Monte Carlo Methods

Monte Carlo Methods

Uwe Jaekel – University of Applied Sciences Koblenz

```python
import numpy as np
import matplotlib.pyplot as plt
#from mpl_toolkits.mplot3d import Axes3D
#%matplotlib inline
%matplotlib widget
from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets
import matplotlib
matplotlib.rcParams['text.usetex'] = True
import time
```

What are Monte Carlo (MC) methods?

- Solving stochastic or deterministic problems using pseudo random numbers
- Often "quadrature" – estimation of integrals
  - Expectation values
  - High dimensional integrals
  - Integrals over complicated and irregular shapes
- Many other applications
  - Partial differential equations
  - Computational finance
  - Computational statistics (Bayesian approaches)

Advantages of MC methods

- Easy to implement
- Well-suited for high-dimensional problems
- In some cases the only known methods for otherwise intractable problems

Reminder: Facts we need from probability and statistics

Expectation value of a random variable X

- Discrete: $X \in \{x_1, x_2, \dots, x_N\}$ where x_k comes with probability p_k:

$$E[X] := \sum_{k=1}^{n} p_k x_k$$

- Continuous: $X \in (a, b)$ with probability density p:

$$E[X] := \int_a^b p(x) x \, dx$$

Variance and standard deviation

- $$\mathrm{var}[X] = E\Big[(X - E[X])^2\Big] = E[X^2] - E[X]^2$$
- $$\mathrm{std}[X] = \sqrt{\mathrm{var}[X]}$$

Expectations are linear

When α, β are constants, and X, Y random variables, then

$$E[\alpha X + \beta Y] = \alpha E[X] + \beta E[Y]$$

Variances of independent random variables X, Y are additive

$$\mathrm{var}[X+Y] = \mathrm{var}[X] + \mathrm{var}[Y]$$

Together with linearity, we find: ### Error of averages (sample means) of independent, identically distributed variables is proportional to $1/\sqrt{\text{Sample Size}}$

$$\mathrm{var}[\frac{1}{N}\sum_{k=1}^{N} X_k] = \frac{1}{N^2}\sum_{k=1}^{N}\mathrm{var}[X_k] = \frac{\mathrm{var}[X]}{N}$$

$$\mathrm{std}[\frac{1}{N}\sum_{k=1}^{N} X_k] = \frac{\mathrm{std}[X]}{\sqrt{N}}$$

Idea

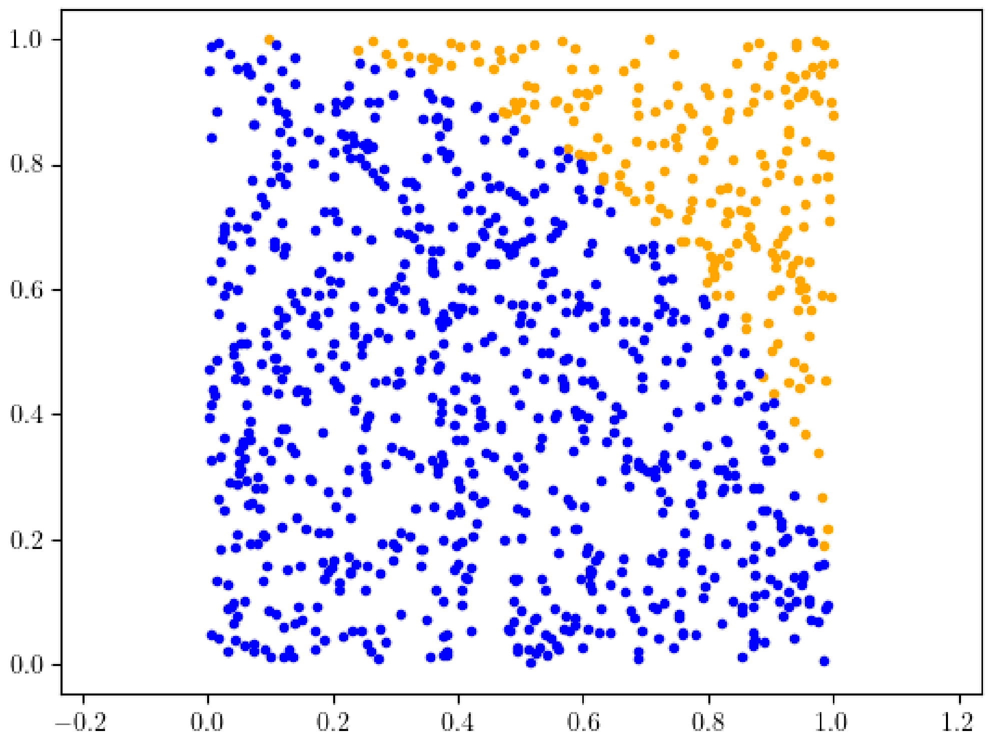
Classical example: Estimation of π.

```
from random import uniform

def mc_pi1(n):
    sum = 0
    for k in range(n):
        # Draw random points from the unit square
        x = uniform(0, 1)
        y = uniform(0, 1)
        # Check if point is inside unit sphere
        if x*x + y*y < 1:
            sum += 1
    return 4.0*sum/n

mc_pi1(10000)
```

```
3.1308
```

```
N = 1000
x = np.random.rand(N)
y = np.random.rand(N)
hit = (x*x + y*y <= 1.0)
plt.figure()
plt.scatter(x[np.logical_not(hit)], y[np.logical_not(hit)], color="orange", marker=".")
plt.scatter(x[hit], y[hit], color="blue", marker=".")
plt.axis("equal")
hit = 1.0*hit
pi_est = 4*hit.sum()/N
pi_err = 4*np.std(hit)/np.sqrt(N)
print(f"pi_est = {pi_est} +/- {pi_err}  [hits: {hit.sum()}/{N}]")
```

```
pi_est = 3.14 +/- 0.05196537308631586  [hits: 785.0/1000]
```



Summary so far

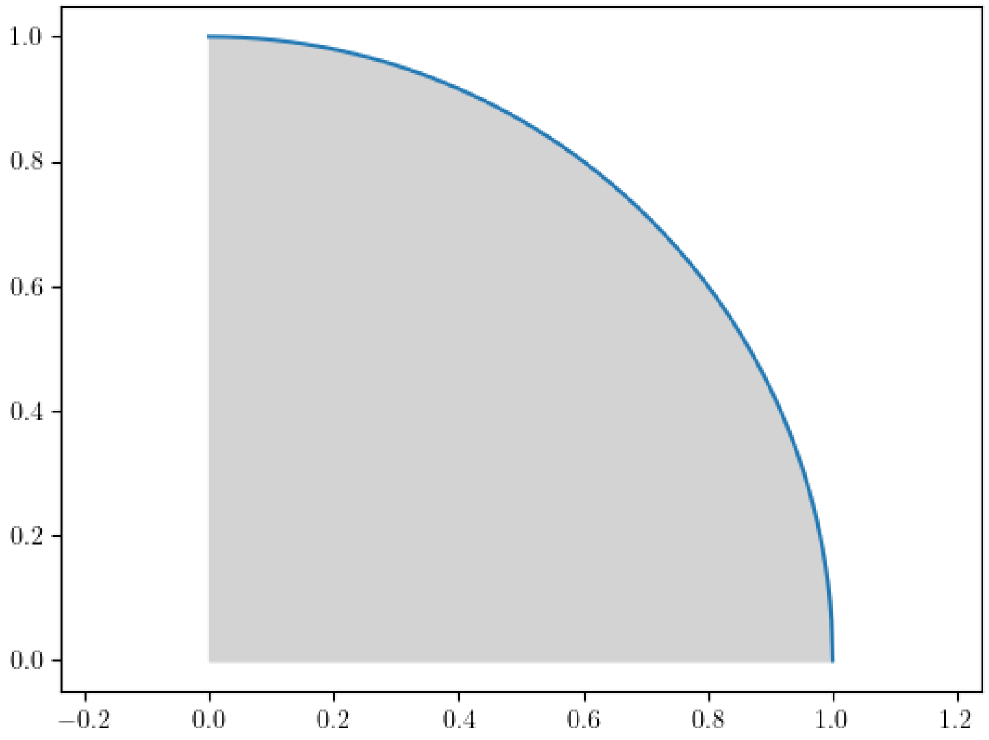
- MC often easy to implement
- Estimate expectation values from sample means
- Sample standard deviation gives an error estimate (!)

Another MC method for the computation of π

Area under the circle:

$$\pi = 4 \int_0^1 \sqrt{1 - x^2} dx.$$

```
plt.figure()
x = np.linspace(0,1,500)
y = np.sqrt(1 - x*x)
plt.fill_between(x,y, color='lightgray')
plt.plot(x,y)
plt.axis('equal')
plt.show()
```



Rewrite the deterministic integral as an expectation over a uniform distribution

$$\int_0^1 f(x)dx = \int_0^1 p(x)f(x)dx = E_X[f(X)]$$

where $X \sim U(0,1)$, uniformly distributed on $[0,1]$ with constant density $p(x) = 1$.

MC estimator: Draw N uniformly distributed samples $x_1, \ldots, x_N$ from $[0,1]$

$$I = E_X[f(X)] \approx \hat{I}_{MC} = \frac{1}{N}\sum_{k=1}^{N} f(x_k).$$

Let's implement it for π

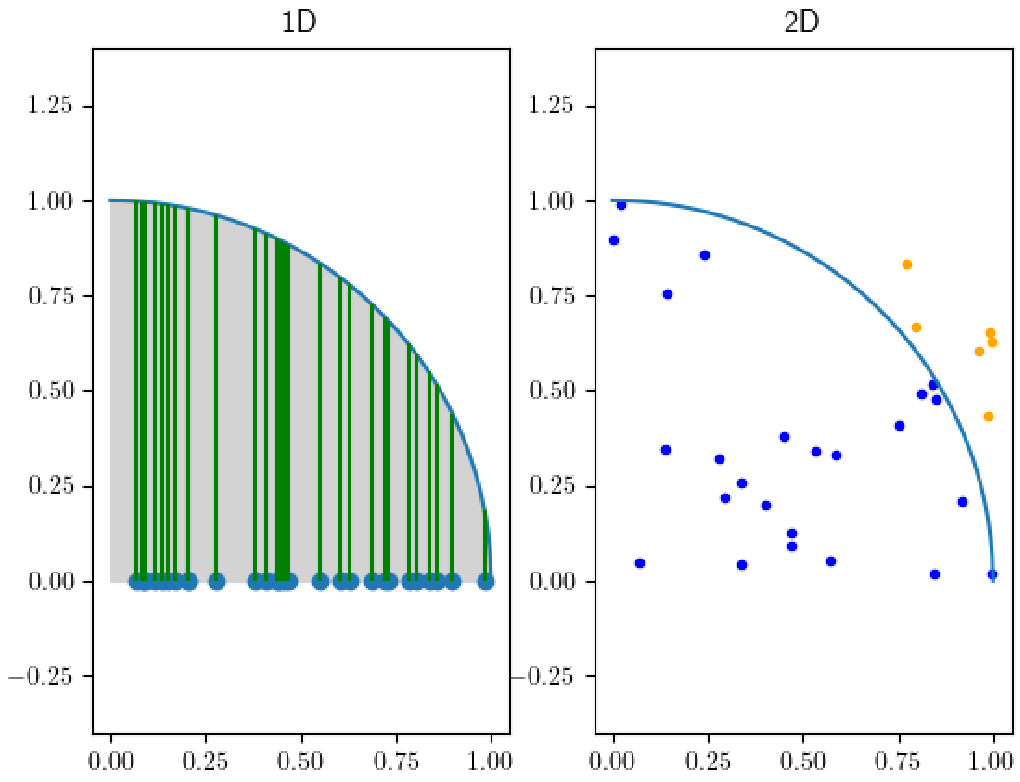
```
N = 100000
# np.random.seed(42)   # Generates the same random number sequence on every run and every co
x = np.random.rand(N)
f = np.sqrt(1 - x*x)
I_MC = 4.0*f.mean()
err_I = 4.0*f.std()/np.sqrt(N)
print(f"I_MC = {I_MC} +/- {err_I}")
```

```
I_MC = 3.1433374910581224 +/- 0.0028202065223852653
```

Which estimator is better? 1D oder 2D?

```
NMC = 30
x1d = np.random.rand(NMC)
y1d = np.sqrt(1 - x1d*x1d)

x2d = np.random.rand(NMC)
y2d = np.random.rand(NMC)
hit2d = (x2d*x2d + y2d*y2d <= 1.0)

fig1, (ax1, ax2) = plt.subplots(1, 2)
x = np.linspace(0,1,500)
y = np.sqrt(1 - x*x)
ax1.fill_between(x,y, color='lightgray')
ax1.plot(x,y)
```

```
ax1.vlines(x=x1d, ymin=0*y1d, ymax=y1d, color='green')
ax1.scatter(x1d, 0*y1d)
ax1.axis('equal')
ax1.set_title("1D")

ax2.plot(x,y)
ax2.axis('equal')
ax2.scatter(x2d[np.logical_not(hit2d)], y2d[np.logical_not(hit2d)], color="orange", marker="
ax2.scatter(x2d[hit2d], y2d[hit2d], color="blue", marker=".")
ax2.set_title("2D")

plt.show()
```



Solution

- 1D is better, since it has a smaller variance, hence smaller MC error

- Try to make the variance as small as possible!

Exercise

Estimate an integral such as

$$I = \Gamma(z) := \int_0^\infty t^{z-1} e^{-t}\, dt$$

using MC.

Hint:

$$p(t) = e^{-t}$$

is the density of a distribution.

cdf:

$$F(t) = \int_0^t e^{-\tau}\, d\tau = 1 - e^{-t}$$

Solution

```python
import scipy.special
N = 100000
z = 0.5
u = np.random.rand(N)
t = - np.log(1-u)
f_t = t**(z-1)
I_MC = f_t.mean()
I_err = f_t.std()/np.sqrt(N)
print(f"I_MC = {I_MC} +/- {I_err}, I_exact = {scipy.special.gamma(z)}")
```

```
I_MC = 1.7712822043001943 +/- 0.013081502476711965, I_exact = 1.7724538509055159
```

Why use MC methods?

- MC error: $O(1/\sqrt{N})$
- To get 1 more digit correct, you need 100 times more simulations
- Terrible?

Deterministic alternative: Integration with Simpsons rule

- Error: $O(h^2)$ with grid size h
- For 1D: $h = 1/N$ on unit interval, where N is the number of function evaluations
- Hence error $= O(1/N^2)$
- Compare this to MC error: $O(1/\sqrt{N})$
- This looks really bad for MC!

The curse of dimensionality

- Assume you want to integrate in d dimensions, let's say on the unit hypercube
- Divide the unit hypercube into smaller hypercube with length h
- You have an integration method with error $O(h^2)$
- Number of grid points $= (1/h)^d$
- So you need $N = (1/h)^d$ function evaluations
- Error after N function evaluations: $O(h^2) = O(1/N^{2/d})$
- For $d = 4$, same order as MC
- Now imagine $d = 100$

High dimensional integrals

- For $d = 100$: $O(1/N^{2/d}) = O(1/N^{1/50})$
- Using 100 times more function evaluations makes your code run 100 times slower …
- … and reduces the error by the factor

```
100**(-1/50)
```

```
0.9120108393559098
```

- i.e. not even 10%
- compared to a factor 0.1 (90% error reduction) with MC
- To reduce the error by a factor of 0.1 with a regular grid, you need an T times longer computation with
$$T^{-1/50} = 0.1 \Rightarrow T = 10^{50}$$

- Slicing the hypercube in “only” 2 slices per dimension results in 2^d small hypercubes
- $2^{100} \approx 10^{30}$

```
2.0**100
```

```
1.2676506002282294e+30
```

Suddenly, MC doesn’t look so bad

- For many high-dimensional problems, MC is the **only** known viable method
- Often, MC is quite easy to implement

So, let’s do MC in high-dimensional spaces

- Test problem: Volume of the unit ball $B(d)$ in d dimensions
- Analytically known solution for comparison

d	vol B(d)
1	2
2	π
3	$4\pi/3$

- For general d

$$\mathrm{vol} B(d) = \frac{\pi^{d/2}}{\Gamma(d/2+1)}$$

where

$$\Gamma(x) = \int_0^\infty t^{x-1} e^{-t} dt$$

$$\Gamma(n+1) = n! \text{ for integer } n$$

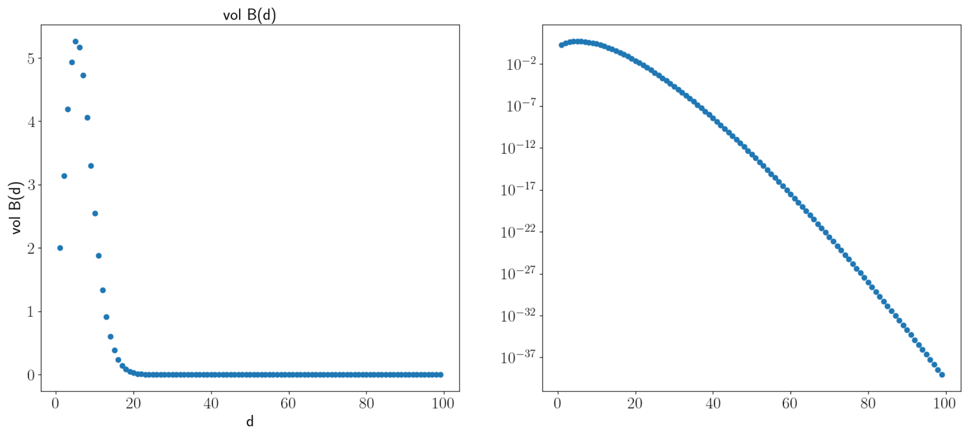
```
import scipy.special
def vol_exact(d):
    return np.pi**(d/2) / scipy.special.gamma(d/2 + 1)
```

```
d = 20
N = 1000000
x = np.random.rand(N,d)
hits = (np.sum(x*x,1) < 1).sum()
hits
vol_mc = 2**d * (hits/N)
print(f"vol_mc = {vol_mc}, vol_exact = {vol_exact(d)}, #hits = {hits} of {N}")
```

```
vol_mc = 0.0, vol_exact = 0.02580689139001405, #hits = 0 of 1000000
```

In high dimensions, the unit ball is tiny

```
dList = np.arange(1, 100)
volList = vol_exact(dList)
fig_gamma, (ax1, ax2) = plt.subplots(1, 2, figsize=(20,8))
ax1.scatter(dList, volList)
ax1.set_xlabel('d')
ax1.set_ylabel('vol B(d)')
ax1.set_title('vol B(d)')
ax2.scatter(dList, volList)
ax2.set_yscale('log')

for ax in (ax1, ax2):
    for item in ([ax.title, ax.xaxis.label, ax.yaxis.label] +
             ax.get_xticklabels() + ax.get_yticklabels()):
        item.set_fontsize(20)

plt.show()
```



Problem

- The probability to land a hit is tiny
- We are using an MC estimator for

$$I = \int f(x)p(x)dx$$

where

$$p(x) = 1$$

and

$$f(x) = \begin{cases} 1 & \text{if } |x|^2 < 1 \\ 0 & \text{otherwise} \end{cases}$$

What can we do?

- **Importance sampling**: Sample from a "wrong distribution" p_w with much more hits
- i.e. in regions where f is large
- Give the samples a weight to compensate for using the wrong density

$$I = E_p[f(X)] = \int f(x)p(x)dx = \int f(x)\frac{p(x)}{p_w(x)}p_w(x)dx = E_{p_w}[f(X)\frac{p(X)}{p_w(X)}]$$

Sample distribution

- Draw every component from a normal distribution with mean 0 and variance $1/d$

$$E[|x|^2] = d \cdot \frac{1}{d} = 1$$

- Density:

$$p_w(x) = \frac{1}{(2\pi d)^{d/2}} e^{-d|x|^2/2}$$

- MC estimator:

$$\hat{I}_{MC} = \frac{1}{N} \sum_{k=1}^{N} f(x_k) \frac{p(x_k)}{p_w(x_k)}$$

```
def p_Gauss(x, d):
    return np.exp(-(x*x).sum(1)*d/2)/(2*np.pi/d)**(d/2)

NMC = 200000
d = 100
x = np.random.randn(NMC, d)/np.sqrt(d)
hits = ( (x*x).sum(1) < 1 )
x = x[hits]
weights = 1/p_Gauss(x,d)
I_MC = weights.sum()/NMC
[I_MC, vol_exact(d), vol_exact(d)/I_MC]
```

```
[2.3818847616071623e-40, 2.368202101882829e-40, 0.9942555324485551]
```

Alternatives to Importance Sampling for variance reduction

- Control variates
- Antithetic variates
- Stratified sampling

In more complex cases

- Markov Chain Monte Carlo

Markov Chain Monte Carlo (MCMC) Methods

How to sample from more complex distributions.

```python
import plotly
import plotly.graph_objects as go
def f(x, y):
    return np.exp(x*np.sin(y) - x**2 - 0.3*x*y - 0.3*y**2)
NG = 100
x = np.linspace(-5, 5, NG)
y = np.linspace(-5, 5, NG)
xv, yv = np.meshgrid(x,y)
fv = f(xv,yv) + 0.3*f(xv - 3., (yv+2)/7) + 0.5*f((xv + 3.)/3, (yv+2)) + 0.3*f((yv-2)/7,xv + 
fig = go.Figure(data=[go.Surface(z=fv, x=xv, y=yv)])
fig.show()
```

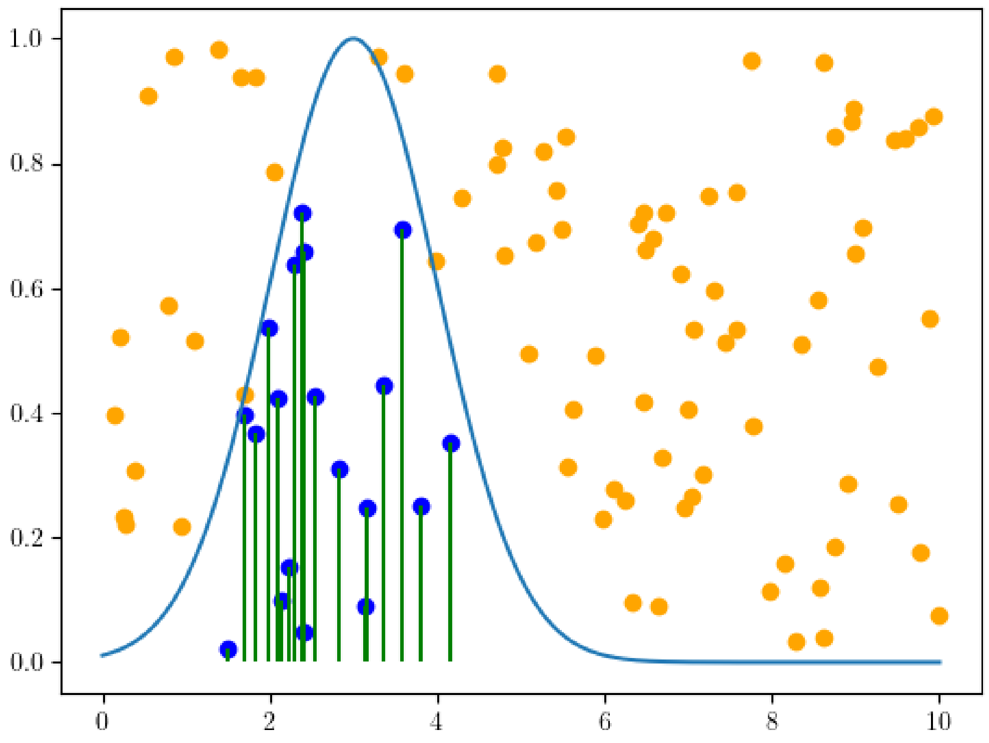
```
Unable to display output for mime type(s): text/html
```

Rejection method

- Goal: Sample from distribution with density $p(x)$ with $0 \leq p(x) \leq p_{max}$
  - Sample random x (let's say uniformly distributed)

- Sample y uniformly from the interval $[0, p_{max}]$
- Accept x if $y \leq p(x)$

```python
def f_1(x):
    return np.exp(-(x - 3)**2/2)  # Note: this is not normalized -- and this doesn't matter
NT = 100
x = np.linspace(0,10, 200)
f_1x = f_1(x)
x1r = np.random.rand(NT)*10
y1r = np.random.rand(NT)
f_1r = f_1(x1r)
fig = plt.figure()
accept = (y1r <= f_1(x1r))
reject = np.logical_not(accept)
plt.scatter(x1r[accept], y1r[accept], color="blue")
plt.scatter(x1r[reject], y1r[reject], color="orange")
plt.plot(x, f_1x)
plt.vlines(x=x1r[accept], ymin=0*y1r[accept], ymax=y1r[accept], color='green')
plt.show()
```



Is this a good method?

- Very easy to implement
- The normalization factor of the density doesn't matter / can be unknown

However, ...

- Very inefficient in region with small density
- Even for simple high-dimensional distributions, almost all points will be rejected

Alternatives?

- Possible to generalize, if e.g. a simpler majorant of the distribution can be found
- Again, this helps only up to small numbers of dimensions

Metropolis algorithm and importance sampling

Idea: Create a Markov chain that preferably samples in regions of large density * Start anywhere * Make a step in a random direction * Preferably walk into regions of higher probability

To sample from a distribution with density function $p(x)$: 1. Start with an arbitrary initial state x_0 2. To move from x_j to x_{j+1}: 2. Propose a new state x^* from a "proposal distribution" with a symmetric density $p(x^*|x_0) = p(x_0|x^*)$ 3. Accept new state if $p(x^*) > p(x_0)$, i.e. set $x_1 = x^*$ 4. If $p(x^*) < p(x_0)$: * Accept x^* (i.e. set $x_{j+1} = x^*$) with probability $p(x^*)/p(x_j)$ * Otherwise set $x_{j+1} = x_j$ 5. Repeat the procedure at step 2 to sample next point, until the chain is long enough

Properties of the Markov chain

If every possible proposal state x^* can be reached with finite probability (condition can be weakened): * the distribution of X_k converges to p for $k \to \infty$ and every choice of x_0, * ergodicity holds:

$$\lim_{n\to\infty} \frac{1}{n} \sum_{k=0}^{n} f(X_k) \to E_p[f(X)]$$

* for every starting value X_0, and every integrable function f. Note: X_k, X_{k+1} are not independent. Markov chain needs some time to "forget" starting point x_0 (burn-in time)

A glimpse at theory

- For $k \to \infty$, the Markov chain will reach a unique stationary distribution p^*
- If $p(x_{k+1}|x_k)$ is the transition probability density for a move from x_k to x_{k+1}, the densities $p_{k+1}(x_{k+1})$ and $p_k(x_k)$ are related by

$$p_{k+1}(x_{k+1}) = \int p(x_{k+1}|x_k) p_k(x_k) \, \mathrm{d}x_k$$

- The stationary distribution is invariant, i.e.

$$p^*(x) = \int p(x|y) p^*(y) \, \mathrm{d}y$$

Detailed balance condition

Sufficient (not necessary) condition that p^* is invariant under transition from y to x:

$$p(x|y)p^*(y) = p(y|x)p^*(x)$$

Proof:

$$\begin{aligned}\int p(x|y)p^*(y)\,\mathrm{d}y &= \int p(y|x)p^*(x)\,\mathrm{d}y \\ &= p^*(x)\int p(y|x)\,\mathrm{d}y = p^*(x)\end{aligned}$$

Why does the Metropolis algorithm converge?

- Transition density for a proposed step in Metropolis algorithm:

$$p(x^*|x_k) = \underbrace{\min\left(1, \frac{p(x^*)}{p(x_k)}\right)}_{\text{acceptance prob.}} \underbrace{\pi(x^*|x_k)}_{\text{proposal dens.}}$$

- This fulfils detailed balance, hence p is invariant distribution:

$$\begin{aligned}p(x^*|x_k)p(x_k) &= p(x_k)\min\left(1, \frac{p(x^*)}{p(x_k)}\right)\pi(x^*|x_k) \\ &= \min\left(p(x_k), p(x^*)\right)\pi(x^*|x_k) \qquad \text{everything symmetric!} \\ &= p(x_k|x^*)p(x^*)\end{aligned}$$

Example

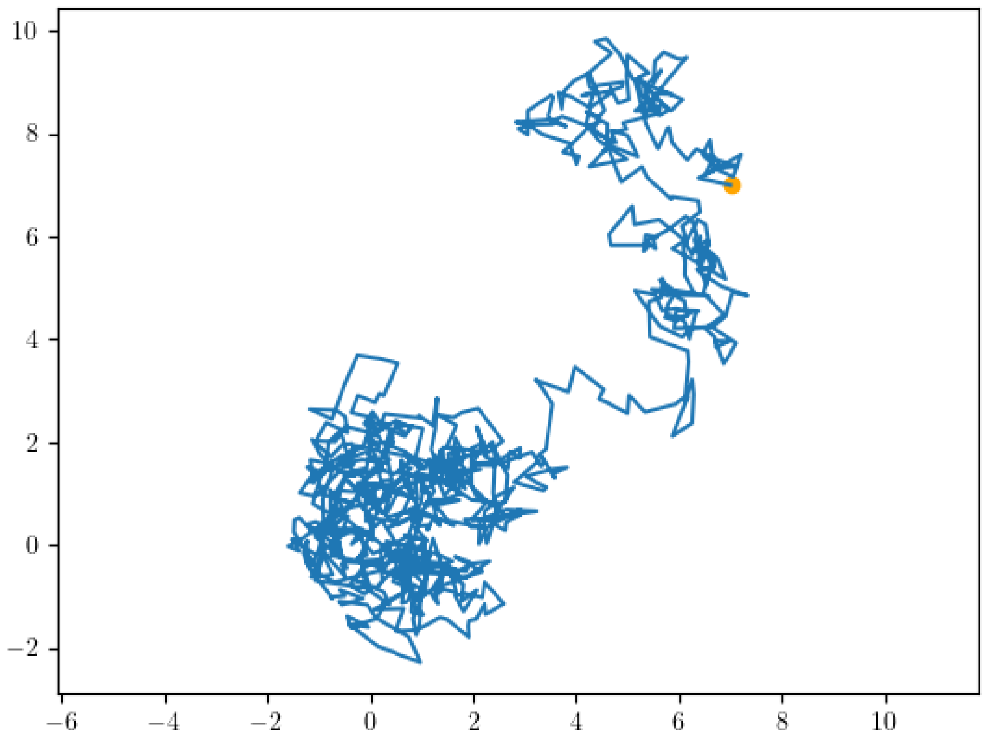
Density proportional to $\exp(-||x||_q)$ where $||x||_q = (|x_1|^q + |x_2|^q)^{1/q}$.

```python
plt.figure()
def unit_circle(q):
    plt.clf()
    x = np.linspace(-1, 1, 200)
    y = (1 - np.abs(x)**q)**(1/q)
    plt.plot(x,y, color='blue')
    plt.plot(x,-y, color='blue')
    plt.axis('equal')
    plt.show()
interact(unit_circle, q=(0.5, 3))
```

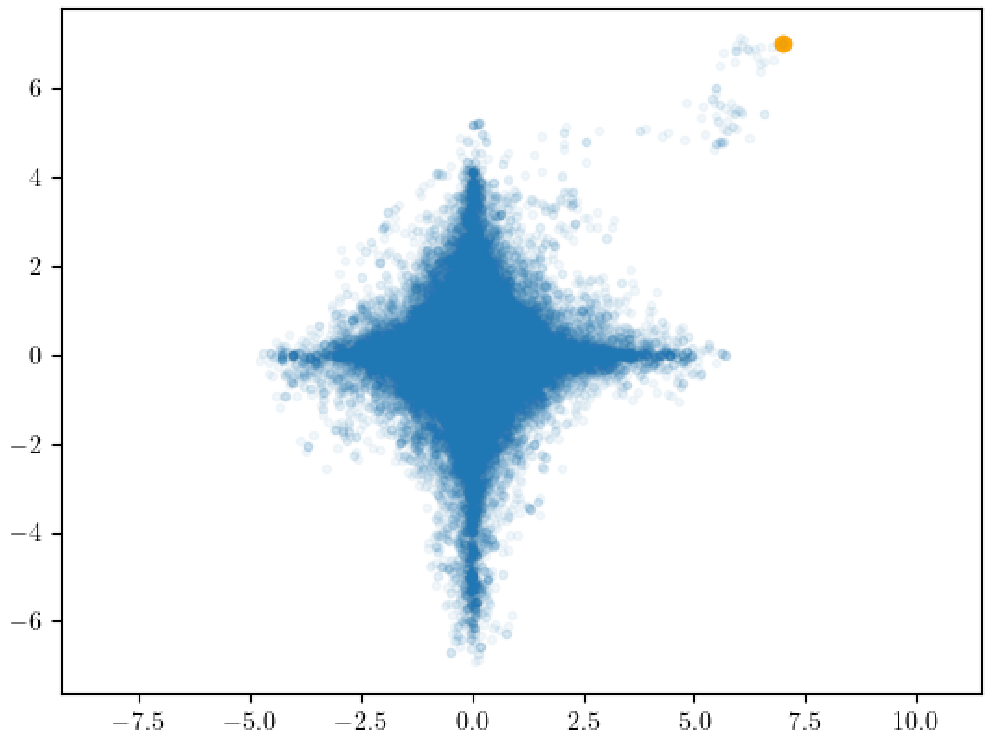
```
interactive(children=(FloatSlider(value=1.75, description='q', max=3.0, min=0.5), Output()),
```

```
<function __main__.unit_circle(q)>
```

```
def q_norm(x, q=2):
    return ((np.abs(x)**q).sum())**(1/q)

def p_q(x, q=2):
    return np.exp(-q_norm(x, q))

def metropolis_chain(NMC=1000, delta_x=0.3, x0=[7,7], q=2):
    x0 = np.array(x0)
    delta_x = 0.3
    chain = np.empty([NMC,2])
    chain[:] = np.nan,
    chain[0,:] = x0
    xk = x0
    for k in range(1, NMC):
        p_old = p_q(xk, q)
        x_prop = xk + delta_x * np.random.randn(2)
        p_new = p_q(x_prop, q)
        # Accept with probability min(1, p_new/p_old)
        if np.random.rand() < p_new/p_old:
            xk = x_prop
        chain[k, :] = xk
    return chain

chain = metropolis_chain()

plt.figure()
plt.scatter(chain[0,0], chain[0,1], color='orange')
plt.plot(chain[:,0], chain[:,1])
plt.axis('equal')
plt.show()
```



```
q = 3
chain = metropolis_chain(100000, 0.1, q=0.5)
plt.figure()
plt.scatter(chain[0,0], chain[0,1], color='orange')
plt.scatter(chain[:,0], chain[:,1], marker='.', alpha=0.05)
plt.axis('equal')
plt.show()
```



Remarks

- x_k, x_{k+1} are not independent
- For error estimation, wait until x_k, x_{k+J} become uncorrelated
- Always wait until the chain has “forgotten” its starting point (“burn-in time”)
- Most important statistic for convergence diagnostic: Gelman-Rubin diagnostic
- Chain may get caught in a local maximum
  - Possible solution: “Parallel Tempering”
  - Run multiple chains for densities proportional to p^β, $0 < \beta \leq 1$
  - Allow transitions between chains, fulfilling detailed balance

An important observation

- The chain depends only on the ratios $p(x^*)/p(x_k)$
- It is not necessary to know the normalization factor to compute expectations

- Getting the normalization factor is an important and notoriously difficult problem
  - Bayesian statistics:
    - Estimating the evidence
    - Model selection
  - Statistical physics
    - Partition function
- **MCMC allows estimation of normalization factors** by thermodynamic integration

Generalizations and alternative MCMC methods

- Asymmetric proposal distributions (Metropolis-Hastings)
- Gibbs sampling (special case of Metropolis)
- Exploiting gradient information
  - Metropolis adjusted Langevin algorithm (MALA)
  - Hamiltonian Monte Carlo (HMC)

Very good Python implementation: PyMC (https://github.com/pymc-devs/pymc)

Backup material

Central limit theorem at work

```
plt.figure()
def f(d):
    plt.clf()
    N = 1000000
    x = np.random.rand(N,d)
    xs = x.sum(axis=1)
    plt.hist(xs, bins=100, density=False)
    plt.show()
interact(f, d=widgets.IntSlider(min=1,max=12,value=1))
```

```
interactive(children=(IntSlider(value=1, description='d', max=12, min=1), Output()), _dom_cla
```

```
<function __main__.f(d)>
```