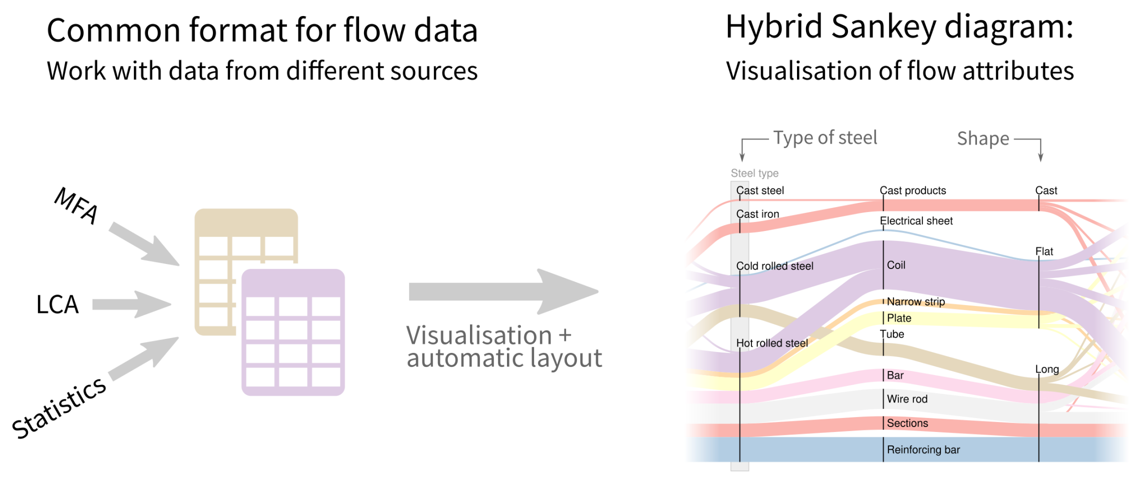
Sankey-view Documentation

Release 2.0.0a2

Rick Lupton

Jan 25, 2018

Contents



floWeaver generates Sankey diagrams from a dataset of flows. For a descriptive introduction, see the paper Hybrid Sankey diagrams: Visual analysis of multidimensional data for understanding resource use. For a more hands-on introduction, read on.

CHAPTER 1

Getting started

Note: You can try the tutorials online without installing anything! Click here to open MyBinder.

Start by installing floWeaver:

1.1 Installation

See below for more detailed instructions for Linux, Windows and OS X. In brief: install floweaver using pip:

```
$ pip install floweaver
```

If you use Jupyter notebooks – a good way to get started – you will also want to install ipysankeywidget, an IPython widget to interactively display Sankey diagrams:

```
$ pip install ipysankeywidget
$ jupyter nbextension enable --py --sys-prefix ipysankeywidget
```

Note: If this is the first time you have installed IPython widgets, you also need to make sure they are enabled:

```
$ jupyter nbextension enable --py --sys-prefix widgetsnbextension
```

If you use multiple virtualenvs or conda environments, make sure `ipywidgets` and `ipysankeywidget` are installed and enabled in both the environment running the notebook server and the kernel.

1.1.1 Install on Windows

Floweaver requries the latest version of Python to be installed. This can be done by installing the Anaconda platform from Link here .

The procedure described in section *Installation* should be performed in the Anaconda Prompt, which can be found among the installed programs.

To open Jupyter Notebook and begin to work on the Sankey. Write in the Anaconda Prompt the following

```
$ jupyter notebook
```

1.1.2 Install on macOS

Floweaver requries the latest version of Python to be installed. This can be done by installing the Anaconda platform from Link here .

The procedure described in section *Installation* should be performed in the Command Line

To open Jupyter Notebook and begin to work on the Sankey. Write in the Command Line the following

```
$ jupyter notebook
```

[not sure about this :D]

1.2 Changelog

1.2.1 v2.0.0 (renamed to floWeaver)

- sankeyview is now called floWeaver!
- There is a new top-level interface to creating a Sankey diagram, the `floweaver.weave()` function. This gives more flexibility about the appearance of the diagram, and lets you save the results in different formats (other than showing directly in the Jupyter notebook), while still being simple to use for the most common cases.
- No longer any need for `from sankeyview.jupyter import show_sankey`; use `floweaver.weave()` instead.
- New way to specify link colours using `floweaver.CategoricalScale` and `floweaver.QuantitativeScale`, replacing `hue` and related arguments to `show_sankey`. See *Colour-intensity scales* for examples.

Then the tutorials introduce the concepts used to generate and manipulate Sankey diagrams:

1.3 Quickstart tutorial

This tutorial will go through the basic ways to use `floweaver` to process and transform data into many different Sankey diagrams.

> If you are reading the static documentation, you can also try an interactive version of this tutorial online using MyBinder

Let's start by making a really simple dataset. Imagine we have some farms, which grow apples and bananas to sell to a few different customers. We can describe the *flow* of fruit from the farms (the *source* of the flow) to the customers (the *target* of the flow):

```
In [1]: import pandas as pd
        flows = pd.read_csv('simple_fruit_sales.csv')
        flows
```

```
Out[1]: source target     type  value
        0  farm1   Mary   apples      5
        1  farm1  James   apples      3
        2  farm2   Fred   apples     10
        3  farm2   Fred  bananas     10
        4  farm2  Susan  bananas      5
        5  farm3  Susan   apples     10
        6  farm4  Susan  bananas      1
        7  farm5  Susan  bananas      1
        8  farm6  Susan  bananas      1
```

Drawn directly as a Sankey diagram, this data would look something like this:

```
In [2]: from ipysankeywidget import SankeyWidget
        SankeyWidget(links=flows.to_dict('records'))
```

```
SankeyWidget(links=[{'target': 'Mary', 'value': 5, 'type': 'apples', 'source': 'farm1'}, {'target':
```

But you don't always want a direct correspondence between the flows in your data and the links that you see in the Sankey diagram. For example:

- Farms 4, 5 and 6 are all pretty small, and to make the diagram clearer we might want to group them in an "other" category.
- The flows of apples are mixed in with the flows of bananas – we might want to group the kinds of fruit together to make them easier to compare
- We might want to group farms or customers based on some other attributes – to see difference between genders, locations, or organic/non-organic farms, say.

This introduction shows how to use `floweaver` to do some of these for this simple example, in the simplest possible way. Later tutorials will show how to use it on real data, and more efficient ways to do the same things.

1.3.1 Basic diagram

Let's start with the first example: grouping farms 4, 5 and 6 into an "other" category. `floweaver` works by setting up a "Sankey diagram definition" which describes the structure of the diagram we want to see. In this case, we need to set up some groups:

```
In [3]: from floweaver import *

        # Set the default size to fit the documentation better.
        size = dict(width=570, height=300)

        nodes = {
            'farms': ProcessGroup(['farm1', 'farm2', 'farm3',
                                   'farm4', 'farm5', 'farm6']),
            'customers': ProcessGroup(['James', 'Mary', 'Fred', 'Susan']),
        }
```

We need to describe roughly how these groups should be placed in the final diagram by defining an "ordering" – a list of vertical slices, each containing a list of node ids:

```
In [4]: ordering = [
            ['farms'],       # put "farms" on the left...
            ['customers'],   # ... and "customers" on the right.
        ]
```

And we also need to say which connections should appear in the diagram (sometimes you don't want to actually see all the connections). This is called a "bundle" because it bundles up multiple flows – in this case all of them.

```
In [5]: bundles = [
            Bundle('farms', 'customers'),
        ]
```

Putting that together into a Sankey diagram definition (SDD) and applying it to the data gives this result:

```
In [6]: sdd = SankeyDefinition(nodes, bundles, ordering)
        weave(sdd, flows).to_widget(**size)
```

```
SankeyWidget(layout=Layout(height='300', width='570'), links=[{'color': '#FBB4AE', 'opacity': 1.0, 't
```

That's not very useful. What's happened? Every farm and every customer has been lumped together into one group. To get the picture we want – like the first one, but with an "other" group containing farms 4, 5 and 6 – we need to *partition* the groups:

```
In [7]: # The first argument is the dimension name -- for now we're using
        # "process" to group by process ids. The second argument is a list
        # of groups.
        farms_with_other = Partition.Simple('process', [
            'farm1',  # the groups within the partition can be a single id...
            'farm2',
            'farm3',
            ('other', ['farm4', 'farm5', 'farm6']),   # ... or a group
        ])

        # This is another partition.
        customers_by_name = Partition.Simple('process', [
            'James', 'Mary', 'Fred', 'Susan'
        ])

        # Update the ProcessGroup nodes to use the partitions
        nodes['farms'].partition = farms_with_other
        nodes['customers'].partition = customers_by_name

        # New Sankey!
        weave(sdd, flows).to_widget(**size)
```

```
SankeyWidget(groups=[{'id': 'farms', 'title': '', 'type': 'process', 'nodes': ['farms^farm1', 'farms
```

That's better: now the farms are split up appropriately with an "other" category, and the customers are shown separately as well. We don't have to stop there – what about showing sales to men and women?

```
In [8]: customers_by_gender = Partition.Simple('process', [
            ('Men', ['Fred', 'James']),
            ('Women', ['Susan', 'Mary']),
        ])

        nodes['customers'].partition = customers_by_gender

        weave(sdd, flows).to_widget(**size).auto_save_png('quickstart_example1.png')
```

```
SankeyWidget(groups=[{'id': 'farms', 'title': '', 'type': 'process', 'nodes': ['farms^farm1', 'farms
```

There is a better way of specifying this type of partition that doesn't involve hard-coding who's a man and who's a woman into the code – see the later tutorial on using *dimension tables*.

1.3.2 Distinguishing flow types

These diagrams have lost sight of the kind of fruit that is actually being sold – are the men buying apples, bananas or both from farm1? To show this we need to split up the flows in the diagram based on their *type*. Just like we split up the `ProcessGroups` by defining a partition of processes, we split up flows by defining a partition of flows.

(While we're at it let's choose some colours that look vaguely like apples and bananas)

```
In [9]: # Another partition -- but this time the dimension is the "type"
        # column of the flows table
        fruits_by_type = Partition.Simple('type', ['apples', 'bananas'])

        # Set the colours for the labels in the partition.
        palette = {'apples': 'yellowgreen', 'bananas': 'gold'}

        # New SDD with the flow_partition set
        sdd = SankeyDefinition(nodes, bundles, ordering,
                               flow_partition=fruits_by_type)

        weave(sdd, flows, palette=palette).to_widget(**size)
```

```
SankeyWidget(groups=[{'id': 'farms', 'title': '', 'type': 'process', 'nodes': ['farms^farm1', 'farms
```

As a last step, it would be nice to label which flows are apples and which are bananas. One way to do this would be to use a legend next to the diagram, or to put labels on every flow. Here, we'll add a new layer in the middle of the diagram which temporarily groups together the different fruit types on their way from the farms to the customers. This temporary/additional grouping point is called a *waypoint*.

To add a waypoint, we need to do three things:

1. Define it as a node
2. Position it in the ordering (between `farms` and `customers`)
3. Add it to the bundle

```
In [10]: # 1. Define a new waypoint node
         nodes['waypoint'] = Waypoint()

         # 2. Update the ordering to show where the waypoint goes: in the middle
         ordering = [
             ['farms'],
             ['waypoint'],
             ['customers'],
         ]

         # 3. Update the bundle definition to send the flows via the waypoint
         bundles = [
             Bundle('farms', 'customers', waypoints=['waypoint']),
         ]

         # Update the SDD with the new nodes, ordering & bundles.
         sdd = SankeyDefinition(nodes, bundles, ordering,
                                flow_partition=fruits_by_type)

         weave(sdd, flows, palette=palette).to_widget(**size)
```

```
SankeyWidget(groups=[{'id': 'farms', 'title': '', 'type': 'process', 'nodes': ['farms^farm1', 'farms
```

That's not yet very useful. Just like above, the default for Waypoints is to group everything togeter. We need to set a partition on the waypoint to split apart apples and bananas:

```
In [11]: # Redefine the waypoint with a partition (same one defined above)
         nodes['waypoint'] = Waypoint(fruits_by_type)

         weave(sdd, flows, palette=palette).to_widget(**size)
```

```
SankeyWidget(groups=[{'id': 'farms', 'title': '', 'type': 'process', 'nodes': ['farms^farm1', 'farms
```

1.3.3 Summary

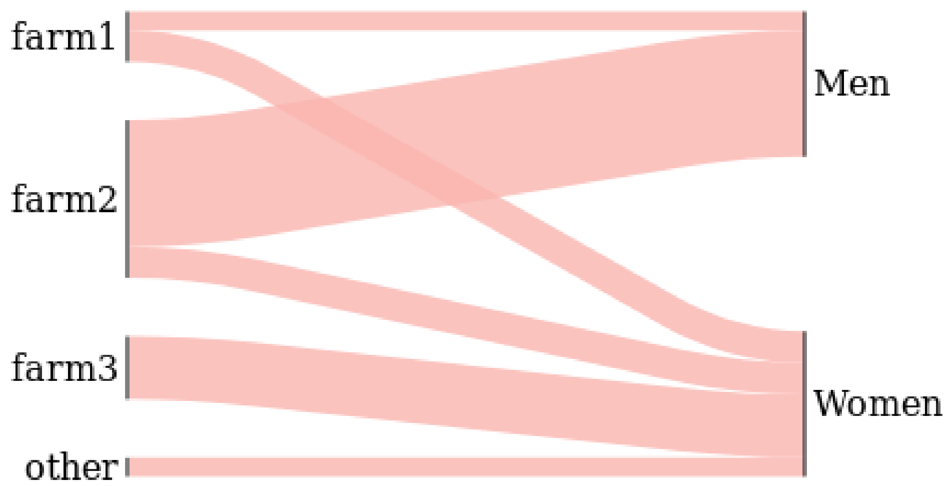
This has demonstrated the basic usage of floweaver: defining ProcessGroups, Waypoints, Partitions, and Bundles. If you are reading the interactive version, why not go back and try out some different ways to present the data? Here are some suggestions:

1. Farms 1, 3 and 5 are organic. Can you change the farm Partition to show two groups, organic and non-organic?
2. What happens if you remove "farm1" from the original definition of the farms ProcessGroup? (Hint: those apples that James and Mary are eating have to come from somewhere – so they are shown as coming from "elsewhere". See later tutorial on moving the system boundary)

If you are reading the static documentation, you can easily experiment with editing and rerunning this tutorial online using MyBinder, or download it to run on your computer from GitHub.

1.4 Dimension tables: efficiently adding details of processes and flows

In the Quickstart tutorial we saw how to draw some simple Sankey diagrams and partition them in different ways, such as this:



But to do the grouping on the right-hand side we had to explicitly list which people were "Men" and which were "Women", using a partition like this:

```
customers_by_gender = Partition.Simple('process', [
    ('Men', ['Fred', 'James']),
    ('Women', ['Susan', 'Mary']),
])
```

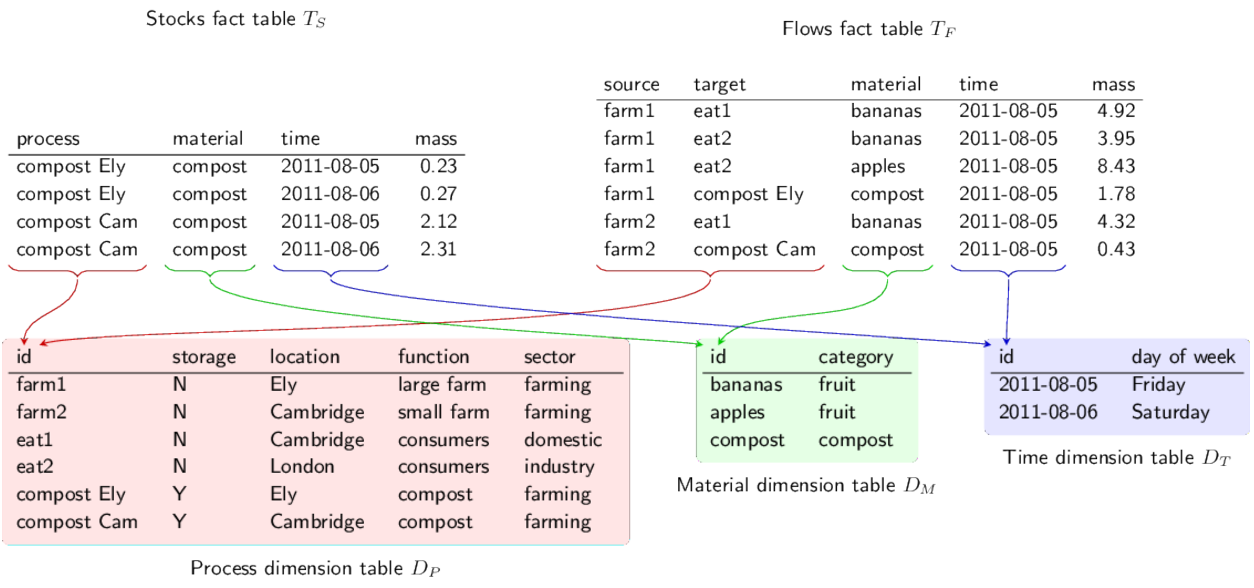
We can show this type of information more efficiently – and with less code – by using *dimension tables*.

1.4.1 Dimension tables

The table we've seen before is a **flow fact table** – it lists basic information about each flow:

- *source*: where the flow comes from
- *target*: where the flow goes to
- *type* or *material*: what is flowing
- *value*: the size (in tonnes, GJ, £ etc) of the flow

An example of this type of table is shown at the top right of this diagram:



Stocks fact table T_S

process	material	time	mass
compost Ely	compost	2011-08-05	0.23
compost Ely	compost	2011-08-06	0.27
compost Cam	compost	2011-08-05	2.12
compost Cam	compost	2011-08-06	2.31

Flows fact table T_F

source	target	material	time	mass
farm1	eat1	bananas	2011-08-05	4.92
farm1	eat2	bananas	2011-08-05	3.95
farm1	eat2	apples	2011-08-05	8.43
farm1	compost Ely	compost	2011-08-05	1.78
farm2	eat1	bananas	2011-08-05	4.32
farm2	compost Cam	compost	2011-08-05	0.43

id	storage	location	function	sector
farm1	N	Ely	large farm	farming
farm2	N	Cambridge	small farm	farming
eat1	N	Cambridge	consumers	domestic
eat2	N	London	consumers	industry
compost Ely	Y	Ely	compost	farming
compost Cam	Y	Cambridge	compost	farming

Process dimension table D_P

id	category
bananas	fruit
apples	fruit
compost	compost

Material dimension table D_M

id	day of week
2011-08-05	Friday
2011-08-06	Saturday

Time dimension table D_T

The **dimension tables** add extra information about the source/target and type of the flows (the diagram above also shows extra information about the time period the flow relates to, but we're not worrying about time in this tutorial). For example, "farm2" has a *location* attribute set to "Cambridge".

This tutorial will show how to use dimension tables in floweaver.

```
In [1]: # Load the same data used in the quickstart tutorial
        import pandas as pd
        flows = pd.read_csv('simple_fruit_sales.csv')
        flows
```

```
Out[1]: source target      type  value
        0   farm1    Mary   apples      5
        1   farm1   James   apples      3
        2   farm2    Fred   apples     10
        3   farm2    Fred  bananas     10
        4   farm2   Susan  bananas      5
        5   farm3   Susan   apples     10
        6   farm4   Susan  bananas      1
        7   farm5   Susan  bananas      1
        8   farm6   Susan  bananas      1
```

```
In [2]: # Load another table giving extra information about the
        # farms and customers. `index_col` says the first column
        # can be used to lookup rows.
        processes = pd.read_csv('simple_fruit_sales_processes.csv',
                                 index_col=0)
        processes
```

```
Out[2]: type   location organic    sex
        id
        farm1      farm     Barton     yes    NaN
        farm2      farm     Barton     yes    NaN
        farm3      farm        Ely      no    NaN
        farm4      farm        Ely     yes    NaN
        farm5      farm    Duxford      no    NaN
        farm6      farm     Milton     yes    NaN
        Mary   customer  Cambridge     NaN  Women
        James  customer     Milton     NaN    Men
        Fred   customer  Cambridge     NaN  Women
        Susan  customer  Cambridge     NaN    Men
```

Each `id` in this table matches a `source` or `target` in the flows table above. We can use this extra information to build the Sankey.

```
In [3]: # Setup
        from floweaver import *

        # Set the default size to fit the documentation better.
        size = dict(width=570, height=300)
```

Because we now have two tables (before we only had one so didn't have to worry) we must put them together into a Dataset:

```
In [4]: dataset = Dataset(flows, dim_process=processes)
```

Now we can use the `type` column in the process table to more easily pick out the relevant processes:

```
In [5]: nodes = {
            'farms': ProcessGroup('type == "farm"'),
            'customers': ProcessGroup('type == "customer"'),
        }
```

Compare this to how the same thing was written in the Quickstart:

```
nodes = {
    'farms': ProcessGroup(['farm1', 'farm2', 'farm3',
                           'farm4', 'farm5', 'farm6']),
    'customers': ProcessGroup(['James', 'Mary', 'Fred', 'Susan']),
}
```

Because we already know from the process dimension table that James, Mary, Fred and Susan are "customers", we don't have to list them all by name in the ProcessGroup definition – we can write the *query* `type == "customer"` instead.

Note: See the API Documentation for *`floweaver.ProcessGroup`* for more details.

The rest of the Sankey diagram definition is the same as before:

```
In [6]: ordering = [
            ['farms'],        # put "farms" on the left...
            ['customers'],    # ... and "customers" on the right.
        ]
        bundles = [
            Bundle('farms', 'customers'),
        ]
        sdd = SankeyDefinition(nodes, bundles, ordering)
        weave(sdd, dataset).to_widget(**size)
```

```
SankeyWidget(layout=Layout(height='300', width='570'), links=[{'type': '*', 'time': '*', 'target': 'c
```

Again, we need to set the partition on the ProcessGroups to see something interesting. Here again, we can use the process dimension table to make this easier:

```
In [7]: # Create a Partition which splits based on the `sex` column
        # of the dimension table
        customers_by_gender = Partition.Simple('process.sex',
                                               ['Men', 'Women'])

        nodes['customers'].partition = customers_by_gender
        weave(sdd, dataset).to_widget(**size)
```

```
SankeyWidget(groups=[{'type': 'process', 'id': 'customers', 'title': '', 'nodes': ['customers^Men',
```

For reference, this is what we wrote before in the Quickstart:

```
customers_by_gender = Partition.Simple('process', [
    ('Men', ['Fred', 'James']),
    ('Women', ['Susan', 'Mary']),
])
```

And we can use other columns of the dimension table to set other partitions:

```
In [8]: farms_by_organic = Partition.Simple('process.organic', ['yes', 'no'])

        nodes['farms'].partition = farms_by_organic
        weave(sdd, dataset).to_widget(**size)
```

```
SankeyWidget(groups=[{'type': 'process', 'id': 'farms', 'title': '', 'nodes': ['farms^yes', 'farms^no
```

Finally, a tip for doing quick exploration of the data with partitions: you can automatically get a Partition which includes all the values that actually occur in your dataset using the `dataset.partition` method:

```
In [9]: # This is the logical thing to write but
        # it doesn't actually work at the moment :(
        # nodes['farms'].partition = dataset.partition('process.organic')

        # It works with 'source.organic'... we can explain later
        nodes['farms'].partition = dataset.partition('source.organic')

        # This should be the same as before
        weave(sdd, dataset).to_widget(**size)
```

```
SankeyWidget(groups=[{'type': 'process', 'id': 'farms', 'title': '', 'nodes': ['farms^yes', 'farms^no
```

1.4.2 Summary

The process dimension table adds extra information about each process. You can use this extra information to:

1. Pick out the processes you want to include in a ProcessGroup (selection); and
2. Split apart groups of processes based on different attributes (partitions).

Things to try:

- Make a diagram showing the locations of farms on the left and the locations of customers on the right

1.5 System boundaries

Often we don't want to show all of the data in one Sankey diagram: you focus on one part of the system. But we still want conservation of mass (or whatever is being shown in the diagram) to work, so we end up with flows to & from "elsewhere". These can also be thought of as *imports* and *exports*.

Let's start by recreating the Quickstart example:

```
In [1]: import pandas as pd
        flows = pd.read_csv('simple_fruit_sales.csv')
```

```
In [2]: from floweaver import *

        # Set the default size to fit the documentation better.
        size = dict(width=570, height=300)

        # Same partitions as the Quickstart tutorial
        farms_with_other = Partition.Simple('process', [
            'farm1',
            'farm2',
            'farm3',
            ('other', ['farm4', 'farm5', 'farm6']),
        ])

        customers_by_name = Partition.Simple('process', [
            'James', 'Mary', 'Fred', 'Susan'
        ])

        # Define the nodes, this time setting the partition from the start
        nodes = {
            'farms': ProcessGroup(['farm1', 'farm2', 'farm3',
                                   'farm4', 'farm5', 'farm6'],
                                  partition=farms_with_other),
            'customers': ProcessGroup(['James', 'Mary', 'Fred', 'Susan'],
                                      partition=customers_by_name),
        }

        # Ordering and bundles as before
        ordering = [
            ['farms'],       # put "farms" on the left...
            ['customers'],   # ... and "customers" on the right.
        ]

        bundles = [
            Bundle('farms', 'customers'),
        ]
```

```
In [3]: sdd = SankeyDefinition(nodes, bundles, ordering)
        weave(sdd, flows).to_widget(**size)
```

```
SankeyWidget(groups=[{'nodes': ['farms^farm1', 'farms^farm2', 'farms^farm3', 'farms^other'], 'id': '
```

What happens if we remove `farm2` from the ProcessGroup?

```
In [4]: nodes['farms'].selection = [
            'farm1', 'farm3', 'farm4', 'farm5', 'farm6'
        ]
        weave(sdd, flows).to_widget(**size)
```

```
SankeyWidget(groups=[{'nodes': ['farms^farm1', 'farms^farm3', 'farms^other'], 'id': 'farms', 'type':
```

The flow is still there! But it is labelled with a little arrow to show that it is coming "from elsewhere". This is important because we are still showing Susan and Fred in the diagram, and they get fruit from farm2. If we didn't show those flows, Susan's and Fred's inputs and outputs would not balance.

Try now removing Susan and Fred from the diagram:

```
In [5]: nodes['customers'].selection = ['James', 'Mary']
        weave(sdd, flows).to_widget(**size)
```

```
SankeyWidget(groups=[{'nodes': ['farms^farm1', 'farms^farm3', 'farms^other'], 'id': 'farms', 'type':
```

Now they're gone, we no longer see the incoming flows from `farm2`. But we see some outgoing flows "to elsewhere" from `farm3` and the `other` group. This is because `farm3` is within the system boundary – it is shown in the diagram – so its output flow has to go somewhere.

1.5.1 Controlling Elsewhere flows

These flows are added automatically to make sure that mass is conserved, but because they are automatic, we have little control over them. By explicitly adding a flow to or from Elsewhere to the diagram, we can control where they appear and what they look like.

To do this, add a Waypoint for the outgoing flows to 'pass through' on their way across the system boundary:

```
In [6]: # Define a new Waypoint
        nodes['exports'] = Waypoint(title='exports here')

        # Update the ordering to include the waypoint
        ordering = [
            ['farms'],                    #     put "farms" on the left...
            ['customers', 'exports'],     # ... and "exports" below "customers"
        ]                                 #     on the right.

        # Add a new bundle from "farms" to Elsewhere, via the waypoint
        bundles = [
            Bundle('farms', 'customers'),
            Bundle('farms', Elsewhere, waypoints=['exports']),
        ]

        sdd = SankeyDefinition(nodes, bundles, ordering)
        weave(sdd, flows).to_widget(**size)
```

```
SankeyWidget(groups=[{'nodes': ['farms^farm1', 'farms^farm3', 'farms^other'], 'id': 'farms', 'type':
```

This is pretty similar to what we had already, but now the waypoint is explicitly listed as part of the `SankeyDefinition`, we have more control over it.

For example, we can put the exports above James and Mary by changing the ordering:

```
In [7]: ordering = [
            ['farms'],
            ['exports', 'customers'],
        ]
        sdd = SankeyDefinition(nodes, bundles, ordering)
        weave(sdd, flows).to_widget(**size)
```

```
SankeyWidget(groups=[{'nodes': ['farms^farm1', 'farms^farm3', 'farms^other'], 'id': 'farms', 'type':
```

Or we can partition the exports Waypoint to show how much of it is apples and bananas:

```
In [8]: fruits_by_type = Partition.Simple('type', ['apples', 'bananas'])
        nodes['exports'].partition = fruits_by_type
        weave(sdd, flows).to_widget(**size)
```

```
SankeyWidget(groups=[{'nodes': ['farms^farm1', 'farms^farm3', 'farms^other'], 'id': 'farms', 'type':
```

1.5.2 Horizontal bands

Often, import/exports and loss flows are shown in a separate horizontal "band" either above or below the main flows. We can do this by modifying the `ordering` a little bit.

The `ordering` style we have used so far looks like this:

```
ordering = [
    [list of nodes in layer 1],  # left-hand side
    [list of nodes in layer 2],
    ...
    [list of nodes in layer N],  # right-hand side
]
```

But we can add another layer of nesting to make it look like this:

```
ordering = [
    # |top band|  |bottom band|
    [ [........], [...........] ],  # left-hand side
    [ [........], [...........] ],
    ...
    [ [........], [...........] ],  # right-hand side
]
```

Here's an example:

```
In [9]: ordering = [
            [[],            ['farms'    ]],
            [['exports'], ['customers']],
        ]
        sdd = SankeyDefinition(nodes, bundles, ordering)
        weave(sdd, flows).to_widget(**size)
```

```
SankeyWidget(groups=[{'nodes': ['farms^farm1', 'farms^farm3', 'farms^other'], 'id': 'farms', 'type':
```

1.5.3 Summary

- All the flows to/from a ProcessGroup are shown, even if the other end of the flow is outside the system boundary (i.e. not part of any ProcessGroup)
- You can control the automatic flows by explicitly adding Bundles to/from `Elsewhere` with a `Waypoint`
- The `ordering` can contain horizontal bands

1.6 Colour-intensity scales

In this tutorial we will look at how to use colours in the Sankey diagram. We have already seen how to use a palette, but in this tutorial we will also create a Sankey where the intensity of the colour is proportional to a numerical value.

First step is to import all the requried packages and data:

```
In [1]: import pandas as pd
        import numpy as np
        from floweaver import *
```

```
df1 = pd.read_csv('holiday_data.csv')
```

Now take a look at the dataset we are using. This is a very insightful [made-up] dataset about how differnt types of people lose weight while on holiday enjoying themselves.

```
In [2]: dataset = Dataset(df1)
        df1
```

```
Out[2]: source          target  Calories Burnt  Enjoyment Employment Job  \
        0   Activity  Employment Job             2.5         35          Student
        1   Activity  Employment Job             4.5         20          Student
        2   Activity  Employment Job             8.0          5          Student
        3   Activity  Employment Job             1.0          5          Student
        4   Activity  Employment Job             8.0         30          Student
        5   Activity  Employment Job             1.0         35          Trainee
        6   Activity  Employment Job             3.0         40          Trainee
        7   Activity  Employment Job             2.0         40          Trainee
        8   Activity  Employment Job             6.0          5          Trainee
        9   Activity  Employment Job            12.0         45          Trainee
        10  Activity  Employment Job             4.5         20   Administrator
        11  Activity  Employment Job             9.0         10   Administrator
        12  Activity  Employment Job             7.5         50   Administrator
        13  Activity  Employment Job             1.5         35   Administrator
        14  Activity  Employment Job             1.5         50   Administrator
        15  Activity  Employment Job            11.0         55          Manager
        16  Activity  Employment Job             2.0         45          Manager
        17  Activity  Employment Job             7.5         10          Manager
        18  Activity  Employment Job             1.5         90          Manager
        19  Activity  Employment Job             2.0         40          Manager
        20  Activity  Employment Job             3.0         35        Pensioner
        21  Activity  Employment Job             9.0         15        Pensioner
        22  Activity  Employment Job             9.0         15        Pensioner
        23  Activity  Employment Job             3.0         60        Pensioner
        24  Activity  Employment Job             0.0          0        Pensioner

               Activity
        0        Reading
        1       Swimming
        2       Sleeping
        3     Travelling
        4    Working out
        5        Reading
        6     Travelling
        7       Swimming
        8       Sleeping
        9    Working out
        10      Swimming
        11      Sleeping
        12   Working out
        13       Reading
        14    Travelling
        15   Working out
        16       Reading
        17      Sleeping
        18    Travelling
        19      Swimming
        20       Reading
        21      Swimming
        22      Sleeping
```

```
23   Travelling
24  Working out
```

We now define the partitions of the data. Rather than listing the categories by hand, we use `np.unique` to pick out a list of the unique values that occur in the dataset.

```
In [3]: partition_job = Partition.Simple('Employment Job', np.unique(df1['Employment Job']))
        partition_activity = Partition.Simple('Activity', np.unique(df1['Activity']))
```

In fact, this is pretty common so there is a built-in function to do this:

```
In [4]: # these statements or the ones above do the same thing
        partition_job = dataset.partition('Employment Job')
        partition_activity = dataset.partition('Activity')
```

We then go on to define the structure of our sankey. We define nodes, bundles and the order. In this case its pretty straightforward:

```
In [5]: nodes = {
            'Activity': ProcessGroup(['Activity'], partition_activity),
            'Job': ProcessGroup(['Employment Job'], partition_job),
        }

        bundles = [
            Bundle('Activity', 'Job'),
        ]

        ordering = [
            ['Activity'],
            ['Job'],
        ]
```

Now we will plot a Sankey that shows the share of time dedicated to each activity by each type of person.

```
In [6]: # These are the same each time, so just write them here once
        size_options = dict(width=500, height=400,
                            margins=dict(left=100, right=100))

        sdd = SankeyDefinition(nodes, bundles, ordering)
        weave(sdd, dataset, measures='Calories Burnt').to_widget(**size_options)
```

```
SankeyWidget(groups=[{'title': '', 'nodes': ['Activity^Reading', 'Activity^Swimming', 'Activity^Slee
```

We can start using colour by specifying that we want to partition the flows according to type of person. Notice that this time we are using a pre-determined palette.

You can find all sorts of palettes listed here.

```
In [7]: sdd = SankeyDefinition(nodes, bundles, ordering, flow_partition=partition_job)

        weave(sdd, dataset, palette='Set2_8', measures='Calories Burnt').to_widget(**size_options)
```

```
SankeyWidget(groups=[{'title': '', 'nodes': ['Activity^Reading', 'Activity^Swimming', 'Activity^Slee
```

Now, if we want to make the colour of the flow to be proprtional to a numerical value. Use the `hue` parameter to set the name of the variable that you want to display in colour. To start off, let's use "value", which is the width of the lines: wider lines will be shown in a darker colour.

```
In [8]: weave(sdd, dataset, link_color=QuantitativeScale('Calories Burnt'), measures='Calories Burnt
```

```
SankeyWidget(groups=[{'title': '', 'nodes': ['Activity^Reading', 'Activity^Swimming', 'Activity^Slee
```

It's more interesting to use colour to show a different attribute from the flow table. But because a line in the Sankey diagram is an aggregation of multiple flows in the original data, we need to specify how the new dimension will be aggregated. For example, we'll use the *mean* of the flows within each Sankey link to set the colour. In this case we will use the colour to show how much each type of person emjoys each activity. We can be interested in either the cumulative enjoyment, or the mean enjoyment: try both!

Aggregation is specified with the `ameasures` parameter, which should be set to a dictionary mapping dimension names to aggregation functions (`'mean'`, `'sum'` etc).

```
In [9]: weave(sdd, dataset, measures={'Calories Burnt': 'sum', 'Enjoyment': 'mean'}, link_width='Calo
              link_color=QuantitativeScale('Enjoyment')).to_widget(**size_options)
```

```
SankeyWidget(groups=[{'title': '', 'nodes': ['Activity^Reading', 'Activity^Swimming', 'Activity^Sleep
```

```
In [10]: weave(sdd, dataset, measures={'Calories Burnt': 'sum', 'Enjoyment': 'mean'}, link_width='Cal
               link_color=QuantitativeScale('Enjoyment', intensity='Calories Burnt')).to_widget(**siz
```

```
/home/rick/ownCloud/devel/sankey-view/floweaver/color_scales.py:114: RuntimeWarning: invalid value en
  value /= measures[self.intensity]
```

```
SankeyWidget(groups=[{'title': '', 'nodes': ['Activity^Reading', 'Activity^Swimming', 'Activity^Sleep
```

You can change the colour palette using the `palette` attribute. The palette names are different from before, because those were *categorical* (or *qualitative*) scales, and this is now a *sequential* scale. The palette names are listed here.

```
In [11]: scale = QuantitativeScale('Enjoyment', palette='Blues_9')
         weave(sdd, dataset,
               measures={'Calories Burnt': 'sum', 'Enjoyment': 'mean'},
               link_width='Calories Burnt',
               link_color=scale) \
             .to_widget(**size_options)
```

```
SankeyWidget(groups=[{'title': '', 'nodes': ['Activity^Reading', 'Activity^Swimming', 'Activity^Sleep
```

```
In [12]: scale.domain
```

```
Out[12]: (0, 90)
```

It is possible to create a colorbar / scale to show the range of intensity values, but it's not currently as easy as it should be. This should be improved in future.

CHAPTER 2

API Documentation

2.1 Datasets

class Dataset(*flows*, *dim_process=None*, *dim_material=None*, *dim_time=None*)

2.2 Sankey diagram definitions

Sankey diagram definitions (SDDs) describe the structure of the Sankey diagram you want to end up with. They are *declarative*: you declare what you want up front, but the diagram isn't created until later. This is useful if you want to use the same diagram structure for different data sources.

class SankeyDefinition(*nodes*, *bundles*, *ordering*, *flow_selection=None*, *flow_partition=None*, *time_partition=None*)

class ProcessGroup(*selection=None*, *partition=None*, *direction='R'*, *title=None*)

A ProcessGroup represents a group of processes from the underlying dataset.

The processes to include are defined by the *selection*. By default they are all lumped into one node in the diagram, but by defining a *partition* this can be controlled.

selection
list or string – If a list of strings, they are taken as process ids. If a single string, it is taken as a Pandas query string run against the process table.

partition
Partition, optional – Defines how to split the ProcessGroup into subgroups.

direction
'R' or 'L' – Direction of flow, default 'R' (left-to-right).

title
string, optional – Label for the ProcessGroup. If not set, the ProcessGroup id will be used.

class Waypoint(*partition=None*, *direction='R'*, *title=None*)

A Waypoint represents a control point along a `Bundle` of flows.

There are two reasons to define Waypoints: to control the routing of `Bundle` s of flows through the diagram, and to split flows according to some attributes by setting a *partition*.

`partition`
Partition, optional – Defines how to split the Waypoint into subgroups.

`direction`
'R' or 'L' – Direction of flow, default 'R' (left-to-right).

`title`
string, optional – Label for the Waypoint. If not set, the Waypoint id will be used.

`class Bundle`(*source*, *target*, *waypoints=NOTHING*, *flow_selection=None*, *flow_partition=None*, *default_partition=None*)
A Bundle represents a set of flows between two :class:'ProcessGroup's.

`source`
string – The id of the `ProcessGroup` at the start of the Bundle.

`target`
string – The id of the `ProcessGroup` at the end of the Bundle.

`waypoints`
list of strings – Optional list of ids of :class:'Waypoint's the Bundle should pass through.

`flow_selection`
string, optional – Query string to filter the flows included in this Bundle.

`flow_partition`
Partition, optional – Defines how to split the flows in the Bundle into sub-flows. Often you want the same Partition for all the Bundles in the diagram, see `SankeyDefinition.flow_partition`.

`default_partition`
Partition, optional – Defines the Partition applied to any Waypoints automatically added to route the Bundle across layers of the diagram.

2.3 Weaving the Sankey diagram

The `weave()` function actually creates a Sankey diagram from the *Sankey diagram definitions* and a *Datasets*.

`weave`(*sankey_definition*, *dataset*, *measures='value'*, *link_width=None*, *link_color=None*, *palette=None*)

CHAPTER 3

Contributing

Contributions are very welcome.

3.1 Contributing to floWeaver

Contributions are welcome! Please get in touch via email or creating a GitHub issue with any questions.

3.1.1 Documentation

These are draft guidelines for getting started contributing to the documentation on Windows. Improvements are welcome, or get in touch if you need better instructions.

1. *Required software*: Anaconda, Github Desktop App.
   (a) **Install pandoc package.**
      i. Clone Github Repository using the following URL: https://github.com/ricklupton/floweaver.git
2. *Modify Content*. The content is kept in the `/docs` directory. Each page is saved as a text file formatted in reStructured Text.
3. *Save Modifications*. To save the changes made to the content, open the Anaconda Prompt, go to the `/floweaver/docs` directory and run

```
make.bat html
```

CHAPTER 4

Citing floweaver

If floweaver has been significant in a project that leads to a publication, please acknowledge that by citing the paper linked above:

18. C. Lupton and J. M. Allwood, ‘Hybrid Sankey diagrams: Visual analysis of multidimensional data for understanding resource use’, Resources, Conservation and Recycling, vol. 124, pp. 141–151, Sep. 2017. DOI: 10.1016/j.resconrec.2017.05.002

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Index