Regression in C

Release 0.1.0a0

Neeraj Shah

Mar 15, 2023

USER GUIDE:

Regression in C is a package focussing on regression for complex-valued problems. There are already multiple packages for regression in Python, however these offer varying amounts of support for the complex-valued variables. Numpy's least squares implementation has partial support for complex-valued variables and other packages support multi-target regression, which can be used for complex-valued problems, but again support is spotty.

This package is currently being developed. The plan for current and future development includes:

- Least squares and weighted least squares implementation
- Visualisation of complex-valued linear problems
- Robust linear regression for complex-valued variables
- Uncertainty estimation

CHAPTER

ONE

NEXT STEPS

1.1 Examples

Check out some examples here

1.1.1 Linear regression

This example begins with linear regression in the real domain and then builds up to show how linear problems can be thought of in the complex domain.

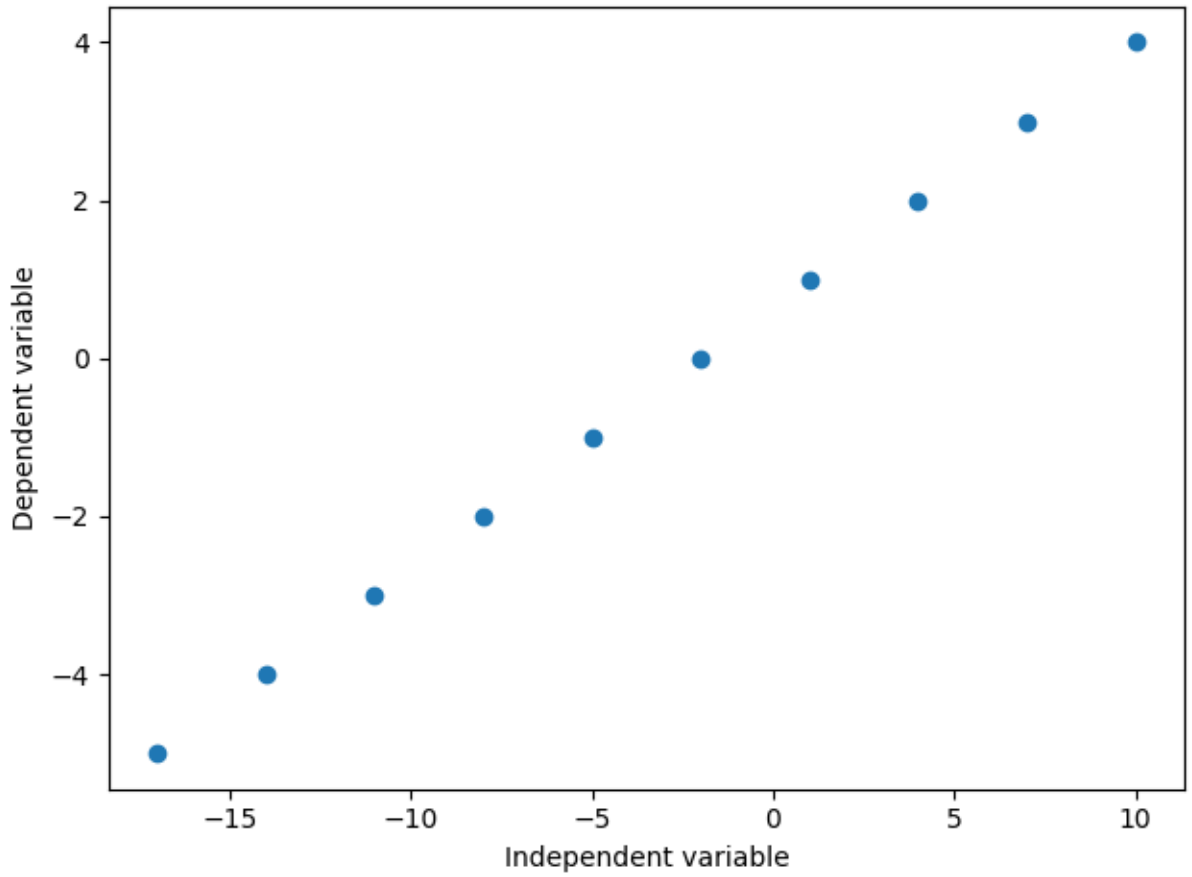
Useful references:

- https://stats.stackexchange.com/questions/66088/analysis-with-complex-data-anything-different
- https://www.chrishenson.net/article/complex_regression

```
import numpy as np
import matplotlib.pyplot as plt
from regressioninc.linear import add_intercept, LeastSquares
from regressioninc.testing.complex import ComplexGrid
```

One of the most straightforward linear problems to understand is the equation of line. Let's look at a line with gradient 3 and intercept -2.

```
coef = np.array([3])
intercept = -2
X = np.arange(-5, 5).reshape(10, 1)
y = X * coef + intercept

fig = plt.figure()
plt.scatter(y, X)
plt.xlabel("Independent variable")
plt.ylabel("Dependent variable")
plt.tight_layout()
fig.show()
```



When performing linear regressions, the aim is to:

- calculate the coefficients (coef, also called parameters)
- given the regressors (X, values of the independent variable)
- and values of the observations (y, values of the dependent variable)

This can be done with linear regression, and the most common method of linear regression is least squares, which aims to estimate the coefficients whilst minimising the squared misfit between the observations and estimated observations calculated using the estimated coefficients.

```
X = add_intercept(X)
model = LeastSquares()
model.fit(X, y)
print(model.coef)
```

```
[[ 3.]
 [-2.]]
```

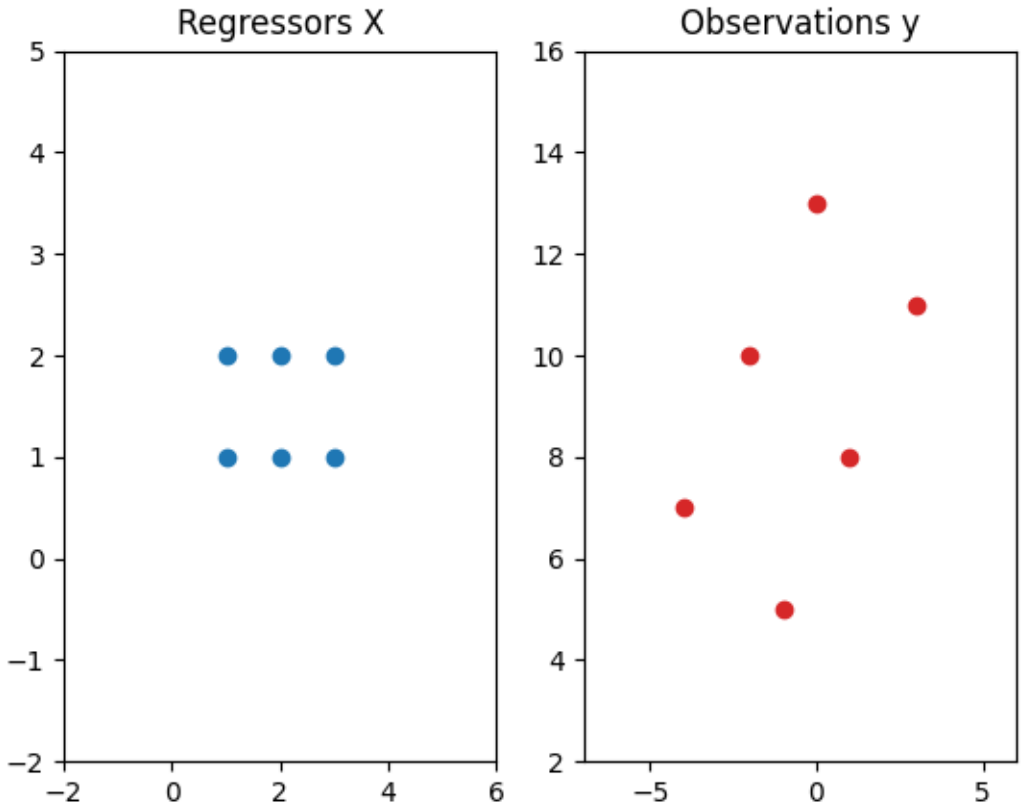
Least squares was able to correctly calculate the slope and intercept for the real-valued regression problem.

It is also possible to have linear problems in the complex domain. These commonly occur in signal processing problems. Let's define coefficients and regressors X and calculate out the observations y.

```
coef = np.array([2 + 3j])
X = np.array([1 + 1j, 2 + 1j, 3 + 1j, 1 + 2j, 2 + 2j, 3 + 2j]).reshape(6, 1)
y = np.matmul(X, coef)
```

It is a bit harder to visualise the complex-valued version, but let's try and visualise the regressors X and observations y.

```
fig, axs = plt.subplots(nrows=1, ncols=2)
plt.sca(axs[0])
plt.scatter(X.real, X.imag, c="tab:blue")
plt.xlim(X.real.min() - 3, X.real.max() + 3)
plt.ylim(X.imag.min() - 3, X.imag.max() + 3)
plt.title("Regressors X")
plt.sca(axs[1])
plt.scatter(y.real, y.imag, c="tab:red")
plt.xlim(y.real.min() - 3, y.real.max() + 3)
plt.ylim(y.imag.min() - 3, y.imag.max() + 3)
plt.title("Observations y")
plt.show()
```

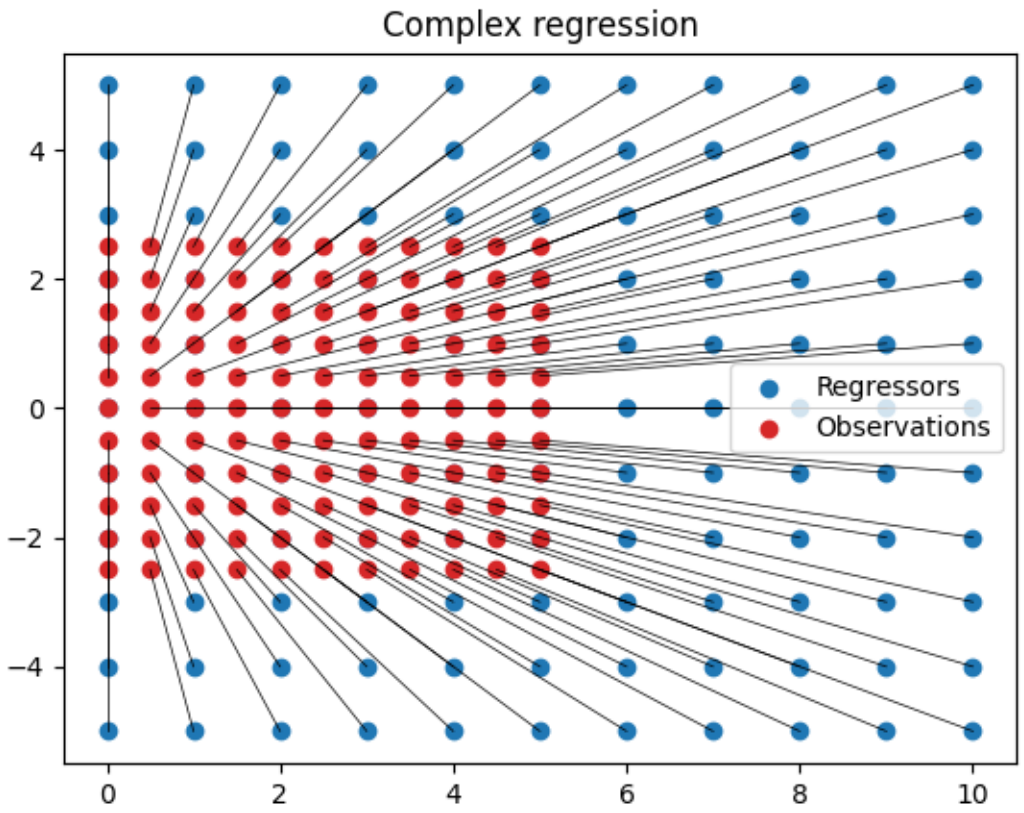


Visualsing the regressors X and the observations y this way gives a geometric indication of the linear problem in the complex domain. Multiplying the regressors by the coefficients can be considered like a scaling and a rotation of the independent variables to give the observations y (or the dependent variables).

With more samples, this can be a bit easier to visualise. In the below example regressors and observations are generated

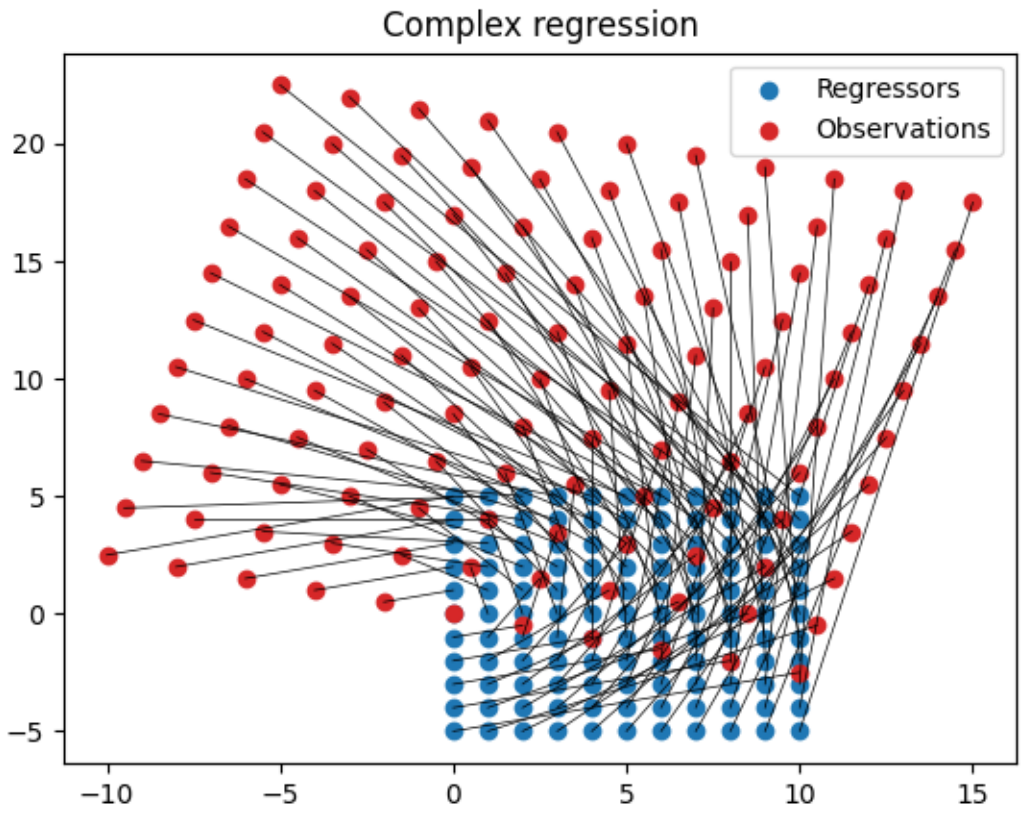
again, this time with more samples. To start off with, the coefficient is a real number to demonstrate the scaling without any rotation. Both the regressors and observations are plotted on the same axis with lines to show the mapping between independent and dependent values.

```
grid = ComplexGrid(r1=0, r2=10, nr=11, i1=-5, i2=5, ni=11)
X = grid.flat_grid()
coef = np.array([0.5])
y = np.matmul(X, coef)

fig = plt.figure()
for iobs in range(y.size):
    plt.plot(
        [y[iobs].real, X[iobs, 0].real],
        [y[iobs].imag, X[iobs, 0].imag],
        color="k",
        lw=0.5,
    )
plt.scatter(X.real, X.imag, c="tab:blue", label="Regressors")
plt.grid()
plt.title("Regressors X")
plt.scatter(y.real, y.imag, c="tab:red", label="Observations")
plt.grid()
plt.legend()
plt.title("Complex regression")
plt.show()
```



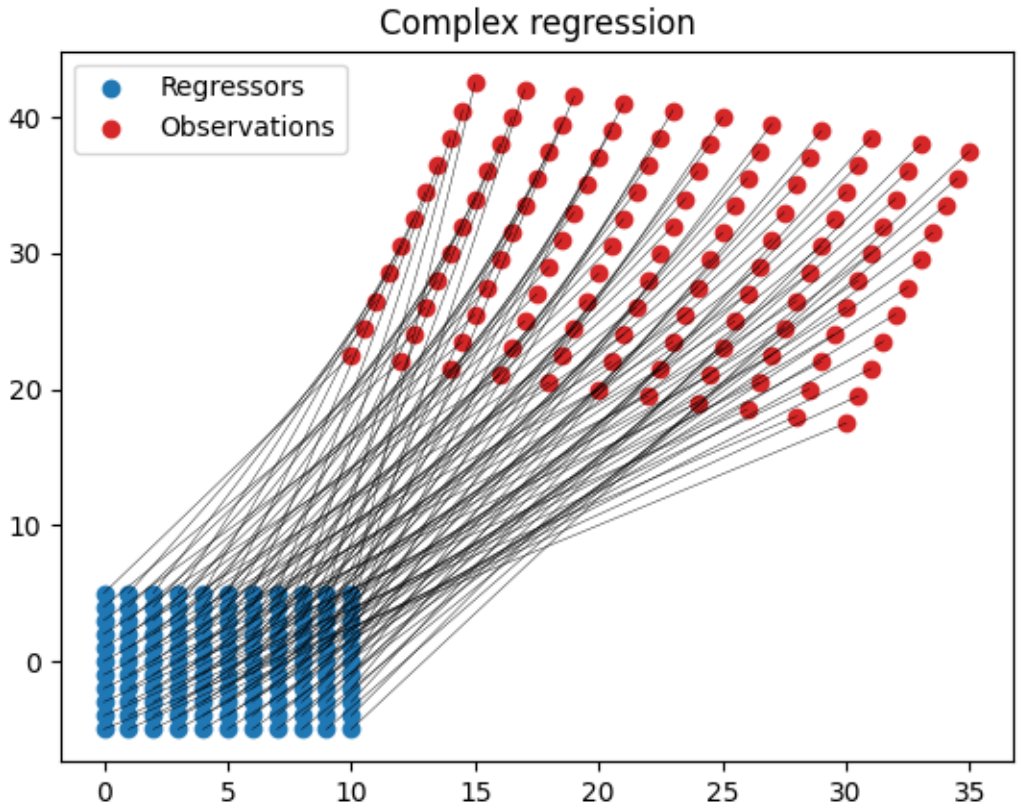
Now let's add a complex component to the coefficient to demonstrate the rotational aspect.

```
coef = np.array([0.5 + 2j])
y = np.matmul(X, coef)

fig = plt.figure()
for iobs in range(y.size):
    plt.plot(
        [y[iobs].real, X[iobs, 0].real],
        [y[iobs].imag, X[iobs, 0].imag],
        color="k",
        lw=0.5,
    )
plt.scatter(X.real, X.imag, c="tab:blue", label="Regressors")
plt.grid()
plt.title("Regressors X")
plt.scatter(y.real, y.imag, c="tab:red", label="Observations")
plt.grid()
plt.legend()
plt.title("Complex regression")
plt.show()
```



Finally, adding an intercept gives a translation.

```
coef = np.array([0.5 + 2j])
intercept = 20 + 20j
y = np.matmul(X, coef) + intercept

fig = plt.figure()
for iobs in range(y.size):
    plt.plot(
        [y[iobs].real, X[iobs, 0].real],
        [y[iobs].imag, X[iobs, 0].imag],
        color="k",
        lw=0.3,
    )
plt.scatter(X.real, X.imag, c="tab:blue", label="Regressors")
plt.grid()
plt.title("Regressors X")
plt.scatter(y.real, y.imag, c="tab:red", label="Observations")
plt.grid()
plt.legend()
plt.title("Complex regression")
plt.show()
```



Similar to the real-valued problem, linear regression can be used to estimate the values of the coefficients for the complex-valued problem. Again, least squares is one of the most common methods of linear regression. However, not all least squares algorithms support complex data, though some do such as the least squares in numpy. The focus of regressioninc is to provide regression methods for complex-valued data.

Note that adding an intercept column to X allows for solving of the intercept. Regression in C does not automatically solve for the intercept and if desired, an intercept column needs to be added to the regressors.

```
X = add_intercept(X)
model = LeastSquares()
model.fit(X, y)
print(model.coef)
```

```
[ 0.5 +2.j 20. +20.j]
```

Total running time of the script: (0 minutes 2.124 seconds)

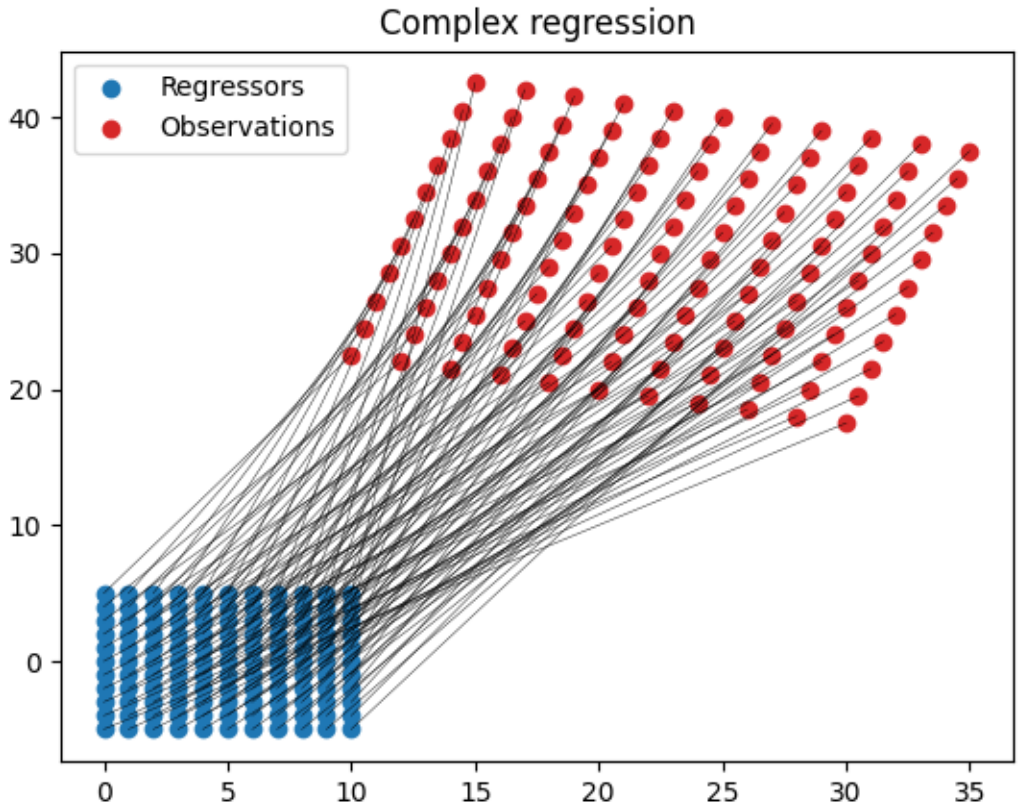
1.1.2 Multiple regressors

The previous example showed complex-valued regression with a single regressor. In practice, it is common to have multiple regressors. The following example will generate a complex-valued linear problem with multiple regressors and try and visualise it.

```
import numpy as np
import matplotlib.pyplot as plt
from regressioninc.linear import add_intercept, LeastSquares
from regressioninc.testing.complex import generate_linear_random, plot_complex
from regressioninc.testing.complex import ComplexGrid

np.random.seed(42)
```

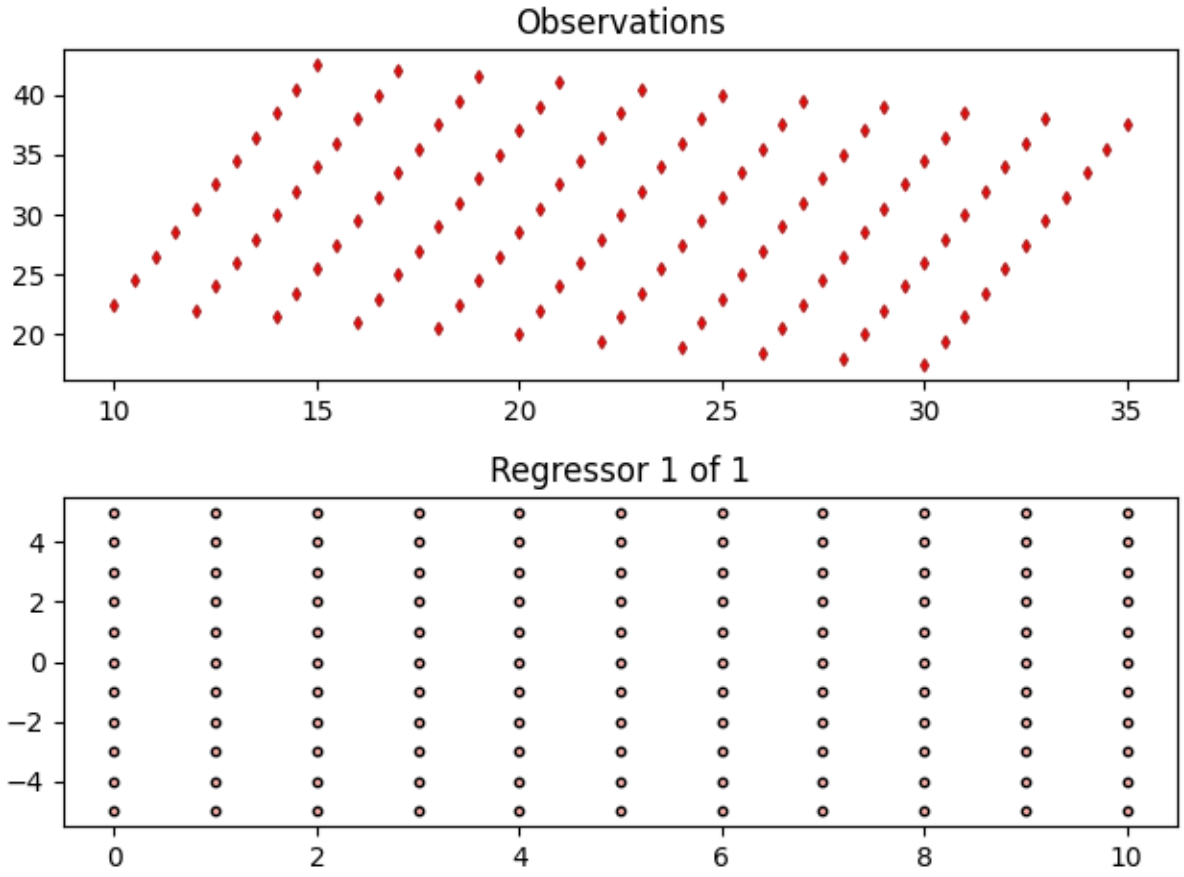
Let's begin where the previous example ended.

```
grid = ComplexGrid(r1=0, r2=10, nr=11, i1=-5, i2=5, ni=11)
X = grid.flat_grid()
coef = np.array([0.5 + 2j])
intercept = 20 + 20j
y = np.matmul(X, coef) + intercept

fig = plt.figure()
for iobs in range(y.size):
    plt.plot(
        [y[iobs].real, X[iobs, 0].real],
        [y[iobs].imag, X[iobs, 0].imag],
        color="k",
        lw=0.3,
    )
plt.scatter(X.real, X.imag, c="tab:blue", label="Regressors")
plt.grid()
plt.title("Regressors X")
plt.scatter(y.real, y.imag, c="tab:red", label="Observations")
plt.grid()
plt.legend()
plt.title("Complex regression")
plt.show()
```



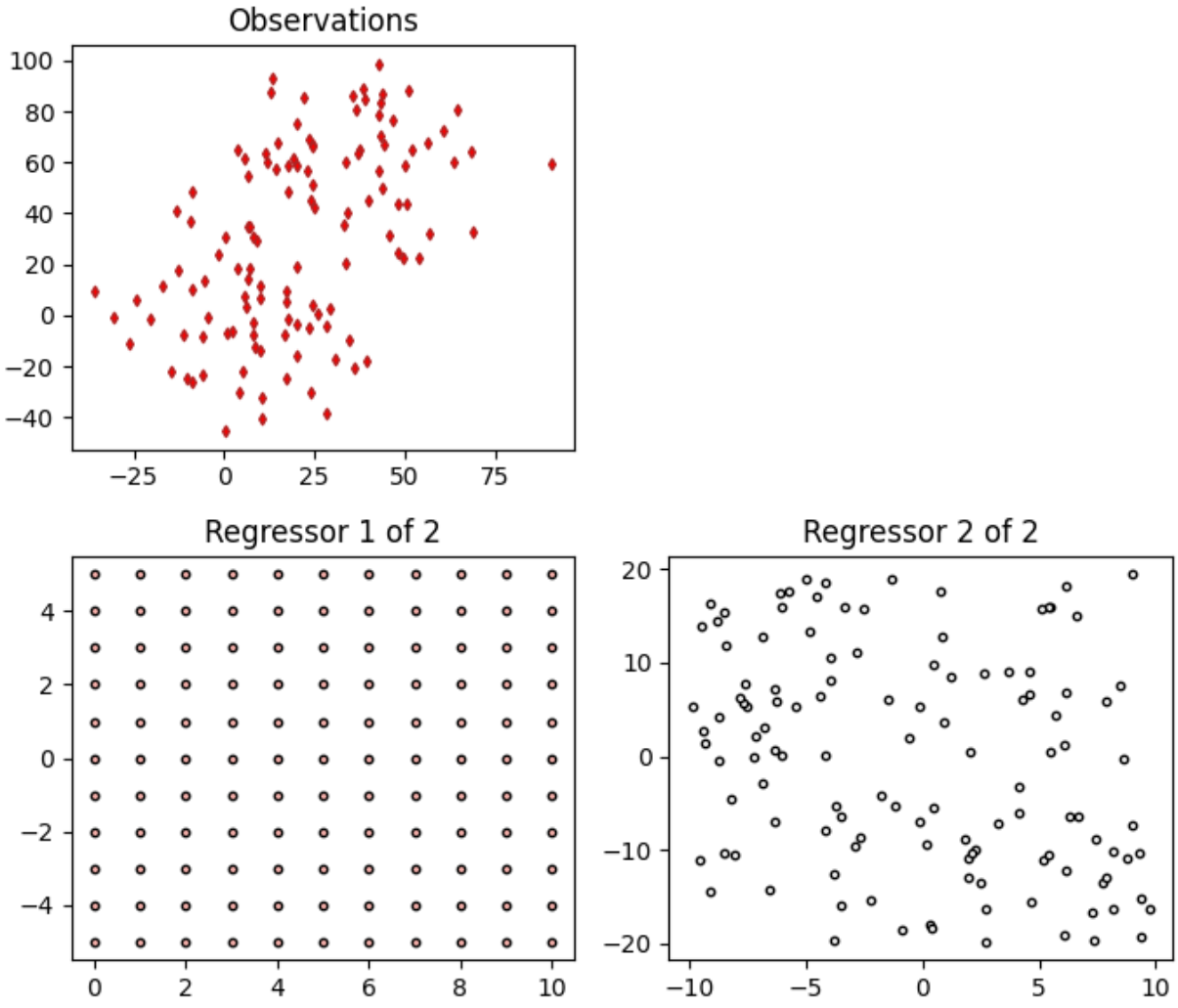
Now plot this in a different way that will make it easier to visualise more than a single regressor.

```
fig = plot_complex(X, y, {})
plt.show()
```



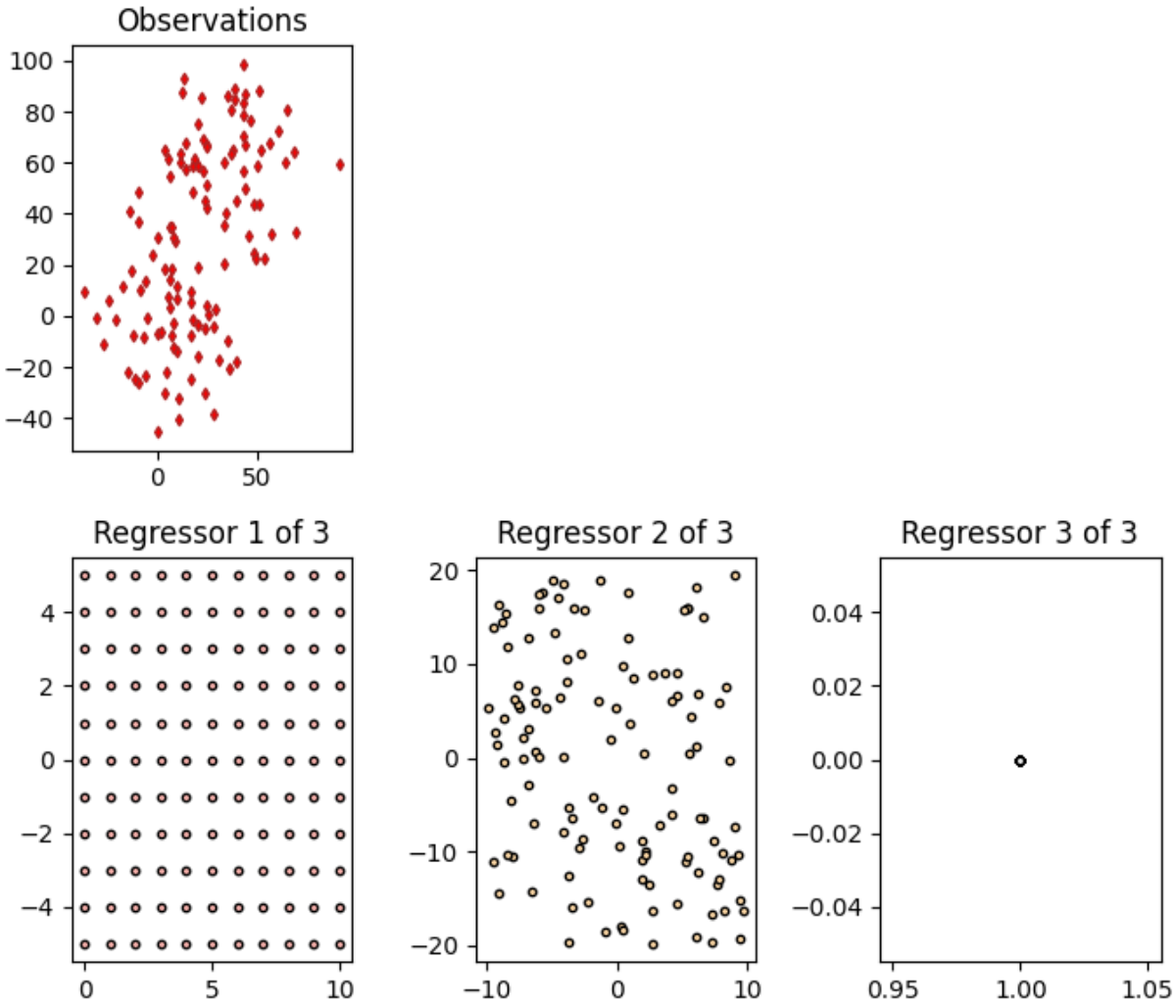
Let's add in a second regressor but rather than having a grid of input points use some random input points.

```
coef_random = np.array([2.7 - 1.8j])
X_random, _ = generate_linear_random(coef_random, y.size)
coef = np.concatenate((coef, coef_random))
X = np.concatenate((X, X_random), axis=1)
intercept = 20 + 20j
y = np.matmul(X, coef) + intercept

fig = plot_complex(X, y, {})
fig.set_size_inches(7, 6)
plt.tight_layout()
plt.show()
```



These examples have been adding an intercept to y. To solve for an intecerpt, an intercept column needs to be explicitly added to the regressors X before passing X through to the model. Adding an intercept simply adds a column of 1s to the regressors.

```
X = add_intercept(X)
fig = plot_complex(X, y, {})
fig.set_size_inches(7, 6)
plt.tight_layout()
plt.show()
```



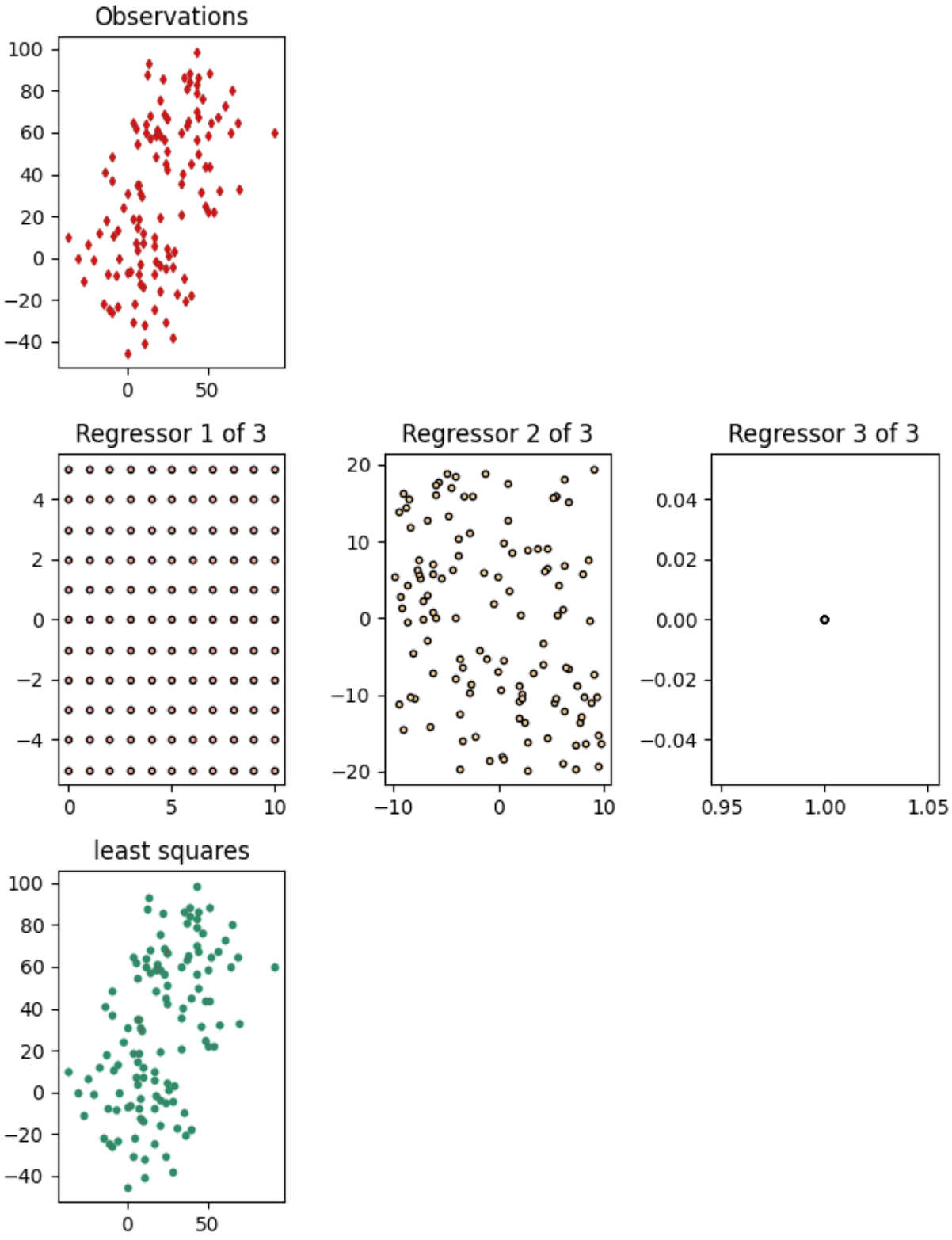
Now the coefficients can be solved for using the observations y and the regressors X.

```
model = LeastSquares()
model.fit(X, y)
for idx, coef in enumerate(model.coef):
    print(f"Coefficient {idx}: {coef:.6f}")
```

```
Coefficient 0: 0.500000+2.000000j
Coefficient 1: 2.700000-1.800000j
Coefficient 2: 20.000000+20.000000j
```

Finally, the estimated model observations can be added to the visualisation.

```
fig = plot_complex(X, y, {"least squares": model})
fig.set_size_inches(7, 9)
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 2.293 seconds)

1.1.3 Random noise

Unlike the prior examples, most real world data have noise. Noise can occur in both the observations and the regressors depending on how data is acquired.

Typical types of noise include:

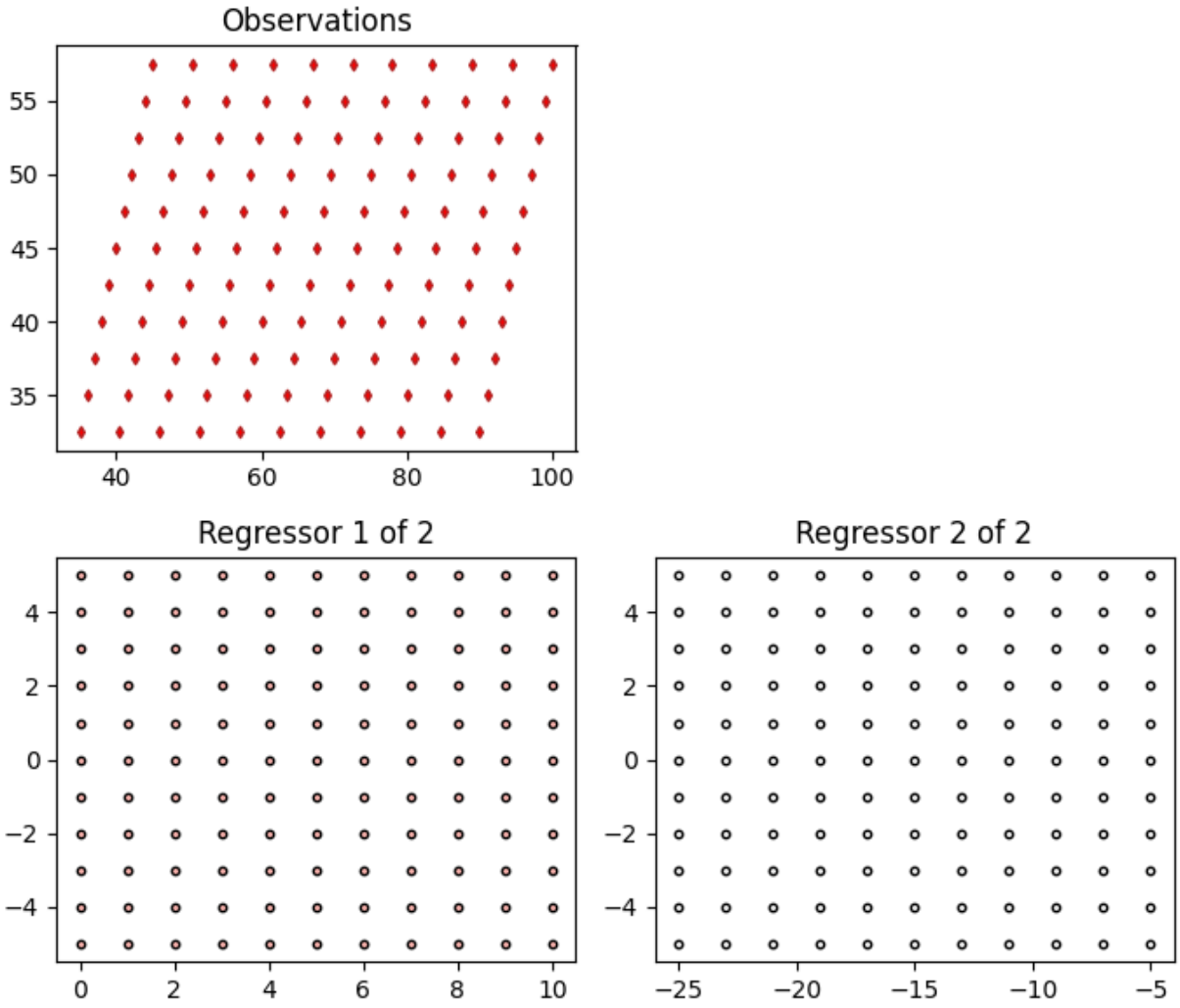
- Random measurement error, which can occur on both the observations and the regressors (for instance if both come from measurements)
- Gross outliers which can occur for many reasons

In this example, we'll explore adding gaussian distributed random noise to the data and seeing the impact this has on the estimated coefficients.

```
import numpy as np
import matplotlib.pyplot as plt
from regressioninc.linear import add_intercept, LeastSquares
from regressioninc.testing.complex import ComplexGrid, generate_linear_grid
from regressioninc.testing.complex import add_gaussian_noise, plot_complex

np.random.seed(42)
```

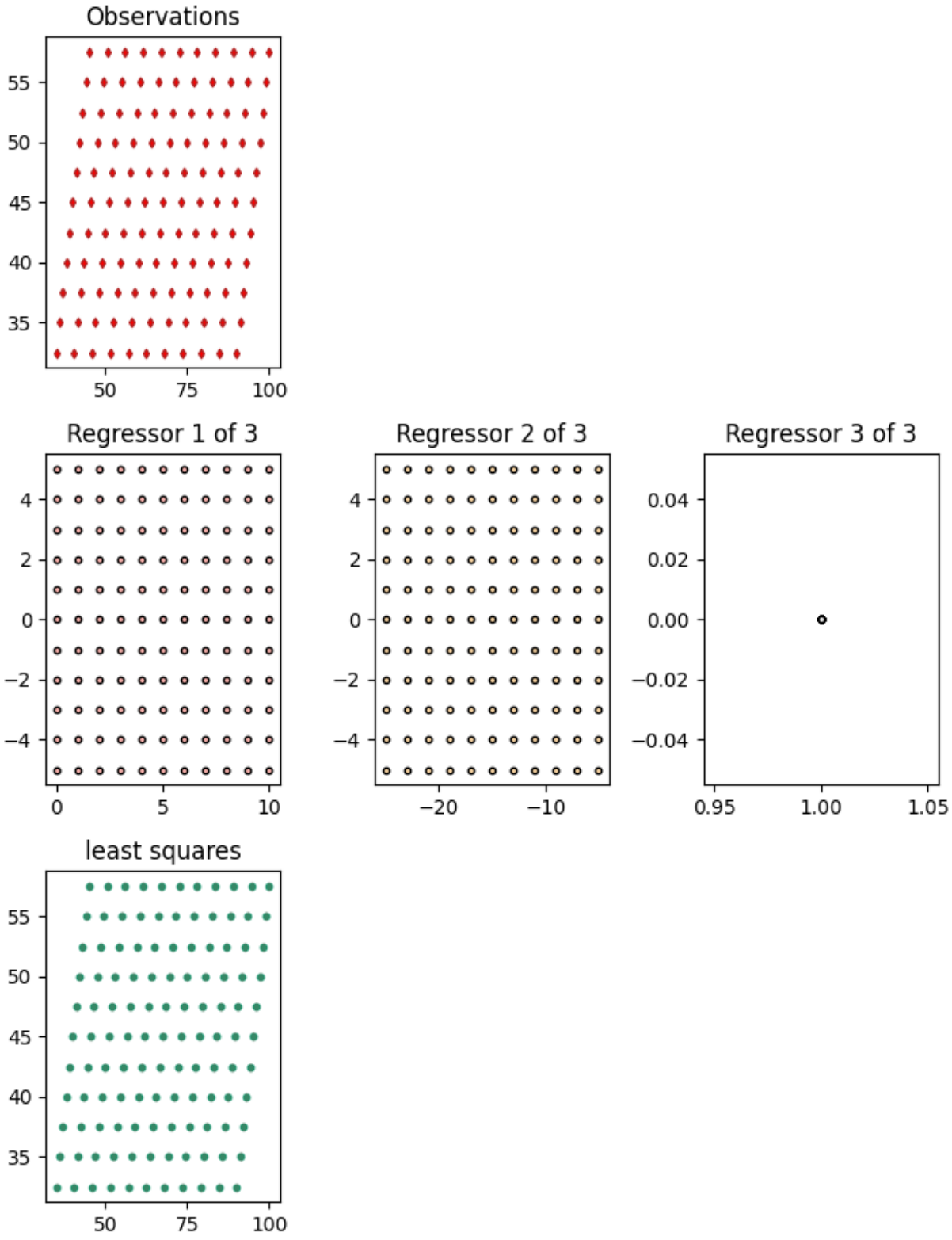
Let's setup another linear regression problem with complex values.

```
coef = np.array([0.5 + 2j, -3 - 1j])
grid_r1 = ComplexGrid(r1=0, r2=10, nr=11, i1=-5, i2=5, ni=11)
grid_r2 = ComplexGrid(r1=-25, r2=-5, nr=11, i1=-5, i2=5, ni=11)
X, y = generate_linear_grid(coef, [grid_r1, grid_r2], intercept=20 + 20j)

fig = plot_complex(X, y, {})
fig.set_size_inches(7, 6)
plt.tight_layout()
plt.show()
```



Estimating the coefficients using least squares gives the expected values.

```
X = add_intercept(X)
model = LeastSquares()
model.fit(X, y)
for idx, coef in enumerate(model.coef):
    print(f"Coefficient {idx}: {coef:.6f}")

fig = plot_complex(X, y, {"least squares": model})
fig.set_size_inches(7, 9)
plt.tight_layout()
plt.show()
```




```
Coefficient 0: 0.500000+2.000000j
Coefficient 1: -3.000000-1.000000j
```

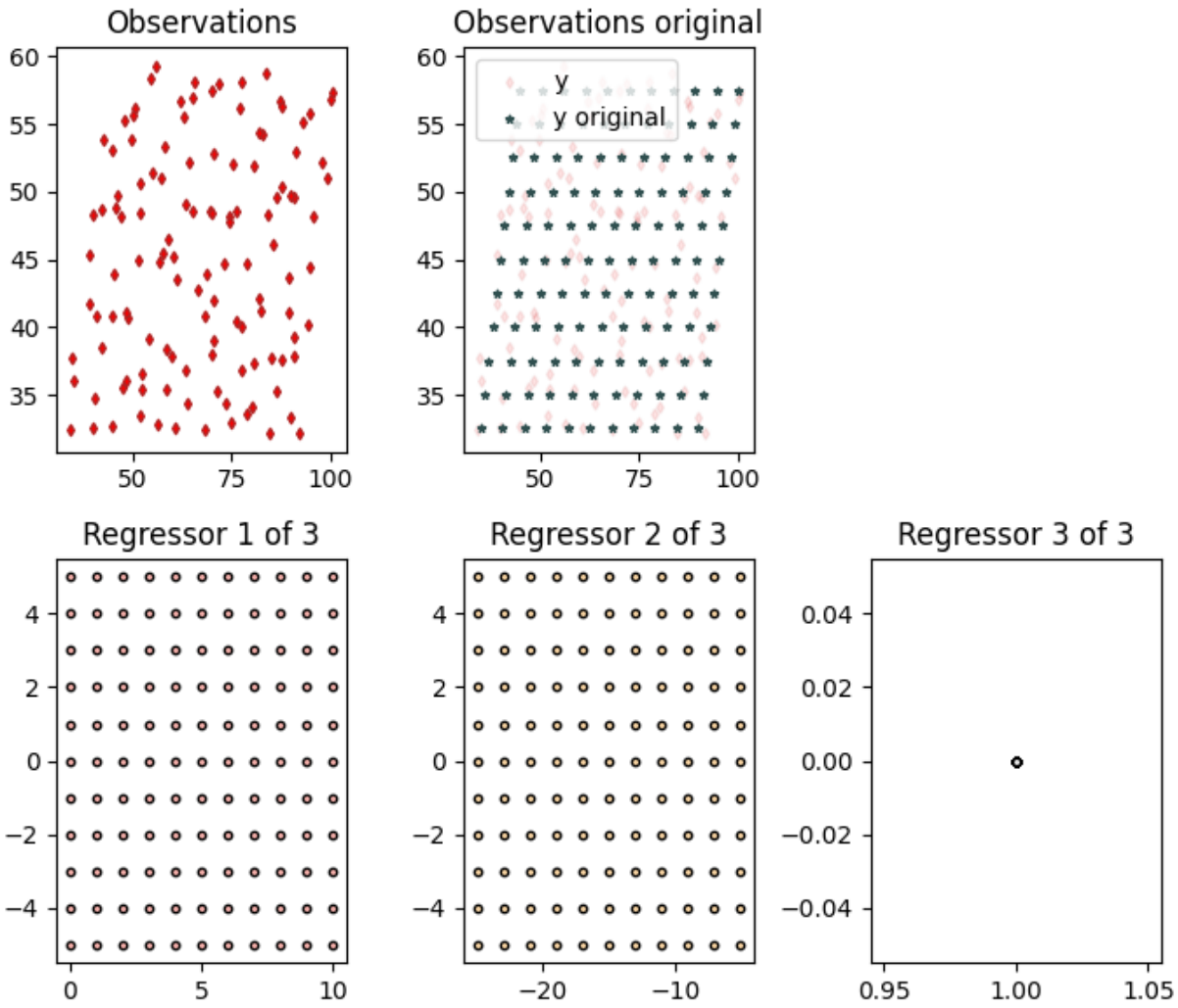
(continues on next page)

(continued from previous page)

```
Coefficient 2: 20.000000+20.000000j
```

Add some gaussian distributed random noise to the observations and let's see what they look like now.

```
y_noise = add_gaussian_noise(y, loc=(0, 0), scale=(3, 3))

fig = plot_complex(X, y_noise, {}, y_orig=y)
fig.set_size_inches(7, 6)
plt.tight_layout()
plt.show()
```



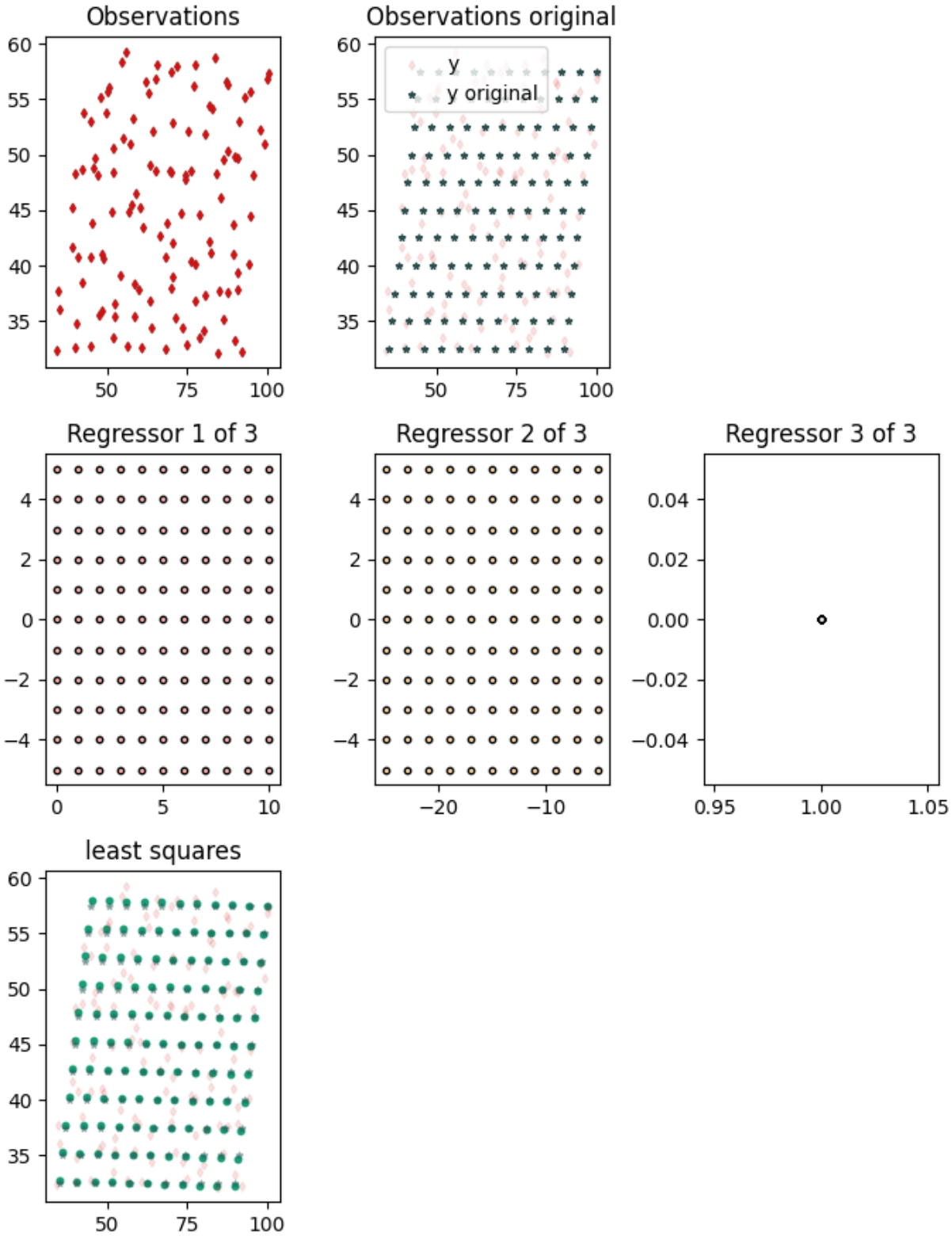
Now let's try and estimate the coefficients again but with the noisy observations. In this case, the coefficient estimates are slightly off the actual value due to the existence of the noise. Note that least squares is the maximum likelihood estimator for gaussian random noise. However, with other types of noise, there may be more effective regression methods.

```
model = LeastSquares()
model.fit(X, y_noise)
```

(continues on next page)

(continued from previous page)

```
for idx, coef in enumerate(model.coef):
    print(f"Coefficient {idx}: {coef:.6f}")

fig = plot_complex(X, y_noise, {"least squares": model}, y_orig=y)
fig.set_size_inches(7, 9)
plt.tight_layout()
plt.show()
```




```
Coefficient 0: 0.416040+1.964015j
Coefficient 1: -2.949279-0.955329j
```

(continues on next page)

(continued from previous page)

```
Coefficient 2: 21.072774+20.942814j
```

Total running time of the script: (0 minutes 2.637 seconds)

1.1.4 Outliers

Outliers in data can occur for a variety of reasons. Depending on the ways in which they appear, they can be worth investigating in more detail. However, the presence of outliers can skew estimation of the coefficients of interest.

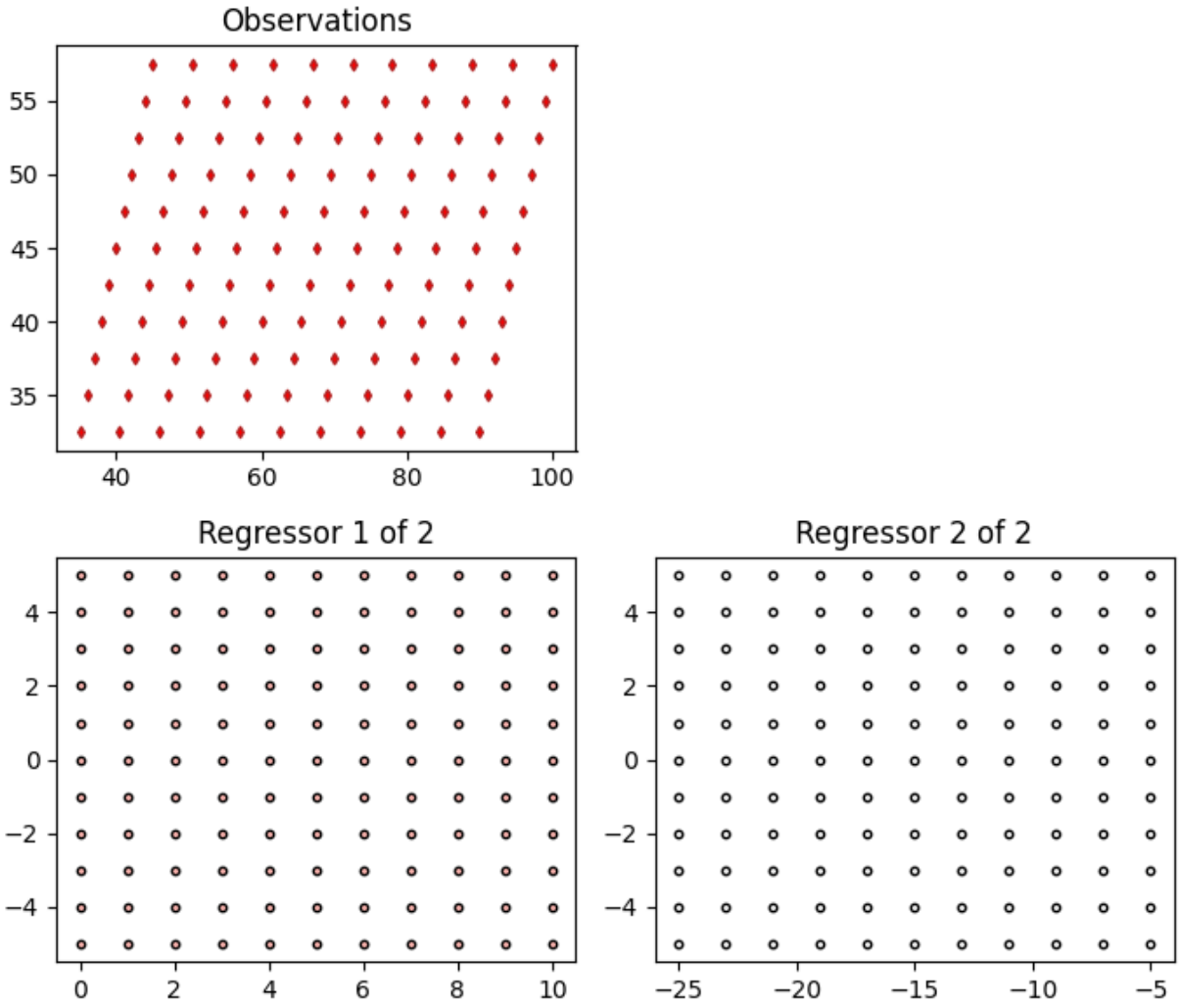
More information about outliers:

- https://en.wikipedia.org/wiki/Outlier

```
import numpy as np
import matplotlib.pyplot as plt
from regressioninc.linear import add_intercept, LeastSquares
from regressioninc.testing.complex import ComplexGrid, generate_linear_grid
from regressioninc.testing.complex import add_outliers, plot_complex

np.random.seed(42)
```

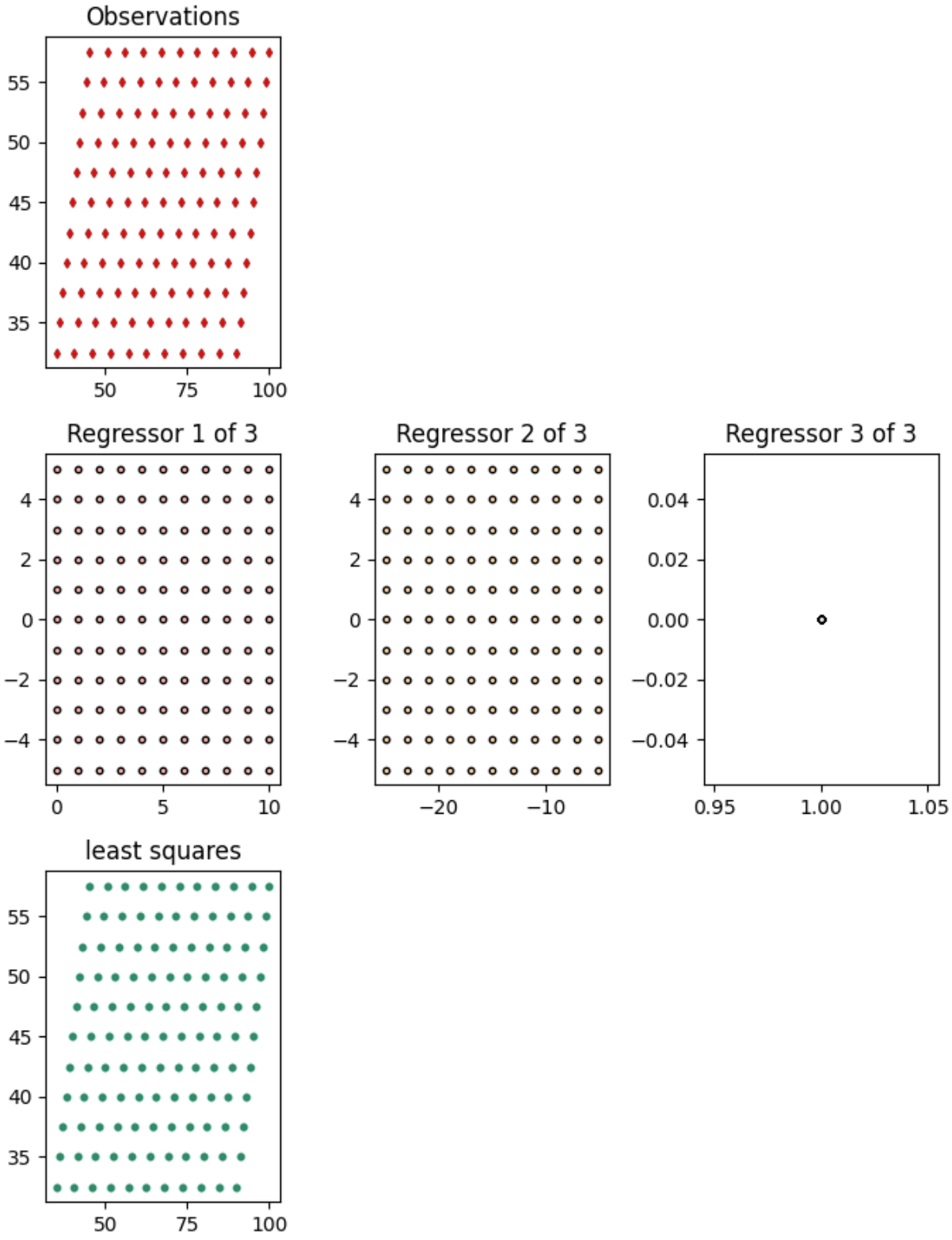
Let's setup another linear regression problem with complex values.

```
coef = np.array([0.5 + 2j, -3 - 1j])
grid_r1 = ComplexGrid(r1=0, r2=10, nr=11, i1=-5, i2=5, ni=11)
grid_r2 = ComplexGrid(r1=-25, r2=-5, nr=11, i1=-5, i2=5, ni=11)
X, y = generate_linear_grid(coef, [grid_r1, grid_r2], intercept=20 + 20j)

fig = plot_complex(X, y, {})
fig.set_size_inches(7, 6)
plt.tight_layout()
plt.show()
```



Estimating the coefficients using least squares gives the expected values.

```
X = add_intercept(X)
model = LeastSquares()
model.fit(X, y)
for idx, coef in enumerate(model.coef):
    print(f"Coefficient {idx}: {coef:.6f}")

fig = plot_complex(X, y, {"least squares": model})
fig.set_size_inches(7, 9)
plt.tight_layout()
plt.show()
```




```
Coefficient 0: 0.500000+2.000000j
Coefficient 1: -3.000000-1.000000j
```

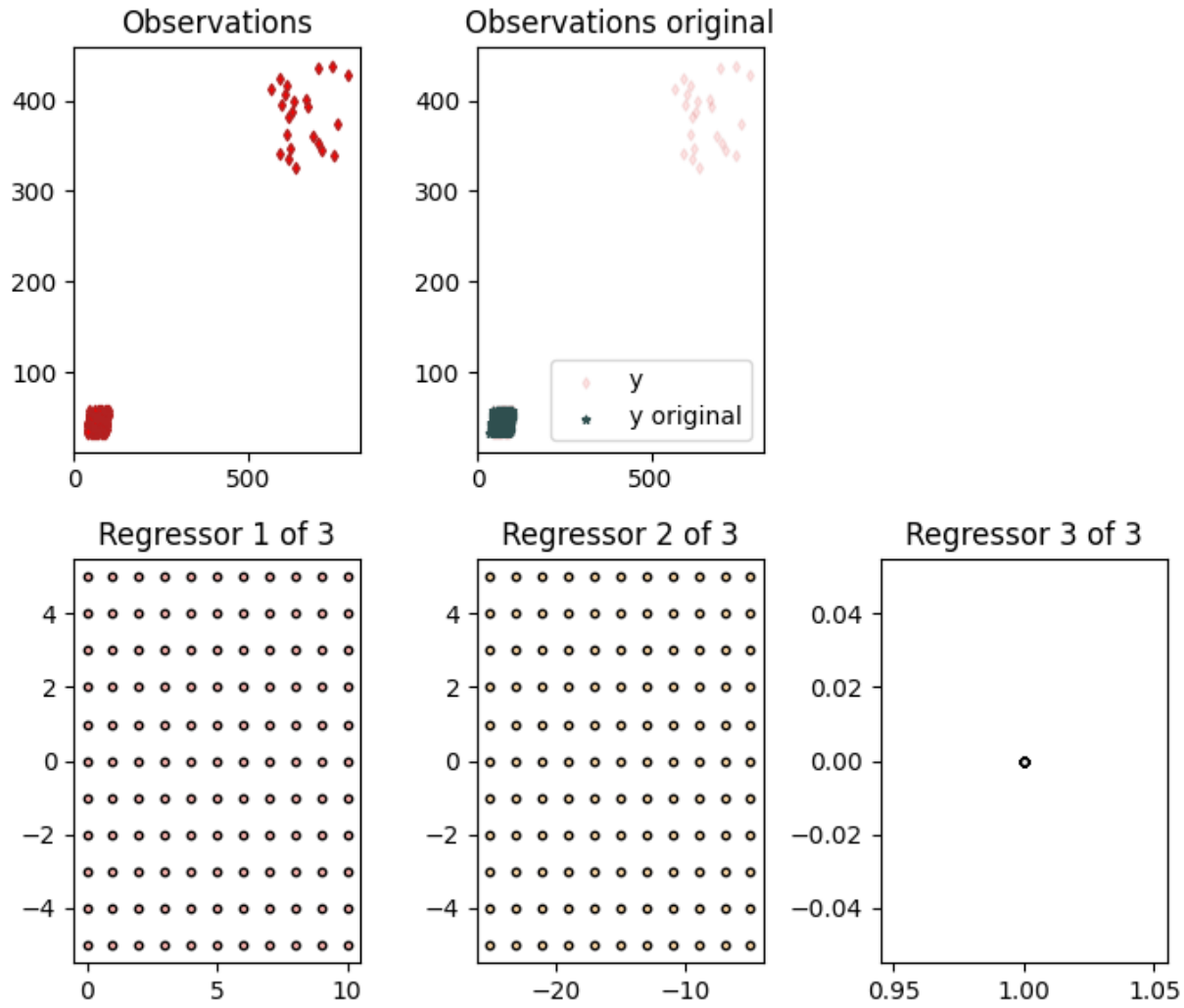
(continues on next page)

(continued from previous page)

```
Coefficient 2: 20.000000+20.000000j
```

Add some outliers to the observations.

```
y_noise = add_outliers(y, outlier_percent=20, mult_min=5, mult_max=7)

fig = plot_complex(X, y_noise, {}, y_orig=y)
fig.set_size_inches(7, 6)
plt.tight_layout()
plt.show()
```



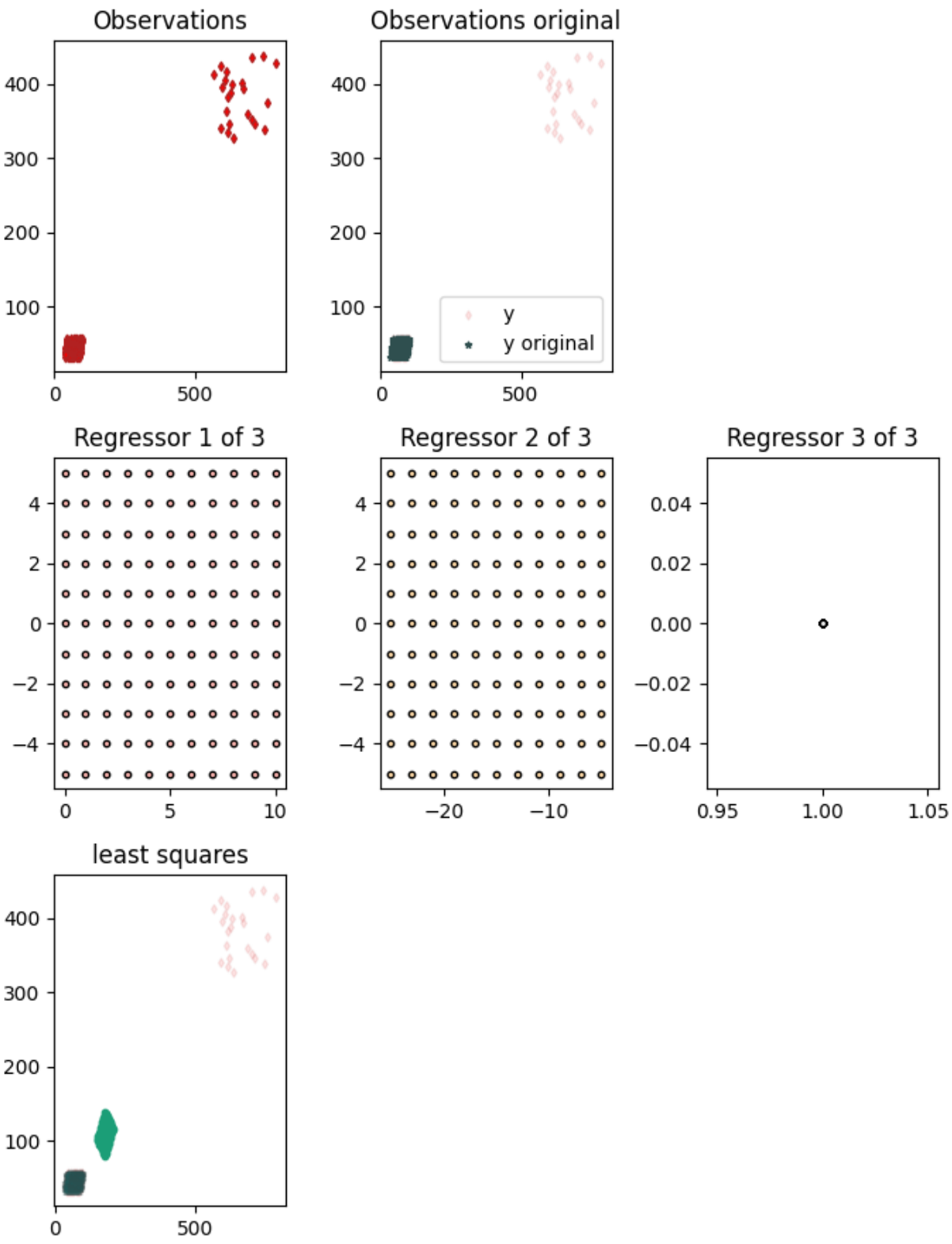
Now let's try and estimate the coefficients again but with the noisy observations. In this case, the coefficients estimates are slightly off the actual value due to the existence of the noise.

```
model = LeastSquares()
model.fit(X, y_noise)
for idx, coef in enumerate(model.coef):
    print(f"Coefficient {idx}: {coef:.6f}")
```

(continues on next page)

(continued from previous page)

```
fig = plot_complex(X, y_noise, {"least squares": model}, y_orig=y)
fig.set_size_inches(7, 9)
plt.tight_layout()
plt.show()
```




```
Coefficient 0: -4.696029+3.210866j
Coefficient 1: 1.053848-0.427605j
```

(continues on next page)

(continued from previous page)

```
Coefficient 2: 219.256159+86.717843j
```

Total running time of the script: (0 minutes 2.518 seconds)

1.1.5 Leverage points

Leverage points are large points in the regressors that can have a significant influence on coefficient estimates. High leverage points can be considered outliers with respect to independent variables or the regressors.

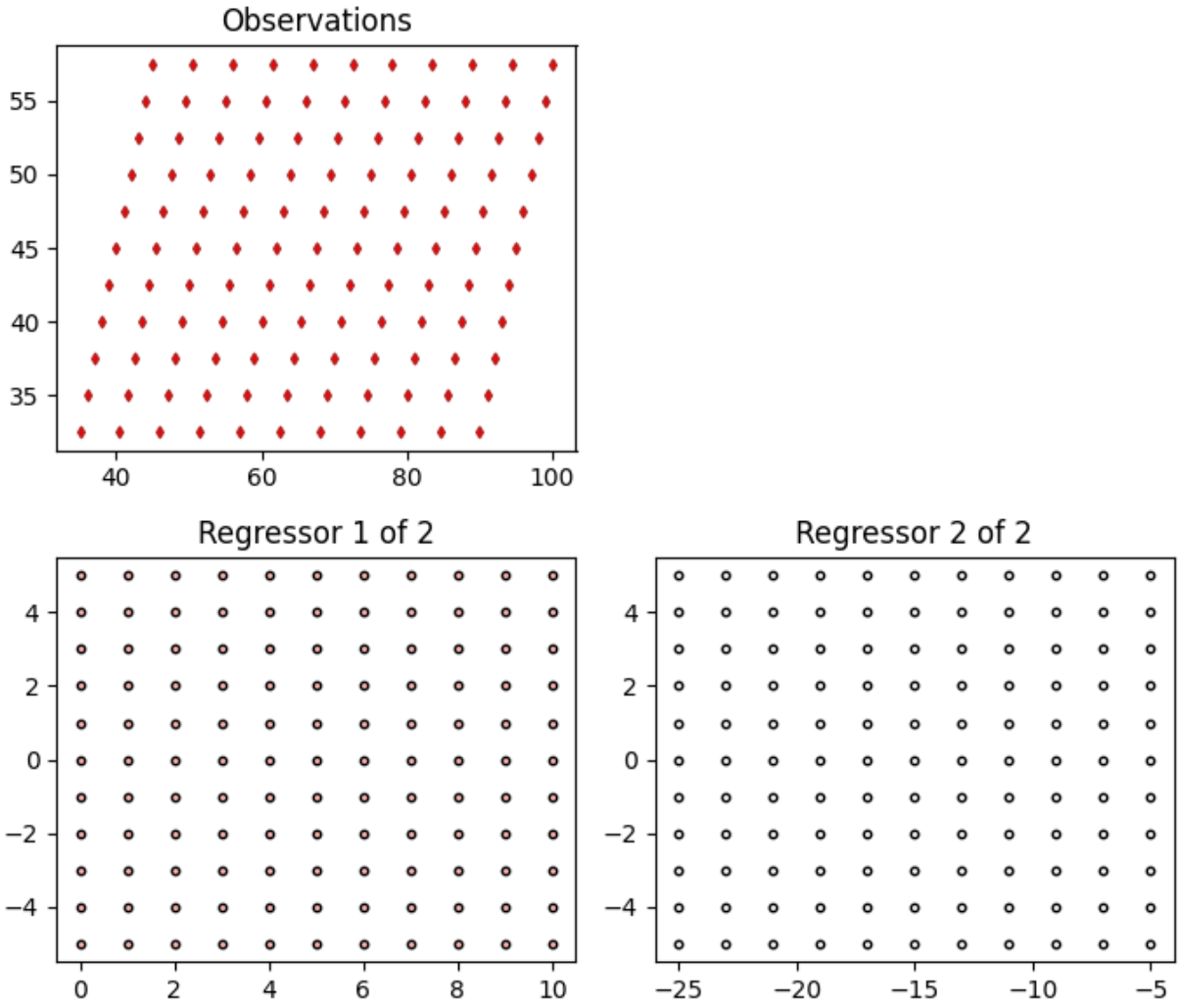
For more information on leverage points, see:

- https://en.wikipedia.org/wiki/Leverage_(statistics)

```
import numpy as np
import matplotlib.pyplot as plt
from regressioninc.linear import add_intercept, LeastSquares
from regressioninc.testing.complex import ComplexGrid, generate_linear_grid
from regressioninc.testing.complex import add_gaussian_noise, add_outliers, plot_complex

np.random.seed(42)
```

Let's setup another linear regression problem with complex values.

```
coef = np.array([0.5 + 2j, -3 - 1j])
grid_r1 = ComplexGrid(r1=0, r2=10, nr=11, i1=-5, i2=5, ni=11)
grid_r2 = ComplexGrid(r1=-25, r2=-5, nr=11, i1=-5, i2=5, ni=11)
X, y = generate_linear_grid(coef, [grid_r1, grid_r2], intercept=20 + 20j)

fig = plot_complex(X, y, {})
fig.set_size_inches(7, 6)
plt.tight_layout()
plt.show()
```



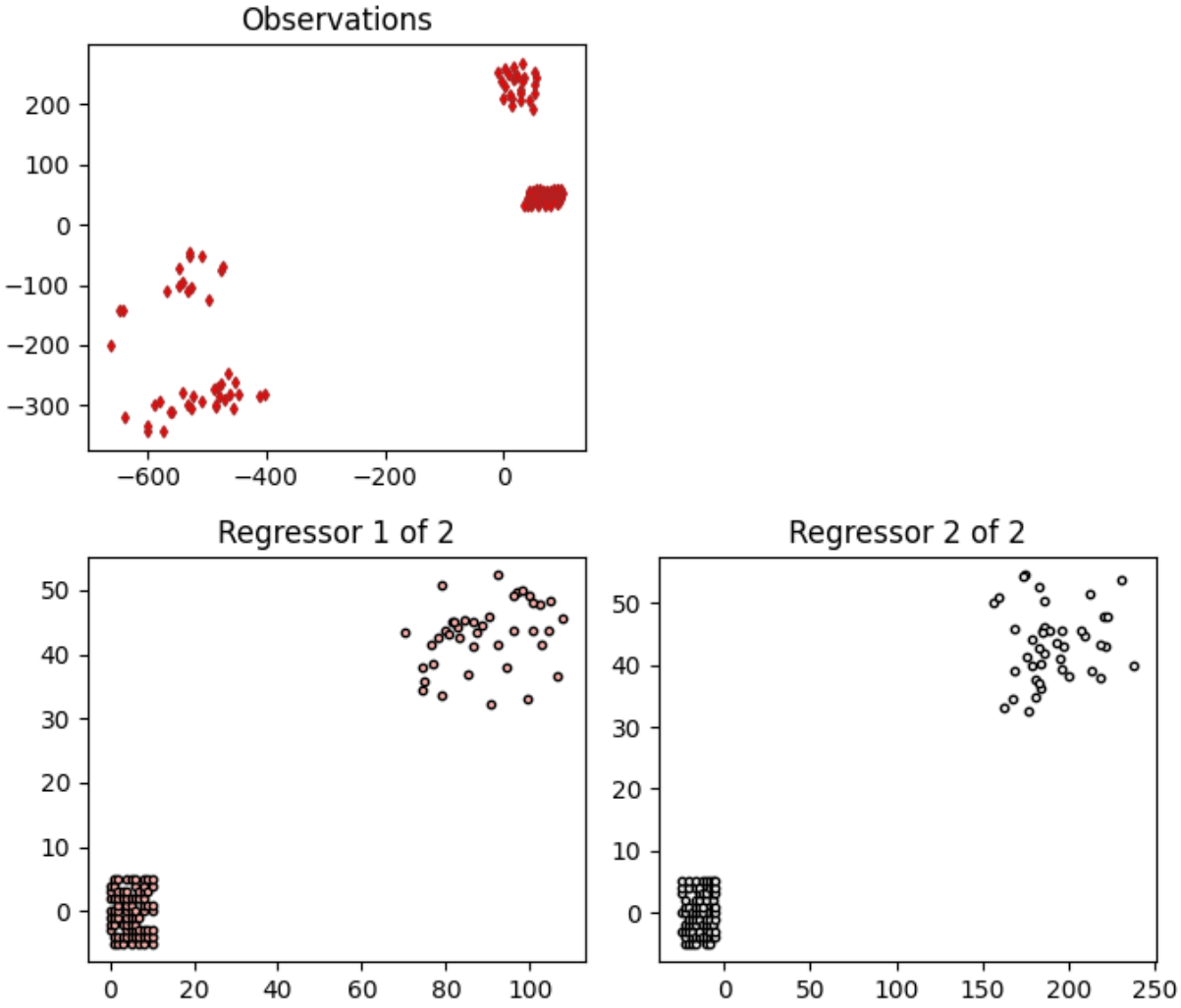
Add high leverage points to our regressors. Use different seeds for the two regressors to avoid getting the same outliers repeated twice.

```
seeds = [22, 36]
for ireg in range(X.shape[1]):
    np.random.seed(seeds[ireg])
    X[:, ireg] = add_outliers(
        X[:, ireg],
        outlier_percent=40,
        mult_min=7,
        mult_max=10,
        random_signs_real=False,
        random_signs_imag=False,
    )
np.random.seed(42)
intercept = 20 + 20j
y = np.matmul(X, coef) + intercept

fig = plot_complex(X, y, {})
```

(continues on next page)

(continued from previous page)

```
fig.set_size_inches(7, 6)
plt.tight_layout()
plt.show()
```



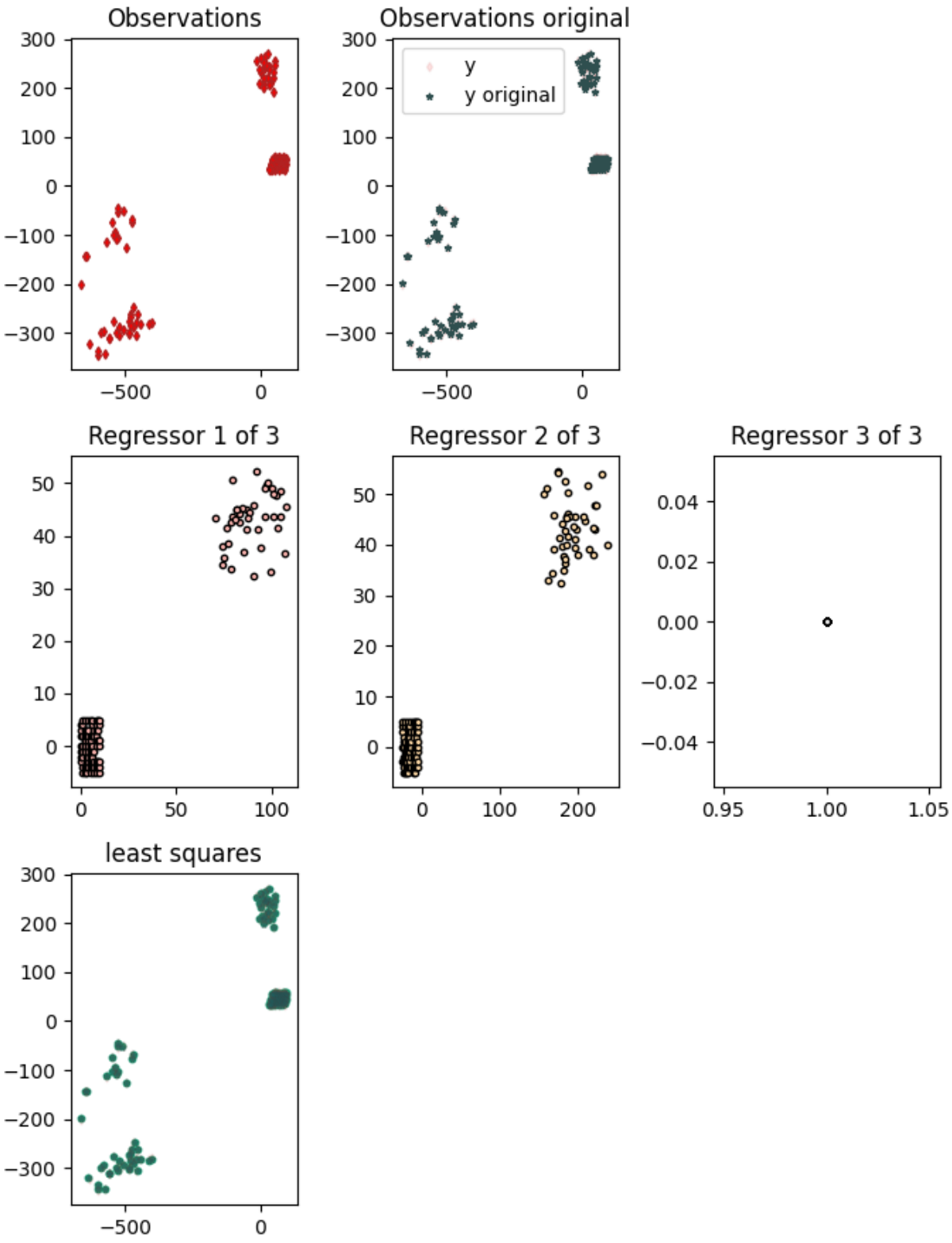
Solve the regression problem. Note that there is no noise on the observations so whilst there are high leverage points in the regressors, everything is consistent.

```
X = add_intercept(X)
model = LeastSquares()
model.fit(X, y)
for idx, coef in enumerate(model.coef):
    print(f"Coefficient {idx}: {coef:.6f}")
```

```
Coefficient 0: 0.500000+2.000000j
Coefficient 1: -3.000000-1.000000j
Coefficient 2: 20.000000+20.000000j
```

As a next stage, add some outliers to the data and see what happens.

```
# y_noise = add_outliers(
#     y,
#     outlier_percent=20,
#     mult_min=7,
#     mult_max=10,
# )
y_noise = add_gaussian_noise(y, loc=(0, 0), scale=(5, 5))
model = LeastSquares()
model.fit(X, y_noise)
for idx, coef in enumerate(model.coef):
    print(f"Coefficient {idx}: {coef:.6f}")

fig = plot_complex(X, y_noise, {"least squares": model}, y_orig=y)
fig.set_size_inches(7, 9)
plt.tight_layout()
plt.show()
```




```
Coefficient 0: 0.500994+2.000610j
Coefficient 1: -2.999265-1.000917j
```

(continues on next page)

(continued from previous page)

```
Coefficient 2: 19.781232+20.126984j
```

Total running time of the script: (0 minutes 1.730 seconds)

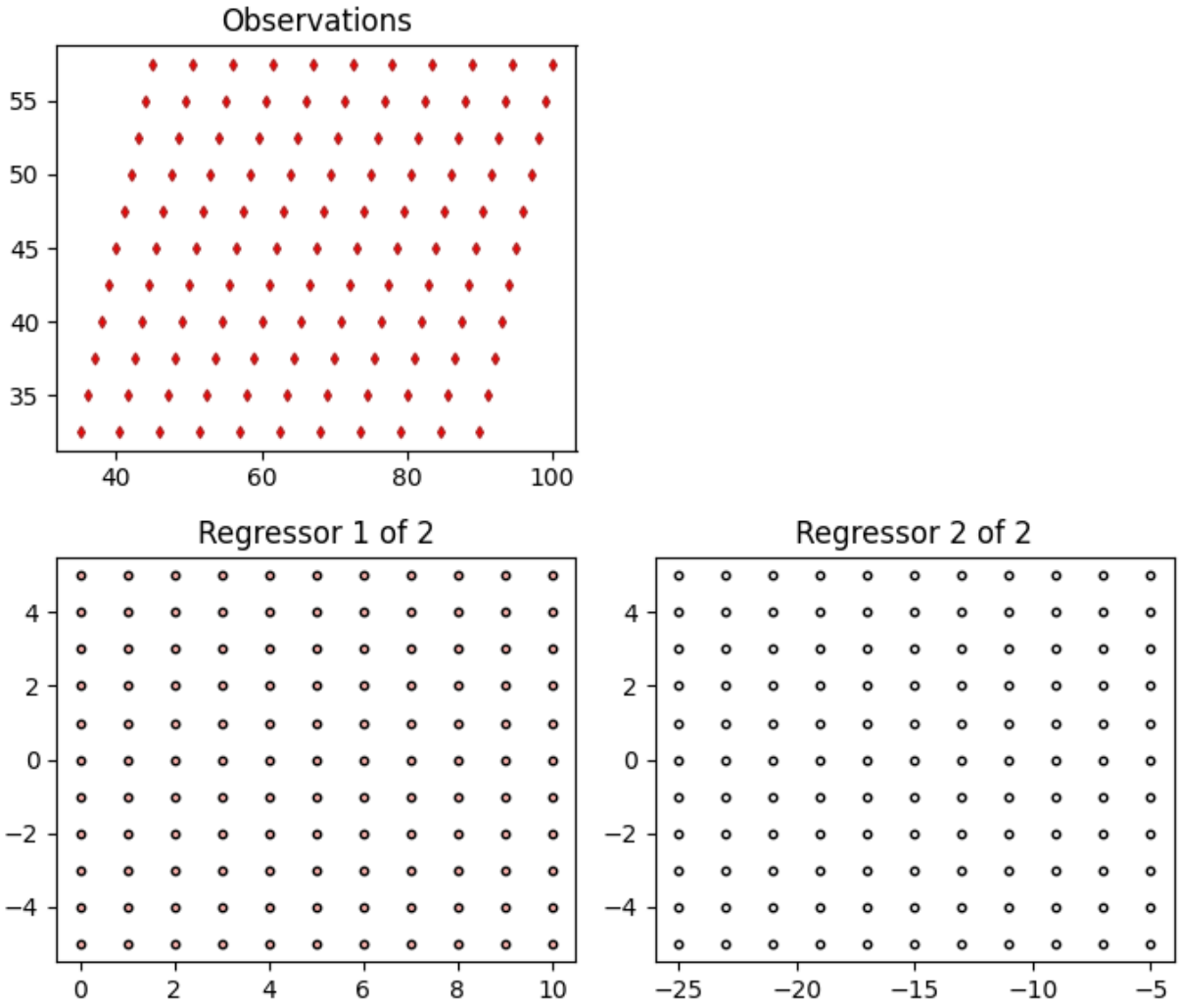
1.1.6 Robust regression

Robust regression

```
import numpy as np
import matplotlib.pyplot as plt
from regressioninc.linear import add_intercept, LeastSquares, M_estimate
from regressioninc.testing.complex import ComplexGrid, generate_linear_grid
from regressioninc.testing.complex import add_gaussian_noise, add_outliers, plot_complex

np.random.seed(42)
```

Let's setup another linear regression problem with complex values

```
coef = np.array([0.5 + 2j, -3 - 1j])
grid_r1 = ComplexGrid(r1=0, r2=10, nr=11, i1=-5, i2=5, ni=11)
grid_r2 = ComplexGrid(r1=-25, r2=-5, nr=11, i1=-5, i2=5, ni=11)
X, y = generate_linear_grid(coef, [grid_r1, grid_r2], intercept=20 + 20j)

fig = plot_complex(X, y, {})
fig.set_size_inches(7, 6)
plt.tight_layout()
plt.show()
```



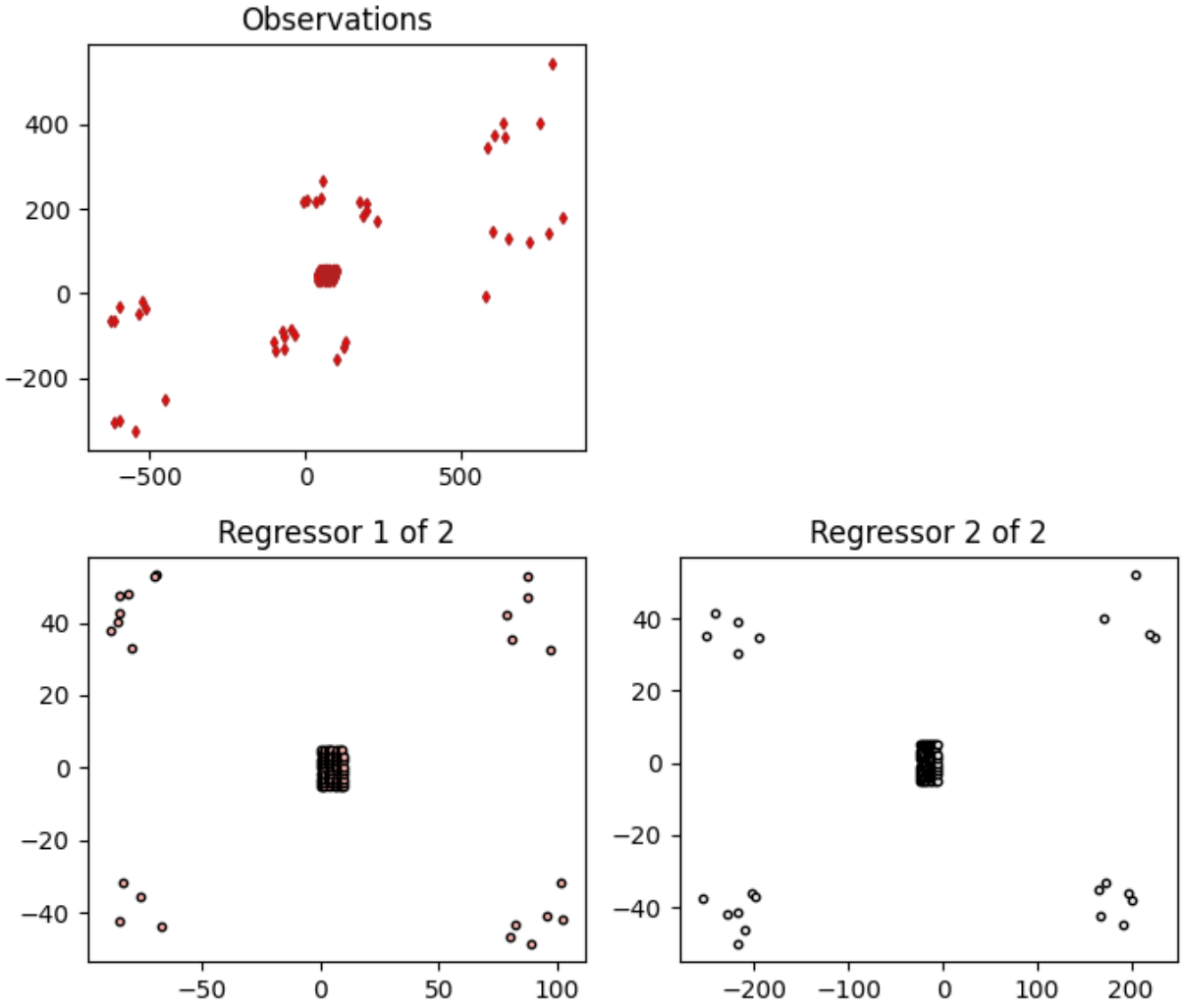
Add high leverage points to our regressors

```
seeds = [22, 36]
for ireg in range(X.shape[1]):
    np.random.seed(seeds[ireg])
    X[:, ireg] = add_outliers(
        X[:, ireg],
        outlier_percent=20,
        mult_min=7,
        mult_max=10,
        random_signs_real=True,
        random_signs_imag=True,
    )
np.random.seed(42)

intercept = 20 + 20j
y = np.matmul(X, coef) + intercept

fig = plot_complex(X, y, {})
```

(continues on next page)

(continued from previous page)

```
fig.set_size_inches(7, 6)
plt.tight_layout()
plt.show()
```



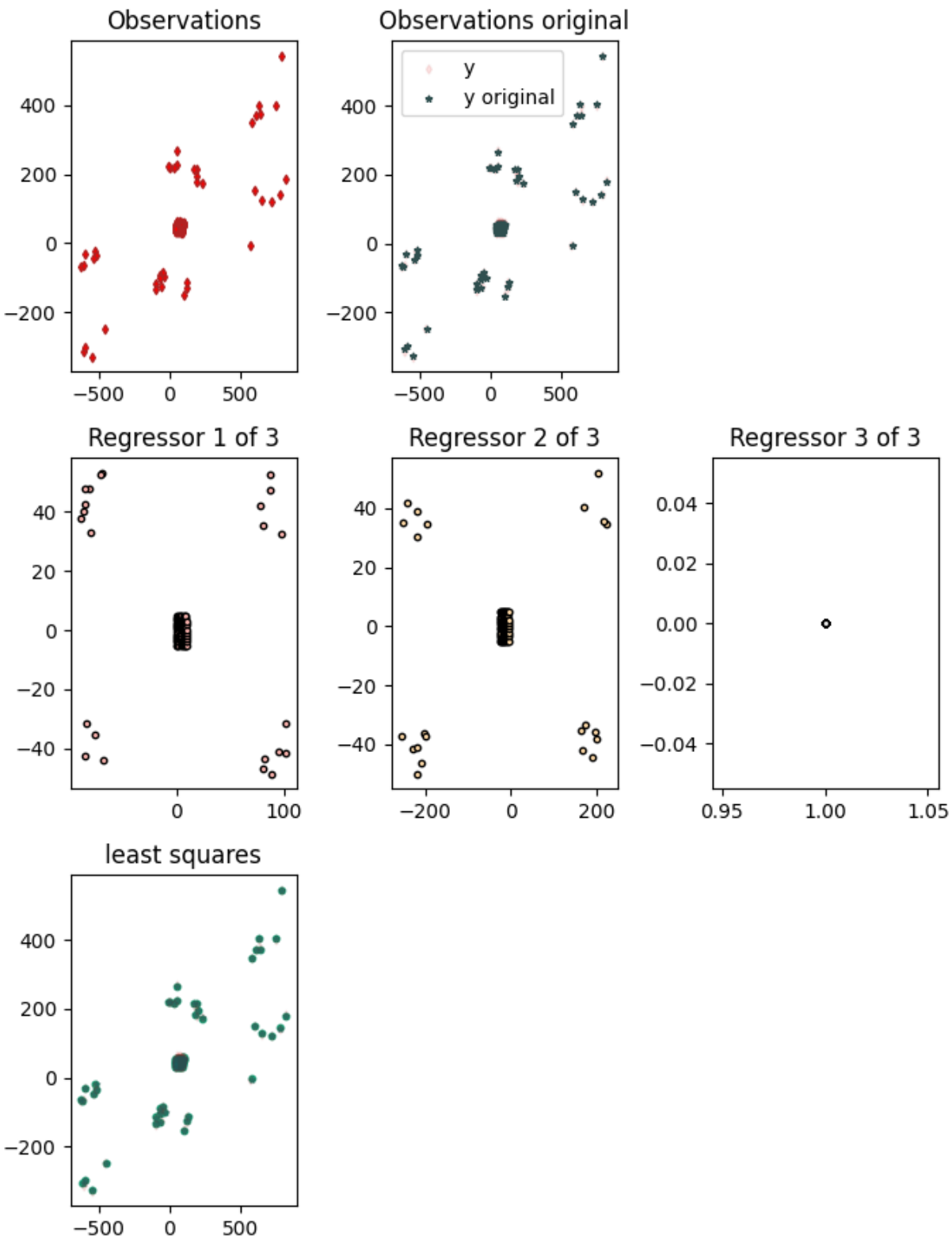
Solve

```
X = add_intercept(X)
model = LeastSquares()
model.fit(X, y)
for idx, coef in enumerate(model.coef):
    print(f"Coefficient {idx}: {coef:.6f}")
```

```
Coefficient 0: 0.500000+2.000000j
Coefficient 1: -3.000000-1.000000j
Coefficient 2: 20.000000+20.000000j
```

Add some outliers

```
y_noise = add_gaussian_noise(y, loc=(0, 0), scale=(21, 21))
model_ls = LeastSquares()
model_ls.fit(X, y_noise)
for idx, coef in enumerate(model_ls.coef):
    print(f"Coefficient {idx}: {coef:.6f}")

fig = plot_complex(X, y_noise, {"least squares": model_ls}, y_orig=y)
fig.set_size_inches(7, 9)
plt.tight_layout()
plt.show()
```




```
Coefficient 0: 0.504175+1.992051j
Coefficient 1: -3.002991-1.001990j
```

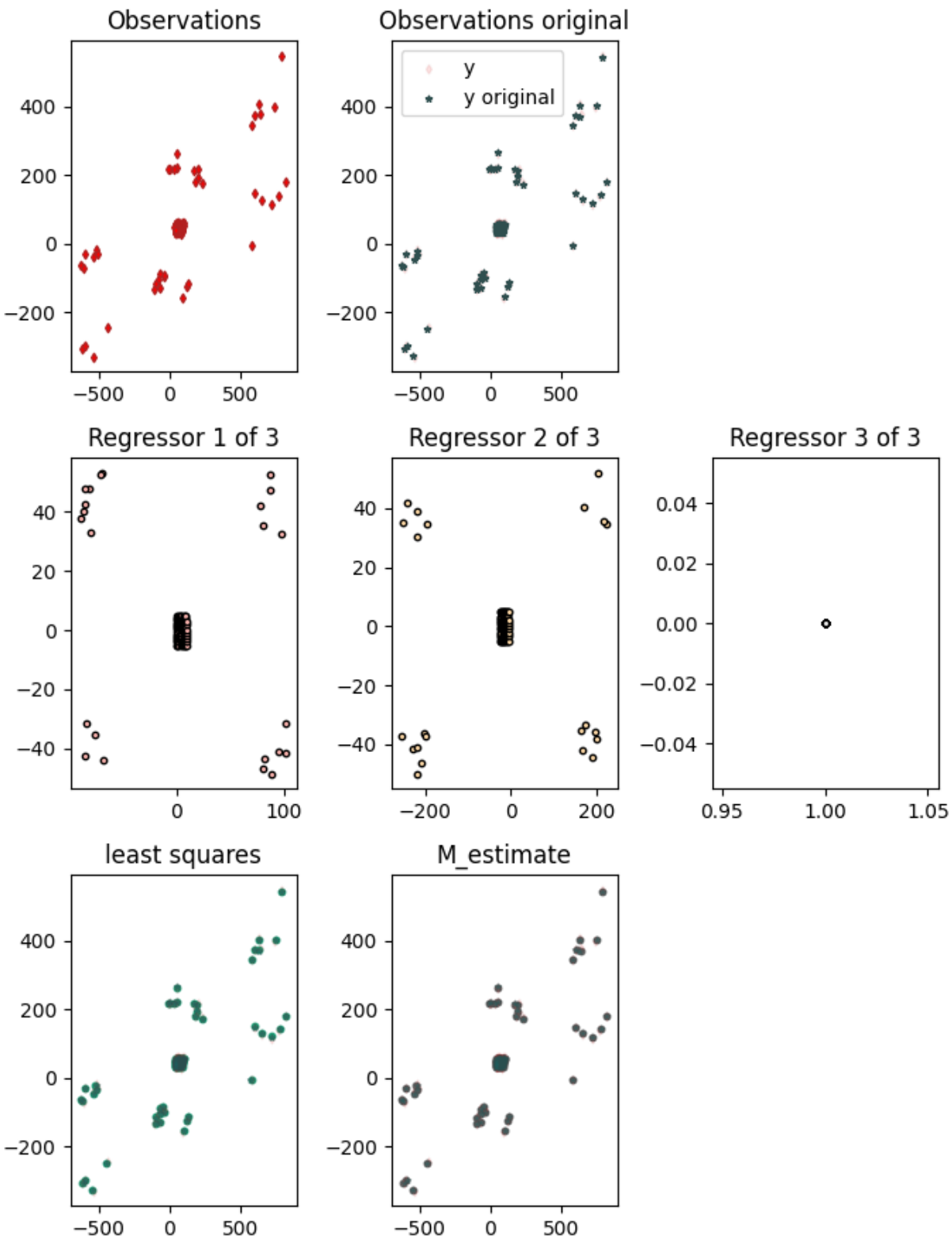
(continues on next page)

(continued from previous page)

```
Coefficient 2: 19.634988+20.233521j
```

Add some outliers

```
y_noise = add_gaussian_noise(y, loc=(0, 0), scale=(21, 21))
model_mest = M_estimate()
model_mest.fit(X, y_noise)
for idx, coef in enumerate(model_mest.coef):
    print(f"Coefficient {idx}: {coef:.6f}")

fig = plot_complex(
    X, y_noise, {"least squares": model_ls, "M_estimate": model_mest}, y_orig=y
)
fig.set_size_inches(7, 9)
plt.tight_layout()
plt.show()
```




```
Coefficient 0: 0.503769+1.990233j
Coefficient 1: -3.005947-1.000431j
```

(continues on next page)

(continued from previous page)

```
Coefficient 2: 20.021505+19.804239j
```

Total running time of the script: (0 minutes 2.543 seconds)

1.1.7 Complex to real

Complex-valued linear problems can be reformulated as real-valued problems by splitting out the real and imaginary parts of the equations.

$$a + ib = C_1(x + iy).$$

Remember that we are solving for $C_1 = (c_{1r} + c_{1i})$, which is also complex-valued, therefore,

$$a + ib = (c_{1r} + c_{1i})(x + iy).$$

This can be expanded out

$$\begin{aligned} a + ib &= (c_{1r} + ic_{1i})(x + iy) \\ &= c_{1r}x - c_{1i}y + ic_{1r}y + ic_{1i}x \\ &= (c_{1r}x - c_{1i}y) + i(c_{1r}y + ic_{1i}x), \end{aligned}$$

which gives,

$$\begin{aligned} a &= c_{1r}x - c_{1i}y \\ b &= c_{1r}y + ic_{1i}x. \end{aligned}$$

For the complex-valued problem, the aim is to solve for C_1. Making this real-valued means we are solving for c_{1r} and c_{1i}.

Moving from complex-valued to real-valued results in the following

- Doubling the number of observations as the real and imaginary parts of the observations are split up
- Doubling the number of regressors as we are now solving for the real and imaginary component of each regressor explicitly

```
import numpy as np
import matplotlib.pyplot as plt
from regressioninc.linear import add_intercept, LeastSquares, M_estimate
from regressioninc.testing.complex import ComplexGrid, generate_linear_grid
from regressioninc.testing.complex import add_gaussian_noise
from regressioninc.testing.complex import add_outliers, plot_complex
from regressioninc.linear import complex_to_glr, glr_coef_to_complex

np.random.seed(42)
```

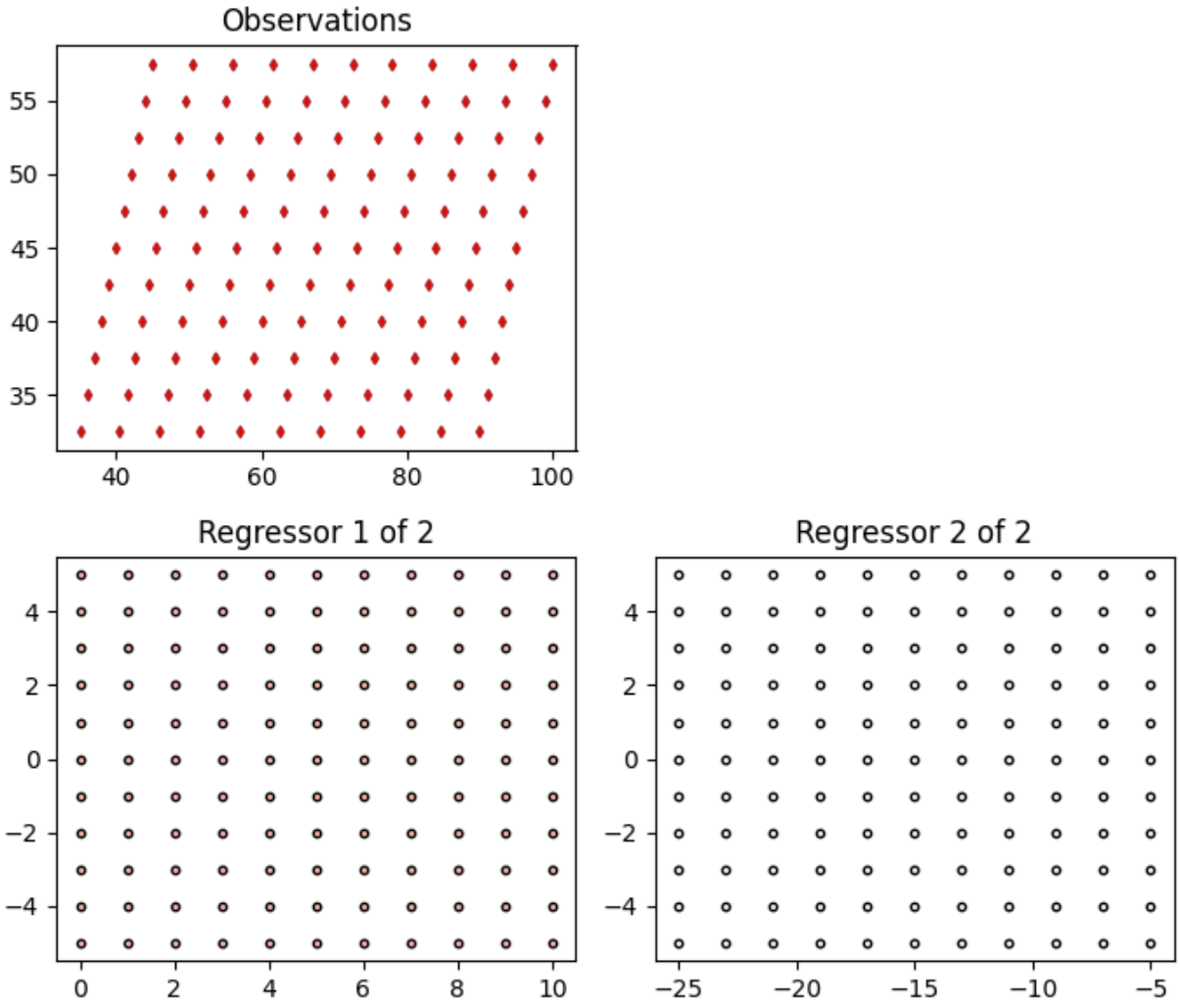
Let's setup another linear regression problem with complex values

```
coef = np.array([0.5 + 2j, -3 - 1j])
grid_r1 = ComplexGrid(r1=0, r2=10, nr=11, i1=-5, i2=5, ni=11)
grid_r2 = ComplexGrid(r1=-25, r2=-5, nr=11, i1=-5, i2=5, ni=11)
X, y = generate_linear_grid(coef, [grid_r1, grid_r2], intercept=20 + 20j)
```

(continues on next page)

(continued from previous page)

```
fig = plot_complex(X, y, {})
fig.set_size_inches(7, 6)
plt.tight_layout()
plt.show()
```



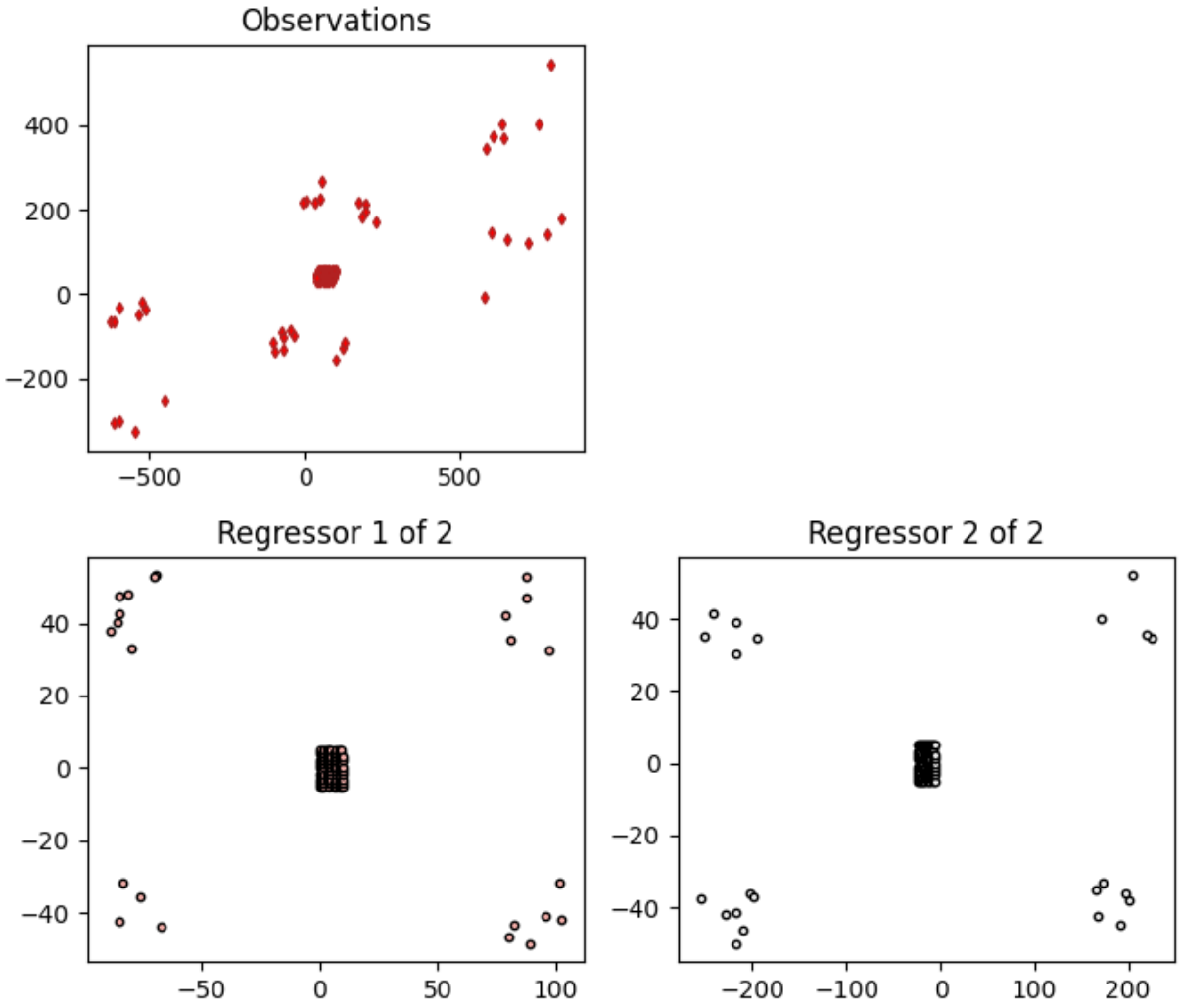
Add high leverage points to our regressors

```
seeds = [22, 36]
for ireg in range(X.shape[1]):
    np.random.seed(seeds[ireg])
    X[:, ireg] = add_outliers(
        X[:, ireg],
        outlier_percent=20,
        mult_min=7,
        mult_max=10,
        random_signs_real=True,
        random_signs_imag=True,
    )
```

(continues on next page)

(continued from previous page)

```
np.random.seed(42)

intercept = 20 + 20j
y = np.matmul(X, coef) + intercept

fig = plot_complex(X, y, {})
fig.set_size_inches(7, 6)
plt.tight_layout()
plt.show()
```



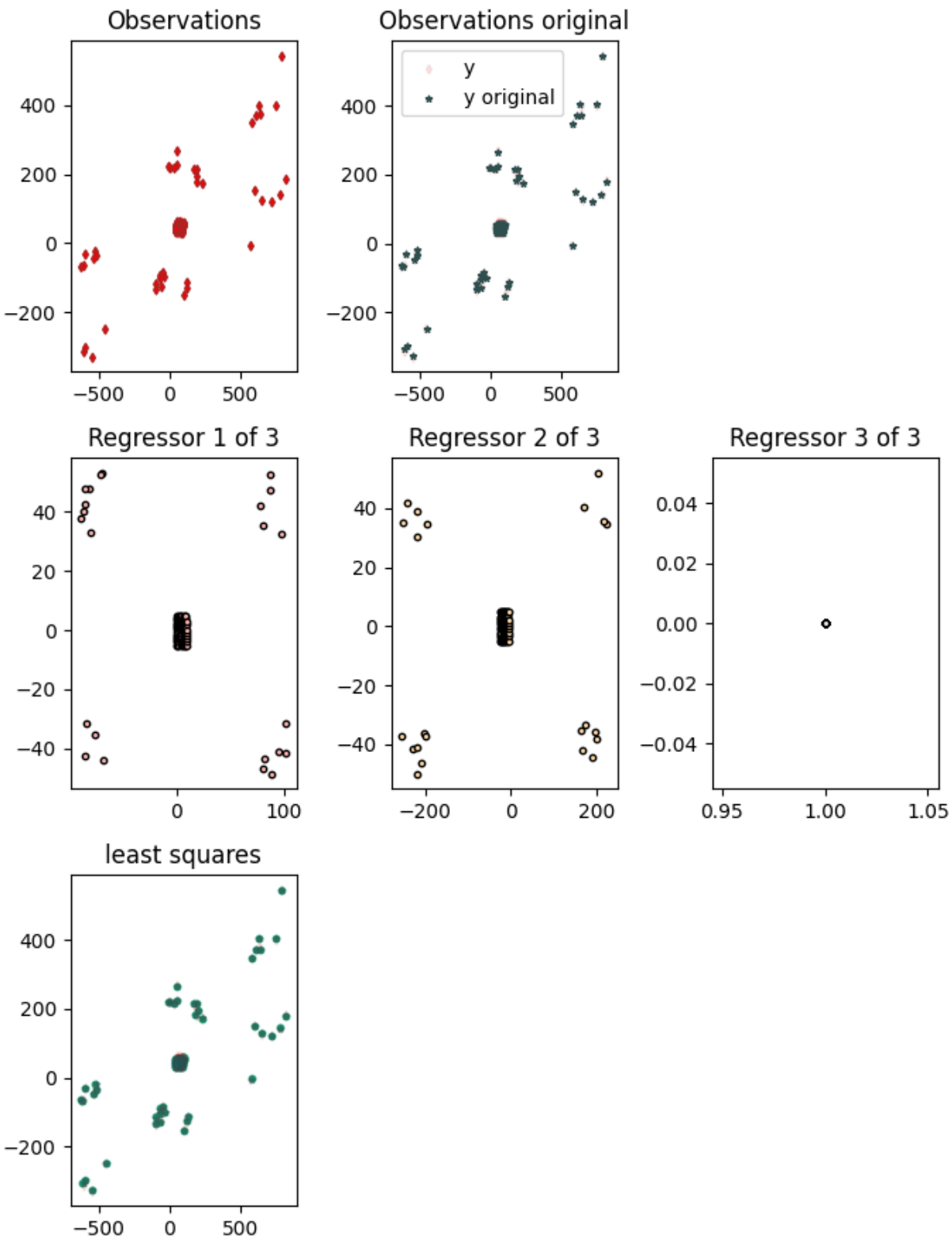
Solve

```
X = add_intercept(X)
model = LeastSquares()
model.fit(X, y)
for idx, coef in enumerate(model.coef):
    print(f"Coefficient {idx}: {coef:.6f}")
```

```
Coefficient 0: 0.500000+2.000000j
Coefficient 1: -3.000000-1.000000j
Coefficient 2: 20.000000+20.000000j
```

Add some outliers

```
y_noise = add_gaussian_noise(y, loc=(0, 0), scale=(21, 21))
model_ls = LeastSquares()
model_ls.fit(X, y_noise)
for idx, coef in enumerate(model_ls.coef):
    print(f"Coefficient {idx}: {coef:.6f}")

fig = plot_complex(X, y_noise, {"least squares": model_ls}, y_orig=y)
fig.set_size_inches(7, 9)
plt.tight_layout()
plt.show()
```




```
Coefficient 0: 0.504175+1.992051j
Coefficient 1: -3.002991-1.001990j
```

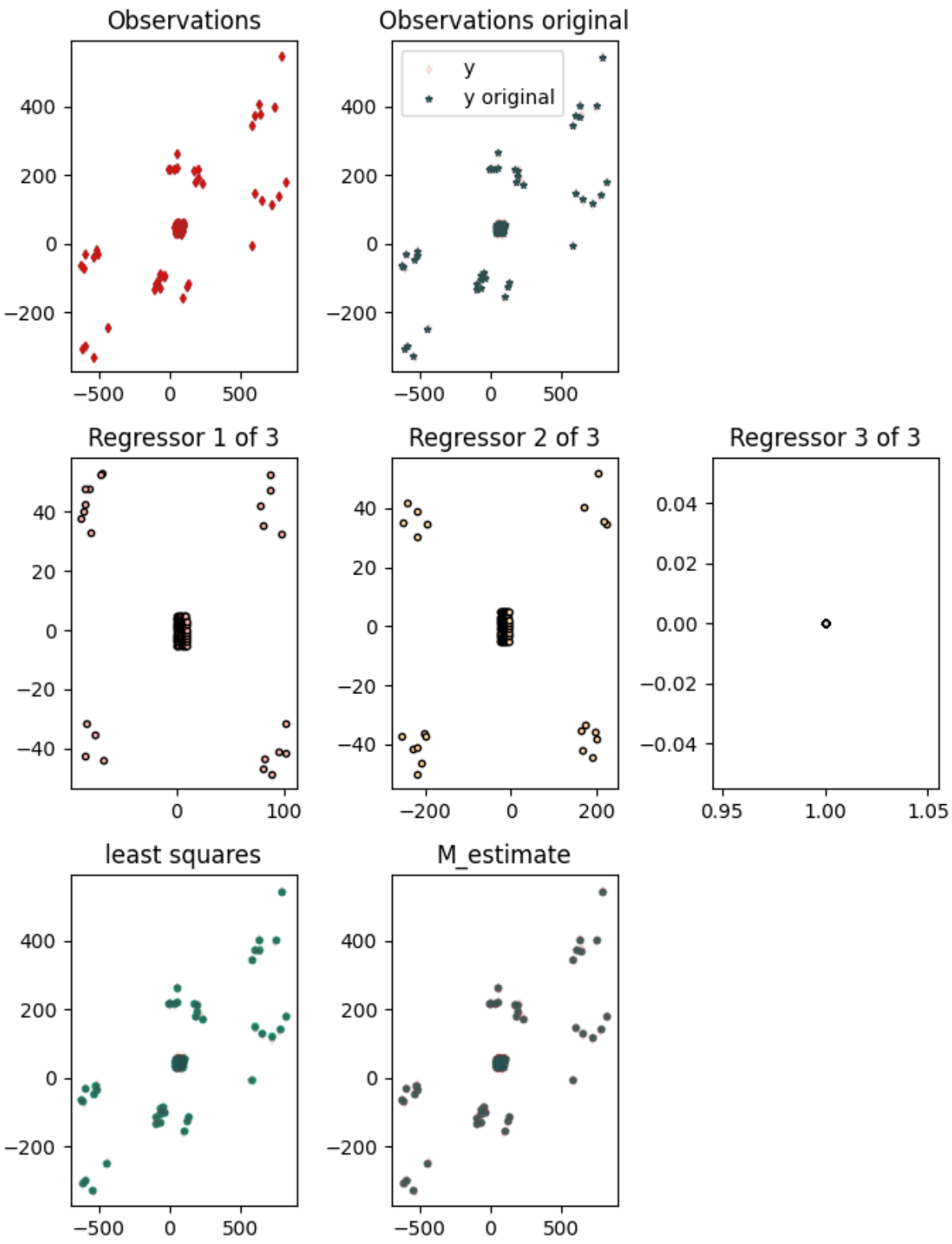
(continues on next page)

(continued from previous page)

```
Coefficient 2: 19.634988+20.233521j
```

Add some outliers

```
y_noise = add_gaussian_noise(y, loc=(0, 0), scale=(21, 21))
model_mest = M_estimate()
model_mest.fit(X, y_noise)
for idx, coef in enumerate(model_mest.coef):
    print(f"Coefficient {idx}: {coef:.6f}")

fig = plot_complex(
    X, y_noise, {"least squares": model_ls, "M_estimate": model_mest}, y_orig=y
)
fig.set_size_inches(7, 9)
plt.tight_layout()
plt.show()
```




```
Coefficient 0: 0.503769+1.990233j
Coefficient 1: -3.005947-1.000431j
```

(continues on next page)

(continued from previous page)

```
Coefficient 2: 20.021505+19.804239j
```

Try running as a real-valued problem

```
X_real, y_real = complex_to_glr(X, y_noise)
model_ls = LeastSquares()
model_ls.fit(X_real, y_real)
coef = glr_coef_to_complex(model_ls.coef)
for idx, coef in enumerate(coef):
    print(f"Coefficient {idx}: {coef:.6f}")
```

```
Coefficient 0: 0.509492+1.998838j
Coefficient 1: -2.999218-0.998313j
Coefficient 2: 20.273556+19.907992j
```

Try running using real-valued M_estimates

```
model_mest = M_estimate()
model_mest.fit(X_real, y_real)
coef = glr_coef_to_complex(model_mest.coef)
for idx, coef in enumerate(coef):
    print(f"Coefficient {idx}: {coef:.6f}")
```

```
Coefficient 0: 0.505285+1.993242j
Coefficient 1: -3.005300-0.999968j
Coefficient 2: 20.040712+19.764842j
```

Total running time of the script: (0 minutes 2.535 seconds)

1.2 regressioninc package

1.2.1 Subpackages

regressioninc.testing package

Submodules

regressioninc.testing.complex module

Functions for generating and visualising complex-valued testing data

pydantic model regressioninc.testing.complex.**ComplexGrid**

Bases: BaseModel

A class to generate a grid for complex data

```
{
   "title": "ComplexGrid",
   "description": "A class to generate a grid for complex data",
```

(continues on next page)

(continued from previous page)

```
   "type": "object",
   "properties": {
      "r1": {
         "title": "R1",
         "type": "number"
      },
      "r2": {
         "title": "R2",
         "type": "number"
      },
      "nr": {
         "title": "Nr",
         "type": "integer"
      },
      "i1": {
         "title": "I1",
         "type": "number"
      },
      "i2": {
         "title": "I2",
         "type": "number"
      },
      "ni": {
         "title": "Ni",
         "type": "integer"
      }
   },
   "required": [
      "r1",
      "r2",
      "nr",
      "i1",
      "i2",
      "ni"
   ]
}
```

field r1: float [Required]

Starting real pt

field r2: float [Required]

End real pt

field nr: int [Required]

Number of real pts

field i1: float [Required]

Start imaginary pt

field i2: float [Required]

End imaginary pt

field ni: int [Required]

Number of imaginary pts

property n_pts: int

Get the number of points grid

Returns
Number of points in grid

Return type
int

Examples

```
>>> from regressioninc.testing.complex import ComplexGrid
>>> grid = ComplexGrid(r1=0, r2=5, nr=6, i1=0, i2=5, ni=6)
>>> grid.n_pts
36
```

grid() → ndarray

Get the grid points as an 2-D array

Returns
The grid points as a 2-D array

Return type
np.ndarray

Examples

```
>>> from regressioninc.testing.complex import ComplexGrid
>>> grid = ComplexGrid(r1=-1, r2=1, nr=3, i1=-1, i2=1, ni=3)
>>> grid.grid()
array([[-1.-1.j, -1.+0.j, -1.+1.j],
       [ 0.-1.j,  0.+0.j,  0.+1.j],
       [ 1.-1.j,  1.+0.j,  1.+1.j]])
```

flat_grid() → ndarray

Get the grid as a flat array

Returns
The grid of points as a 1-D flattened array

Return type
np.ndarray

Examples

```
>>> from regressioninc.testing.complex import ComplexGrid
>>> grid = ComplexGrid(r1=-1, r2=1, nr=3, i1=-1, i2=1, ni=3)
>>> grid.grid()
array([[-1.-1.j, -1.+0.j, -1.+1.j],
       [ 0.-1.j,  0.+0.j,  0.+1.j],
       [ 1.-1.j,  1.+0.j,  1.+1.j]])
```

regressioninc.testing.complex.**generate_linear_grid**(*coef: ndarray*, *grids: list[regressioninc.testing.complex.ComplexGrid]*, *intercept: complex = 0*) → tuple[numpy.ndarray, numpy.ndarray]

Generate complex regression data from coefficients and grids of regressors

Parameters

- **coef** (*np.ndarray*) – The coefficients for the regressors
- **grids** (*list[ComplexGrid]*) – The grid of points for each regressor
- **intercept** (*complex, optional*) – The intercept, by default 0

Returns
X, y for the regression problem

Return type
tuple[np.ndarray, np.ndarray]

Raises

- **ValueError** – If the number of grids provided does not equal the number of coefficients
- **ValueError** – If the grids have different numbers of points

regressioninc.testing.complex.**generate_linear_random**(*coef: ndarray*, *n_samples: int*, *intercept: complex = 0*, *min_rand=-10*, *max_rand=10*)

Produce complex data for testing without any noise

regressioninc.testing.complex.**add_gaussian_noise**(*data: ndarray*, *loc: tuple[float] | None = None*, *scale: tuple[float] | None = None*) → ndarray

Add Gaussian noise to data

regressioninc.testing.complex.**add_outliers**(*y: ndarray*, *outlier_percent: float = 5*, *mult_min=3*, *mult_max=5*, *random_signs_real: bool = False*, *random_signs_imag: bool = False*) → ndarray

Add outliers to a complex-valued 1-D observations array

regressioninc.testing.complex.**plot_observations**(*y: ndarray*, *size: int = 10*, *alpha: float = 1.0*) → None

Plot observation data

regressioninc.testing.complex.**plot_observations_original**(*y_orig: ndarray*, *size: int = 10*, *alpha: float = 1.0*) → None

Plot original observations, meant for data without noise

regressioninc.testing.complex.**plot_regressor**(*reg: ndarray*, *color: ndarray*, *size: int = 10*, *alpha: float = 1.0*) → None

Plot regression data

regressioninc.testing.complex.**plot_estimate**(*est: ndarray*, *color: ndarray*, *size: int = 10*, *alpha: float = 1.0*, *label: str = 'estimate'*) → None

Plot model estimates

regressioninc.testing.complex.**plot_complex**(*X*, *y*, *models: dict[str, regressioninc.base.Regressor]*, *y_orig: ndarray | None = None*, *size_obs: int = 10*, *size_reg: int = 10*, *size_est: int = 10*)

Plot the complex data

regressioninc.testing.real module

Functions for generating and visualising real-valued testing data

regressioninc.testing.real.**generate_linear**(*coef: ndarray*, *n_samples: int*, *intercept: float = 0*)

Generate real-valued linear testing data

regressioninc.testing.real.**add_gaussian_noise**(*data: ndarray*, *loc: float = 0*, *scale: float = 3*) → ndarray

Add gaussian noise to an array

Parameters

- **data** (*np.ndarray*) – The data to add the noise to
- **loc** (*float, optional*) – The location (mean) of the gaussian, by default 0. Usually this should# be left as 0.
- **scale** (*float, optional*) – The scale (or standard deviation) of the noise, by default 3

Returns

Data with noise added

Return type

np.ndarray

regressioninc.testing.real.**add_outliers**(*y: ndarray*, *outlier_percent: float = 5*, *outlier_mult=3*) → ndarray

Add outliers to a 1-D observations array

regressioninc.testing.real.**linear_real_with_leverage**()

regressioninc.testing.real.**linear_real_with_outliers_and_leverage**()

regressioninc.testing.real.**plot_1d**(*X: ndarray*, *y: ndarray*, *coefs: Dict[str, Tuple[ndarray, float]] | None = None*) → Figure

regressioninc.testing.real.**get_X_plane**(*X: ndarray*) → ndarray

regressioninc.testing.real.**plot_2d**(*X: ndarray*, *y: ndarray*, *coefs: Dict[str, Tuple[ndarray, float]] | None = None*) → Figure

Module contents

1.2.2 Submodules

regressioninc.base module

Base class for regression in C

The base class implements some basic functions and the abstract classes for regressors.

regressioninc.base.**htranspose**(*arr: ndarray*) → ndarray

Hermitian transpose of an array (transpose and complex conjugation)

Parameters

arr (*np.ndarray*) – Input array

Returns
Hermitian transpose

Return type
np.ndarray

regressioninc.base.**sum_square_residuals**(*X: ndarray*, *y: ndarray*, *coef: ndarray*) → float

Calculate sum of square residuals

Parameters

- **X** (*np.ndarray*) – The regressors
- **y** (*np.ndarray*) – The observations
- **coef** (*np.ndarray*) – The estimated coefficients

Returns
The sum of square residuals

Return type
float

pydantic model regressioninc.base.**Regressor**

Bases: BaseModel

Base class for any regressor


```
{
   "title": "Regressor",
   "description": "Base class for any regressor",
   "type": "object",
   "properties": {}
}
```


regressioninc.linear module

Regressors for estimating parameters for linear problems

These are built on numpy as numpy.linalg.lstsq does allow and support complex-valued inputs. However, it does not output residuals for complex-valued variables, therefore these are calculated out explicitly.

regressioninc.linear.**add_intercept**(*X: ndarray*) → ndarray

Add an intercept to the regressors

Parameters
X (*np.ndarray*) – The regressors

Returns
The regressors with an extra column of ones to represent to allow solving for the intercept term

Return type
np.ndarray

Examples

```
>>> import numpy as np
>>> from regressioninc.linear import add_intercept
>>> X = np.array([[2,3], [4,5]])
>>> X
array([[2, 3],
       [4, 5]])
```

Now add the intercept

```
>>> X = add_intercept(X)
>>> X
array([[2, 3, 1],
       [4, 5, 1]])
```

regressioninc.linear.**complex_to_glr**(*X: ndarray*, *y: ndarray*) → tuple[numpy.ndarray, numpy.ndarray]

Convert complex-valued linear problem to a real-valued generalised linear regression

Parameters

- **X** (*np.ndarray*) – The regressors
- **y** (*np.ndarray*) – The observations

Returns

X, y converted to real-valued generalised linear regression

Return type

tuple[np.ndarray, np.ndarray]

Examples

```
>>> from regressioninc.linear import add_intercept, complex_to_glr
>>> X = np.array([3 + 4j, 2 + 8j, 6 + 4j]).reshape(3,1)
>>> X = add_intercept(X)
>>> y = np.array([-2 - 4j, -1 - 3j, 5 + 6j])
>>> X_glr, y_glr = complex_to_glr(X, y)
>>> X_glr
array([[ 3., -4.,  1., -0.],
       [ 4.,  3.,  0.,  1.],
       [ 2., -8.,  1., -0.],
       [ 8.,  2.,  0.,  1.],
       [ 6., -4.,  1., -0.],
       [ 4.,  6.,  0.,  1.]])
>>> y_glr
array([-2., -4., -1., -3.,  5.,  6.])
```

regressioninc.linear.**glr_coef_to_complex**(*coef: ndarray*) → ndarray

Transform coefficients from real to complex-values for complex-valued problems that were posed

Parameters

coef (*np.ndarray*) – Coefficients array

Returns

The complex-valued coefficients

Return type
np.ndarray

Examples

Let's generate a complex-valued linear problem, pose it as a linear problem and then convert the returned coefficients back to complex

Generate the linear problem and add an intercept column to the regressors

```
>>> import numpy as np
>>> np.set_printoptions(precision=3, suppress=True)
>>> from regressioninc.testing.complex import ComplexGrid, generate_linear_grid
>>> from regressioninc.linear import add_intercept, LeastSquares
>>> from regressioninc.linear import complex_to_glr, glr_coef_to_complex
>>> coef = np.array([3 + 2j])
>>> grid = ComplexGrid(r1=-1, r2=1, nr=3, i1=4, i2=6, ni=3)
>>> X, y = generate_linear_grid(coef, [grid], intercept=10)
>>> X = add_intercept(X)
```

Convert the complex-valued problem to a real-valued problem

```
>>> X_glr, y_glr = complex_to_glr(X, y)
```

Solve the real-valued linear problem

```
>>> model = LeastSquares()
>>> model.fit(X_glr, y_glr)
LeastSquares...
```

Look at the real-valued coefficients

```
>>> model.coef
array([ 3.,  2., 10.,  0.])
```

Convert the coefficients back to the complex domain

```
>>> glr_coef_to_complex(model.coef)
array([ 3.+2.j, 10.+0.j])
```

pydantic model regressioninc.linear.**LinearRegressor**

Bases: *Regressor*

Base class for LinearRegression

```
{
   "title": "LinearRegressor",
   "description": "Base class for LinearRegression",
   "type": "object",
   "properties": {
      "coef": {
         "title": "Coef"
      },
      "residuals": {
```

(continues on next page)

(continued from previous page)

```
            "title": "Residuals"
        },
        "rank": {
            "title": "Rank",
            "type": "integer"
        },
        "singular": {
            "title": "Singular"
        }
    }
}
```

field coef: ndarray | None = None

The coefficients

field residuals: ndarray | None = None

The square residuals

field rank: int | None = None

The rank of the predictors

field singular: ndarray | None = None

The singular values of the predictors

fit(*X: ndarray*, *y: ndarray*) → ndarray

Fit the linear model

predict(*X: ndarray*) → ndarray

Predict using the linear model

Parameters
X (*np.ndarray*) – The predictors

Returns
The predicted observations

Return type
np.ndarray

Raises
ValueError – If the coefficients are None. This is most likely the case if the model has not been fitted first.

fit_predict(*X: ndarray*, *y: ndarray*) → ndarray

Fit and predict on predictors X and observations y

Parameters

- **X** (*np.ndarray*) – The predictors to use for the fit and predict
- **y** (*np.ndarray*) – The observations for the fit

Returns
The predicted observations

Return type
np.ndarray

pydantic model regressioninc.linear.**LeastSquares**

Bases: *LinearRegressor*

Standard linear regression


```
{
   "title": "LeastSquares",
   "description": "Standard linear regression",
   "type": "object",
   "properties": {
      "coef": {
         "title": "Coef"
      },
      "residuals": {
         "title": "Residuals"
      },
      "rank": {
         "title": "Rank",
         "type": "integer"
      },
      "singular": {
         "title": "Singular"
      }
   }
}
```


fit(*X: ndarray*, *y: ndarray*) → ndarray

Fit the linear problem using least squares regression

Parameters

- **X** (*np.ndarray*) – The predictors
- **y** (*np.ndarray*) – The observations

Returns
The coefficients

Return type
np.ndarray

pydantic model regressioninc.linear.**WeightedLeastSquares**

Bases: *LinearRegressor*

Transform X and y using the weights to perform a weighted least squares

$$\sqrt{weights}y = \sqrt{weights}Xcoef,$$

is equivalent to,

$$X^H weightsy = X^H weightsXcoef,$$

where X^H is the hermitian transpose of X.

In this method, both the observations y and the predictors X are multipled by the square root of the weights and then returned.

```
{
   "title": "WeightedLeastSquares",
   "description": "Transform X and y using the weights to perform a weighted least␣
↪squares\n\n.. math::\n    \\sqrt{weights} y = \\sqrt{weights} X coef ,\n\nis␣
↪equivalent to,\n\n.. math::\n    X^H weights y = X^H weights X coef ,\n\nwhere␣
↪:math:`X^H` is the hermitian transpose of X.\n\nIn this method, both the␣
↪observations y and the predictors X are multipled\nby the square root of the␣
↪weights and then returned.",
   "type": "object",
   "properties": {
      "coef": {
         "title": "Coef"
      },
      "residuals": {
         "title": "Residuals"
      },
      "rank": {
         "title": "Rank",
         "type": "integer"
      },
      "singular": {
         "title": "Singular"
      }
   }
}
```


fit(*X: ndarray*, *y: ndarray*, *weights: ndarray*) → ndarray

Apply weights to observations and predictors and find the coefficients of the linear model

Parameters

- **X** (*np.ndarray*) – The predictors
- **y** (*np.ndarray*) – The observations
- **weights** (*np.ndarray*) – The weights to apply to the samples

Returns

The coefficients for the model

Return type

np.ndarray

Raises

ValueError – If the size of weights does not match the size of the observations y

pydantic model regressioninc.linear.**M_estimate**

Bases: *LinearRegressor*


```
{
   "title": "M_estimate",
   "description": "Base class for LinearRegression",
   "type": "object",
   "properties": {
      "coef": {
         "title": "Coef"
```


(continues on next page)

(continued from previous page)


```
      },
      "residuals": {
         "title": "Residuals"
      },
      "rank": {
         "title": "Rank",
         "type": "integer"
      },
      "singular": {
         "title": "Singular"
      },
      "n_iter": {
         "title": "N Iter",
         "default": 50,
         "type": "integer"
      },
      "early_stopping": {
         "title": "Early Stopping",
         "default": 1e-05,
         "type": "number"
      }
   }
}
```


field n_iter: int = 50

field early_stopping: float = 1e-05

fit(*X: ndarray*, *y: ndarray*)

Fit the linear model

pydantic model regressioninc.linear.**MM_estimate**

Bases: *LinearRegressor*


```
{
   "title": "MM_estimate",
   "description": "Base class for LinearRegression",
   "type": "object",
   "properties": {
      "coef": {
         "title": "Coef"
      },
      "residuals": {
         "title": "Residuals"
      },
      "rank": {
         "title": "Rank",
         "type": "integer"
      },
      "singular": {
         "title": "Singular"
      }
```


(continues on next page)

(continued from previous page)

```
    }
}
```

fit(*X: ndarray*, *y: ndarray*) → ndarray

Two stage M estimate

pydantic model regressioninc.linear.**ComplexAsGLR**

Bases: *LinearRegressor*

```
{
   "title": "ComplexAsGLR",
   "description": "Base class for LinearRegression",
   "type": "object",
   "properties": {
      "coef": {
         "title": "Coef"
      },
      "residuals": {
         "title": "Residuals"
      },
      "rank": {
         "title": "Rank",
         "type": "integer"
      },
      "singular": {
         "title": "Singular"
      }
   }
}
```

fit(*X*, *y*, *model*)

Fit the linear model

1.2.3 Module contents

A package for doing regression in the complex domain

1.3 References

Complex regression versus making it a real problem https://stats.stackexchange.com/questions/66088/analysis-with-complex-data-anything-different

Random gaussian noise for complex data https://stackoverflow.com/questions/55700338/how-to-generate-a-complex-gaussian-white-noise-signal-in-pythonor-numpy-scipy

Robust models in statsmodels - particularly for the robust norms https://www.statsmodels.org/stable/examples/notebooks/generated/robust_models_1.html

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

r

INDEX

R

S