# aufgabe2

January 10, 2019

# 1 Aufgabe 29 - Entfaltung mit quadratischen Matrizen

## 1.1 Teilaufgabe a)

DIe Matrix beschreibt einen Messprozess, in welchem die Daten $n$ verschiedenen Bins zugeordnet werden können und Fehlklassifikation nur zu nächsten Nachbarn hin stattfinden können. Dies geschieht mit Wahrscheinlichkeit $\epsilon$.

```
In [1]: import numpy as np
        from scipy.sparse import diags # to easily make a tridiagonal matrix
        import matplotlib.pyplot as plt
```

```
In [2]: def getResponseMatrix(n, epsilon):
            A = 1-2*epsilon*np.ones(n)
            A[0] = A[n-1] = 1-epsilon
            diagonals = [A, epsilon*np.ones(n-1), epsilon*np.ones(n-1)]
            A = diags(diagonals, [0, 1, -1]).toarray() # 0,1,-1 means place the subarrays of d
            # in the main diag and the two sub diags
            return A
```

## 1.2 Teilaufgabe b)

```
In [3]: n = 20
        epsilon = 0.23
        A = getResponseMatrix(n, epsilon)
        f = np.array([193,485,664,763,804,805,779,736,684,626,
                      566,508,452,400,351,308,268,233,202,173])
        g = np.dot(A,f)
        prng = np.random.RandomState(0) # Das ist zur Reproduzierbarkeit!
        gmess = prng.poisson(g, np.size(g)) # Ziehe np.size(g) poissonverteilte Zufallszahlen m
        print('Die gemessenen g sind bei uns dann',gmess)
```

```
Die gemessenen g sind bei uns dann [262 465 640 745 873 825 780 684 705 623 534 510 438 398 358
 209 167]
```

### 1.3 Teilaufgabe c)

Die Faltungsgleichung $\mathbf{g} = \mathbf{A}\,\mathbf{f}$ lautet mit $\mathbf{A} = \mathbf{U}\,\mathbf{D}\,\mathbf{U}^{-1}$ dann

$$\mathbf{g} = \mathbf{U}\,\mathbf{D}\,\mathbf{U}^{-1}\mathbf{f} \iff \mathbf{c} = \mathbf{D}\mathbf{b} \tag{1}$$

mit $\mathbf{c} = \mathbf{U}^{-1}\mathbf{g}$ und $\mathbf{b} = \mathbf{U}^{-1}\mathbf{f}$.

In [4]:
```
w, U = np.linalg.eig(A) # w are the eigenvalues, U is a matrix of eigenvectors
index = w.argsort()[::-1]
w = w[index]
U = U[:,index]
Uinv = np.linalg.inv(U)
D = np.diag(w)
Dinv = np.linalg.inv(D)
# Nun sind die EW sortiert und die EV in der Matrix U auch.
```

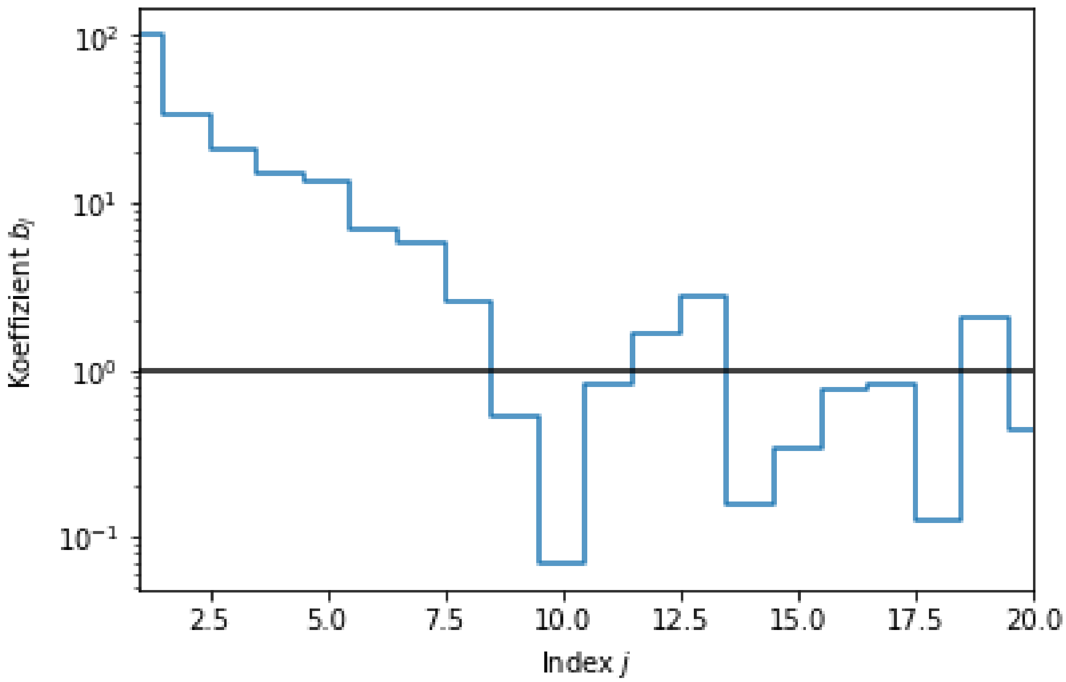
### 1.4 Teilaufgabe d)

In [5]:
```
c = Uinv@gmess #Transformation in EV-Basis
b = Dinv@c #Transformation in EV-Basis

Vg = np.diag(g) #Kovarianzmatrix von gmess, Poissonverteilung: Varianz=Erwartungswert
B = Dinv@Uinv
Vb = B@Vg@B.T ##Kovarianzmatrix von b, transformiert mit BVB-Formel
bvar = np.diag(Vb)
bstan = np.sqrt(bvar)
bskal = np.abs(b/bstan) # auf ihre Standardabweichungen skalierte b-Koeffizienten
print('Die skallierten b-Koeffizienten:')
print(bskal)

plt.yscale('log')
plt.step(np.linspace(1,np.size(bskal),np.size(bskal)), bskal, where='mid')
plt.plot([1,20],[1,1], 'k')
plt.xlim(1,20)
plt.xlabel(r'Index $j$')
plt.ylabel(r'Koeffizient $b_j$')
plt.show()
```

```
Die skallierten b-Koeffizienten:
[1.00670000e+02 3.33459553e+01 2.12545443e+01 1.48061583e+01
 1.35532763e+01 6.96412667e+00 5.83471359e+00 2.58989981e+00
 5.31579199e-01 6.99953203e-02 8.30000000e-01 1.64280665e+00
 2.80088219e+00 1.55432551e-01 3.38590440e-01 7.72013059e-01
 8.21317610e-01 1.23602133e-01 2.05064543e+00 4.44481149e-01]
```




Koeffizienten, die unterhalb der 1 liegen, sind kleiner als ihre eigene Standardabweichung. Deshalb bieten diese Werte keine nützliche Information und sollten nicht berücksichtigt werden.

### 1.5 Teilaufgabe e)

In [6]:
```
#In der folgenden Schleife wird ein Vektor mit regularisierten b-Koeffizienten erstell
# Wird das erste Mal ein skallierter b-Koeffizient kleiner als eins erreicht werden al

breg = np.array(b)
bvar_reg = np.array(bvar) # auch die Varianz von b muss reguliert werden.

k = 0
cut = False
for i in bskal:
    if cut:
        breg[k] = 0
        bvar_reg[k] = 0
    else:
        if i < 1:
            cut = True
            breg[k] = 0
            bvar_reg[k] = 0

    k +=1
```

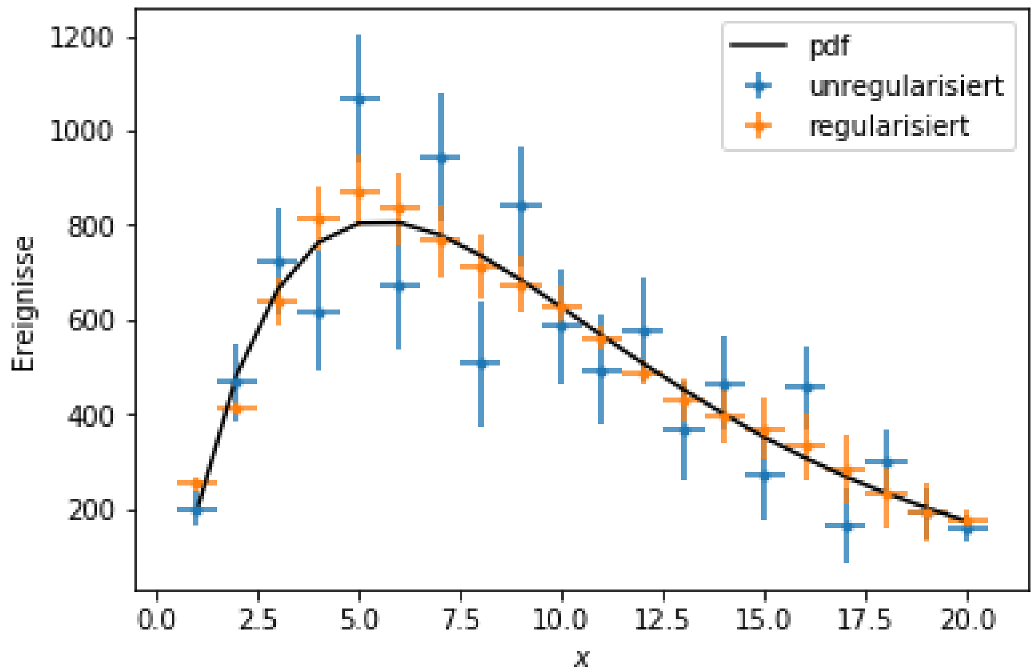
```
print(b)
print(breg)
print(bvar)
print(bvar_reg)
```

```
[-2251.04963295  -690.91922954  -466.52623091  -342.23670464
  -329.3666805   -179.20573364  -160.66734023   -77.26719097
    17.4191047      2.55736399    34.36919299   -78.49226697
  -157.473201      10.5043462    -28.13801846    80.73244377
   110.35416432   -21.61573022   464.69860952   124.47914325]
[-2251.04963295  -690.91922954  -466.52623091  -342.23670464
  -329.3666805   -179.20573364  -160.66734023   -77.26719097
     0.             0.             0.             0.
     0.             0.             0.             0.
     0.             0.             0.             0.        ]
[  500.           429.30726075   481.77974627   534.2793592
   590.56912879   662.17148923   758.25646618   890.07032339
  1073.78096732  1334.89491805  1714.6776406   2282.8681989
  3160.99366723  4567.24285488  6906.17592073 10935.70796261
 18053.24116231 30583.54964367 51352.50082362 78430.77171644]
[500.         429.30726075 481.77974627 534.2793592  590.56912879
 662.17148923 758.25646618 890.07032339   0.           0.
   0.           0.           0.           0.           0.
   0.           0.           0.           0.           0.        ]
```

In [7]:
```
#Rüchtransformation in alte Basis
funreg = U@b
freg = U@breg

# Standardabweichung von rücktransformierten f mit BVB-Formel
funreg_stan = np.sqrt(np.diag(U@Vb@U.T))
Vbreg = Vb-np.diag(bvar)+np.diag(bvar_reg) #regularisierte Kovarianzmatrix
freg_stan = np.sqrt(np.abs((np.diag(U@Vbreg@U.T))))


plt.errorbar(np.linspace(1,np.size(funreg),np.size(funreg)), funreg, xerr=0.5, yerr=fun
plt.errorbar(np.linspace(1,np.size(freg),np.size(freg)), freg, xerr=0.5, yerr=freg_stan
plt.plot(np.linspace(1,np.size(f),np.size(f)), f, 'k', label='pdf')
plt.xlabel(r'$x$')
plt.ylabel(r'Ereignisse')
plt.legend()
plt.show()
```




Die Lösung mit Regularisierung hat deutlich kleinere Fehler und liegt näher an der wahren Verteilung. Dafür entstehen durch die Glättung höhere, positive Korrelation zwischen Nachbarbins.