litholog

Release 0.1

Ross Meyer, Zane Jobe

Mar 08, 2022

TUTORIALS:

`litholog` is focused on providing a framework to digitize, store, plot, and analyze sedimentary graphic logs (example log shown below).

Graphic logs are the most common way geologists characterize and communicate the composition and variability of clastic sedimentary successions; through a simple drawing, a graphic log imparts complex geological concepts (e.g., the Bouma turbidite sequence or a shoreface parasequence). The term 'graphic log' originates from a geologist graphically drawing (i.e., 'logging') an outcrop or core; other synonymous terms include measured section and stratigraphic column.

`litholog` is a package-level extension of agile-geoscience/striplog, with additional features that focus on lithology, and an API that is geared toward facilitating machine learning and quantitative analysis.

The package provides two primary data structures:

- `Bed`, which stores data from one lithologic bed or unit (e.g., top, base, lithology, thickness, grain size, etc). `Bed` is equivalent to a `striplog.Interval`
- `BedSequence`, which stores a collection of `Beds` in stratigraphic order (either elevation or depth order). `BedSequence` is equivalent to a `striplog.Striplog`

CHAPTER

ONE

UTILITIES

Several utilities for working with graphic logs are included with litholog:

- default lithology colors for Beds that can be easily modified
- transformations for grain-size data from millimeter (mm) to log2 (a.k.a. *psi*) units, which are far easier to work with than mm.
- calculation of the following metrics at the BedSequence level:
  - net-to-gross
  - amalgamation ratio
  - psuedo gamma ray log
  - Hurst statistics (for determining facies clustering)

CHAPTER

TWO

DATA

The data provided with this demo come from two papers, and all logs were digitized using the Matlab digitizer included with this release.

- 7 logs from Jobe et al. 2012 (html, pdf).
- 6 logs from Jobe et al. 2010 (html, pdf).

2.1 `litholog` basics

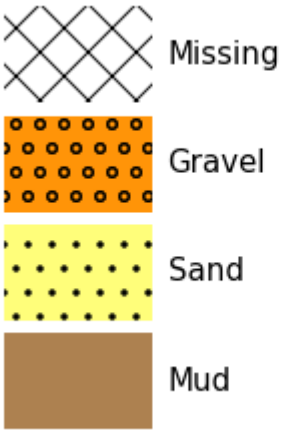
[1]:
```
# import some libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
plt.style.use('ggplot')

import litholog
from litholog import utils, Bed
from litholog.sequence import io, BedSequence

from striplog import Component
```

2.1.1 Default Colors and Legend

[2]:
```
#defaults legend for plotting
litholog.defaults.litholegend
```

[2]:
```
Legend(Decor({'component': Component({'lithology': 'mud'}), '_colour': '#ad8150', 'hatch
↪': None, 'width': -6.0})
Decor({'component': Component({'lithology': 'sand'}), '_colour': '#fffe7a', 'hatch': '.',
↪ 'width': -1.0})
Decor({'component': Component({'lithology': 'gravel'}), '_colour': '#ff9408', 'hatch': 'o
↪', 'width': 4.0})
Decor({'component': Component({'lithology': 'missing'}), '_colour': '#ffffff', 'hatch':
↪'x', 'width': -1.0}))
```

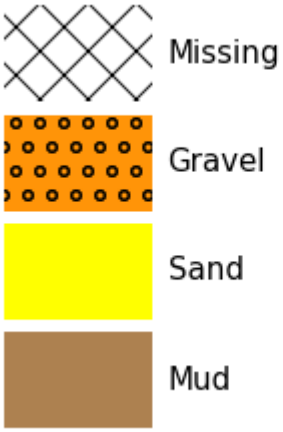
[3]:
```
# and this is how Beds will look when plotted
litholog.defaults.litholegend.plot()
```



[4]:
```
# modify the legend
litholog.defaults.sand_decor.colour = 'blue'

# and see if it worked
litholog.defaults.litholegend
```

[4]:
```
Legend(Decor({'component': Component({'lithology': 'mud'}), '_colour': '#ad8150', 'hatch
↪': None, 'width': -6.0})
Decor({'component': Component({'lithology': 'sand'}), '_colour': '#0000ff', 'hatch': '.',
↪ 'width': -1.0})
Decor({'component': Component({'lithology': 'gravel'}), '_colour': '#ff9408', 'hatch': 'o
↪', 'width': 4.0})
Decor({'component': Component({'lithology': 'missing'}), '_colour': '#ffffff', 'hatch':
↪'x', 'width': -1.0}))
```

[5]:
```
# modify things again
litholog.defaults.sand_decor.colour = 'yellow'
litholog.defaults.sand_decor.hatch = None

# call the plot function - note sand doesnt have dots any more
litholog.defaults.litholegend.plot()
```



2.1.2 Make a Bed and a BedSequence from scratch

[6]:
```
# Make a Bed

# make some fake data
top, base = 1, 2
data = {'lit1': 5, 'arr1': [1,2,3], 'arr2': [4,5,6]}

# assign to a Bed
B = litholog.Bed(top, base, data)
print(B.order) # litholog determines order based on whether the top is larger than the␣
↪base (see more below)
B
```

```
depth
```

[6]:
```
Interval({'data': {'lit1': 5, 'arr1': [1, 2, 3], 'arr2': [4, 5, 6]}, 'top': Position({
↪'middle': 1.0, 'units': 'm'}), 'base': Position({'middle': 2.0, 'units': 'm'}),
↪'description': '', 'components': []})
```

If we want to make a more realistic example, here is one below. We need the `Component` from `striplog` to assign the `primary` lithology to each bed. We also use `data` and `metadata` from litholog to store other data, and could also add `metadata`

[7]:
```
bed1 = Bed(top = 1, base = 0, data = {'grain_size_mm':0.125}, components = [Component({
↪'lithology' : 'sand'})])
bed2 = Bed(top = 1.1, base = 1, data = {'grain_size_mm':0.02}, components = [Component({
↪'lithology' : 'mud'})])
bed3 = Bed(top = 1.8, base = 1.1, data = {'grain_size_mm':50}, components = [Component({
↪'lithology' : 'gravel'})])

bed_x = Bed(top=0, base=1, data={'lithology':'sand'})

print(bed2,'\n')

print(bed1.order)
print(bed_x.order,'\n')

print(bed1['lithology'])
```

```
{'data': {'grain_size_mm': 0.02}, 'top': Position({'middle': 1.1, 'units': 'm'}), 'base':
↪ Position({'middle': 1.0, 'units': 'm'}), 'description': '', 'components': [Component({
↪'lithology': 'mud'})]}

elevation
depth

None
```

[8]:
```
seq1 = BedSequence([bed1, bed2, bed3],metadata={'name':'litholog test BedSequence'})

print(seq1.metadata)

# let's access the first bed in the sequence
seq1[0] # first bed is the uppermost because elevation-ordered
```

```
{'name': 'litholog test BedSequence'}
```

[8]:
```
Interval({'data': {'grain_size_mm': 50}, 'top': Position({'middle': 1.8, 'units': 'm'}),
↪'base': Position({'middle': 1.1, 'units': 'm'}), 'description': '', 'components':␣
↪[Component({'lithology': 'gravel'})]})
```

[9]:
```
#access the tops in the sequence using list comprehension
print('Bed tops:',[bed.top.upper for bed in seq1],'\n')

# To access the data fields, use "get_field"
print('grain size data:',seq1.get_field('grain_size_mm'),'\n')

# you can also look at the summary of a bed
print('uppermost bed summary:',seq1[0].summary(),'\n')

# or all the summaries
print('Summaries',[bed.summary() for bed in seq1],'\n')
```

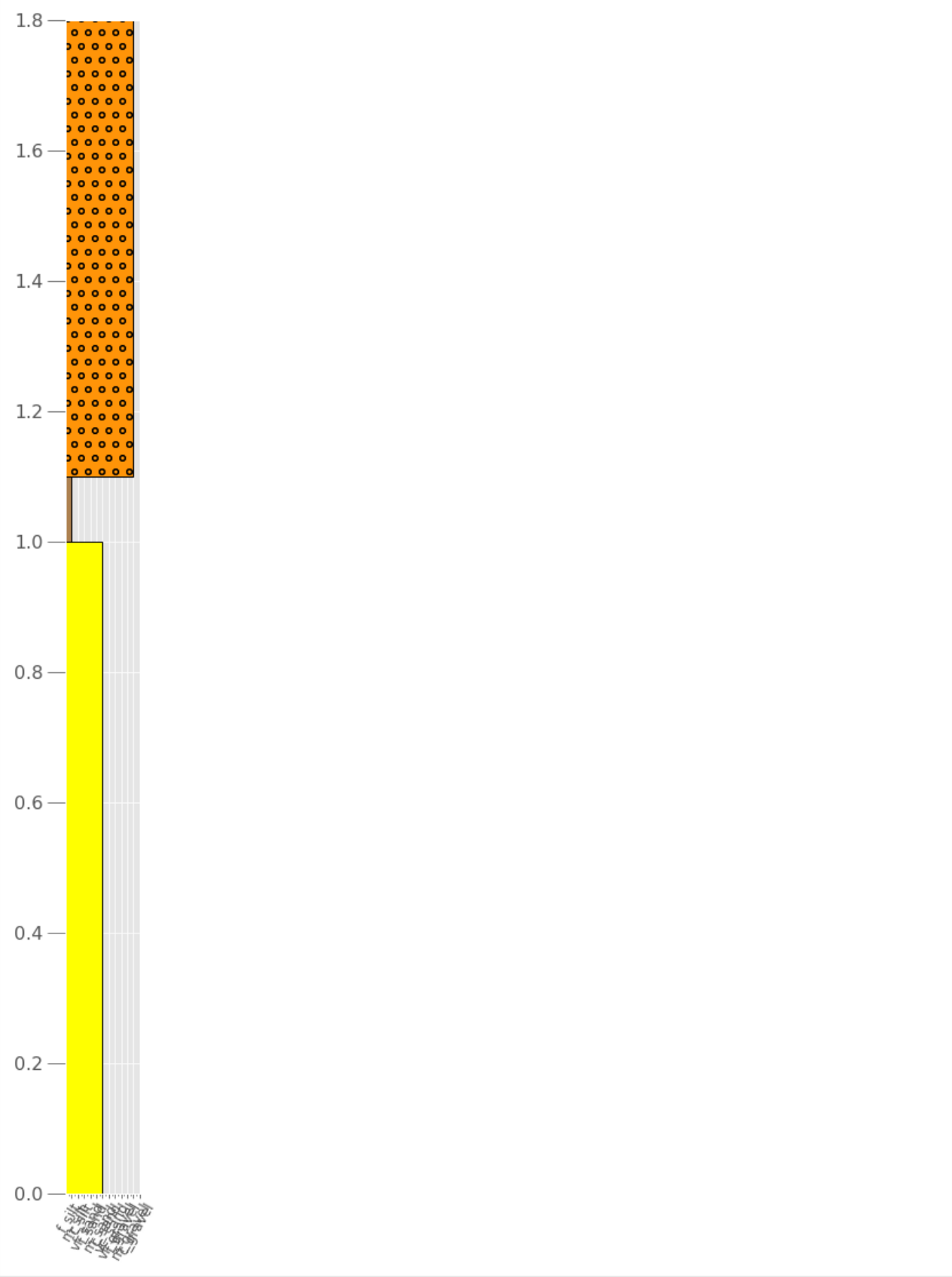
```
Bed tops: [1.8, 1.1, 1.0]

grain size data: [5.00e+01 2.00e-02 1.25e-01]

uppermost bed summary: 0.70 m of gravel

Summaries ['0.70 m of gravel', '0.10 m of mud', '1.00 m of sand']
```

2.1.3 Plotting a BedSequence

2.1.3.1 Simple plotting

With no arguments, the aspect of the figure is default to 10 and the width of each bed (i.e., the grain size) is the default for the lithology given in `primary`, not the data in `grain_size_mm`. See below on how to further control this:

[10]:
```
seq1.plot()
```

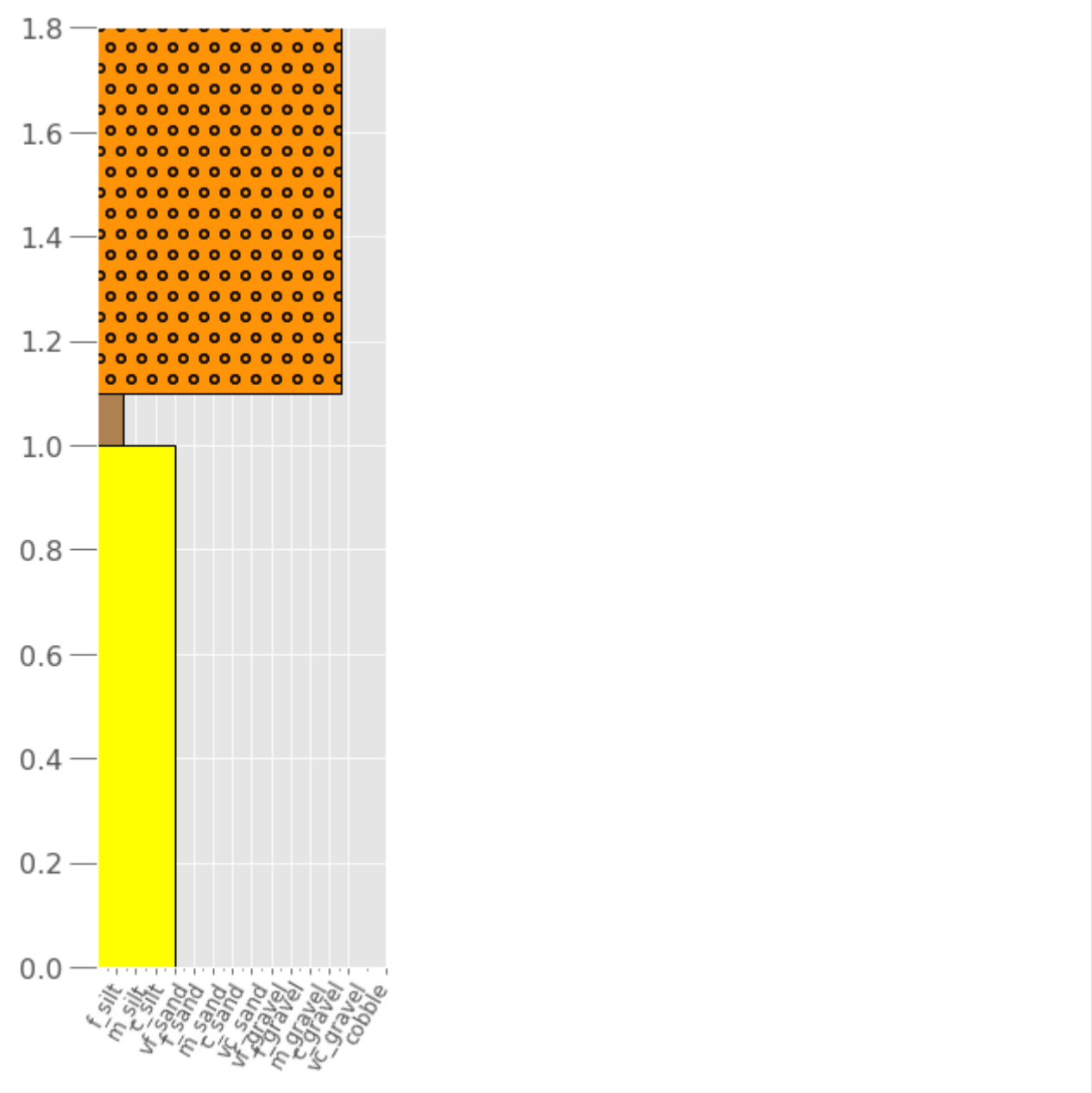

1.8
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0

If we want to make a nicer plot that usess the exaact grain size, we need to make a log2 grain-size, which is much easier to plot and visualize rather than using millimeters. We use the functions of `wentworth` to do this, and we will create PSI units instead of PHI units, because they increase with increasing grain size:

[11]:
```
# make a grain_size_psi column
for bed in seq1:
    bed.data['grain_size_psi'] = litholog.wentworth.gs2psi(bed.data['grain_size_mm'])

seq1[-1] # see that it happened for the lowermost bed
```

[11]:
```
Interval({'data': {'grain_size_mm': 0.125, 'grain_size_psi': -3.0}, 'top': Position({
↪'middle': 1.0, 'units': 'm'}), 'base': Position({'middle': 0.0, 'upper': 0.0, 'lower':␣
↪0.0, 'units': 'm'}), 'description': '', 'components': [Component({'lithology': 'sand'}
↪)]})
```

[12]:
```
# Now we can make a nicer looking plot that uses the grain size data

# set up a figure
fig, ax = plt.subplots(figsize=[3,10])

# call the plot method, using the grain_size_psi as the width_field
seq1.plot(ax=ax,
          legend=litholog.defaults.litholegend,
          width_field='grain_size_psi',
          wentworth='fine'
         )
plt.show()
# The plot below might not look drastically different from the one above, but it is␣
↪several PSI units different...
```



2.1.3.2 Adding intra-Bed grain-size data to a plot

What if you want to display grain-size profiles within a bed? Not to worry, you just need to feed litholog that data:

[13]:
```
# let's just change the lower sand bed to have a fining-up profile. I chose exact PSI␣
↪units here, but you get the idea
bed1 = Bed(top = 1, base = 0, data = {'depth_m':[0, 0.05, 0.1, 0.9, 1],'grain_size_mm':
↪[1, 0.5, 0.25, 0.125, 0.0884]}, components = [Component({'lithology' : 'sand'})])

# and the conlomgerate bed to have coarsening up
bed3 = Bed(top = 1.8, base = 1.1, data = {'depth_m':[1.1, 1.2, 1.5, 1.8],'grain_size_mm':
↪[5, 20, 30, 80]}, components = [Component({'lithology' : 'gravel'})])
```

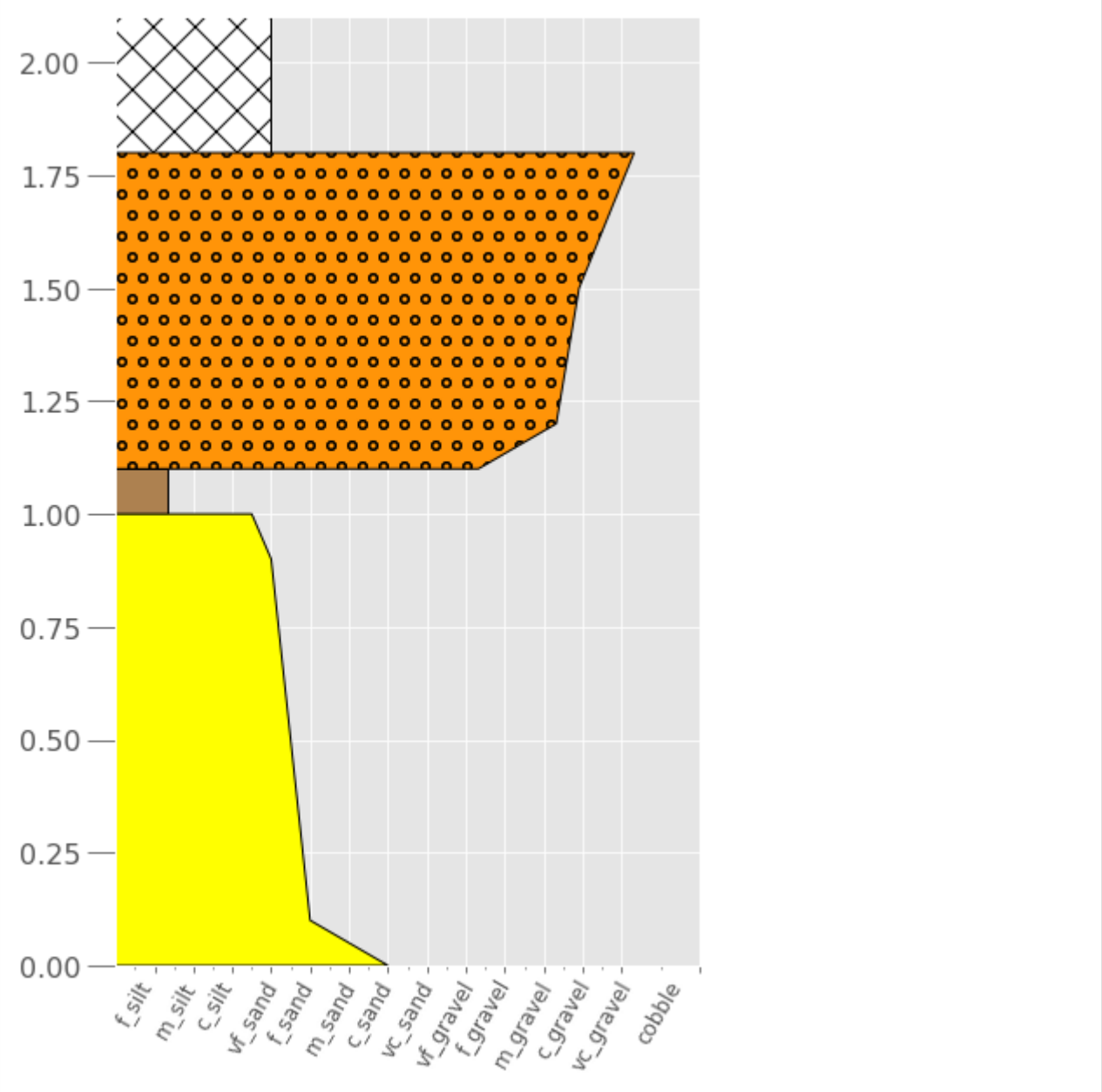
(continues on next page)

(continued from previous page)

```
# and let's also add a missing (i.e., covered) interval at the top:
# NOTE - you can make whatever grain size you want for missing intervals... I choose 0.
↪125 so it plots nicely
bed4 = Bed(top = 2.1, base = 1.8, data = {'grain_size_mm':0.125}, components =␣
↪[Component({'lithology' : 'missing'})])

# make a new BedSequence
seq2 = BedSequence([bed1, bed2, bed3, bed4])

# create grain_size_psi
for bed in seq2:
    bed.data['grain_size_psi'] = litholog.wentworth.gs2psi(bed.data['grain_size_mm'])

# and look at the covered 'bed'
seq2[1]
```

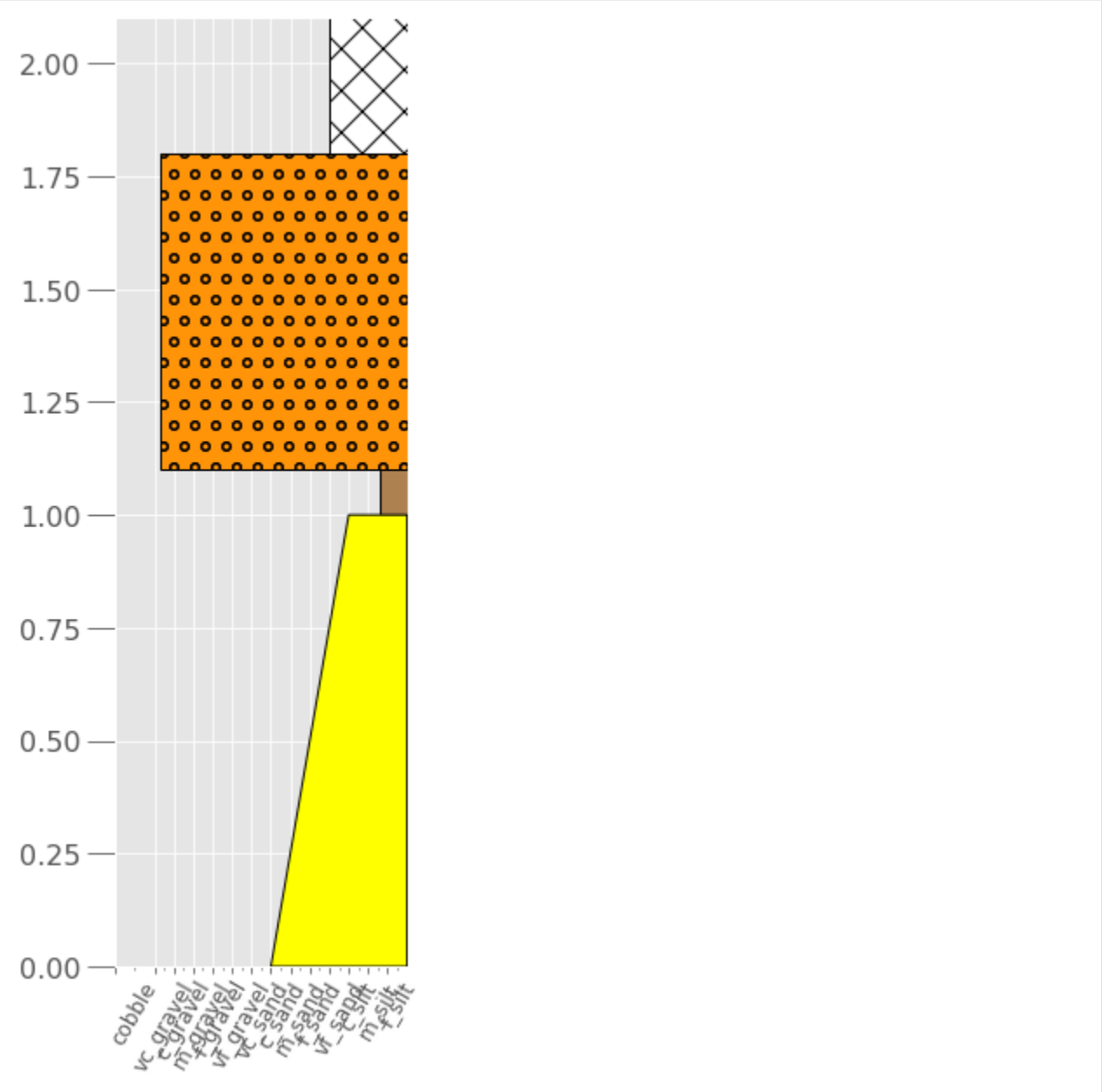
[13]:
```
Interval({'data': {'depth_m': [1.1, 1.2, 1.5, 1.8], 'grain_size_mm': [5, 20, 30, 80],
↪'grain_size_psi': array([2.32192809, 4.32192809, 4.9068906 , 6.32192809])}, 'top':␣
↪Position({'middle': 1.8, 'units': 'm'}), 'base': Position({'middle': 1.1, 'units': 'm'}
↪), 'description': '', 'components': [Component({'lithology': 'gravel'})]})
```

[43]:
```
# Plot it just like before, but we need to add a depth field
fig, ax = plt.subplots(figsize=[6,10])

seq2.plot(ax=ax,
          legend=litholog.defaults.litholegend,
          width_field='grain_size_psi',
          depth_field='depth_m',
          wentworth='fine',
         )

# and let's save it out
fig.savefig('fig2.eps',format='eps')
```

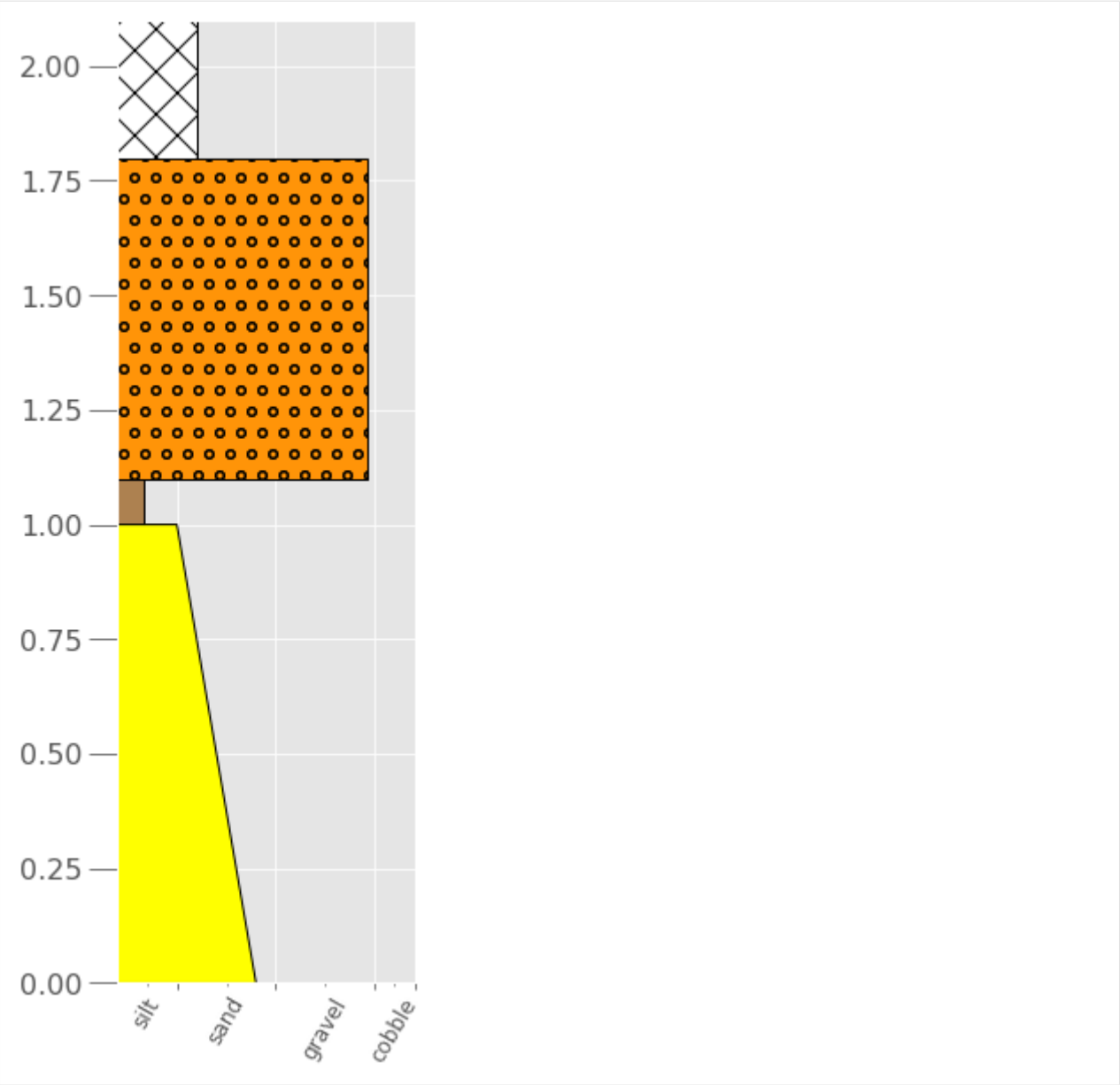


2.1.3.3 Other plotting methods

[15]:
```
# And if you want to plot it Exxon-style, just add that argument
fig, ax = plt.subplots(figsize=[3,10])

seq2.plot(ax=ax,
          legend=litholog.defaults.litholegend,
          width_field='grain_size_psi',
          depth_field='depth_m',
          wentworth='fine',
          exxon_style=True
         );
```



[16]:
```
# Or if you want to make the x-axis simpler (can use fine, medium, or coarse):
fig, ax = plt.subplots(figsize=[3,10])

seq2.plot(ax=ax,
          legend=litholog.defaults.litholegend,
          width_field='grain_size_psi',
          depth_field='depth_m',
          wentworth='coarse',
          exxon_style=False
         );
```



2.2 Demo of `litholog` functionality using the included demo data

litholog is a package-level extension of agile-geoscience/striplog, with additional features that focus on lithology, and an API that is geared toward facilitating machine learning and quantitative analysis.

The package provides two primary data structures:

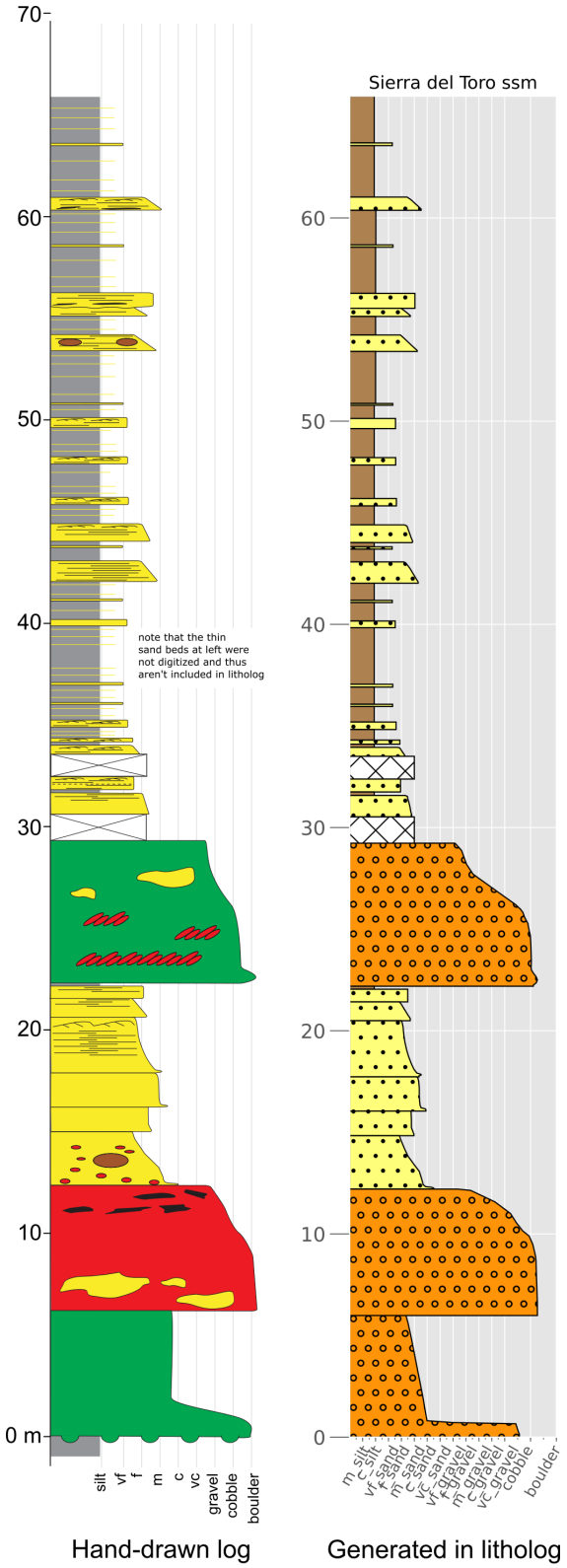
- `Bed`
  - stores data from one bed (e.g., top, base, lithology, thickness, grain size, etc).
  - is equivalent to a `striplog.Interval`
- `BedSequence`

- stores a collection of `Beds` in stratigraphic order
- is equivalent to a `striplog.Striplog`

Other utilities include: - transformations for grain-size data from millimeter (mm) to log2 (a.k.a. Psi) units, which are far easier to work with than mm. - calculation of the following metrics at the `BedSequence` level: - net-to-gross - amalgamation ratio - psuedo gamma ray log - Hurst statistics (for determining facies clustering) - default lithology colors

The data provided with this demo come from two papers, and all logs were digitized using the Matlab digitizer included with this release. - 7 logs from Jobe et al. 2012 (html, pdf) - 11 logs from Jobe et al. 2010 (html, pdf),

An example log from that paper is shown here, hand-drawn in vector-art software (left) and plotted with litholog (right):



Example graphic log

(South Side Margin log from Jobe et al. (2010) J. Sed. Res. doi:10.2110/jsr.2010.092

[1]:
```
# import stuff
import collections
import inspect

import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
plt.style.use('ggplot')

import litholog
from litholog import utils, Bed
from litholog.sequence import io, BedSequence

from striplog import Component
```

2.2.1 Load the demo data from a csv using pandas

This first step uses `utils` within litholog to converts depth-grainsize pairs (e.g., that define a fining-upward profile in a bed) into pandas-friendly arrays. The outputs of this will be the fields shown below (e.g., `depth_m`, `grain_size_mm`). If you have differently formatted csv data, this step may not apply, or you may need a different util.

[2]:
```
# Converts 'string' arrays to numpy
transforms = {c : utils.string2array_matlab for c in ['depth_m',
                                                      'grain_size_mm']}

# Read the demo data
df = pd.read_csv('../data/demo_data.csv', converters=transforms)

# counts and prints the names of the logs in the data file
print(len(df.name.unique()),
      'graphic logs imported:',
      '\n',
      df.name.unique())

df.head() # displays the first five rows of data
```

```
13 graphic logs imported:
 ['Karoo krf1' 'Karoo krf2' 'Karoo krf3' 'Karoo krf4' 'Karoo krf5'
 'Magnolia' 'Pukearuhue' 'Sierra del Toro dc1' 'Sierra del Toro dc2'
 'Sierra del Toro flame' 'Sierra del Toro h2o' 'Sierra del Toro ssm'
 'Sierra del Toro wc']
```

[2]:

	name	count	collection	eod	eodnum	tops	th	gs_tops_mm	snd_shl	mean_gs_mm	max_gs_mm	ng	ar
0	Karoo krf1	1	Skoorsteenberg	fan	1	21.042456	0.392483	0.173762	1.0	0.173762	0.173762	0.772615	0.104762
1	Karoo krf1	1	Skoorsteenberg	fan	1	20.649974	0.244327						
2	Karoo krf1	1	Skoorsteenberg	fan	1	20.405647	0.041588						
3	Karoo krf1	1	Skoorsteenberg	fan	1	20.364059	0.046786						
4	Karoo krf1	1	Skoorsteenberg	fan	1	20.317273	0.106568						

(continues on next page)

(continued from previous page)

```
1    0.031250      0.0    0.031250   0.031250  0.772615  0.104762
2    0.119996      1.0    0.119996   0.119996  0.772615  0.104762
3    0.031250      0.0    0.031250   0.031250  0.772615  0.104762
4    0.133082      1.0    0.133082   0.133082  0.772615  0.104762

              depth_m     grain_size_mm
0    [21.0425, 20.65]  [0.1738, 0.1738]
1    [20.65, 20.4056]  [0.0583, 0.0583]
2  [20.4056, 20.3641]      [0.12, 0.12]
3  [20.3641, 20.3173]  [0.0583, 0.0583]
4  [20.3173, 20.2107]  [0.1331, 0.1331]
```

The variable `df` is just a pandas DataFrame at this point - it hasn't been put into `litholog` format yet.

Notice that the columns `ng` (net-to-gross) and `ar` (amalgamation ratio) are the same for each graphic log. This csv was processed using Matlab, and `ng` and `ar` were calculated there. Not to worry, `litholog` can also calculate these metrics, which we will show you how to do below.

2.2.1.1 Use `wentworth` to make log2 grain size data

Before we translate the dataframe into `BedSequences`, let's create a log2 grain-size column. We will use the functionality of `wentworth` to do this, and we will create PSI units instead of PHI units, because they increase with increasing grain size. We simply take the `grain_size_mm` column and translate it to psi units (`grain_size_psi`):

[3]:
```
df['grain_size_psi'] = df.grain_size_mm.apply(lambda x: np.round(litholog.wentworth.
↪gs2psi(x),4))
df.head()
```

[3]:
```
         name  count      collection  eod  eodnum       tops        th  \
0  Karoo krf1      1  Skoorsteenberg  fan       1  21.042456  0.392483
1  Karoo krf1      1  Skoorsteenberg  fan       1  20.649974  0.244327
2  Karoo krf1      1  Skoorsteenberg  fan       1  20.405647  0.041588
3  Karoo krf1      1  Skoorsteenberg  fan       1  20.364059  0.046786
4  Karoo krf1      1  Skoorsteenberg  fan       1  20.317273  0.106568

   gs_tops_mm  snd_shl  mean_gs_mm  max_gs_mm        ng        ar  \
0    0.173762      1.0    0.173762   0.173762  0.772615  0.104762
1    0.031250      0.0    0.031250   0.031250  0.772615  0.104762
2    0.119996      1.0    0.119996   0.119996  0.772615  0.104762
3    0.031250      0.0    0.031250   0.031250  0.772615  0.104762
4    0.133082      1.0    0.133082   0.133082  0.772615  0.104762

              depth_m     grain_size_mm      grain_size_psi
0    [21.0425, 20.65]  [0.1738, 0.1738]  [-2.5245, -2.5245]
1    [20.65, 20.4056]  [0.0583, 0.0583]  [-4.1004, -4.1004]
2  [20.4056, 20.3641]      [0.12, 0.12]  [-3.0589, -3.0589]
3  [20.3641, 20.3173]  [0.0583, 0.0583]  [-4.1004, -4.1004]
4  [20.3173, 20.2107]  [0.1331, 0.1331]  [-2.9094, -2.9094]
```

2.2.2 Convert dataframe to BedSequences

This is the step that will convert our dataframe into `BedSequences` (equivalent to a `striplog.Striplog`) that contains `Beds` (equivalent to `striplog.Intervals`).

The component map sets the `primary` data for each `Bed` - see other ways to do this in the `litholog_basics.ipynb`

[4]:
```
# Columns shared by whole sequences (i.e., shared by an entire graphic log)
METACOLS = ['name', 'collection', 'ng', 'ar']

# Columns of bed-level data, including the psi column we just made
DATACOLS = ['th', 'gs_tops_mm', 'snd_shl', 'depth_m',
            'gs_tops_mm', 'mean_gs_mm', 'max_gs_mm', 'grain_size_mm','grain_size_psi']

# Convert dataframe to a list of `BedSequence`s
seqs = []
for group, values in df.groupby('name'):
    seqs.append(
        BedSequence.from_dataframe(
            values,
            thickcol='th',
            component_map=litholog.defaults.DEFAULT_COMPONENT_MAP,
            metacols=METACOLS,
            datacols=DATACOLS,
        )
    )

# Show name + eod + number of beds of each
print(len(seqs),'logs imported as BedSequences')

[(s.metadata['name'], len(s),'Beds') for s in seqs]
```

```
13 logs imported as BedSequences
```

[4]:
```
[('Karoo krf1', 105, 'Beds'),
 ('Karoo krf2', 20, 'Beds'),
 ('Karoo krf3', 42, 'Beds'),
 ('Karoo krf4', 15, 'Beds'),
 ('Karoo krf5', 51, 'Beds'),
 ('Magnolia', 181, 'Beds'),
 ('Pukearuhue', 211, 'Beds'),
 ('Sierra del Toro dc1', 166, 'Beds'),
 ('Sierra del Toro dc2', 44, 'Beds'),
 ('Sierra del Toro flame', 63, 'Beds'),
 ('Sierra del Toro h2o', 66, 'Beds'),
 ('Sierra del Toro ssm', 54, 'Beds'),
 ('Sierra del Toro wc', 38, 'Beds')]
```

[5]:
```
# nice way to parse data to get only BedSequences with more than 30 sand beds
thirty = list(filter(lambda s: len(s.get_field('th', 'sand')) >= 30, seqs))

# display the names
[(s.metadata['name']) for s in thirty]
```

[5]:
```
['Karoo krf1',
 'Karoo krf5',
 'Magnolia',
 'Pukearuhue',
 'Sierra del Toro dc1',
 'Sierra del Toro h2o']
```

2.2.3 Basic data retrieval from a BedSequence and its Beds

2.2.3.1 BedSequence

[6]:
```
# Choose two logs to use as examples
magnolia = seqs[5] # a core description from the Gulf of Mexico

testseq = seqs[-2] # an outcrop log from Chile

# see what they look like
[magnolia, testseq]
```

[6]:
```
[Striplog(181 Intervals, start=0.0, stop=59.999347598),
 Striplog(54 Intervals, start=0.0, stop=65.99495201)]
```

[ ]:
```
# Order is important for logs, and it is either elevation or depth
print('Magnolia is a core so it has order:',magnolia.order,'\n'
      'all other demo BedSequences are elevation ordered')

# here are all the orders:
[[s.order, s.metadata['name']] for s in seqs]
```

[ ]:
```
# access the base and top of a BedSequence
testseq.start, testseq.stop
```

[ ]:
```
# Access the metadata
print(testseq.metadata)

testseq.metadata['ng']
```

[ ]:
```
# Access the `data` fields, which are numpy arrays
print(type(testseq.get_field('th')))
print(testseq.get_field('th'))
```

[ ]:
```
print('total thickness is',testseq.get_field('th').sum(),'meters')
print('maximum Bed thickness is',testseq.max_field('th'),'meters')
print('minimum Bed thickness is',testseq.min_field('th'),'meters')
print('maximum grain size is',testseq.max_field('grain_size_mm'),'mm')
```

2.2.3.2 Bed

Now let's take a look at one `Bed`. All `Beds` must have a `top`, `base`, and `data`, which can be array or dict-like. You will note that there is a `primary` field as well, which is defined using the `Component` from `striplog` or a component map, as we did above

```
[ ]: testseq[-1] # In elevation order, the lowermost bed is the last one [-1] instead of the␣
     ↪first one [0]
```

```
[ ]: # if a log is elevation-ordered, the last bed should have a base at zero
     testseq[-1].base
```

```
[ ]: # what about a depth-oprdered core?
     magnolia[0].top
     # if this core was really in depth units (e.g., measured depth),
     # the top would be whatever depth the core top was. In this case,
     # the depths were manually transformed to meters prior to digitization
```

```
[ ]: print(testseq[-1].top.upper) # see the striplog Interval class for info on upper, middle,
     ↪ and lower
     print(testseq[-1].primary.lithology)
     print(testseq[-1].data['mean_gs_mm'])
     print('Mean grain size:',litholog.wentworth.gs2name(testseq[-1].data['mean_gs_mm']))
     print('Grain size at top of bed:',litholog.wentworth.gs2name(testseq[-1].data['gs_tops_mm
     ↪']))
```

```
[ ]: # Let's look at the uppermost bed now
     print(testseq[0].lithology)
     print(testseq[0].summary())
```

```
[ ]: # get one bed top using an index
     print(testseq[0].top.middle)

     # get the first five bed tops using list comprehension (see striplog docs for attributes␣
     ↪of top and base (e.g., base, middle, upper))
     print('first five',[bed.top.middle for bed in testseq[0:5]])

     # or you can build a simple loop
     for bed in testseq[0:5]:
         print(bed.top.middle)
```

```
[ ]: # lets look at a covered interval - note  the NaNs for grain size
     testseq[-11]
```

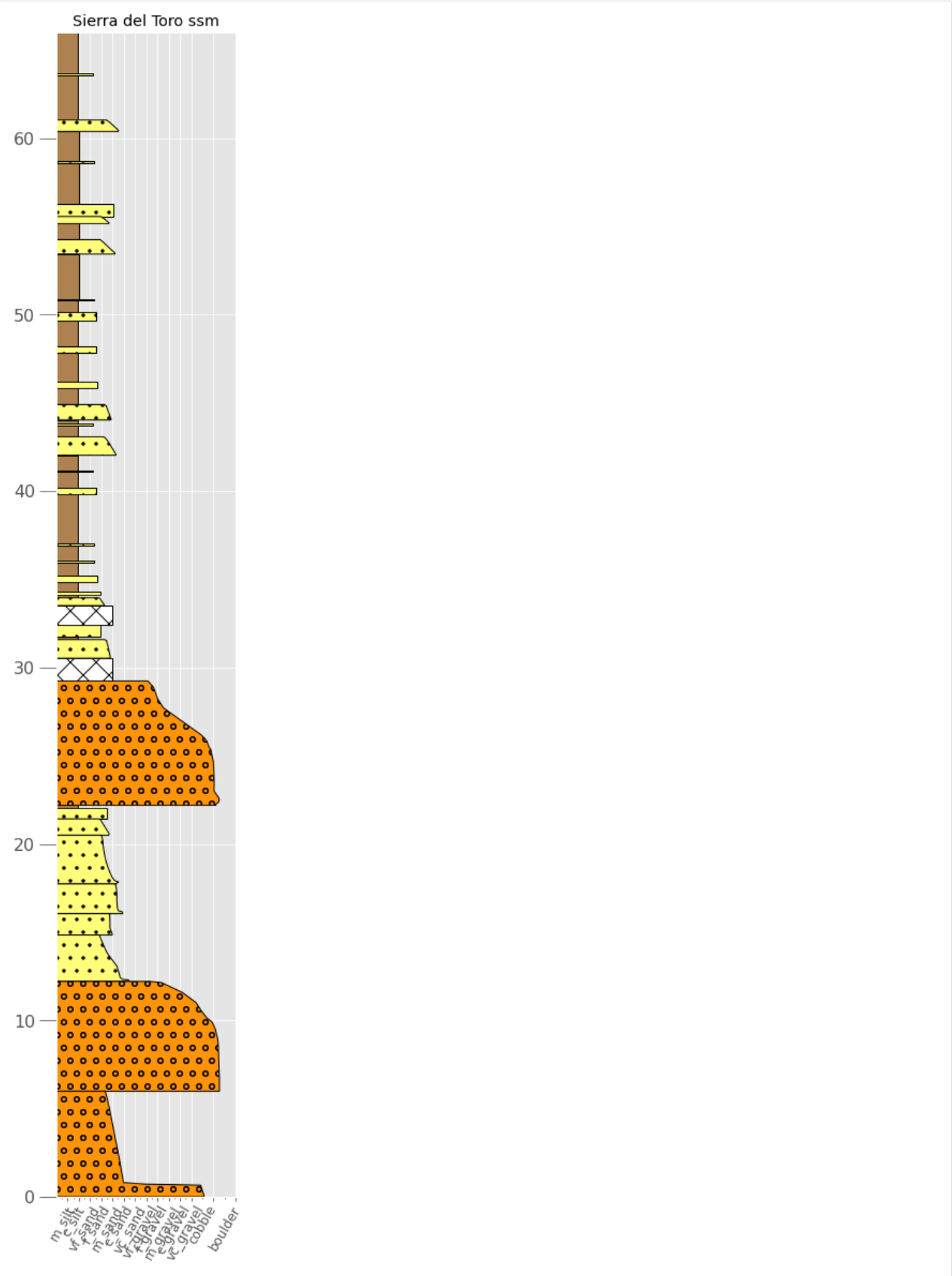
2.2.4 Plotting

2.2.4.1 Basic plotting

[ ]:
```
# the most simple plot, but doesnt include any bed-level grain size data (e.g., fining-
↪upwards)
testseq.plot()
```

2.2.4.2 Including intra-Bed grain-size data in a plot

[7]:
```
# here is a little nicer way to plot it that includes the grain size data for each bed

fig, ax = plt.subplots(figsize=[3,20])

testseq.plot(ax=ax,
             legend=litholog.defaults.litholegend,
             width_field='grain_size_psi',
             depth_field='depth_m',
             wentworth='fine'
            )

ax.set_title(testseq.metadata['name']);

# can save it by uncommenting this line
#plt.savefig('testseq.svg')
```


Sierra del Toro ssm
60
50
40
30
20
10
0
m_silt
c_silt
vf_sand
f_sand
m_sand
c_sand
vc_sand
vf_gravel
f_gravel
m_gravel
c_gravel
vc_gravel
cobble
boulder

Extracting fining-upwards profiles from a `Bed`

[ ]:
```
gs=testseq[-1].data['grain_size_psi'] # get just one field from a Bed
de=testseq[-1].data['depth_m']

plt.plot(gs,de,'k')
if testseq.order=='depth':
    plt.gca().invert_yaxis()
    print('depth ordered')

plt.title('Fining upward trend from one Bed');
```

[ ]:
```
# and we can do it for all beds that are classified as `sand`:
fig, ax = plt.subplots()
for bed in testseq:
    if bed.lithology=='sand':
        gs=bed.data['grain_size_psi']

        de=bed.data['depth_m']
        de=np.max(de)-de # normalize to zero so they will all plot together
        if testseq.order=='elevation':
            de=np.flip(de)
        ax.plot(gs,de,'k')
```

2.2.4.3 Flipping the order of a BedSequence

[ ]:
```
testseq.order
```

[ ]:
```
# flipping to depth order (Note - this doesnt change the BedSequence at all, just␣
↪replots it)
testseq.flip_convention(depth_key='depth_m').plot(
    legend=litholog.defaults.litholegend,
    fig_width=3,
    aspect=5,
    width_field='grain_size_psi',
    depth_field='depth_m',
    wentworth='coarse');

# note that this doesn't change the order, it just creates the plot
print(testseq.order)
```

[ ]:
```
# Now let's flip Magnolia from depth to elevation
magnolia.flip_convention(depth_key='depth_m').plot(
    legend=litholog.defaults.litholegend,
    fig_width=3,
    aspect=5,
    width_field='grain_size_psi',
    depth_field='depth_m',
    wentworth='coarse')
```

2.2.4.4 Plot mulitple logs at the same scale (i.e., a correlation panel)

By using matplotlib's `subplots`, we can plot several `BedSequences` at the same scale, which is commonly done for many purposes, including creating correlation panels. Note that this functionality could be vastly improved upon, but it'll do for now.

[ ]:
```
fig, ax = plt.subplots(ncols=3, sharey=True, figsize=(10,20))

seqs[-4].plot(legend=litholog.defaults.litholegend,
              width_field='grain_size_psi',
              depth_field='depth_m',
              ax=ax[0])

seqs[-3].plot(legend=litholog.defaults.litholegend,
              width_field='grain_size_psi',
              depth_field='depth_m',
              ax=ax[1])

seqs[-2].plot(legend=litholog.defaults.litholegend,
              width_field='grain_size_psi',
              depth_field='depth_m',
              ax=ax[2])

ax[0].set_ylim([0,110]); # could clean this up into a function to recommend ylim, but␣
↪for now it's fine
```

2.2.5 Save out images

[ ]:
```
for i,s in enumerate(seqs):
    print(i, end=',')
    fig, ax = plt.subplots(figsize=[5,25])
    s.plot(ax=ax,
              legend=litholog.defaults.litholegend,
              width_field='grain_size_psi',
              depth_field='depth_m',
              wentworth='fine'
             )
    title_str = s.metadata['name']
    ax.set_title(title_str);
    plt.tight_layout()
    fig.savefig(title_str+'.png')
    plt.close(fig)
```

2.2.6 Statistics for BedSequences

[ ]:
```
print(testseq.metadata)

# Properties computed on the fly
print(testseq.net_to_gross,';',testseq.amalgamation_ratio)

print('N:G from Matlab:',round(testseq.metadata['ng'],3),'\n',
      'N:G from litholog:',round(testseq.net_to_gross,3))
```

[ ]:
```
# Hurst statistics
print(testseq.hurst_K('th', 'sand'))

# Returns (D, p, hurst_K)
testseq.hurst_D('th', 'sand', nsamples=10000)
```

2.2.7 Pseudo gamma-ray log

We include functionality to create a simple pseudo GR curve of a `BedSequence`.

First, let's define some functions to help us make this plot. Note, these functions only really work when you are plotting the same log (i.e., the log itself and the pseudo GR) - if you are plotting two logs together, see above example…

[ ]:
```
def suggest_figsize(sequence, aspect=10):
    """
    Defining default a total thickness -> figure size mapping.
    """
    suggest_h = max(10, min(sequence.cum, 50))
    suggest_w = suggest_h / aspect
    return (suggest_w, suggest_h)

def strip_fig_extra_columns(ax_num, sequence, ncols, exxon_style=True, figsize=None,␣
↪aspect=10):
    """
    Creates a fig with `ncol` axes and plots `sequence` on one of them.
    If `exxon_style`, plots `sequence` on first axis, otherwise last axis.
    Returns
    -------
    fig, ax
    """
    w, h = suggest_figsize(sequence, aspect=aspect)
    print(w, h)

    fig, ax = plt.subplots(ncols=ncols, sharey=True, figsize=(w*ncols, h))
    #fig.subplots_adjust(wspace=0.)

    sequence.plot(legend=litholog.defaults.litholegend,
                  width_field='grain_size_psi',
                  depth_field='depth_m',
                  ax=ax[ax_num])

    return fig, ax
```

[ ]:
```
# Now let's plot it
# The values you see are the defaults:
ds, pgr = testseq.pseudo_gamma_simple(
    gs_field='grain_size_mm',
    depth_field='depth_m',
    resolution=0.2,
    gs_cutoff=0.0625,
    gamma_range=(30, 180),
    sigma=0.1,
    noise=10.
)

fig, ax = strip_fig_extra_columns(0, testseq, 2, aspect=9)

cutoff = 100

ax[1].plot(pgr, ds, 'k')
ax[1].fill_betweenx(ds, pgr, np.repeat(cutoff, ds.size), where=(pgr<cutoff), color=
↪'yellow')

ax[1].set_xlim([0,200])
```

2.2.8 Other methods not demonstrated here

There are a few methods that we didn't demonstrate here, including a `resample_data` , which is handy for depth-based resampling of grain size data (e.g., to export into Petrel or other software).

We would love your feedback on `litholog` and pull-requests to make it better!

[ ]:

2.3 litholog

2.3.1 litholog package

2.3.1.1 Subpackages

litholog.sequence package

Submodules

litholog.sequence.io module

IO classes & functions

class litholog.sequence.io.**SequenceIOMixin**

Bases: `abc.ABC`

Defines the IO interface for *BedSequence*.

classmethod from_dataframe(*df*, *topcol='tops'*, *basecol=None*, *thickcol=None*, *component_map=None*, *datacols=[]*, *metacols=[]*, *metasafe=True*, *tol=0.001*)

Create an instance from a pd.DataFrame or subclass (e.g., a GroupBy object). Must provide *topcol* and one of *basecol* or *thickcol*.

Parameters

- **df** (*pd.DataFrame or subclass*) – Table from which to create *list_of_Beds*.
- **topcol** (*str*) – Name of top depth/elevation column. Must be present. Default='top'.
- **basecol, thickcol** (*str*) – Either provide a base depth/elevation column, or a thickness column. Must provide at least one.
- **component_map** (*tuple(str, func), optional*) – Function that maps values of a column to a primary *striplog.Component* for individual Beds. TODO: if *func* is a str with 'wentworth', maybe just map using grainsize bins?
- **datacols** (*list(str), optional*) – Columns to use as *Bed* data. Should reference numeric columns only.
- **metacols** (*list(str), optional*) – Columns to read into *metadata* dict attribute.
- **metasafe** (*bool, optional*) – If True, enforces that df[metacols] have a single unique value per column. If False, just attaches any + all unique values.

classmethod from_numpy(*arr*, *other=None*, *keys=None*, *split_key=None*, *component_map=None*)

TODO: Implement a method to convert numpy (e.g., from GAN) to *BedSequence* instance.

Use keys from *other*, or provide list of *keys*. Provide a *component_map* to group samples into `Bed`s?

litholog.sequence.io.**check_order**(*df*, *topcol*, *basecol*, *raise_error=True*)

Check that all rows are either depth ordered or elevation_ordered. Returns 'elevation' or 'depth'.

litholog.sequence.io.**check_samples**(*df*, *depthcol*, *valuecol*)

Check that *depth_col* and *sample_col* have equal number of entries per bed,

Returns **good** – True if sizes match in all rows, False otherwise.

Return type bool

litholog.sequence.io.**check_thicknesses**(*df*, *topcol*, *thickcol*, *order*, *basecol='bases'*, *tol=0.001*)

Check that gap between tops and adjacent bases implied by 'th' are consistent and small.

Returns **(df, good)** – *df* has new *basecol* added with implied base positions *good* is *True* if the average gap < *tol*, else *False*

Return type (DataFrame, bool)

litholog.sequence.io.**preprocess_dataframe**(*df*, *topcol*, *basecol=None*, *thickcol=None*, *tol=0.001*)

Check for position order + consistency in *df*, return preprocessed DataFrame.

This doesn't check for all possible inconsistencies, just the most obvious ones.

litholog.sequence.sequence module

Notes:

- Just need top depths/elevations.

class litholog.sequence.sequence.**BedSequence**(*list_of_Beds*, *metadata={}*)

Bases: *litholog.sequence.io.SequenceIOMixin*, *litholog.sequence.viz.SequenceVizMixin*, *litholog.sequence.stats.SequenceStatsMixin*, striplog.striplog.Striplog

Ordered collection of Bed instances.

flip_convention(*depth_key=None*)

Changes the depth convention (elevation <-> depth), setting to base or top to 0.0, respectively.

If *depth_key* is given, that data column in each bed should be shifted the same amount.

get_field(*field*, *lithology=None*, *default_value=0.0*)

Get 'vertical' array of field values.

If *lithology* provided, will only use the beds matching that lithology (in primary component).

get_values(*exclude_keys=[]*)

Getter for values that allows dropping exclude_keys (e.g., sample depths) from array

max_field(*field*)

Override method from striplog.Striplog to account for iterable Bed data.

min_field(*field*)

Override method from striplog.Striplog to account for iterable Bed data.

property nfeatures

The number of columns in values.

property nsamples

The number of sample rows in values. NOTE: len(striplog.Striplog) will already give number of beds.

reduce_field(*field*, *fn*)

Apply fn to the output of get_field(field)

reduce_fields(*field_fn_dict*)

Return array, result of applying *fn* values to *field* keys.

The funcs can return scalars or arrays, but all of the return values should be numpy-concatable. The concatenation

resample_data(*depth_key*, *step*, *kind='linear'*)

Resample the data at approximate depth intervals of size *step*. *depth_key* can be a *str* (for dict-like bed data) or column index (for array bed data).

I think we probably want to maintain top/base samples, and sample to the nearest *step* b/t. Maybe this could be the default of multiple options? Implement it as the default first though.

NOTE: We could return a new instance rather than modify inplace, since it's hard to undo.

shift(*delta=None*, *start=None*, *depth_key=None*)

Shift all the intervals by *delta* (negative numbers are 'up'), or by setting a new start depth. Returns a new copy of the BedSequence.

property values

Get the instance as a 2D array w/ shape (nsamples, nfeatures).

litholog.sequence.stats module

Sequence stats + related.

class litholog.sequence.stats.**SequenceStatsMixin**
Bases: abc.ABC

Defines the plot/viz interface for *BedSequence*.

property amalgamation_ratio

1. dont count mud on mud contacts
2. find sand on sand contacts
3. divide sand-on-sand contacts by total number of contacts

NOTE: 'gravel' counts as a 'sand'

hurst_D(*field*, *lithology*, *take_log=True*, *safe=True*, *nsamples=1000*, *return_K=True*)
Returns (D, p, K) if `return_K`, else (D, p) where:

D : Bootstrapped Hurst value from `nsamples` resamples p : p-value of `D` K : Hurst K value with original values

hurst_K(*field*, *lithology*, *safe=True*)
Hurst K value for data from a sequence `field`.

If `safe`, will only accept fields with at least 20 values.

property interfaces
Get all pairs of adjacent Beds, ignoring any pairs with either Bed 'missing'

property net_to_gross
Returns (total thickness of 'sand' & 'gravel' Beds) / (total thickness of all Beds)

pseudo_gamma_simple(*gs_field='grain_size_mm'*, *depth_field='depth_m'*, *resolution=0.2*, *gs_cutoff=0.0625*, *gamma_range=(30, 180)*, *sigma=0.1*, *noise=10.0*)
Compute a 'pseudo' gamma ray log by thresholding *gs_field* + Gaussian convolution.

Parameters

- **gs_field** (*str*) – Which field to use for grainsize
- **depth_field** (*str*) – Which field to use for depth
- **resolution** (*float*) – Scale at which to resample (in *depth_field* units)
- **gs_cutoff** (*float*) – Cutoff for *gs_field* thresholding. Values above/below get mapped to *gamma_range*.
- **gamma_range** (*tuple or list*) – (low, high) sample values for *gs_field* values (above, below) *gs_cuttoff*.
- **sigma** (*float*) – Width of Gaussian, in depth units.
- **noise** (*float or None*) – Magnitude of uniform noise to add, or None to add no noise.

litholog.sequence.stats.**filter_nan_gaussian**(*arr*, *sigma*, *noise=None*)
Gaussian convolution. (Allows intensity to leak into the NaN area.)

If *noise* magnitude given, adds uniform noise.

Implementation from stackoverflow answer: https://stackoverflow.com/a/36307291/7128154

litholog.sequence.viz module

Visualization funcs and the *SequenceVizMixin* interface for *BedSequence*.

class litholog.sequence.viz.**SequenceVizMixin**
Bases: abc.ABC

Defines the plot/viz interface for BedSequence.

plot(*legend=None*, *fig_width=1.5*, *aspect=10*, *width_field=None*, *depth_field=None*, *wentworth='fine'*, *exxon_style=False*, *yticks_right=False*, *set_ylim=True*, *xlim=None*, *ax=None*, ***kwargs*)
Plot as a Striplog of ``Bed``s.

Parameters

- **legend** (*striplog.Legend, optional*) – If beds have primary component with 'lithology' field, will use defaults.litholegend, otherwise random.
- **fig_width** (*int, optional*) – Width of figure, if creating one.
- **aspect** (*int, optional*) – Aspect ratio of figure, if creating one.
- **width_field** (*str or int*) – The Bed.data` field or Bed.values column used to define polyon widths.
- **depth_field** – The Bed.data field or Bed.values column defining depths of width_field samples
- **wentworth** (*one of {'fine', 'coarse'}*) – Which Wentworth scale to use for xlabels/ticks.
- **exxon_style** (*bool, optional*) – Set to true to invert the x-axis (so GS increases to the left).
- **yticks_right** (*bool, optional*) – If True, will move yticks/labels to right side. Defualt=False.
- **set_ylim** (*bool, optional*) – Whether to set the y-limits of the ax to [self.start, self.stop]. Default=True.
- ****kwargs** (*optional*) – ylabelsize, yticksize, xlabelsize, xlabelrotation

litholog.sequence.viz.**make_pair_figure**()
Generate a figure with two column axes and no space horizontal between them.

litholog.sequence.viz.**set_wentworth_ticks**(*ax*, *min_psi*, *max_psi*, *wentworth='fine'*, ***kwargs*)
Set the *xticks* of *ax* for Wentworth grainsizes.

Parameters

- **ax** (*matplotlib.Axes*) – Axes to modify.
- **min_psi, max_psi** (*float*) – Define the *xlim* for the axis.
- **wentworth** (*one of {'fine', 'medium', 'coarse'}*) – Which scale to use. Default='fine'.
- ****kwargs**

Module contents

2.3.1.2 Submodules

2.3.1.3 litholog.bed module

class litholog.bed.**Bed**(*top*, *base*, *data*, *keys=None*, ***kwargs*)
Bases: striplog.interval.Interval

Represents an individual bed or layer.

Essentially a striplog.Interval with some additional restrictions and logic.

Beds are required to have a top, base, and data (which can be an array or dict-like).

as_patch(*legend*, *width_field=None*, *depth_field=None*, *min_width=0.0*, *max_width=1.5*, ***kwargs*)
Representation of the Bed as a matplotlib.patches object [Polygon or Rectangle].

Parameters

- **legend** (striplog.Legend) – Legend to get a matching striplog.Decor from.
- **width_field** (*str or int, optional*) – data key or values column index to use as width field.
- **depth_field** (*str or int, optional*) – Data key or values column index to use as positions of width_field samples. If not provided and width_field values are iterable, created from np.linspace(top, base). Ignored if width_field is a scalar. Sizes must match if both fields return iterables.

Returns patch

Return type instance from matplotlib.patches

compatible_with(*other*)
Check that self.data and other.data have compatible values shapes and matching data key order.

NOTE: Should both have to be constructed from similar dtypes, or just have concatable *values*?

get_values(*exclude_keys=[]*)
Getter for values that allows dropping exclude_keys (e.g., sample depths) from array

property lithology
Just shorthand.

max_field(*key*)
Return the maximum value of data[key], or None if it doesn't exist.

min_field(*key*)
Return the minimum value of data[key], or None if it doesn't exist.

property nfeatures
The number of columns in values.

property nsamples
The number of sample rows in values.

resample_data(*depth_key*, *step*, *kind='linear'*)
Resample data to approximately step, but preserving at least top/base samples.

Parameters

- **depth_key** (*str, int, or None*) – Dict key or column index pointing to sample depths
- **step** (*float*) – Depth step at which to (approximately) resample data values

- **kind** (*one of {'linear','slinear','quadratic','cubic',... }, optional*) – Kind of interpolation to use, default='linear'. See `scipy.intepolate.interp1d` docs.

spans(*d*, *eps=0.001*)
Determines if position `d` is within this `Bed`. Overridden from `striplog.Interval` to accomodate small tolerance `eps`

Parameters **d** (*float*) – Position (depth or elevation) to evaluate.

Returns **in_bed** – True if `d` is within the `Bed`, False otherwise.

Return type bool

property values

2.3.1.4 litholog.defaults module

Define default *striplog.Decor`s when using grainsize as the width field. Also specify default csv/DataFrame fields to map to `Bed* attributes/data.

litholog.defaults.**gs2litho**(*gs*, *units='psi'*)
Map grainsize value *gs* to *striplog.Component*.

If *gs* is None or np.nan, maps to 'missing' Component.

If *units* is 'mm' or 'phi', will convert to 'psi' first.

Psi is log2(gs_mm), so medium sand is -1 to -2. See more at https://en.wikipedia.org/wiki/Grain_size#/media/File:Wentworth_scale.png

2.3.1.5 litholog.utils module

Utility functions.

litholog.utils.**safelen**(*x*)
Return the length of an array or iterable, or 1 for literals.

litholog.utils.**saferep**(*x*, *n*)
Repeat *x* to array of length *n* if it's a literal, or check that *len(x) == n* if it's iterable.

litholog.utils.**string2array_matlab**(*s*)
Parse matlab-style array string (e.g., "1.0, 2.0, 3.0") to float array.

litholog.utils.**string2array_pandas**(*s*)
Parse pandas-style array string (e.g, "[1.0 2.0 3.0]") to float array.

litholog.utils.**strings2array**(*elems*)
Convert iterable of numeric strings to (float) array.

2.3.1.6 litholog.wentworth module

Utility functions for Wentworth/Krumbein logarithmic grainsize scale.

See this link for a nice chart: https://en.wikipedia.org/wiki/Grain_size#/media/File:Wentworth_scale.png

litholog.wentworth.**gs2name**(*gs*)

litholog.wentworth.**gs2phi**(*gs*)

litholog.wentworth.**gs2psi**(*gs*)

litholog.wentworth.**phi2gs**(*phi*)

litholog.wentworth.**phi2name**(*phi*)

litholog.wentworth.**phi2psi**(*phi*)

litholog.wentworth.**psi2gs**(*psi*)

litholog.wentworth.**psi2name**(*psi*, *scale=[('colloid', - 10), ('clay', - 8), ('vf_silt', - 7), ('f_silt', - 6), ('m_silt', - 5), ('c_silt', - 4), ('vf_sand', - 3), ('f_sand', - 2), ('m_sand', - 1), ('c_sand', 0), ('vc_sand', 1), ('vf_gravel', 2), ('f_gravel', 3), ('m_gravel', 4), ('c_gravel', 5), ('vc_gravel', 6), ('cobble', 8), ('boulder', None)]*)

Map single *psi* value to Wentworth bin name.

litholog.wentworth.**psi2phi**(*psi*)

2.3.1.7 Module contents

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

l

INDEX

N

P

R

S

V