River Network Analysis

Release 2019

Bram van Meurs

Nov 23, 2019

BASIC PARTS:

Welcome to the documentation of the code belonging to my master thesis. In this thesis I have used Muskingum routing principles and applied them to a network structure. Documenting is a ongoing process and will emphasize on the developed scripts.

CHAPTER

ONE

LITERATURE AND PRINCIPLES

There is plenty of literature available on Muskingum routing. Here is a short list:

- For a comprehensive overview of flood routing and other hydrology concepts the National Engineering Handbook Hydrology gives a good overview. Chapter 17, section channel flood routing methods explains the Muskingum methods.
- For the full derivation of the muskingum equation see Todini 2007
- This is a very clear and practical example by University of Colorado Boulder

This documentation is divided into three sections. In the first section the basic parts or building blocks of the model are presented and analysed. In the second section small networks are tested and analysed. This section also explains some concepts on how to calculate flows in these networks. The third section shows some code to extract the Ganges Brahmaputra watershed from the Hydrosheds dataset. (Code not well documented yet)

1.1 Single Reach

In this notebook the effect of different parameter settings of x and k is shown.

Note: This is written in old code and will be replaced later using the network model class. The other parts already use the network model class.

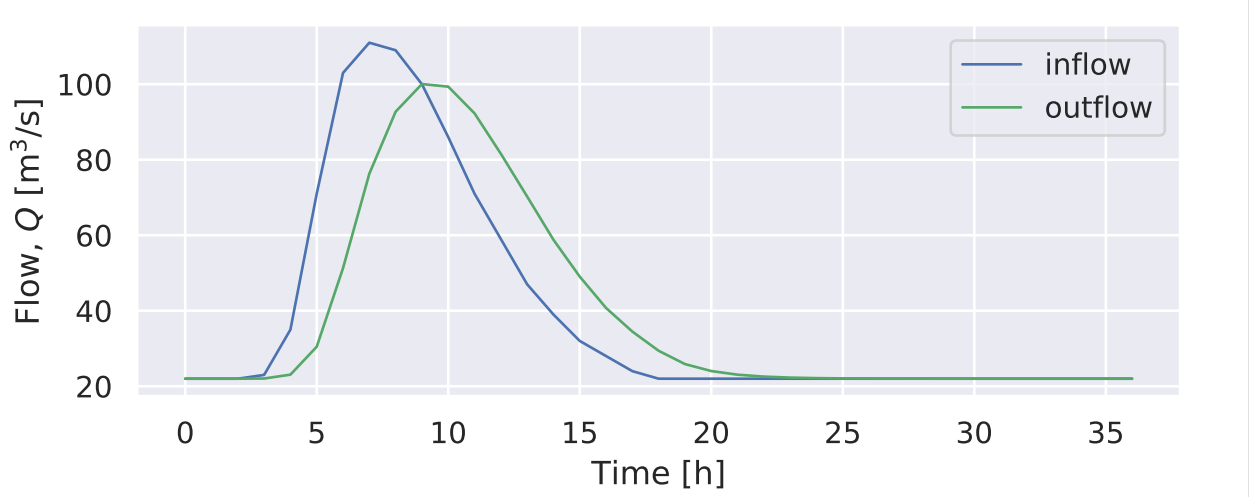
1.1.1 Initialising model

[1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

[2]:
```
from context import fit_muskingum
from fit_muskingum import getParams
from fit_muskingum import calc_Out
from fit_muskingum import calc_C
```

[3]:
```
df = pd.read_excel('../data/example-inflow-karahan-adjusted.xlsx')
df = df.set_index('Time')
```

[4]:
```
t = df.index.values
I = np.array(df['Inflow'])
fig = plt.figure(figsize=(7,2.5),dpi=150)
fig.patch.set_alpha(0)


ax = fig.add_subplot(111)
for axis in ['top','bottom','left','right']:
    ax.spines[axis].set_linewidth(0.5)

plt.plot(t,I,linewidth = 1 , label = 'inflow')
plt.rcParams.update({'font.size': 8, 'pgf.rcfonts' : False})

x = 0.2
k = 2
dt = 1

C0 = calc_C(k,x,dt) # k,x,dt
O0 = calc_Out(I,C0)
plt.plot(t, O0 ,'g',linewidth = 1, label = 'outflow')

plt.ylabel('Flow, $Q$ [m$^3$/s]')
plt.xlabel('Time [h]')
plt.legend();
# save to file
#plt.savefig('../thesis/report/figs/1reach.pdf', bbox_inches = 'tight')
#plt.savefig('../thesis/report/figs/1reach.pgf', bbox_inches = 'tight')
```



The blue line is the inflow to the reach. The reach has parameters $x = 0.2$, $k = 2$ and $\Delta t = 1$ The resulting outflow is shown in green.

1.1.2 Understanding k

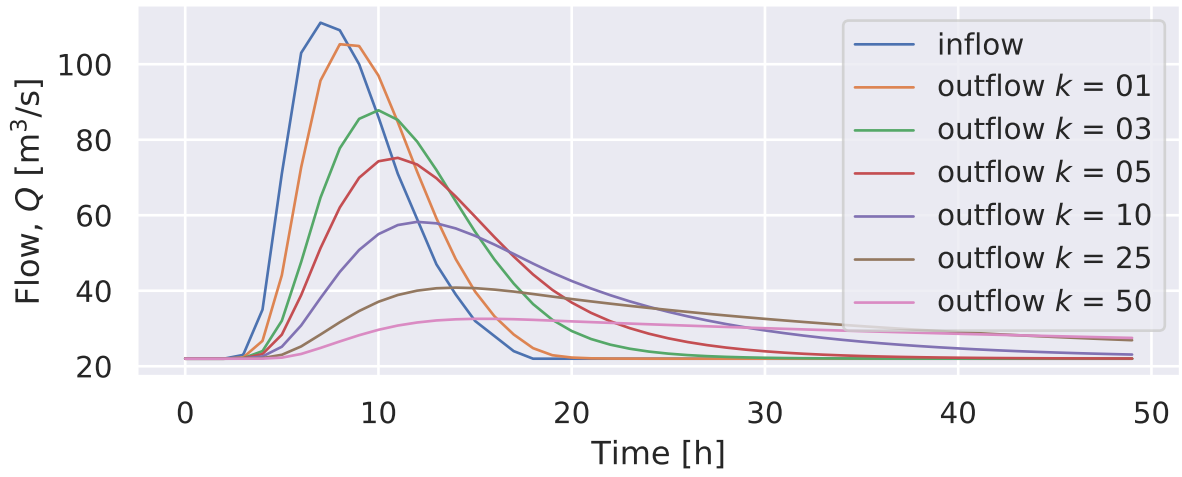
To understand what happens the effect is of k, it is varied while keeping x constant. x is fixed to 0.01 while k takes the values: 1, 3, 5, 10, 25, 50. Again Δt is set to 1.

[5]:
```
t = df.index.values
I = np.array(df['Inflow'])
```

(continues on next page)

(continued from previous page)

```
length = 50
t = range(0,length,1)
I = np.append(I,np.full((1,length - len(I)),22))

fig = plt.figure(figsize=(7,2.5),dpi=150)
fig.patch.set_alpha(0)
ax = fig.add_subplot(111)
for axis in ['top','bottom','left','right']:
    ax.spines[axis].set_linewidth(0.5)
plt.rcParams.update({'font.size': 8, 'pgf.rcfonts' : False})

plt.plot(t,I,linewidth = 1 , label = 'inflow')

klist = [1,3,5,10,25,50]
for k in klist:
    x = 0.01
    dt = 1
    out = calc_Out(I,calc_C(k,x,dt))
    plt.plot(t, out,linewidth = 1, label = 'outflow $k$ = ' + '{:02d}'.format(k))

plt.ylabel('Flow, $Q$ [m$^3$/s]')
plt.xlabel('Time [h]')
plt.legend();
```



It is clear that k is related to the delay or lag of the peak. The peaks shift to the right with increasing k. While the peaks shift, also the attenuation increases. Meanwhile, flow the total volume passed by, the area under the graph, remains the same.

1.1.3 Understanding x

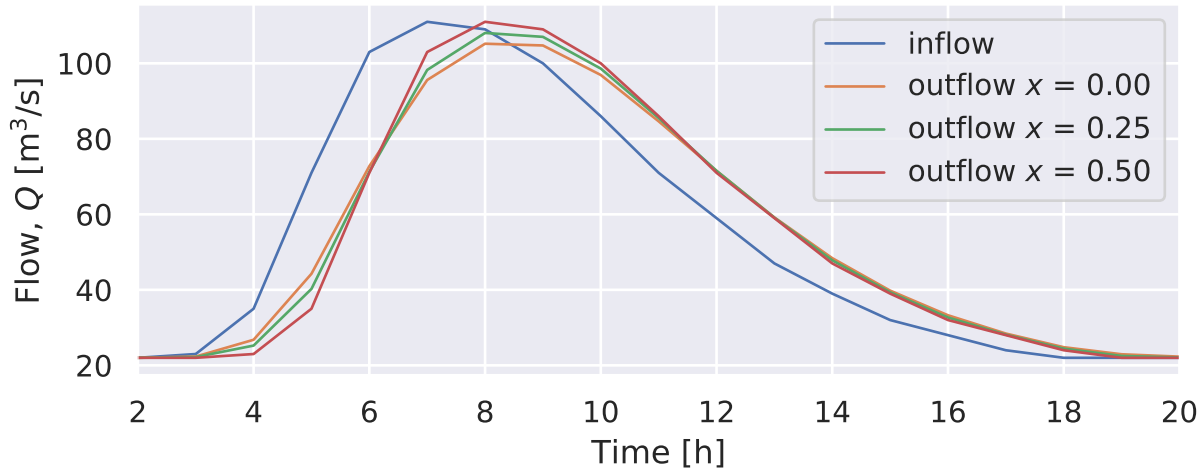
In the following section we do the same for x. It will take the values: 0, 0.25, 0.5. Both k and Δt are kept constant at 1

[6]:
```
t = df.index.values
I = np.array(df['Inflow'])
```

(continues on next page)

(continued from previous page)

```
fig = plt.figure(figsize=(7,2.5),dpi=150)
fig.patch.set_alpha(0)
ax = fig.add_subplot(111)
for axis in ['top','bottom','left','right']:
    ax.spines[axis].set_linewidth(0.5)
plt.rcParams.update({'font.size': 8, 'pgf.rcfonts' : False})

plt.plot(t,I,linewidth = 1 , label = 'inflow')

for x in [0,0.25,0.5]:
    k = 1
    dt = 1
    out = calc_Out(I,calc_C(k,x,dt))
    plt.plot(t, out,linewidth = 1, label = 'outflow $x$ = ' + '{:1.2f}'.format(x))


plt.ylabel('Flow, $Q$ [m$^3$/s]')
plt.xlabel('Time [h]')
plt.legend()
plt.xlim(2,20);
```



As a result we can see that the x behaves as the attenuation parameter. All graphs have the peak at the same timestep, so no shift in time has occurred. What differs is the height of each peak. For $x = 0.5$ no attenuation occurs, while for $x = 0$ maximum attenuation occurs.

1.2 Single Confluence

[1]:
```
import pandas as pd
import networkx as nx
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
```

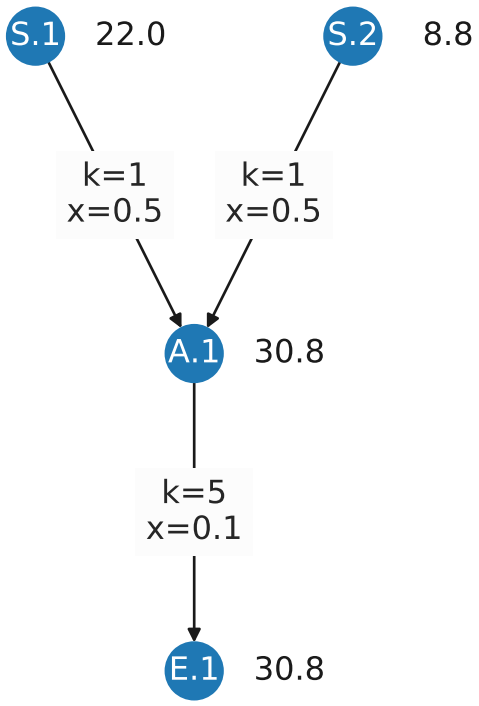
[2]:
```
from context import RiverNetwork
from RiverNetwork import RiverNetwork
```

1.2.1 Loading network structure

[3]:
```
structure1 = RiverNetwork('../data/single-confluence.xlsx')
```

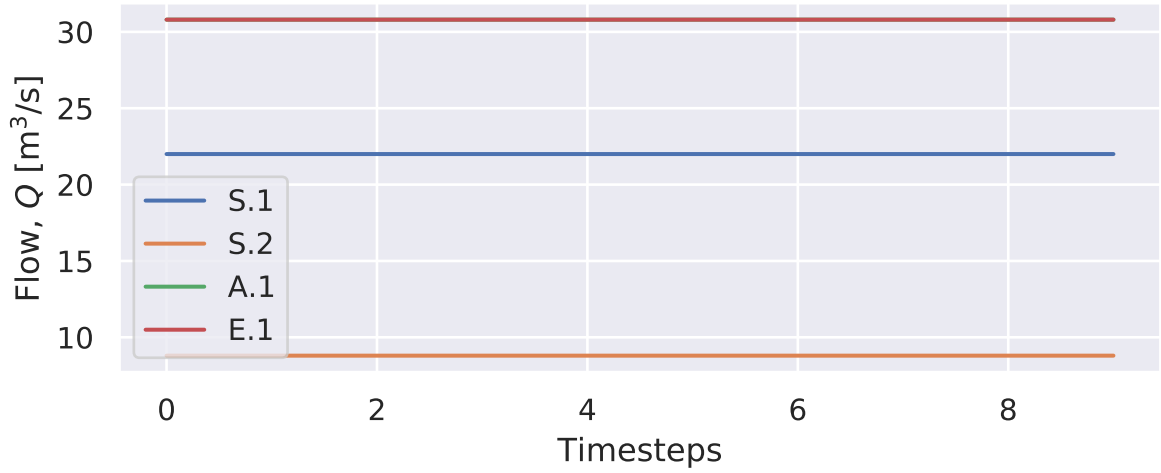
[4]:
```
structure1.draw(figsize=(4,4))
```

```
/home/docs/checkouts/readthedocs.org/user_builds/rna/envs/latest/lib/python3.7/
↪site-packages/networkx/drawing/nx_pylab.py:579: MatplotlibDeprecationWarning:
The iterable function was deprecated in Matplotlib 3.1 and will be removed in 3.3.␣
↪Use np.iterable instead.
  if not cb.iterable(width):
/home/docs/checkouts/readthedocs.org/user_builds/rna/envs/latest/lib/python3.7/
↪site-packages/networkx/drawing/nx_pylab.py:676: MatplotlibDeprecationWarning:
The iterable function was deprecated in Matplotlib 3.1 and will be removed in 3.3.␣
↪Use np.iterable instead.
  if cb.iterable(node_size):  # many node sizes
```



Here we see the network structure as defined with its nodes and edges. Each edge shows its corresponding k and x. The incoming reaches have an x of 0.5 such that only a delay occurs and no attenuation. The numbers next to the nodes show the base loads, which is a static flow based on a long term average or can be an initial value. These base loads can also be plotted:

[5]:
```
structure1.draw_base_loads(figsize=(7,2.5))
```



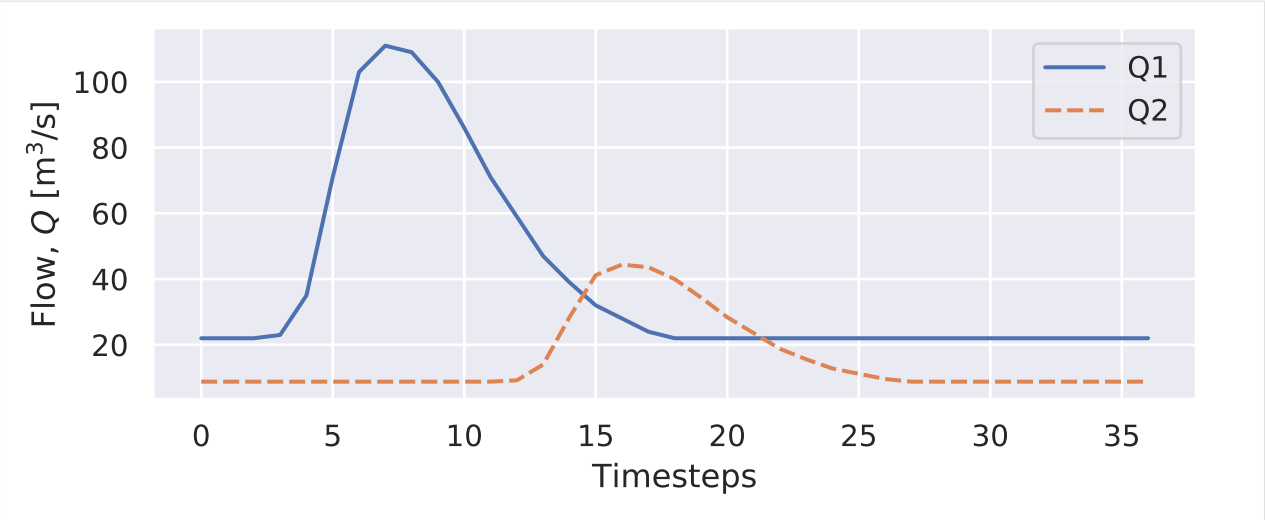
1.2.2 Setting the inflows

We load a basic waveform from an excel file. Then this wave is translated to create a second waveform.

[6]:
```
I = pd.read_excel('../data/example-inflow-karahan-adjusted.xlsx').Inflow
t = pd.read_excel('../data/example-inflow-karahan-adjusted.xlsx').Time
I2 = I*0.4
I2 = np.append(I2[28:37],I2[0:28])
inflow = pd.DataFrame({'Q1':I,'Q2':I2},index=t)
```

The inflows are plotted:

[7]:
```
fig = plt.figure(figsize=(7,2.5),dpi=150)
fig.patch.set_alpha(0)
ax = fig.add_subplot(111)
for axis in ['top','bottom','left','right']:
    ax.spines[axis].set_linewidth(0.5)
plt.rcParams.update({'font.size': 8, 'pgf.rcfonts' : False})

sns.lineplot(data = inflow);
plt.ylabel('Flow, $Q$ [m$^3$/s]');
plt.xlabel('Timesteps');
```



Then the inflows are set to the sourcenodes

[8]:
```
structure1.set_shape('S.1',36,I - min(I))
structure1.set_shape('S.2',36,I2 - min(I2))
```

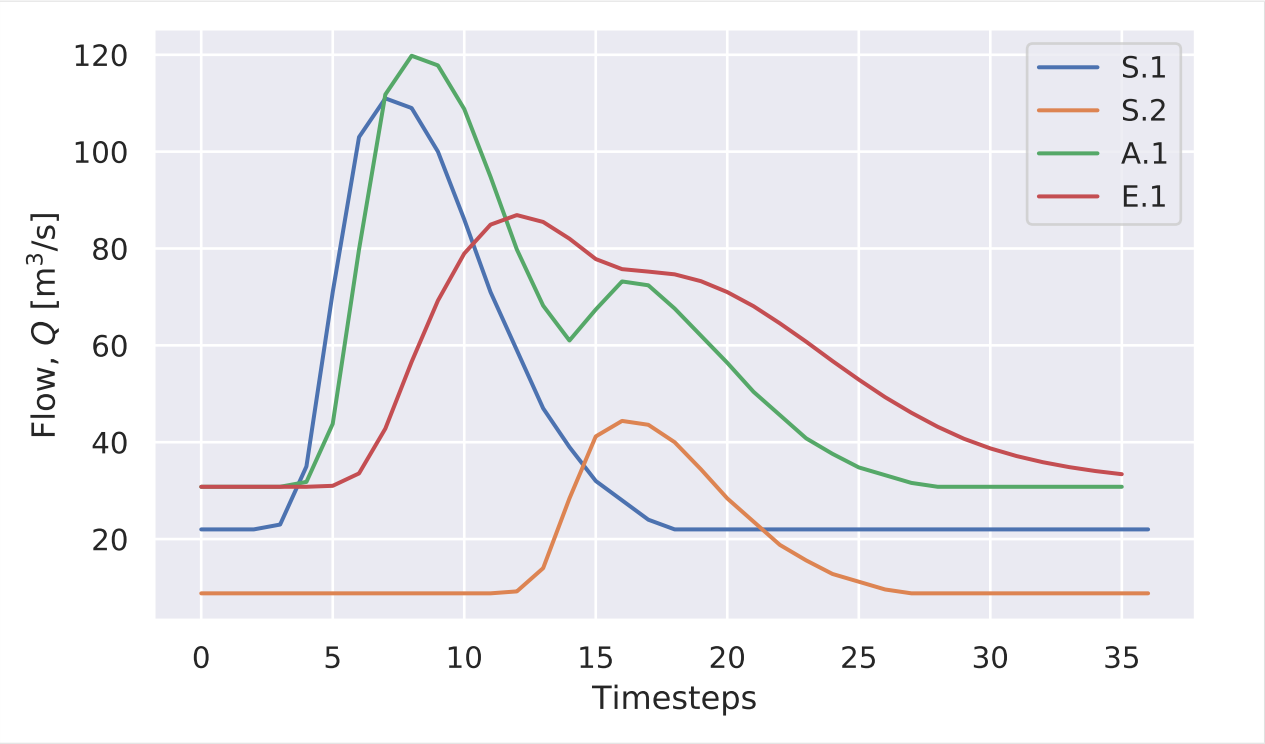
Note: The minimum flow is subtracted from the input because set_shape adds flow relative to the defined baseload. This behaviour might change in the future.

1.2.3 Calculating wave propagation

[9]:
```
structure1.calc_flow_propagation(36)
```

[10]:
```
structure1.draw_Qin(figsize=(7,4))
```

[10]:
```
(<Figure size 672x384 with 1 Axes>,
 <matplotlib.axes._subplots.AxesSubplot at 0x7fbe062a72e8>)
```



In the graph we can see that A.1 is a superposition of S.1 and S.2. and is shifted one timestep to the right. E.1 the outflow of the last reach is than a muskingum transformation of A.1 with $k = 5$ and $x = 0.1$. This behaviour is as expected.

1.3 Single Bifurcation

[1]:
```
import pandas as pd
import networkx as nx
import matplotlib.pyplot as plt
import numpy as np
```

[2]:
```
from context import RiverNetwork
from RiverNetwork import RiverNetwork
```

1.3.1 Loading network structure

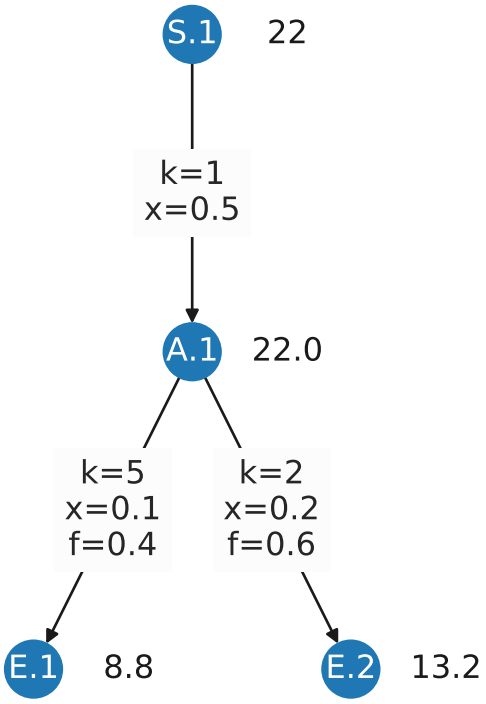
[3]:
```
structure1 = RiverNetwork('../data/single-bifurcation.xlsx')
structure1.draw(figsize=(4,4))
```

```
/home/docs/checkouts/readthedocs.org/user_builds/rna/envs/latest/lib/python3.7/
↪site-packages/networkx/drawing/nx_pylab.py:579: MatplotlibDeprecationWarning:
The iterable function was deprecated in Matplotlib 3.1 and will be removed in 3.3.␣
↪Use np.iterable instead.
  if not cb.iterable(width):
/home/docs/checkouts/readthedocs.org/user_builds/rna/envs/latest/lib/python3.7/
↪site-packages/networkx/drawing/nx_pylab.py:676: MatplotlibDeprecationWarning:
The iterable function was deprecated in Matplotlib 3.1 and will be removed in 3.3.␣
↪Use np.iterable instead.
```

(continues on next page)

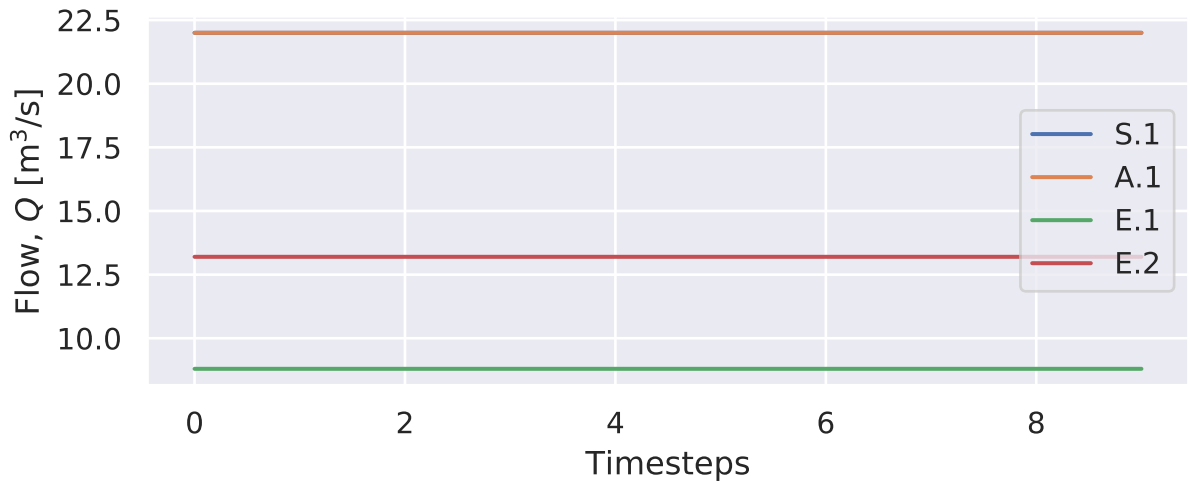
(continued from previous page)

```
if cb.iterable(node_size):  # many node sizes
```



Here we see the network with a single bifurcation. Each reach after such bifurcation has another parameter: f. This parameter determines the (volume) fraction of water flowing into each reach, which is considered static. In this case 40% to the left and 60% to the right. This can also be seen in the base loads.

[4]:
```
structure1.draw_base_loads(figsize=(7,2.5))
```



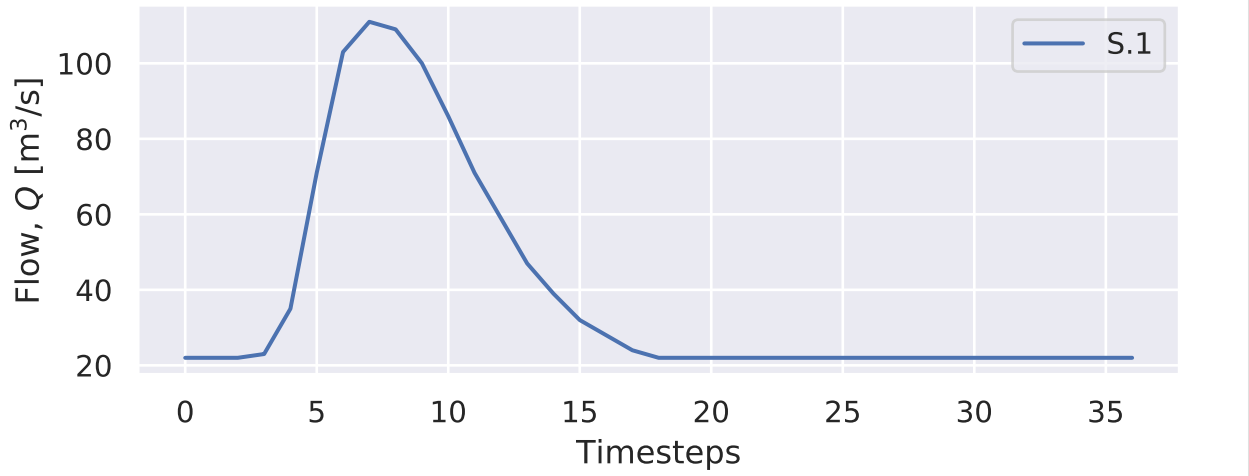
1.3.2 Setting the inflow

The same inflow is set to the single source node

[5]:
```
inflow = np.array(pd.read_excel('../data/example-inflow-karahan-adjusted.xlsx').
↪Inflow)-22
```

[6]:
```
structure1.set_shape('S.1',36,inflow)
structure1.draw_Qin(figsize=(7,2.5))
```

[6]:
```
(<Figure size 672x240 with 1 Axes>,
 <matplotlib.axes._subplots.AxesSubplot at 0x7f61a6a89438>)
```

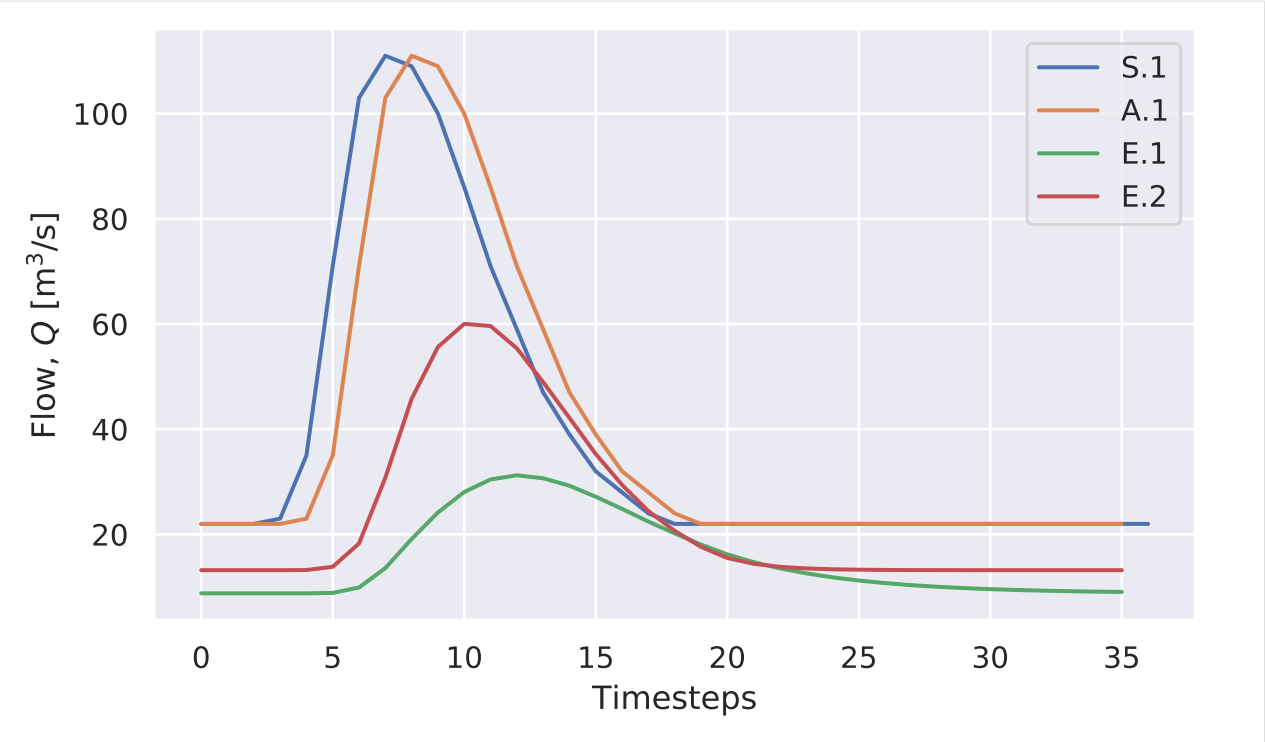


1.3.3 Calculating wave propagation

The same steps are repeated to see calculate the resulting flows

[7]:
```
structure1.calc_flow_propagation(36)
structure1.draw_Qin(figsize=(7,4))
```

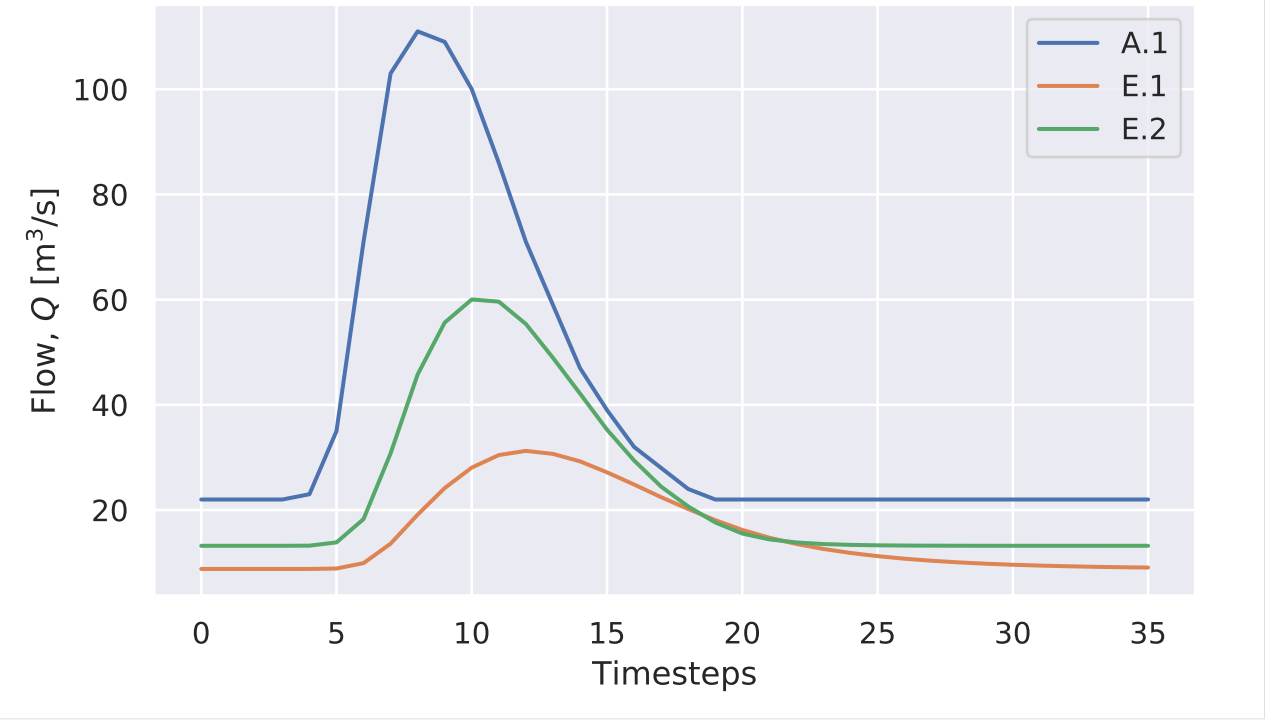
[7]:
```
(<Figure size 672x384 with 1 Axes>,
 <matplotlib.axes._subplots.AxesSubplot at 0x7f61a6910470>)
```



A.1 is a single timestep shift from S.1 ($x = 0.5$, $k = 1$). For clarity S.1 is omitted from the figure:

[8]:
```
structure1.draw_Qin(figsize=(7,4),no='S.1')
```

[8]:
```
(<Figure size 672x384 with 1 Axes>,
 <matplotlib.axes._subplots.AxesSubplot at 0x7f61a6a4b630>)
```



Here we clearly see how the flow through A.1 is divided into two flows. Both flows have a different lag and attenuation. In the first few timesteps it is also clear that the distribution is 60-40.

1.4 Model verification

In this notebook the model is verified by comparing outcomes of two different examples.

[1]:
```
import pandas as pd
import networkx as nx
import matplotlib.pyplot as plt
import numpy as np
```

[2]:
```
from context import RiverNetwork
from RiverNetwork import RiverNetwork
```

[3]:
```
%load_ext autoreload
%autoreload 2
```

1.4.1 University of Colorado Boulder example

This is the most straightforward example that I could find. It is a simple one segment model. k and x are estimated to 2.3h and 0.151559154 respectively.

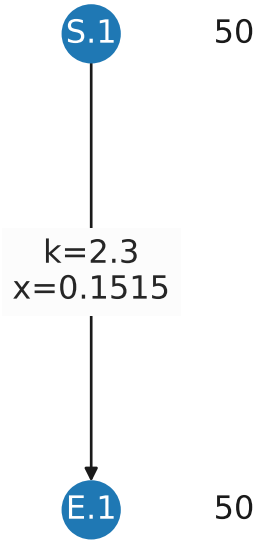
http://www.engr.colostate.edu/~ramirez/ce_old/classes/cive322-Ramirez/CE322_Web/Example_MuskingumRouting.htm

Loading data

The structure is loaded as well as the inflow. On the graph we can see the correct k and x

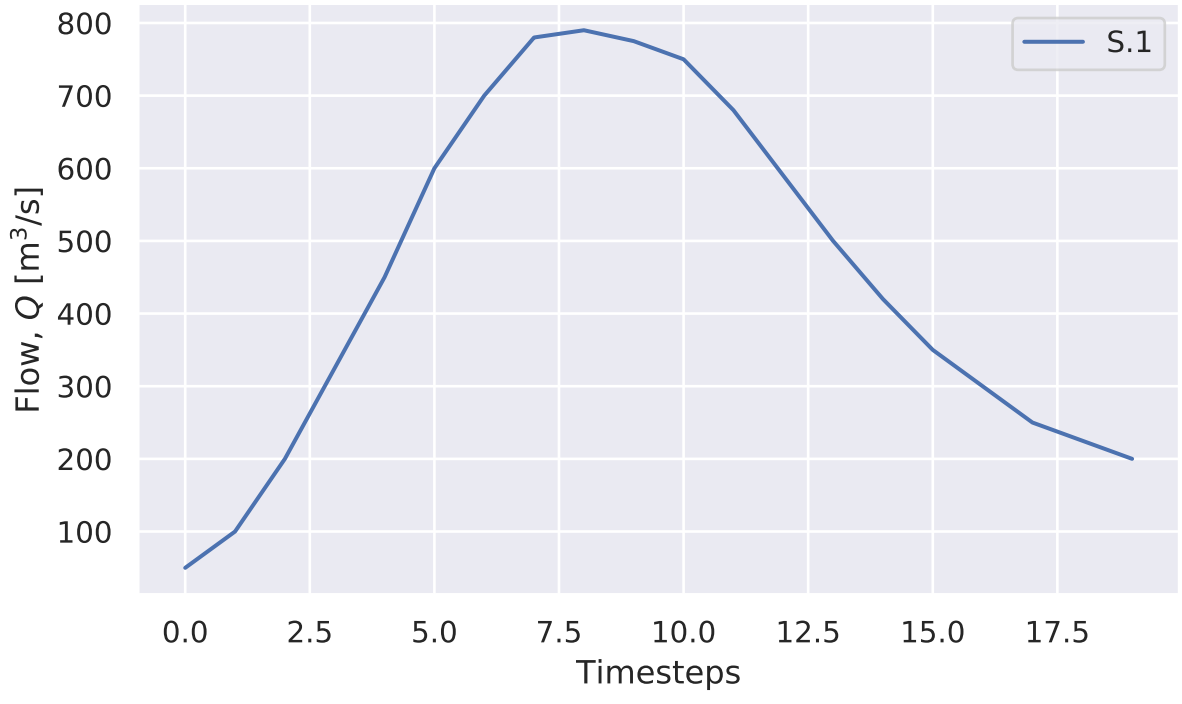
[4]:
```
structure1 = RiverNetwork('../data/single-segment-boulder.xlsx')
structure1.draw(figsize=(3,3))
```

```
/home/docs/checkouts/readthedocs.org/user_builds/rna/envs/latest/lib/python3.7/
↪site-packages/networkx/drawing/nx_pylab.py:579: MatplotlibDeprecationWarning:
The iterable function was deprecated in Matplotlib 3.1 and will be removed in 3.3.␣
↪Use np.iterable instead.
  if not cb.iterable(width):
/home/docs/checkouts/readthedocs.org/user_builds/rna/envs/latest/lib/python3.7/
↪site-packages/networkx/drawing/nx_pylab.py:676: MatplotlibDeprecationWarning:
The iterable function was deprecated in Matplotlib 3.1 and will be removed in 3.3.␣
↪Use np.iterable instead.
  if cb.iterable(node_size):  # many node sizes
```



[5]:
```
inflow = np.array(pd.read_excel('../data/example-inflow-boulder.xlsx').Inflow)
structure1.set_shape('S.1',21,inflow-50)
structure1.draw_Qin(only_sources=True,figsize=(7,4))
```

[5]:
```
(<Figure size 672x384 with 1 Axes>,
 <matplotlib.axes._subplots.AxesSubplot at 0x7f6a110e6a58>)
```

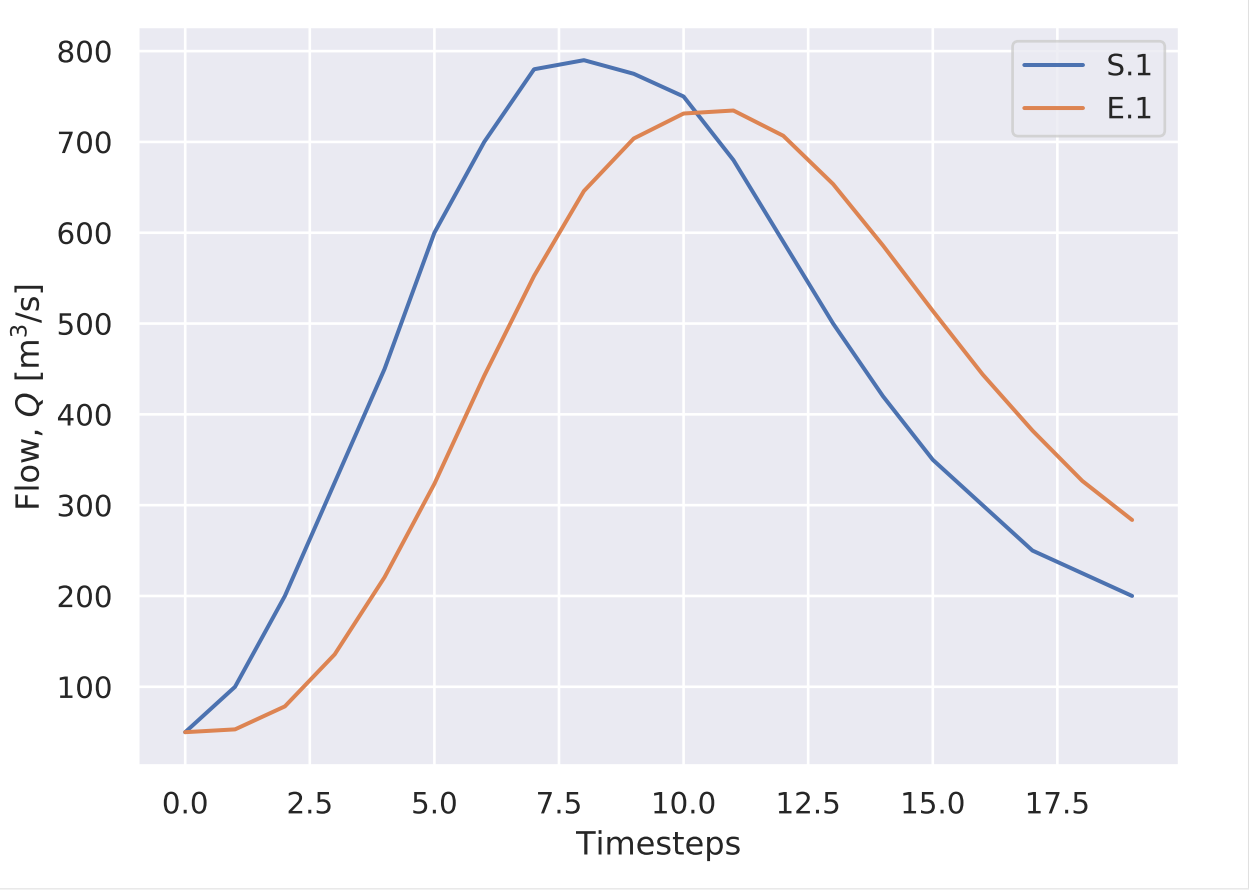


Results of flow propagation

The flow is calculated for the sink node.

[6]:
```
structure1.calc_flow_propagation(20)
structure1.draw_Qin(figsize=(7,5))
```

[6]:
```
(<Figure size 672x480 with 1 Axes>,
 <matplotlib.axes._subplots.AxesSubplot at 0x7f6a110533c8>)
```



Inflow for node S.1

[7]:
```
print(structure1.get_Graph().nodes['S.1']['Qin'])
```

```
[ 50 100 200 325 450 600 700 780 790 775 750 680 590 500 420 350 300 250
 225 200]
```

Inflow/outflow for sink E.1

[8]:
```
print(structure1.get_Graph().nodes['E.1']['Qin'])
```

```
[ 50.          53.09090167  78.40717029 135.73277234 220.66311342
 323.48352768 442.45783876 552.45606895 645.89060264 703.74630509
 731.26560713 734.58020111 706.75303009 653.56523866 585.97973616
 513.94847587 443.98213789 382.16017306 326.70590002 283.67408341]
```

These results are then compared to the results of the webpage. The figure shows that the results are almost similar. The small differences in outflow can be explained by rounding errors.

Time (h)	Inflow (m3/s)	C_o x I_{i+1} (m3/s)	C_1 x I_i (m3/s)	C_2 x O_i (m3/s)	Outflow (m3/s)
1	50				50
2	100	6.1787	17.3037	29.60695	53.08935
3	200	12.3574	34.6074	31.43627	78.40107
4	325	20.08078	69.2148	46.42433	135.7199
5	450	27.80415	112.4741	80.36505	220.6433
6	600	37.0722	155.7333	130.6515	323.457
7	700	43.2509	207.6444	191.5315	442.4268
8	780	48.19386	242.2518	261.9782	552.4238
9	790	48.81173	269.9377	327.1117	645.8611
10	775	47.88493	273.3985	382.4396	703.723
11	750	46.34025	268.2074	416.7018	731.2494
12	680	42.01516	259.5555	433.0013	734.572
13	590	36.45433	235.3303	434.9687	706.7534
14	500	30.8935	204.1837	418.4962	653.5734
15	420	25.95054	173.037	387.0063	585.9938
16	350	21.62545	145.3511	346.9898	513.9663
17	300	18.5361	121.1259	304.3395	444.0015
18	250	15.44675	103.8222	262.9106	382.1796
19	225	13.90208	86.5185	226.3034	326.724
20	200	12.3574	77.86665	193.466	283.6901

1.4.2 Karahan example

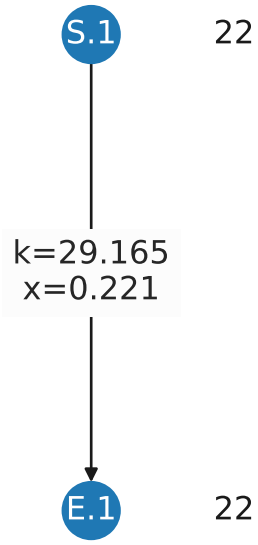
In this example the Wilson dataset from the Karahan paper is used. Karahan compared different estimation techniques. It is a simple one segment model. The most interesting difference here is that Δt is not 1 but karahan uses a value of 6. For this dataset the x is estimated on 0.021 and k to 29.165. This can also be seen in the figure. The base load is set to 22.

https://onlinelibrary.wiley.com/doi/full/10.1002/cae.20394

Loading data

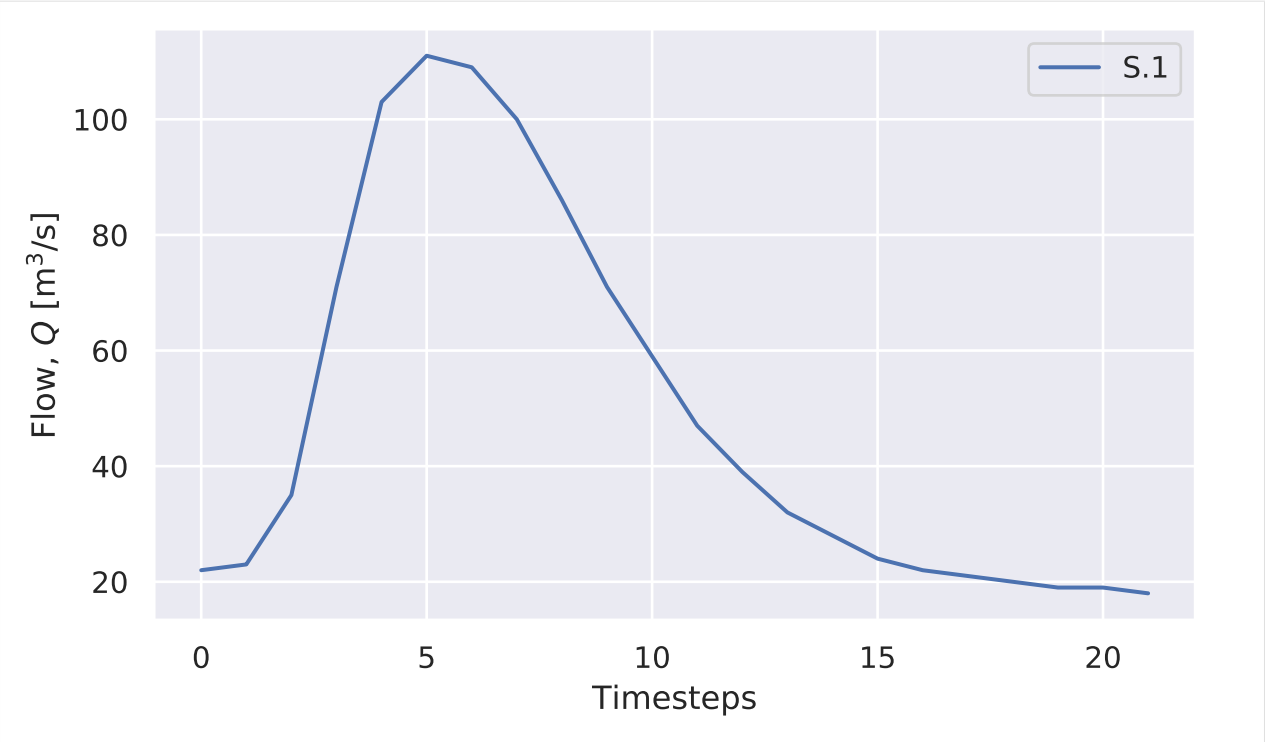
[9]:
```
structure2 = RiverNetwork('../data/single-segment-karahan.xlsx',dt=6)
structure2.draw(figsize=(3,3))
```

```
/home/docs/checkouts/readthedocs.org/user_builds/rna/envs/latest/lib/python3.7/
↪site-packages/networkx/drawing/nx_pylab.py:579: MatplotlibDeprecationWarning:
The iterable function was deprecated in Matplotlib 3.1 and will be removed in 3.3.␣
↪Use np.iterable instead.
  if not cb.iterable(width):
/home/docs/checkouts/readthedocs.org/user_builds/rna/envs/latest/lib/python3.7/
↪site-packages/networkx/drawing/nx_pylab.py:676: MatplotlibDeprecationWarning:
The iterable function was deprecated in Matplotlib 3.1 and will be removed in 3.3.␣
↪Use np.iterable instead.
  if cb.iterable(node_size):  # many node sizes
```



[10]:
```
inflow = np.array(pd.read_excel('../data/example-inflow-karahan.xlsx').Inflow)
structure2.set_shape('S.1',21,inflow-22)
structure2.draw_Qin(only_sources=True,figsize=(7,4))
```

[10]:
```
(<Figure size 672x384 with 1 Axes>,
 <matplotlib.axes._subplots.AxesSubplot at 0x7f6a10eb22e8>)
```

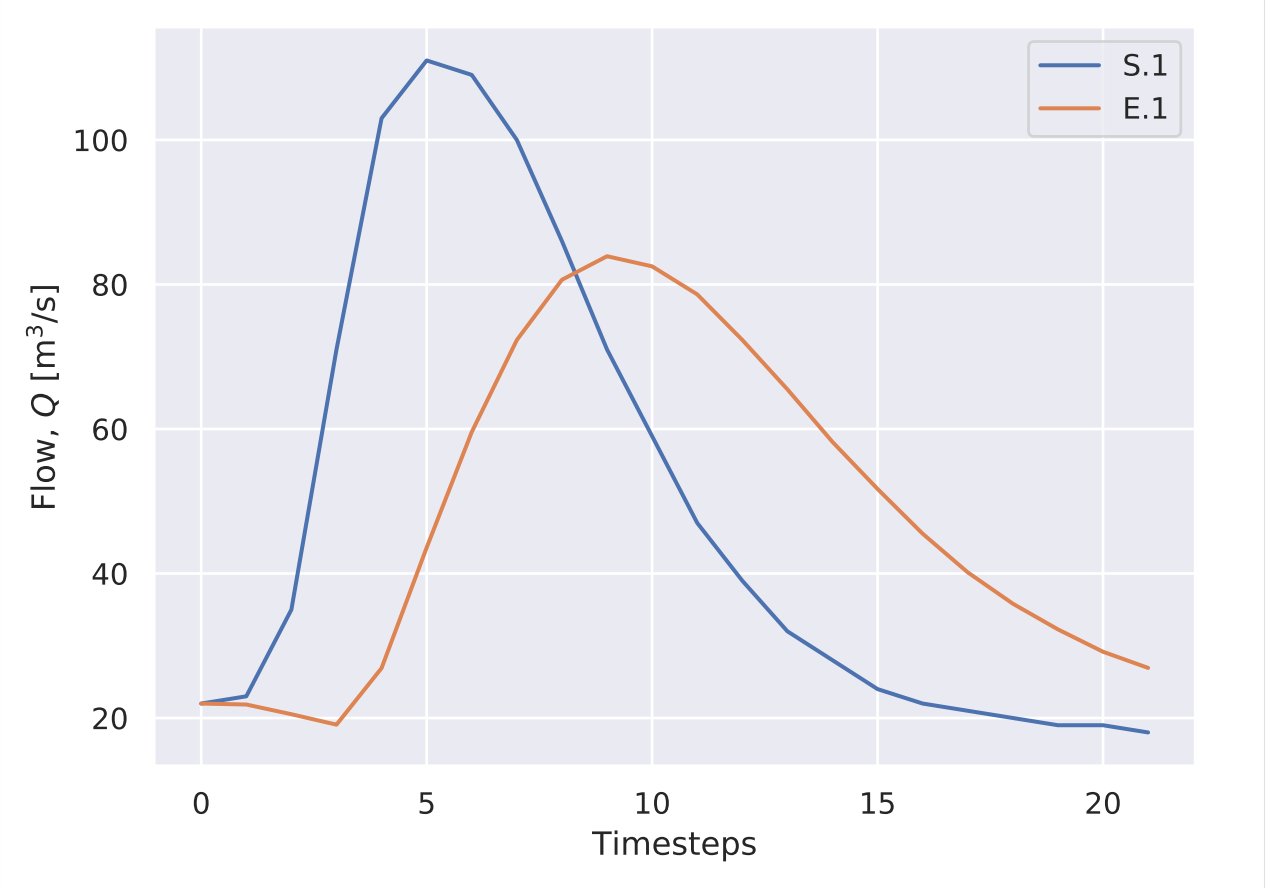


Results of flow propagation

The flow is calculated for the sink node.

[11]:
```
structure2.calc_flow_propagation(22)
structure2.draw_Qin(figsize=(7,5))
```

[11]:
```
(<Figure size 672x480 with 1 Axes>,
 <matplotlib.axes._subplots.AxesSubplot at 0x7f6a1182ec18>)
```



The input and output data is shown:

Inflow for node S.1

```
[12]: structure2.get_Graph().nodes['S.1']['Qin']
```

```
[12]: array([ 22,  23,  35,  71, 103, 111, 109, 100,  86,  71,  59,  47,  39,
              32,  28,  24,  22,  21,  20,  19,  19,  18])
```

Inflow/outflow for node E.1

```
[13]: structure2.get_Graph().nodes['E.1']['Qin']
```

```
[13]: array([22.        , 21.86603704, 20.52301889, 19.07762521, 26.90355864,
             43.58406737, 59.57916704, 72.31400781, 80.64823559, 83.90617013,
             82.50290056, 78.6275652 , 72.32100985, 65.4854351 , 58.20961341,
             51.69799411, 45.50437379, 40.1551022 , 35.82045353, 32.26373075,
             29.16949189, 26.93105771])
```

These results are compared to the results of Karahan. And as can be seen in the following figure, the output of the model is the same for Procedure II.

Table 2 Observed and Computed Values of Outflow for Data Set Given by Wilson [16]

			Computed O (m^3/s)				
			Ref. [18]			This study	
Time (h)	I (m^3/s)	O (m^3/s)	LSM	Method I	Method II	Procedure I	Procedure II
0	22.0	22.0	22.0	22.0	22.0	22.00	22.00
6	23.0	21.0	21.8	21.8	21.7	21.86	21.87
12	35.0	21.0	20.0	19.5	18.9	20.45	20.52
18	71.0	26.0	17.5	16.4	12.0	18.76	19.07
24	103.0	34.0	24.9	27.1	12.5	26.30	26.90
30	111.0	44.0	42.4	50.2	23.8	42.81	43.58
36	109.0	55.0	59.3	70.6	37.1	58.76	59.58
42	100.0	66.0	72.9	85.3	49.9	71.57	72.32
48	86.0	75.0	81.8	93.3	60.9	80.06	80.65
54	71.0	82.0	85.4	94.3	68.4	83.52	83.91
60	59.0	85.0	84.0	89.3	71.8	82.32	82.51
66	47.0	84.0	80.0	82.1	73.0	78.64	78.63
72	39.0	80.0	73.4	72.4	73.2	72.48	72.32
78	32.0	73.0	66.3	63.0	68.3	65.76	65.49
84	28.0	64.0	58.7	53.7	64.0	58.56	58.21
90	24.0	54.0	52.0	46.2	59.7	52.10	51.70
96	22.0	44.0	45.6	39.3	55.0	45.92	45.50
102	21.0	36.0	40.0	33.9	50.4	40.56	40.15
108	20.0	30.0	35.6	29.9	46.3	36.20	35.82
114	19.0	25.0	33.0	26.9	42.7	32.62	32.26
120	19.0	22.0	28.9	24.3	39.2	29.49	29.17
126	18.0	19.0	26.6	22.8	36.5	27.22	26.93

Karahan, H. (2012). Predicting Muskingum flood routing parameters using spreadsheets. Computer Applications in Engineering Education, 20(2), 280-286.

Warning: It should be noted that Karahan seems to use an invalid value for $\Delta t = 6$. According to theory, the minimum value should be $2kx = 13$.

1.4.3 Verification of multiple river segments - IJssel

The only model known that incorporates multiple river segements is the model of Ciullo. In this paper the IJssel is modelled as multiple muskingum segments in sequence. His code is published on Github: https://github.com/quaquel/epa1361_open.

Ciullo, A., de Bruijn, K. M., Kwakkel, J. H., & Klijn, F. (2019). Accounting for the uncertain effects of hydraulic interactions in optimising embankments heights: Proof of principle for the IJssel River. Journal of Flood Risk Management, e12532. https://onlinelibrary.wiley.com/doi/pdf/10.1111/jfr3.12532

Parts of this code were extracted to understand the techniques used. Then my own code was run on the same data and yielded exactly the same results. These two notebooks are not commented but show the same results. The model is thus verified for multiple segments.

IJssel model in my code

IJssel model verification script

Muskingum routing in the IJssel

Script to verify code generating flows in the IJssel, or multiple river segments in sequence. Output of this script corresponds to *verfication model*.

[1]:
```
import sys
import os
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

[2]:
```
from context import fit_muskingum
from fit_muskingum import getParams
from fit_muskingum import calc_Out
from fit_muskingum import calc_C
import generate_network
```

[3]:
```
G, dike_list = generate_network.get_network()
```

[4]:
```
for x in dike_list:
    print(G.node[x])
```

```
{'id': 1, 'pnode': 'A.0', 'type': 'dike', 'C1': 0.7037044883065382, 'C2': 0.
↪07822399739748591, 'C3': 0.2180715142959759}
{'id': 2, 'pnode': 'A.1', 'type': 'dike', 'C1': 0.8976112018759458, 'C2': 0.
↪1099915720737832, 'C3': -0.007602773949729169}
{'id': 3, 'pnode': 'A.2', 'type': 'dike', 'C1': 0.8159335544521713, 'C2': 0.
↪1571571140947814, 'C3': 0.0269093314530472}
{'id': 4, 'pnode': 'A.3', 'type': 'dike', 'C1': 0.6240289660529375, 'C2': 0.
↪6138997464211343, 'C3': -0.2379287124740719}
{'id': 5, 'pnode': 'A.4', 'type': 'dike', 'C1': 0.6240289660529375, 'C2': 0.
↪6138997464211343, 'C3': -0.2379287124740719}
```

[5]:
```
G.node['A.0']['Qout'] = 2000 * G.node['A.0']['Qevents_shape'].loc[0]
```

[6]:
```
Qin = 2000 * G.node['A.0']['Qevents_shape'].loc[0]
```

[7]:
```
df = pd.DataFrame({'Qin':Qin})
```

[8]:
```
x = -1.055501123
k = 0.378929169
dt = 1

C1 = calc_C(k,x,dt)
QA1 = calc_Out(Qin,C1)
#df['A.1test'] = QA1
```

[9]:
```
params = pd.read_excel('../../data/params.xlsx',index_col=0)
params
```

[9]:
```
            K         X        C1        C2        C3
A.0  0.378929 -1.055501  0.703704  0.078224  0.218072
A.1  0.101616 -3.846220  0.897611  0.109992 -0.007603
A.2  0.189157 -1.789507  0.815934  0.157157  0.026909
A.3  0.303710 -0.013471  0.624029  0.613900 -0.237929
A.5  0.779782  0.074716  0.361629  0.457023  0.181347
A.4  0.303710 -0.013471  0.624029  0.613900 -0.237929
```

[10]:
```
params.loc['A.0']['K']
```

[10]:
```
0.3789291694964745
```

[11]:
```
Qin = 2000 * G.node['A.0']['Qevents_shape'].loc[0]

nodes = ['A.0','A.1','A.2','A.3','A.4']
nodes_title = ['A.1','A.2','A.3','A.4','A.5']
i=0
for node in nodes:
    k = params.loc[node]['K']
    x = params.loc[node]['X']
    dt = 1
    C = calc_C(k,x,dt)
    Qin = calc_Out(Qin,C)
    df[nodes_title[i]] = Qin
    i = i+1
```

[12]:
```
df
```

[12]:
```
            Qin          A.1          A.2          A.3          A.4  \
0    449.789315   449.789315   449.789315   449.789315   449.789315
1    428.608569   434.884329   436.410432   438.873036   442.977241
2    448.566119   444.021351   443.074220   441.913907   439.794119
3    460.705731   456.117734   454.886401   452.680952   449.137213
4    438.468008   444.056434   445.300738   447.005789   449.982643
5    407.024353   417.559643   420.263158   424.917618   432.513852
6    382.125718   391.800518   394.417410   399.299993   407.124116
7    374.944239   379.181877   380.453989   383.155573   387.363824
8    448.863039   427.885339   422.888978   415.150818   402.120269
9    650.612166   586.260166   570.082344   542.780756   497.895938
10   967.408445   859.509791   831.655087   782.773655   703.222663
11  1247.376736  1140.893765  1112.294971  1059.323219   974.276058
12  1562.398600  1445.838133  1414.832676  1357.720201  1265.766741
13  1841.356795  1733.284216  1704.088685  1649.309501  1561.558739
14  1972.873757  1910.338320  1892.431930  1856.290187  1799.349871
15  2000.000000  1978.325419  1971.500439  1955.974030  1932.043529
16  1952.257016  1961.676439  1963.432997  1964.500137  1966.988321
17  1836.171355  1872.621123  1881.726035  1896.794261  1921.657699
18  1635.558763  1702.948030  1720.251431  1750.378964  1799.511149
19  1424.241434  1501.549490  1522.038890  1559.333880  1619.471340
20  1196.013888  1280.495371  1302.973060  1344.299312  1410.837652
21   975.827326  1059.490621  1081.948139  1123.743472  1190.834698
22   832.736179   893.378025   910.215355   942.950282   994.960356
23   704.739634   755.888695   769.838052   796.557678   839.222367
24   597.517524   640.441107   652.155593   674.535992   710.261457
25   530.017304   559.377727   567.588646   583.756825   609.387049
26   463.353857   489.508609   496.599998   510.101701   531.695728
27   439.228066   452.080036   455.858388   463.720873   476.020882
```

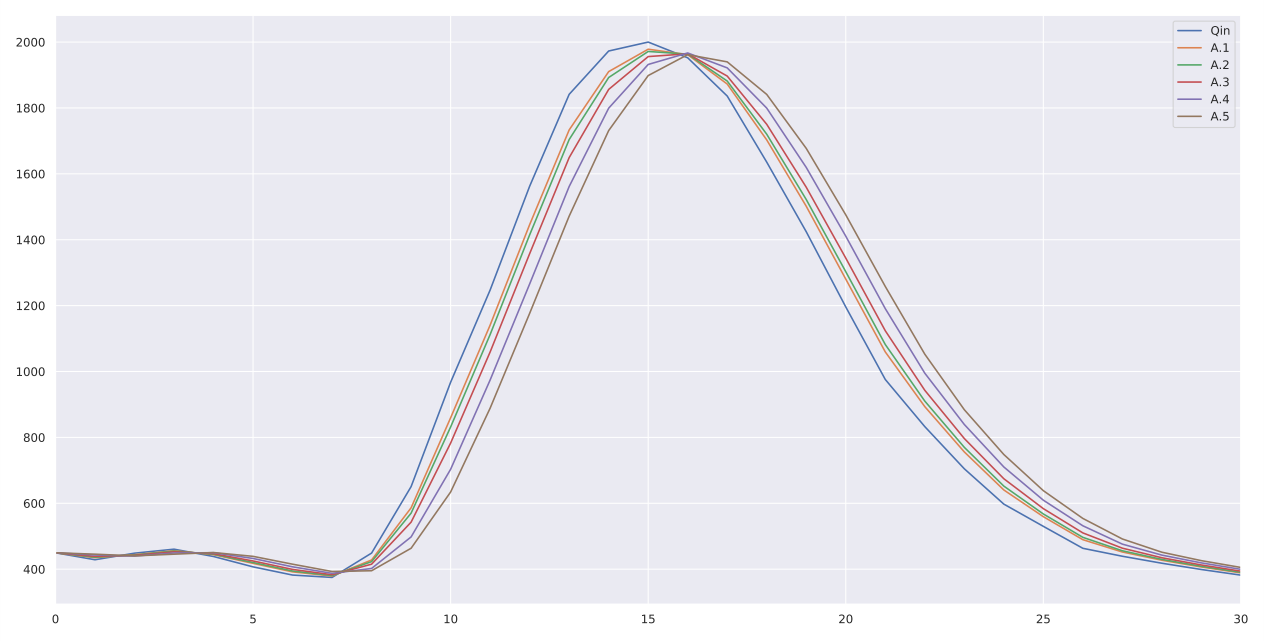
(continues on next page)

(continued from previous page)

```
28   418.041637   427.121729   429.648454   434.684398   442.674746
29   398.997576   406.620356   408.700256   412.691631   419.059141
30   381.892728   388.623129   390.450030   393.916690   399.460510

            A.5
0    449.789315
1    445.538383
2    440.381511
3    445.484723
4    450.533818
5    438.950471
6    415.138465
7    392.886278
8    395.258323
9    463.519715
10   634.204853
11   888.789152
12  1176.514477
13  1471.585192
14  1731.354583
15  1898.332588
16  1961.870892
17  1939.918283
18  1841.089996
19  1677.268291
20  1475.526321
21  1258.158144
22  1052.585254
23   884.064722
24   748.077740
25   638.315325
26   554.022547
27   491.640820
28   451.495496
29   425.839215
30   405.215853
```

[13]:
```
df.plot(figsize=(20,10))
```

[13]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f095ecbc9e8>
```



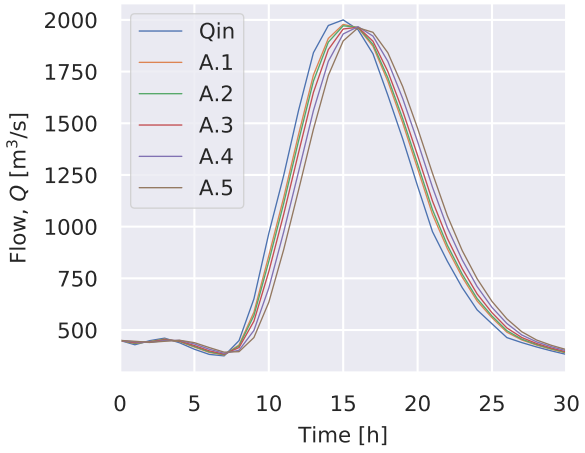
[14]:
```
#%qtconsole
```

Figure 5.2 for thesis

[15]:
```
import seaborn as sns
sns.set_context("paper", rc={"font.size":8.0,
                             'lines.linewidth':0.5,
                             'patch.linewidth':0.5,
                             "axes.titlesize":8,
                             "axes.labelsize":8,
                             'xtick.labelsize':8,
                             'ytick.labelsize':8,
                             'legend.fontsize':8 ,
                             'pgf.rcfonts' : False})
```

[16]:
```
fig = plt.figure(figsize=(3,2.5),dpi=150)
ax = fig.add_subplot(111)

#fig.set_size_inches(6,3)
fig.patch.set_alpha(0)
fig.set_dpi(150)

#plt.plot(t,I,linewidth = 1 , label = 'inflow')
df.plot(ax=ax, linewidth = 0.5)

plt.ylabel('Flow, $Q$ [m$^3$/s]')
plt.xlabel('Time [h]')
plt.legend()
#plt.tight_layout()
# save to file
#plt.savefig('../../../thesis/report/figs/ijssel.pdf', bbox_inches = 'tight')
#plt.savefig('../../../thesis/report/figs/ijssel.pgf', bbox_inches = 'tight')
```

[16]:
```
<matplotlib.legend.Legend at 0x7f0951050e10>
```



Muskingum routing in the IJssel - verification script

Code adjusted from https://github.com/quaquel/epa1361_open. The output of this model is equal to my own model, as can be seen in *this script*.

Source: https://onlinelibrary.wiley.com/doi/abs/10.1111/jfr3.12532

Ciullo, A., de Bruijn, K. M., Kwakkel, J. H., & Klijn, F. (2019). Accounting for the uncertain effects of hydraulic interactions in optimising embankments heights: Proof of principle for the IJssel River. Journal of Flood Risk Management, e12532.

[1]:
```
import sys
import os
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

[2]:
```
import generate_network
from functions_ijssel_muskingum import Muskingum
```

[3]:
```
G, dike_list = generate_network.get_network()
```

[4]:
```
class DikeNetwork(object):
    def __init__(self):
        # planning steps
        self.num_events = 30

        # load network
        G, dike_list = generate_network.get_network()

        self.Qpeaks = 2000 #np.random.uniform(1000,16000,100)

        self.G = G
        self.dikelist = dike_list

    def printG(self):
```

(continues on next page)

(continued from previous page)

```
        print(G.nodes.data())

    def getG(self):
        return G


    def init_node(self,value, time):
        init = np.repeat(value, len(time)).tolist()
        return init

    def _initialize_hydroloads(self, node, time, Q_0):
        #node['cumVol'], node['wl'], node['Qpol'], node['hbas'] = (
        #    self.init_node(0, time) for _ in range(4))
        node['Qin'], node['Qout'] = (self.init_node(Q_0, time) for _ in range(2))
        #node['status'] = self.init_node(False, time)
        #node['tbreach'] = np.nan
        return node

    def calc_wave(self,timestep=1):
        startnode = G.node['A.0']
        waveshape_id = 0
        Qpeak = self.Qpeaks#[0]
        dikelist = self.dikelist
        time = np.arange(0, startnode['Qevents_shape'].loc[waveshape_id].shape[0],
                         timestep)
        startnode['Qout'] = Qpeak * startnode['Qevents_shape'].loc[waveshape_id]

        # Initialize hydrological event:
        for key in dikelist:
            node = G.node[key]
            #Q_0 = int(G.node['A.0']['Qout'][0])
            Q_0 = G.node['A.0']['Qout'][0]
            self._initialize_hydroloads(node, time, Q_0)

        # Run the simulation:
        # Run over the discharge wave:
        for t in range(1, len(time)):
            # Run over each node of the branch:
            for n in range(0, len(dikelist)):
                # Select current node:
                node = G.node[dikelist[n]]
                if node['type'] == 'dike':
                    # Muskingum parameters:
                    C1 = node['C1']
                    C2 = node['C2']
                    C3 = node['C3']

                    prev_node = G.node[node['pnode']]
                    # Evaluate Q coming in a given node at time t:
                    node['Qin'][t] = Muskingum(C1, C2, C3,
                                               prev_node['Qout'][t],
                                               prev_node['Qout'][t - 1],
                                               node['Qin'][t - 1])

                    node['Qout'][t] = node['Qin'][t]

    def __call__(self, timestep=1, **kwargs):
```

(continues on next page)

(continued from previous page)

```
        G = self.G
        Qpeaks = self.Qpeaks
        dikelist = self.dikelist
```

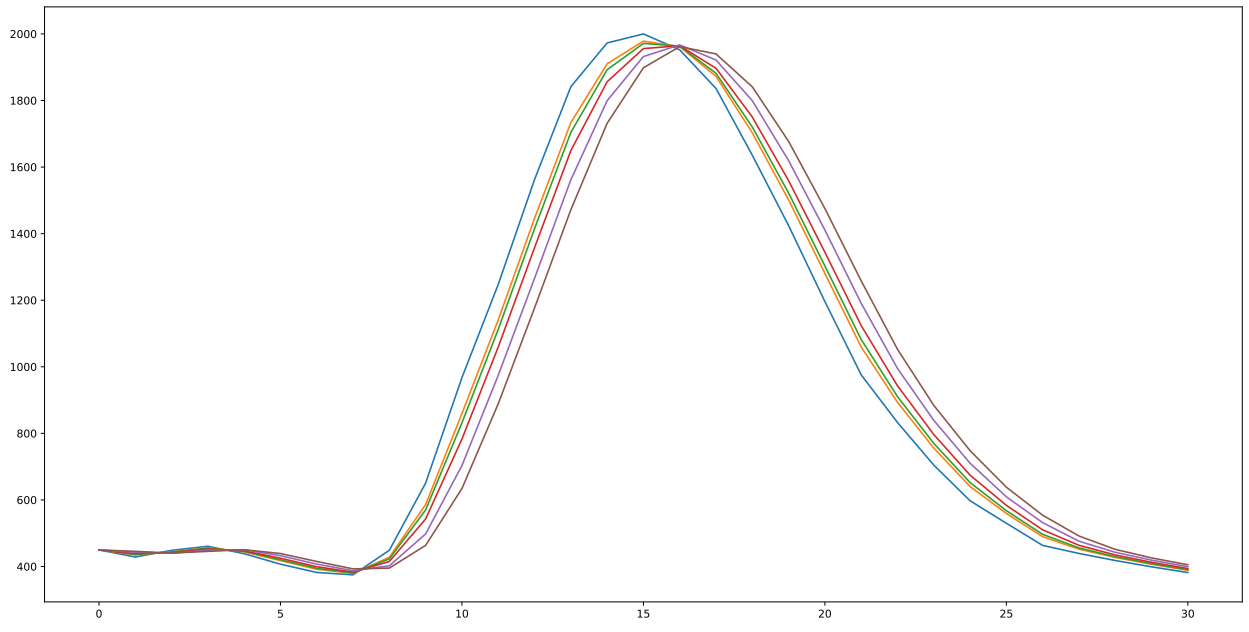
[5]:
```
dikeNetwork = DikeNetwork()
```

[6]:
```
dikeNetwork.calc_wave()
```

[7]:
```
G = dikeNetwork.getG()
```

[8]:
```
#G.nodes['A.1']
```

[9]:
```
plt.figure(figsize=(20,10))
plt.plot(G.node['A.0']['Qout'])
df = pd.DataFrame({'Qin':G.node['A.0']['Qout']})

dikelist = dikeNetwork.dikelist
for n in range(0, len(dikelist)):
    node = G.node[dikelist[n]]
    plt.plot(node['Qin'])
    df[dikelist[n]] = node['Qin']
```



[10]:
```
df
```

[10]:

	Qin	A.1	A.2	A.3	A.4	\
0	449.789315	449.789315	449.789315	449.789315	449.789315	
1	428.608569	434.884329	436.410432	438.873036	442.977241	
2	448.566119	444.021351	443.074220	441.913907	439.794119	
3	460.705731	456.117734	454.886401	452.680952	449.137213	
4	438.468008	444.056434	445.300738	447.005789	449.982643	
5	407.024353	417.559643	420.263158	424.917618	432.513852	
6	382.125718	391.800518	394.417410	399.299993	407.124116	
7	374.944239	379.181877	380.453989	383.155573	387.363824	

(continues on next page)

(continued from previous page)

8	448.863039	427.885339	422.888978	415.150818	402.120269
9	650.612166	586.260166	570.082344	542.780756	497.895938
10	967.408445	859.509791	831.655087	782.773655	703.222663
11	1247.376736	1140.893765	1112.294971	1059.323219	974.276058
12	1562.398600	1445.838133	1414.832676	1357.720201	1265.766741
13	1841.356795	1733.284216	1704.088685	1649.309501	1561.558739
14	1972.873757	1910.338320	1892.431930	1856.290187	1799.349871
15	2000.000000	1978.325419	1971.500439	1955.974030	1932.043529
16	1952.257016	1961.676439	1963.432997	1964.500137	1966.988321
17	1836.171355	1872.621123	1881.726035	1896.794261	1921.657699
18	1635.558763	1702.948030	1720.251431	1750.378964	1799.511149
19	1424.241434	1501.549490	1522.038890	1559.333880	1619.471340
20	1196.013888	1280.495371	1302.973060	1344.299312	1410.837652
21	975.827326	1059.490621	1081.948139	1123.743472	1190.834698
22	832.736179	893.378025	910.215355	942.950282	994.960356
23	704.739634	755.888695	769.838052	796.557678	839.222367
24	597.517524	640.441107	652.155593	674.535992	710.261457
25	530.017304	559.377727	567.588646	583.756825	609.387049
26	463.353857	489.508609	496.599998	510.101701	531.695728
27	439.228066	452.080036	455.858388	463.720873	476.020882
28	418.041637	427.121729	429.648454	434.684398	442.674746
29	398.997576	406.620356	408.700256	412.691631	419.059141
30	381.892728	388.623129	390.450030	393.916690	399.460510

	A.5
0	449.789315
1	445.538383
2	440.381511
3	445.484723
4	450.533818
5	438.950471
6	415.138465
7	392.886278
8	395.258323
9	463.519715
10	634.204853
11	888.789152
12	1176.514477
13	1471.585192
14	1731.354583
15	1898.332588
16	1961.870892
17	1939.918283
18	1841.089996
19	1677.268291
20	1475.526321
21	1258.158144
22	1052.585254
23	884.064722
24	748.077740
25	638.315325
26	554.022547
27	491.640820
28	451.495496
29	425.839215
30	405.215853

1.5 Network 1: confluences only

In the following sections two examples of networks are given. These exmples show how the developed class can be used to model small river networks. This simple network only contains confluences and no bifurcations.

[1]:
```
import pandas as pd
import networkx as nx
import matplotlib.pyplot as plt
import numpy as np
```

[2]:
```
from context import RiverNetwork
from RiverNetwork import RiverNetwork
```

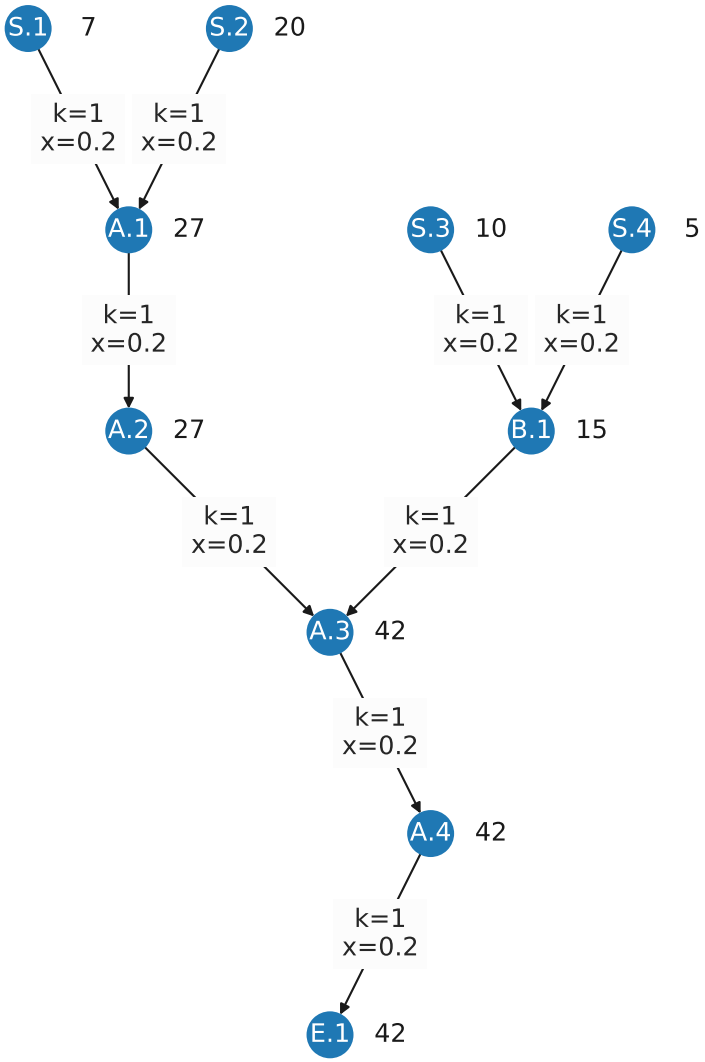
1.5.1 Loading network structure

An extra file containing wave shapes is loaded as well. This file makes it possible to select arbitrary wave shapes as input flows.

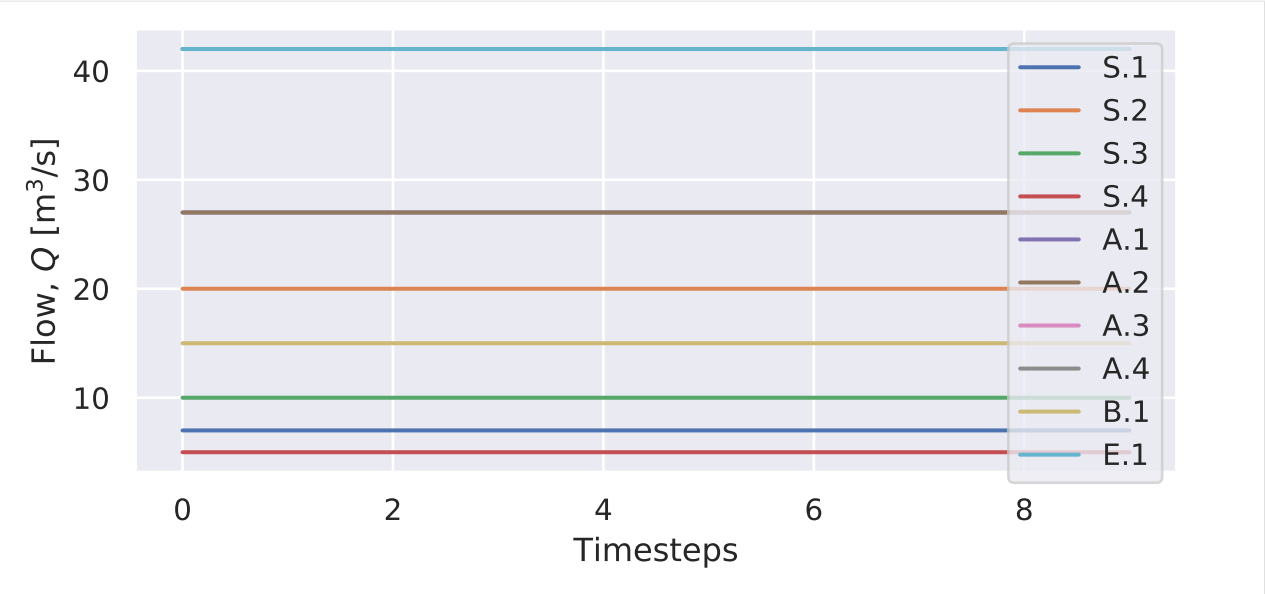
[3]:
```
structure1 = RiverNetwork('../data/network-structure-1.xlsx',wave_shapes_location='../
↪data/wave_shapes.xls')
```

[4]:
```
structure1.draw()
```

```
/home/docs/checkouts/readthedocs.org/user_builds/rna/envs/latest/lib/python3.7/
↪site-packages/networkx/drawing/nx_pylab.py:579: MatplotlibDeprecationWarning:
The iterable function was deprecated in Matplotlib 3.1 and will be removed in 3.3.␣
↪Use np.iterable instead.
  if not cb.iterable(width):
/home/docs/checkouts/readthedocs.org/user_builds/rna/envs/latest/lib/python3.7/
↪site-packages/networkx/drawing/nx_pylab.py:676: MatplotlibDeprecationWarning:
The iterable function was deprecated in Matplotlib 3.1 and will be removed in 3.3.␣
↪Use np.iterable instead.
  if cb.iterable(node_size):  # many node sizes
```



[5]:
```
structure1.draw_base_loads(figsize=(7,3))
```



1.5.2 Determining calculation order

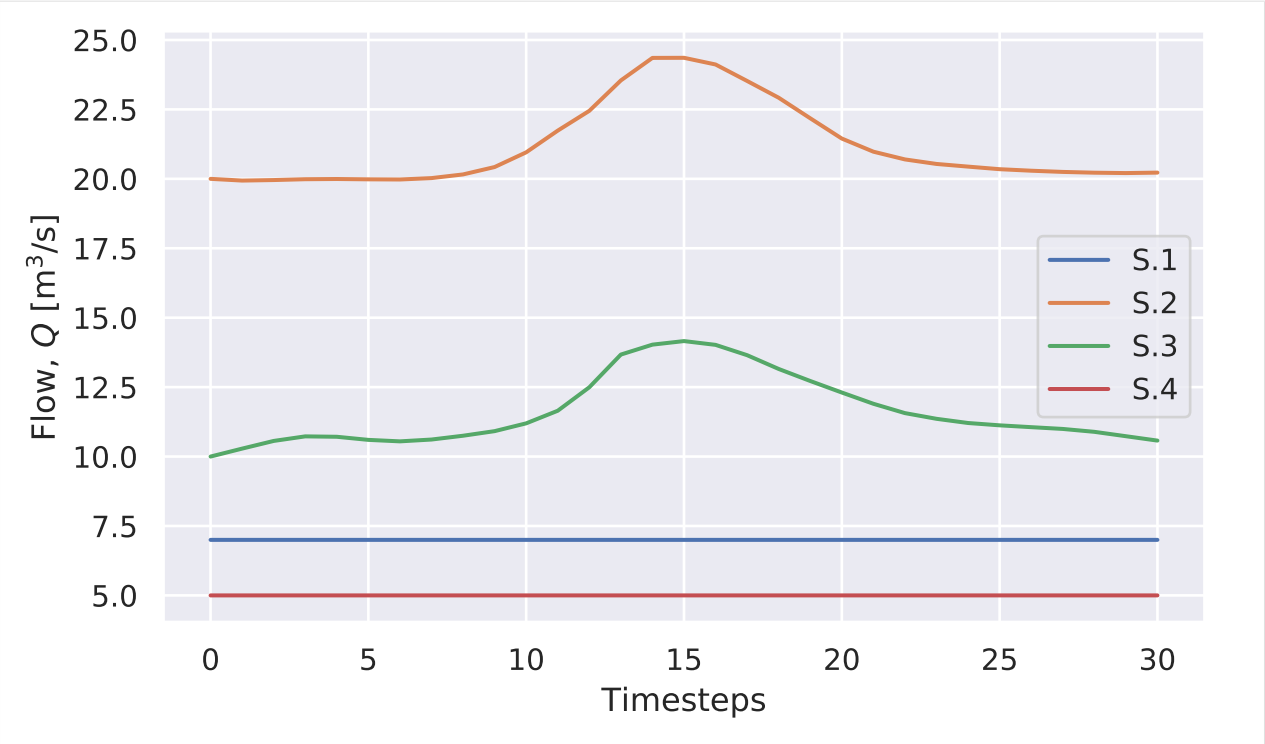
In order to determine the calculation order the following following procedure is used: A Depth First Search (DFS) algorithm is used starting at the sink E.1. The output of this algorithm is a list and tells us all steps from the sink all the way to the sources. The source furthest away from the sink is last in the list (BFS opposed to DFS). By reversing this list a new list with a safe calculation order is created. This list guarantees that all edges are traversed and calculated before they are used.

1.5.3 Experiment 1

With this baseload and network structure it is possible to perform experiments by setting different inflows on top of the base flows. There are two constant flows selected: these are based on the base load. And two waves are selected from the wave shape file.

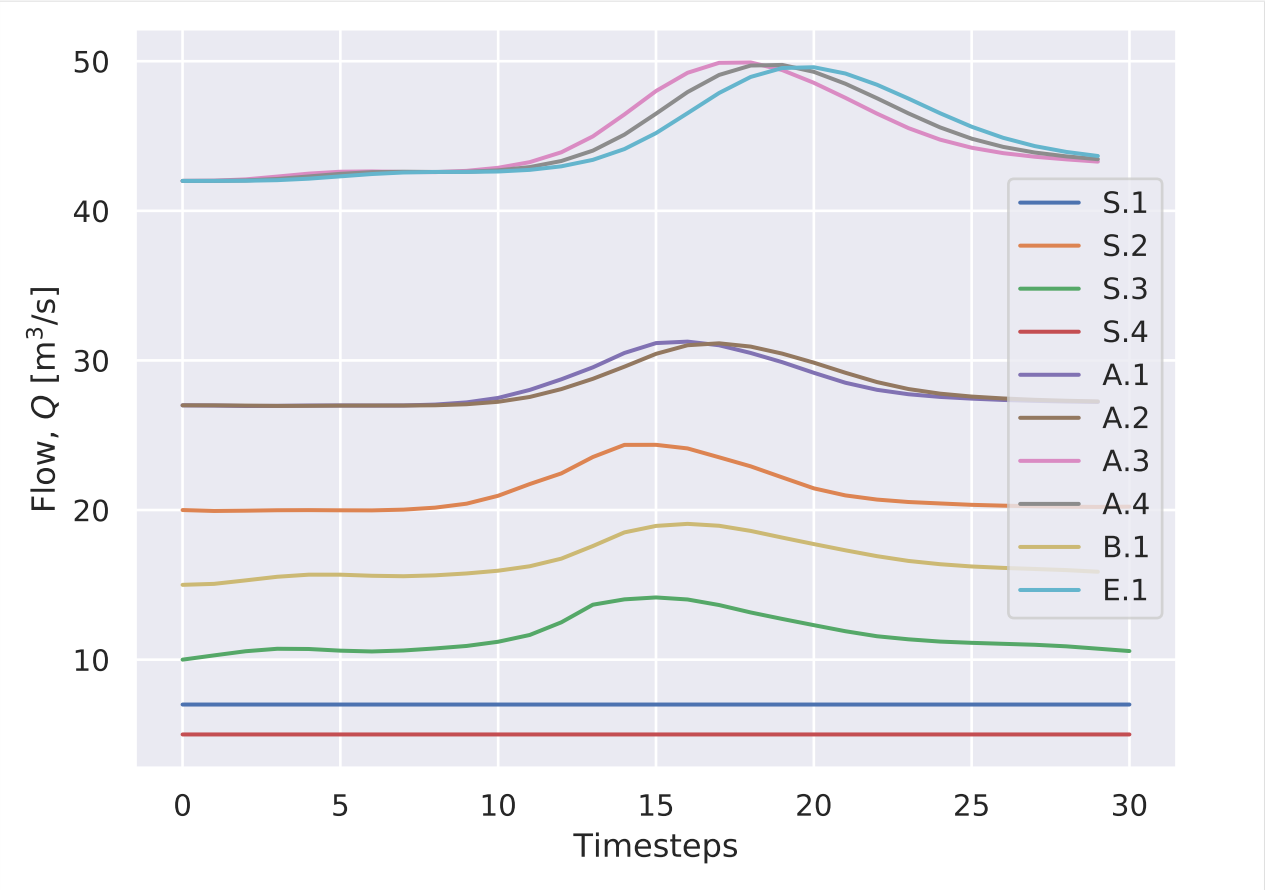
[6]:
```
structure1.set_constant_flow('S.1',31)
structure1.set_wave('S.2',shape_number=5,strength=5)
structure1.set_wave('S.3',shape_number=90,strength=5)
structure1.set_constant_flow('S.4',31)
structure1.draw_Qin(only_sources=True,figsize=(7,4))
```

[6]:
```
(<Figure size 672x384 with 1 Axes>,
 <matplotlib.axes._subplots.AxesSubplot at 0x7fae67dc23c8>)
```



[7]:
```
structure1.calc_flow_propagation(30)
structure1.draw_Qin(figsize=(7,5))
```

[7]:
```
(<Figure size 672x480 with 1 Axes>,
 <matplotlib.axes._subplots.AxesSubplot at 0x7fae67d04358>)
```



Warning: Coloring should change to improve readability. E.g. cluster nodes and give similar colors.

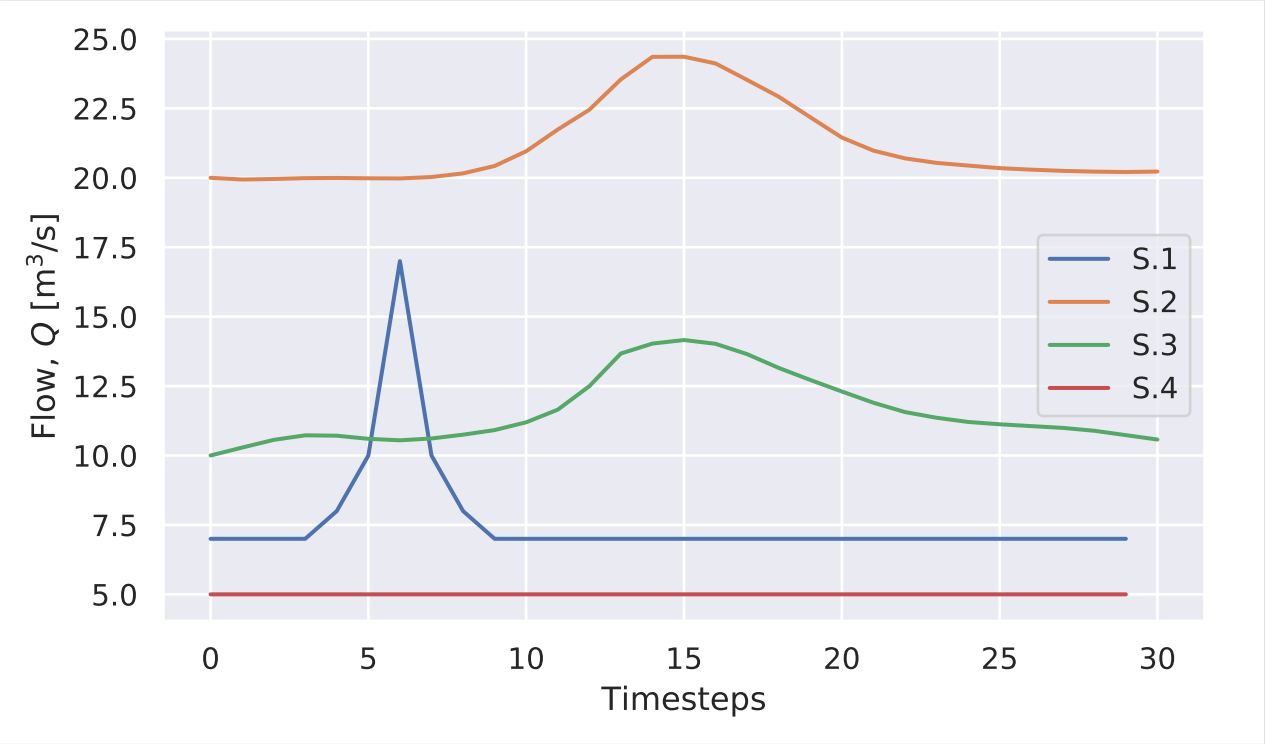
Here we see that: S.1 (blue) + S.2 (orange) gives A.1 (purple). A.2 (brown) is shifted relative to A.1. S.3 (green) + S.4 (red) gives B.1 (yellow). A.2 and B.1 give A.3 (pink). And clearly A.3, A.4 (grey) and E.1 (light blue) are just simple muskingum transformations.

1.5.4 Experiment 2

Now it is possible to add any other extra inflow. In the following experiment a peak flow is added to S.1.

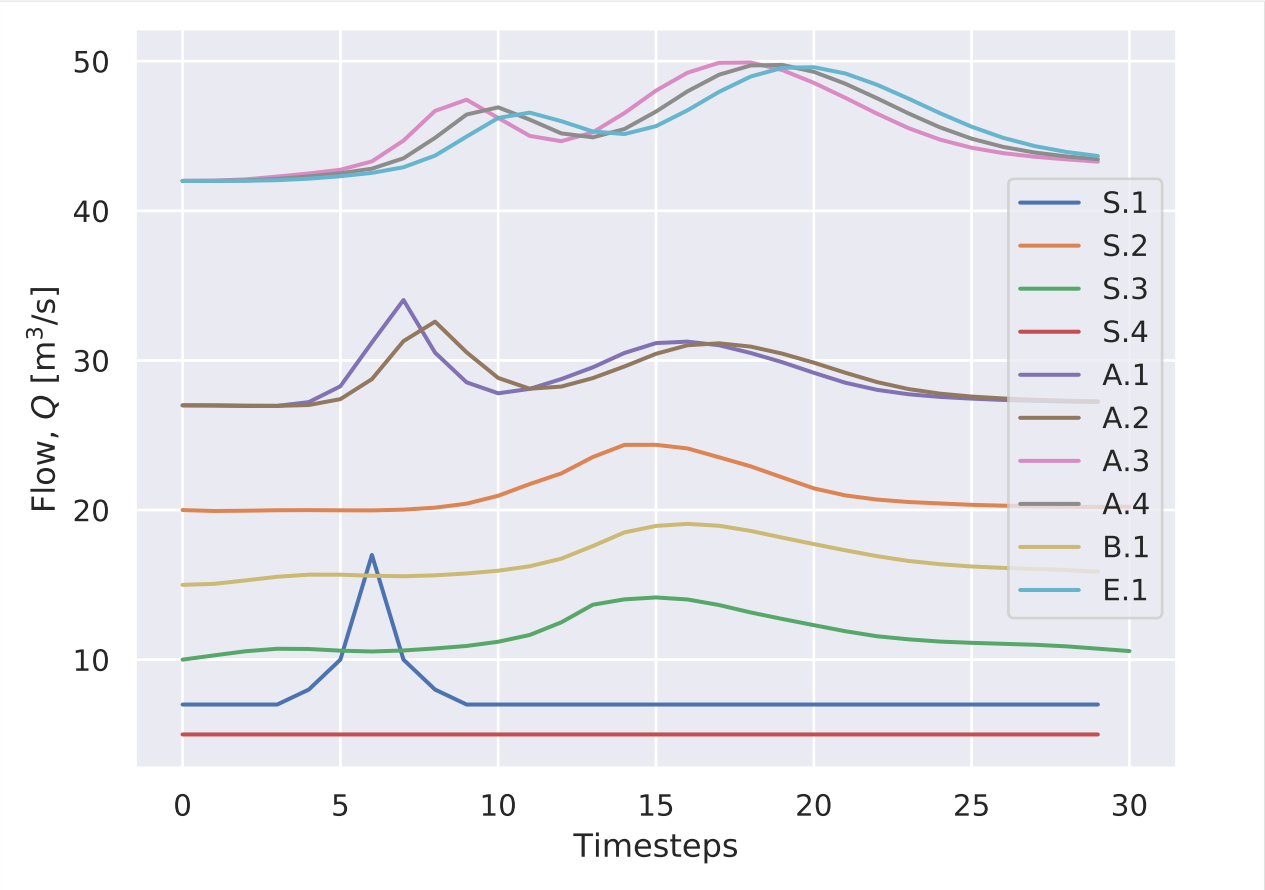
[8]:
```
shape = np.zeros(30)
shape[4] = 1
shape[5] = 3
shape[6] = 10
shape[7] = 3
shape[8] = 1
structure1.set_shape('S.1',30,shape)
structure1.set_wave('S.2',shape_number=5,strength=5)
structure1.set_wave('S.3',shape_number=90,strength=5)
structure1.set_constant_flow('S.4',30)
structure1.draw_Qin(only_sources=True,figsize=(7,4))
```

[8]:
```
(<Figure size 672x384 with 1 Axes>,
 <matplotlib.axes._subplots.AxesSubplot at 0x7fae68089eb8>)
```



[9]:
```
structure1.calc_flow_propagation(30)
structure1.draw_Qin(figsize=(7,5))
```

[9]:
```
(<Figure size 672x480 with 1 Axes>,
 <matplotlib.axes._subplots.AxesSubplot at 0x7fae6a949eb8>)
```



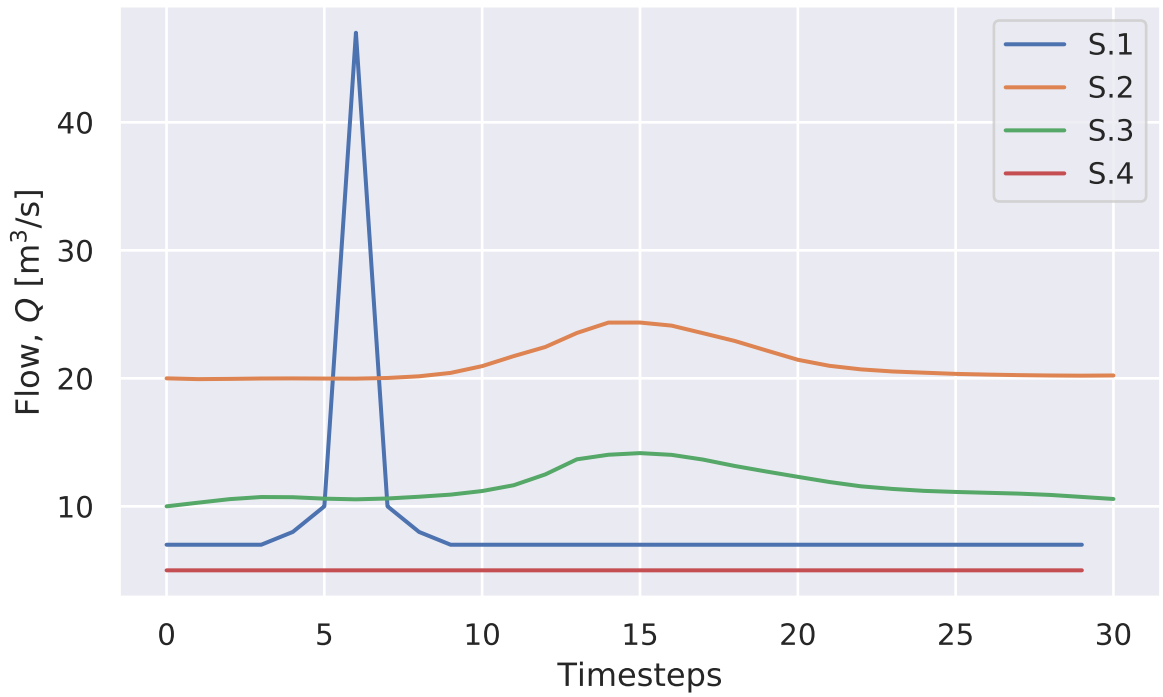
Because of the peak shape the propagation through the network can clearly be seen.

1.5.5 Experiment 3

This effect can even be made larger:

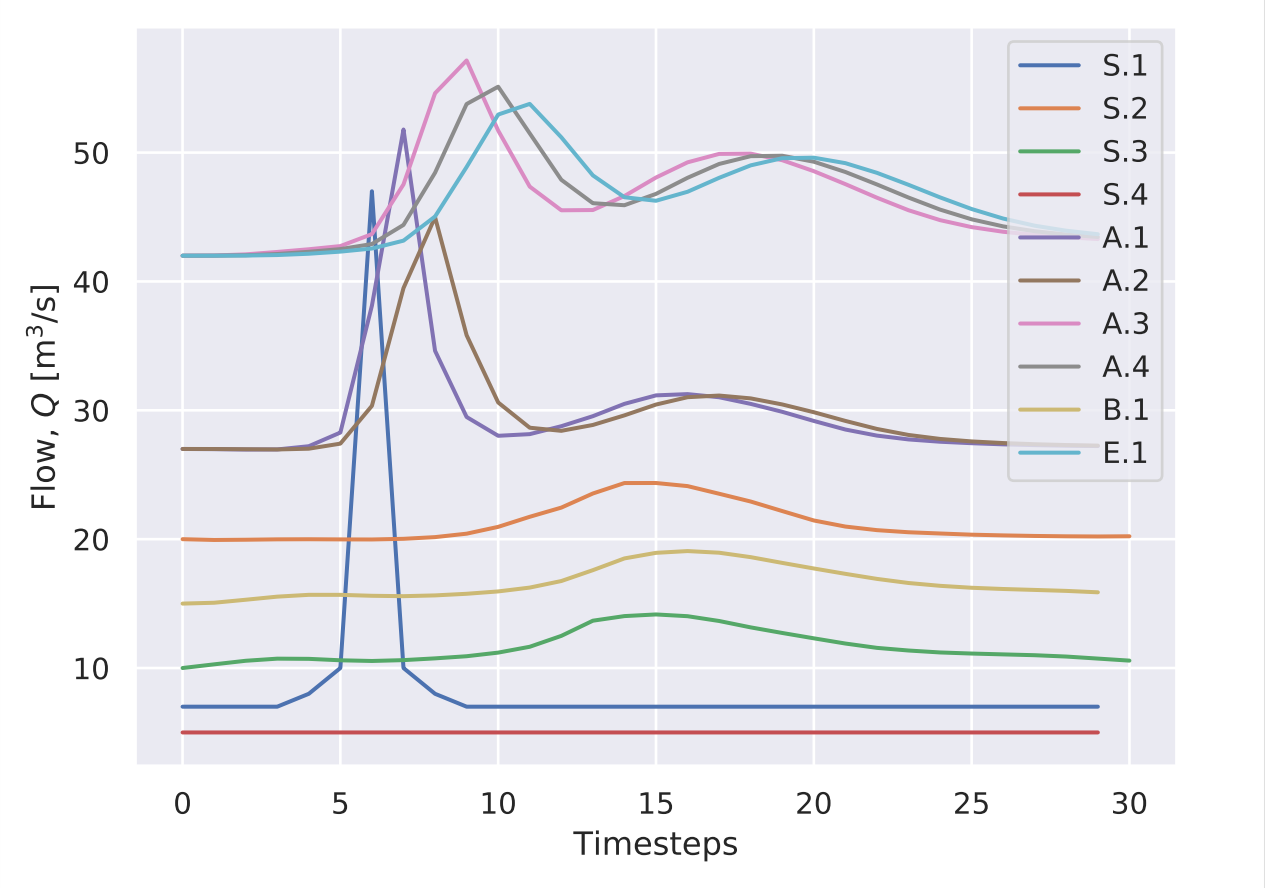
[10]:
```
shape = np.zeros(30)
shape[4] = 1
shape[5] = 3
shape[6] = 40
shape[7] = 3
shape[8] = 1
structure1.set_shape('S.1',30,shape)
structure1.draw_Qin(only_sources=True,figsize=(7,4))
```

[10]:
```
(<Figure size 672x384 with 1 Axes>,
 <matplotlib.axes._subplots.AxesSubplot at 0x7fae687026a0>)
```



[11]:
```
structure1.calc_flow_propagation(30)
structure1.draw_Qin(figsize=(7,5))
```

[11]:
```
(<Figure size 672x480 with 1 Axes>,
 <matplotlib.axes._subplots.AxesSubplot at 0x7fae6865b240>)
```



1.6 Network 2: adding a bifurcation

This simple network contains confluences and a single bifurcations.

[1]:
```
import pandas as pd
import networkx as nx
import matplotlib.pyplot as plt
import numpy as np
```

[2]:
```
from context import RiverNetwork
from RiverNetwork import RiverNetwork
```

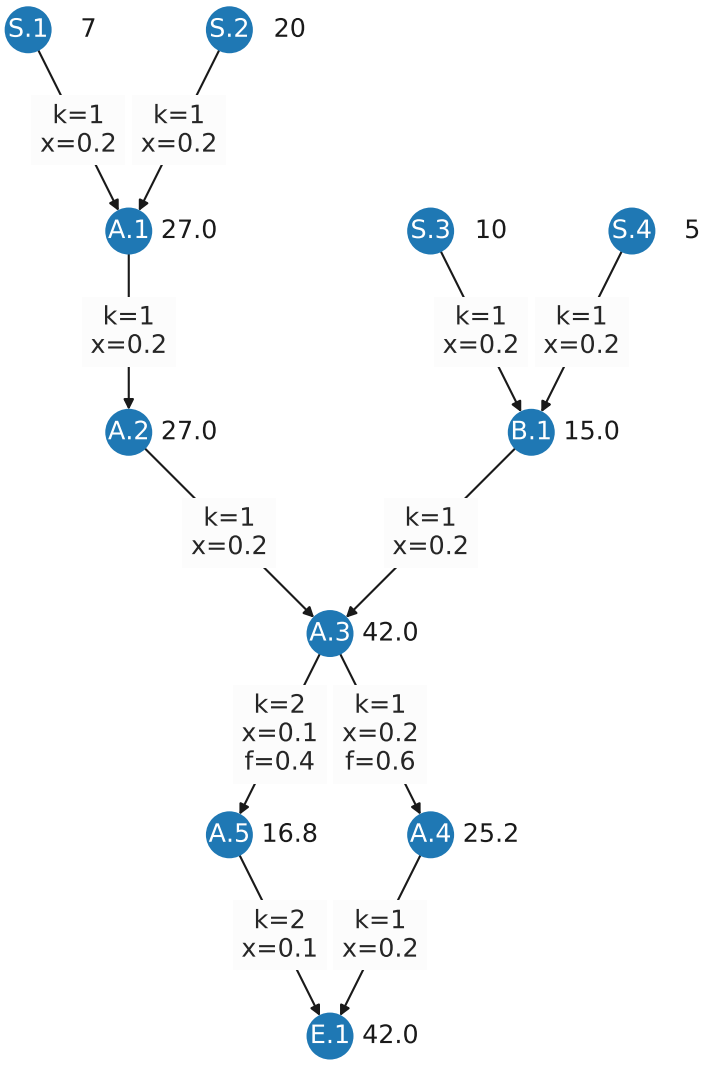
1.6.1 Loading network structure

An extra file containing wave shapes is loaded as well. This file makes it possible to select arbitrary wave shapes as input flows.

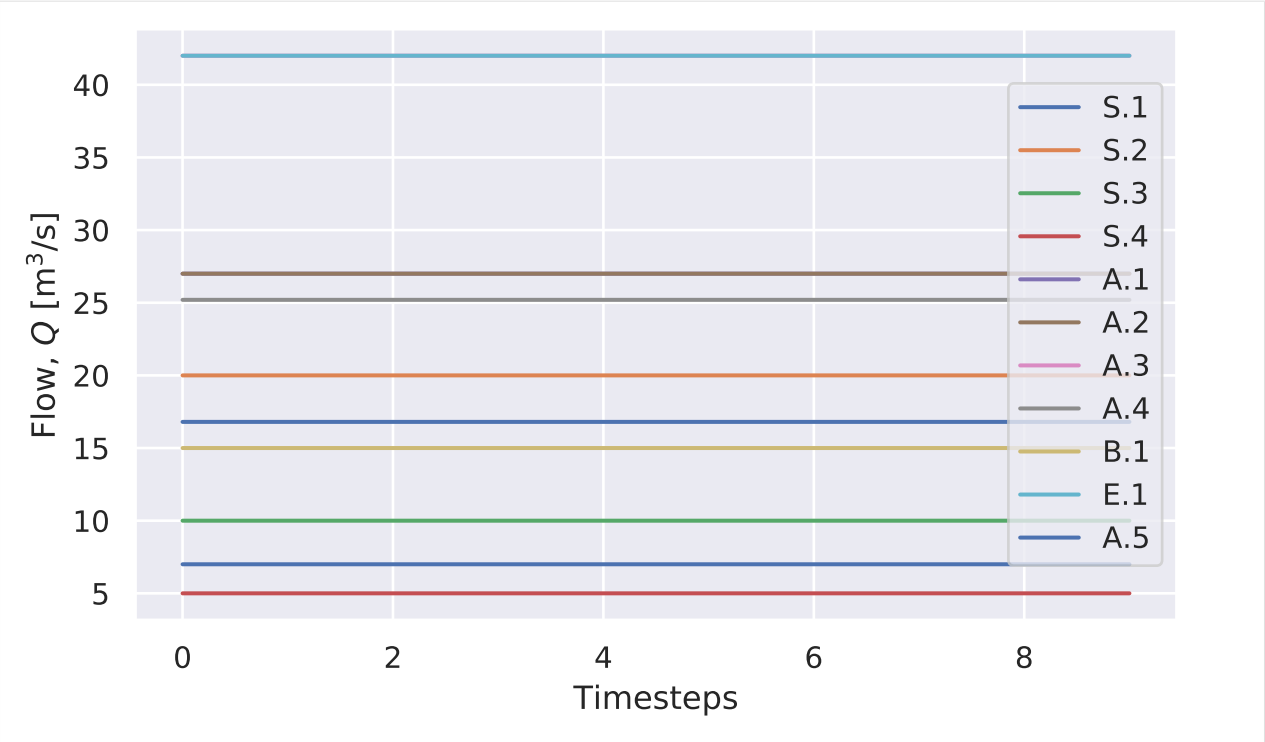
[3]:
```
structure1 = RiverNetwork('../data/network-structure-2.xlsx',wave_shapes_location='../
↪data/wave_shapes.xls')
```

[4]:
```
structure1.draw()
```

```
/home/docs/checkouts/readthedocs.org/user_builds/rna/envs/latest/lib/python3.7/
↪site-packages/networkx/drawing/nx_pylab.py:579: MatplotlibDeprecationWarning:
The iterable function was deprecated in Matplotlib 3.1 and will be removed in 3.3.␣
↪Use np.iterable instead.
  if not cb.iterable(width):
/home/docs/checkouts/readthedocs.org/user_builds/rna/envs/latest/lib/python3.7/
↪site-packages/networkx/drawing/nx_pylab.py:676: MatplotlibDeprecationWarning:
The iterable function was deprecated in Matplotlib 3.1 and will be removed in 3.3.␣
↪Use np.iterable instead.
  if cb.iterable(node_size):  # many node sizes
```

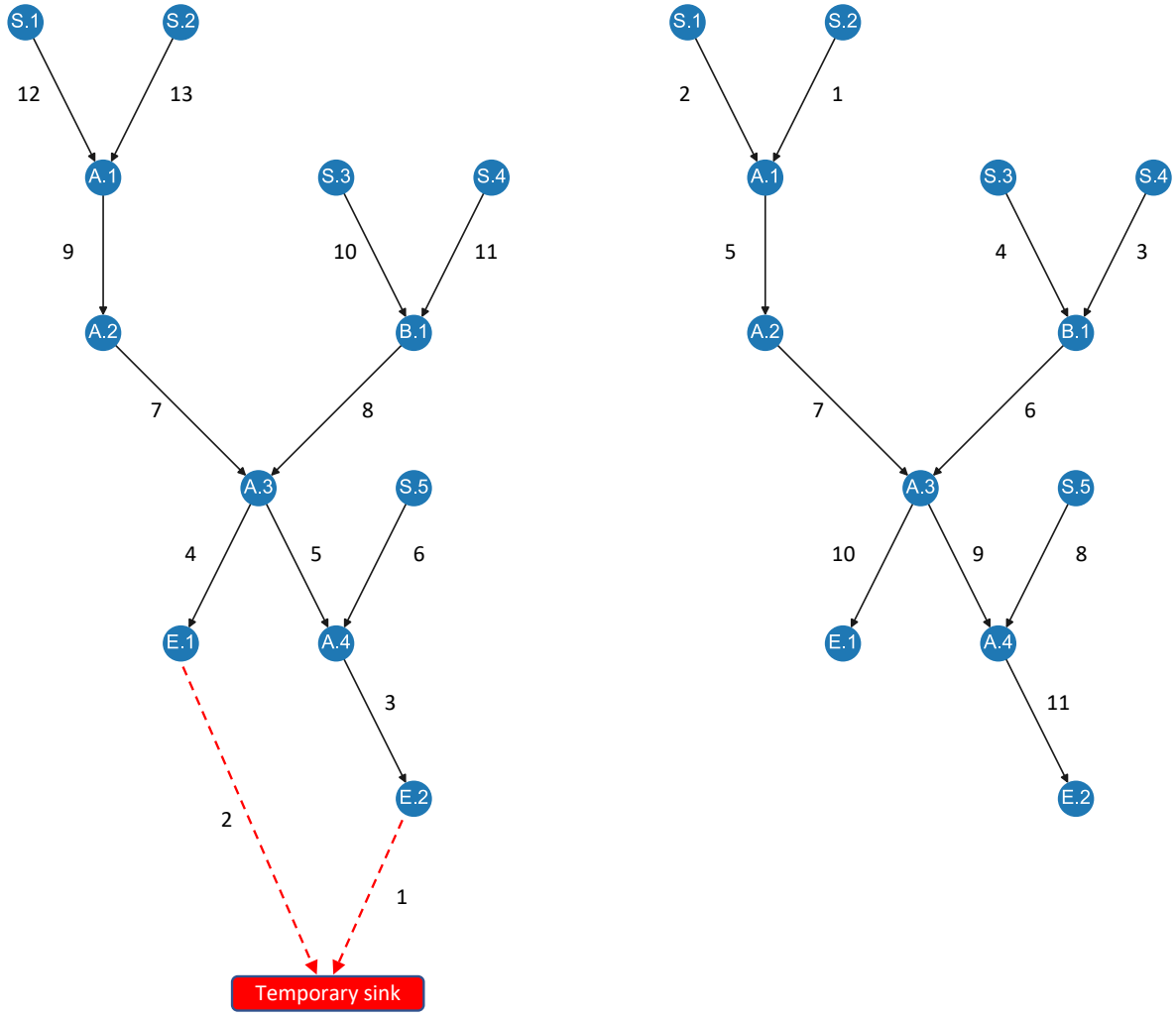


[5]:
```
structure1.draw_base_loads(figsize=(7,4))
```



1.6.2 Determining calculation order

The procedure for determining calculation order with multiple sinks is slight different. In that case a virtual sink is added to the system and all other sinks are connected to this node. Then a the same procedure is repeated: reversed BFS starting at this virtual sink. The resulting list is reversed and the result is a calculation order that guarantees that always all upstream flows are already calculated.

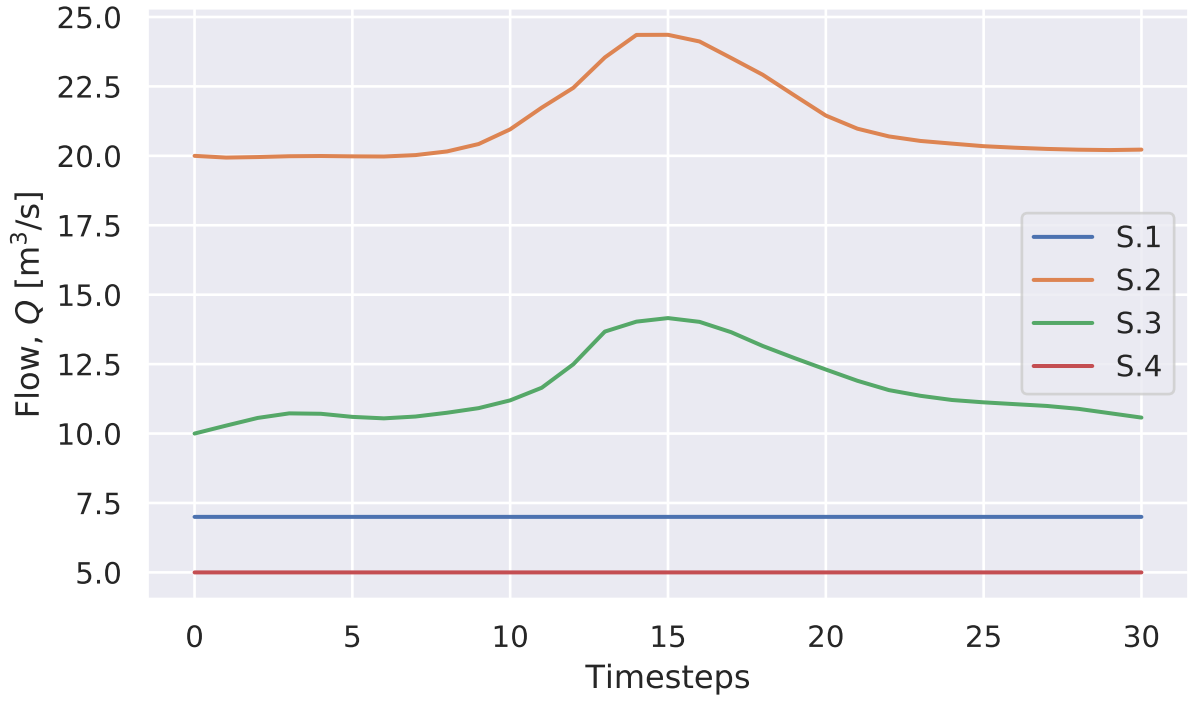


Example of the calculation order determining. In the left graph a temporary node and edges are added in red. From this node the reversed breadth-first search algorithm finds edges in a certain order. The order is indicated with numbers next to the edges. In the right graph the order of these edges is reversed and the temporary edges and node are removed. The order now corresponds to the calculation order that guarantees that all upstream parts are calculated before traversing downstream. This can be seen best at node A.3: all upstream edges are traversed before going downstream.

1.6.3 Experiment 1

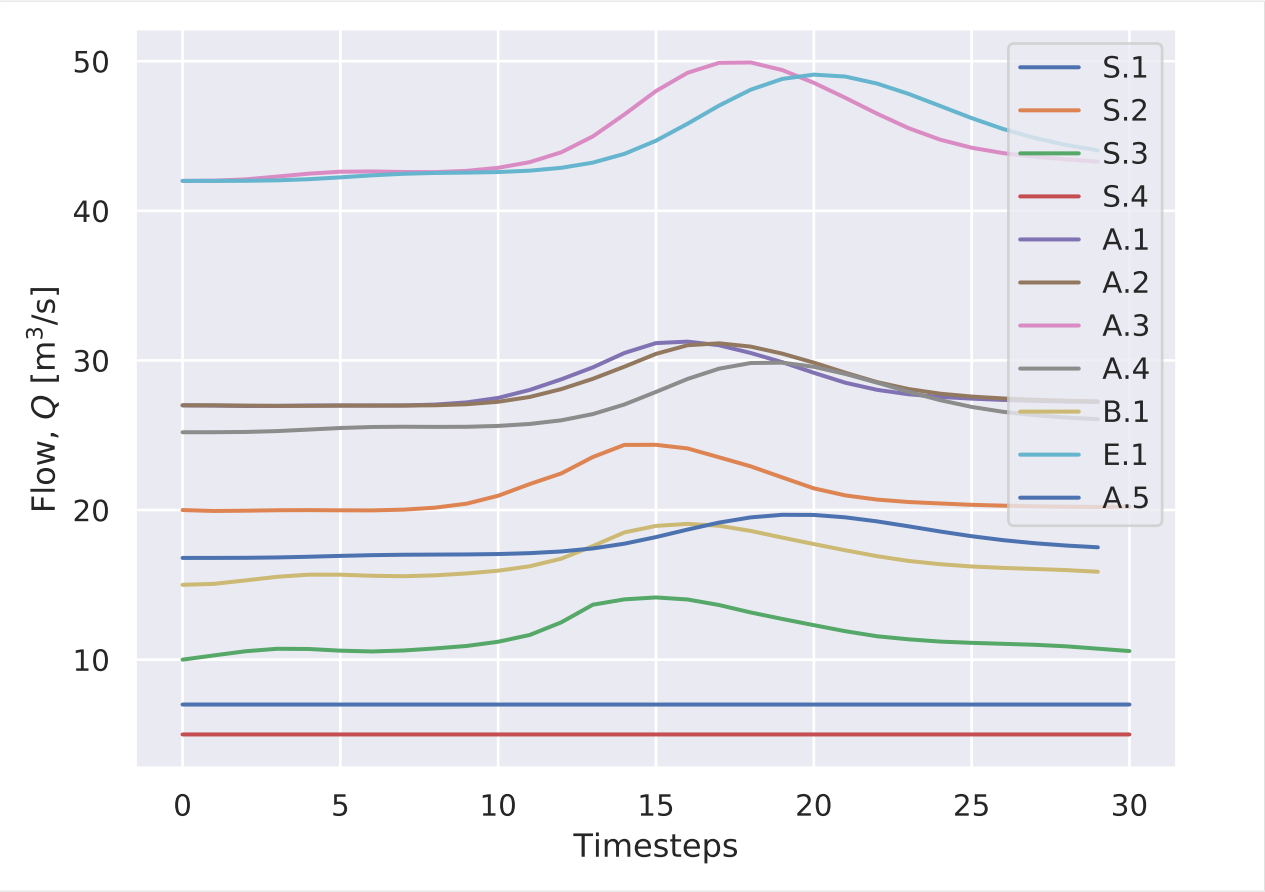
[6]:
```
structure1.set_constant_flow('S.1',31)
structure1.set_wave('S.2',shape_number=5,strength=5)
structure1.set_wave('S.3',shape_number=90,strength=5)
structure1.set_constant_flow('S.4',31)
structure1.draw_Qin(only_sources=True,figsize=(7,4))
```

[6]:
```
(<Figure size 672x384 with 1 Axes>,
 <matplotlib.axes._subplots.AxesSubplot at 0x7f94db287c88>)
```



[7]:
```
structure1.calc_flow_propagation(30)
structure1.draw_Qin(figsize=(7,5))
```

[7]:
```
(<Figure size 672x480 with 1 Axes>,
 <matplotlib.axes._subplots.AxesSubplot at 0x7f94db33a710>)
```



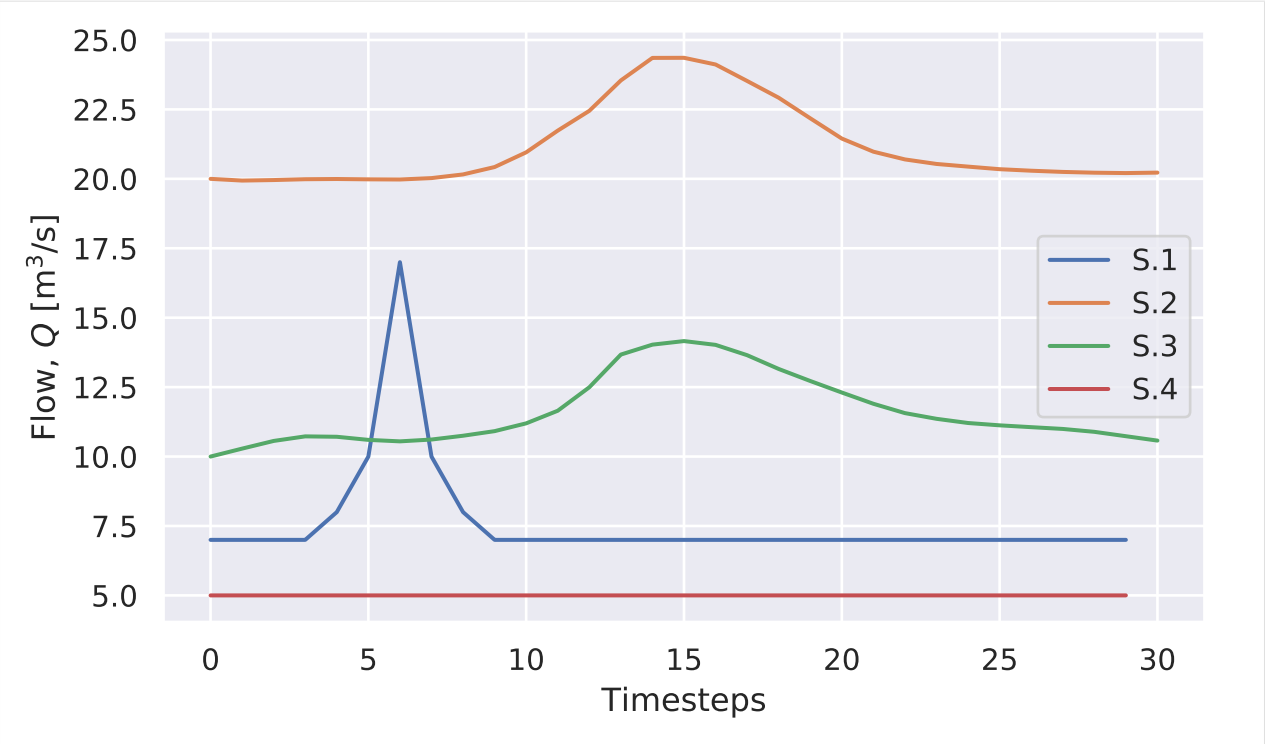
Warning: Coloring should change to improve readability. E.g. cluster nodes and give similar colors.

1.6.4 Experiment 2

Now it is possible to add any other extra inflow. In the following experiment a peak flow is added to S.1.

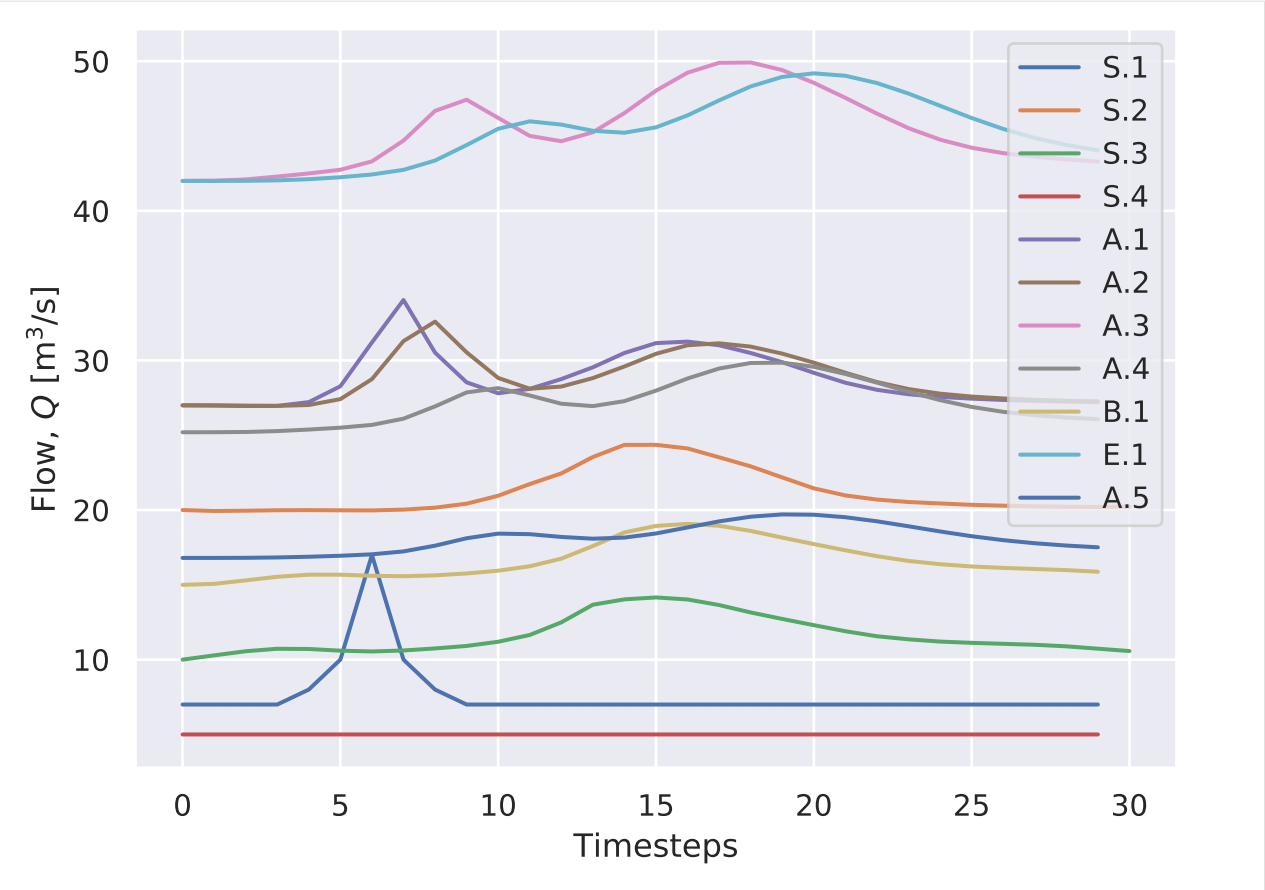
[8]:
```
shape = np.zeros(30)
shape[4] = 1
shape[5] = 3
shape[6] = 10
shape[7] = 3
shape[8] = 1
structure1.set_shape('S.1',30,shape)
structure1.set_wave('S.2',shape_number=5,strength=5)
structure1.set_wave('S.3',shape_number=90,strength=5)
structure1.set_constant_flow('S.4',30)
structure1.draw_Qin(only_sources=True,figsize=(7,4))
```

[8]:
```
(<Figure size 672x384 with 1 Axes>,
 <matplotlib.axes._subplots.AxesSubplot at 0x7f94db315438>)
```



[9]:
```
structure1.calc_flow_propagation(30)
structure1.draw_Qin(figsize=(7,5))
```

[9]:
```
(<Figure size 672x480 with 1 Axes>,
 <matplotlib.axes._subplots.AxesSubplot at 0x7f94db401908>)
```



1.7 Isolate watershed

[1]:
```
from context import show_function
from main_data import create_watershed
```

[2]:
```
%load_ext autoreload
%autoreload 2
```

1.7.1 Script to isolate the watershed

steps involved:

- filter on reaches in Asia and flow larger than 1 m^3
- isolate watershed starting at latest reach

In this script the watershed starting from the padma river, is isolated from the HydroSheds dataset.

By running main_data.py in the case folder, the hydrosheds data is downloaded and processed. The reaches are first filtered to select the reaches in Asia with a flow larger than 1 m^3. This is done with the function `filter_gloric()` the result is saved as `'data_gloric/padma_gloric_1m3_final.shp'` The code to extract a watershed looks as follows:

[3]:
```
show_function(create_watershed)
```

```
def create_watershed():
    print('Creating watershed')
    destination_filename = 'data_gloric/padma_gloric_1m3_final.shp'
    if not os.path.isfile(destination_filename):
        start_id = 41067217
        gloric_orig = gp.read_file('data_gloric/gloric_asia_1m3.shp')
        gloric = gloric_orig.copy()
        gloric = gloric.set_index('Reach_ID',drop=False)

        print('\tCreating spatial index')
        start = time.time()
        spatial_index = gloric.sindex
        end = time.time()
        print('\t\tDuration: '+ '{:.2f}'.format(end - start) + 's')
        id_set = GloricHydrosheds.get_watershed(spatial_index,gloric,start_id)
        selection = gloric.loc[id_set]
        selection.to_file(destination_filename)

    if not os.path.isfile('data_gloric/padma_gloric_1m3_final_no_geo.pkl'):
        gp.read_file('data_gloric/padma_gloric_1m3_final.shp')\
            .drop(columns={'geometry'})\
            .to_pickle('data_gloric/padma_gloric_1m3_final_no_geo.pkl')
```

A spatial index is created to search fast. The `get_watershed()` function from `GloricHydrosheds.py` extracts the watershed. It starts with an ID, selects the inflowing reach and adds it to the selection. For each found reach, the same procedure is repeated, until no more reaches can be found. The result is a shapefile that contains the reaches of the Ganges-Brahmaputra watershed

1.7.2 Plotting the result

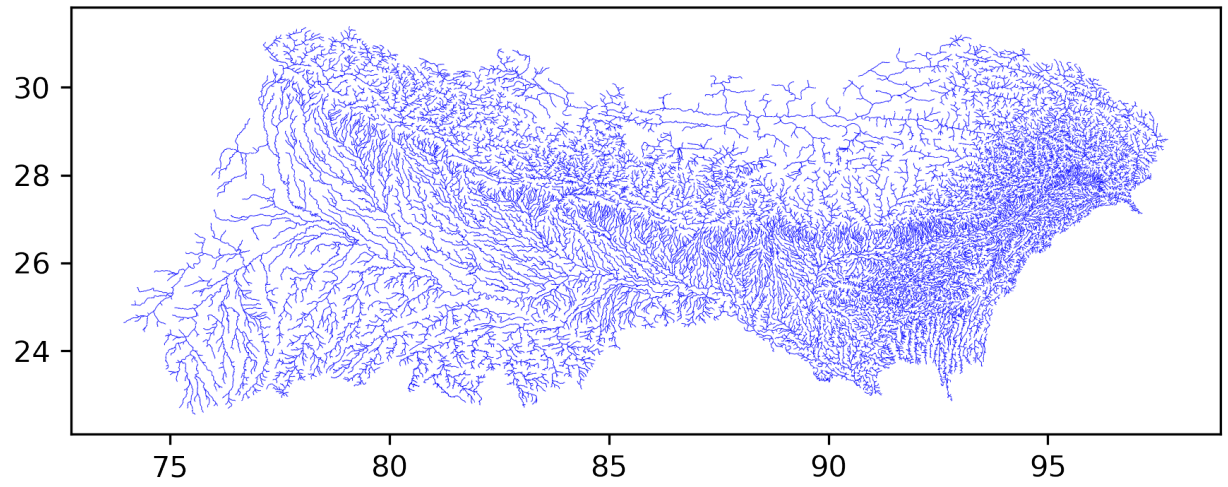
[4]:
```
import geopandas as gp
from matplotlib import pyplot as plt
```

[5]:
```
filename = 'data_gloric/padma_gloric_1m3_final.shp'
```

[6]:
```
rivernetwork = gp.read_file(filename)
```

The first option is to simply use Geopandas to plot the watershed. This gives some insight in the shape and structure.

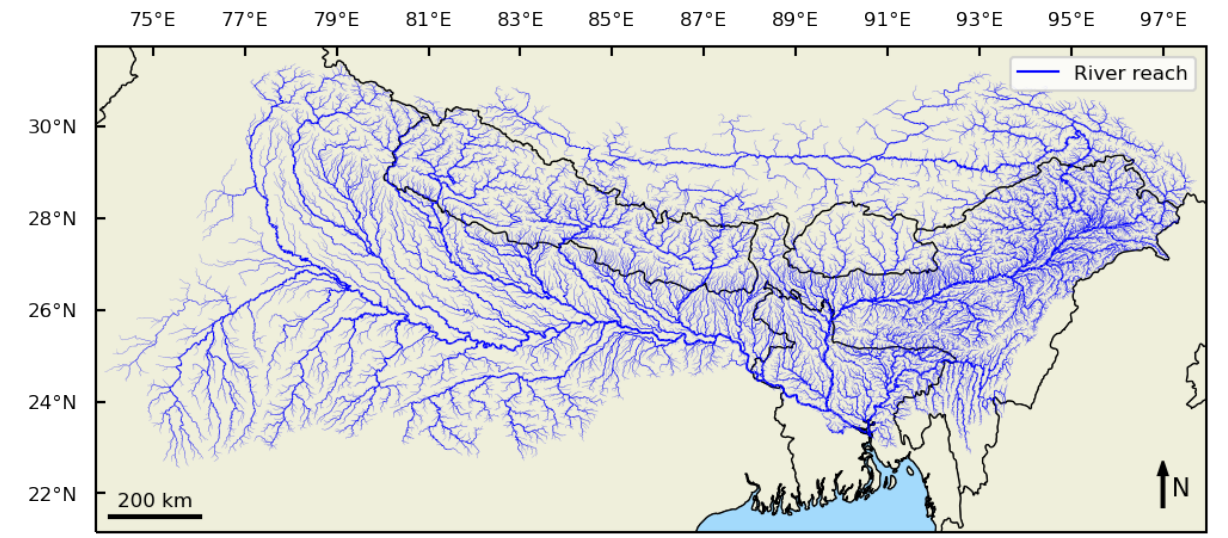
[7]:
```
fig = plt.figure(figsize=(7,7),dpi=300)
ax = fig.add_subplot(111)
rivernetwork.plot(ax=ax,color='b',linewidth=0.2);
```



[8]:
```
from custom_plot import *
```

The river network becomes clearer when the average flow is scaled with thickness.

[11]:
```
plot_river_map(rivernetwork);
```

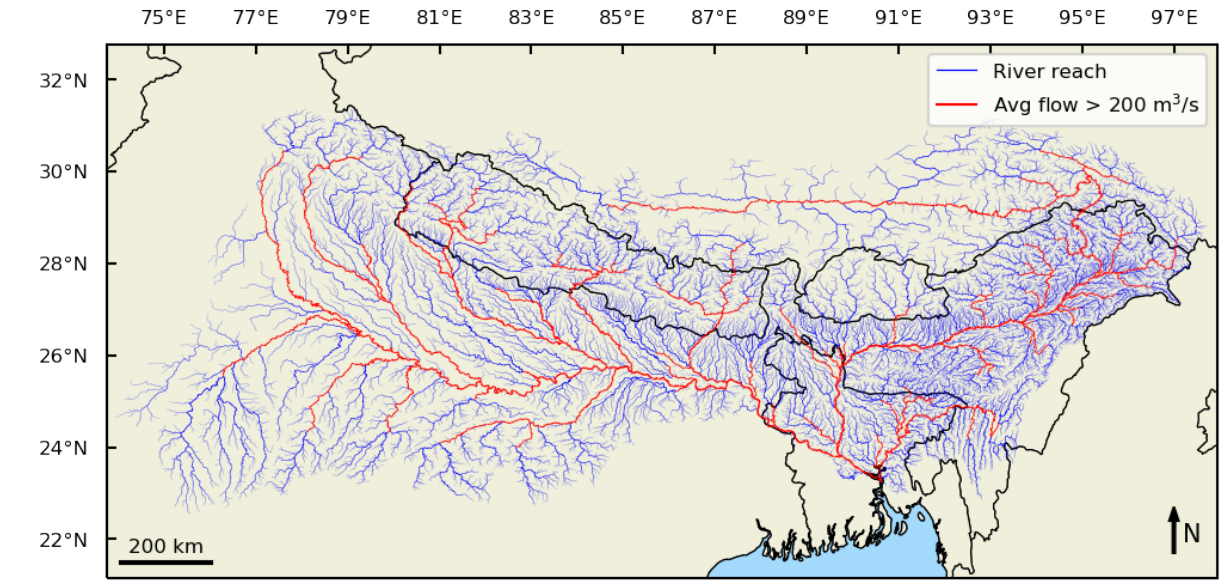


Note: Right click on any figure and select open image in new tab to see full resolution images.

1.7.3 Comparing with actual map

The main river system becomes clear, when we highlight the reaches with an average flow larger than 200 m^3/s. This figure shows many similarities with an actual river map of the area. One notable difference is that the HydroSHEDS dataset does not contain bifurcations. This can be seen for example in Bangladesh where the Hoogly river is missing that flows through Kolkata.

[12]:
```
plot_river_map_2(rivernetwork,split = 200);
```



[ ]:
```
#plot_river_map_2(rivernetwork,split = 200,figsize=(6.2,6.2), printoption = True,␣
↪filename = '../../thesis/report/figs/watershed.png');
```

1.8 Simulation initialisation and run

[1]:
```
import context
```

[2]:
```
%matplotlib inline
%load_ext autoreload
%autoreload 2
```

[3]:
```
from RiverNetworkCase import RiverNetwork
import pandas as pd
import time
```

[4]:
```
river_data = 'data_gloric/padma_gloric_1m3_final_no_geo.pkl'
watersheds_data = 'data_gloric/areas_gloric_no_geo.pkl'
rain_data = 'data_pmm/20130616-S000000-E002959-20130617-S233000-E235959.pkl'
result_data = 'data_results/overflow_130616_130617_2.pkl'
```

[5]:
```
padma = pd.read_pickle(river_data)
padma = padma.set_index('Reach_ID',drop=False)
padma.head()
```

[5]:

Reach_ID	Reach_ID	Next_down	Length_km	Log_Q_avg	Log_Q_var	Class_hydr	\
40763397	40763397	40764488	6.753	0.98529	0.60554	13	
40894470	40894470	40894471	1.246	0.22194	0.35383	12	
40894471	40894471	40893932	1.244	0.36154	0.35388	12	
40763399	40763399	40762358	2.722	0.61847	0.60323	13	
40763402	40763402	40763652	1.544	3.13560	0.48803	43	

Reach_ID	Temp_min	CMI_indx	Log_elev	Class_phys	Lake_wet	Stream_pow	\
40763397	16.1	-0.441	2.25768	311	0	0.19790	
40894470	16.9	0.228	2.47712	331	0	0.05036	
40894471	17.0	0.231	2.48287	331	0	0.08349	
40763399	16.1	-0.393	2.24055	321	0	0.08569	
40763402	16.6	0.026	2.16732	321	1	0.00000	

Reach_ID	Class_geom	Reach_type	Kmeans_30
40763397	11	311	4
40894470	11	511	11
40894471	11	511	11
40763399	11	611	4
40763402	21	643	24

[6]:
```
start = time.time()
rivernetwork = RiverNetwork(river_data, rain_data, x = 0.2, speed = 2, runoff_coeff =␣
↪0.5, t_max = 2*24*60)
end = time.time()
print('Duration: {:.2f}'.format(end - start) + 's')
```

```
Duration: 298.56s
```

[5]:
```
start = time.time()
rivernetwork.calculate_flows()
```

(continues on next page)

(continued from previous page)

```
end = time.time()
print('Duration: {:.2f}'.format(end - start) + 's')
```

```
Duration: 828.75s
```

[6]:
```
overflow = rivernetwork.get_overflow()
```

[7]:
```
overflow.to_pickle(result_data)
```

[ ]:

[8]:
```
import pickle
f = open("data_results/raw_object_130616_130617_2.pkl","wb")
pickle.dump(rivernetwork,f)
f.close()
```

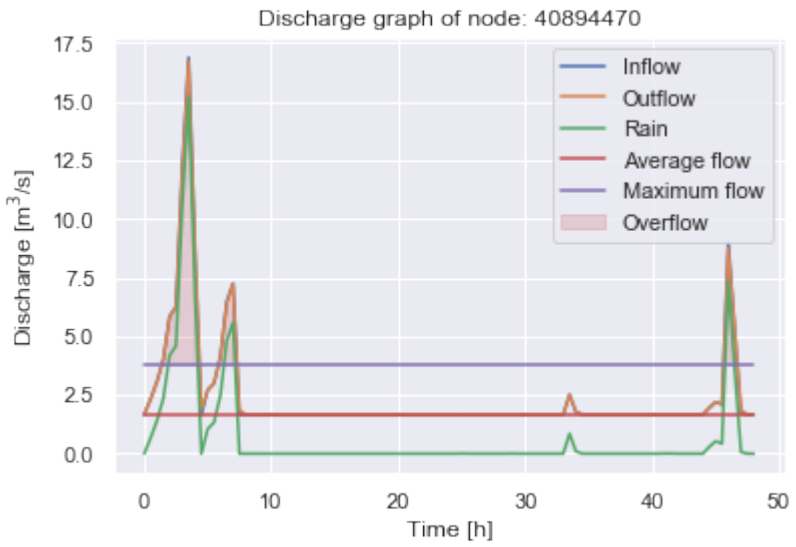
[ ]:

[ ]:

[9]:
```
for node_str in rivernetwork.G.nodes:
    node = rivernetwork.G.nodes[node_str]
    if node['source'] == True:
        #print(node);
        break;
```

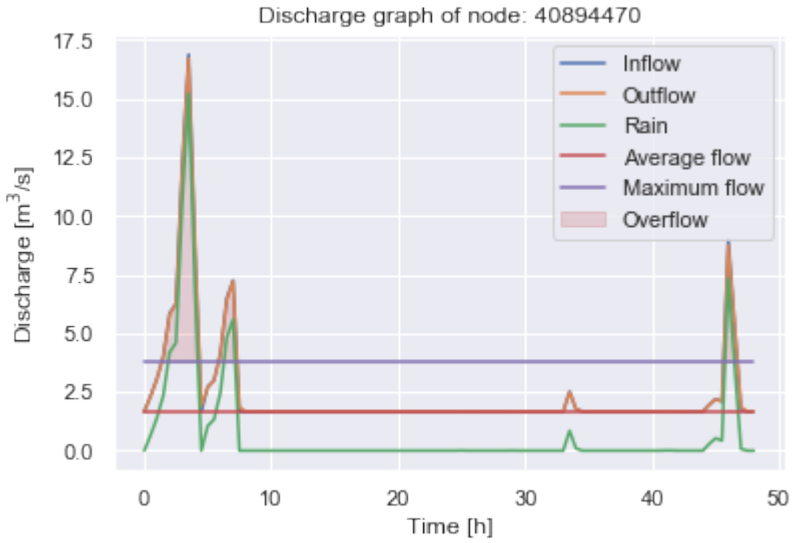
[10]:
```
rivernetwork.plot_node_flows(node_str)
```

[10]:
```
(<Figure size 432x288 with 1 Axes>,
 <matplotlib.axes._subplots.AxesSubplot at 0x121773950>)
```



[11]:
```
rivernetwork.plot_node_flows(node_str)
```

[11]:
```
(<Figure size 432x288 with 1 Axes>,
 <matplotlib.axes._subplots.AxesSubplot at 0x1a28397810>)
```

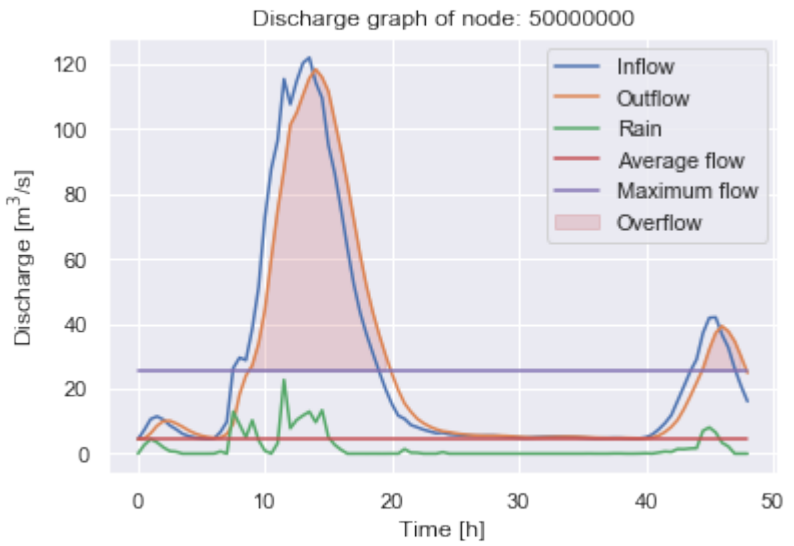


[12]:
```
len(rivernetwork.get_node(50000000)['Qout'])
```

[12]:
```
97
```

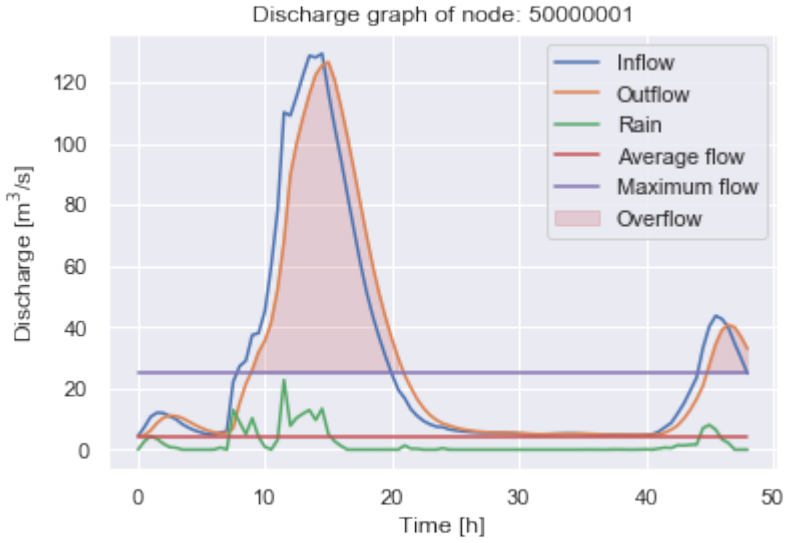
[13]:
```
rivernetwork.plot_node_flows(50000000)
```

[13]:
```
(<Figure size 432x288 with 1 Axes>,
 <matplotlib.axes._subplots.AxesSubplot at 0x1a37553650>)
```



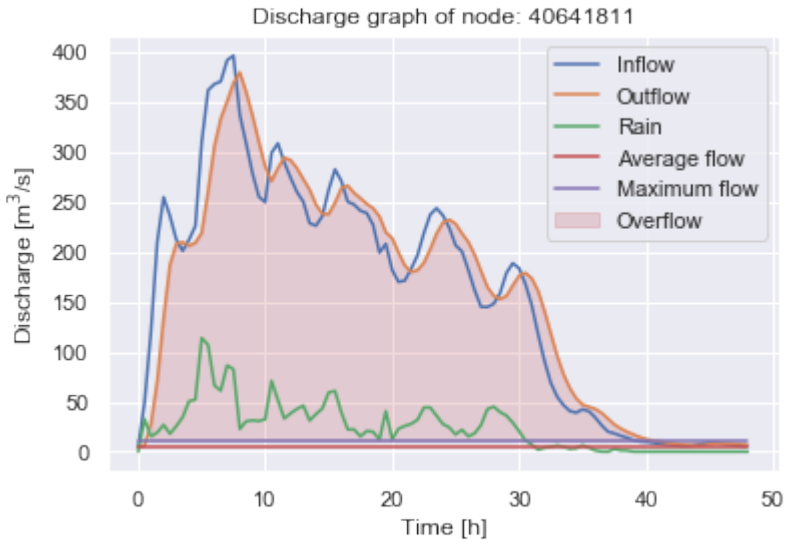
[14]:
```
rivernetwork.plot_node_flows(50000001)
```

[14]:
```
(<Figure size 432x288 with 1 Axes>,
 <matplotlib.axes._subplots.AxesSubplot at 0x1a3f540610>)
```



[15]:
```
rivernetwork.plot_node_flows(40641811)
```

[15]:
```
(<Figure size 432x288 with 1 Axes>,
 <matplotlib.axes._subplots.AxesSubplot at 0x1a37faba50>)
```



[ ]:
```
```

1.9 Plotting results

1.9.1 Overflow

[1]:
```
import context
import geopandas as gp
import pandas as pd
```

[2]:
```
%load_ext autoreload
%autoreload 2
```

Loading the files: one with the river geometries, the other with the overflow results

[3]:
```
river_geo = 'data_gloric/padma_gloric_1m3_final.shp'
result_data = 'data_results/overflow_130616_130617_2.pkl'
river = gp.read_file(river_geo).set_index('Reach_ID')
results = pd.read_pickle(result_data)
```

Creating a maximum flow by taking the max over the time axis.

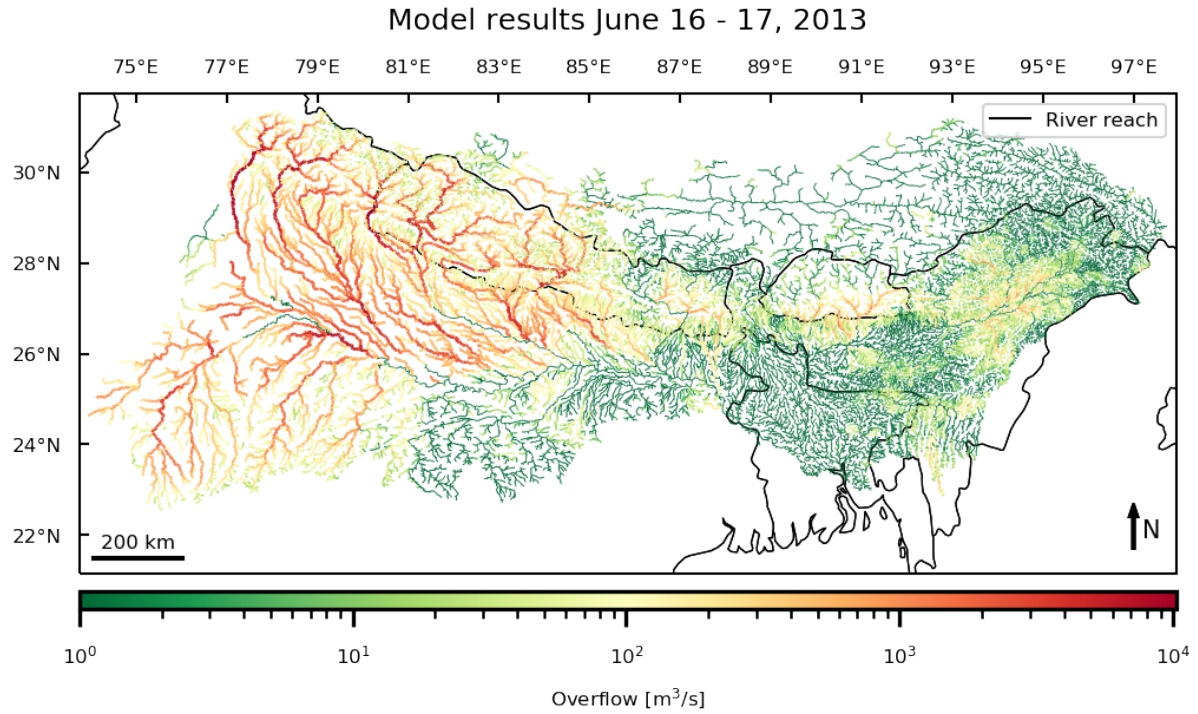
[4]:
```
max_overflow = pd.DataFrame(results.max(axis=1))\
        .rename({0:'max_overflow'},axis=1)\
        .join(river[['geometry']])
max_overflow = gp.GeoDataFrame(max_overflow).reset_index()
max_overflow.crs = {'init':'EPSG:4326'}
max_overflow.head()
```

[4]:

	Reach_ID	max_overflow	geometry
0	40633088	74.218533	LINESTRING (79.09374999999974 31.3854166666661...
1	40633506	114.017808	LINESTRING (79.0812499999997 31.35624999999951...
2	40633927	129.560853	LINESTRING (79.07708333333304 31.3437499999995...
3	40634367	121.140140	LINESTRING (77.88958333333304 31.3270833333328...
4	40634489	111.109322	LINESTRING (78.00208333333305 31.3229166666661...

[5]:
```
from custom_plot import *
```

[7]:
```
plot_results_map(max_overflow);
```



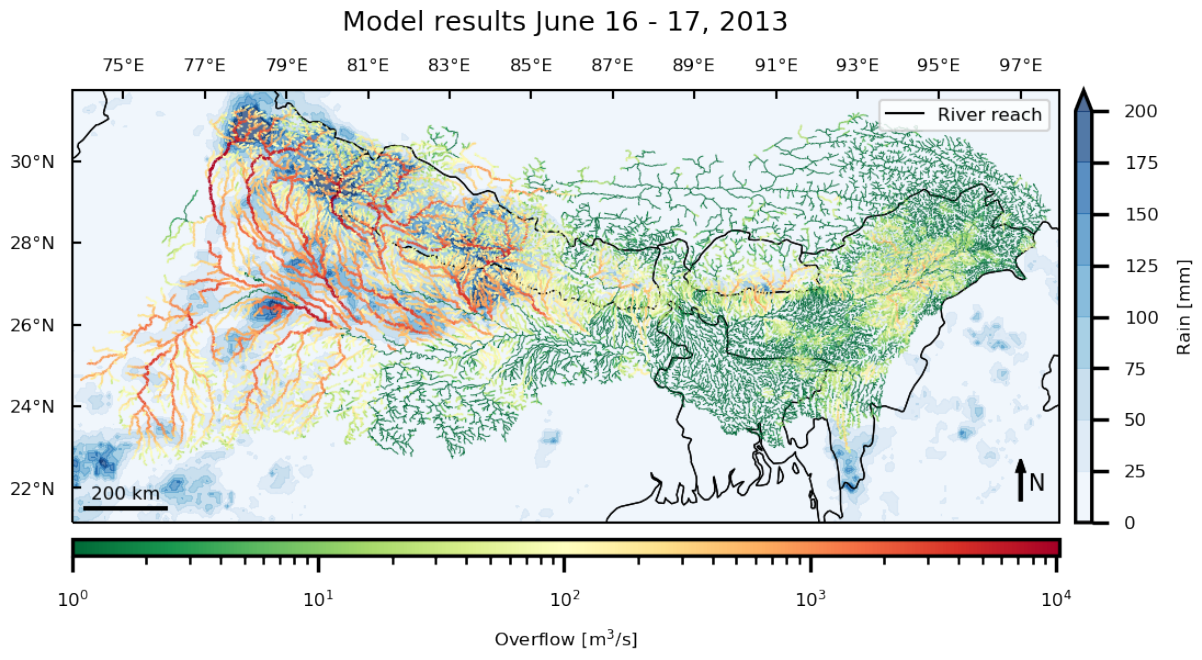
[ ]:
```
#plot_results_map(max_overflow, figsize = (9.3,6.2) ,printoption = True, filename = '.
↪./../thesis/report/figs/result_basin_large_4.pdf');
```

1.9.2 Overflow with rain

We are adding rainfall data agregated over the two days.

[8]:
```
import helper_functions
import globals
bounds = globals.bounds(0.2)
west, south, east, north = bounds
south -= 1.2
north += 0.2
bounds = west, south, east, north
dates = globals.dates()
filenames = helper_functions.get_hdf_list(dates[2:4])
gpm_data = helper_functions.GPM(filenames[0],bounds)
newLats, newLons = gpm_data.coordinates(bounds)
total_rain = np.zeros(gpm_data.get_crop().shape)
for filename in filenames:
    gpm_data = helper_functions.GPM(filename,bounds)
    #gpm_data.save_cropped_tif()
    total_rain += gpm_data.get_crop()
total_rain = total_rain/2
rain_data = (total_rain, newLats, newLons)
```

[9]:
```
plot_results_map_rain(max_overflow,rain_data);
```



[ ]:
```
#plot_results_map_rain(max_overflow, rain_data, figsize = (9.3,6.2) ,printoption =␣
↪True, filename = '../../thesis/report/figs/result_rain_large_2.pdf');
```

1.10 Correlation between rainfall and overflow

[1]:
```
import context
import pandas as pd
import seaborn as sns
from matplotlib import pyplot as plt
```

[2]:
```
sns.set()
sns.set_context("paper", rc={"font.size":8.0,
                             'lines.linewidth':1,
                             'patch.linewidth':0.5,
                             "axes.titlesize":8,
                             "axes.labelsize":8,
                             'xtick.labelsize':8,
                             'ytick.labelsize':8,
                             'legend.fontsize':8 ,
                             'pgf.rcfonts' : False})
```

[72]:
```
result_data = 'data_results/overflow_130616_130617_2.pkl'
results = pd.read_pickle(result_data)
max_overflow = pd.DataFrame(results.max(axis=1))\
        .rename({0:'max_overflow'},axis=1)
#max_overflow.head()
```

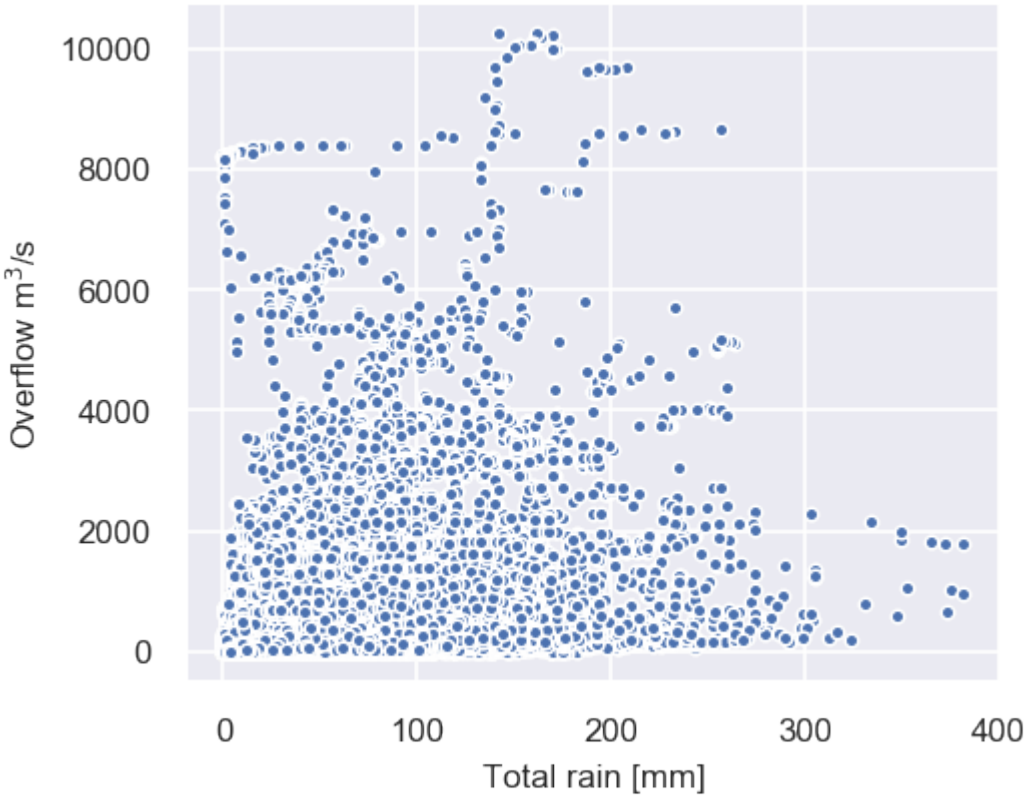
[73]:
```
rain_data = 'data_pmm/20130616-S000000-E002959-20130617-S233000-E235959.pkl'
rain_dataf = pd.read_pickle(rain_data).set_index('Reach_ID')
area = rain_dataf[['area_sk']]
rain = rain_dataf.drop('area_sk',axis=1)
```

[76]:
```
rain_flow = pd.DataFrame(rain\
            .sum(axis=1))\
            .rename({0:'total_rain_mmh'},axis=1)\
            .join(max_overflow)
rain_flow['total_rain_mm'] = rain_flow['total_rain_mmh'] / 2
rain_flow.head()
```

[76]:

Reach_ID	total_rain_mmh	max_overflow	total_rain_mm
40763397	126.416490	745.046714	63.208245
40894470	23.594527	12.946060	11.797264
40894471	45.016647	16.197745	22.508323
40763399	91.566055	308.959039	45.783027
40763402	181.004435	3655.129534	90.502218

[77]:
```
fig = plt.figure(figsize=(3.5,3),dpi=150)
fig.patch.set_alpha(0)
ax = fig.add_subplot(111)
ax = sns.scatterplot(data=rain_flow,x='total_rain_mm',y='max_overflow',s = 10,
↪edgecolor='white')
plt.ylabel('Overflow m$^3$/s');
plt.xlabel('Total rain [mm]');
#plt.savefig('../../thesis/report/figs/scatter.pdf', bbox_inches = 'tight')
```



We can obtain a measure for locality of effect by plotting rainfall versus overflow for each reach. This is done in the figure. If rainfall causes a local overflow, we would expect an correlation between increasing rainfall and overflow in that reach. If that is not the case, rainfall causes overflows downstream than there is no clear correlation between the two. The figure shows that there is no clear correlation between rainfall and overflow within the same reach. This figure shows that the effect of rainfall is not local, but downstream. This is an indication of the cascading effect.

Another observation from the plot are the dots that form horizontal patterns, for example just above 8000 m^3/s. These patters exist because of subsequent connected reaches. If a reach is flooded, the neighbouring reaches are likely to be flooded by the same amount. These dots at the same overflow level, with slightly different rainfall appear as 'lines' in the graph.

1.11 Chapter 5 figures

1.11.1 Figure 5.1

[1]:
```
%load_ext autoreload
%autoreload 2
```

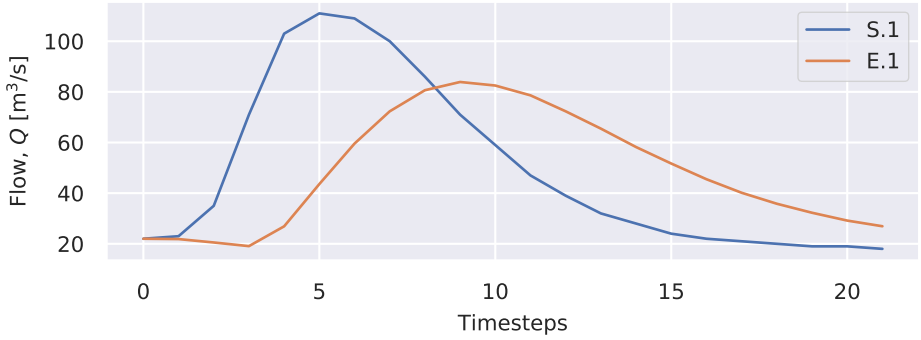
[2]:
```
import pandas as pd
import networkx as nx
import matplotlib.pyplot as plt
import numpy as np
```

[3]:
```
from context import RiverNetwork
from RiverNetwork import RiverNetwork
```

[4]:
```
structure2 = RiverNetwork('../data/single-segment-karahan.xlsx',dt=6)
```

[5]:
```
inflow = np.array(pd.read_excel('../data/example-inflow-karahan.xlsx').Inflow)
structure2.set_shape('S.1',21,inflow-22)
structure2.calc_flow_propagation(22)
```

[6]:
```
import seaborn as sns
import pickle
import matplotlib.pyplot as plt
sns.set()
sns.set_context("paper", rc={"font.size":8.0,
                             'lines.linewidth':1,
                             'patch.linewidth':0.5,
                             "axes.titlesize":8,
                             "axes.labelsize":8,
                             'xtick.labelsize':8,
                             'ytick.labelsize':8,
                             'legend.fontsize':8 ,
                             'pgf.rcfonts' : False})
(fig,ax) = structure2.draw_Qin(figsize=(5,2))
#fig.set_size_inches(5,3)
fig.set_dpi(150)
fig.patch.set_alpha(0)
#plt.title('Single reach example')
plt.tight_layout()
#plt.savefig('../../thesis/report/figs/karahan.pgf', bbox_inches = 'tight')
```



1.11.2 Figure 5.2

Link

1.11.3 Figure 5.3

[7]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

(continues on next page)

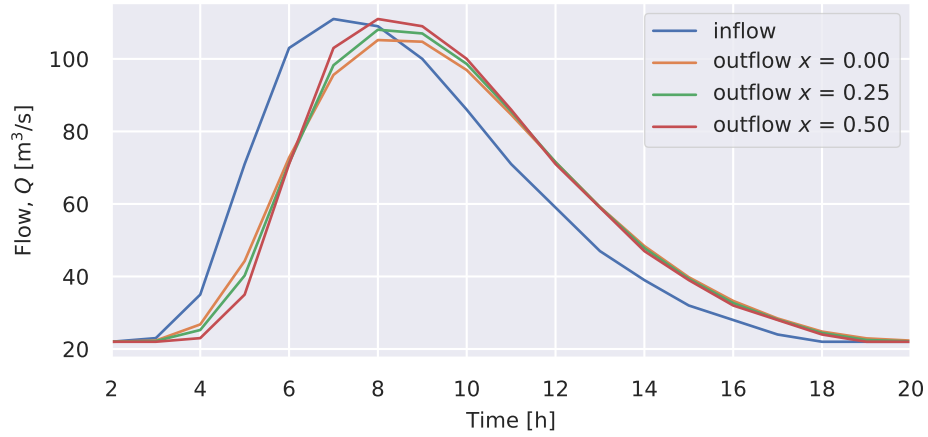
(continued from previous page)

```
from context import fit_muskingum
from fit_muskingum import getParams
from fit_muskingum import calc_Out
from fit_muskingum import calc_C
```

[8]:
```
df = pd.read_excel('../data/example-inflow-karahan-adjusted.xlsx')
df = df.set_index('Time')
```

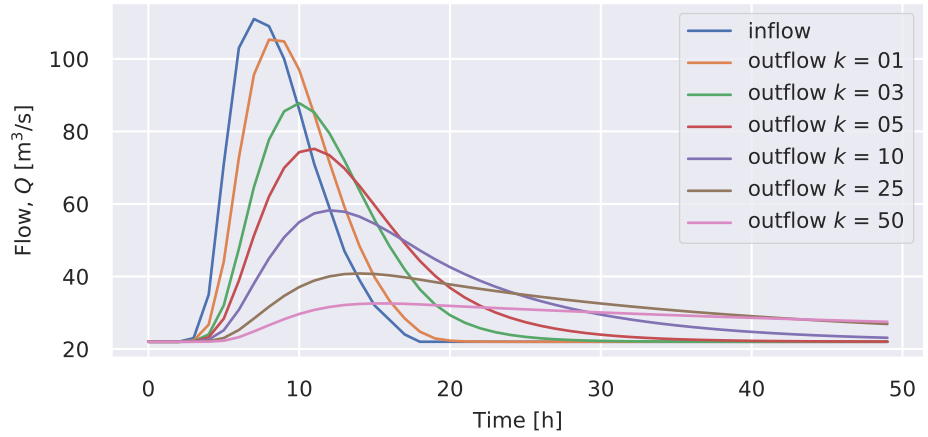
[9]:
```
import seaborn as sns
sns.set()
sns.set_context("paper", rc={"font.size":8.0,
                             'lines.linewidth':1,
                             'patch.linewidth':0.5,
                             "axes.titlesize":8,
                             "axes.labelsize":8,
                             'xtick.labelsize':8,
                             'ytick.labelsize':8,
                             'legend.fontsize':8 ,
                             'pgf.rcfonts' : False})
```

[10]:
```
t = df.index.values
I = np.array(df['Inflow'])

fig = plt.figure(figsize=(5,2.5),dpi=150)
fig.patch.set_alpha(0)
ax = fig.add_subplot(111)

plt.plot(t,I,linewidth = 1 , label = 'inflow')

for x in [0,0.25,0.5]:
    k = 1
    dt = 1
    out = calc_Out(I,calc_C(k,x,dt))
    plt.plot(t, out,linewidth = 1, label = 'outflow $x$ = ' + '{:1.2f}'.format(x))


plt.ylabel('Flow, $Q$ [m$^3$/s]')
plt.xlabel('Time [h]')
plt.legend()
plt.xlim(2,20);
plt.tight_layout()
#plt.savefig('../../thesis/report/figs/1reachx.pgf', bbox_inches = 'tight')
```



1.11.4 Figure 5.4

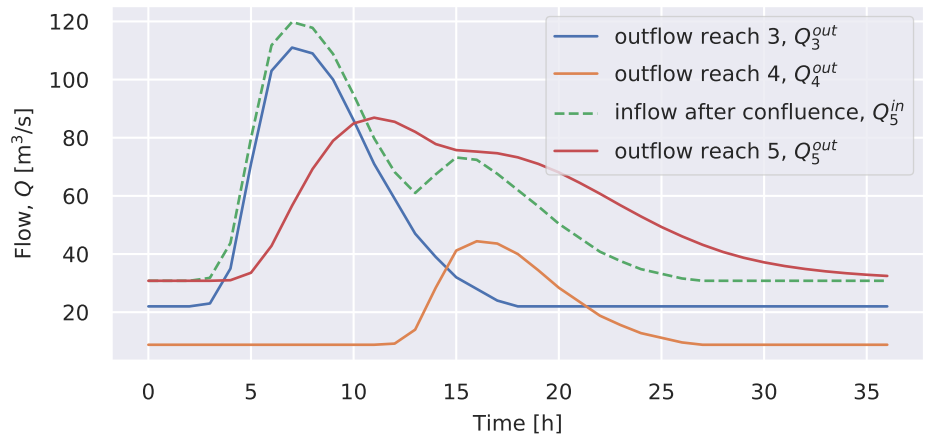
[11]:
```
t = df.index.values
I = np.array(df['Inflow'])

length = 50
t = range(0,length,1)
I = np.append(I,np.full((1,length - len(I)),22))

fig = plt.figure(figsize=(5,2.5),dpi=150)
fig.patch.set_alpha(0)
ax = fig.add_subplot(111)

plt.plot(t,I,linewidth = 1 , label = 'inflow')

klist = [1,3,5,10,25,50]
for k in klist:
    x = 0.01
    dt = 1
    out = calc_Out(I,calc_C(k,x,dt))
    plt.plot(t, out,linewidth = 1, label = 'outflow $k$ = ' + '{:02d}'.format(k))

plt.ylabel('Flow, $Q$ [m$^3$/s]')
plt.xlabel('Time [h]')
plt.legend();
plt.tight_layout()
#plt.savefig('../../thesis/report/figs/1reachk.pgf', bbox_inches = 'tight')
```



1.11.5 Figure 5.5

[12]:
```
df = pd.read_excel('../data/example-inflow-karahan-adjusted.xlsx')
df = df.set_index('Time')
```

[13]:
```
t = df.index.values
I = np.array(df['Inflow'])
I2 = np.array(df['Inflow'])*0.4
I2 = np.append(I2[28:37],I2[0:28])
fig = plt.figure(figsize=(5,2.5),dpi=150)
ax = fig.add_subplot(111)
fig.patch.set_alpha(0)

plt.plot(t,I,linewidth = 1 , label = 'outflow reach 3, $Q^{out}_3$')
plt.plot(t,I2,linewidth = 1 , label = 'outflow reach 4, $Q^{out}_4$')
plt.plot(t,I+I2,'--',linewidth = 1 , label = 'inflow after confluence, $Q^{in}_5$')
plt.rcParams.update({'font.size': 8, 'pgf.rcfonts' : False})

x = 0.1
k = 5
dt = 1

C0 = calc_C(k,x,dt) # k,x,dt
O0 = calc_Out(I+I2,C0)
plt.plot(t, O0 ,'r',linewidth = 1, label = 'outflow reach 5, $Q^{out}_5$')

plt.ylabel('Flow, $Q$ [m$^3$/s]')
plt.xlabel('Time [h]')
plt.legend()
# save to file
plt.tight_layout()
#plt.savefig('../../thesis/report/figs/1confluence.pgf', bbox_inches = 'tight')
```



1.11.6 Figure 5.6

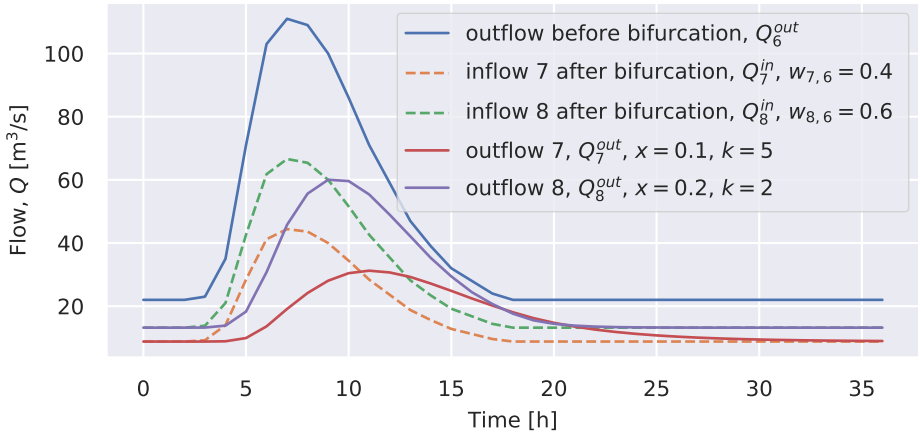
[14]:
```
df = pd.read_excel('../data/example-inflow-karahan-adjusted.xlsx')
df = df.set_index('Time')
```

[15]:
```
t = df.index.values
I = np.array(df['Inflow'])

frac = 0.4
I1 = np.array(df['Inflow'])*frac
I2 = np.array(df['Inflow'])*(1-frac)
fig = plt.figure(figsize=(5,2.5),dpi=150)
ax = fig.add_subplot(111)

fig.patch.set_alpha(0)

plt.plot(t,I,linewidth = 1 , label = 'outflow before bifurcation, $Q^{out}_6$')

plt.plot(t,I1,'--',linewidth = 1 , label = 'inflow 7 after bifurcation, $Q^{in}_7$,
↪$w_{7,6}=0.4$ ')
plt.plot(t,I2,'--',linewidth = 1 , label = 'inflow 8 after bifurcation, $Q^{in}_8$,
↪$w_{8,6}=0.6$ ')

x = 0.1
k = 5
dt = 1
C1 = calc_C(k,x,dt) # k,x,dt
O1 = calc_Out(I1,C1)
plt.plot(t, O1 ,linewidth = 1, label = 'outflow 7, $Q^{out}_7$, $x=0.1$, $k=5$')

x = 0.2
k = 2
dt = 1
C2 = calc_C(k,x,dt) # k,x,dt
O2 = calc_Out(I2,C2)
plt.plot(t, O2 ,linewidth = 1, label = 'outflow 8, $Q^{out}_8$, $x=0.2$, $k=2$')

plt.ylabel('Flow, $Q$ [m$^3$/s]')
```

(continues on next page)

(continued from previous page)

```
plt.xlabel('Time [h]')
plt.legend()
# save to file
plt.tight_layout()
#plt.savefig('../../thesis/report/figs/1bifurcation.pgf', bbox_inches = 'tight')
```



[ ]:

1.12 Chapter 7 figures

1.12.1 Figure 7.1

[1]:
```
import context
```

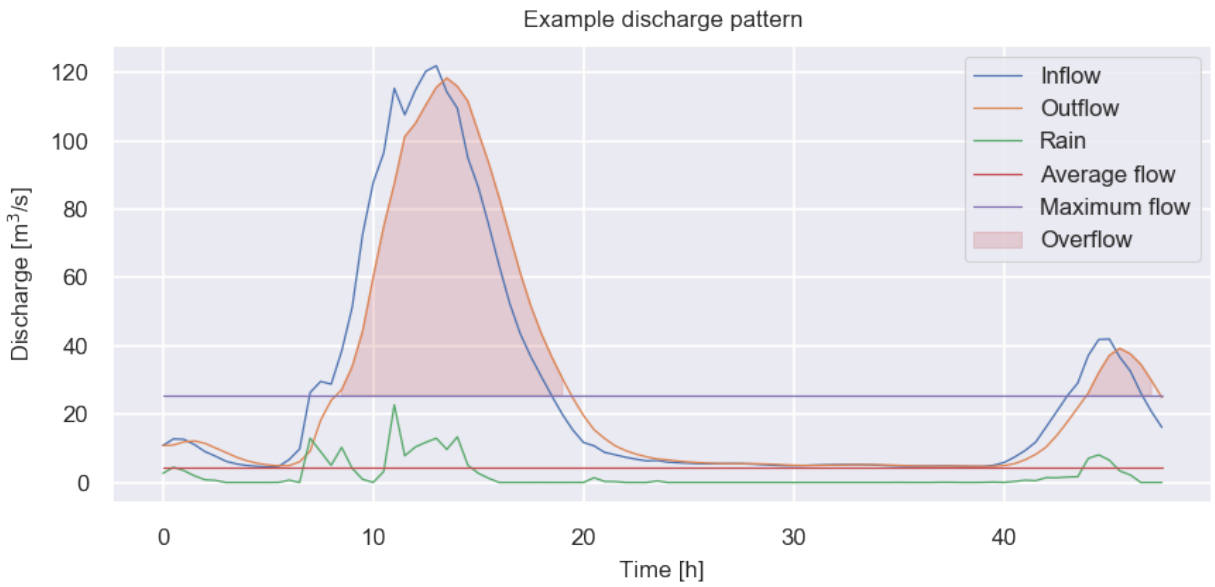
[2]:
```
import seaborn as sns
import pickle
import matplotlib.pyplot as plt
f = open("data_results/raw_object_130616_130617.pkl","rb")
raw_data = pickle.load(f)
sns.set()
#sns.set_style("whitegrid")
sns.set_context("paper", rc={"font.size":8.0,
                             'lines.linewidth':0.6,
                             'patch.linewidth':0.5,
                             "axes.titlesize":8,
                             "axes.labelsize":8,
                             'xtick.labelsize':8,
                             'ytick.labelsize':8,
                             'legend.fontsize':8 ,
                             'pgf.rcfonts' : False})
```

[3]:
```
(fig,ax) = raw_data.plot_node_flows(50000000)
fig.set_size_inches(6,3)
fig.patch.set_alpha(0)
fig.set_dpi(150)
```

(continues on next page)

(continued from previous page)

```
plt.title('Example discharge pattern')
plt.tight_layout()
#plt.savefig('../../thesis/report/figs/discharge_example.pdf', bbox_inches = 'tight')
#plt.savefig('../../thesis/report/figs/discharge_example.pgf', bbox_inches = 'tight')
```



1.12.2 Figure 7.2

Link

1.12.3 Figure 7.3

Link

[ ]: