

1	2
8/12	8/8
	16/20

blatt03_nitschke_grisard

November 8, 2018

1 Blatt 3

1.1 Aufgabe 8: Importance Sampling

```
In [1]: import numpy as np
import matplotlib.pyplot as plt
import timeit
```

a) Die Planck-Verteilung:

```
In [2]: def Planck(x, N = 15 / np.pi**4):
return N * x**3 / (np.exp(x) - 1)
```

Bestimme zunächst numerisch das Maximum:

```
In [3]: from scipy.optimize import brentq
```

```
N = 15 / np.pi**4
```

```
def diff_Planck(x):
return N * (3 * x**2 * (np.exp(x) - 1) - x**3 * np.exp(x)) / (np.exp(x) - 1)**2
```

```
xmax = brentq(diff_Planck, 1, 4) #root of derivation
ymax = Planck(xmax) # max. value of planck distribution
```

Funktion für *Rejection Sampling*, um eine vordefinierte Länge des Samples zu erreichen, wird die gewünschte Anzahl aus der erzeugten Verteilung gezogen:

```
In [4]: def Rejection_sampling(u1, u2, function, length = 100000):
sample = u1[u2 <= function(u1)]
assert len(sample) >= length
return sample[np.random.randint(0, len(sample), length)], len(sample)
```

Verwende Funktion um normales Rejection Sampling durchzuführen:

```
In [5]: start_time = timeit.default_timer()
```

```
xcut = 20 # cutoff
```

da
das ist
hier
nicht
das
richtige
Rückweisung
verfahren,

daher geht
es auch so
schnell
bei such
-1P.

ich glaube bei Aufgabe 5 war das auch nicht ganz richtig
sorry, dass ich das da nicht erkannt hab!

-0,5P. Ihr erzeugt nicht genau 10^5 Zahlen
aber die Verteilung kommt schön raus :)

```
totnumber = 500000
uniformx = np.random.uniform(0, xcut, totnumber)
uniformy = np.random.uniform(0, ymax, totnumber)

planck_sample_a, sample_len_a = Rejection_sampling(uniformx, uniformy, Planck) ✓

elapsed_a = timeit.default_timer() - start_time
```

2,5/4

b) Bestimme zunächst den Schnittpunkt der Majoranten x_s :

```
In [6]: def diff(x, N = 15 / np.pi**4, xmax = xmax):
        return ymax - 200 * N * x**(-0.1) * np.exp(-x**(0.9))
        #difference of functions has to be zero

x_s = brentq(diff, 1, 6)
print(f'Schnittpunkt x_s = {x_s}')
```

Schnittpunkt $x_s = 5.678208598337659$ ✓

Implementiere die Majorante $g(x)$:

```
In [7]: def func_g(x, x_s = x_s, N = 15 / np.pi**4):
        y = np.zeros(len(x))
        y[x <= x_s] = ymax
        y[x > x_s] = 200 * N * x[x >= x_s]**(-0.1) * np.exp(-x[x >= x_s]**(0.9))
        return y
```

Nun soll zunächst ein Sample erzeugt werden, dass gemäß $g(x)$ verteilt ist. Dies wird einzeln für $x \leq x_s$ und $x > x_s$ gemacht. Die richtige Anzahl an Zufallszahlen rechts und links von x_s ist aus dem Verhältnis der Flächen unter $g(x)$ berechenbar.

```
In [8]: def inv_G(y, x_s = x_s): #inverse CDF for x > x_s
        return (- np.log(np.exp(-x_s**(0.9))
                        + y * (- np.exp(-x_s**(0.9)))) ** (10 / 9))
```

Normierung
fehlt

```
In [9]: start_time = timeit.default_timer()
```

es muss berechnet
werden:

```
norm = x_s * ymax + 2000 / 9 * N * (np.exp(-x_s**(9/10))) ✓
#total norm of g(x)
```

$$f(x) = \frac{f(x)}{A}$$

Normierung

```
part_uniform = int(x_s * ymax / norm * totnumber)
```

also
noch
eine

```
uniformx_greater_x_s = np.random.uniform(0, 1, totnumber - part_uniform)
sample_greater_x_s = inv_G(uniformx_greater_x_s)
```

2. Normierung

```
sample_less_x_s = np.random.uniform(0, x_s, part_uniform)
```

-1P.

```

uniformy = np.random.uniform(0, 1, totnumber)
sample_g = np.concatenate([sample_less_x_s, sample_greater_x_s])

planck_sample_b, sample_len_b = Rejection_sampling(sample_g,
                                                    func_g(sample_g) * uniformy,
                                                    Planck)

elapsed_b = timeit.default_timer() - start_time

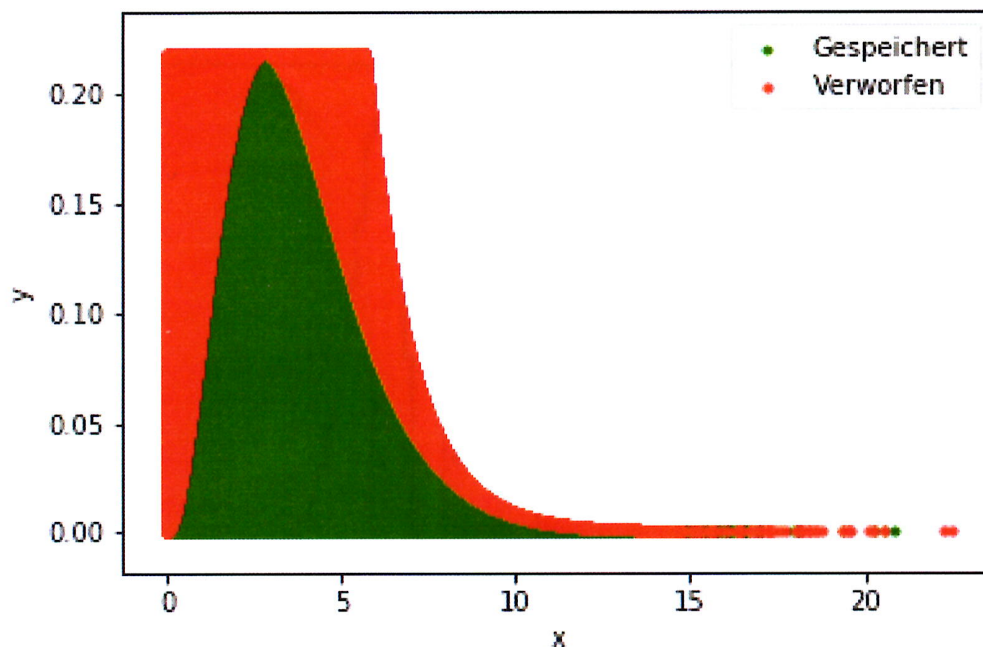
In [10]: plt.clf()
         xplot = np.linspace(0.01, 30, 1000)

         mask = func_g(sample_g) * uniformy <= Planck(sample_g)
         plt.scatter(sample_g[mask], func_g(sample_g[mask]) * uniformy[mask],
                     marker = '.', color = 'g', label = 'Gespeichert')

         plt.scatter(sample_g[mask == False],
                     func_g(sample_g[mask == False]) * uniformy[mask == False],
                     marker = '.', color = 'r', label = 'Verworfen')

         plt.xlabel('x')
         plt.ylabel('y')
         plt.legend()
         plt.show()

```



4/6

ein Histogramm wäre
schöner ☺
aber das kommt ja
auf der nächsten Seite

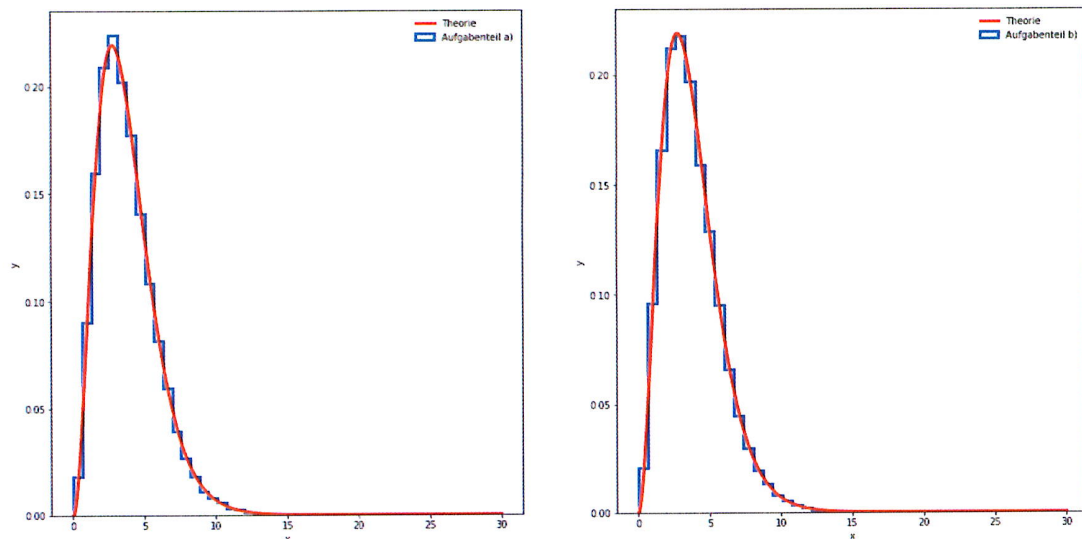
c) Stelle die Datensätze aus a) und b), sowie die Theoriekurve dar:

```
In [11]: fig = plt.figure(figsize = (20, 10))

fig.add_subplot(121)
plt.hist(planck_sample_a, bins = 30, histtype = 'step',
         density = True,
         label = 'Aufgabenteil a)',
         linewidth = 3)
plt.plot(xplot, Planck(xplot),
         color = 'r', linewidth = 3, label = 'Theorie')
plt.legend()
plt.xlabel('x')
plt.ylabel('y')

fig.add_subplot(122)
plt.hist(planck_sample_b, bins = 30, histtype = 'step',
         density = True,
         label = 'Aufgabenteil b)',
         linewidth = 3)
plt.plot(xplot, Planck(xplot),
         color = 'r', linewidth = 3, label = 'Theorie')
plt.legend()
plt.xlabel('x')
plt.ylabel('y')

plt.show()
```



c) Vergleiche Laufzeiten und Effizienzen:


```
In [12]: print(f'Laufzeit a): {elapsed_a}s')
         print(f'Effizienz a): {sample_len_a / totnumber * 100}%')
```

```
print(f'Laufzeit b): {elapsed_b}s')
print(f'Effizienz b): {sample_len_b / totnumber * 100}%')
```

```
Laufzeit a): 0.07996194999998352s
Effizienz a): 23.0402%
Laufzeit b): 0.10444585200002621s
Effizienz b): 65.3304%
```

bei mir dauert die erste Methode viel länger (etwa 2,5s)

das erscheint mir komisch

Kommentar: Die Laufzeit ist bei Methode a) etwas besser. Das liegt daran, dass für Methode b) zunächst noch Rechnungen durchgeführt werden müssen, um die Daten gemäß der Majorante zu verteilen. Die Effizienz ist dadurch in b) jedoch wesentlich besser.

1.2 Aufgabe 9: Metropolis-Hastings-Algorithmus

a) Offenbar gilt für symmetrische Schrittvorschlagsverteilungen $g(x_i|x_j) = g(x_j|x_i)$. Daher geht z.B. für eine Gaußverteilung der Metropolis-Hastings-Algorithmus in den Metropolis-Algorithmus über.

b) Implementiere den Metropolis Hastings Algorithmus mit einer Gleichverteilung als Schrittvorschlagsfunktion:

```
In [13]: def metropolis(distribution, x0, stepsize = 2, length = 100000):
```

```
    x = [x0]
```

```
    for i in range(length):
```

```
        next_x = np.random.uniform(x[i] - stepsize, x[i] + stepsize)
```

```
        prob = min(1, distribution(next_x) / distribution(x[i]))
```

```
        xi = np.random.uniform(0, 1)
```

```
        if prob >= xi and next_x >= 0:
```

```
            x.append(next_x)
```

```
        else:
```

```
            x.append(x[i])
```

```
    return np.array(x)
```

```
In [14]: x = metropolis(distribution = Planck, x0 = 30)
```

```
In [15]: counts, binedges = np.histogram(x, bins = 20)
```

```
         normed_counts = counts / sum(counts * np.diff(binedges))
```

```
xplot = np.linspace(0.01, 30, 1000)
```

```
plt.errorbar(x = (binedges[:-1] + binedges[1:]) * 0.5,
```

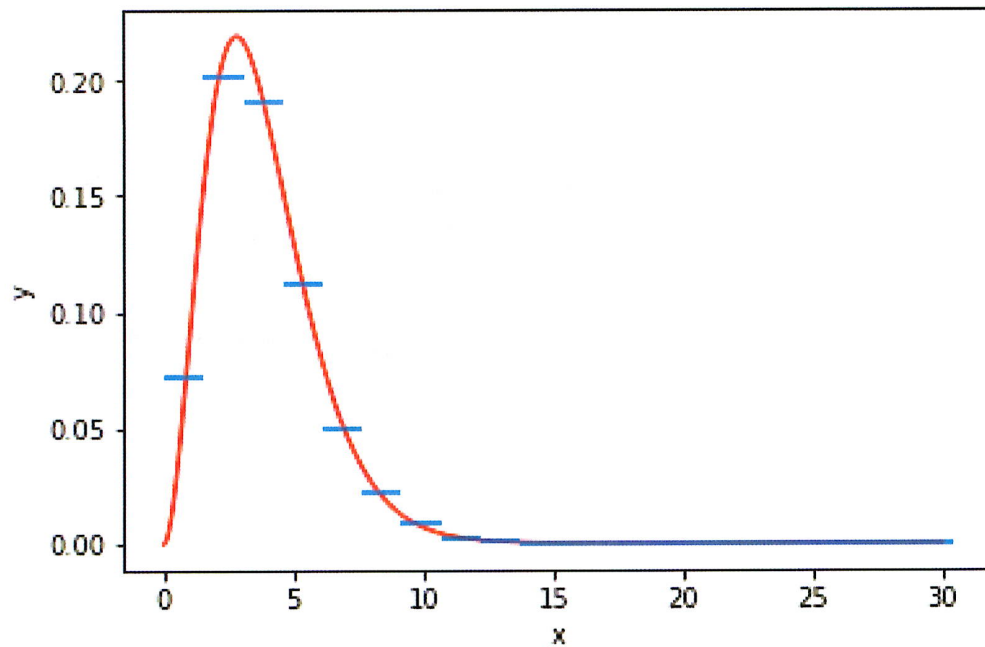
```
             y = normed_counts, xerr = np.diff(binedges) * 0.5, linestyle = '',
             label = 'Daten')
```

```
plt.plot(xplot, Planck(xplot), zorder = 1, color = 'r')
```

```
plt.xlabel('x')
```

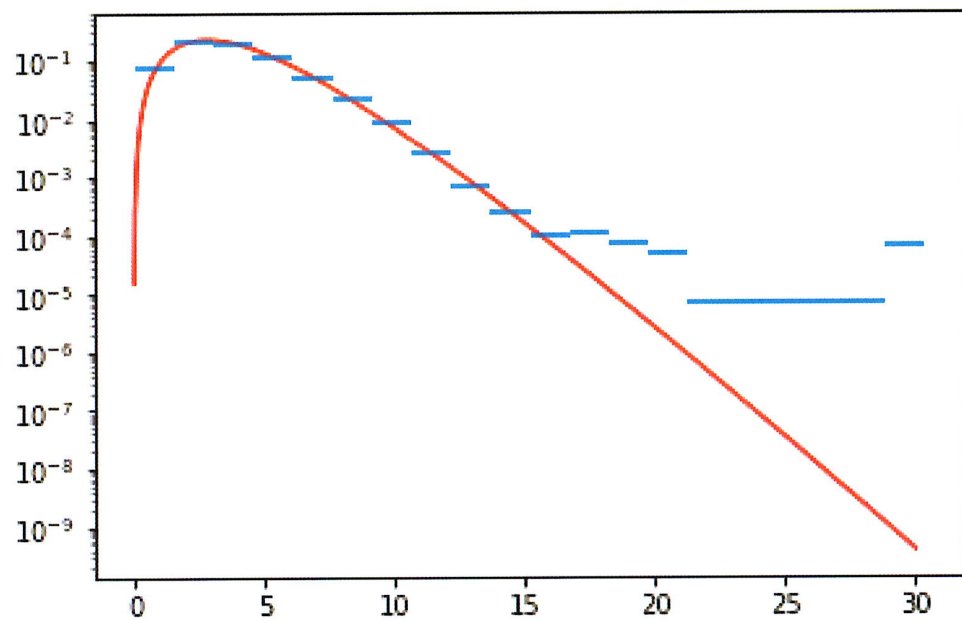
```
plt.ylabel('y')
```

```
plt.show()
```



Logarithmische Darstellung zeigt, dass die erzeugte Verteilung nicht so gut an die Theoriekurve passt für Werte, ab ca. $x = 20$. Dies könnte an dem schlecht gewählten Startpunkt liegen. ✓

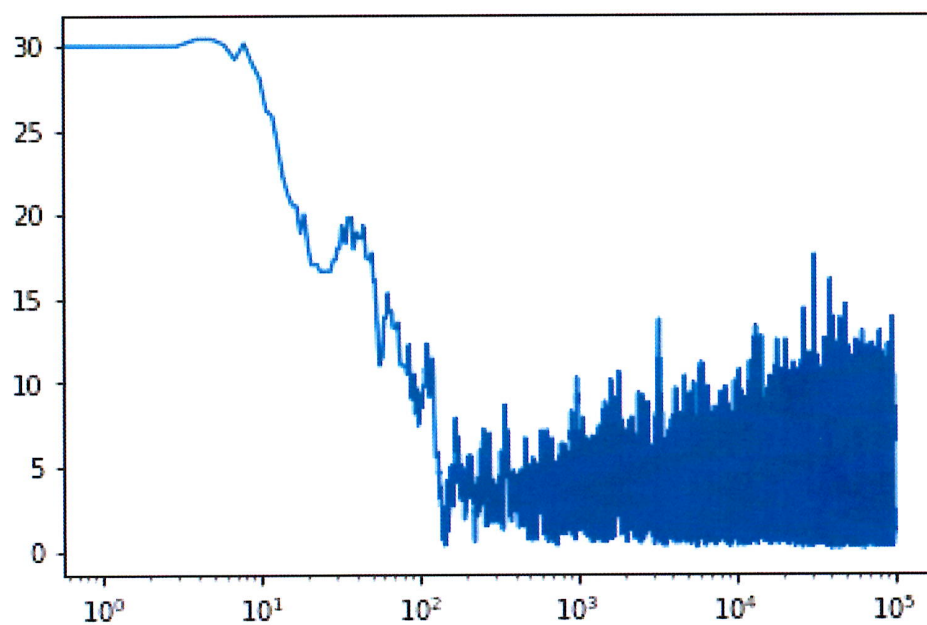
```
In [16]: xplot = np.linspace(0.01, 30, 1000)
plt.errorbar(x = (binedges[:-1] + binedges[1:]) * 0.5,
             y = normed_counts, xerr = np.diff(binedges) * 0.5, linestyle = '',
             label = 'Daten')
plt.plot(xplot, Planck(xplot), zorder = 1, color = 'r')
plt.yscale('log')
```



✓

d) Traceplot

```
In [17]: plt.clf()
plt.plot(range(len(x)), x)
plt.xscale('log')
plt.show()
```



0,5 P

Kommentar: Man erkennt deutlich die Burn-In-Phase. Für weitere Iterationen schwanken die Werte um das Maximum der Verteilung bei etwa 3. Man erkennt einen Trend hin zu einer grösseren Streuung der Daten um die Maximalstelle.



0,5P

8/8