MoMeDa_8

July 11, 2024

1 Exercise 8: Unfolding

In almost any experiment, the measured variable is not directly the physics quantity of interest. Instead, the answer of the experimental apparatus has to be translated by the experimentalist, e.g., the signal height might be related to the energy of a particle. In real experiments, it is impossible to always find the true value of the physics quantity, since the translation suffers from various uncertainties. E.g., the measured signal might be distorted due to noise and systematic limitations of the experiment. Moreover, the *response function* of the experiment is convoluted in the measured signal. In most cases, an analytical deconvolution of the signal is impossible or at least not practical. Instead, numerical methods are applied to find the distribution of true values and their uncertainties for a given distribution of measured values. This process is called unfolding.

In this exercise, we consider only discrete distributions with N bins, corresponding either to histograms of continuous variables (like energy) or naturally discrete variables (like the mass number of atomic nuclei). Provided that the true and measured distributions have the same number of bins, the experimental response can be described by an $N \times N$ migration (or transfer) matrix R, where each element R_{ij} describes the probability of a true value in bin i to be classified or measured, respectively, in bin j. This is reflected in the relation

$$\vec{g} = R \cdot \vec{f},$$

where $\vec{f}$ is a vector with N elements containing the true values for each bin, R is the migration matrix corresponding to the experimental response, and $\vec{g}$ is a vector containing the measured / observed values for each bin. This means that an ideal experiment would have the unit matrix as migration matrix.

The first step of the unfolding procedure is the determination of the migration matrix, e.g., by calibration measurements or simulations. We assume that this step has been done already, and the exercises focus on the second step: unfolding of a given measured distribution provided that the migration matrix is known.

1.0.1 Hints:

Numpy offers several matrix operations, for example:

`linalg.inv(R)` returns the inverse of the a (migration) matrix `R`. See also `np.matrix.I` which performs the same operation.

`lambda, U = linalg.eig(R)` returns the eigenvalues and the matrix `U` of eigenvecotrs you need to diagonalize `R`.

Normally you should not capitalize variables in python!

1.1 Exercise 8.1: Simple Case with 2 Categories (voluntary)

The simplest situation is an experiment with $N = 2$, for example the number of fouls per team in a soccer match between team A and B. Let us assume that the referee is biased and favors team B. Whenever the referee detects a foul committed by team B, he gives a free kick to team A in only **70%** of the cases, but a free kick to team B in the remaining **30%** of the cases. Vice-versa, when team A commits a foul, team B gets the free kick in **90%** of the cases, but team A only in the remaining **10%** of the cases.

Thus, the diagonal entries of the migration matrix R are 0.9 and 0.7, and the off-diagonal elements are 0.1 and 0.3 (think about which is the correct location of each element).

In the match there have been **22** free kicks for team A, and **24** for team B. Construct the migration matrix R, and estimate by inversion of R how many fouls each team has committed.

You can do this using LaTeX in Markdown, or in Python with Numpy arrays.

[84]:
```
import numpy as np
```

[85]:
```
epsilon1 = 0.1  # 10% wrong detection probability, if team A commits foul
epsilon2 = 0.3  # 30% wrong detection probability, if team B commits foul
foulsDetectedForTeamA = 24. # = number of free kicks for team B
foulsDetectedForTeamB = 22. # = number of free kicks for team A

# Data from exercise sheet
print(f"Free kicks of team A:\t{foulsDetectedForTeamB} \t (= foul count for␣
 ↪team B)"),
print(f"Free kicks of team B:\t{foulsDetectedForTeamA} \t (= foul count for␣
 ↪team A)")
print(f"Fraction of wrong decisions, for team A:\t{epsilon1}")
print(f"Fraction of wrong decisions, for team B:\t{epsilon2}")

# TODO: Unfold the referees bias decisions using numpy!
```

```
Free kicks of team A:   22.0     (= foul count for team B)
Free kicks of team B:   24.0     (= foul count for team A)
Fraction of wrong decisions, for team A:        0.1
Fraction of wrong decisions, for team B:        0.3
```

`# TODO` … or in Markdown!

1.2 Exercise 8.2: Regularization (obligatory)

Now we consider a counting experiment with 7 categories (bins): each bin contains a number of observed events (e.g., particle decays, cars with different colors, classes of stars in an astronomical survey). We start from the following true distribution $\vec{f}$ of 3000 events in the 7 bins:

1 | 2 | 3 | 4 | 5 | 6 | 7 |
:– | :– | :– | :– | :– | :– | :- - | :– |

35 | 218 | 814 | 1069 | 651 | 195 | 18 |

In our experiment some of the events are misclassified. For an event which truly belongs to bin i, there is a 30% chance each that it is measured instead in bin $i-1$ or bin $i+1$ (except if the measured bin would be outside of the possible range from 1 to 7). Consequently, the migration matrix R has 0.3 in all elements next to the diagonal, and 0.4 in the diagonal elements except for the corner elements which are 0.7.

[86]:
```
import numpy as np
from matplotlib import pyplot as plt
```

[87]:
```
# Here are some nice colors for your illustrations:
green = '#009682'
blue = '#4664aa'
maygreen = '#8cb63c'
yellow = '#fce500'
orange = '#df9b1b'
brown = '#a7822e'
red = '#a22223'
purple = '#a3107c'
cyan = '#23a1e0'
black = '#000000'
light_grey = '#bdbdbd'
grey = '#797979'
dark_grey = '#4e4e4e'

clrs = [
    blue,
    maygreen,
    orange,
    green,
    cyan,
    red,
    grey
]
```

[88]:
```
# Number of Categories/Bins. Transfer matrix will be n_bins x n_bins
n_bins = 7
n_events = 3000

 # True event vector
f = np.array([35., 218., 814., 1069., 651., 195., 18.])

R = np.zeros((7,7))
R += np.identity(7)*0.4
R += np.diag([1]*6, 1)*0.3
R += np.diag([1]*6, -1)*0.3
R[0,0] = 0.7
```

```
R[-1,-1] = 0.7
R
```

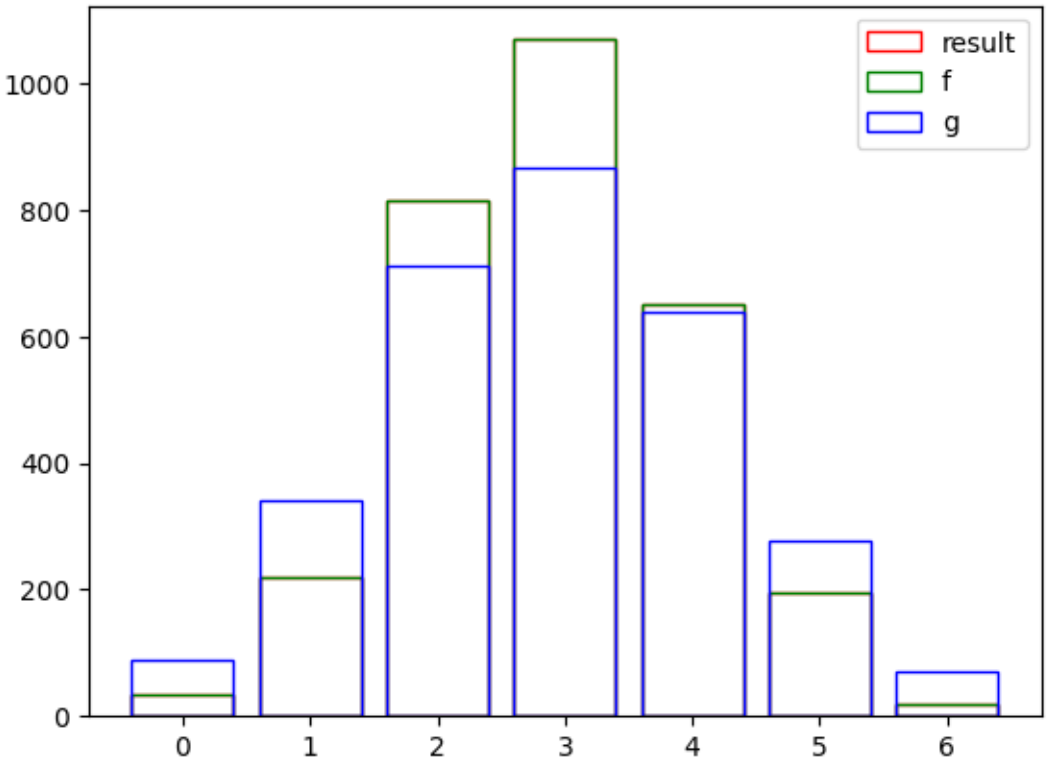
[88]:
```
array([[0.7, 0.3, 0. , 0. , 0. , 0. , 0. ],
       [0.3, 0.4, 0.3, 0. , 0. , 0. , 0. ],
       [0. , 0.3, 0.4, 0.3, 0. , 0. , 0. ],
       [0. , 0. , 0.3, 0.4, 0.3, 0. , 0. ],
       [0. , 0. , 0. , 0.3, 0.4, 0.3, 0. ],
       [0. , 0. , 0. , 0. , 0.3, 0.4, 0.3],
       [0. , 0. , 0. , 0. , 0. , 0.3, 0.7]])
```

First, let's consider an ideal experiment without any uncertainties:

a) Obtain the observed distribution of events $\vec{g}$. Then, unfold the observation by multiplication with R^{-1}, and compare the result to the true data. For this, plot the true distribution, the observed distribution, and the unfolded distribution.

[89]:
```
g = R.dot(f)
fin = np.linalg.inv(R).dot(g)

plt.bar(list(np.arange(7)),fin, label="result", color='none', edgecolor='red')
plt.bar(list(np.arange(7)),f, label="f", color='none', edgecolor='green')
plt.bar(list(np.arange(7)),g, label="g", color='none', edgecolor='blue')
plt.legend()
```

[89]:
```
<matplotlib.legend.Legend at 0x7f723353d160>
```



Its the same.

Now we consider a more realistic case with uncertainties on the observed events (e.g., Poissonian uncertainties in the case of particle decays). Thus, the number of observed events deviates from the ideal case, and it is more difficult to reconstruct the original distribution. In our specific case, the experiment has observed the following distribution $\vec{g}_{\text{obs}}$ for the true distribution $\vec{f}$ given above:

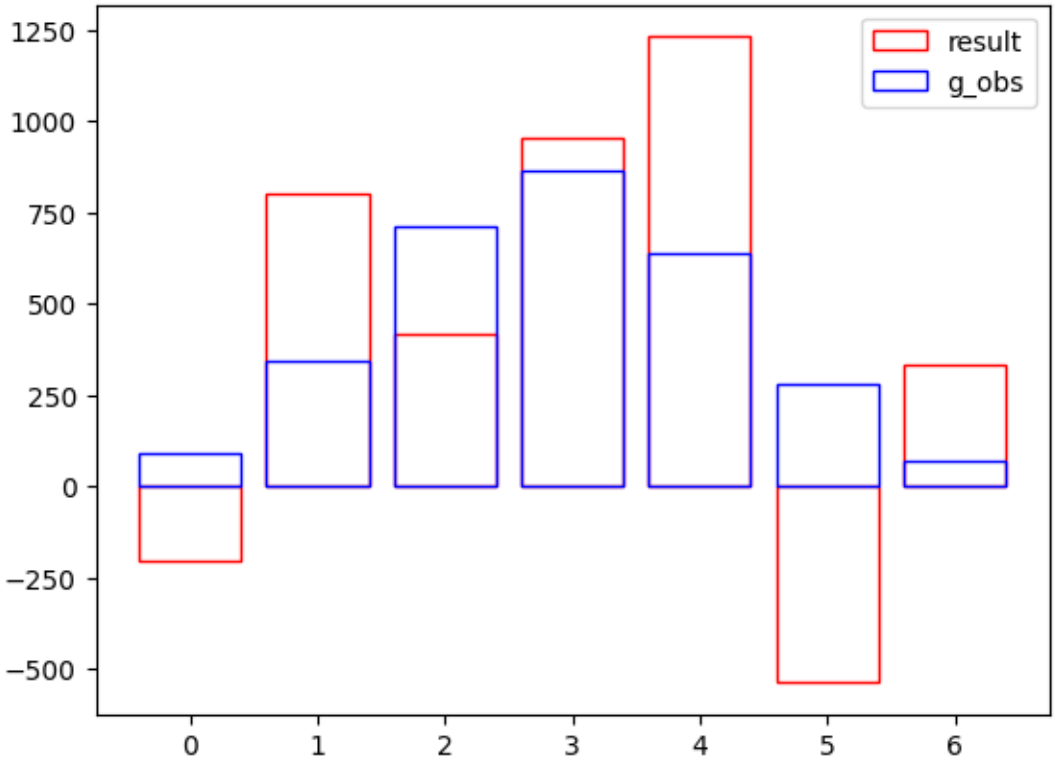
1	2	3	4	5	6	7
99	386	695	877	618	254	71

b) Unfold the observation by multiplication with R^{-1}, and compare the true, observed, and unfolded distribution by plotting them together. Which problems do you encounter?

[90]:
```
# Measurement including uncertainties
g_obs = np.array([99., 386., 695., 877., 618., 254., 71])
```

[91]:
```
fin = np.linalg.inv(R).dot(g_obs)

plt.bar(list(np.arange(7)),fin, label="result", color='none', edgecolor='red')
plt.bar(list(np.arange(7)),g, label="g_obs", color='none', edgecolor='blue')
plt.legend()
```

[91]: <matplotlib.legend.Legend at 0x7f7233cdf9e0>



Some of them are negative

Such unphysical *oscillations* in the unfolded result are typical for many unfolding techniques. Suppressing them is a major challenge when unfolding real data. One way to achieve this is regularization. There are various sophisticated methods for regularization (cf. lecture). Here is a simple method which can be programmed easily in Python:

- Diagonalize the migration matrix R: $R_{\text{diag}} = U^{-1}RU$ such that the eigenvalues λ_i of R form the diagonal of R_{diag}.
- Construct the observation vector $\vec{g}_{\text{diag}} = U^{-1}\vec{g}_{\text{obs}}$ and multiply each component i of the resulting vector with the corresponding component of $\vec{\lambda}^{-1}$, where $\vec{\lambda}^{-1}$ is a vector containing the reciprocals of the eigenvalues of R.
- The *regularization*: Set all elements i of $\vec{g}_{\text{diag}}$ to 0, for which λ_i is smaller than a chosen threshold λ_{reg}. The choice of λ_{reg} is critical and a compromise between suppressing the unphysical oscillations, and keeping as much information as possible of the measurement.
- The unfolded result is $U \cdot \vec{g}_{\text{diag}}$.

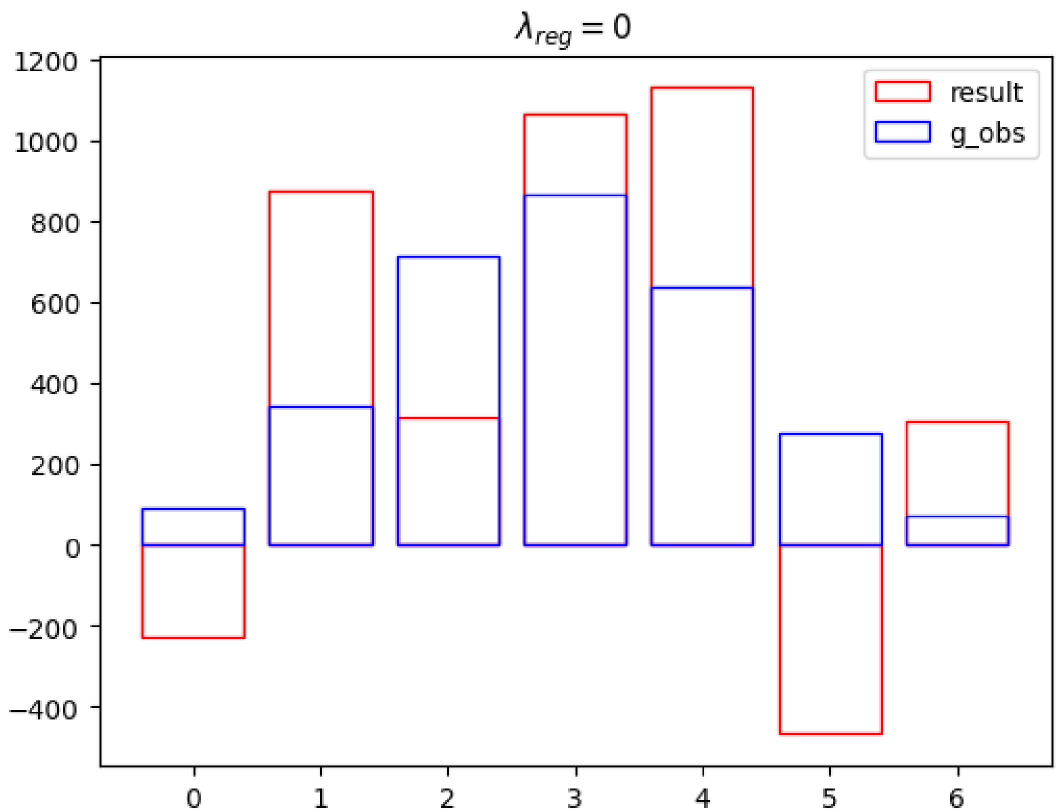
c) Apply the regularization method. Use different thresholds λ_{reg} from -1 to 1. As a cross-check: if the algorithm is implemented correctly, the result should be identical to the one of exercise 8.2 b), provided that λ_{reg} is smaller than the smallest eigenvalue of R. Discuss different choices for λ_{reg}: as measure for how similar the true and the unfolded distributions are, calculate the mean quadratic deviation (average over all

bins). For which choice of λ_{reg} is the unfolded distribution most similar to the true distribution?

[94]:
```
def do_unfolding_with_regularization(lamb_regu):
    λ, U = np.linalg.eig(R)
    print(λ)
    Rdig = np.linalg.inv(U).dot(R.dot(U))
    gdig = np.linalg.inv(U).dot(g_obs)*1/λ
    gdig -= gdig*(λ<lamb_regu)
    res = U.dot(gdig)
    return res
```

[98]:
```
fin = do_unfolding_with_regularization(0)
plt.title(r"$\lambda_{reg}=0$")
plt.bar(list(np.arange(7)),fin, label="result", color='none', edgecolor='red')
plt.bar(list(np.arange(7)),g, label="g_obs", color='none', edgecolor='blue')
plt.legend()
plt.show()

import scipy
scipy.optimize.minimize(lambda t: np.mean((f-
do_unfolding_with_regularization(t)**2)),(-10))

t = np.linspace(-1,1)
z = [f.dot(do_unfolding_with_regularization(tt)) for tt in t]
plt.plot(t, z)
plt.show()

t = np.linspace(-1200,-800)
z = [f.dot(do_unfolding_with_regularization(tt)) for tt in t]
plt.plot(t, z)
plt.show()


fin = do_unfolding_with_regularization(0.2)
plt.title(r"$\lambda_{reg}=-1000$")
plt.bar(list(np.arange(7)),fin, label="result", color='none', edgecolor='red')
plt.bar(list(np.arange(7)),g, label="g_obs", color='none', edgecolor='blue')
plt.legend()
plt.show()
```

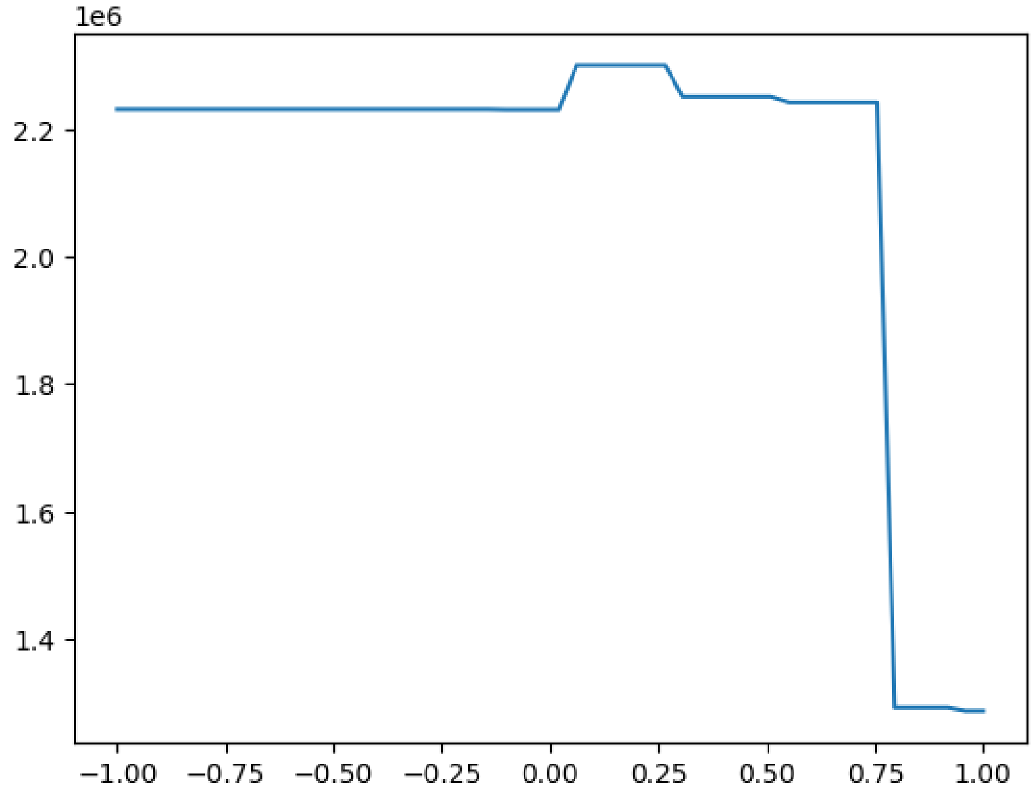
```
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
```




```
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
```

```
  0.94058132]
 [-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
 [-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
 [-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
 [-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
 [-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
 [-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
 [-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
 [-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
 [-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
 [-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
 [-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
 [-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
 [-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
 [-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
 [-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
 [-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
 [-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
 [-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
 [-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
 [-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
 [-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
 [-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
 [-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
 [-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
```

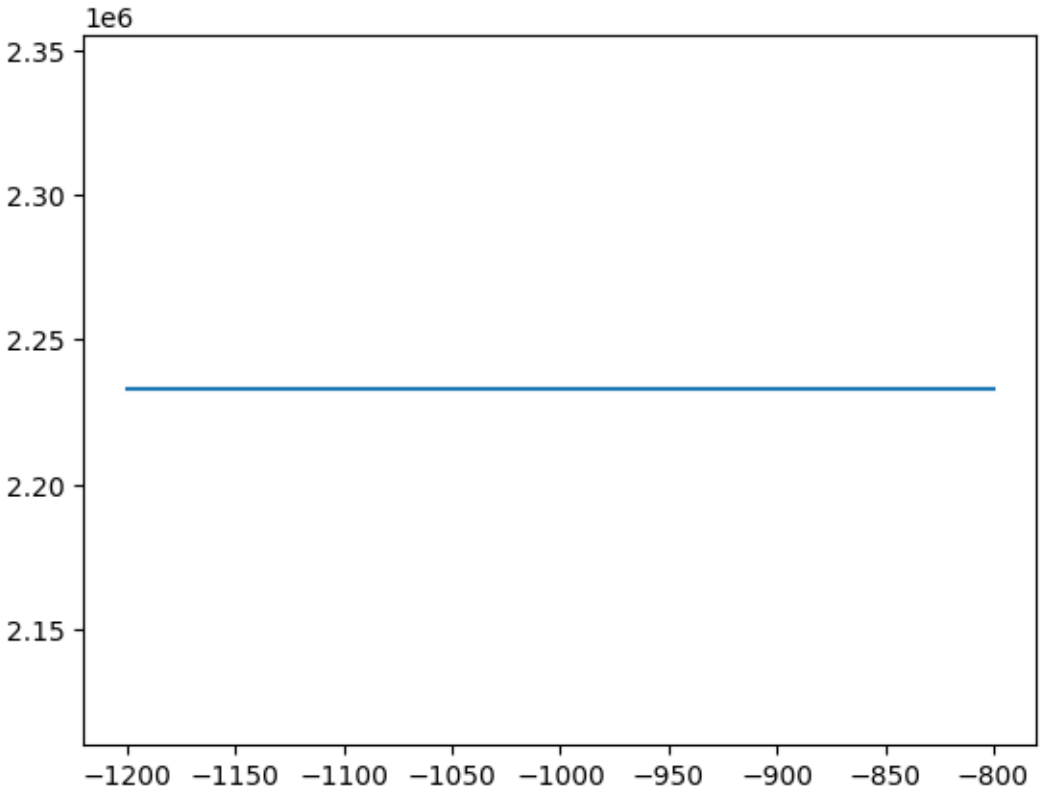
```
 0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
 0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
 0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
 0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
 0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
 0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
 0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
 0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
 0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
 0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
 0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
 0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
 0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
 0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
 0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
 0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
 0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
 0.94058132]
```



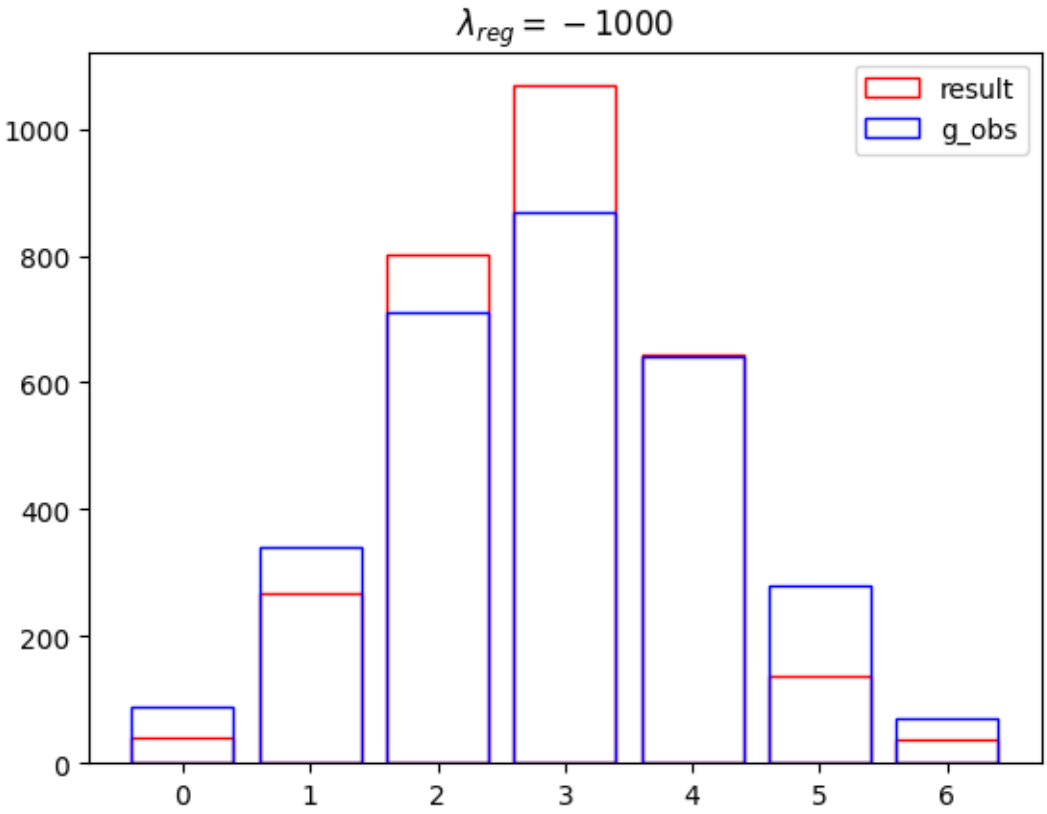

```
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
```

```
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
```

```
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
```




```
[-0.14058132  0.02590612  0.26648744  0.53351256  0.77409388  1.
  0.94058132]
```



There are no negative results anymore

Finally, we will apply an iterative method for unfolding. We start with a flat assumption for $\vec{f}_0$, i.e., each element is 3000 / 7 in our case, since we have 7 bins and 3000 events. Then, we fold $\vec{f}_0$ by multiplication with R and compare the resulting $\vec{g}_0$ with the observation g_{obs}. Depending on the discrepancy between g_0 and g_{obs}, we *tune* our next guess for the true distribution $\vec{f}_1$ and repeat the procedure to obtain $\vec{g}_1$. If the *tuning* is reasonable, the guessed distribution will become closer to the the true distribution with each step. However, in the presence of uncertainties, too many iteration can lead to unreasonable results. Thus, choosing the number of iterations is again a challenge, similar to choosing the best threshold λ_{reg} in the regularization method.

We will apply the following unfolding algorithm to improve our guess from iteration step k to step $k+1$ (in a simplified version for symmetric response matrices):

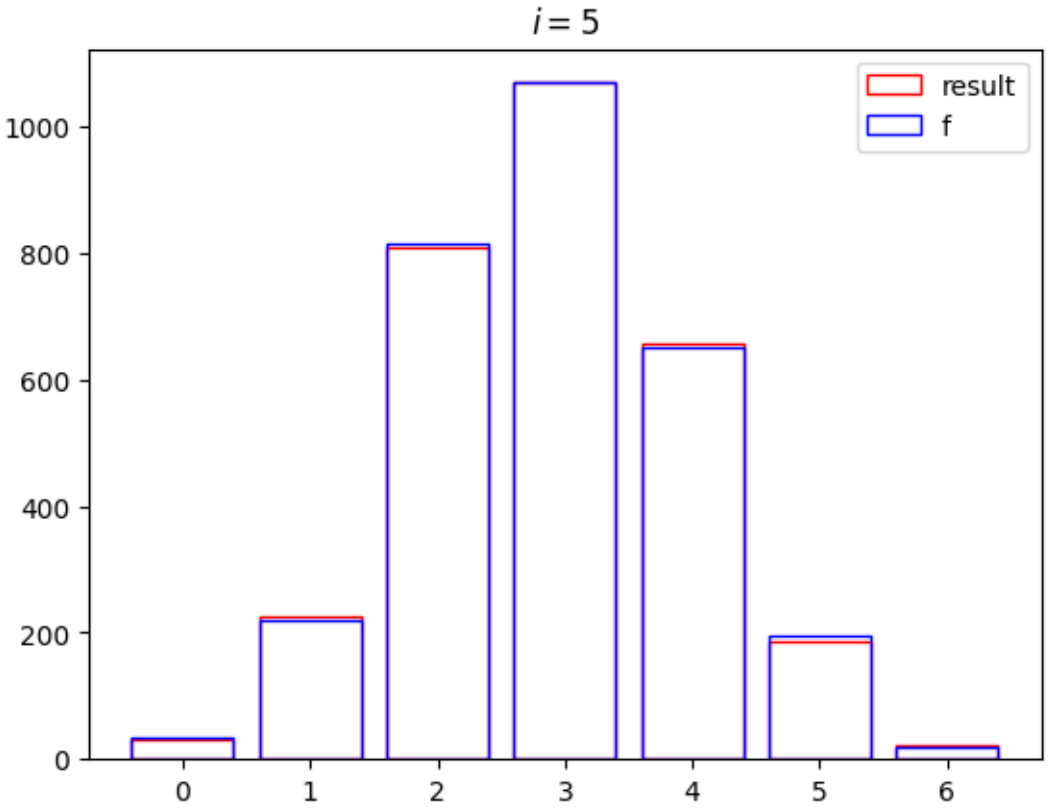
1) $\vec{g}_{k+1} = R \cdot \vec{f}_k$
2) Tuning: calculate a vector $\vec{c}$ with weights to scale each bin i: $c^i = g^i_{\text{obs}}/g^i_{k+1}$. Consequently, the weight for bin i is 1 if the observation is already reproduced by the result of step k.
3) Calculate $\vec{f}_{k+1}$: Multiply each element of $ $\vec{f}_k$$ $with the corresponding element of the vector R \cdot \vec{c}$.

d) Apply the iterative method on the distribution you would observe without any experimental uncertainties. How many iterations do you need to achieve a maximum

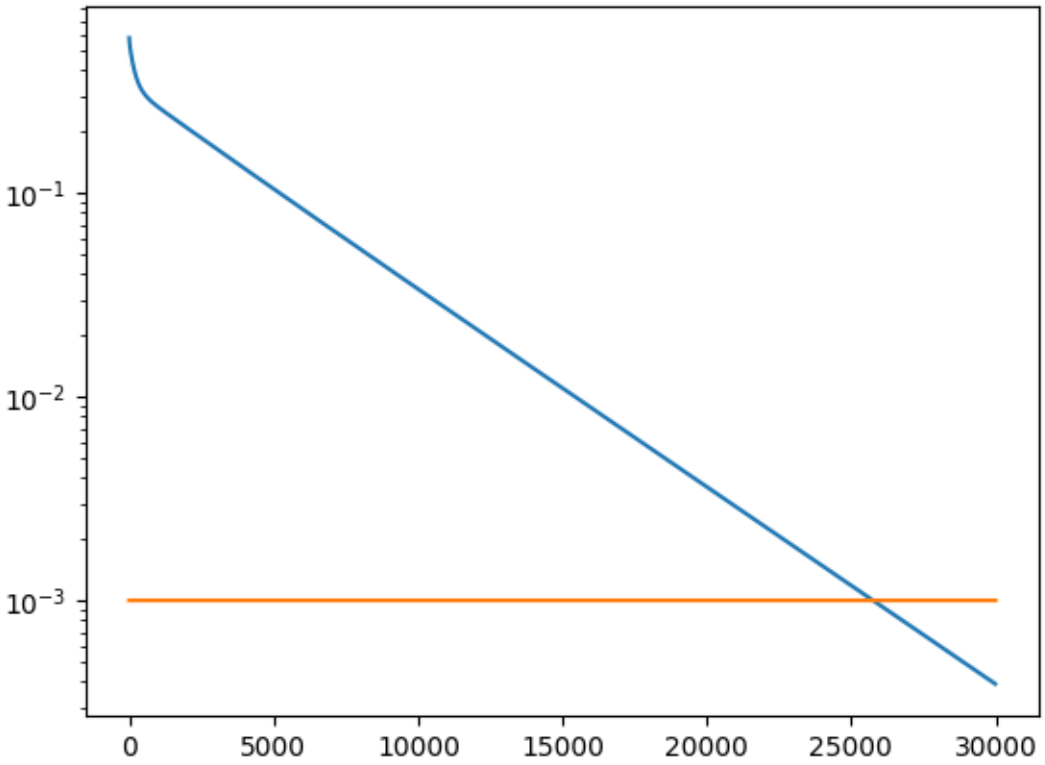
deviation of 0.1% per bin between the true distribution and the unfolded observation?

[34]:
```
def do_iterative_unfolding(g_in, max_iterations):
    f1 = np.array([3000/7]*7)
    for i in range(max_iterations):
        g1 = R.dot(f1)
        c = g_in/g1
        f1 = f1*R.dot(c)
    return f1
    # TODO: Implement the iterative approach to unfolding in this function.
    #       The parameter g_in can be used to provide the idealy folded events as input, or the measured events with uncertainty.
    #       max_iterations shall be used to define the maximal number of interations used by the algorithm.
```

[48]:
```
fin = do_iterative_unfolding(g,2700)
plt.title(r"$i=5$")
plt.bar(list(np.arange(7)),fin, label="result", color='none', edgecolor='red')
plt.bar(list(np.arange(7)),f, label="f", color='none', edgecolor='blue')
plt.legend()
plt.show()
```



[51]:
```
max_iterations = 30000
fs = []
f1 = np.array([3000/7]*7)
for i in range(max_iterations):
    g1 = R.dot(f1)
    c = g/g1
    f1 = f1*R.dot(c)
    fs.append(np.max(np.abs((f1-f)/f)))

plt.plot(fs[10:])
plt.plot(np.arange(max_iterations), np.full((max_iterations),1e-3))
plt.yscale("log")
plt.show()
```

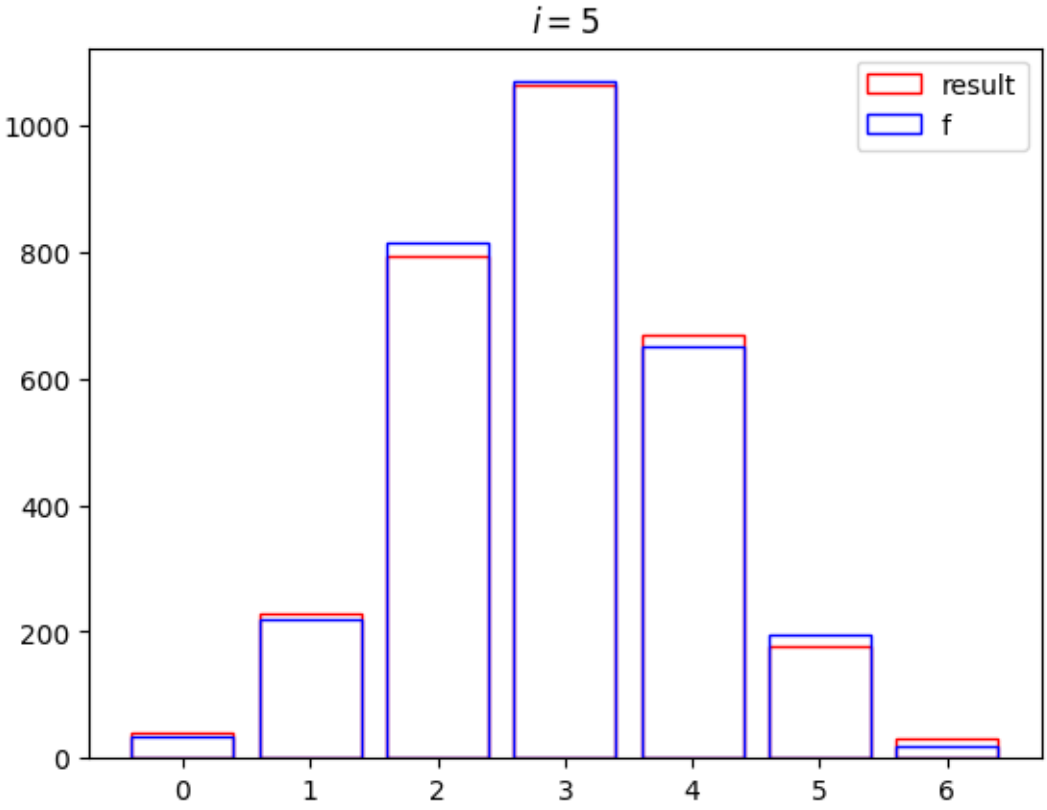


It's slow

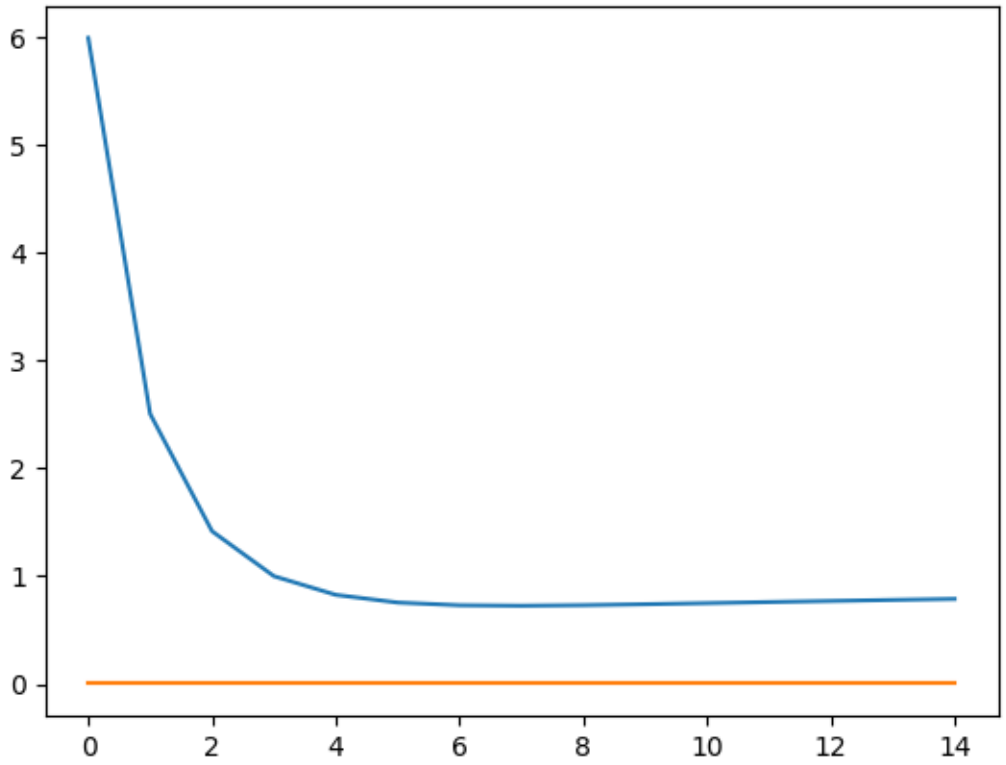
e) Now, apply the method on the observation with uncertainties $\vec{g}_{\text{obs}}$. Try and discuss different choices for the number of iterations.

[54]:
```
fin = do_iterative_unfolding(g,6)
plt.title(r"$i=5$")
plt.bar(list(np.arange(7)),fin, label="result", color='none', edgecolor='red')
plt.bar(list(np.arange(7)),f, label="f", color='none', edgecolor='blue')
```

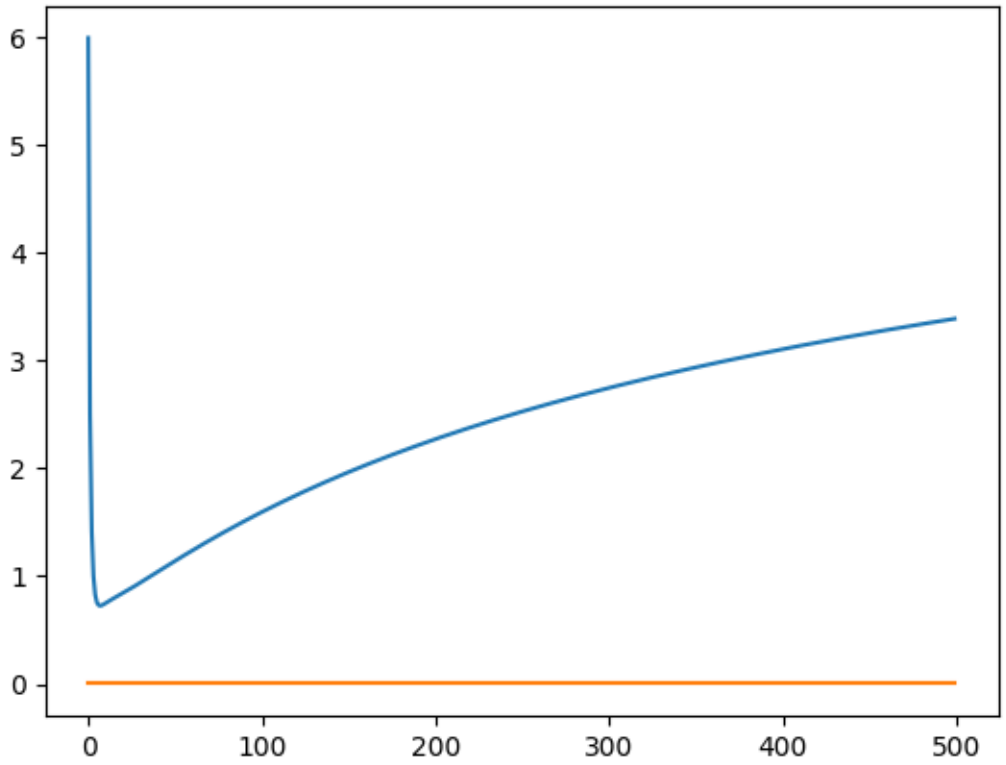
```
plt.legend()
plt.show()
```



[53]:
```
max_iterations = 15
fs = []
f1 = np.array([3000/7]*7)
for i in range(max_iterations):
    g1 = R.dot(f1)
    c = g_obs/g1
    f1 = f1*R.dot(c)
    fs.append(np.max(np.abs((f1-f)/f)))

plt.plot(fs)
plt.plot(np.arange(max_iterations), np.full((max_iterations),1e-3))
plt.show()
```



[55]:
```
max_iterations = 500
fs = []
f1 = np.array([3000/7]*7)
for i in range(max_iterations):
    g1 = R.dot(f1)
    c = g_obs/g1
    f1 = f1*R.dot(c)
    fs.append(np.max(np.abs((f1-f)/f)))

plt.plot(fs)
plt.plot(np.arange(max_iterations), np.full((max_iterations),1e-3))
plt.show()
```



It get's worse for more iterations after 6.

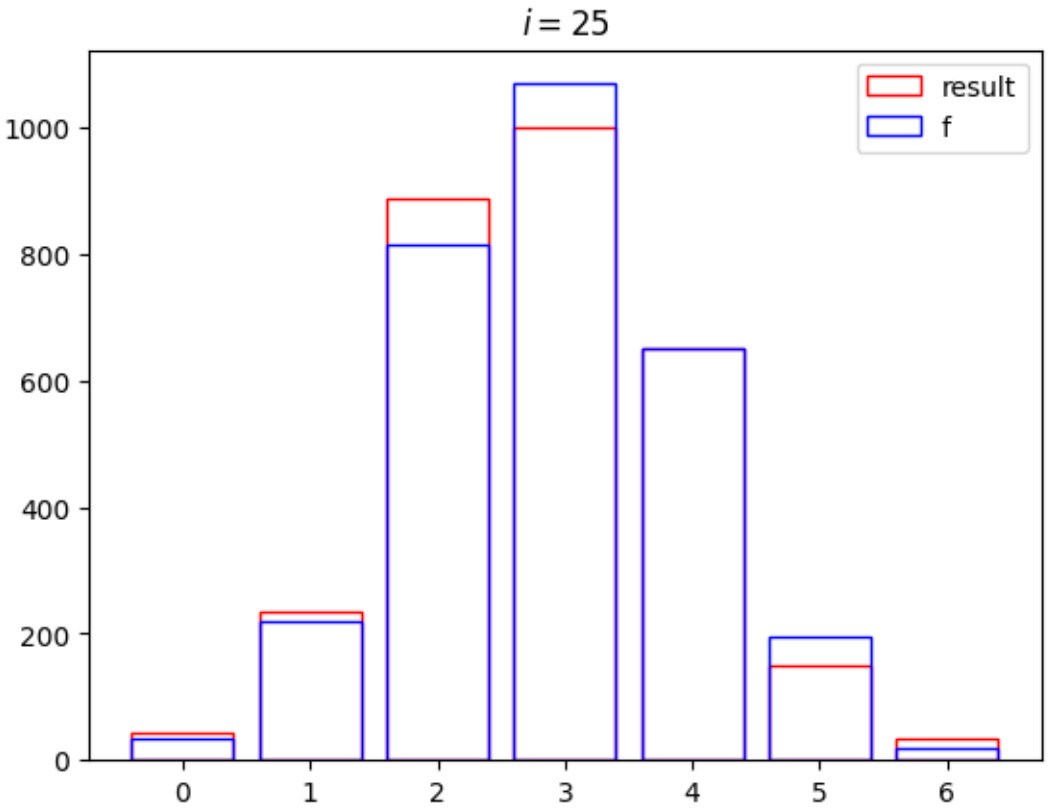
f) The choice of $\vec{f}_0$ is similar to the choice of a prior in a Bayesian analysis and influences the result. Try different choices for $\vec{f}_0$ and test the influence on the result for small and large number of iterations.

[15]:
```
# Different options for priors of f_start:
prior1 = np.array([1.,20.,300.,400.,300.,20.,1.])
prior2 = np.array([4.,3.,2.,1.,2.,3.,4.])
```

[57]:
```
# TODO: Extend the function do_iterative_unfolding so that a prior can be␣
 ↪provided via an argument. This prior should than be used as starting vector␣
 ↪for f_0.
import copy
def do_iterative_unfolding(g_in, max_iterations, prior):
    f1 = prior[:]
    for i in range(max_iterations):
        g1 = R.dot(f1)
        c = g_in/g1
        f1 = f1*R.dot(c)
    return f1
    # TODO: Implement the iterative approach to unfolding in this function.
```

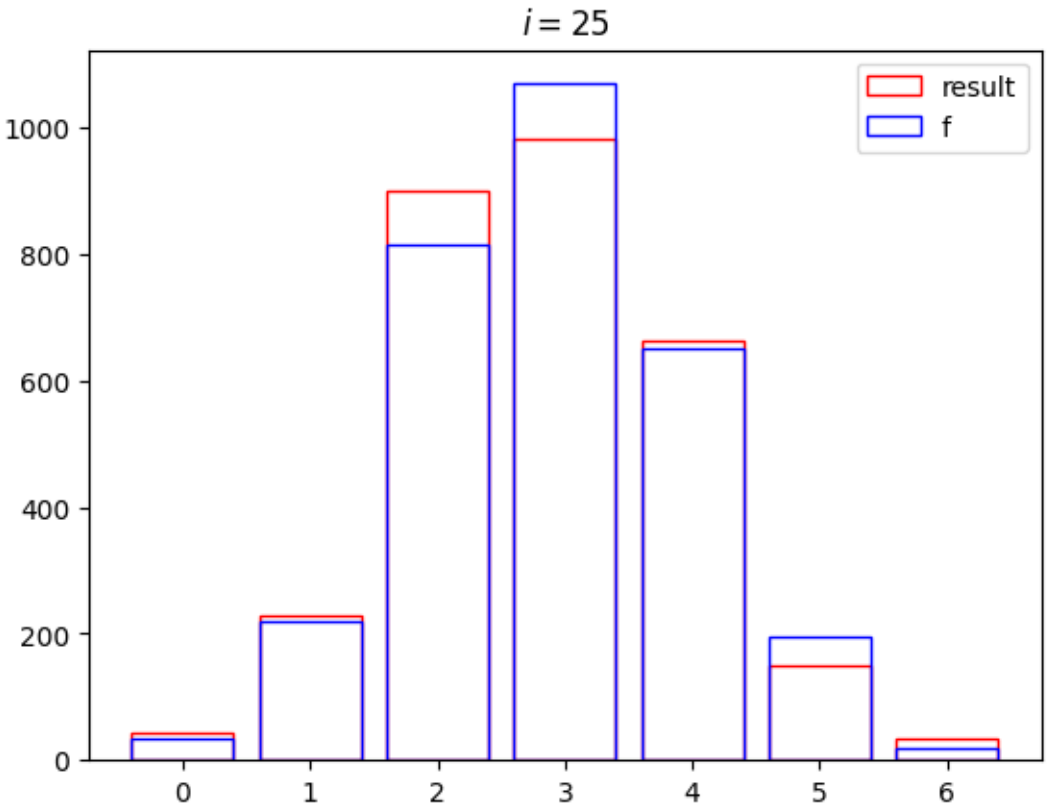
```
    #       The parameter g_in can be used to provide the idealy folded events␣
 ↪as input, or the measured events with uncertainty.
    #       max_iterations shall be used to define the maximal number of␣
 ↪interations used by the algorithm.
```

[75]:
```
# TODO: Apply this extended function on the measurements with uncertainties␣
 ↪using different priors and max_iterations values to optimize the mean␣
 ↪quadratic deviation!
fin = do_iterative_unfolding(g_obs, 25, prior1)
plt.title(r"$i=25$")
plt.bar(list(np.arange(7)),fin, label="result", color='none', edgecolor='red')
plt.bar(list(np.arange(7)),f, label="f", color='none', edgecolor='blue')
plt.legend()
plt.show()
```

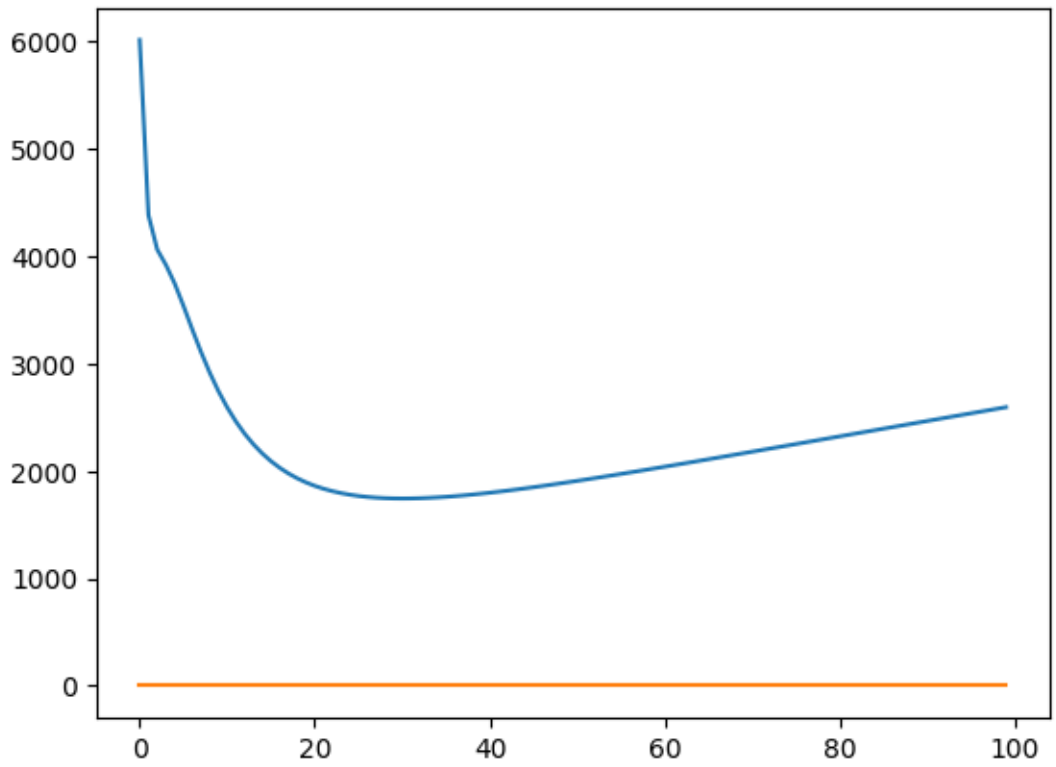


[100]:
```
# TODO: Apply this extended function on the measurements with uncertainties␣
 ↪using different priors and max_iterations values to optimize the mean␣
 ↪quadratic deviation!
fin = do_iterative_unfolding(g_obs, 25, prior2)
plt.title(r"$i=25$")
```

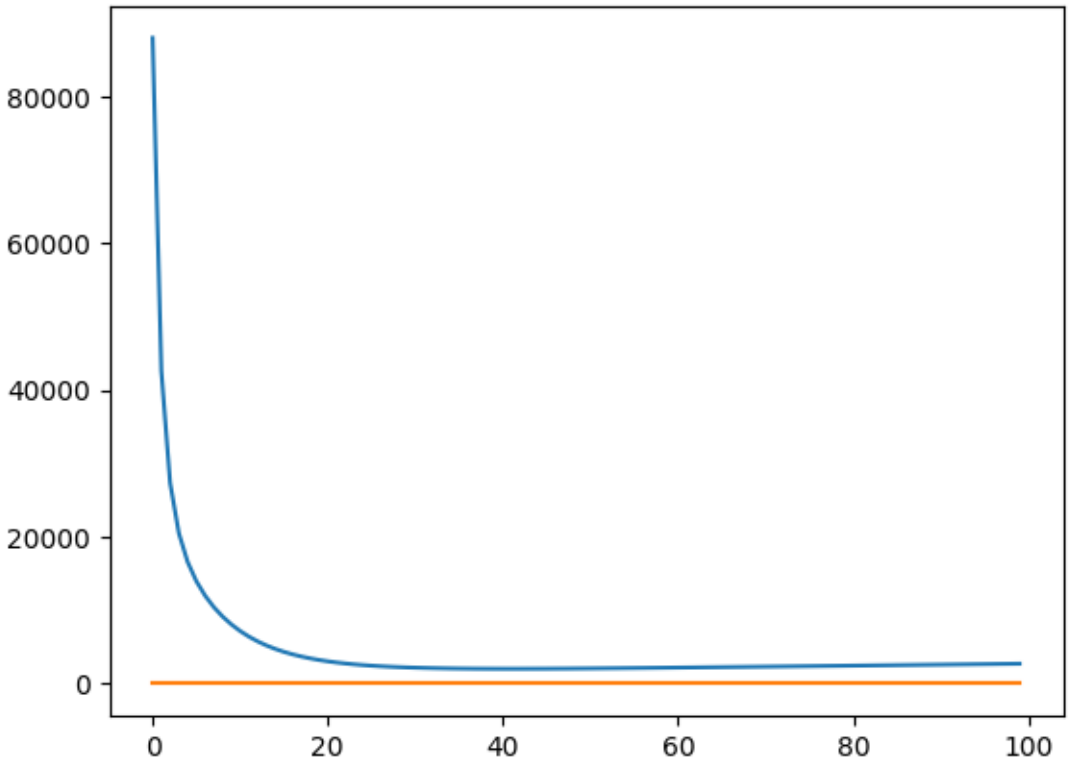
```
plt.bar(list(np.arange(7)),fin, label="result", color='none', edgecolor='red')
plt.bar(list(np.arange(7)),f, label="f", color='none', edgecolor='blue')
plt.legend()
plt.show()
```



[73]:
```
max_iterations = 100
fs = []
f1 = prior1[:]
for i in range(max_iterations):
    g1 = R.dot(f1)
    c = g_obs/g1
    f1 = f1*R.dot(c)
    fs.append(np.mean((f1-f)**2))

plt.plot(fs)
plt.plot(np.arange(max_iterations), np.full((max_iterations),1e-3))
plt.show()
```



[74]:
```
max_iterations = 100
fs = []
f1 = prior2[:]
for i in range(max_iterations):
    g1 = R.dot(f1)
    c = g_obs/g1
    f1 = f1*R.dot(c)
    fs.append(np.mean((f1-f)**2))

plt.plot(fs)
plt.plot(np.arange(max_iterations), np.full((max_iterations),1e-3))
plt.show()
```



[ ]:

```
# TODO: Explain what you observe using the markdown syntax
```