools.exceptions.ModelParsingError
    Exception for problems generating symbolic equations from string

exception BondGraphTools.exceptions_SOLVERException
    Exception for issues running numerical solving.

exception BondGraphTools.exceptions.SymbolicException
    Exception for when there are issues in model reduction or symbolic manipulation
```

5.6 BondGraphTools.fileio

The file save/load interface and file format data model

This module provides the basic IO functionality such as saving and loading to file.

```
BondGraphTools.fileio.load (file_name, model=None, as_name=None)
    Loads a model from file

Parameters
    file_name (str or Path) – The file to load.

Returns
    An instance of BondGraph

Raises
    NotImplementedError for incorrect file version.
```

```
BondGraphTools.fileio.save (model, filename)
    Saves the model to the specified path

Parameters
    • model – The model to be saved
    • filename – The file to save to
```

5.7 BondGraphTools.sim_tools

Tools for running model simulations

```
BondGraphTools.sim_tools.to_julia_function_string (model, control_vars=None, in_place=False)
    Produces a Julia function string from the given model.

We expect that that control_vars is a dict with the same keys, or list of the same size, as the model.control_vars

Parameters
    • model –
• `control_vars` –
• `in_place` –

**Returns** (string, list) A string containing the function definition, and a list of bools identifying which variables contain derivatives.

## 5.8 BondGraphTools.reaction_builder

A set of common tools for building and manipulating chemical reactions, and for producing bond graph models from a reaction network.

```python
class BondGraphTools.reaction_builder.Reaction_Network(reactions=None, name=None, temperature=300, volume=1):
    Bases: object

    Parameters
    • `reactions` –
    • `name` –
    • `temperature` – Temperature in Kelvin (Default 300K, or approx 27c)
    • `volume` –

    `add_chemostat(species, concentration=None)`
    Add a (generalised) Chemostat to the reaction network. This provides a variable source of the particular species so as to maintain a particular concentration. This can also act as a I/O source.

    **Notes**
    Only one chemostat is available per species; so adding a duplicate will overwrite the previous chemostatic concentration (if defined)

    Parameters
    • `species` – The name/identifier of the particular chemical species
    • `concentration` – (default None) The fixed concentration. Left as a free parameter if None

    `add_flowstat(species, flux=None)`
    Adds a (generalised) Flowstat, which provides a flux of the particular species in or out of the reaction network.

    **Notes**
    Only one flowstat per species, so adding duplicate flowstats will overwrite the previous flowstatic fluxes (if defined)

    **See also:**
    `add_chemostat()`

    Parameters
    • `species` – The name/identifier of the chemical species
• **flux** – (default None) The rate at which this species is added/removed. Left as a free parameter if None

```python
add_reaction(reaction, forward_rates=None, reverse_rates=None, name="")
```

Adds a new reaction to the network.

**Parameters**

- **reaction** *(str)* – A sequence of reactions to be added.
- **forward_rates** *(list)* – The forward rates of these reactions.
- **reverse_rates** *(list)* – The reverse rates of these reactions.
- **name** – The name of this set of reactions.

Reactions are assumed to be of the form:

```
"A + B = C + D = E + F"
```

Where the “math:`i`’th equals sign denotes a reversible reaction, with forward and reverse rates (if they exist) denoted by ‘forward_rate[i]’ and reverse_rate[i] respectively.

**Warning:** Rate functionality is not yet implemented!

```python
as_network_model(normalised: bool = False)
```

Produces a bond graph `BondGraphTools.BondGraph` model of the system

**Parameters**

- **normalised** – If true, sets pressure and temperature to 1

**Returns** A new instance of `BondGraphTools.BondGraph` representing this reaction system.

**fluxes**

The reaction fluxes.

A tuple `(V, x)` contain the vector `V(x)` and the coordinates `x[i]` such that for the the stoichiometric matrix `N` and the reaction rates \( \kappa = extdiag(\kappa_1, \kappa_2, \ldots) \), the mass action description of the system is

```
\dot{x} = N \kappa V(x)
```

**forward_stoichiometry**

The forward stoichiometric matrix

**name** = None

The name of this reaction network

**reverse_stoichiometry**

The reverse stoichiometric matrix

**species**

A list of the chemical species involved in this reaction network

**stoichiometry**

The stoichiometric matrix
6.1 Overview

BondGraphTools is a python library for systems modelling based on the bond graph methodology [Gaw1996], [Gaw2007].

**BondGraphTools is intended to be used by:**

- software developers; as a framework to build modelling interfaces on top of
- engineers; to quickly build and simulate physical systems
- mathematicians; to perform model reduction and analysis

BondGraphTools is built upon the scientific python stack; numpy, scipy, sympy and matplotlib, and relies on Julia and DifferentialEquations.jl to handle numerical simulation.

6.2 How to read the docs

The documentation for this library is organised into five sections.

- This page show how to install BondGraphTools
- **Tutorials** contains a list of step-by-step tutorials demonstrating how to use BondGraphTools.
- **How-To Guides** contains recipes for common tasks.
- **Discussion** contains high level discussion about to library.
- **API Reference** is the library reference documentation.
6.3 Getting Started

6.3.1 Requirements

- Python 3.6 or above.
- Julia 0.6. Note: Julia 0.7 and above are not yet supported

6.3.2 Installation

1. Install Julia 0.6.4 which can be downloaded from here.
2. Make sure Julia 0.6.4 is in your path variables. You can test this by typing `julia -v` at the command prompt. If this returns `julia version 0.6.4` then julia is in your path variables.
3. Install BondGraphTools via pip: `pip install BondGraphTools`

BondGraphTools is now installed and will download and install the required julia packages the first time you import it.

6.3.3 Usage

BondGraphTools can be loaded inside a jupyter-notebook, python interpreter, or script by using:

```python
import BondGraphTools
```

Users need an internet connection during the first load as the it will download the required packages from the julia package repository.

Api reference for the package and it’s contents can be accessed using the help command:

```python
help(BondGraphTools)
```

6.3.4 Contributing

The best way to contribute right now is to use it! If you wish to help out, please visit the project github.

6.4 Bibliography
Bibliography


<table>
<thead>
<tr>
<th>Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>BondGraphTools</td>
<td>17</td>
</tr>
<tr>
<td>BondGraphTools.atomic</td>
<td>21</td>
</tr>
<tr>
<td>BondGraphTools.base</td>
<td>21</td>
</tr>
<tr>
<td>BondGraphTools.component_manager</td>
<td>22</td>
</tr>
<tr>
<td>BondGraphTools.exceptions</td>
<td>24</td>
</tr>
<tr>
<td>BondGraphTools.fileio</td>
<td>24</td>
</tr>
<tr>
<td>BondGraphTools.reaction_builder</td>
<td>25</td>
</tr>
<tr>
<td>BondGraphTools.sim_tools</td>
<td>24</td>
</tr>
<tr>
<td>A</td>
<td>BondGraphTools.add() (in module BondGraphTools), 19</td>
</tr>
<tr>
<td>--------------------</td>
<td>-----------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>BondGraphTools.add_chemostat() (BondGraphTools.reaction_builder.Reaction_Network.method), 25</td>
</tr>
<tr>
<td></td>
<td>BondGraphTools.add_flowstat() (BondGraphTools.reaction_builder.Reaction_Network.method), 25</td>
</tr>
<tr>
<td></td>
<td>BondGraphTools.add_reaction() (BondGraphTools.reaction_builder.Reaction_Network.method), 26</td>
</tr>
<tr>
<td></td>
<td>BondGraphTools.as_network_model() (BondGraphTools.reaction_builder.Reaction_Network.method), 26</td>
</tr>
<tr>
<td>B</td>
<td>BondGraphTools.atomic.basis_vectors (BondGraphTools.atomic.Component attribute), 21</td>
</tr>
<tr>
<td></td>
<td>BondGraphTools.atomic.basis_vectors (BondGraphTools.base.BondGraphBase attribute), 22</td>
</tr>
<tr>
<td></td>
<td>BondGraphTools.atomic.basis_vectors (BondGraphTools.BondGraph attribute), 17</td>
</tr>
<tr>
<td></td>
<td>BondGraphTools.Bond (class in BondGraphTools.base), 21</td>
</tr>
<tr>
<td></td>
<td>BondGraphTools.Bond (class in BondGraphTools), 17</td>
</tr>
<tr>
<td></td>
<td>BondGraphTools.BondGraphBase (class in BondGraphTools.base), 22</td>
</tr>
<tr>
<td></td>
<td>BondGraphTools.BondGraphTools (module), 17</td>
</tr>
<tr>
<td></td>
<td>BondGraphTools.BondGraphTools.atomic (module), 21</td>
</tr>
<tr>
<td></td>
<td>BondGraphTools.BondGraphTools.base (module), 21</td>
</tr>
<tr>
<td></td>
<td>BondGraphTools.BondGraphTools.component_manager (module), 22</td>
</tr>
<tr>
<td></td>
<td>BondGraphTools.BondGraphTools.exceptions (module), 24</td>
</tr>
<tr>
<td></td>
<td>BondGraphTools.BondGraphTools.fileio (module), 24</td>
</tr>
<tr>
<td></td>
<td>BondGraphTools.BondGraphTools.reaction_builder (module), 25</td>
</tr>
<tr>
<td></td>
<td>BondGraphTools.BondGraphTools.sim_tools (module), 24</td>
</tr>
<tr>
<td></td>
<td>BondGraphTools.BondGraphTools.bonds (BondGraphTools.BondGraph attribute), 17</td>
</tr>
<tr>
<td>C</td>
<td>BondGraphTools.base.Port.index (BondGraphTools.base.Port attribute), 22</td>
</tr>
<tr>
<td></td>
<td>BondGraphTools.base.Port.component (class in BondGraphTools.atomic), 21</td>
</tr>
<tr>
<td></td>
<td>BondGraphTools.BondGraphTools.BondGraphTools.bonds (BondGraphTools.BondGraph attribute), 17</td>
</tr>
<tr>
<td></td>
<td>BondGraphTools.BondGraphTools.component_manager (module), 23</td>
</tr>
<tr>
<td></td>
<td>BondGraphTools.BondGraphTools.exceptions (module), 24</td>
</tr>
<tr>
<td></td>
<td>BondGraphTools.BondGraphTools.fileio (module), 24</td>
</tr>
<tr>
<td></td>
<td>BondGraphTools.BondGraphTools.reaction_builder (module), 25</td>
</tr>
<tr>
<td></td>
<td>BondGraphTools.BondGraphTools.sim_tools (module), 24</td>
</tr>
<tr>
<td></td>
<td>BondGraphTools.BondGraphTools.bonds (BondGraphTools.BondGraph attribute), 17</td>
</tr>
<tr>
<td></td>
<td>BondGraphTools.base.Port.index (BondGraphTools.base.Port attribute), 22</td>
</tr>
</tbody>
</table>
Index

internal_ports (BondGraphTools.BondGraph attribute), 18
InvalidComponentException, 24
InvalidPortException, 24
is_connected (BondGraphTools.base.Port attribute), 22

L
load() (in module BondGraphTools.fileio), 24
load_library() (in module BondGraphTools.component_manager), 23

M
map_port() (BondGraphTools.BondGraph method), 18
metamodel (BondGraphTools.base.BondGraphBase attribute), 22
ModelError, 24
ModelParsingError, 24

N
name (BondGraphTools.base.BondGraphBase attribute), 22
name (BondGraphTools.reaction_builder.Reaction_Network attribute), 26
new() (in module BondGraphTools), 19

P
params (BondGraphTools.atomic.Component attribute), 21
params (BondGraphTools.BondGraph attribute), 18
parent (BondGraphTools.base.BondGraphBase attribute), 22
Port (class in BondGraphTools.base), 22

R
Reaction_Network (class in BondGraphTools.reaction_builder), 25
remove() (in module BondGraphTools), 19
reverse_stoichiometry (BondGraphTools.reaction_builder.Reaction_Network attribute), 26
root (BondGraphTools.base.BondGraphBase attribute), 22

S
save() (in module BondGraphTools.fileio), 24
set_param() (BondGraphTools.atomic.Component method), 21
set_param() (in module BondGraphTools), 20
simulate() (in module BondGraphTools), 20
SolverException, 24

species (BondGraphTools.reaction_builder.Reaction_Network attribute), 26
state_vars (BondGraphTools.atomic.Component attribute), 21
state_vars (BondGraphTools.BondGraph attribute), 18
stoichiometry (BondGraphTools.reaction_builder.Reaction_Network attribute), 26
swap() (in module BondGraphTools), 19
SymbolicException, 24
system_model() (BondGraphTools.BondGraph method), 18

T
tail (BondGraphTools.base.Bond attribute), 22
template (BondGraphTools.base.BondGraphBase attribute), 22
to_julia_function_string() (in module BondGraphTools.sim_tools), 24

U
view (BondGraphTools.BondGraph attribute), 18

V