Projekt 17 - Singulärwertzerlegung (SVD)

Gruppe D: Colin Müntener (muentcol@students.zhaw.ch), Robin Schwarz (schwaro3@students.zhaw.ch)

Zusammenfassung

Dieses Projekt behandelt die sogenannte Singulärwertzerlegung von Matrizen (SVD für 'singular value decomposition'). Diese Methode der linearen Algebra hat unzählige Anwendungen in der Technik und insbesondere im Machine Learning. Zudem ist sie verwandt mit der Eigenwertzerlegung, die aus anderen Mathematikmodulen bekannt ist. Im Laufe der Projektbearbeitung wird die SVD schrittweise erarbeitet. Den Anfang macht eine geometrische Interpretation der Methode, wo sich ein Berührungspunkt mit der Ausgleichsrechnung ergibt. Schliesslich folgt mit der Gesichtserkennung bei der Bildbearbeitung eine konkrete Anwendung.

Bemerkung

Die numerische Berechnung der SVD ist aufwändig und anspruchsvoll. Im Rahmen dieses Projekts wird deshalb vornehmlich die Interpretation der SVD behandelt. Es sind die damit verbundenen Einsichten, welche in technischen Anwendungen helfen, die entsprechenden Methoden zu nutzen und zu beherrschen. Die eigentliche numerische Bestimmung von U, V und Σ wird einer Blackbox (matlab: svd(M), resp. python: numpy.linalg.svd(M)) überlassen. dieser Tools.

1 Gerade mit minimalen Abständen zu Punktwolke

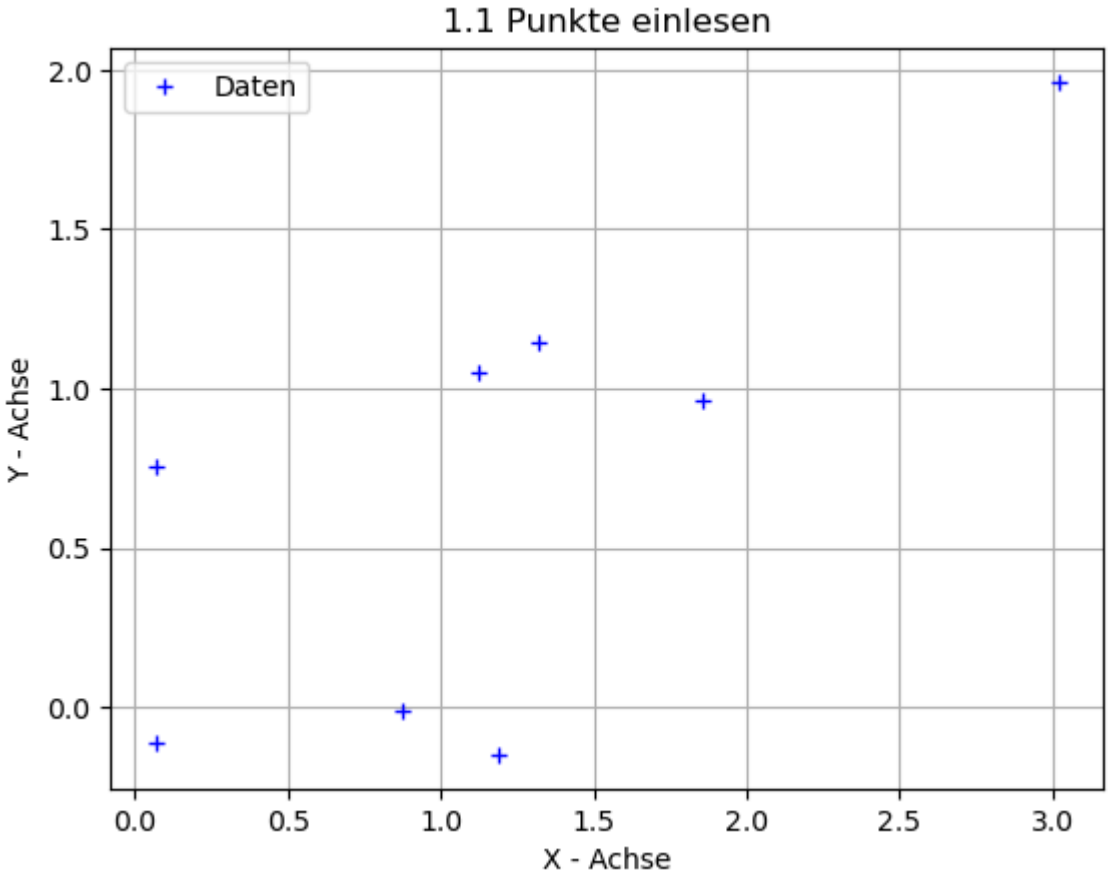
Das Bild unter 1.1 zeigt $n = 8$ Punkte, gegeben durch deren Ortsvektoren $p_i = (x_i, y_i)^t$ und eine Gerade. Die Gerade soll so bestimmt werden, dass die Summe der Abstandsquadrate minimal wird. Wichtig: der Abstand ist geometrisch definiert, d.h. er wird rechtwinklig zur Geraden gemessen und entspricht somit nicht der vertikalen Distanz zur Geraden, wie bei der linearen Ausgleichsrechnung.

1.1. Punktwolke einlesen

Punktkoordinaten in der Datei 'datenPunkte.txt'.

In [ ]:
```
import numpy as np
import scipy
import matplotlib.pyplot as plt
from numpy.linalg import svd
from numpy import polyfit
from numpy.linalg import norm
from numpy.linalg import eig
```

```
np.set_printoptions(precision=10,suppress=True)      # Anzahl Dezimalstellen im

data = np.genfromtxt('material/datenPunkte.txt')
x_data = data[:,0]
y_data = data[:,1]

plt.plot(x_data,y_data,'+b')
plt.legend(['Daten'])
plt.xlabel('X - Achse')
plt.ylabel('Y - Achse')
plt.title('1.1 Punkte einlesen')
plt.grid()
plt.show()
```



1.2. Koordinatenursprung in Mitte verschieben und Matrix Q erstellen

Zuerst wird der Ursprung in die Mitte der Punktwolke gesetzt, definiert durch $p_M = (x_M, y_M)^t$ mit

$$x_m = \frac{1}{n}\sum_{i=1}^{n} x_i$$

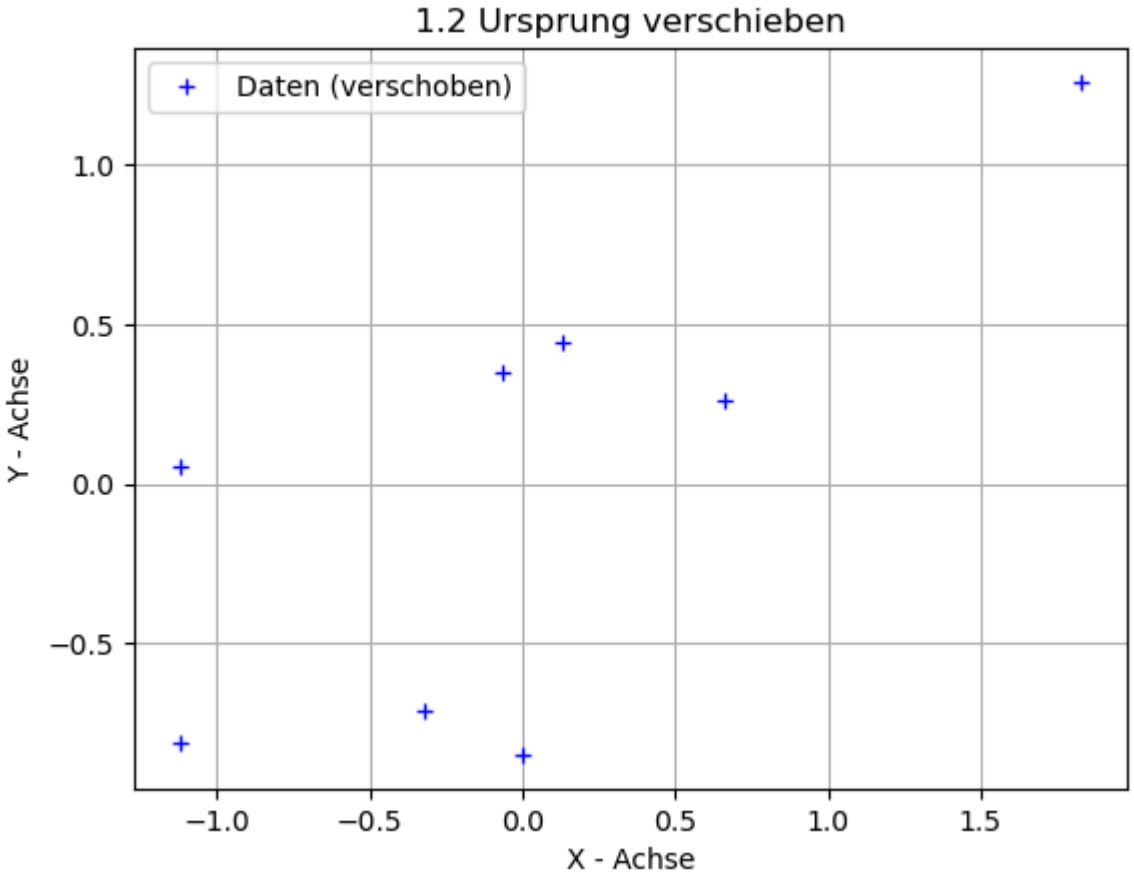
$$y_m = \frac{1}{n}\sum_{i=1}^{n} y_i$$

Die neuen Ursprungskoordinaten lautet also $x_{shift} = x - x_m, y_{shift} = y - y_m$.

Die Ortsvektorkomponenten werden in eine $n \times 2$-Matrix $Q = (x_{shift}, y_{shift})$ gepackt.

In [ ]:
```
xm = np.mean(x_data)        # Mittelwert der X-Punkte
ym = np.mean(y_data)        # Mittelwert der Y-Punkte

# Ursprung um (xm,ym) verschieben
x_data_shift = x_data - xm  # X-Achse um Xm verschieben
y_data_shift = y_data - ym  # Y-Achse um Ym verschieben

Q = np.c_[x_data_shift,y_data_shift]        # Aus Ortsvekroten Matrix Q(nx2) bil
print('Verschiebung des Ursprungs: ( %0.2f | %0.2f )'%(xm,ym))

plt.plot(x_data_shift,y_data_shift,'+b')
plt.legend(['Daten (verschoben)'])
plt.xlabel('X - Achse')
plt.ylabel('Y - Achse')
plt.title('1.2 Ursprung verschieben')
plt.grid()
plt.show()
```

```
Verschiebung des Ursprungs: ( 1.19 | 0.70 )
```



1.3. SVD von Q berechnen und Gerade bestimmen

Die SVD gibt für eine Matrix Q der Form $\mathbb{R}^{n\times m}$ die Matrizen $U^{n\times n}$, $\Sigma^{n\times m}$ und V^T der Form $\mathbb{R}^{m\times m}$ zurück.

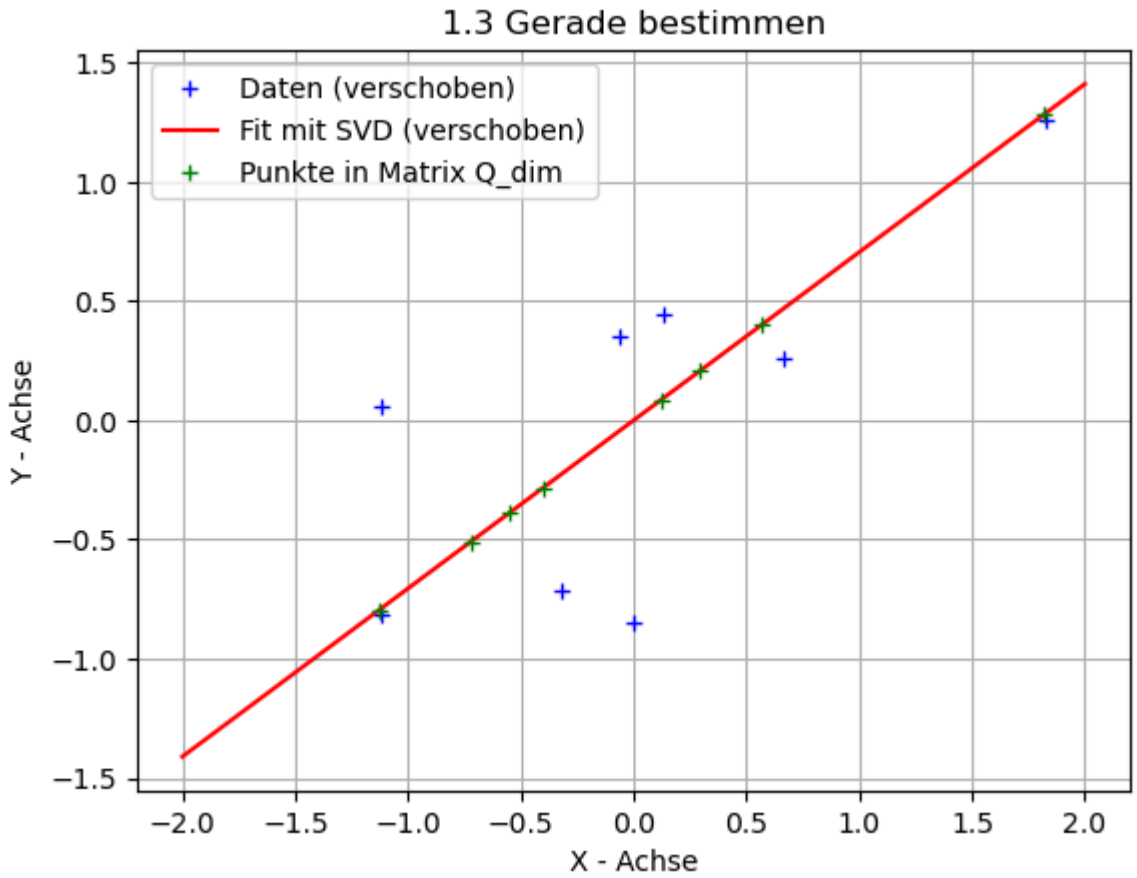
Mithilfe der Formel:

$$Q = \sum_{i=1}^{k} \sigma_i \mathbf{u_i} \mathbf{v_i^T}$$

kann die Punktwolke, welche in Q definiert wurde, für $k = 1$ auf eine Gerade, welche durch die Punkte in Q_dim definiert wird, reduziert werden (siehe güne Punkte in Plot). Um die Steigung dieser Gerade zu bestimmen, wurde die folgende Formel verwendet: $m = \frac{y_2 - y_1}{x_2 - x_1}$ wobei für x_1 bzw. y_1 der Ursprung 0 gewählt wurde und für x_1 und y_1 ein Punkt in Q_dim.

Frage: Wie muss die Punktwolke beschaffen sein, dass der erste Singulärwert (SV) viel grösser ist als der zweite?
Antwort: Damit der erste SV grösser wird, müssen die Punkte der Punktwolke den quadratisch, orthogonalen Fehler zueinander verkleinern.

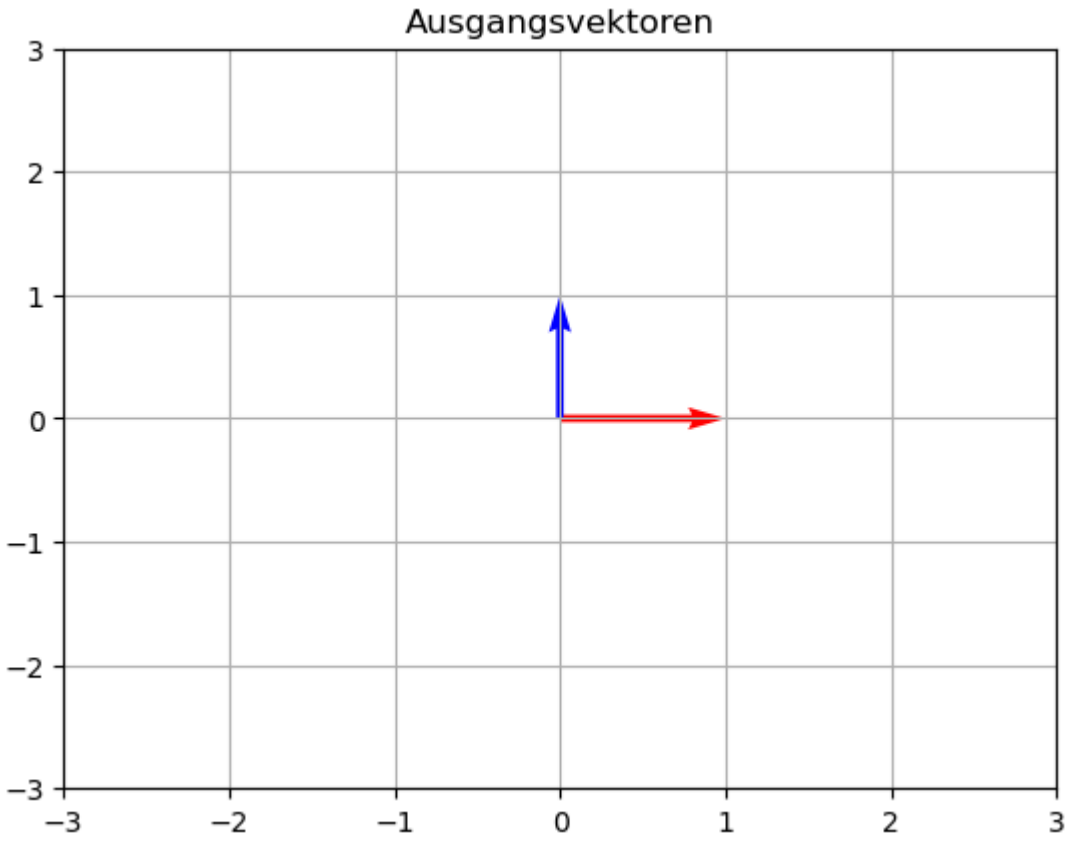
In [ ]:
```
U, ESW, VT = svd(Q,full_matrices=True)      # SVD von Matrix Q

Q_dim = ESW[0]*(np.row_stack((U[:,0]))@np.column_stack((VT[0,:])))          #

x_shift = np.linspace(-2,2,1024)            # x für plot
m = Q_dim[0,1]/Q_dim[0,0]                   # Steigung der Gerade
y_shift = x_shift*m                         # y für plot


# Plot
plt.plot(x_data_shift,y_data_shift,'b+',x_shift,y_shift,'r-',Q_dim[:,0],Q_dim[:,

plt.legend(['Daten (verschoben)','Fit mit SVD (verschoben)','Punkte in Matrix Q_
plt.xlabel('X - Achse')
plt.ylabel('Y - Achse')
plt.title('1.3 Gerade bestimmen')
plt.grid()
plt.show()
```

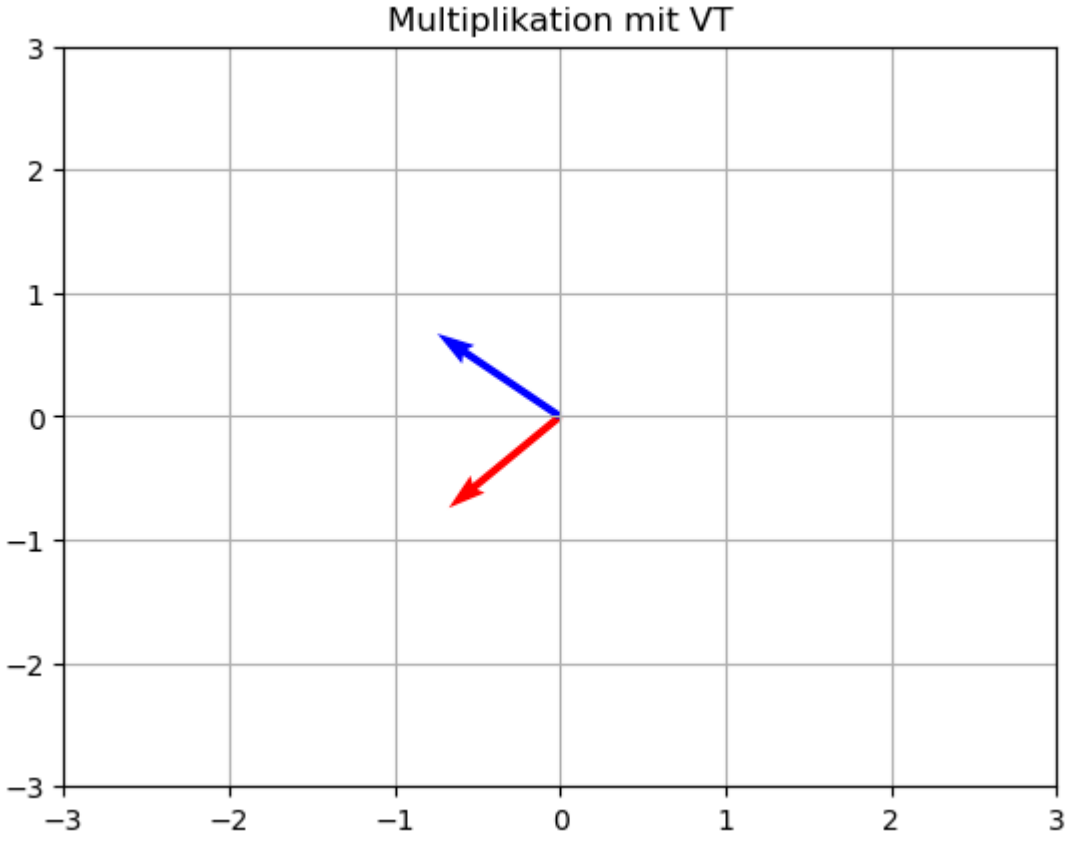


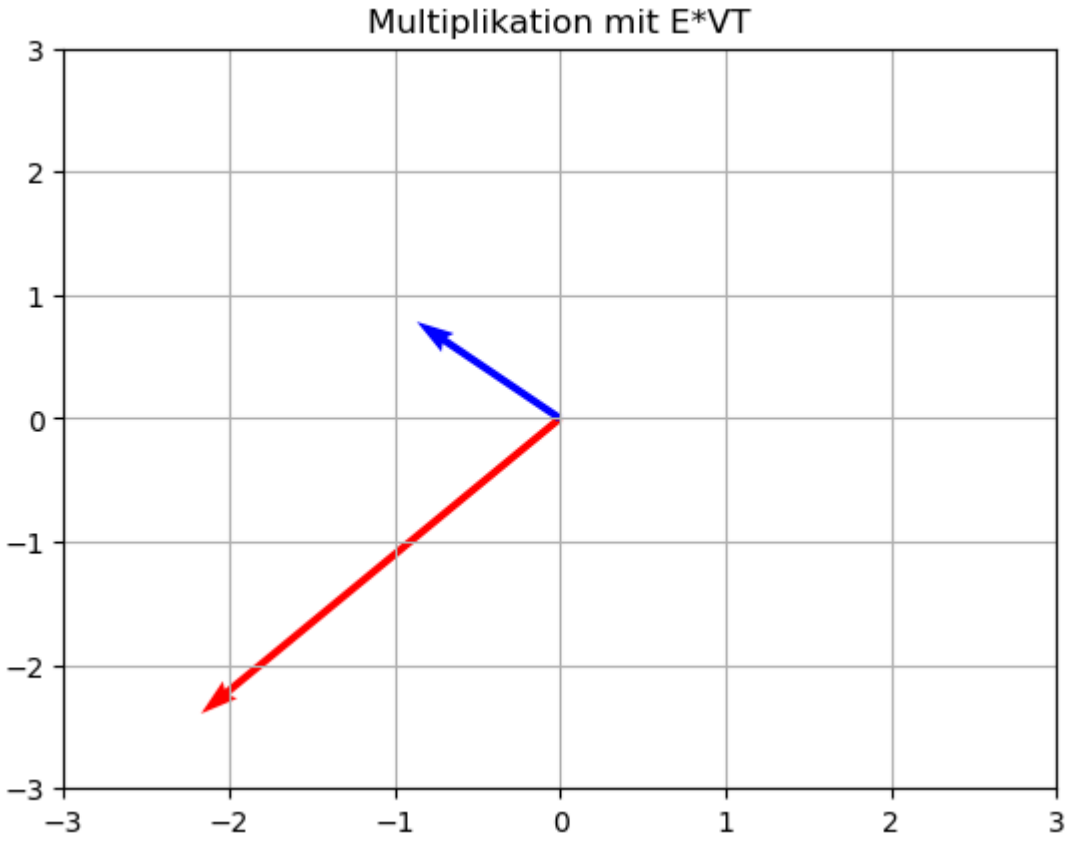
Einschub: Geometrische Visualisierung der SVD

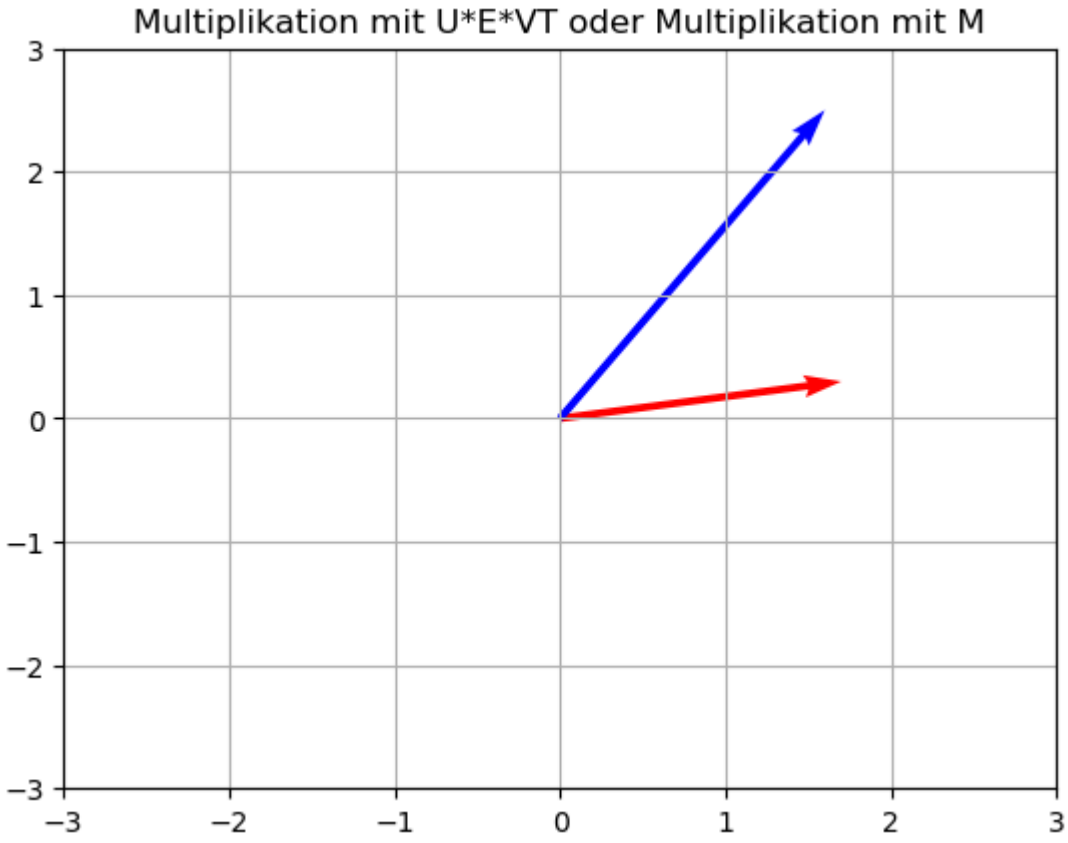
Die SVD zerlegt eine beliebige Matrix M der Form $\mathbb{R}^{n \times m}$ in 3 Matrizen $U\Sigma V^T$. Dadurch kann die transformation von M auf einen Vektor $\vec{v}$ durch die Drehmatrizen $U \in \mathbb{R}^{m \times m}$ und $V^T \in \mathbb{R}^{n \times n}$ und die Skalierungsmatrix $\Sigma \in \mathbb{R}^{m \times n}$ ausgedrückt werden.
Gutes Video zur Visualisierung der SVD: https://www.youtube.com/watch?v=vSczTbgc8Rc&ab_channel=VisualKernel

In [ ]:
```
origin = np.array([[0, 0],[0, 0]])

M = np.array([[1.7,0.3],[1.6,2.5]])
U,ESW,VT = svd(M)
E = np.diag(ESW)

V1 = np.array([1,0])
V2 = np.array([0,1])

plt.quiver(*origin, V1, V2, color=['r','b'],angles='xy', scale_units='xy', scale
plt.title('Ausgangsvektoren')
plt.xlim(-3, 3)
plt.ylim(-3, 3)
plt.grid()
plt.show()

plt.quiver(*origin, VT@V1, VT@V2, color=['r','b'],angles='xy', scale_units='xy',
plt.title('Multiplikation mit VT')
plt.xlim(-3, 3)
plt.ylim(-3, 3)
plt.grid()
plt.show()

plt.quiver(*origin, E@VT@V1, E@VT@V2, color=['r','b'],angles='xy', scale_units='
plt.title('Multiplikation mit E*VT')
plt.xlim(-3, 3)
plt.ylim(-3, 3)
plt.grid()
plt.show()

plt.quiver(*origin, U[[0,1],:]@E@VT@V1, U[[0,1],:]@E@VT@V2, color=['r','b'],angl
plt.title('Multiplikation mit U*E*VT oder Multiplikation mit M')
plt.xlim(-3, 3)
plt.ylim(-3, 3)
plt.grid()
plt.show()
```


Ausgangsvektoren
3
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
3


Multiplikation mit VT
3
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
3





1.4. Verschiebung des Ursprungs rückgängig machen

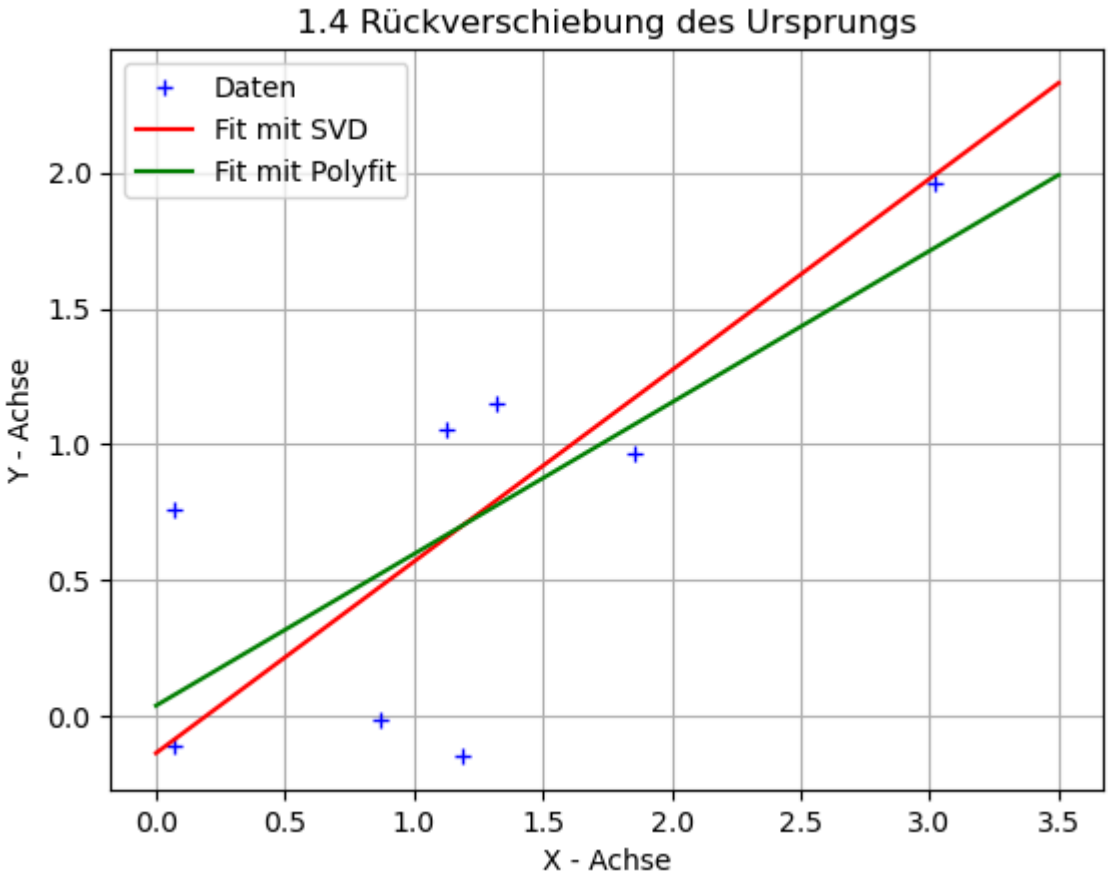
Um die Verschiebung rückgängig zu machen, muss die Gerade um $q = -(x_m * m - ym)$ in Y-Richtung verschoben werden. Die resultierende Gerade ist von der Form $f(x) = mx + q$.

Anschliessend wird der Fit mit SVD mit dem Fit der Numpy-Funktion polyfit verglichen. Der SVD-Fit beschreibt die Gerade mit dem orthogonalen kleinsten Fehlerquadrat zu den Punkten. Der Polyfit beschreibt die Gerade mit dem minimal quadratischen Abstand in Y-Richtung zu den Punkten.

Um den Fit mit SVD zu kontrollieren, wurde die SciPy-Funktion ODR zur Überprüfung verwendet. Die Funktion gibt die gleichen Koeffizienten zurück wie von der SVD berechnet.

In [ ]:
```
x = np.linspace(0,3.5,1024)             # x für plot
q = -(xm*m-ym)                          # Offset q der Geradengleichung
y = m*x + q                             # y für plot

print('Die Komponenten der Geradengleichung mit SVD: m = %0.3f, q = %0.3f'%(m,q)

# Vergleich Koeffizientem mit SciPy-Funktion ODR (nicht aktiviert)
def target_function(p, x):
    m, c = p
    return m*x + c
odr_model = scipy.odr.Model(target_function)
data_sp = scipy.odr.Data(x_data, y_data)
ordinal_distance_reg = scipy.odr.ODR(data_sp, odr_model,beta0=[0.2, 1.])
out = ordinal_distance_reg.run()
# out.pprint()

# Vergleich mit Numpy-Funktion polyfit
a = polyfit(x_data,y_data,1)
y_poly = a[0]*x + a[1]

plt.plot(x_data,y_data,'b+',x,y,'r-',x,y_poly,'g-')
plt.legend(['Daten','Fit mit SVD','Fit mit Polyfit'])
plt.xlabel('X - Achse')
plt.ylabel('Y - Achse')
plt.title('1.4 Rückverschiebung des Ursprungs')
plt.grid()
plt.show()
```

```
Die Komponenten der Geradengleichung mit SVD: m = 0.705, q = -0.138
```



2 Gesichtserkennung in Graustufenbilder

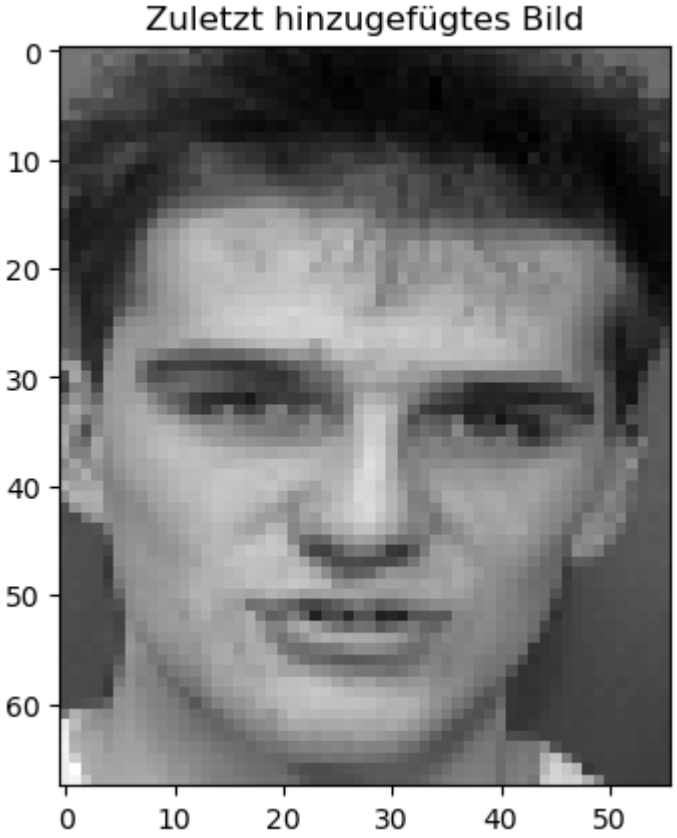
Eine Anwendung der SVD ist die Gesichserkennung. Ausgangslage ist eine Sammlung von Beispielgesichtern, zu finden im Verzeichnis 'material'. Mit Hilfe dieser Daten kann der zu entwickelnde Algorithmus ‘lernen’, Gesichter zu erkennen. Mit Standardfunktionen (matlab: imread() oder python: matplotlib.pyplot.imread()), werden Bilddateien als Matrizen eingelesen: Jedes Graustufenpixel entspricht einem Matrixelement. Bevor diese Bildinformation mit Hilfe der SVD genutzt werden kann, müssen die Matrizen in Vektoren umgewandelt werden, indem deren Spalten hintereinander platziert einen Zeilenvektor der Dimension $3808 = 56 \cdot 68$ bilden. Diese Beispielgesichter sollen als Ortsvektoren in einem Vektorraum der Dimension 3808 vorliegen. Dort bevölkern diese Vektoren jedoch nur einen ‘kleinen’ Unterraum (dazu wird wiederum den Mittelwert aller Beispielvektoren subtrahiert, um das Zentrum dieser Punktwolke in den Ursprung zu verschieben). Zudem erwartet man, dass andere Gesichter, welche nicht Teil der Beispielmenge sind, ebenfalls ‘einen grossen Überlapp’ mit diesem Unterraum aufweisen. Somit können Gesichtsbilder und Nicht-Gesichtsbilder anhand des ‘Überlapps mit dem Unterraum’ der Beispielbilder unterschieden werden. Die Methode zur Identifikation des relevanten Unterraums ist wiederum die SVD, nun angewendet auf eine sehr grosse Matrix.

2.1. Vergleichsmatrix P erstellen

Um die Matrix P zu erstellen, wurden 300 der 370 Bilder eingelesen. Die Matrizen der einzelnen Bilder wurden in einen Zeilenvektor geschrieben und zeilenweise der Matrix P hinzugefügt.

In [ ]:
```
# Parameter zum Bilder einlesen
PATH = "material/s%d/f%d.png"                    # Pfad für Bilder

zeile = 68                                       # Anz Zeilenpixel Bild
spalte = 56                                      # Anz Spaltenpixel Bild
pix = zeile*spalte                               # Anzahl Zeilen in P Matrix

s_n = 30                                         # Anzahl Ordner mit Fotos
f_n = 10                                         # Anzahl Fotos in Ordner
N = s_n*f_n                                      # Anzahl Fotos insgesamt
s = np.arange(1,s_n+1,1)                         # Array für Anzahl Ordner
f = np.arange(1,f_n+1,1)                         # Array für Anzahl Fotos in Ordn


# Liest ein Bild ein und transformiert es in einen Zeilenvektor (i = Ordnernumme
def read_imag(i,j):
    img = PATH%(i,j)                             # Pfad zu spezifischem Foto
    Ptemp = plt.imread(img)                      # Foto in Matrix importieren
    Ptemp = to_rowvector(Ptemp)                  # Matrix in Zeilenvektor umwande
    return Ptemp


# Transformiert Matix zu Zeilenvektor
def to_rowvector(M):
    return M.flatten()


# Transformiert Zeilenvektor zurück in Matrix zum plotten
def to_imag(x):
    return np.reshape(x,([zeile,spalte]))


# Erstellt Matrix P aus den eingelesenen Bildern
P = np.zeros((1,pix))                            # 0-Array zum initialisieren der
for i in s:
    for j in f:
        img = PATH%(i,j)                         # Pfad zu spezifischem Foto
        Ptemp = plt.imread(img)                  # Foto in Matrix importieren
        Ptemp = to_rowvector(Ptemp)              # Matrix in Zeilenvektor umwande
        P = np.vstack((P,Ptemp))                 # Zeilenvektor zu Matrx P hinzuf
P = np.delete(P,0,0)                             # Initialisierte 1. Zeile lösche

print('Dimension der Matrix P:',P.shape)
print('Anzahl Elemente in Matrix P:',P.size)

# Zeigt letztes Foto, dass eingelesen wurde (zur Kontrolle)
plt.imshow(to_imag(Ptemp),cmap='gray')
plt.title('Zuletzt hinzugefügtes Bild')
plt.show()
```

```
Dimension der Matrix P: (300, 3808)
Anzahl Elemente in Matrix P: 1142400
```



2.2. Mittelwert der Vergleichsbilder, SVD der Matrix P, Grösse des Bildunterraumes und durchschnittliche Abweichung von Vergleichsbildern zum Bildunterraum

Um das Durchschnittliche Bild zu berechnen, wurde folgende Formel verwendet:

$$\overline{\mathbf{p}} = \frac{1}{N}\sum_{i=1}^{N}\vec{p}_i$$

wobei p_i die Spaltenvektoren der Matrix P sind.

Um den Ursprung in die Mitte zu verschieben, wurden alle Zeilenvektoren der Matrix P mit $\overline{p}$ subtrahiert. Anschliessend wurde die SVD auf die gemittelte Matrix P angewendet.

Durch einige Versuche mit der Grösse des Bildunterraums wurde die Grösse als 40 definiert.

Um einen Vergleichswert zwischen Bildern zu berechnen, wurde die Norm der quadratischen Abweichung verwendet:

$$\epsilon_i = \|V_{dim}(\mathbf{x} - \overline{\mathbf{p}})\|$$

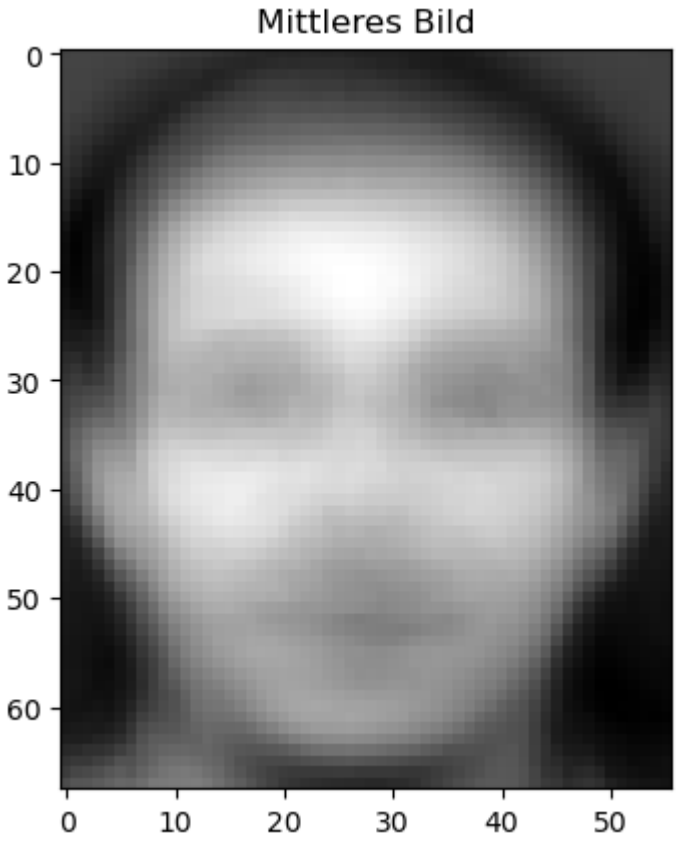
wobei V_{dim} die ersten 40 Spaltenvektoren der Matrix V^T aus der SVD ist, $\mathbf{x}$ der Zeilenvektor des eingelesenen Bildes und $\overline{\mathbf{p}}$ das berechnete "Mittlere Bild".

Aus dem Set der Vergleichsbilder wurden von allen Bildern die Norm der quadratischen Abweichung gebildet und der minimale und maximale Wert ermittelt:

$$\epsilon_{min} = 5.548, \epsilon_{max} = 12.329$$

In [ ]:

```
# Mittleres Bild
P_mean = np.zeros(pix)                          # Nullvektor initialisieren
for i in range(0,N):                            # gemäss Formel
    P_mean = P_mean + P[i,:]
P_mean = 1/N * np.column_stack(P_mean)


# Shift
P_shift = P
for i in range(0,N):
    P_shift[i,:] = P_shift[i,:] - P_mean


# Plot des mittleren Bildes
plt.imshow(to_imag(P_mean[0,:]),cmap='gray')
plt.title('Mittleres Bild')
plt.show()


# SVD
U,ESW,VT = svd(P_shift)


# Quadratische Abweichung Vergleichbilder
k = 40                                          # Grösse des Unterraumes
V_dim = VT[range(0,k),:]                        # Verminderter Unterraum
x_comp = V_dim@P_shift.T
e_comp = []
for i in range(0,N):                            # Array mit minimalen Abständen
    e_comp.append(norm(x_comp[:,i]))


print('Die Abweichungen der Vergleichsbilder zum Bildunterraum befinden sich im


# Funktion für quadratische Abweichung zum Bildunterraum
def deviation (M):
    M = np.column_stack(M - P_mean)
    J = V_dim@M
    return norm(J)
```




```
Die Abweichungen der Vergleichsbilder zum Bildunterraum befinden sich im Bereich
von: e_max = 12.329 und e_min = 5.548.
```

2.3. Threshold ϵ_0 definieren und Bilder auswerten

Um die eingelesenen Bilder in "Bild mit Gesicht" und "Bild ohne Gesicht" zu kategorisieren, muss ein Threshold definiert werden, ab dem die Abweichung zum Gesichtsunterraum zu gross ist, um ein Gesicht zu sein.

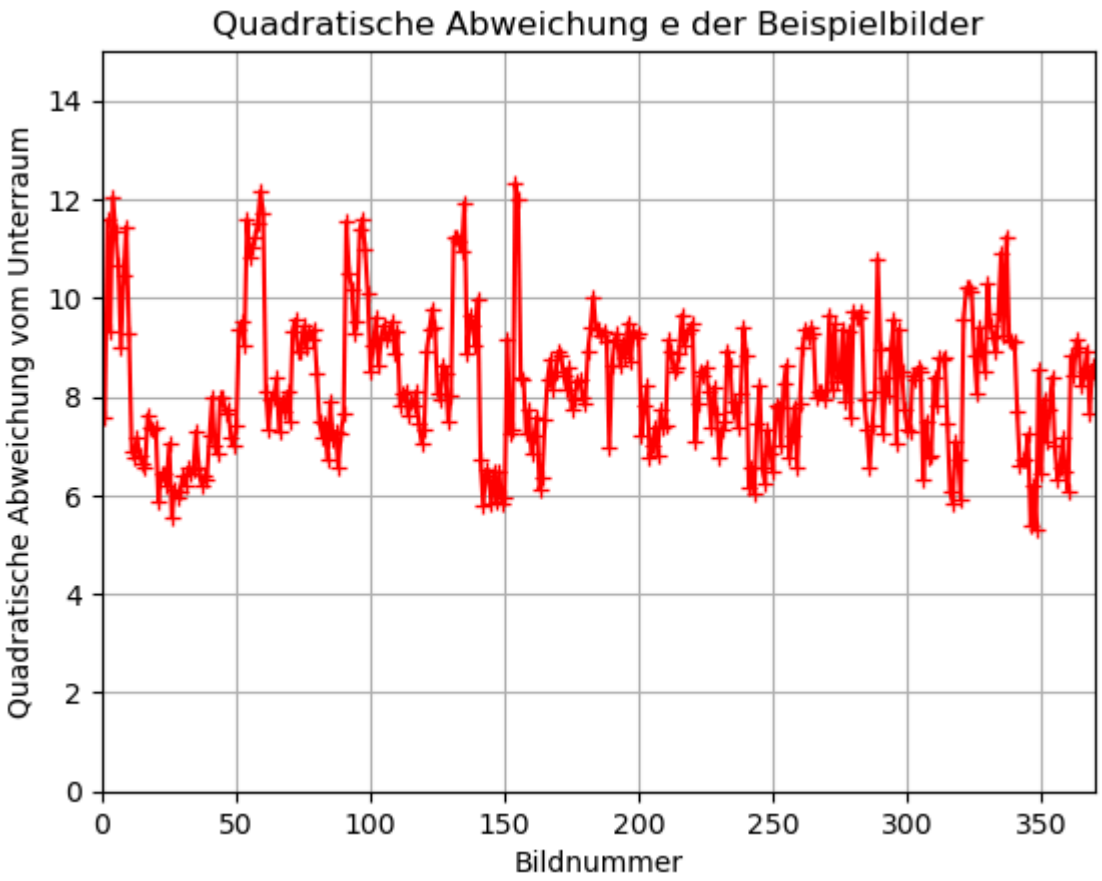
Um zu ermitteln, welche Abweichung Bilder mit Gesicht haben, wurde die Abweichung aller 370 beispielbilder berechnet und in der unteren Grafik geplottet ($\epsilon_{min} = 5.296$, $\epsilon_{max} = 12.329$). Als Beispiel für ein Bild ohne Gesicht wurde ein Bild einer Tulpe eingelesen und ϵ dieses Bildes bestimmt ($\epsilon_{Tulpe} = 22.269$). Auf Basis dieser Werte wurde der Threshold $\epsilon_0 = 14$ gesetzt.

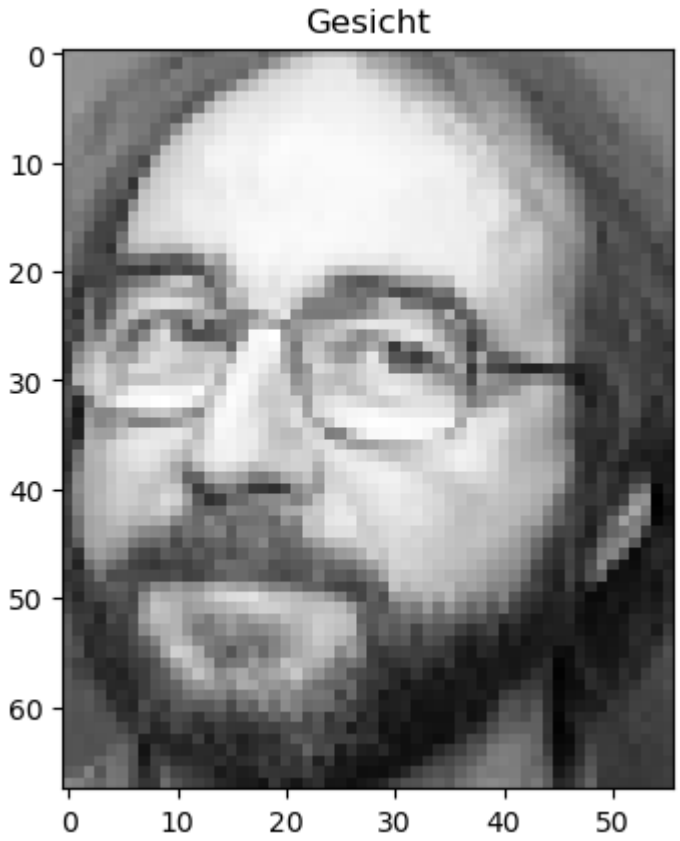
Anschliessend wurde eine Funktion implementiert, welche Bilder als Zeilenvektoren als Input hat und, wenn das Bild ein Gesicht hat, "Gesicht" zurückgibt und sonst "Kein Gesicht".

In [ ]:
```
e_0 = 14                    # Definierter Schwellenwert für Gesicht


# Tulpenbild einlesen und quadratische Abweichung berechnen
e_t = deviation(read_imag(38,1))
print('Das Tulpenbild hat einen Wert von e_t = %.3f'%(e_t))


# Gibt zurück, ob Bild ein Gesicht ist oder nicht
def is_it_a_face(M):
    e = deviation(M)
```

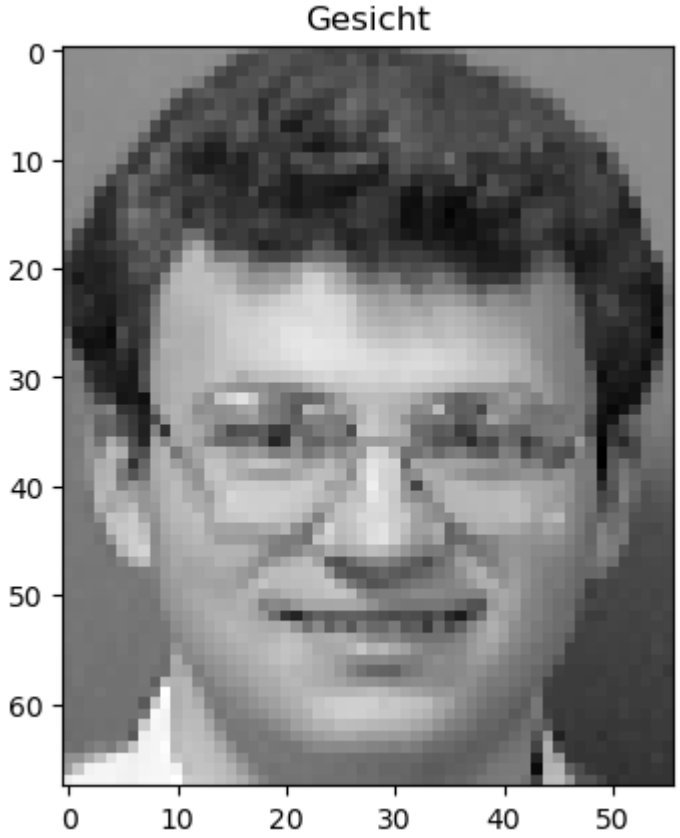
```
    # Wenn Bild ist unter Threshold, Ausgabe "Gesicht"
    if e < e_0:
        plt.imshow(to_imag(M),cmap='gray')
        plt.title('Gesicht')
        plt.show()
        print('Das obige Bild hat einen Wert von e = %.3f < %.1f = e_0.\nEs ist
    # Wenn Bild ist unter Threshold, Ausgabe "Kein Gesicht"
    else:
        plt.imshow(to_imag(M),cmap='gray')
        plt.title('Kein Gesicht')
        plt.show()
        print('Das obige Bild hat einen Wert von e = %.3f > %.1f = e_0.\nEs ist


# Array mit quadratischen Abweichungen aller 370 Bilder
e_bsp = []
s = 37
f = 10
N_bsp = np.arange(1,s*f+1,1)
s = np.arange(1,s+1,1)
f = np.arange(1,f+1,1)
for i in s:
    for j in f:
        Ptemp = read_imag(i,j)
        e_bsp.append(deviation(Ptemp))
print('Die Abweichungen der "Bilder mit Gesicht" zum Bildunterraum befinden sich
print('Der Threshold als Grenzwert wird gesetzt als e_0 = %.1f.'%(e_0))

# Plot mit quadratischen Abweichungen aller Beispielbilder
plt.plot(N_bsp,e_bsp,'+-r')
plt.grid()
plt.title('Quadratische Abweichung e der Beispielbilder')
plt.xlabel('Bildnummer')
plt.ylabel('Quadratische Abweichung vom Unterraum')
plt.axis([0,370,0,15])
plt.show()


# Beispiele der Funktion "Is_it_a_face"
is_it_a_face(read_imag(28,10))
is_it_a_face(read_imag(6,2))
is_it_a_face(read_imag(38,1))
is_it_a_face(read_imag(35,5))
```

```
Das Tulpenbild hat einen Wert von e_t = 22.269
Die Abweichungen der "Bilder mit Gesicht" zum Bildunterraum befinden sich im Bere
ich von: e_max = 12.329 und e_min = 5.296.
Der Threshold als Grenzwert wird gesetzt als e_0 = 14.0.
```

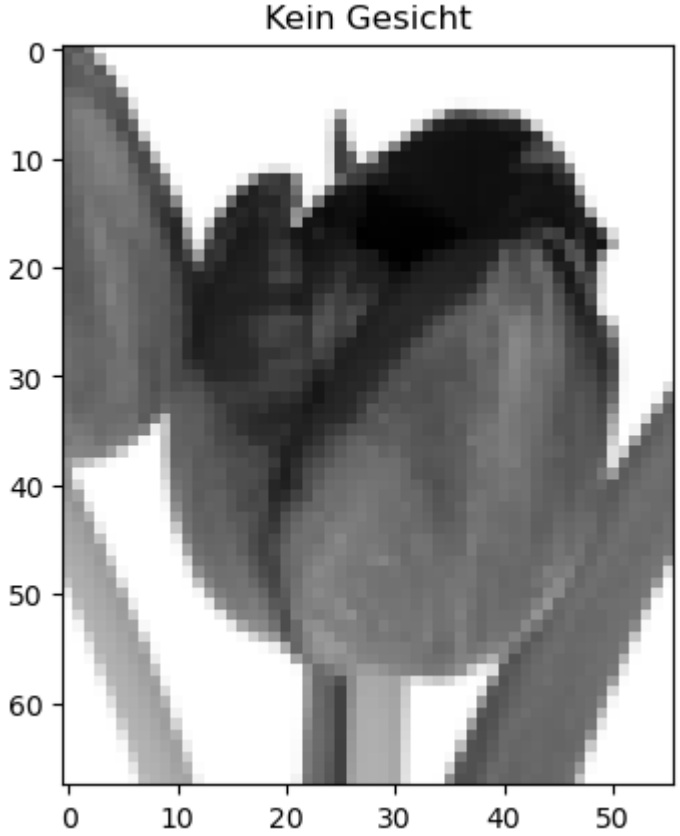




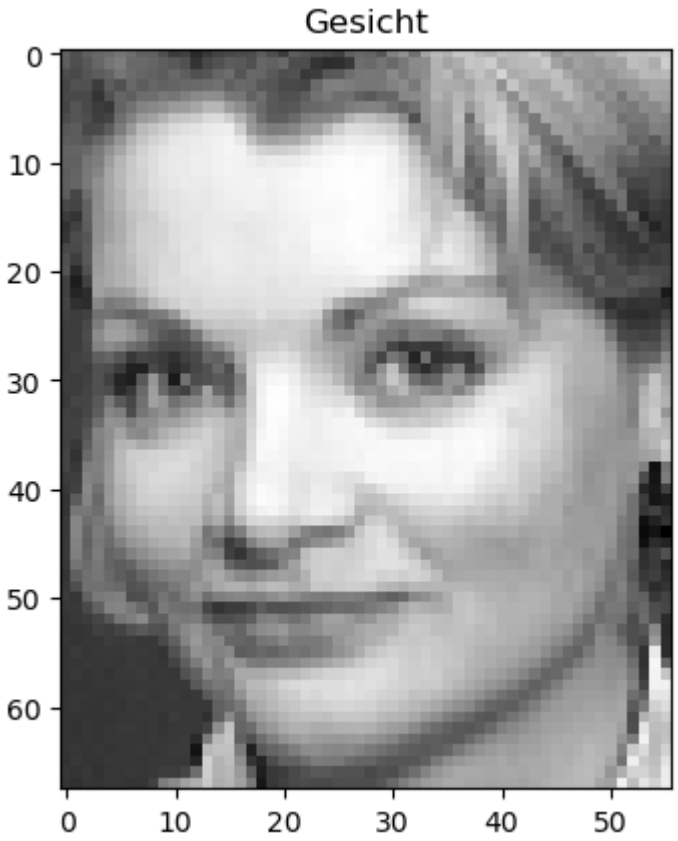
Das obige Bild hat einen Wert von e = 9.738 < 14.0 = e_0.
Es ist also ein Gesicht.



Das obige Bild hat einen Wert von e = 9.529 < 14.0 = e_0.
Es ist also ein Gesicht.



Das obige Bild hat einen Wert von e = 22.269 > 14.0 = e_0.
Es ist also kein Gesicht.



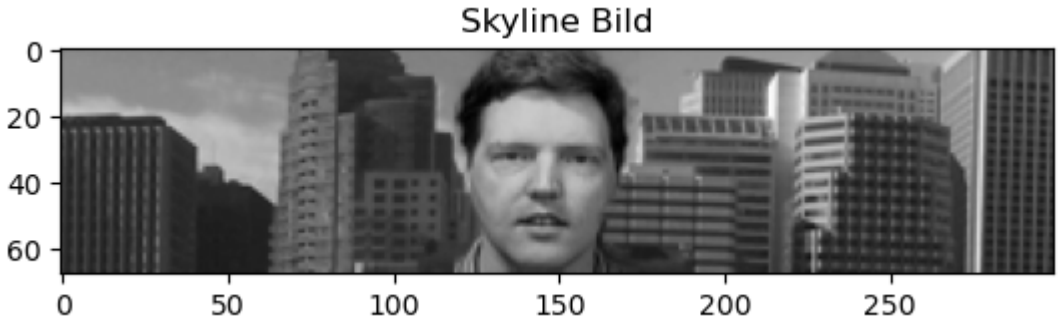

```
Das obige Bild hat einen Wert von e = 7.237 < 14.0 = e_0.
Es ist also ein Gesicht.
```

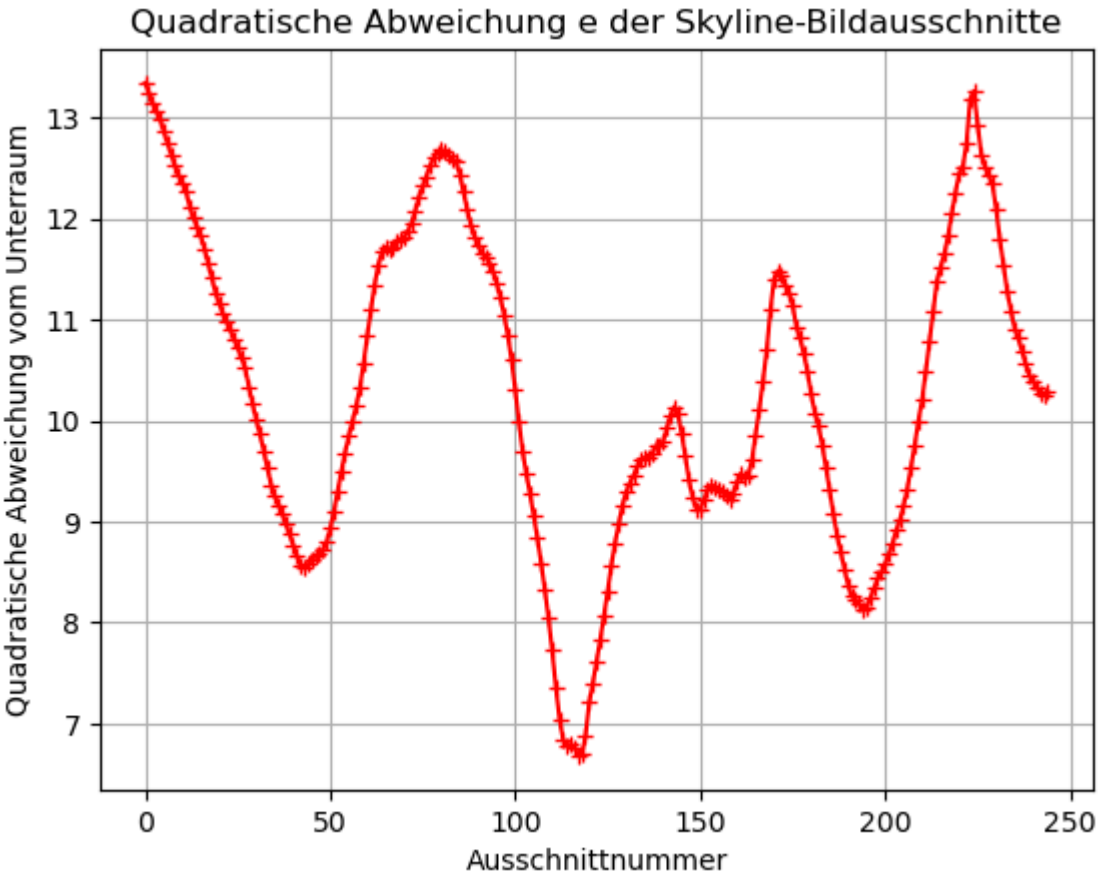
2.4. Gesicht in Skyline

Um das Gesicht im Skyline-Bild zu finden, wurde das Bild zuerst eingelesen und anschliessend ein Algorithmus definiert, welcher von links nach recht einen Bildausschnitt 68 x 56 Pixel ausschneidet und ϵ dieses Ausschnittes berechnet. Danach wird der Bildausschnitt ein Pixel nach rechts geschoben, bis das ganze Bild eingelesen wurde. Anschliessen werden alle ϵ zu ihrer Iteration der Verschiebung des Bildausschnittes geplotted. Bei der Iteration, welche das kleinste ϵ besitzt. Da sollte das Gesicht im Bild sein.

In [ ]:
```
# Skyline Bild einlesen und plotten
img = PATH%(39,1)                               # Pfad zu spezifischem Foto
SkyL = plt.imread(img)                          # Foto in Matrix importieren

plt.imshow(SkyL,cmap='gray')
plt.title('Skyline Bild')
plt.show()


# Quadratische Abweichung aller Bildausschnitte in Array
e_SkyL_i = []
for i in np.arange(0,245,1):
    SkyL_sec = SkyL[:,np.arange(0+i,56+i)]
    SkyL_sec = to_rowvector(SkyL_sec)
    e_SkyL_i.append(deviation(SkyL_sec))
e_SkyL_i = np.array(e_SkyL_i)
```

```
# Plot der quadratischen Abweichung
plt.plot(np.arange(0,e_SkyL_i.size),e_SkyL_i,'+-r')
plt.grid()
plt.title('Quadratische Abweichung e der Skyline-Bildausschnitte')
plt.xlabel('Ausschnittnummer')
plt.ylabel('Quadratische Abweichung vom Unterraum')
plt.show()


# Minimum der Funktion bestimmen und Index zurückgeben
e_SkyL_min_i = np.argmin(e_SkyL_i)
e_SkyL_min = np.amin(e_SkyL_i)
SkyL_face = SkyL[:,np.arange(0+e_SkyL_min_i,56+e_SkyL_min_i)]

print('Die tiefste Ausschnitt-Abweichung e = %.3f befindet sich im Skylinebild i
print('Das Gesicht befindet sich demnach im %d. Ausschnitt:'%(e_SkyL_min_i))

# Bildausschnitt des Gesichtes plotten
plt.imshow(to_imag(SkyL_face),cmap='gray')
plt.title('Gesicht in Skylinebild')
plt.show()
```





```
Die tiefste Ausschnitt-Abweichung e = 6.682 befindet sich im Skylinebild im 117.
Ausschnitt.
Das Gesicht befindet sich demnach im 117. Ausschnitt:
```

Fazit und Ausblick

Die SVD bietet eine weitere Möglichkeit, Fehlerabweichungen zu berechnen und daraus eine Mittelwertsgerade zu bilden. Es kommt auf den Anwendungsfall an, welche Methode bzw. welches Abstands-Fehlerquadrat (vertikale, horizontale, orthogonale) am besten geeignet ist. Eine Anwendung ist die Gesichtserkennung. Diese wurde hier relativ simpel durchgeführt, mit tiefen Auflösungen und Graustufen-Bilder, dafür in diesem Umfang fehlerlos korrekt. Grundsätzlich kann das auch erweitert werden: Hochauflösendere Bilder führen "nur" zu einer grösseren Matrix P, wie auch ein besseres "Lernen" mit mehr eingelesenen Beispielbildern. Allerdings dauert die Berechnung und SVD der Matrix P bereits jetzt einige Sekunden und könnte dadurch einiges an Zeit kosten, welche beispielsweise nicht akzeptabel ist für eine schnelle Gesichtserkennung beim Fokussieren einer Kamera. Zudem könnte das Verfahren auch auf Farbbilder angewendet werden, indem anstatt Graustufen die Hex-Farbcodes ausgewertet werden.