# Test

## Test method

This project is tested by **unittest**, a built-in Python module that provides a framework for writing and running unit tests.

## Test functions

### IO: class TestIO(unittest.TestCase):

#### read\_data function:

```

def test\_read\_data(self):

    # Test if read\_data function returns list

    geology = io.read\_data('InputData/geology.txt')

    self.assertIsInstance(geology, list)

    population = io.read\_data('InputData/population.txt')

    self.assertIsInstance(population, list)

    transport = io.read\_data('InputData/transport.txt')

    self.assertIsInstance(transport, list)

```

#### write\_data function:

```

def test\_write\_data(self):

    # Test if write\_data function output correctly

    # Define input data

    data = [[1, 2, 3], [4, 5, 6]]

    # Define output file path

    file\_path = 'OutputData/test\_output.txt'

    # Write data to file

    io.write\_data(file\_path, data)

    # Read data from file

    output\_data = io.read\_data(file\_path)

    # Remove output file

    os.remove(file\_path)

    # Assert that data is correctly written to file

    self.assertEqual(output\_data, data)

```

### Geometry: class TestGeometry(unittest.TestCase):

#### Multiply function:

```

def test\_mul(self):

    # Define input data and factors

    data1 = [[1, 2], [3, 4]]

    factor1 = 0.2

    data2 = [[5, 6], [7, 8]]

    factor2 = 0.3

    data3 = [[9, 10], [11, 12]]

    factor3 = 0.5

    # Define expected output

    expected\_output = [[6.2, 7.2], [8.2, 9.2]]

    # Call function and get actual output

    actual\_output = gm.multiply(data1, factor1, data2, factor2, data3, factor3)

    # Assert that actual output matches expected output

    self.assertEqual(actual\_output, expected\_output)

```

#### Get\_rows\_cols function:

```

def test\_get\_rows\_cols(self):

    data = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

    n\_rows, n\_cols = gm.get\_rows\_cols(data)

    self.assertEqual(n\_rows, 3)

    self.assertEqual(n\_cols, 3)

```

#### Get\_max\_min function:

```

def test\_get\_max\_min(self):

    # Define input data

    data = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

    # Get expected output

    expected\_max = 9

    expected\_min = 1

    # Get actual output

    actual\_max, actual\_min = gm.get\_max\_min(data)

    # Assert that actual output equals expected output

    self.assertEqual(actual\_max, expected\_max)

    self.assertEqual(actual\_min, expected\_min)

```

#### Rescale function:

```

def test\_rescale(self):

    # Define input data

    data = [[10, 235], [127, 63]]

    # Define expected output data

    expected\_output = [[0, 255], [133, 60]]

    # Get actual output data

    actual\_output = gm.rescale(data)

    # Assert that actual output matches expected output

    self.assertEqual(actual\_output, expected\_output)

```

## Test result

Every function goes well and correctly.

**```**

......

----------------------------------------------------------------------

Ran 6 tests in 0.101s

OK

<unittest.runner.TextTestResult run=6 errors=0 failures=0>

```

# Major issues

## Factor weight issue

#### Issue:

The factor weights are determined by the sliders. In the first edition of the code, all the sliders ranged from 0 to 1 and are not affected by each other. In this case, users could not control the proportion of the factor weights. For example, all three factors could be set to 50% (0.5) and the three weights contributed the same, which meant that the numerical value of the weight number itself did not make any sense in the calculation.

图形用户界面

描述已自动生成

#### Solution:

To enable users to directly understand the proportion of each factor in the result raster through the weight value, I keep the sum of three factors weights to 1. To achieve this, I set a function to control the range of sliders.

#### Code & Explanation:

1. **Get the value of the sliders**

```

    geoget = int(scale1.get())

    popget = int(scale2.get())

```

1. **When the geo slider changes its value, the range of the pop slider changes into the range from 0 to (100-geoget)**

```

    if (geoget != (geofac\*100)): # when geoslider changes

        scale1\_label.config(text='Geology factor = ' + str(geoget) + '%') # change the label1 text

        scale2.config(to=(100-geoget), length=(100-geoget)\*5, state="normal") # change scale range

        write\_button.config(state='disabled') # close write button

```

1. **When the pop slider changes its value, the value of the tra slider can be determined as (100-geoget-popget)**

```

    if (popget != (popfac\*100)): # when popslider changes

        scale2\_label.config(text='Population factor = ' + str(popget) + '%') # change the label2 text

        scale3.config(from\_=(100-geoget-popget), to=(100-geoget-popget), length=(100-geoget-popget)\*5, state="normal") # change scale range

        scale3\_label.config(text='Transport factor = ' + str(int(100-geoget-popget)) + '%') #autoly change label3 text

        scale1.config(state='disabled') # close scale1

```

1. **Set the factor weight**

```

    # Change the factors' value to the new ones, in two decimal places ranging (0, 1)

    geofac = geoget/100

    popfac = popget/100

    trafac = (100-geoget-popget)/100

```

#### Result:

At first, the user can only move the first slider to determine the weight of the geology factor. When the value of the first slider is moving, the range of the second one and third one is changing corresponding to the first slider and ensure that the sum of the maximum value of the second slider range and the value of the first slider is 1. After setting up the first slider, the user can move the second slider. Similarly, the range of the third slider changes in the same rules when the second slider moves. The sum of the final three weight values is equal to 1.

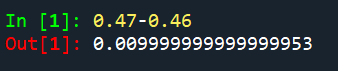
图形用户界面

描述已自动生成

## The calculation error of the floating-point number

#### Issue:

In the first edition of the code, the ranges of factor weights were set from 0 to 1. In this case, the `scale.get()` method got the value in the floating-point number type. So, when resetting the factor sliders range, the code had to do a floating-point number calculation. However, floating-point numbers have a finite representation in a computer and are stored in binary, so there are rounding errors when performing calculations. For example, when the code was trying to calculate the formula of 0.47-0.46, the result was not 0.01 but 0.009999999999999953, which means the calculation result is lack of accuracy. Following is the example screenshot.



#### Solution:

If using the floating-point number to calculate, this kind of error cannot be avoided. As a result, the only solution is to abandon floating-point type numbers and use integer numbers to calculate. The ranges of sliders should be set from 0 to 100 and the values should be forced to convert to int type numbers and stored in variables. All subsequent calculations should be performed via variables in int type. In this case, the int type number will have no error.

#### Code & Explanation:

**Take scale1 (geo slider) as an example**

1. **Set the range of the slider from 0 to 100**

```

scale1 = ttk.Scale(frame, from\_=0, to=100, command=labeling, length=500)

```

1. **In the labelling function, the values of the sliders are forced to be converted to int and stored in a variable**

```

    # Get scale value

    geoget = int(scale1.get())

```

1. **Perform subsequent calculations**

```

    scale2.config(to=(100-geoget), length=(100-geoget)\*5, state="normal") # change scale range

```

1. **Rescale the slider value to between 0 and 1 and update the calculating weight value**

```

    # Change the factors' value to the new ones, in two decimal places ranging (0, 1)

    geofac = geoget/100

```

#### Result:

Users can only move the sliders one by one to ensure the accurate calculation of the slider range. Regardless of how the sliders move, the sum of the weights of the three factors is equal to 1. Users can check the labels behind the sliders to be informed of the weights of factors. Following is the example screenshot.

文本

中度可信度描述已自动生成

## Reset the weights

#### Issue:

To ensure that all calculations were correct, the slider was locked after it had been moved and could not be changed again. Therefore the user could not change the weights if they had accidentally selected the wrong weights.

#### Solution:

A reset button should be added to allow users to move the sliders again. The button should be linked to a reset function which sets the value of all sliders to 0 and returns the state of the slider to its initial state.

#### Code & Explanation:

```

def reset():

    # Transfer the 3 global variable

    global geofac, popfac, trafac

    # Change the factors' value to 0

    geofac = 0

    popfac = 0

    trafac = 0

    # Reset widgets' state and value

    scale1.config(state='normal')

    scale1.set(0)

    scale1\_label.config(text='Transport factor = ' + str(0) + '%')

    scale2.config(state='normal')

    scale2.set(0)

    scale2.config(length=0)

    scale2\_label.config(text='Transport factor = ' + str(0) + '%')

    scale3.config(state='normal')

    scale3.config(from\_=0, to=0, length=0)

    scale3.set(0)

    scale3\_label.config(text='Transport factor = ' + str(0) + '%')

    scale1.config(state='normal')

    scale2.config(state='disabled')

    scale3.config(state='disabled')

```

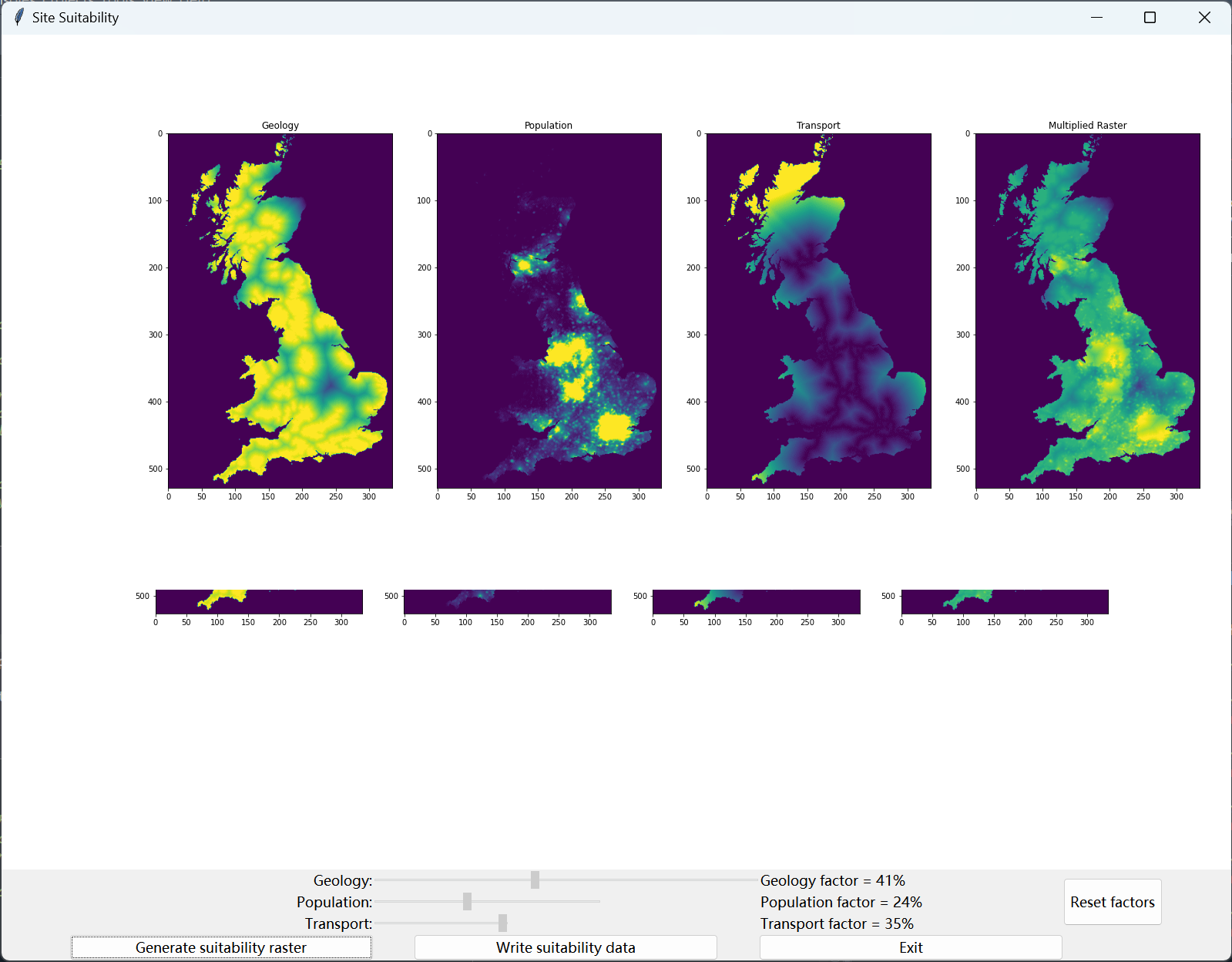
#### Result:

When users click the reset button, the values of the geo slider and pop slider are set to 0 and users can move the first slider again. However, there is still a problem that the third slider (tra slider) is set to 100% when users use the reset button.

## Interface issue

#### Issue:

The images of three factors and the final multiplied resulting raster are shown on one figure and linked to one canvas in the interface. In the first edition, when users chose to generate the result raster, the old figure which showed the three factors images closed and a new figure showed factors images and resulting raster appeared in the canvas. However, due to the different screen resolutions, the size of the new figure was inconsistent with the size of the original figure and the new figure overlapped with the original one.



#### Solution:

I have tried to abandon using the Agg in matplotlib but using TkAgg. One of the main differences between Agg and TkAgg is that TkAgg allows interactive figure display, while Agg is used for non-interactive image rendering. In the other word, Agg suppresses the appearance of figure pop and lets the figure run on the software backend while TkAgg let the figure pop up with the main software interface. However, the canvas in this project is initialised by tk, so the code cannot run normally in Agg backend.

I continued to try to use other methods. Finally, I chose to change the figure itself rather than create a new figure. Whenever a new resulting raster is generated, subfigure 4 is first cleared and then draws the new resulting raster. What is more, to prevent pop-ups in the spyder from disturbing the user, the main interface was kept at the top of all windows.

#### Code & Explanation:

1. **Import packages and use TkAgg backend renderer**

```

    import matplotlib

    matplotlib.use('TkAgg')

```

1. **Add the new resulting raster to the subfigure4**

```

    # subplot 4: multiplied raster

    plt.subplot(1, 4, 4)# the fourth one

    plt.cla() # clear the subfigure

    plt.imshow(sum\_raster)

    plt.title('Multiplied Raster')

    # Set the interval between individual subplots

    plt.subplots\_adjust(wspace=0.3, hspace=0)

    # Show the overall plot at canva1

    canva1.draw()

```

1. **Keep the main interface of the software at the most top**

```

    # Create the tkinter window

    root = tk.Tk()

    root.attributes("-topmost", True) # keep window at top

```

#### Result:

Each time the user clicks the Generate button, a new figure window will pop up and disappear in a flash and the user can only see the main interface. No matter how many other windows users have open, the software is always displayed at the top of all windows. No matter how many times the users repeat the generation of the resulting raster, no image overlap occurs.

