BondGraphTools Documentation

Release 0.4.5

Peter Cudmore

Aug 17, 2021

CONTENTS

CHAPTER

ONE

TUTORIAL: DRIVEN FILTER CIRCUIT

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":

```
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.:

```
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:

```
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):

```
bgt.connect(R, zero_law)
bgt.connect(zero_law, C)
```

Draw the model to make sure everything is wired up:

```
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 points and a corresponding vector x of data points which can then be plotted against each other with `matplotlib`:

```
timespan = [0, 5]
x0 = [1]
t, x = bgt.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:

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

The model should now look something like this:

```
bgt.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:

```
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:

```
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:

```
timespan = [0, 5]
x0 = [1]
t, x = bgt.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 π^{-1} Hz sine wave:

```
t, x = bgt.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 ot the control law, here x and dx refer to the state-space of the model:

```
def step_fn(t,x,dx):
    return 1 if t < 1 else 0

t, x = bgt.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:

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

CHAPTER

TWO

ADDITIONAL TUTORIALS

2.1 Biochemical Networks Tutorials

Dr. Michael Pan has prepared a number of tutorials here for applications involving biochemical systems.

CHAPTER

THREE

TUTORIAL: BUILDING MODULAR ENZYMATIC REACTIONS

Goal:	To build and simulate a simple reaction A = B, and exploit modularity to substitute in different reactions structures.
Difficulty:	Intermediate.
Requirements:	BondGraphTools, jupyter.
How to follow:	Enter each block of code in consecutive cells in a jupyter notebook.

3.1 Part 1: Building a basic chemical reaction

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

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

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

```
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.:

```
for param_index, (component, parameter_name) in model.params.items():
    if parameter_name in ("T", "R"):
        bgt.set_param(model, param_index, 1)
```

(continues on next page)

(continued from previous page)

```
    elif component is reaction:
        bgt.set_param(model, param_index, None)
```

Draw the model to inspect the network topology.:

```
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:

```
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)
```

3.2 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::

```
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
    """

    cat_model = bgt.new(name=name)

    # Construct the external ports.
    substrate = bgt.new("SS", name="S")
    product = bgt.new("SS", name="P")
```

(continues on next page)

(continued from previous page)

```
    # Here we build the reaction, again with the rate as a control variable.
    # Again, we assume parameterised have be normalised with respect to
    # pressure and temperature.
    cat_reaction = bgt.new("Re", name="Re", library="BioChem", value={'r':None, 'R':1, 'T
↪':1})

    # We choose 'k' to be 1 for demonstration.
    enzyme = bgt.new("Ce", name="E", library="BioChem", value={'k':1, 'R':1, 'T':1})

    # Substrate + Enzyme flux conservation law
    SE = bgt.new('1')
    # Product + Enzyme flux conservation law
    PE = bgt.new('1')

    # Conservation of enzyme law.
    law_E = bgt.new("0")

    bgt.add(cat_model, substrate, product, enzyme, SE, PE, law_E, cat_reaction)

    connections = [
        (substrate, SE),
        (law_E, SE),
        (law_E, enzyme),
        (SE, cat_reaction),
        (cat_reaction, PE),
        (PE, law_E),
        (PE, product)
    ]
    for tail, head in connections:
        bgt.connect(tail, head)

    bgt.expose(substrate, 'S')
    bgt.expose(product, 'P')

    return cat_model
```

Use this function to build a new enzyme catalysed reactions, and draw it to make sure the topology is correct:

```
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.:

```
bgt.disconnect(A_junction, reaction)
bgt.disconnect(reaction, B_junction)
```

Then remove the old reaction and add *E1*:

```
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 vefify the

substitution is complete

```
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:

```
# 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.

3.3 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:

```
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, label='S')
bgt.expose(product, label='P')
```

Now, add the first step in the linear chain of reactions, and connect it to the substrate law.:

```
reaction_step = enzyme_catalysed_reaction('E1')

```

(continues on next page)

(continued from previous page)

```
bgt.add(chain, 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.:

```
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":1,"T":1, "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)


last_product_port, = (port for port in reaction_step.ports if port.name == "P")
bgt.connect(last_product_port, product_law)
```

Draw the chain to make sure everything is connected.:

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

Observe that the constitutive relations:

```
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:

```
bgt.disconnect(E1, A_junction)
bgt.disconnect(E1, B_junction)

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')}
```

CHAPTER

FOUR

DISCUSSION

4.1 Additional Material

4.1.1 Bond Graph Clinic Slides 2018

Slides are available for 4 introductory workshops on bond graph modelling which were held at the University of Melbourne in 2018.

4.2 BondGraphsTools from 10,000 feet

BondGraphTools is designed to be a modelling tool, to allow the user to programmatically describe the power storing, transforming and dissipating elements of a system, and generate simplified/reduced symbolic equations for the system as a whole.

4.2.1 Model Capture

A bond graph model is created by:

1. Instantiating a new bond graph - a composite container for the system, or part of a larger system, that the user wishes to model.
2. Existing components (either atomics drawn from standard libraries, or composites built via this process) are added to the bond graph model, describing the various processes (eg, dissipation, or energy storage) active within the system in question.
3. Connections are defined between the ports of components, capturing the channels through which power flows.
4. If necessary, ports are *exposed*, defining a power port outside the container which mirrors the power flowing through a corresponding interior port.

Once a model is created, it is ready for reduction and/or simulation, or for integration with other models to form a larger, more compex system.

4.2.2 Model Reduction

In the context of BondGraphTools, model reduction refers to the process by which the constituative relations (equations which constrain power flows and storage within a given component) and connectivity constraints (ie, bonds) are passed through a series of symbolic algebraic simplifications so as to generate a lower-dimensional represenation of the system.

The model capture procedure results in composite state X containing all dynamic variables (eg, position/momentum), control variables and power variables (efforts and flows). The behaviour of the system is then governed by the level sets of $\Phi(X) = 0$. By exploiting inherient structures (such a sparsity and a only small amounts of nonlinearly), an invertible co-ordinate transform $Y = P(X)$ can be found such that $\dim(Y) \ll \dim(X)$, along with a reduced order constitutive relation for the composed system $\tilde{\Phi}(Y) = 0$, where $\tilde{\Phi}$ is 'simpler' in the sense that it contains a minimal set of relations describing the systems dynamics.

This model reduction happens behind the scenes when the user asks for the constiutive relations of their newly-constructed model, or in prepation for running numerical simulation.

CHAPTER

FIVE

API REFERENCE

5.1 BondGraphTools

class BondGraphTools.**BondGraph**(**args*, ***kwargs*)

Representation of a bond graph model.

property 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:

```
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:

```
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:

```
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

The components, instances of `BondGraphBase`, that make up this model

property constitutive_relations

The equations governing this objects dynamics.

property control_vars

A *dict* of all control variables in the form:

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

property 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.

property metamodel
The meta-model type of this object.

property params

A dictionary of parameters for this model in the form:: i: (component, param_name)

property state_vars
A *dict* of all state variables in the form:

```
{
   "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

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

See also:

BondGraph.basis_vectors

property template
The model template from which this was created.

BondGraphTools.**new**(*component=None*, *name=None*, *library='base'*, *value=None*, ***kwargs*)
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 the component will be created from the appropriate libaray. 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
- **string)** (*component is specified by*) –
- **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; 1 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* –

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**(**args*, ***kwargs*)
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.

todo: detial control variables.

:param system *BondGraph*: The system to simulate :param timespan: A pair (*list* or *tuple*) containing the start and end

points of the simulation.

Parameters

- **x0** – The initial conditions of the system.
- **dx0** (*Optional*) – The initial rates of change of the system. The default value (*None*) indicates that the system should be initialised from the state variable initial conditions.
- **dt** – The time step between reported (not integrated) values.
- **control_vars** – A *dict*, *list* or *tuple* specifing the values of the control variables.

Returns numpy array of timesteps x: numpy array of state values

Return type t

Raises *ModelException*, *SolverException* –

BondGraphTools.**draw**(*system*)
Produces a network layout of the system.

Parameters **system** – The system to visualise

Returns matplotlib.Plot

5.2 BondGraphTools.atomic

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

class BondGraphTools.atomic.**Component**(**args*, ***kwargs*)
Bases: *BondGraphTools.base.BondGraphBase*, BondGraphTools.port_managers.PortManager

Components defined by constitutive relations.

property basis_vectors
See *BondGraphBase.basis_vectors*

property constitutive_relations
See *BondGraphBase*

property control_vars
See *BondGraphBase*

property metamodel
The meta-model type of this object.

property params
See *BondGraphBase*

set_param(*param*, *value*)
Warning: Scheduled to be deprecated

property state_vars
See *BondGraphBase*

property template
The model template from which this was created.

5.3 BondGraphTools.base

Base classes for bond graph models and connections.

class BondGraphTools.base.**Bond**(*tail*, *head*)
Bases: *BondGraphTools.base.Bond*

Stores the 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**(**args*, ***kwargs*)
Bases: object

Base class definition for all bond graphs.

Parameters

- **name** – Name of this model, assumed to be unique.
- **parent** – Parent model, or none.
- **metamodel** – Metamodel class.

property **basis_vectors**
The input/ouput, dynamic and interconnect vectors.

property **constitutive_relations**
The equations governing this objects dynamics.

property **metamodel**
The meta-model type of this object.

parent
Model that contains this object.

property **root**
The root of the tree to which this object belongs.

property **template**
The model template from which this was created.

property **uri**
Model reference locator.

class BondGraphTools.base.**Port**(*component*, *index*)
Bases: object

Basic object for representing ports.

Looks and behaves like a namedtuple: component, index = Port

Parameters

- **component** – The owner of this port.
- **index** – The index of this port in the component.

component
(*PortManager*) The component that this port is attached to

index
(int) The numerical index of this port

is_connected
(bool) True if this port is plugged in.

5.4 BondGraphTools.component_manager

Component Libraries.

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:

```
{
    "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*)

Find 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'*)

Fetch the component data for the specified component.

Parameters

- **component** – The id of the specific component
- **library** – The id of the library to which the component belongs

Returns dict - the component dictionary

BondGraphTools.component_manager.**get_components_list**(*library*)

Fetch 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**()
Fetch a list of the libraries available for use.

Returns list of (library id, description) tuples

BondGraphTools.component_manager.**load_library**(*filename*)
Load the library specified by the filename.

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*)
Load a model from file.

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

Returns An instance of *BondGraph*

Raises **NotImplementedError** –

BondGraphTools.fileio.**save**(*model*, *filename*)
Save the model to file.

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.**simulate**(**args*, ***kwargs*)

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.

todo: detial control variables.

:param system BondGraph: The system to simulate :param timespan: A pair (*list* or *tuple*) containing the start and end

points of the simulation.

Parameters

- **x0** – The initial conditions of the system.
- **dx0** (*Optional*) – The initial rates of change of the system. The default value (*None*) indicates that the system should be initialised from the state variable initial conditions.
- **dt** – The time step between reported (not integrated) values.
- **control_vars** – A *dict*, *list* or *tuple* specifing the values of the control variables.

Returns numpy array of timesteps x: numpy array of state values

Return type t

Raises *ModelException*, *SolverException* –

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.

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 paramter if None

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!

as_network_model(*normalised: bool = False*)
Produces a bond graph `BondGraph.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.

property fluxes
The reaction fluxes.

A tuple (V,x) contain the vector $V(x)$ and the coordinates :math:'x_{i}` such that for the the stoichiometric matrix N and the reation rates $\kappa = extdiag(\kappa_1, \kappa_2, \ldots)$, the mass action description of the system is math:

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

property forward_stoichiometry
The forward stoichiometric matrix.

name
The name of this reaction network

property reverse_stoichiometry
The reverse stoichiometric matrix.

property species
A list of the chemical species invoved in this reaction network

property stoichiometry
The stoichiometric matrix

CHAPTER

SIX

BONDGRAPHTOOLS

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, scikit.odes and matplotlib

6.2 How to read the docs

The documentation for this library is organised into the following sections.

- This page show how to install BondGraphTools
- *Tutorials* contains a list of step-by-step tutorials demonstrating how to use BondGraphTools.
- *Discussion* contains high level discussion about to library.
- *API Reference* is the library reference documentation.

6.3 Getting Started

6.3.1 Requirements

- Python 3.7 or above.

6.3.2 Installation

1. Install Python 3.7 or greater.
2. Install Sundials and `scikits.odes` [dependencies](https://scikits-odes.readthedocs.io/en/stable/installation.html)
3. Install BondGraphTools via pip: `pip install BondGraphTools`

6.3.3 Usage

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

```
import BondGraphTools
```

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

```
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

[Gaw2007] https://ieeexplore.ieee.org/document/4140745

[Gaw1996] http://www.gawthrop.net/Books/GawSmi96.pdf

PYTHON MODULE INDEX

b

INDEX

I

L

M

N

P

R

S

T

U