4,5/10

Albert-Ludwigs-Universität Freiburg Wintersemester 2022/23

Datenanalyse für Naturwissenschaftler*Innen

Statistische Methoden in Theorie und Praxis

Vorlesung: Dr. Andrea Knue
Übungsleitung: Dr. Constantin Heidegger

Übung 7

Ausgabe: 2. Dezember 2022 10:00 Uhr, Abgabe: 9. Dezember 2022 bis 10:00 Uhr via Ilias

Aufgabe 4: Monte Carlo Simulation (10P)

In der Vorlesung haben wir so-genannte Monte Carlo (MC) Simulationen kennengelernt. Diese wollen wir in dieser Aufgabe näher untersuchen indem wir eine eigene MC Simulation entwerfen.

a) Zufallsgenerator (1P) 1/1

Kernel vor der Abgabe neu starten & alles ausführen -1P

Wir betrachten zunächst die Funktion `ran0(seed)`, die für ein gewisses Seed `seed` bei jedem Aufruf eine neue Zufallszahl generiert. Bearbeiten Sie diese Funktion, sodass Sie zwei Parameter `maximum` and `minimum` nehmen um den Bereich der generierten Zahlen dadurch definieren. Geben Sie dann ein paar Zufallszahlen in verschiedenen Bereichen aus, um Ihre Änderungen zu testen.

In [18]:

```
def ran0(seed, minimum=0, maximum=1):
    m = 2**31-1
    a = 7**5
    c = 0
    while True:
        seed = (a * seed + c) % m
        yield seed/(m+1.)
```

Ein bisschen Python Erklärung: die Funktion `ran0` gibt einen so-genannten `Generator` zurück, weswegen sie anstelle des Befehls `return` den Befehl `yield` verwendet. Ein Generator ist praktisch ein "iterable" (= ein Objekt, über das man iterieren kann, also z.B. eine Liste von Elementen wie `mylist` in `for x in mylist`), das aber nur einmal iteriert werden kann. Wofür ist das gut? Performance. Und Flexibilität. In unserem Fall sehen wir dass `ran0` einen infinite loop enthält (`while True:`), also eine Schleife die aufgrund der Schleifenbedingung nie aufhören würde. In jeder Iteration gibt der `yield` Befehl eine neue Zufallszahl zurück, die außerhalb der Funktion via `next(x)` extrahiert werden kann, wobei `x` der Pointer zum Generator ist, also `x=ran0(...)`. D.h. außerhalb muss man eine Schleife machen, die eine fixe Anzahl

Iterationen durchläuft (entsprechend der Anzahl Zufallszahlen, di eman haben möchte) und dann pro Iteration eine neue Zufallszahl vom Generator extrahiert. Bemerke, dass nichts davon funktionieren würde, wenn wir `return` statt `yield` verwenden würden. Dann müsste `ran0` eine Liste mit einer vorher festgelegten Anzahl Elemente befüllen und alle Zufallszahlen im Speicher behalten, selbst wenn sie später nicht verwendet werden - etwas, was man vermeiden möchte wenn man es mit sehr sehr viel Zufallszahlen zu tun hat (was üblicherweise in der Physik der Fall ist).

In [19]:
```
def ran01(n, seed=1, minimum=0, maximum=1):
    x = ran0(seed)
    numbers = []
    for i in range(n):
        number = (maximum-minimum)*next(x) + minimum
        numbers.append(number)
    return numbers
```

✓

In [20]:
```
print(ran01(10, seed=1, minimum=50, maximum=100))
```

```
[50.00039131846279, 56.57688940409571, 87.78026609215885, 72.93250
658549368, 76.63836185820401, 60.94795931130648, 52.3522308096289
6, 83.9432358276099, 83.9648202760145, 96.73464477527887]
```

In [21]:
```
print(ran01(10, seed=9, minimum=20, maximum=30))
```

```
[20.00070437323302, 21.83840093202889, 28.004478993825614, 21.2785
11872515082, 27.949051363393664, 29.70632676500827, 24.23401545733
2134, 21.09782451763749, 21.13667652476579, 24.122360632754862]
```

In [22]:
```
print(ran01(10, seed=100, minimum=1000, maximum=1050))
```

```
[1000.0391318462789, 1007.6889407122508, 1028.0266109621152, 1043.
2506595971063, 1013.8361870544031, 1044.795931619592, 1035.2230810
560286, 1044.323584320955, 1046.482029161416, 1023.4644796932116]
```

✓

b) Darstellung (2P)

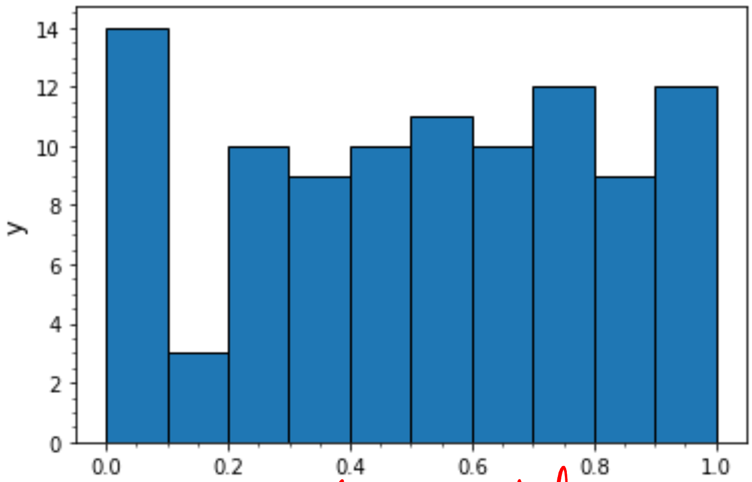
2/2

Die Qualität eines Zufallszahlgenerators kann validiert werden durch die Darstellung der Frequenz der generierten Zufallszahlen in einem gegebenen Intervall mit gleichmäßigem Binning. Erstellen Sie zwei Diagramme, eines für 100 und eines für 10000 Zufallszahlen. Verwenden Sie 10 Bins im Intervall $[0, 1]$. Die Anzahl Einträge sollte die gleiche sein für alle Bins unter Berücksichtigung der statistischen Fehler.

```
In [23]: import numpy as np
         import matplotlib.pyplot as plt


         def plotRan0(n):
             a = ran01(n, seed=1, minimum=0, maximum=1)
             plt.hist(a, bins=10, edgecolor="k")
             plt.xlabel("x", fontsize=13)
             plt.ylabel("y", fontsize=13)
             plt.minorticks_on()
             plt.show()


         plotRan0(100)
```

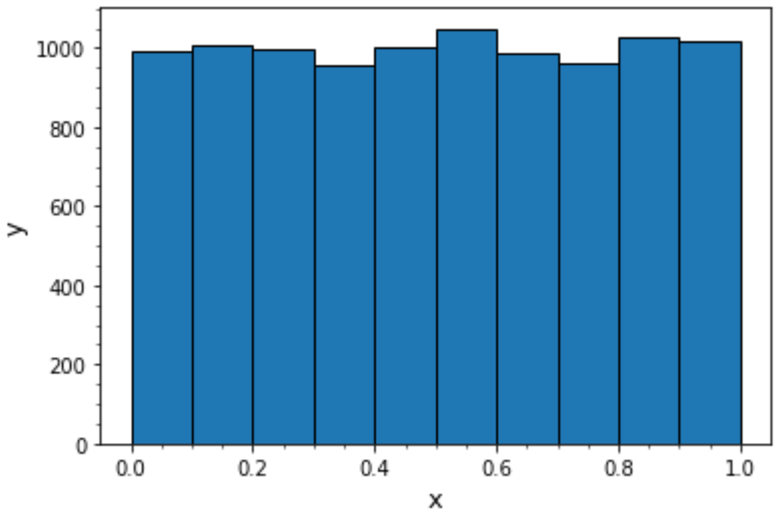
n= 100

Anzahl



Zufallszahl

```
In [24]: plotRan0(10000)
```



✓

c) Transformationsmethode (4P)

Generieren Sie nun 1000 Zufallszahlen um mit der Transformationsmethode die Funktion ( ) = 1 + cos( ) + cos(3) zu simulieren und befüllen Sie dann ein Histogramm für = cos( ) mit 20 Bins im Intervall [1, 1]. Zum Vergleich, plotten Sie auch die eigentliche Funktion (diese müssen Sie ggf. zu Ihrem Histogramm normieren).

Hinweis: Falls Sie das Integral nicht mit Python lösen möchten, dürfen Sie das Integral auch per Hand durchführen.

Hinweis: Um eine Python-Funktion zu integrieren, verwenden Sie am besten die Funktion `integrate.quad` aus der Bibliothek `scipy` , [https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.quad.html (https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.quad.html)](https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.quad.html).

Hinweis: Sie können die Inverse der Funktion auf die folgende Art und Weise via der Bibliothek `sympy` erstellen, die die Möglichkeit zu symbolischen Berechnungen bietet (z.B. können Sie damit eine Funktion integrieren ohne Integrationsgrenzen geben zu müssen):

In [25]:
```
import sympy as sy
x,y,z = sy.symbols("x y z", positive=True)  ## define variables within sympy
f     = 1+z+z**2                            ## define the function using z = c
cdf   = sy.integrate(f, (z, -1, x))         ## symbolic integration of f to get
tot   = sy.simplify(cdf.subs(x, 1))         ## integral of f from -1 to 1
cdf   = cdf/tot                             ## normalization of CDF
ppf   = sy.simplify(sy.solve(cdf-y,x)[2])   ## the PPF, i.e. the inverse functi
func  = sy.lambdify(y,ppf)                  ## convert it to a python function
```

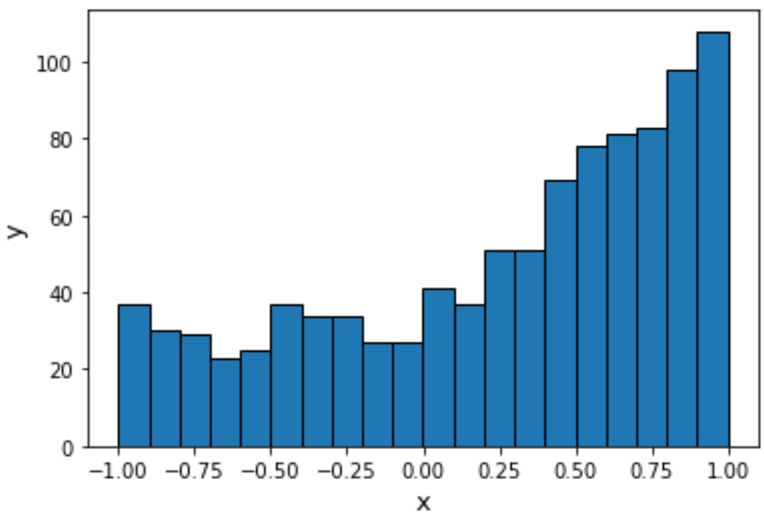
In [26]:
```
print(ppf)
```

```
-(-32*y + sqrt((32*y - 5)**2 + 27) + 5)**(1/3)/2 - 1/2 + 3/(2*(-32
*y + sqrt((32*y - 5)**2 + 27) + 5)**(1/3))
```

In [27]:
```
def invMethod(f, n, r=20):
    y = np.random.uniform(0, 1, n)
    plt.hist(func(y), bins=r, edgecolor="k")
    plt.xlabel("x", fontsize=13)
    plt.ylabel("y", fontsize=13)
    plt.show()


invMethod(func, 1000, r=20)
```

2/4

f



+ vgl mit F(x)
mit $x = \cos\theta$
$\Rightarrow f = 1 + x + x^2$
$f(-1) = 1$
$f(1) = 3$
die Normierung ist falsch

d) Hit-und-Miss Methode (4P)

Wiederholen Sie Aufgabenteil c) mit der Hit-und-Miss Methode.

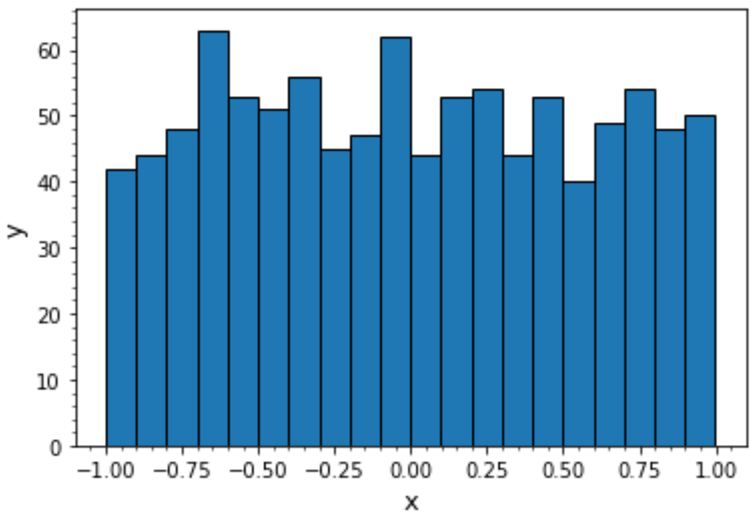
In [28]:
```python
def funct(a):
    return 1+a+a**2


def rejMethod(f, n, r=20):
    numbers = []
    misses = []
    while len(numbers) < n:
        u1 = np.random.uniform(0, 1)
        u2 = np.random.uniform(0, 1)
        x = -1+u1*2
        w = -0.5*u2
        if w < f(x):
            numbers.append(x)
    plt.hist(numbers, bins=r, edgecolor="k")
    plt.xlabel("x", fontsize=13)
    plt.ylabel("y", fontsize=13)
    plt.minorticks_on()
    plt.show()


rejMethod(funct, 1000, r=20)
```

falsche Intervalle

$u_1 \in [-1, 1]$

$u_2 \in [0, p_{max}]$

2.



f

1,5/4