Problem 1)

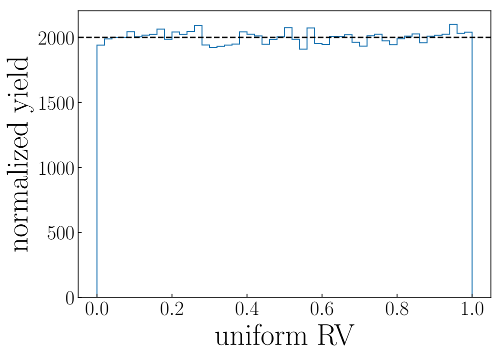
Exploring RNGs:

(a) Master using a random number generator available for you. Your preference should go, if possible, with the Mersenne Twister generator.

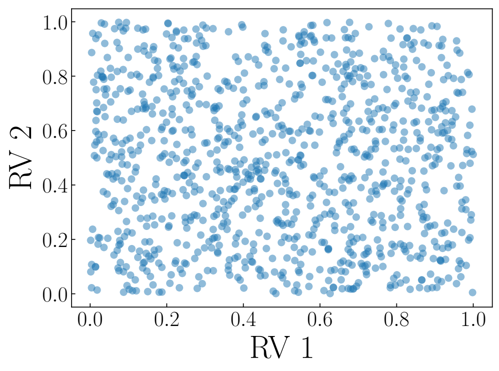
(b) Calculate 5th moment of the random number distribution and compare with the expected value. Explore how your agreement with the expected value changes as you increase the number of generated random numbers.

(c) Calculate the near-neighbor correlation for (x_i, x_{i+5}) and compare with the expected analytical value.

(a) For the purposes of this assignment, I use the default random number generator provided by the *numpy* package. A couple plots using the generator are shown in Fig. 1. Observe that this generator produces a fairly high quality sample of a uniform distribution without a visually obvious correlation between nearby sequence members.



(a)



(b)

Figure 1: (a) – Uniformity test for random number generator with 10^6 samples. **(b)** – Parking lot test for correlation between samples with 10^3 samples for each sequence.

(b–c) Using the sample from the random number generator described in part (a), we can calculate the n^{th} moment and near-neighbor correlation between x_i and x_{i+k} as

$$\langle x^n \rangle = \frac{1}{N} \sum_{i=0}^{N} x_i^n \tag{1}$$

$$\langle x_i x_{i+5} \rangle = \frac{1}{N} \sum_{i=0}^{N-k} x_i x_{i+k}, \tag{2}$$

respectively. Note that the expected values of such moments and correlations for samples pulled from a uniform distribution are

$$E[x^n] = \int_0^1 \mathrm{d}x\, x^n = \frac{1}{n+1} \tag{3}$$

$$E[x_i x_{i+k}] = \int_0^1 \mathrm{d}x_1 \int_0^1 \mathrm{d}x_2\, x_1 x_2 = \frac{1}{4}. \tag{4}$$

From the plots, we can see that as the sample size becomes larger, we approach the random number generator produces fifth moment and correlation values close to the expected values with only a small amount of noise.

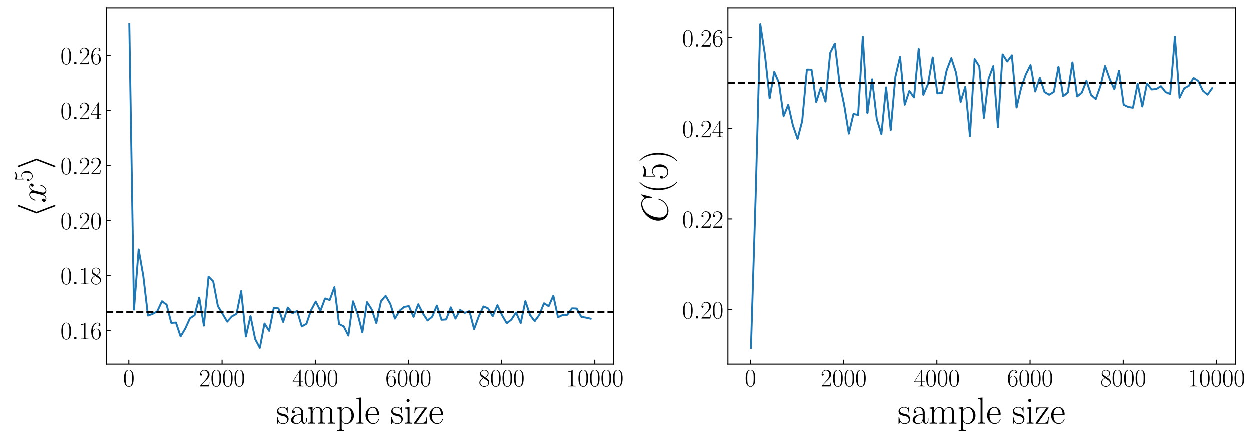


Figure 2: Left – Fifth moment of a sample of a uniform random variable. Note the dashed black line indicates the expected value, assuming such a uniform distribution. **Right** – Near-neighbor correlation between x_i and x_{i+5} for the same sample of a uniform random variable. Again, the dashed black line indicates the expected value.

```
import numpy as np
from scipy.optimize import root_scalar,fsolve
from scipy.integrate import quad

import matplotlib.pyplot as plt
plt.rcParams.update({
    'text.latex.preamble': r'\usepackage{amsmath}',
    'text.usetex': True,
    'font.family': 'sans-serif',
    'font.sans-serif': ['Helvetica']
})


def get_mom(sample,n=1):
    return np.sum(sample**n)/sample.shape[0]

def get_nn_corr(sample,k=1):
    return np.sum(sample[:-k]*sample[k:])/sample.shape[0]



if __name__ == '__main__':
```

```
rng = np.random.default_rng(seed=12345)

N = 100000
sample = np.array([rng.random() for i in range(N)])


# one sequence uniformity
nrows,ncols = 1,1
fig,ax = plt.subplots(nrows=nrows,ncols=ncols,figsize=(7*ncols,5*nrows)
)

bins = 50
counts,edges = np.histogram(sample,bins=bins,density=False)
ax.stairs(counts,edges)

ax.set_xlabel(r'$\rm uniform~RV$',size=30)
ax.set_ylabel(r'$\rm normalized~yield$',size=30)
ax.tick_params(which='major',axis='both',labelsize=20,direction='in')
ax.axhline(N/bins,color='k',ls='--')

plt.tight_layout()
# plt.show()
fig.savefig(r'prob1_uniformity.pdf',bbox_inches='tight')


# parking lot
N = 1000
sample1 = np.array([rng.random() for i in range(N)])
sample2 = np.array([rng.random() for i in range(N)])

nrows,ncols = 1,1
fig,ax = plt.subplots(nrows=nrows,ncols=ncols,figsize=(7*ncols,5*nrows)
)

ax.scatter(sample1,sample2,alpha=0.5)
ax.set_xlabel(r'\rm RV~1',size=30)
ax.set_ylabel(r'\rm RV~2',size=30)
ax.tick_params(axis='both',which='major',direction='in',labelsize=20)

# plt.show()
fig.savefig(r'prob1_parking-lot.pdf',bbox_inches='tight')


# uniform sample 5th moment and near-neighbor correlation
N = np.arange(10,10000,100)

n = 5
k = 5
mom  = []
corr = []
for _ in N:
    temp = np.array([rng.random() for i in range(_)])
    mom.append(get_mom(temp,n=n))
```

```
        corr.append(get_nn_corr(temp,k=k))

    nrows,ncols=1,2
    fig,ax = plt.subplots(nrows=nrows,ncols=ncols,figsize=(7*ncols,5*nrows)
)

    ax[0].plot(N,mom)
    ax[0].axhline(1/(k+1),color='k',ls='--')

    ax[1].plot(N,corr)
    ax[1].axhline(1/4,color='k',ls='--')

    for i in range(2):
        ax[i].set_xlabel(r'$\rm sample~size$',size=30)
        ax[i].tick_params(axis='both',which='major',direction='in',
labelsize=20)
        ax[0].set_ylabel(r'$\langle x^{%d} \rangle$'%n,size=30)
        ax[1].set_ylabel(r'$C(%d)$'%k,size=30)

    plt.tight_layout()
    # plt.show()
    fig.savefig(r'prob1_mom_corr.pdf',bbox_inches='tight')
```

Problem 2)

Generating non-uniform distribution:

(a) Write a code that can generate non-uniform distributions of random numbers based on one, or two or all three of the methods, namely, the rejection method, the transformation method (when possible), and the Metropolis algorithm (importance sampling)

(b) Use your code to generate the following non-uniform distributions:

(a) $p(y) = \frac{1}{a}\exp(-y/a)$ (Poisson distribution)

(b) $p(y) = \frac{1}{\pi}\frac{a}{a^2+y^2}$ (Cauchy-Lorentz distribution)

(c) $p(y) = \frac{1}{\sigma\sqrt{2\pi}}\exp\left(-\frac{1}{2}(\frac{y-\mu}{\sigma})^2\right)$

(c) Analyze the quality of your distributions in any way you find appropriate.

(d) Explore if any of of the above distributions are available to you with either C++, or Python, or MatLab libraries.

(a) A script which implements the transform, rejection, and Metropolis sampling algorithms is shown below. The transform method is simply implemented when the inverse CDF, $F^{-1}(x;\boldsymbol{a})$ (where $\boldsymbol{a}$ are distribution parameters), of a random variable is known. Since the output of a CDF is between 0 and 1, we can sample a uniform random variable

u and construct the desired random variable x via

$$x = F^{-1}(u; \boldsymbol{a}). \tag{5}$$

Of course, there are other numerical schemes we can implement to approximate the inverse CDF. For example, we can solve the problem

$$u = \int_{-\infty}^{x} \mathrm{d}t\, f(t; \boldsymbol{a}) \tag{6}$$

directly, where u is the sampled uniform random variable, and x is the desired random variable distributed according to the density $f(x; \boldsymbol{a})$. However, because in most cases, we require a numerical scheme to compute the integral and find the root x corresponding to u, such an approach is not always more efficient than other more direct methods of sampling the PDF.

For the rejection method, we uniformly sample points $(x, y) \in [a, b] \times [c, d]$ and keep the sample if $|y| < |f(x; \boldsymbol{a})|$.

Lastly, for the Metropolis algorithm, we choose a starting point x_0 and iterate according to the recursive formula

$$x_{i+1} = x_i + \delta(2u - 1), \tag{7}$$

where $u \in [0, 1]$ is uniformly sampled. This point is added to the sample if $f(x_{i+1}; \boldsymbol{a})/f(x_i; \boldsymbol{a}) \geq v$, where $v \in [0, 1]$ is uniformly sampled, and the process is repeated until the desired number of samples is achieved.

```python
import numpy as np

import matplotlib.pyplot as plt
plt.rcParams.update({
    'text.latex.preamble': r'\usepackage{amsmath}',
    'text.usetex': True,
    'font.family': 'sans-serif',
    'font.sans-serif': ['Helvetica']
})

def transform_sample(N,F_inv):
    u = np.random.uniform(size=N)
    sample = F_inv(u)
    return sample

def rejection_sample(N,f,x_ext,y_ext,verb=0):
    x   = np.random.uniform(*x_ext,size=N)
    y   = np.random.uniform(*y_ext,size=N)
    f_  = f(x)
    diff = f_ - y
    sample = x[diff>=0]
    if verb:
        print(f'rejection rate: {1 - sample.shape[0]/N}')
    return sample

def metropolis_sample(N,f,x0=0,delta=1,bounds=[-np.inf,np.inf],verb=0):
    sample = [x0]
    f_     = [f(x0)]

    count = 0
    while len(sample) <= N:
        x1 = sample[-1] + delta*(2*np.random.uniform() - 1)

        if x1 < bounds[0] or x1 > bounds[1]:
            continue

        f1 = f(x1)
        r  = f1/f_[-1]
        u  = np.random.uniform()
        if u <= r:
            sample.append(x1)
            f_.append(f1)

        count += 1

    sample = np.array(sample)
    if verb:
        print(f'rejection rate: {(count - sample.shape[0])/N}')

    return sample

if __name__ == '__main__':

    sample_dict = {}
    N = 100000


    func_a = lambda t: np.exp(-t)*(t > 0)
    sample_dict['Poisson'] = {}
    sample_dict['Poisson']['transform'] = transform_sample(N,F_inv=lambda t
    : -np.log(1-t))
    sample_dict['Poisson']['rejection'] = rejection_sample(N,func_a
    ,[0,10],[0,1],verb=1)
    sample_dict['Poisson']['metropolis'] = metropolis_sample(N,func_a,x0=1,
    delta=0.1,verb=1)

    print()

    func_b = lambda t: 1/(np.pi*(1+t**2))
    sample_dict['Cauchy'] = {}
    sample_dict['Cauchy']['transform'] = transform_sample(N,F_inv=lambda t:
     np.tan(np.pi*(t-0.5)))
    sample_dict['Cauchy']['rejection'] = rejection_sample(N,func_b
    ,[-10,10],[0,1/np.pi],verb=1)
    sample_dict['Cauchy']['metropolis'] = metropolis_sample(N,func_b,x0=0,
    delta=1,verb=1)
```

```
 print()

 func_c = lambda t: np.exp(-t**2/2)/np.sqrt(2*np.pi)
 sample_dict['Gauss'] = {}
# sample_dict['Gauss']['transform'] = transform_sample(N,F_inv=lambda t
: np.tan(np.pi*(t-1)/2))
 sample_dict['Gauss']['rejection'] = rejection_sample(N,func_c
,[-10,10],[0,1/np.sqrt(2*np.pi)],verb=1)
 sample_dict['Gauss']['metropolis'] = metropolis_sample(N,func_c,x0=0,
delta=1,verb=1)


 nrows,ncols = 1,3
 fig,ax = plt.subplots(nrows=nrows,ncols=ncols,figsize=(7*ncols,5*nrows)
)


 Range = {'Poisson': (0,5), 'Cauchy': (-5,5), 'Gauss': (-5,5)}
 colors = {'transform':'r','rejection':'b','metropolis':'g'}
 for i,dist in enumerate(sample_dict):
     for j,method in enumerate(sample_dict[dist]):
         sample = sample_dict[dist][method]
         counts,edges = np.histogram(sample,bins=50,range=Range[dist],
density=True)
         ax[i].stairs(counts,edges,color=colors[method],lw=2,alpha=0.5,
label=r'$\rm %s$'%method)

 x = np.linspace(1e-15,5,1000)
 ax[0].plot(x,func_a(x),'k--',alpha=0.5)

 x = np.linspace(-5,5,1000)
 ax[1].plot(x,func_b(x),'k--',alpha=0.5)
 ax[1].set_xlim(-5,5)

 x = np.linspace(-5,5,1000)
 ax[2].plot(x,func_c(x),'k--',alpha=0.5)
 ax[2].set_xlim(-5,5)


 ax[0].text(s=r'\boldmath $\rm Poisson$',x=0.95,y=0.95,size=30,va='top',
ha='right',transform=ax[0].transAxes)
 ax[1].text(s=r'\boldmath $\rm Cauchy$',x=0.95,y=0.95,size=30,va='top',
ha='right',transform=ax[1].transAxes)
 ax[2].text(s=r'\boldmath $\rm Gaussian$',x=0.95,y=0.95,size=30,va='top'
,ha='right',transform=ax[2].transAxes)

 ax[0].legend(fontsize=25,loc='center right',frameon=False)
 ax[0].set_ylabel(r'$f(x)$',size=30)
 for i in range(3):
     ax[i].tick_params(axis='both',which='major',direction='in',
labelsize=20)
     ax[i].set_xlabel(r'$x$',size=30)
```

```
    plt.tight_layout()
    # plt.show()
    fig.savefig(r'prob2.pdf',bbox_inches='tight')
```

(b) In Fig. 3, we show samples of the Poisson, Cauchy-Lorentz, and Gaussian distributions are shown. The following list details the precise setups used to produce the samples.

- Poisson:
  - Rejection: sample on $[0, 10] \times [0, 1]$ with a rejection rate of 90%
  - Metropolis: sample with starting point $x_0 = 1$ with a step size of $\delta = 0.1$ resulting in a rejection rate of 5%
- Cauchy-Lorentz:
  - Rejection: sample on $[-10, 10] \times [0, 1/\pi]$ with a rejection rate of 85%
  - Metropolis: sample with starting point $x_0 = 0$ with a step size of $\delta = 1$ resulting in a rejection rate of 18%
- Gaussian:
  - Rejection: sample on $[-10, 10] \times [0, 1/\sqrt{2\pi}]$ with a rejection rate of 88%
  - Metropolis: sample with starting point $x_0 = 0$ with a step size of $\delta = 1$ resulting in a rejection rate of 25%

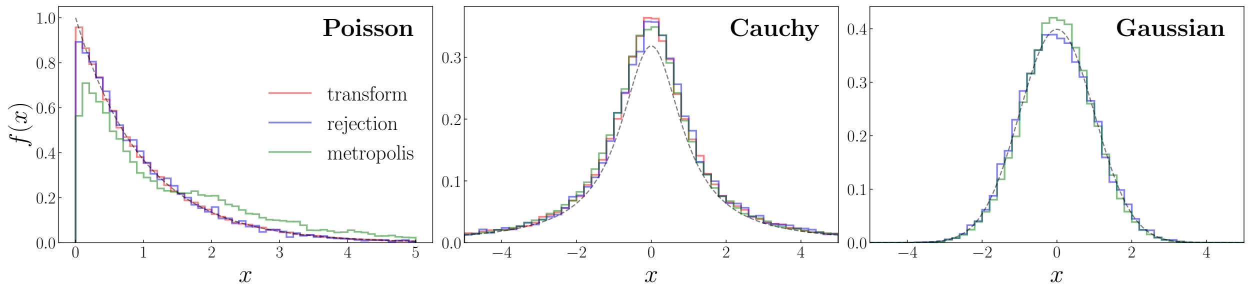


Figure 3: Samples ($N = 10^5$), collected into histograms with 50 bins across the selected ranges, for the Poisson ($a = 1$), Cauchy-Lorentz ($a = 1$), and Gaussian probability distributions ($\mu = 0, \sigma = 1$). Note that we only sample the standard distributions, which can be easily transformed to any linearly related random variable of interest. Additionally, note that the we omit the transform sampling for the standard normal distribution since there is no simple closed form for such an inverse CDF. For comparison, a grey curve representing the PDFs in each panel is inserted.

(c) For the Poisson distribution, the transform and rejection sampling seem to provide the best reconstruction. This seems to be primarily because the PDF can be represented on the entire real line as $f(x;a) = \Theta(x)\exp(-x/a)/a$, where $\Theta(x)$ is the Heaviside step function. In order to accurately construct this sharp discontinuity at $x = 0$, we should decrease the step size so that more samples do not cross into the negative region. In turn, the number of samples should go up to compensate such a small step size, which on average decreases the amount of sample space explored per step. Because the Poisson distribution is quite simple, the rejection method may be sufficient to produce a high fidelity Poisson distribution, but in general, it is much more wasteful than the Metropolis scheme.

For the Cauchy distribution, all methods seem to perform roughly equivalently. It can be observed that the sampling seems to overestimate the density around $x = 0$. While it was not tested, this issue may be resolved by increasing the sample size or increasing the number of bins such that the resolution around $x = 0$ is improved.

Lastly, for the Gaussian distribution, we obtained samples through the rejection and metropolis algorithms, producing fairly good reconstructions of the the density. It appears that the rejection method performs a bit better around $x = 0$ than the metropolis algorithm, but the comments made for the Cauchy distribution should also be applied here as well, that increasing the sample size or increasing the number of bins may help agreement with the PDF.

(d) Note that all the distributions here are implemented in *scipy.stats* module, which is very useful. Essentially, one creates an instance of a random variable with the desired probability distribution implemented in the library and can construct explicitly the PDF or CDF or the random variable, sample from the distribution, and more. Additionally, because of its ubiquitous nature, the (multivariate) Gaussian random number generator is implemented in the *numpy.random* module.

Problem 3)

Evaluate the following integrals using the two above methods (the mean value and rejection) for various numbers of points $N = 10, 10^2, 10^3, 10^4, 10^5$. Evaluate the errors and explain your results.

(a) $\int_0^\pi \sin x \, \mathrm{d}x$

(b) $\int_0^1 \frac{\mathrm{d}x}{1-0.998x^2}$

(c) $\int_0^{2\pi} x \sin(12x)\cos(24x)$

(d) $\int_0^2 \sin^2[\frac{1}{x(2-x)}]$

Suppose we wish to evaluate a generic integral

$$I = \int_a^b f(x)\,\mathrm{d}x\,. \tag{8}$$

For the rejection method, we determine a bounding interval $[c, d]$ for the function $f(x)$ over the interval $[a, b]$ and uniformly sample points (x, y) within this rectangular domain. From these samples, we count the number n_+ of samples satisfying $0 < y < f(x)$ and n_- of samples satisfying $f(x) < y < 0$, and from these counts, we obtain $I \approx (n_+ - n_-)/[(b-a)(d-c)]$. Next, for the mean method, we uniformly sample $x \in [a, b]$ and obtain samples $f(x)$ and approximate

$$I \approx \frac{1}{N}\sum_{i=1}^{N} f(x_i), \tag{9}$$

which is a simple result presented in a typical first-year calculus course.

In general, the rejection method is more wasteful and seems to require many more attempts to approximate the integral to the same degree of accuracy. However, there are more clever methods one can use than the mean value prescription to approximate the integral. Because we are not interested in obtaining samples of x over the integration range but rather samples of the function $f(x)$, we can implement schemes to perform importance sampling which should require less samples to produce results of the same accuracy. Additionally, for one-dimensional quadrature, there are schemes such as (fixed or adaptive) Gaussian quadrature, which are quite efficient and accurate for a wide class of functions, requiring many less function calls to compute an approximate value for I.

Digressing, in Fig. 4, we show the results of the code shown below, which implements the rejection and mean value methods to compute the integrals in the problem statement above for a selection of sample sizes. Note that there are no statistical error bars shown on the plots. For the rejection method, we could produce these by performing many iterations of sampling at each N and report the standard deviation as the standard error. For the mean value method, we could perform iterations of the sampling to obtain an error, but it is actually a fairly simple procedure to propagate statistical errors here. Assuming the samples of x are uncorrelated, we have

$$\delta I = \frac{\delta x}{\sqrt{N}}\sqrt{\sum_{i=1}^{N}[f'(x_i)]^2}. \tag{10}$$

Note that

$$\delta x = \langle x^2 \rangle - \langle x \rangle^2 = \frac{1}{3} - \frac{1}{4} = \frac{1}{12}. \tag{11}$$

Thus, if our function is generally smooth, our statistical precision should be fairly high given a large enough sample size. That is, while a single sample for $N = 10, 100$ are not very stable, around $N \geq 1000$, for the functions shown, the results are generally stable.

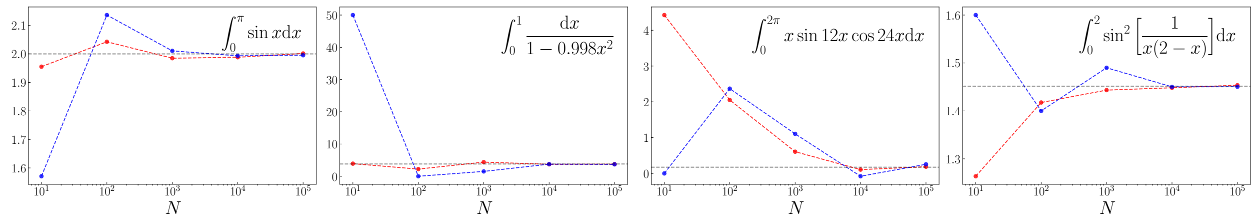


Figure 4: Results for one iteration of rejection (blue) and mean value (red) monte carlo quadrature.

```
import numpy as np
from scipy.integrate import quad

import matplotlib.pyplot as plt
plt.rcParams.update({
    'text.latex.preamble': r'\usepackage{amsmath}',
    'text.usetex': True,
    'font.family': 'sans-serif',
    'font.sans-serif': ['Helvetica']
})


def mean_quad(f,a,b,N=100,err=False):
    x   = np.random.uniform(a,b,size=N)
    f_  = f(x)
    ave_f = np.sum(f_)/N
    res    = (b-a)*ave_f
    if err:
        ave_f2 = np.sum(f_**2)/N
        error  = np.sqrt((ave_f2 - ave_f)/N)
        return res,error
    else:
        return res

def rejection_quad(f,x_bounds,y_bounds,N=100,verb=0):
    x   = np.random.uniform(*x_bounds,size=N)
    y   = np.random.uniform(*y_bounds,size=N)
    f_  = f(x)

    cond1 = np.logical_and(y > 0,y < f_)
    cond2 = np.logical_and(y < 0, y > f_)
    r     = (np.sum(cond1) - np.sum(cond2))/N
    A_box = (x_bounds[1] - x_bounds[0])*(y_bounds[1] - y_bounds[0])

    if verb:
        # print(cond1)
        # print(cond2)
        print(f'rejection rate: {(np.sum(cond1)+np.sum(cond2))/N}')

    return r*A_box

if __name__ == '__main__':
```

```
N = np.array([10,100,1000,10000,100000])

funcs = [
    lambda t: np.sin(t),
    lambda t: 1/(1-0.998*t**2),
    lambda t: t*np.sin(12*t)*np.cos(24*t),
    lambda t: np.sin(1/t/(2-t))**2
]
bounds = [
    [0,np.pi],
    [0,1],
    [0,2*np.pi],
    [0,2]
]
boxes = [
    [0,1],
    [0,1/(1-0.998)],
    [-2*np.pi,2*np.pi],
    [0,1]
]

results = {}
results['mean'] = []
results['rejection'] = []
results['exact'] = []

for i in range(4):
    func = funcs[i]
    results['mean'].append([mean_quad(func,*bounds[i],N=_) for _ in N])
    results['rejection'].append([rejection_quad(func,bounds[i],boxes[i
],N=_,verb=1) for _ in N])

    results['exact'].append(quad(func,*bounds[i])[0])

    print()

nrows,ncols=1,4
fig,ax = plt.subplots(nrows=nrows,ncols=ncols,figsize=(7*ncols,5*nrows)
)

expr = [
    r'$\displaystyle \int_{0}^{\pi} \sin{x} {\rm d}{x}$',
    r'$\displaystyle \int_{0}^{1} \frac{{\rm d}{x}}{1-0.998 x^2}$',
    r'$\displaystyle \int_{0}^{2 \pi} x \sin{12 x} \cos{24 x} {\rm d}{x
}$',
    r'$\displaystyle \int_{0}^{2} \sin^2\Big[ \frac{1}{x(2-x)} \Big] {\
rm d}{x}$'
]
for i in range(4):
    temp = results['mean'][i]
    ax[i].plot(N,temp,marker='o',ls='--',alpha=0.8,color='r')

    temp = results['rejection'][i]
    ax[i].plot(N,temp,marker='o',ls='--',alpha=0.8,color='b')
```

```
        ax[i].axhline(y=results['exact'][i],color='k',ls='--',alpha=0.5)

        ax[i].tick_params(axis='both',which='major',direction='in',
labelsize=20)
        ax[i].set_xlabel(r'$N$',size=30)
        ax[i].semilogx()

        ax[i].text(s=expr[i],x=0.95,y=0.95,va='top',ha='right',transform=ax
[i].transAxes, size=30)

 plt.tight_layout()
# plt.show()
 fig.savefig(r'prob3.pdf',bbox_inches='tight')
```

Problem 4)

Compute the following integrals:

(a) A double integral over a rectangular region

$$\int_0^1 \int_0^2 \sin\left(x^2 + y^2\right) \mathrm{d}x\, \mathrm{d}y\,. \tag{12}$$

(b) A double integral over a circular region:

$$\int_{\text{circle}} e^{-(x^2+y^2)} \,\mathrm{d}x\, \mathrm{d}y\,, \tag{13}$$

where the circle is centered at the origin with radius 1.

(c) A double integral over a non-rectangular region

$$\int_0^1 \int_0^{1-x} (x + y)\, \mathrm{d}y\, \mathrm{d}x\,. \tag{14}$$

(d) Four-dimensional integral

$$\int_0^1 \int_0^1 \int_0^1 \int_0^1 \int_0^1 e^{-(x_1^2+x_2^2+x_3^2+x_4^2)} \,\mathrm{d}x_1\, \mathrm{d}x_2\, \mathrm{d}x_3\, \mathrm{d}x_4\,. \tag{15}$$

(e) Four-dimensional integral over a spherical region

$$I = \int_{\text{sphere}} (x^2 + y^2 + z^2 + w^2) e^{-(x^2+y^2+z^2+w^2)} \,\mathrm{d}x\, \mathrm{d}y\, \mathrm{d}z\, \mathrm{d}w\,. \tag{16}$$

(a) The integral here is performed by producing a uniform sample of $N = 10^6$ points

(x, y) over the rectangular domain $[0,1]\times[0,2]$ and averaging the function values $f(x_i, y_i)$, yielding

$$\int_0^1 \mathrm{d}x \int_0^2 \mathrm{d}y \sin\bigl(x^2+y^2\bigr) \approx 0.435. \tag{17}$$

(b) The integral here is performed by uniformly sampling $N = 10^6$ points in the circular domain specified by the condition $r = \sqrt{x^2+y^2} \leq 1$. This is done by uniformly sampling $(u, \theta) \in [0,1]\times[0,2\pi]$ and computing

$$x = \sqrt{u}\cos\theta \tag{18}$$

$$y = \sqrt{u}\sin\theta. \tag{19}$$

Again, once these samples are produced, we average over the function values, yielding

$$\int_{r\leq 1} e^{-(x^2+y^2)}\,\mathrm{d}x\,\mathrm{d}y \approx 0.632. \tag{20}$$

(c) The integral here is performed by producing a uniform sample of $N = 10^6$ points (x, y) in the rectangular domain $[0,1]^2$ and writing

$$\int_0^1 \mathrm{d}x \int_0^{1-x} \mathrm{d}y\,(x+y) = \int_0^1 \mathrm{d}x \int_0^1 \mathrm{d}y\,(x+y)\Theta(x+y\leq 1). \tag{21}$$

Thus, we convert the condition on the integral bounds to one inside the function definition. Obviously, this is equivalent to rejecting points outside the integration region but upon averaging produces the approximate value

$$\int_0^1 \mathrm{d}x \int_0^{1-x} \mathrm{d}y\,(x+y) \approx 0.667. \tag{22}$$

(d) The integral here is performed by producing a uniform sample of $N = 10^6$ points (x_1, x_2, x_3, x_4) in the domain $[0,1]^4$ and averaging over the function values to obtain

$$\int_0^1 \mathrm{d}x_1 \int_0^1 \mathrm{d}x_2 \int_0^1 \mathrm{d}x_3 \int_0^1 \mathrm{d}x_4\, e^{-(x_1^2+x_2^2+x_3^2+x_4^2)} \approx 0.311. \tag{23}$$

(e) The integral here is performed by producing a uniform sample of $N = 10^6$ points (x_1, x_2, x_3, x_4) as in the previous part. From this sample, we excise those points which do not satisfy $r = \sqrt{x_1^2+x_2^2+x_3^2+x_4^2} \leq 1$ and average over the rest to obtain

$$\int_{r\leq 1} \mathrm{d}x_1\,\mathrm{d}x_2\,\mathrm{d}x_3\,\mathrm{d}x_4\,(x_1^2+x_2^2+x_3^2+x_4^2)e^{-(x_1^2+x_2^2+x_3^2+x_4^2)} \approx 0.316. \tag{24}$$

While, I did not investigate, there may be a way to generate a uniform sample in "hyperspherical" coordinates and transform back to Cartesian coordinates before averaging over function values, allowing one to avoid wasting any samples.

```
import numpy as np

# 4.1
N = int(1e6)
x = np.random.uniform(size=N)
y = 2*np.random.uniform(size=N)
res = np.mean(np.sin(x**2+y**2))
print(res)

# 4.2
N = int(1e6)
r  = np.random.uniform(size=N)
th = 2*np.pi*np.random.uniform(size=N)
x,y = np.sqrt(r)*np.cos(th),np.sqrt(r)*np.sin(th)
res = np.mean(np.exp(-(x**2+y**2)))
print(res)

# 4.3
N = int(1e6)
x = np.random.uniform(size=N)
y = np.random.uniform(size=N)
cond = y < 1-x
x,y = x[cond],y[cond]
res = np.mean(x+y)
print(res)

# 4.4
N  = int(1e6)
x1 = np.random.uniform(size=N)
x2 = np.random.uniform(size=N)
x3 = np.random.uniform(size=N)
x4 = np.random.uniform(size=N)
res = np.mean(np.exp(-(x1**2 + x2**2 + x3**2 + x4**2)))
print(res)

# 4.5
N  = int(1e6)
x1 = np.random.uniform(size=N)
x2 = np.random.uniform(size=N)
x3 = np.random.uniform(size=N)
x4 = np.random.uniform(size=N)
cond = x1**2 + x2**2 + x3**2 + x4**2 < 1
res = np.mean((x1**2 + x2**2 + x3**2 + x4**2)*np.exp(-(x1**2 + x2**2 + x3
    **2 + x4**2)))
print(res)
```