## January 13, 2020

## 1 Monte-Carlo-Simulation

```
© Thomas Robert Holy 2019 Version 1.1.1 Visit me on GitHub: https://github.com/trh0ly ## Grundlegende Einstellungen: Zunächst müssen die notwendigen Pakete (auch Module) importiert werden,
damit auf diese zugegriffen werden kann.
In [1]: import pandas as pd # Programmbibliothek die Hilfsmittel für die Verwaltung von Daten und deren Analyse anbietet
        import scipy.stats as st # SciPy ist ein Python-basiertes Ökosystem für Open-Source-Software für Mathematik, Naturwissenschaften und Ingenieurwissenschaften
        from scipy.stats import rankdata, norm
        from scipy import array, linalg, dot
       import random # Dieses Modul wird verwendet um Zufallszahlen zu ziehen
```

import numpy as np # Programmbibliothek die eine einfache Handhabung von Vektoren, Matrizen oder generell großen mehrdimensionalen Arrays ermöglicht import math # Dieses Modul wird verwendet um Skalardaten zu berechnen, z. B. trigonometrische Berechnungen. import operator # Programmbibliothek, welche die Ausgaben übersichtlicher gestaltet

import matplotlib.pyplot as plt # Programmbibliothek die es erlaubt mathematische Darstellungen aller Art anzufertigen import matplotlib.patches as mpatches from riskmeasure\_module import risk\_measure as rm from IPython.core.display import display, HTML Anschließend werden Einstellungen definiert, die die Formatierung der Ausgaben betreffen. Hierfür wird das Modul operator genutzt. Außerdem wird die Größe der Grafiken modifiziert, welche später angezeigt werden sollen.

In [2]: %%javascript IPython.OutputArea.auto\_scroll\_threshold = 9999; <IPython.core.display.Javascript object>

SCREEN\_WIDTH = 115 centered = operator.methodcaller('center', SCREEN\_WIDTH)

In [3]: display(HTML("<style>.container { width:100% !important; }</style>"))

plt.rcParams["figure.figsize"] = 15,15 <IPython.core.display.HTML object>

1.1 Monte-Carlo Simulation: 1.1.1 Variablen spezifizieren

Im ersten Schritt werden die für die Simulation notwenigen Variablen definiert.

#-----# Anzahl Simulationsdurchläufe

n = 10000#-----# Neue Randverteilungen (Gleichverteilung)

 $rand_x = [10,20]$  $rand_y = [8,22]$ # Varianzen und Korrelation(en)

 $var_x = 4$ 

 $var_y = 9$ corr\_list = [0] std\_list = [math.sqrt(var\_x), math.sqrt(var\_y)]

#-----# Erwartungswerte

mu = [2, 3]

1.1.2 Funktionen definieren Als nächstes werden Funktionen definiert. Die erste Funktion berechnet aus den gegeben Varianzen und dazugehörigen Korrelationen die Varianz-Kovarianz-Matrix, die Zweite legt ein Array mit den gegebenen Varianzen an und die Dritte führt die Cholesky-Zerlegung auf der Grundlage Varianz-Kovarianz-Matrix durch. Die weiteren Funktionen vereinfachen das Plotten.

In [5]: #-----# Argumente:

# Definition einer Funktion, welche eine Varianz-Kovarianz-Matrix erstellt # - std\_list: Liste mit Standardabweichungen # - corr\_list: Liste mit Korrelationskoeffizienten

def var\_covar\_matrix\_func(std\_list, corr\_list): counter\_0, counter\_1 = 0, 0 len\_std\_list = len(std\_list) array = [[0] \* len\_std\_list] \* len\_std\_list

val\_list = [] # Für jedes i und j in len\_std\_list.. for i in range(0,len\_std\_list): for j in range(0,len\_std\_list): # Wenn i = j, dann multipliziere beide Werte if i == j: val = std\_list[i] \* std\_list[i] val\_list.append(val) # Wenn i kleiner j, dann multipliziere beide Standardabweichungen # und die dazugehörige Korrelation if i < j: val = (std\_list[i] \* std\_list[j] \* corr\_list[counter\_0]) counter\_0 += 1 val\_list.append(val) # Wenn i größer j, dann multipliziere beide Standardabweichungen # und die dazugehörige Korrelation if i > j: val = (std\_list[i] \* std\_list[j] \* corr\_list[counter\_1]) counter\_1 += 1 val\_list.append(val) var\_covar = np.array(val\_list).reshape(len\_std\_list, len\_std\_list) return var\_covar #\_\_\_\_\_\_ # Definition einer Funktion, welche ein Varianz-Array erstellt # Argumente: # - std list: Liste mit Standardabweichungen #---def var\_func(std\_list): var\_list = [] for i in range(0, len(std\_list)): var = np.power(std\_list[i],2) var\_list.append(var) return var\_list

#------# Definition einer Funktion, welche die Cholesky-Zerlegung durchführt # Argumente: # - var\_covar: Varianz-Kovarianz-Array #---def cholesky\_func(var\_covar): cholesky = linalg.cholesky(var\_covar, lower=True) return cholesky # Hilfsfunktionen zum Plotten

#-----# Definition einer Funktion, welche eine gegebene Liste mit 2er-Tuples in X- und Y-Realisationen splittet # - liste: Liste mit 2er-Tuple #---def split\_liste(liste): counter = 0 x\_liste, y\_liste = [], [] for i in range(0, len(liste)): x = liste[counter][0] y = liste[counter][1] x\_liste.append(x) y\_liste.append(y) counter += 1 return x\_liste, y\_liste #----# Definition einer Funktion, welche die Grafiken plottet # Argumente: # - x\_liste: x-Realisationen # - y\_liste:y-Realisationen # - show: # --> Wenn True: Grafik wird unmittelbar darstellt # --> Wenn False: Grafik wird nicht unmittelbar darstellt # --> Wenn True: X- und Y- Kooridnaten werden ermittelt und die Grafik wird darauf begrenzt # --> Wenn False: Keine Anpassung der Koordinatenachsen def plot\_func(x\_liste, y\_liste, show=True, get\_xy\_lim=True): plt.scatter(x\_liste, y\_liste)

plt.ylim(left, right) plt.grid() plt.xlabel('Realisation X') plt.ylabel('Realisation Y') plt.axhline(0, color='black') plt.axvline(0, color='black') if show == True: plt.show() #-----# Definition einer Funktion, ein Histogramm plottet def hist\_func(H, X1): dx = X1[1] - X1[0]F1 = np.cumsum(H) \* dxplt.plot(X1[1:], F1)  ${\it\# Definition \ einer \ Funktion, \ welche \ eine \ Verteilungsfunktion \ plottet}$ 

plt.title('Verteilung X+Y, Realisationen gleichverteilte X,Y mit Gauss-Copula') blue\_patch = mpatches.Patch(color='blue', label='Monte-Carlo-Simulation(en)')

plt.legend(handles=[blue\_patch], loc='upper left')

if get\_xy\_lim == True:

def verteilung\_func():

plt.xlabel('PF-Realisation') plt.ylabel('Wahrscheinlichkeit')

# Listen für die zu berechnenden Größen

total\_xy\_list, total\_summe\_liste = [], []

var\_list = var\_func(std\_list)

cholesky = cholesky\_func(var\_covar)

counter\_cholesky\_1 = 0

counter\_standard\_norm\_list = 0

realisation\_cop\_list.append(p\_value)

counter += 1 if simulation\_runs == 1:

1.2 Graphische Ausgaben:

15

10

1.2.1 Realisationen abhängiger bivariat normalverteilter Zufallsvariablen

Die Zufallsvariablen x und y streuen mit der Varianz  $\sigma_x$  bzw.  $\sigma_y$  um ihren jeweiligen Erwartungswert  $\mu_x$  bzw.  $\mu_y$ .

plot\_func(x\_liste, y\_liste, show=True, get\_xy\_lim=True) # Plot erzeugen und anzeigen

In [9]: x\_liste, y\_liste = split\_liste(total\_standard\_norm\_ab\_list) # X- und Y-Realisationen aus gemeinsamer Liste extrahieren

plt.title('Realisationen abhängiger bivariat normalverteilter Zufallsvariablen') # Spezifischer Titel

# Durchführung der n Durchläufe

total\_standard\_norm\_ab\_list, total\_realisation\_cop\_list = [], []

var\_covar = var\_covar\_matrix\_func(std\_list, corr\_list)

left, right = plt.xlim() plt.xlim((left, right)) plt.xlim(left, right)

plt.grid()

left, right = plt.xlim() plt.xlim((left, right)) plt.xlim(left, right) left, right = plt.ylim() plt.ylim((left, right))

plt.axhline(0, color='black') plt.axvline(0, color='black') plt.show() 1.1.3 Funktion für die Monte-Carlo-Simulation definieren Nun wird eine Funktion definiert, in welcher die Simulation durchgeführt wird. Dabei werden zunächst zwei gleichverteilte Pseudo-Zufallsvariablen generiert, welche anschließend in unabhängige standardnormalverteilte Zufallsvariablen transformiert werden. Letztere werden unter Anwendung der Cholesky-Zerlegung in abhängige standardnormalverteilte Zufallsvariablen überführt. Danach werden diese unter Anwendung der oben gegebenen Standardabweichungen und der gegebenen Erwartungswerte in Realisationen der Gauss-Copula überführt, bevor auf diese anschließend die neuen Randverteilungen angewendet werden. Zum Schluss wird die Summe aus beiden Realisationen gebildet, welche den Portfolio-Wert ergibt. In [6]: # Definition einer Funktion, welche die Monte-Carlo-Sumulation durchführt # Argumente: # - simulation\_runs: Anzahl an Durchläufen der Simulation # - randverteilung\_x: Neue Randverteilung der Variablen X # - randverteilung\_y: Neue Randverteilung der Variablen Y # - mu\_list: Liste mit Erwartungswerten # - std\_list: Liste mit Standardabweichungen # - corr\_list: Liste mit Korrelationen # - m: Anzahl der Variablen die pro Lauf gezogen werden (hier gerade nur zwei möglich) # --> Wenn True: Alle berechneten Größen werden den Listen angefügt # --> Wenn False: Nur die Summe (das letztendliche Ergebnis der Simulation) der Realisationen wird einer Liste angefügt def copula\_sim(simulation\_runs, randverteilung\_x, randverteilung\_y, mu\_list, std\_list, corr\_list, m=2, full\_log=False):

for i in range(0, simulation\_runs): #------# Gleichverteilte Zufallsvariablen ziehen random\_ZV\_list = [] for i in range(0, m): x = random.random()random\_ZV\_list.append(x) if simulation\_runs == 1:

print('1) Gleichverteilte Zufallsvariablen: {}\n'.format(random\_ZV\_list))

# Funktionen aufrufen um die für die folgenden Berechnungen nötige Werte zu erhalten

#-----# Transformation der gleichverteilten Zufallsvariablen in unabhängige standardnormalverteilte Zufallsvariablen standard\_norm\_list = norm.ppf(random\_ZV\_list) if simulation\_runs == 1: print('2) Standardnormalverteilte Zufallsvariablen: {}\n'.format(standard\_norm\_list)) #------# Transformation in standardnormalverteilte abhängige Zufallsvariablen standard\_norm\_ab\_list = [] counter\_cholesky\_0, counter\_cholesky\_1 = 0, 0 counter\_mu, counter\_standard\_norm\_list = 0, 0 # Für jede zuvor gezogene unabhängige, standardnormalverteilte Zufallsvariable # wird die innere Abhänigkeitsstruktur (aus der Cholesky-Matrix) auf diese Zufallsvariable übertragen for i in range(0, m): # 1. Schritt: Berechnung a, mit a = 1. unabhängige standardnormalverteilte Zufallsvariable \* Eintrag in Cholesky-Matrix a = cholesky[counter\_cholesky\_0][counter\_cholesky\_1] \* standard\_norm\_list[counter\_standard\_norm\_list] counter\_cholesky\_1 += 1 counter\_standard\_norm\_list += 1 # 2. Schritt: Berechnung b, mit b = 2. unabhängige standardnormalverteilte Zufallsvariable \* Eintrag in Cholesky-Matrix b = cholesky[counter\_cholesky\_0][counter\_cholesky\_1] \* standard\_norm\_list[counter\_standard\_norm\_list] counter\_cholesky\_0 += 1

# 3. Schritt: Endergebnis r ist der Erwartungswert + a + b und stellt die abhängige standardnormalverteilte Zufallsvariable dar r = a + b + mu\_list[counter\_mu] standard\_norm\_ab\_list.append(r) counter\_mu += 1 if simulation\_runs == 1: print('3) Standardnormalverteilte abhängige Zufallsvariablen: {}\n'.format(standard\_norm\_ab\_list)) # Transformation Realisationen der Gauss-Copula counter = 0 realisation\_cop\_list = [] # Transformation indem von jeweiliger abhängiger standardnormalverteilter Zufallsvariablen # der dazugehörige Erwartungswert subtrahiert wird, und dieses Ergebnis dann durch die dazugehörige Standardabweichung geteilt wird # Die Realisation der Gauss-Copula ergibt sich dann als der Wert (Wahrscheinlichkeit) der kumulierten Normalverteilungsfunktion for i in range(0, m): r\_cop = (standard\_norm\_ab\_list[counter] - mu\_list[counter]) / math.sqrt(var\_list[counter]) p\_value = st.norm.cdf(r\_cop)

print('4) Realisationen der Gauss-Copula: {}\n'.format(realisation\_cop\_list))

# Gemeinsame Verteilung: Übertragung der neuen Ränder auf die ermittelte Abhängigkeitsstruktur # Für x und y wird jeweils (Obergrenze - Untergrenze) \* Realisation der Gauss-Copula + Untergrenze gerechnet # Dabei sind Obergrenze bzw. Untergrenze der linke bzw. rechte Rand des Intervalls der neuen Randverteilungen  $x = (randverteilung_x[0] - randverteilung_x[1]) * realisation_cop_list[0] + randverteilung_x[1]$ y = (randverteilung\_y[0] - randverteilung\_y[1]) \* realisation\_cop\_list[1] + randverteilung\_y[1] # Die Summe bzw. das Endergebnis der Simulation ist dann X + Y, also die gemeinsame Realisation summe = x + yif simulation\_runs == 1: print('5) Sumulationsergebnisse: x={}, y={}, Summe={}'.format(x, y, summe)) # Berechnete Werte werden den jeweiligen Listen angefügt if full\_log == True: total\_standard\_norm\_ab\_list.append(standard\_norm\_ab\_list) total\_realisation\_cop\_list.append(realisation\_cop\_list) total\_xy\_list.append((x, y)) total\_summe\_liste.append(summe) return total\_standard\_norm\_ab\_list, total\_realisation\_cop\_list, total\_xy\_list, total\_summe\_liste In [7]: \_, \_, \_ = copula\_sim(1, rand\_x, rand\_y, mu, std\_list, corr\_list, full\_log=False) 1) Gleichverteilte Zufallsvariablen: [0.40195872330818816, 0.5547450117792724] 2) Standardnormalverteilte Zufallsvariablen: [-0.24828042 0.13765893] 3) Standardnormalverteilte abhängige Zufallsvariablen: [1.503439158025408, 3.4129768012545965] 4) Realisationen der Gauss-Copula: [0.4019587233081881, 0.5547450117792724] 5) Sumulationsergebnisse: x=15.980412766918118, y=14.233569835090186, Summe=30.213982602008304 In [8]: total\_standard\_norm\_ab\_list, total\_realisation\_cop\_list, total\_xy\_list, total\_summe\_liste = copula\_sim(n, rand\_x, rand\_y, mu, std\_list, corr\_list, full\_log=True)

Realisationen abhängiger bivariat normalverteilter Zufallsvariablen



0.6

0.2

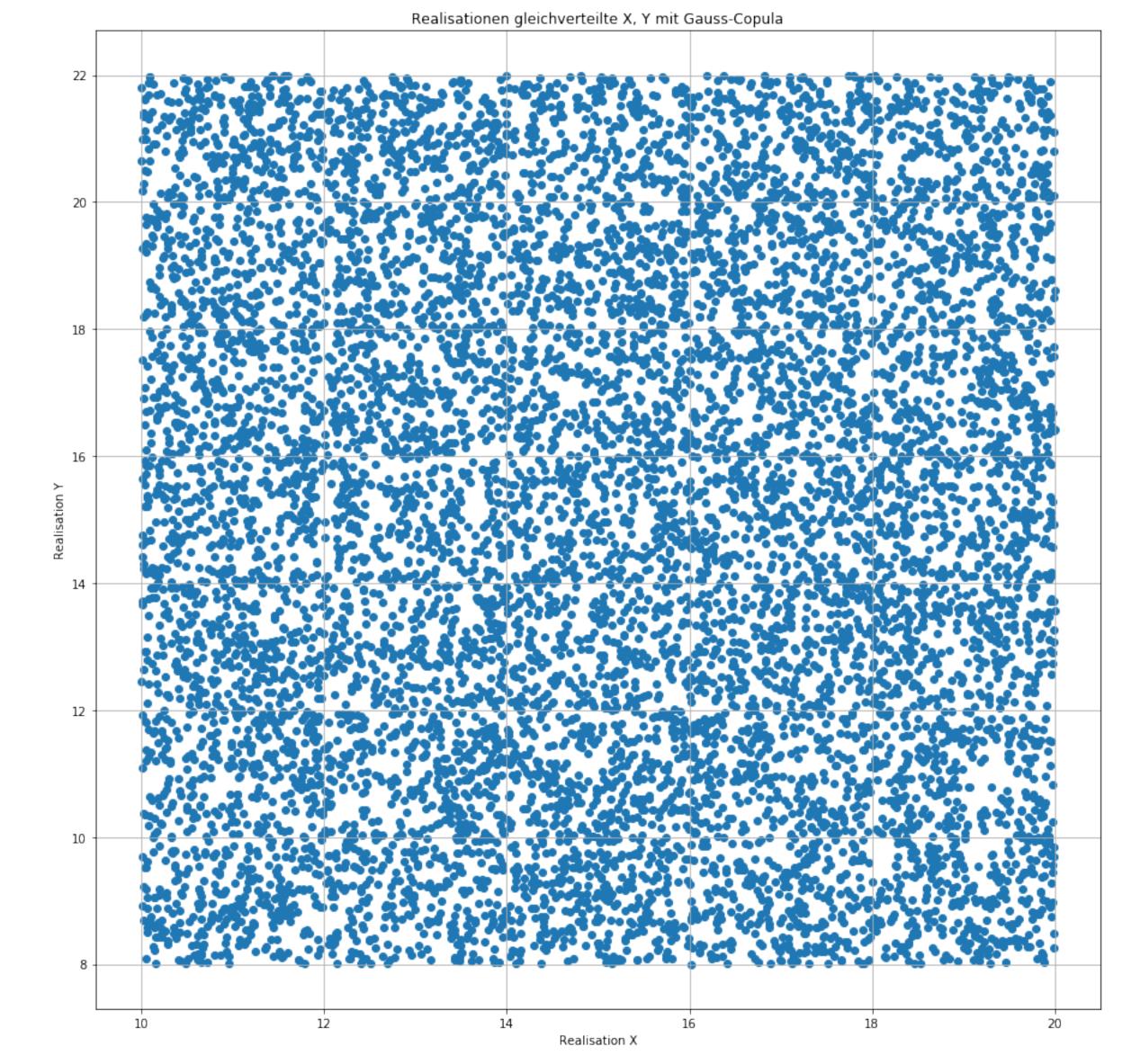
Realisation Y 0.4

0.0 0.0 0.2 0.4 0.6 0.8 Realisation X

1.0

Die innere Abhängikeitsstruktur wurde auf die neuen Randverteilungen übertragen. Die Realisationen sind auf das Intervall [18, 42] beschränkt (siehe rand\_y, rand\_y), da dies die gemeinsame Ober- bzw. Untergrenze der vorgegebenen neuen Randverteilungen ist.

In [11]: x\_liste, y\_liste = split\_liste(total\_xy\_list) # X- und Y-Realisationen aus gemeinsamer Liste extrahieren plt.title('Realisationen gleichverteilte X, Y mit Gauss-Copula') # Spezifischer Titel plot\_func(x\_liste, y\_liste, show=True, get\_xy\_lim=True) # Plot erzeugen und anzeigen



data = total\_summe\_liste plt.hist(data, bins=bins)

plt.xlabel('PF-Realisation')

plt.ylabel('Anzahl')

plt.show()

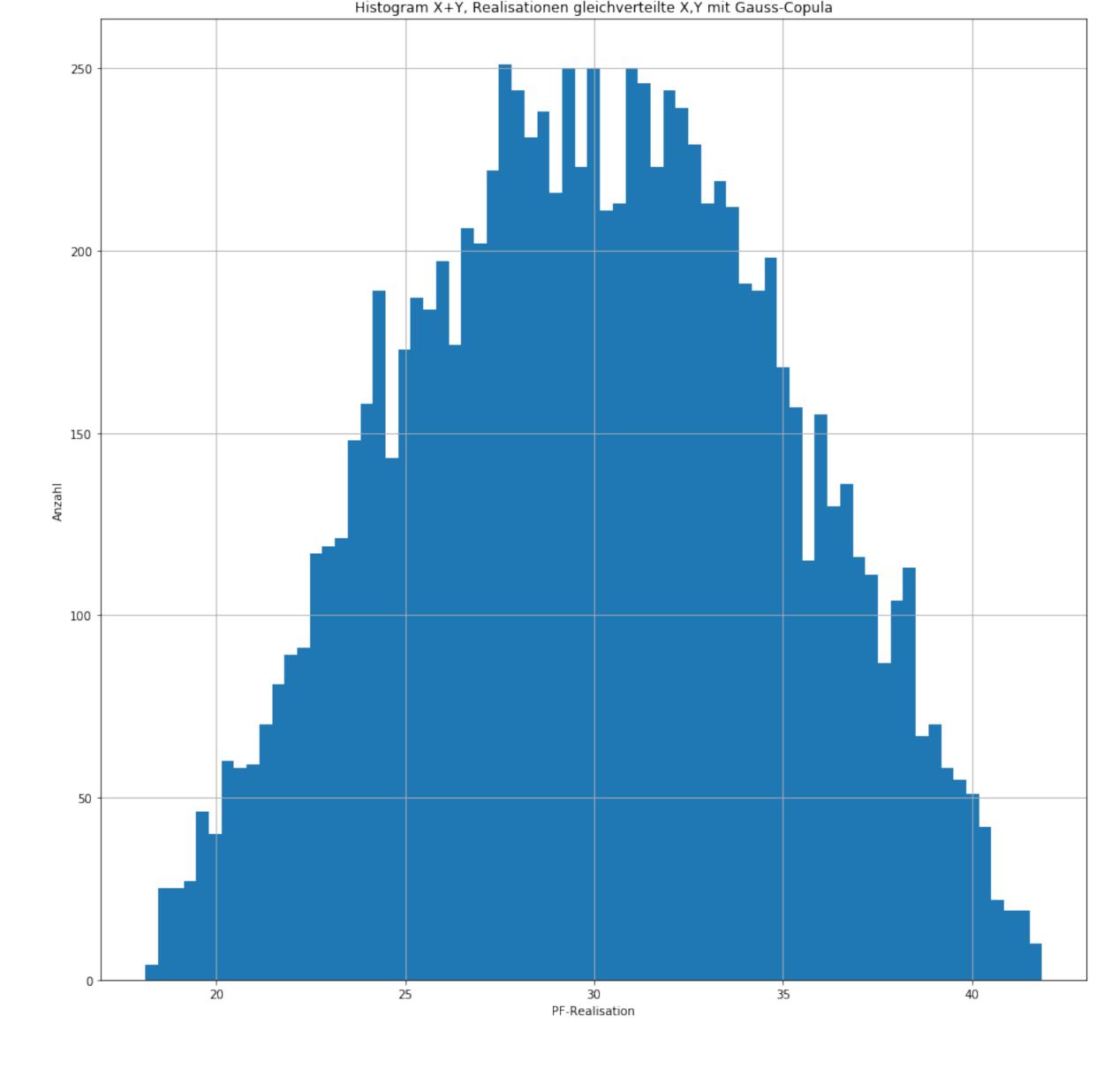
plt.title('Histogram X+Y, Realisationen gleichverteilte X,Y mit Gauss-Copula')

beschränkter glockenförmiger Verlauf, wie er für eine Normalverteilung üblich ist.

1.5 Histogramm X+Y, Realisationen gleichverteilte X, Y mit Gauss-Copula

In [12]: bins = 71 plt.grid()

Darstellung der Simulationsergebnisse als Summe der x und y Realisationen unter Berücksichtung inneren Abhänigkeitsstruktur und neuen Randverteilungen. Es zeigt sich ein auf das Intervall [18,42]



Darstellung der resultierenden kumulativen Verteilungsfunktion. Diese nimmt, wie für eine Normalverteilung üblich, einen S-förmigen Verlauf.

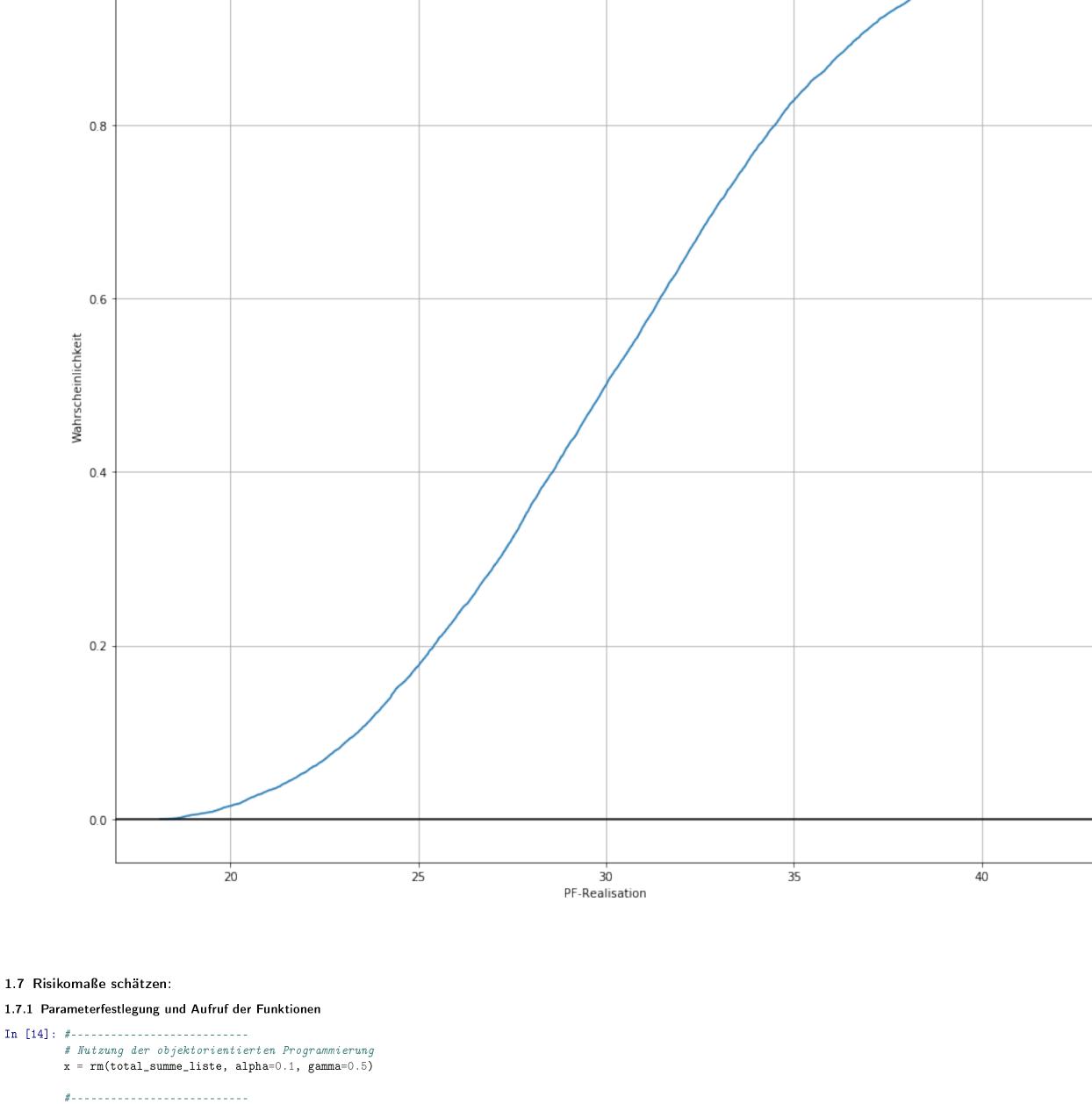
1.6 Verteilung X+Y, Realisationen gleichverteilte X, Y mit Gauss-Copula

In [13]: H, X1 = np.histogram(total\_summe\_liste, bins=n, density=True)

Monte-Carlo-Simulation(en)

hist\_func(H, X1) verteilung\_func()

1.0



Verteilung X+Y, Realisationen gleichverteilte X,Y mit Gauss-Copula

```
print('|' + centered('Das Risiko beträgt: ' + str(x.Power()) + '.') + '| ')
    print('#' + SCREEN_WIDTH * '-' + '#')
#-----#
```

# Value at Risk

print('#' + SCREEN\_WIDTH \* '-' + '#')

#-----# Conditional Value at Risk

# Power-Spektrales Risikomaß

1.8 "Instabilität" Monte-Carlo-Simulation

# Parameter Risikomaße

RM\_VaR\_list, RM\_CVaR\_list = [], [] RM\_PSRM\_list, mega\_summe\_list = [], []

for i in range(0, runs\_func):

alpha = 0.1gamma = 0.5

print('|' + centered('Der VaR beträgt: ' + str(x.VaR()) + '.') + '| ')

print('|' + centered('Der CVaR beträgt: ' + str(x.CVaR()) + '.') + '| ')

#-----

print('|' + centered('Power-Spektrales Risikomaß bei der Monte-Carlo-Simulation:') + '| ')

print('|' + centered('Der Erwartungswert beträgt: ' + str(x.expected\_value) + '.') + '| ')

Der VaR beträgt: -23.4076759890292.

Der CVaR beträgt: -21.58008721670611.

 $\textit{\# F\"{u}hre die Simulation "runs\_func mal" durch und speichere die Ergebnisse in der jeweiligen Liste \\$ 

Power-Spektrales Risikomaß bei der Monte-Carlo-Simulation: #-----# Der Erwartungswert beträgt: 30.00389831202371. Das Risiko beträgt: 26.83089017748455. #-----#

# Wiederholungen der Simulationen runs\_sim = 100 # Legt die Anzahl der Durchläufe einer Simulation fest runs\_func = 10 # Legt fest, wie viele Simulationen durchgeführt werden #-----

Ein Kritikpunkt an der Monte-Carlo-Simulation ist, dass das Ergebnis des Verfahrens großen Schwankungen unterliegen kann, sofern nur wenige Realisationen in einem Simulationslauf simuliert werden.

Dies soll die folgende Grafik veranschaulichen. Dabei kann sowohl die Anzahl der Simulationsläufe als auch die Anzahl der in jeder Simulation durchgeführten Simulationen variiert werden.

```
_, _, _, total_summe_liste = copula_sim(runs_sim, rand_x, rand_y, mu, std_list, corr_list, full_log=False)
           mega_summe_list += total_summe_liste
           x = rm(total_summe_liste, alpha, gamma)
           RM_VaR_list.append(x.VaR())
           RM_CVaR_list.append(x.CVaR())
            RM_PSRM_list.append(x.Power())
        #-----
        # Erzeuge ein DataFrame mit den Simulationsvergebnissen
        # und deren prozentualen Änderung vom jeweils vorherigen Ergebnis
        RM_frame = pd.DataFrame()
        RM_frame['VaR'] = RM_VaR_list
        RM_frame['VaR-Change'] = RM_frame['VaR'].pct_change()
        RM_frame['CVaR'] = RM_CVaR_list
        RM_frame['CVaR-Change'] = RM_frame['CVaR'].pct_change()
        RM_frame['Power'] = RM_PSRM_list
        RM_frame['Power-Change'] = RM_frame['Power'].pct_change()
        #-----
        # Ermittle die kleinste und größte Relaisation des jweiligen Risikomaßes
        Min_Max_VaR = (min(RM_VaR_list), max(RM_VaR_list))
        Min_Max_CVaR = (min(RM_CVaR_list), max(RM_CVaR_list))
        Min_Max_PSRM = (min(RM_PSRM_list), max(RM_PSRM_list))
        # Gib den DataFrame und die Infos zurück
        print('#' + SCREEN_WIDTH * '-' + '#')
        print('|' + centered('[INFO] Der DataFrame mit den auf den auf ' +str(runs_func) + ' mal ' + str(runs_sim) + ' Durchläufen beruhenden Risikomaßen ergibt sich wie folgt: ') + '| ')
        print('#' + SCREEN_WIDTH * '-' + '#')
        print(RM_frame)
        print('#' + SCREEN_WIDTH * '-' + '#')
        print('|' + centered('Nach ' + str(runs_func) + ' Simulationsläufen mit je ' + str(runs_sim) + ' Durchläufen beträgt der kleinste VaR ' + str(round(Min_Max_VaR[0],2)) +', der größ
        print('|' + centered('Nach ' + str(runs_func) + ' Simulationsläufen mit je ' + str(runs_sim) + ' Durchläufen beträgt der kleinste CVaR ' + str(round(Min_Max_CVaR[0],2)) +', der gr
        print('|' + centered('Nach ' + str(runs_func) + ' Simulationsläufen mit je ' + str(runs_sim) + ' Durchläufen beträgt das kleinste P-SRM ' + str(round(Min_Max_PSRM[0],2)) +', das g
        print('#' + SCREEN_WIDTH * '-' + '#')
        #-----
        # Zerlege die mega_summe_list (beinhaltet alle Ergenisse) in Teillisten,
        # welche die Ergebnisse der einzelnen Simulationsläufe beinhalten
        counter_0, counter_1 = 0, runs_sim
        array = []
        for i in range(0, runs_func):
           x = mega_summe_list[counter_0:counter_1]
           array.append(x)
           counter_0 += runs_sim
           counter_1 += runs_sim
        #----
        # Erstelle für jedes Dieser Teillisten ein Histogramm
        # und plotte anschlißend das Ergebnis
        for items in array:
           values_PF = items
           bins = runs_sim
           H, X1 = np.histogram(values_PF, bins, density=True)
           hist_func(H, X1)
        verteilung_func()
   [INFO] Der DataFrame mit den auf den auf 10 mal 100 Durchläufen beruhenden Risikomaßen ergibt sich wie folgt: |
#-----#
        VaR VaR-Change
                           CVaR CVaR-Change
                                                 Power Power-Change
0 -22.330991
                  NaN -21.071164 NaN 26.820230
                                                               {\tt NaN}

      1 -23.642555
      0.058733 -21.376833
      0.014507 26.607497

      2 -22.568408
      -0.045433 -20.699304
      -0.031695 25.943138

                                                          -0.007932
                                                          -0.024969
0.033660
-0.017037

      5 -24.500979
      0.040075 -22.357867
      0.021114 27.148708

      6 -23.204382
      -0.052920 -22.089899
      -0.011985 26.747490

                                                          0.029940
                                                          -0.014779
7 -24.026181 0.035416 -21.949410 -0.006360 26.685129
                                                          -0.002331
8 -23.754698 -0.011299 -21.525206 -0.019326 27.193393
                                                          0.019047
9 -23.033860 -0.030345 -21.903526 0.017576 27.003647
                                                          -0.006978
#-----#
| Nach 10 Simulationsläufen mit je 100 Durchläufen beträgt der kleinste VaR -24.5, der größte -22.33 (\Delta = 9.72%). |
| Nach 10 Simulationsläufen mit je 100 Durchläufen beträgt der kleinste CVaR -22.36, der größte -20.7 (\Delta = 8.01%).
Nach 10 Simulationsläufen mit je 100 Durchläufen beträgt das kleinste P-SRM 25.94, das größte 27.19 (\Delta = -4.6%).
#-----#
                                                  Verteilung X+Y, Realisationen gleichverteilte X,Y mit Gauss-Copula
                 Monte-Carlo-Simulation(en)
           1.0
           0.8
```