Example mesh construction

July 13, 2024

1 Explanation of the FiniteElementVary2DMeshConstructor

This file presents a step-by-step application of the

`sfsimodels.num.mesh.FiniteElementVary2DMeshConstructor` to an `sfsimodels.TwoDSystem`.

The key steps are:

1. Define geometric inputs: surface geometry, model depth and width, the layer depths at each soil profile along a transect and the slope of each soil layer between soil profiles.
2. Define meshing targets (target element height, minimum and maximum element height ratio, ratio of element width to height along the transect, and allowable mesh slope).
3. Estimate the number of elements between each layer at each significant horizontal position (i.e. each defined soil profile or change in surface geometry).
4. Adjust the number of elements to try to have the same thickness of elements in a layer at each significant horizontal position while maintaining the meshing targets
5. Define the position of the x-axis (horizontal) grid points
6. Interpolate the position of the y-coordinates at each x-axis grid point position between the significant horizontal positions
7. Adjust the surface elements to best follow the surface geometry by adjusting both the x and y position of the near surface grid points

1.1 Import python libraries for displaying images

[1]:
```
%gui qt5
from PyQt5.Qt import QApplication


# package for meshing
import sfsimodels as sm
# package needed for loading the PM4Sand and PM4Silt models
import liquepy as lq
# package used for manipulation of transect data before meshing
import numpy as np
```

```
# Packages used for plotting the output
import pyqtgraph as pg
import o3plot
from matplotlib.pyplot import imshow
import matplotlib.pyplot as plt
import matplotlib as mpl
import all_paths as ap
mpl.rcParams['figure.dpi'] = 300
pg.setConfigOption('background', 'w')


def show_plot_as_img(win):
    exp = pg.exporters.ImageExporter(win.plotItem)
    exp.export('', toBytes=True)
    bf, ax = plt.subplots(figsize=(8, 7))
    ax.imshow(exp.bg)
    del win
    ax.set_xticks([])
    ax.set_yticks([])
    plt.show()
```

1.2 Load a transect

In this example the Avon Loop 2 transect (avnlp2) is used.

The transect is saved as a 2D system. The system consists of soil profiles with angled layers, ground surface coordinates, a ground water table height, as well as a system depth and width.

[2]:
```
tran_code = 'avnlp2'
folder = 'selected-transects-ecps-w-pm4silt'
fname = f'ecp_{tran_code}_pm4silt.json'

mods = sm.load_json(f'{ap.MODULE_DATA_PATH}{folder}/{fname}',
                    custom={'soil-pm4sand': lq.num.flac.PM4Sand,
                            'soil-pm4silt': lq.num.flac.PM4Silt})
tds = mods['system'][1]
assert isinstance(tds, sm.TwoDSystem)
tds.inputs += ['h_face', 'x_face', 'face_slope']

# Trim the model to a target length (from free-face to model edge)
# Set the free-face width to be 30m
rb_width = 100.0  # m (Target right bank width) (use only 100m for example)
sm.models.systems.trim_system_to_width(tds, rb_width, ff_width=30)
tds.x_face = tds.x_surf[0]
tds.x_top = tds.x_surf[1]
tds.h_face = tds.y_surf[1]
```

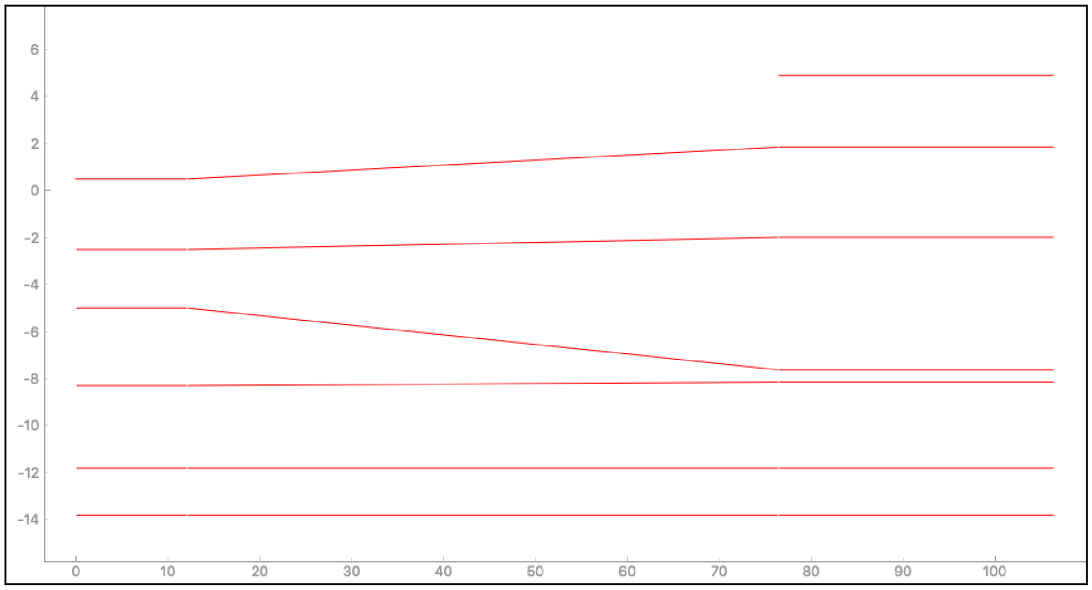
```python
channel_width = tds.channel_width
# limit the channel width to 80m to avoid very large models
channel_width = min(channel_width, 80)
tds.channel_width = channel_width
extra_width = channel_width / 2
x_surf_rhs = tds.x_surf + extra_width
y_surf_rhs = tds.y_surf
tds.x_top += extra_width
for xx in range(1, len(tds.x_sps)):
    tds.x_sps[xx] += extra_width
tds.x_surf = np.array([0] + list(x_surf_rhs))
tds.y_surf = np.array([0] + list(y_surf_rhs))
tds.width += extra_width
x_face = tds.x_face + extra_width
tds.x_face = x_face

for item in tds.inputs:
    val = getattr(tds, item)
    if val is None or hasattr(val, '__len__') and len(val) == 0:
        continue
    print(f'{item}: {getattr(tds, item)}')
```

```
base_type: system
type: two_d_system
id: 1
name: avnlp2
width: 106.5
height: 14.8
sps: [SoilProfile id: 1, name: LHS, SoilProfile id: 2, name: 10012, SoilProfile
id: None, name: free-field]
x_sps: [0.0, 12.196, 76.5]
x_surf: [ 0.   6.5  9.7 26.7 42.7 54.7 76.5]
y_surf: [0.  0.  3.2 3.2 4.9 4.9 4.9]
gwl: 1.5
h_face: 3.2
x_face: 6.5
face_slope: 1.0
```

1.3 Show transect geometry

[3]:
```python
y_sf = 3  # scale factor for y-axis
win = o3plot.create_scaled_window_for_tds(tds, y_sf=y_sf)
o3plot.plot_two_d_system(tds, win)
show_plot_as_img(win)
```



1.4 Initialise the constructor

The constructor is given two primary inputs: - `tds`: an `sfsimodels.TwoDSystem`, A Two dimensional system defined using the sfsimodels `TwoDSystem` object. - `dy_target`: `float` which defines the target height of each element

There are also five optional mesh control inputs.

- `x_scale_pos`: `array_like`, defining positions from the left edge, where the element width should change size, default to [0.0]
- `x_scale_vals`: `array_like`, defining the scale between the target element height and target element width at each zone between the `x_scale_pos` points, defaults to [1.0]
- `min_scale` The target minimum mesh ratio of the target (`dy_target`), default = 0.5
- `max_scale` The target maximum mesh ratio of the target (`dy_target`), default = 2.0
- `allowable_slope` The target change in height over width across a set of blocks before an block should be added/remove, default = 0.25

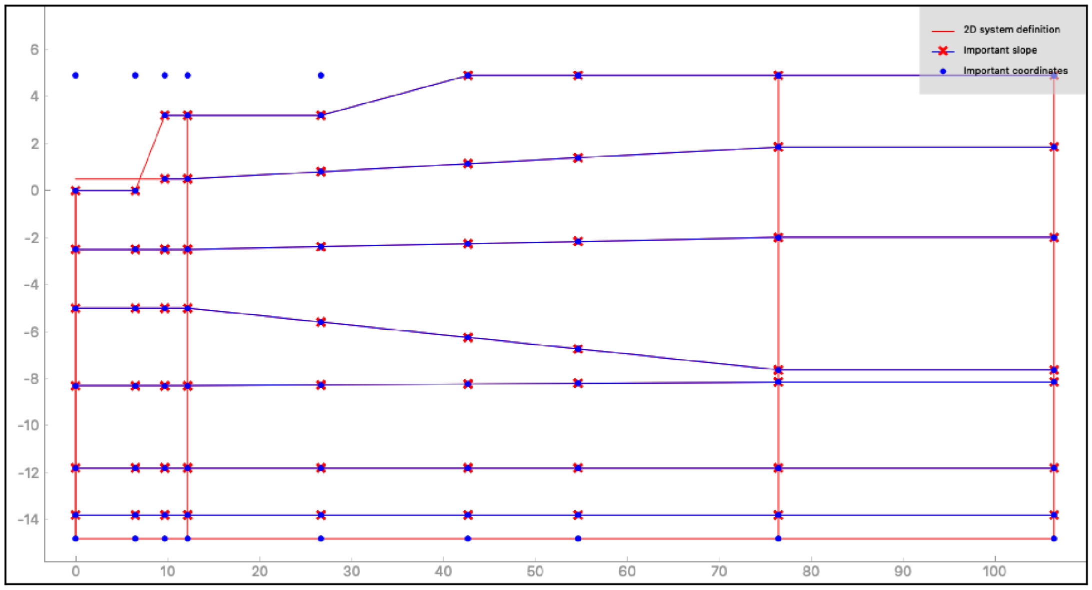
The `auto_run` parameter is also set to `False` to allow each step to be run individually.

[4]:
```
# Define target width-to-height factors for mesh along x-axis
x_scale_pos = np.array([0, tds.x_top + 50, tds.x_top + 100, tds.width])
x_scale_vals = np.array([1.8, 2.5, 3.0, 3.0])
fc = sm.num.mesh.FiniteElementVary2DMeshConstructor(tds, dy_target=0.5,
                                                     x_scale_pos=x_scale_pos,
                                                     x_scale_vals=x_scale_vals,
                                                     fd_eles=1, smooth_surf=True,
                                                     force_x2d=True,␣
     ↪auto_run=False,
                                                     smooth_ratio=1.5)
```

1.5 Step 1: Find the important coordinates and slopes in the 2D system

This includes layer boundaries, foundation corners, and surface slopes that should be maintained in the FE mesh. When coordinates are very close (less than `min_scale*dy_target`) then the top left coordinate is kept.

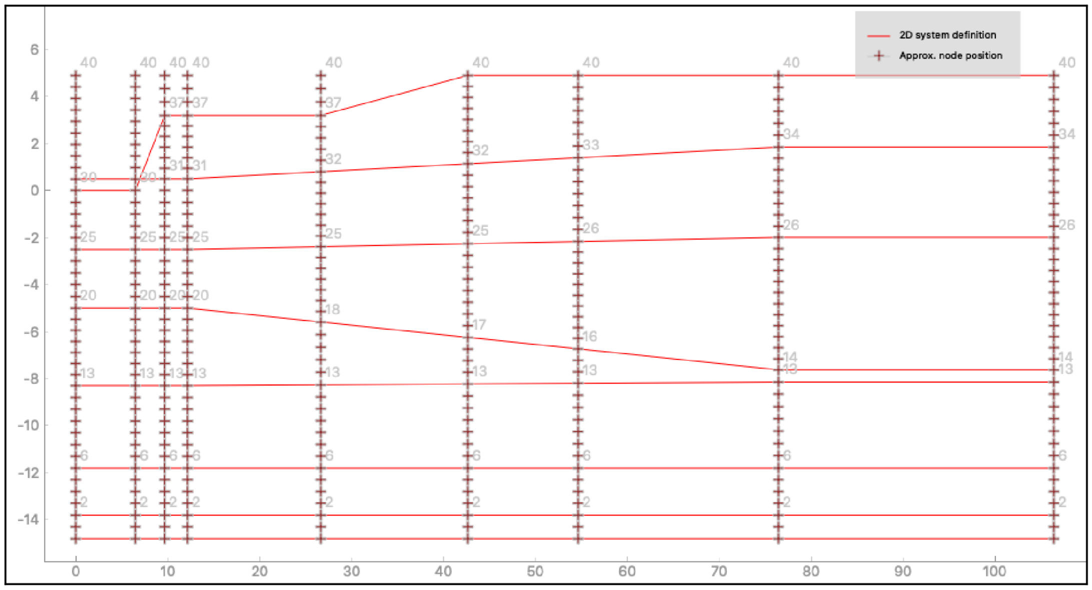
[5]:
```
fc.get_special_coords_and_slopes()  # Step 1
win = o3plot.create_scaled_window_for_tds(tds, y_sf=y_sf)
leg = win.addLegend(offset=(0, .5), brush=(200, 200, 200, 150),␣
↪labelTextColor='k')
o3plot.plot_two_d_system(tds, win, c2='b', cs='b')
leg.addItem(pg.PlotDataItem([0], [0], pen='b'), name='2D system definition')
for i in range(len(fc.sds)):
    win.plot(fc.sds[i][0], fc.sds[i][1], pen='r', symbol='x', symbolPen='b',
             symbolBrush='b', symbolSize=10, name='Important slope')
xcs = fc.xcs_sorted
for i in range(len(xcs)):
    xc = xcs[i]
    xn = xc * np.ones_like(list(fc.yd[xc]))
    win.plot(xn, list(fc.yd[xc]), symbol='o', symbolPen='r', symbolBrush='r',
             symbolSize=5, pen=None, name='Important coordinates')
# leg.setParentItem(win.plotItem)
o3plot.revamp_legend(leg)
show_plot_as_img(win)
```



1.6 Step 2: Set the initial number of elements along important x-lines

Estimate the number of elements high at each important coordinate based on the `dy_target`.

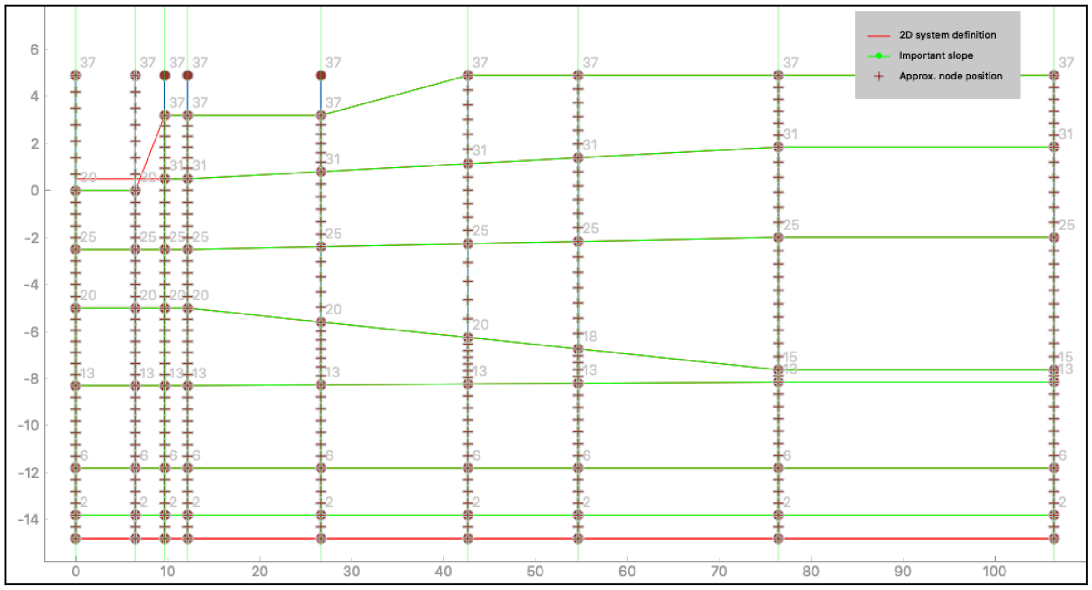
[6]:
```python
fc.set_init_y_blocks()
win = o3plot.create_scaled_window_for_tds(tds, title='set_init_y_blocks',␣
 ↪y_sf=y_sf)
leg = win.addLegend(offset=(750, 5), brush=(200, 200, 200, 150),␣
 ↪labelTextColor='k')
o3plot.plot_two_d_system(tds, win, c2='b', cs='b')
leg.addItem(pg.PlotDataItem([0], [0], pen='b'), name='2D system definition')
xcs = fc.xcs_sorted
for i in range(len(xcs)):
    xc = xcs[i]
    h_blocks = np.diff(fc.yd[xc])
    dhs = h_blocks / fc.y_blocks[xc]
    y_node_steps = [0]
    for hh in range(len(fc.y_blocks[xc])):
        y_node_steps += [dhs[hh] for u in range(fc.y_blocks[xc][hh])]
    y_node_coords = np.cumsum(y_node_steps) - tds.height
    xn = xc * np.ones_like(y_node_coords)
    nbs = np.cumsum(fc.y_blocks[xc])
    for cc in range(len(fc.y_blocks[xc])):
        text = pg.TextItem(f'{nbs[cc]}', anchor=(0, 1))
        win.addItem(text)
        text.setPos(xc, fc.yd[xc][cc + 1])
    win.plot(xn, y_node_coords, symbol='+', name='Approx. node position')
    # win.addItem(pg.InfiniteLine(xcs[i], angle=90, pen=(0, 255, 0, 100)))
o3plot.revamp_legend(leg)
show_plot_as_img(win)
```



1.7 Step 3: Adjust the number of blocks high to try to have the same number of blocks along each slope

Moving from left to right, then bottom to top, try to make the number of blocks along a slope even by checking if a elements can be added or subtracted from the right side of the slope.

[7]:
```
fc.adjust_blocks_to_be_consistent_with_slopes()
win = o3plot.create_scaled_window_for_tds(tds, y_sf=y_sf)
leg = win.addLegend(offset=(750, 5), brush=(200, 200, 200), labelTextColor='k')
o3plot.plot_two_d_system(tds, win, c2='b', cs='b')
leg.addItem(pg.PlotDataItem([0], [0], pen='b'), name='2D system definition')
for i in range(len(fc.sds)):
    win.plot(fc.sds[i][0], fc.sds[i][1], pen='g', symbol='o', symbolPen='g',
             symbolBrush='g', symbolSize=5, name='Important slope')
win.plot([0, fc.tds.width], [-fc.tds.height, -fc.tds.height], pen='b')
xcs = fc.xcs_sorted
for i in range(len(xcs)):
    xc = xcs[i]
    h_blocks = np.diff(fc.yd[xc])
    dhs = h_blocks / fc.y_blocks[xc]
    y_node_steps = [0]
    for hh in range(len(fc.y_blocks[xc])):
        y_node_steps += [dhs[hh] for u in range(fc.y_blocks[xc][hh])]
    y_node_coords = np.cumsum(y_node_steps) - tds.height
    xn = xc * np.ones_like(list(fc.yd[xc]))
    win.plot(xn, list(fc.yd[xc]), symbol='o', pen='r')
    nbs = np.cumsum(fc.y_blocks[xc])
    for cc in range(len(fc.y_blocks[xc])):
        text = pg.TextItem(f'{nbs[cc]}', anchor=(0, 1))
        win.addItem(text)
        text.setPos(xc, fc.yd[xc][cc + 1])
    xn = xc * np.ones_like(y_node_coords)
    win.plot(xn, y_node_coords, symbol='+', name='Approx. node position')
    win.addItem(pg.InfiniteLine(xcs[i], angle=90, pen=(0, 255, 0, 100)))
o3plot.revamp_legend(leg)
show_plot_as_img(win)
```

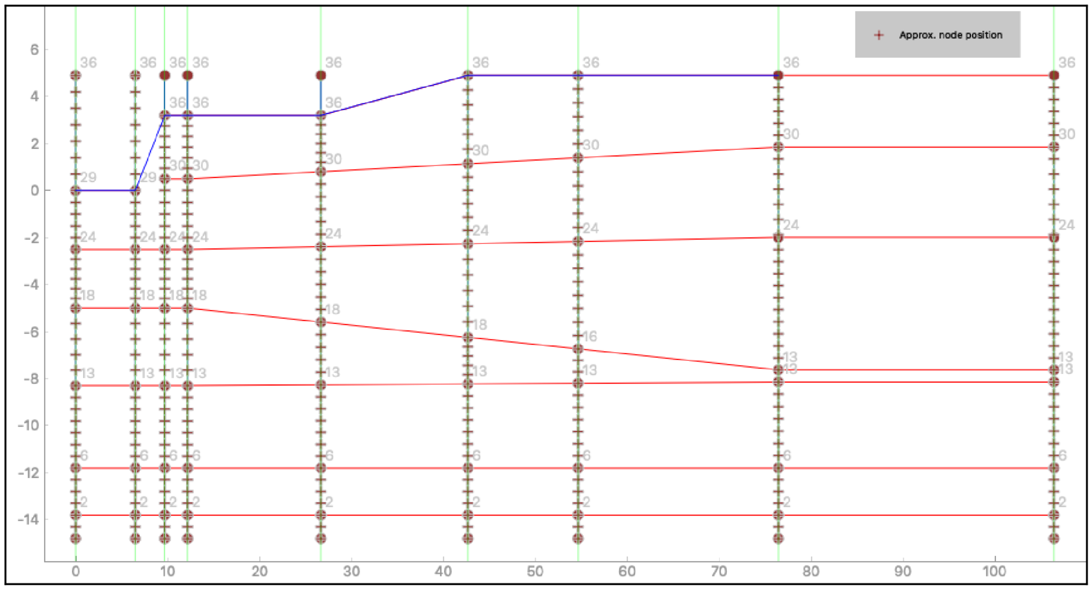


1.8 Adjust the number of blocks to try to eliminate large and small elements

1. Cycle through all zones (vertical space between two important coordinates), compute the average element height.
2. For the smallest element if it is less than the target size (`dy_target`), then try to remove a row of elements, compute the new average block height for each zone at the same height if an element is removed, if the new block height does not exceed the optimally upper limit (`dy_target * (max_scale + 1) / 2`), then remove a row of elements at that height. If an element can not be removed without exceeding the limits, then move to the next smallest. Repeat until no more blocks can be removed.
3. Repeat similar to step 2, but trying to reduce the size of larger blocks. For the largest element if it is greater than the optimally upper limit size, then try to add a row of elements, compute the new average block height for each zone at the same height if an element is added, if the new block height is not less the optimally lower limit (`dy_target * (min_scale + 1) / 2`), then add a row of elements at that height. Repeat until no more blocks can be added.

[8]:
```
fc.trim_grid_to_target_dh()
# show current state
win = o3plot.create_scaled_window_for_tds(tds, y_sf=y_sf)
leg = win.addLegend(offset=(750, 5), brush=(200, 200, 200), labelTextColor='k')
for i in range(len(fc.sds)):
    win.plot(fc.sds[i][0], fc.sds[i][1], pen='b')
win.plot([0, fc.tds.width], [-fc.tds.height, -fc.tds.height], pen='w')
xcs = fc.xcs_sorted
for i in range(len(xcs)):
    xc = xcs[i]
    h_blocks = np.diff(fc.yd[xc])
    dhs = h_blocks / fc.y_blocks[xc]
    y_node_steps = [0]
    for hh in range(len(fc.y_blocks[xc])):
        y_node_steps += [dhs[hh] for u in range(fc.y_blocks[xc][hh])]
    y_node_coords = np.cumsum(y_node_steps) - tds.height
    xn = xc * np.ones_like(list(fc.yd[xc]))
    win.plot(xn, list(fc.yd[xc]), symbol='o', pen='r')
    nbs = np.cumsum(fc.y_blocks[xc])
    for cc in range(len(fc.y_blocks[xc])):
        text = pg.TextItem(f'{nbs[cc]}', anchor=(0, 1))
        win.addItem(text)
        text.setPos(xc, fc.yd[xc][cc + 1])
    xn = xc * np.ones_like(y_node_coords)
    win.plot(xn, y_node_coords, symbol='+', name='Approx. node position')
    win.addItem(pg.InfiniteLine(xcs[i], angle=90, pen=(0, 255, 0, 100)))
win.plot(tds.x_surf, tds.y_surf, pen='r')
o3plot.revamp_legend(leg)
show_plot_as_img(win)
```

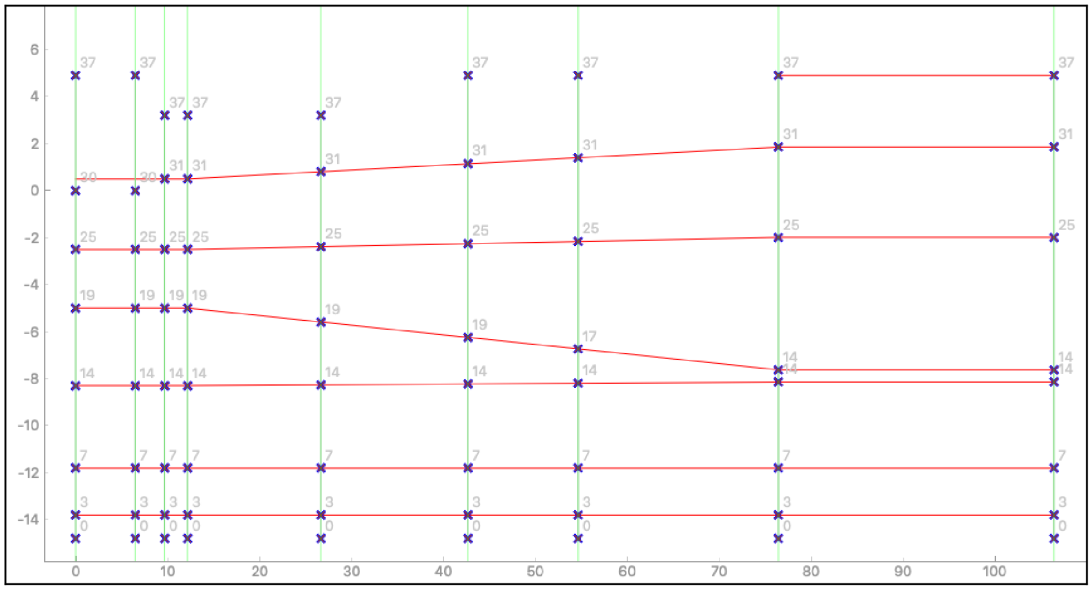
```
min_h:  0
max_h:  11
```



1.9 Finalise the number of elements in each zone

Establish additional important coordinates where there is a steep change in slope.

[9]:
```
fc.build_req_y_node_positions()
win = o3plot.create_scaled_window_for_tds(tds, y_sf=y_sf)
o3plot.plot_two_d_system(tds, win)
xcs = fc.xcs_sorted
for i, xc in enumerate(xcs):
    xn = xc * np.ones_like(list(fc.req_y_coords_at_xcs[i]))
    win.plot(xn, list(fc.req_y_coords_at_xcs[i]), symbol='x', symbolPen='r')
    for j in range(len(fc.req_y_nodes[i])):
        text = pg.TextItem(f'{fc.req_y_nodes[i][j]}', anchor=(0, 1))
        win.addItem(text)
        text.setPos(xc, fc.req_y_coords_at_xcs[i][j])
    win.addItem(pg.InfiniteLine(xcs[i], angle=90, pen=(0, 255, 0, 100)))
show_plot_as_img(win)
```

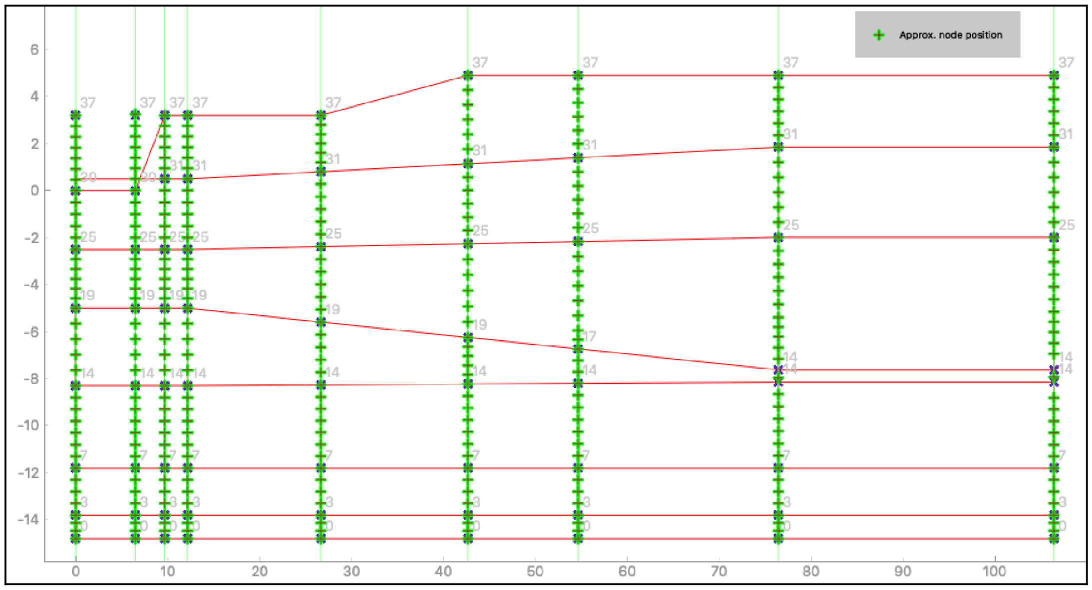


1.10 Build y-coordinates at xcs

Create the y-coordinates for each node along each significant vertical line

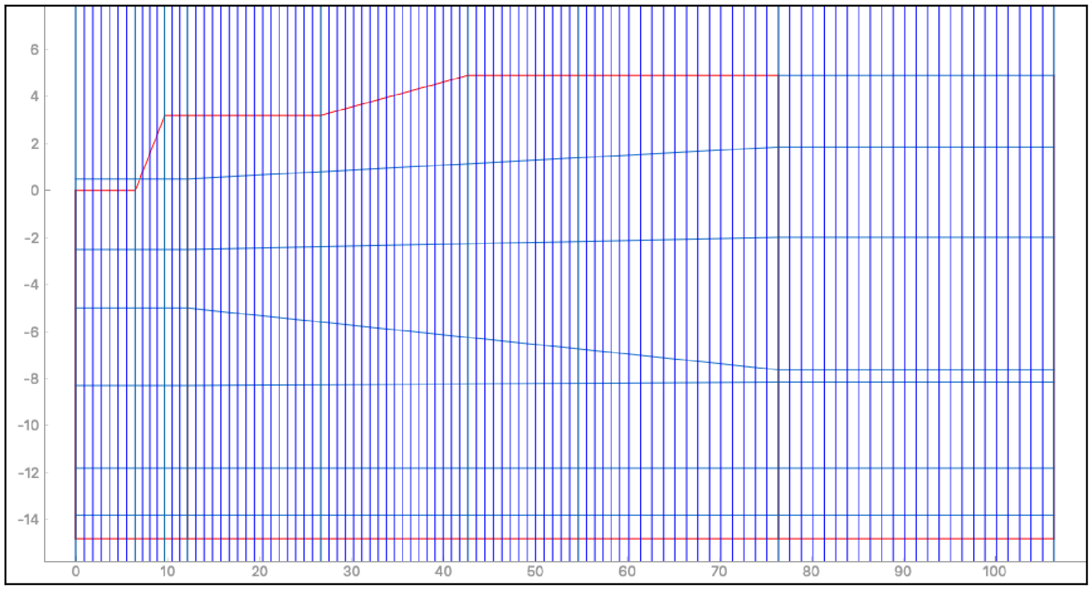
[10]:
```
fc.build_y_coords_at_xcs()
win = o3plot.create_scaled_window_for_tds(tds, y_sf=y_sf)
leg = win.addLegend(offset=(750, 5), brush=(200, 200, 200), labelTextColor='k')
o3plot.plot_two_d_system(tds, win, c2='b', cs='b')
xcs = fc.xcs_sorted
for i in range(len(xcs)):
    xc = xcs[i]
    xn = xc * np.ones_like(list(fc.req_y_coords_at_xcs[i]))
    win.plot(xn, list(fc.req_y_coords_at_xcs[i]), symbol='x', symbolPen='r')
    for j in range(len(fc.req_y_nodes[i])):
        text = pg.TextItem(f'{fc.req_y_nodes[i][j]}', anchor=(0, 1))
        win.addItem(text)
        if j != 0 and fc.req_y_coords_at_xcs[i][j] == fc.
 ↪req_y_coords_at_xcs[i][j - 1]:
            text.setPos(xc, fc.req_y_coords_at_xcs[i][j] + 0.3)
        else:
            text.setPos(xc, fc.req_y_coords_at_xcs[i][j])
    xa = xc * np.ones_like(list(fc.y_coords_at_xcs[i]))
    win.plot(xa, list(fc.y_coords_at_xcs[i]), symbol='+', symbolPen='g',
             name='Approx. node position')
    win.addItem(pg.InfiniteLine(xcs[i], angle=90, pen=(0, 255, 0, 100)))
o3plot.revamp_legend(leg)
```

```
show_plot_as_img(win)
```



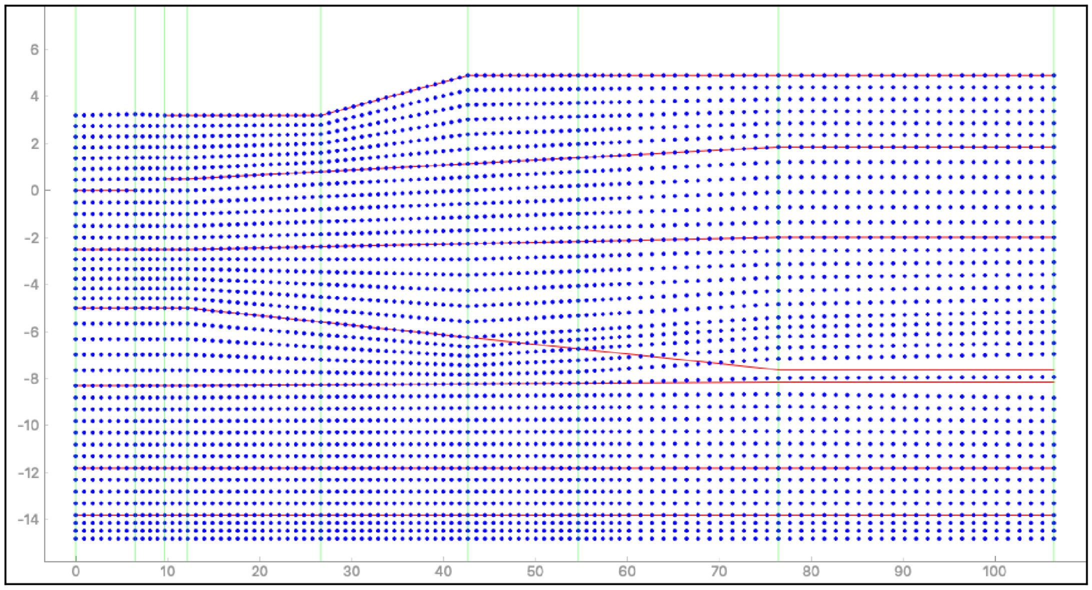
1.11 Set the x position of the nodes

[11]:
```
fc.set_x_nodes()
win = o3plot.create_scaled_window_for_tds(tds, y_sf=y_sf)
o3plot.plot_two_d_system(tds, win, c2='b', cs=(200, 110, 20))
for i in range(len(fc.x_nodes)):
    win.addItem(pg.InfiniteLine(fc.x_nodes[i], angle=90, pen='r'))
for i in range(len(fc.xcs_sorted)):
    win.addItem(pg.InfiniteLine(fc.xcs_sorted[i], angle=90, pen=(0, 255, 0,␣
 ↪100)))
show_plot_as_img(win)
```



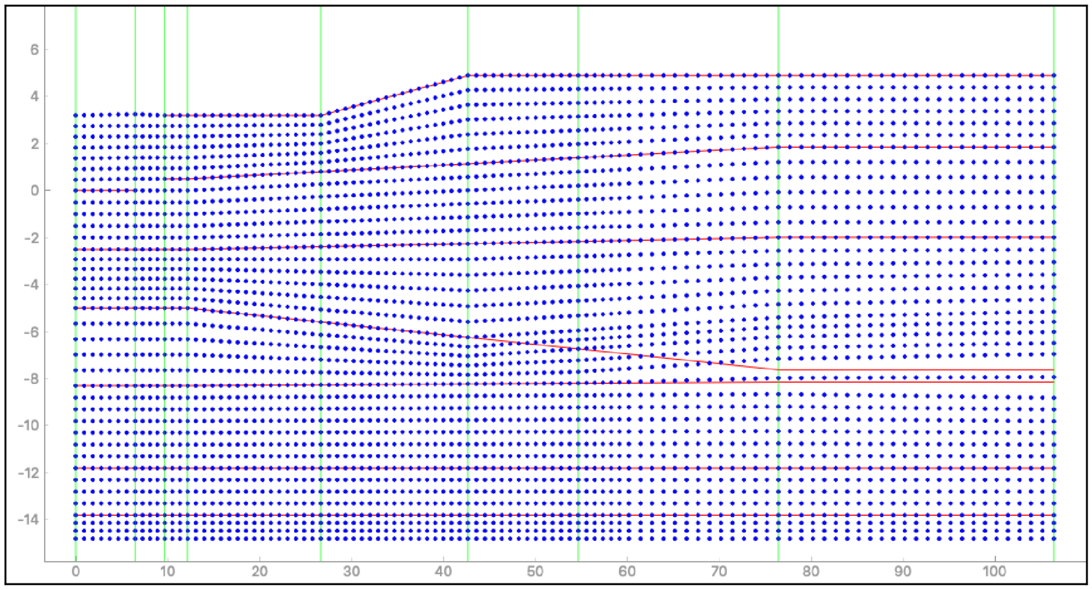
1.12 Determine the y-coordinates of each node by propagating the position of the y-coordinates from left to right.

[12]:
```
fc.build_y_coords_grid_via_propagation()
win = o3plot.create_scaled_window_for_tds(tds,␣
 ↪title='build_y_coords_grid_via_propagation', y_sf=y_sf)
for i in range(len(fc.sds)):
    win.plot(fc.sds[i][0], fc.sds[i][1], pen='b')
win.plot([0, fc.tds.width], [-fc.tds.height, -fc.tds.height], pen='w')
xns = fc.x_nodes
for i in range(len(xns)):
    xc = xns[i]
    xn = xc * np.ones_like(fc.y_nodes[i])
    win.plot(xn, fc.y_nodes[i], pen=None, symbol='o', symbolPen='r',␣
 ↪symbolBrush='r', symbolSize=3)
for i in range(len(fc.xcs_sorted)):
    win.addItem(pg.InfiniteLine(fc.xcs_sorted[i], angle=90, pen=(0, 255, 0,␣
 ↪100)))
show_plot_as_img(win)
```



1.13 Set the coordinates to nearest decimal points

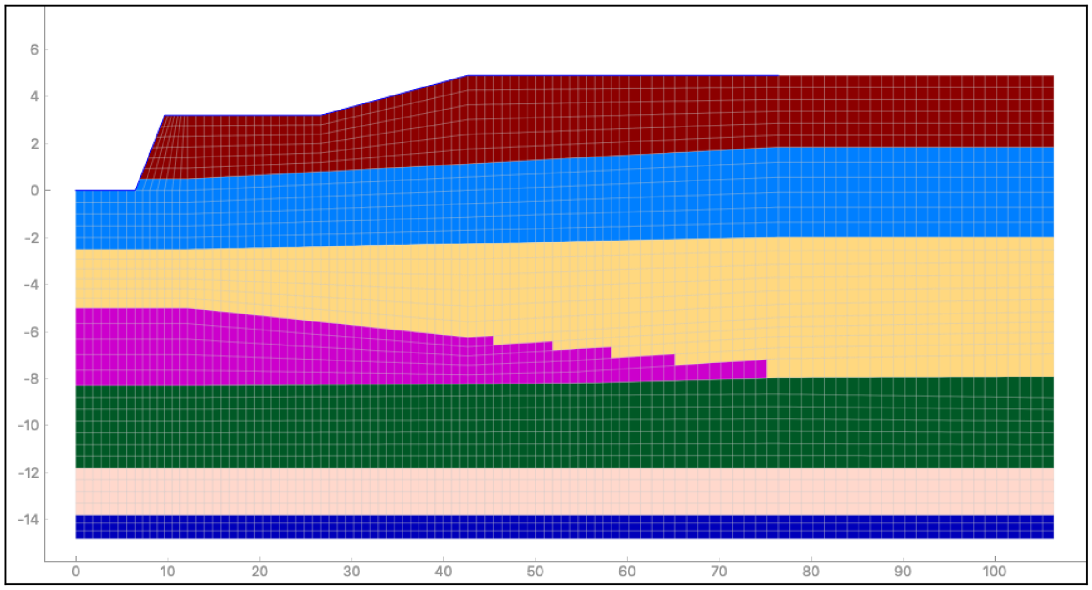
[13]:
```python
# First plot current state
win = o3plot.create_scaled_window_for_tds(tds, title='set_to_decimal_places',␣
 ↪y_sf=y_sf)
for i in range(len(fc.sds)):
    win.plot(fc.sds[i][0], fc.sds[i][1], pen='b')
win.plot([0, fc.tds.width], [-fc.tds.height, -fc.tds.height], pen='w')
xns = fc.x_nodes
for i in range(len(xns)):
    xc = xns[i]
    xn = xc * np.ones_like(fc.y_nodes[i])
    win.plot(xn, fc.y_nodes[i], pen=None, symbol='o', symbolPen=0.5,␣
 ↪symbolBrush=0.5, symbolSize=3)
for i in range(len(fc.xcs_sorted)):
    win.addItem(pg.InfiniteLine(fc.xcs_sorted[i], angle=90, pen=(0, 255, 0,␣
 ↪100)))
# move nodes to bearest decimal point
fc.dp = 2  # number of decimal points
fc.set_to_decimal_places()
# plot new state
xns = fc.x_nodes
for i in range(len(xns)):
    xc = xns[i]
    xn = xc * np.ones_like(fc.y_nodes[i])
    win.plot(xn, fc.y_nodes[i], pen=None, symbol='o', symbolPen='r',␣
 ↪symbolBrush='r', symbolSize=3)
for i in range(len(fc.xcs_sorted)):
    win.addItem(pg.InfiniteLine(fc.xcs_sorted[i], angle=90, pen=(0, 255, 0,␣
 ↪100)))
show_plot_as_img(win)
```



1.14 Smooth the surface

Adjust the x-coordinates of the nodes near the surface to make sure that there are no orthogonal sets of coordinates. If the number of elements in the x-direction is less than the number of elements in the y-direction between two important coordinates, then make the slope smooth, else use a series of sloped steps.

[14]:
```
fc.adjust_for_smooth_surface()
fc.set_soil_ids_to_vary_xy_grid()
fc.create_mesh()
femesh = fc.femesh
win = o3plot.create_scaled_window_for_tds(tds, y_sf=y_sf)
# fc.femesh.soil_grid[np.where(fc.femesh.soil_grid == fc.femesh.inactive_value)]␣
 ↪= 4  # view unused mesh
o3plot.plot_finite_element_mesh_onto_win(win, fc.femesh)
win.plot(tds.x_surf, tds.y_surf, pen='r')
show_plot_as_img(win)
```


6
4
2
0
-2
-4
-6
-8
-10
-12
-14
0
10
20
30
40
50
60
70
80
90
100