AI505/AI801, Optimization – Exercise Sheet 06

2026-03-22

Solutions included.

Exercises with the symbol $^+$ are to be done at home before the class. Exercises with the symbol * will be tackled in class. The remaining exercises are left for self training after the exercise class. Some exercises are from the text book and the number is reported. They have the solution at the end of the book.

Exercise 1 *

Using real-valued encoding for the solution representation, implement in Python a plain version of one of the following metaheuristic optimization algorithms:

- Genetic Algorithms (GA),
- Differential Evolution (DE),
- Particle Swarm Optimization (PSO).

Then, modify your implementation to produce a hybrid version of the algorithm by adding a local search method. You can use any method for the exploitation of design points studied in the previous parts of this course and Lamarkian or Baldwinian learning.

Test and compare your implementations of the plain and the hybrid versions on a few benchmark functions from the module landscapes.

Below an implementation of a Genetic Algorithm.

```python
import numpy as np
from typing import Callable
import random

functype = Callable[[np.ndarray], float]

def ininitialize_population(pop_size: int, dim: int, bounds: tuple) -> list[np.ndarray]:
    return [np.random.uniform(bounds[0], bounds[1], dim) for _ in range(pop_size)]


def truncation_selection(population: list[np.ndarray], fitness: list[float], k: int, \
                            num_pairs: int) -> list[list[np.ndarray]]:
    sorted_indices = list(np.argsort(fitness))
    return [[population[i] for i in random.sample(sorted_indices[:k], 2)]
                for _ in range(num_pairs)]


def crossover_single_point(parent1: np.ndarray, parent2: np.ndarray) -> np.ndarray:
    point = np.random.randint(1, len(parent1) - 1)
    child = np.concatenate((parent1[:point], parent2[point:]))
    return child

def crossover_convex_comb(parent1: np.ndarray, parent2: np.ndarray) -> np.ndarray:
    alpha = np.random.rand()
    child = alpha * parent1 + (1 - alpha) * parent2
    return child

def mutate(individual: np.ndarray, mutation_rate: float) -> np.ndarray:
    for i in range(len(individual)):
        if np.random.rand() < mutation_rate:
            individual[i] = random.gauss(mu=0,sigma=1)
    return individual

def select_new_generation(population: list[np.ndarray], fitness: list[float], \
                            m: int) -> tuple[list[np.ndarray], list[float]]:
    # implements elitist strategy: keep the best m individuals
    sorted_indices = list(np.argsort(fitness))
    new_population = [population[i] for i in sorted_indices[:m]]
    new_fitness = [fitness[i] for i in sorted_indices[:m]]
    return new_population, new_fitness

def genetic_algorithm(f: functype, population: list[np.ndarray], k_max: int=10,
                        m: int=100) -> tuple[np.ndarray, float]:
    sp = population
    sp_y=[f(x) for x in sp]
    x_best = sp[np.argmin(sp_y)]
    y_best = min(sp_y)

    for _ in range(k_max):
        parents = truncation_selection(sp, sp_y, 10, 10) # select top 10
        children = [crossover_convex_comb(p[0],p[1]) for p in parents]
        mutations = [mutate(child, 0.1) for child in children]

        sp, sp_y = select_new_generation(sp + children + mutations,
                                    list(sp_y) + [f(child) for child in children]
                                            + [f(mut) for mut in mutations], m)

        if min(sp_y) < y_best:
            x_best = sp[np.argmin(sp_y)]
            y_best = min(sp_y)
    return x_best, y_best
```

Let's test it on some benchmark functions:

```
#!pip3 install "git+https://github.com/belzebuu/landscapes.git"
#import landscapes as ls
from landscapes.single_objective import sphere

# Rosenbrock function
def rosenbrock_function(x):
    return sum(100 * (x[i+1] - x[i]**2)**2 + (1 - x[i])**2 for i in range(len(x) - 1))

#f = rosenbrock_function
f = sphere

sp = ininitialize_population(100, 2, (-5, 5))
x,y = genetic_algorithm(f, sp, k_max=100, m=100)
print(x,y)
```

```
[0.00972184 0.00840947] 0.00016523335954465163
```

The optimal solution of the sphere function is 0 at $[0, 0]$.

Exercise 2 $^+$ (13.1)

Filling a multidimensional space requires exponentially more points as the number of dimensions increases. To help build this intuition, determine the side lengths of an n-dimensional hypercube such that it fills half of the volume of the n-dimensional unit hypercube.

The volume of an hypercube I^n is:

$$V = r^n$$

to find the hypercube of side s that fills half of V we need to compute:

$$r^n/2 = s^n \implies s = r\frac{1}{\sqrt[n]{2}} = r2^{-1/n}$$

As an example for $n = 3$ with $r = 1$, s must be 0.79.

Exercise 3 $^+$ (13.2)

Suppose that you sample randomly inside a unit sphere in n dimensions. Compute the probability that a randomly sampled point is within ϵ distance from the surface of the sphere as $n \to \infty$. Hint: The volume of a sphere is $C(n)r^n$, where r is the radius and $C(n)$ is a function of the dimension n only.

The probability that a randomly sampled point is within ϵ-distance from the surface is the ratio of the volumes. Thus:

$$P(\|\boldsymbol{x}\|_2 > 1 - \epsilon) = 1 - P(\|\boldsymbol{x}\|_2 \leq 1 - \epsilon) = 1 - (1 - \epsilon)^n$$

that tends to 1 as $n \to \infty$.

Exercise 4 $^+$

Generate a full factorial set of design points in Python using `numpy.meshgrid` for two dimensions and plot the generated points. Find a function from the benchmark suite and evaluate the function in those points.

Then, generate a uniform projection plan.

Repeat the same process for a function with $n = 3$ and for $n > 3$ dimensions.

Assuming that we have decided upper $\boldsymbol{a} \in \Re^n$ and lower limits $\boldsymbol{b} \in \Re^n$ for each of the n dimensions and the number of sample points in each direction $\boldsymbol{m} \in \Re^2$: in each dimension:

```
import numpy as np

def samples_full_factorial_mesh(a: np.ndarray, b: np.ndarray, m: np.ndarray):
    ranges = [np.linspace(a[i], b[i], m[i]) for i in range(len(a))]
    M = np.meshgrid(*ranges)
    return M
```

In Python, the `*` operator before a list or tuple (like `*ranges`) is used for unpacking the elements of that iterable into positional arguments for a function call.
The Variable: `ranges` is a list containing multiple NumPy arrays. For example, working in 2D, ranges might look like `[x_array, y_array]`. The Function: `np.meshgrid` expects individual arrays as separate arguments, like `np.meshgrid(x_array, y_array)`. It does not accept a single list containing the arrays (e.g., `np.meshgrid([x_array, y_array])` would be incorrect). The unpacking by using `*ranges`, takes the list and “unpacks” it, passing each element as a separate argument to the function.

```
a=np.array([3,3])
b=np.array([6,6])
m=np.array([10,10])

samples_full_factorial_mesh(a, b, m)
```

```
(array([[3.        , 3.33333333, 3.66666667, 4.        , 4.33333333,
         4.66666667, 5.        , 5.33333333, 5.66666667, 6.        ],
        [3.        , 3.33333333, 3.66666667, 4.        , 4.33333333,
         4.66666667, 5.        , 5.33333333, 5.66666667, 6.        ],
        [3.        , 3.33333333, 3.66666667, 4.        , 4.33333333,
         4.66666667, 5.        , 5.33333333, 5.66666667, 6.        ],
        [3.        , 3.33333333, 3.66666667, 4.        , 4.33333333,
         4.66666667, 5.        , 5.33333333, 5.66666667, 6.        ],
        [3.        , 3.33333333, 3.66666667, 4.        , 4.33333333,
         4.66666667, 5.        , 5.33333333, 5.66666667, 6.        ],
        [3.        , 3.33333333, 3.66666667, 4.        , 4.33333333,
         4.66666667, 5.        , 5.33333333, 5.66666667, 6.        ],
        [3.        , 3.33333333, 3.66666667, 4.        , 4.33333333,
         4.66666667, 5.        , 5.33333333, 5.66666667, 6.        ],
        [3.        , 3.33333333, 3.66666667, 4.        , 4.33333333,
         4.66666667, 5.        , 5.33333333, 5.66666667, 6.        ],
        [3.        , 3.33333333, 3.66666667, 4.        , 4.33333333,
         4.66666667, 5.        , 5.33333333, 5.66666667, 6.        ],
        [3.        , 3.33333333, 3.66666667, 4.        , 4.33333333,
         4.66666667, 5.        , 5.33333333, 5.66666667, 6.        ]]),
 array([[3.        , 3.        , 3.        , 3.        , 3.        ,
         3.        , 3.        , 3.        , 3.        , 3.        ],
        [3.33333333, 3.33333333, 3.33333333, 3.33333333, 3.33333333,
         3.33333333, 3.33333333, 3.33333333, 3.33333333, 3.33333333],
        [3.66666667, 3.66666667, 3.66666667, 3.66666667, 3.66666667,
         3.66666667, 3.66666667, 3.66666667, 3.66666667, 3.66666667],
        [4.        , 4.        , 4.        , 4.        , 4.        ,
         4.        , 4.        , 4.        , 4.        , 4.        ],
        [4.33333333, 4.33333333, 4.33333333, 4.33333333, 4.33333333,
         4.33333333, 4.33333333, 4.33333333, 4.33333333, 4.33333333],
        [4.66666667, 4.66666667, 4.66666667, 4.66666667, 4.66666667,
         4.66666667, 4.66666667, 4.66666667, 4.66666667, 4.66666667],
        [5.        , 5.        , 5.        , 5.        , 5.        ,
         5.        , 5.        , 5.        , 5.        , 5.        ],
        [5.33333333, 5.33333333, 5.33333333, 5.33333333, 5.33333333,
         5.33333333, 5.33333333, 5.33333333, 5.33333333, 5.33333333],
        [5.66666667, 5.66666667, 5.66666667, 5.66666667, 5.66666667,
         5.66666667, 5.66666667, 5.66666667, 5.66666667, 5.66666667],
        [6.        , 6.        , 6.        , 6.        , 6.        ,
         6.        , 6.        , 6.        , 6.        , 6.        ]]))
```

Alternatively, we can achieve the same with the Cartesian `product` from the `itertools` module:

```python
import itertools

def samples_full_factorial_prod(a: np.ndarray, b: np.ndarray, m: np.ndarray):
    ranges = [np.linspace(a[i], b[i], m[i]) for i in range(len(a))]
    F = itertools.product(*ranges)
    return F
```

We can also construct a uniform projection plan as follows:

```python
def uniform_projection_plan(m, n):
    perms = [np.random.permutation(m) for i in range(n)]
    L = np.array([[perms[i][j] for i in range(n)] for j in range(m)])
    return L
```

```python
import landscapes as ls

a = [0, 0]
b = [1, 1]
m = [5, 5]
F = samples_full_factorial_prod(a, b, m)
F2 = np.array(list(F))
print(F2)
```

```
[[0.   0.  ]
 [0.   0.25]
 [0.   0.5 ]
 [0.   0.75]
 [0.   1.  ]
 [0.25 0.  ]
 [0.25 0.25]
 [0.25 0.5 ]
 [0.25 0.75]
 [0.25 1.  ]
 [0.5  0.  ]
 [0.5  0.25]
 [0.5  0.5 ]
 [0.5  0.75]
 [0.5  1.  ]
 [0.75 0.  ]
 [0.75 0.25]
 [0.75 0.5 ]
 [0.75 0.75]
 [0.75 1.  ]
 [1.   0.  ]
 [1.   0.25]
 [1.   0.5 ]
 [1.   0.75]
 [1.   1.  ]]
```

`F` is a tuple of two matrices X and Y whose n^2 entries indicate the x and the y coordinates, respectively, of the sample points. `F2` is an array of two matrices, ie, a 3D array.

Exercise 5 *

Construct in Python a Latin Hypercube design plan with numbers 1 through $m = 4$ for 3 dimensions. How many different Latin Hypercubes are there? How many different uniform projection plans are there for a single Latin Hypercube? How many different uniform projection plans can be generated from all Latin Hypercubes of m numbers in 3 dimensions?

There are $(4!)^3$ Latin Hypercubes of dimension 3. For each Latin Hypercube there are 4 uniform project plans one for every digit 1,2,3,4. from all Latin Hypercubes of m numbers in 3 dimensions $4(4!)^3$ different uniform projection plans can be generated.

Exercise 6

Consult the documentation of the quasi-Monte Carlo python submodule `scipy.stats.qmc` and repeat the previous exercises using those functions.

Exercise 7 *

Quasi-random sequences are typically constructed for the unit n-dimensional hypercube, $[0, 1]^n$. Any multidimensional function with bounds on each variable can be transformed into such a hypercube. Show how.

Let $\boldsymbol{x} \in [a_1, b_1] \times [a_2, b_2] \times ... \times [a_n, b_n]$. We want to tranform to and from variables $x' \in [0, 1]^n$, ie, belonging to the hypercube $I^n = [0, 1]^n$.
For every dimension $i = 1, ..., n$:

$$x'_i = x_i \cdot (b_i - a_i) + a_i \qquad\qquad x_i = \frac{x'_i - a_i}{b_i - a_i}$$

This is also called *normalization* (which is not the same as *standardization.*
In Python terms, we can transform a sampling plan generated in the hypercube I^n to a sampling plan for the hyperrectangle $[a_1, b_1] \times [a_2, b_2] \times ... \times [a_n, b_n]$ as follows:

```
import numpy as np

def rev_unit_hypercube_parameterization(x, a, b):
    return x*(b-a)+a
```

Alternatively, we can stay in the hypercube and transform the function to a function in the hypercube as follows:

```
def reparameterize_to_unit_hypercube(f, a, b):
  delta = b - a
  return lambda x: f(x * delta + a)
```

Exercise 8 * (13.3)

Suppose we have a sampling plan $X = \{\boldsymbol{x}_1, ..., \boldsymbol{x}_{10}\}$, where

$$x_i = [\cos(2\pi i/10), \sin(2\pi i/10)]$$

Compute the Morris-Mitchell criterion for X using an L_2 norm when the parameter q is set to 2. In other words, evaluate $\Phi_2(X)$. If we add $[2, 3]$ to each $\boldsymbol{x}_i$, will $\Phi_2(X)$ change? Why or why not?

Compare the MM criterion for the points given above with a set of 10 uniformly sampled random points.

```python
import numpy as np

def pairwise_distances(X: np.ndarray, p: int = 2) -> np.ndarray:
    m = X.shape[0]
    dists = []
    for i in range(m - 1):
        for j in range(i + 1, m):
            d = np.linalg.norm(X[i] - X[j], ord=p)
            dists.append(d)
    return np.array(dists)

def phiq(X: np.ndarray, q: float = 1.0, p: int = 2) -> float:
    dists = pairwise_distances(X, p)
    return np.sum(dists ** (-q)) ** (1.0 / q)

# Create 10 points evenly spaced on the unit circle
X = np.array([
    [np.cos(2 * np.pi * i / 10), np.sin(2 * np.pi * i / 10)]
    for i in range(10)
])

print("phiq(X, 2):", phiq(X, q=2))


np.random.seed(42)
random_points = np.random.rand(10, 2)

print("phiq(X, 2):", phiq(random_points, q=2))
```

```
phiq(X, 2): 6.422616289332565
phiq(X, 2): 31.659636916479467
```

```
phiq(X, 2): 6.422616289332565
phiq(X, 2): 31.659636916479467
```

No. The Morris-Mitchell criterion is based entirely on pairwise distances. Shifting all of the points by the same amount does not change the pairwise distances and thus will not change $\Phi_2(X)$.

Exercise 9 (13.4)

Additive recurrence requires that the multiplicative factor c be irrational. Why can c not be rational?

A rational number can be written as a fraction of two integers a/b. It follows that the sequence repeats every b iterations:

$$\begin{aligned} x_{k+1} &= x_k + \tfrac{a}{b} (\bmod 1) \\ x_k &= x_0 + k\tfrac{a}{b} (\bmod 1) \end{aligned}$$

Exercise 10 *

Consider the set X of random points given by:

```
    np.random.seed(42)
    X = np.random.rand(6, 2)
```

Plot them and:

- calculate the discrepancy of the sample plan.
- find the best space-filling subset of size 3.

Exercise 11 *

Consider the sets X and Y of random points given by:

```
    np.random.seed(42)
    X = np.random.rand(10, 2)
    Y = np.random.rand(10, 2)
```

Plot them and decide which is the most space-filling on the basis of pairwise distances.

Exercise 12

Calculate an approximation of the value of π using the Monte Carlo method. Draw a circle inscribed in a unit square and use a sampling plan to create sample points in the square. Use the ratio between the number of points inside the circle and the total number of points to approximate π.

Compare the convergence rate of uniform random sampling against a quasi-Monte Carlo low discrepancy method like Sobol method.

Plot the sample plans generated.

The unit square is $[0, 1] \times [0, 1]$. The equation of a circle of radius 1 inscribed in the square is

$$x^2 + y^2 = 1$$

Points internal to the circle satisfy $x^2 + y^2 \leq 1$. We denote the area under the curve by H.
The area of the square is 1 and the one of the circle is $\pi r^2 = \pi$. The area inscribed in the square is $\pi/4$.

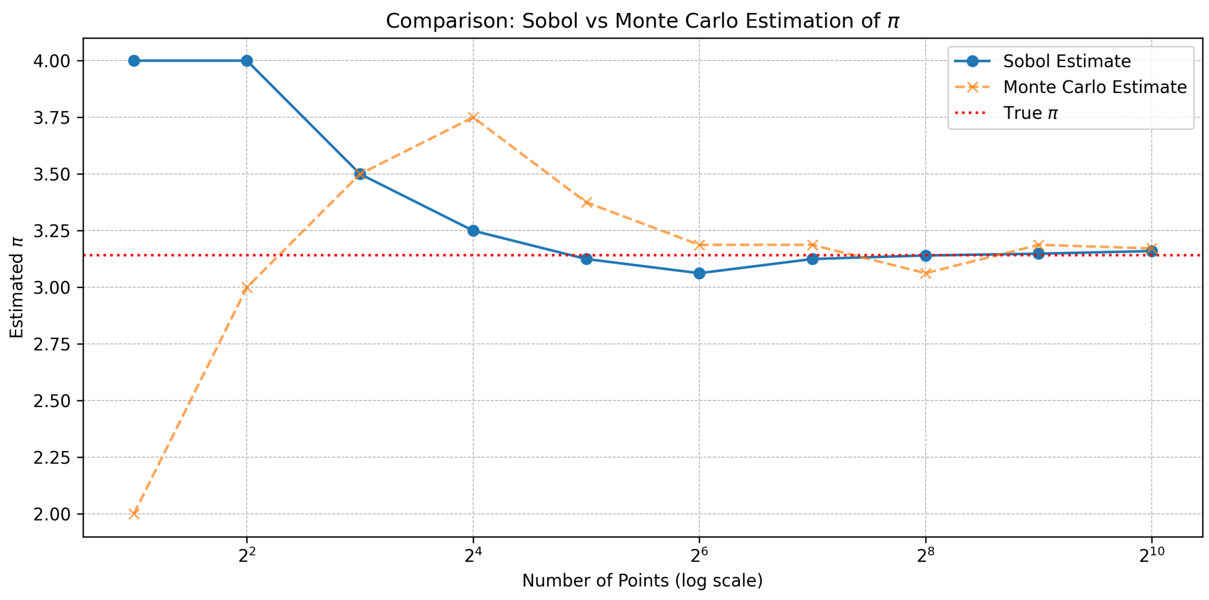
$$\frac{\#H}{\#X} = r = \frac{\pi}{4}$$

Hence,

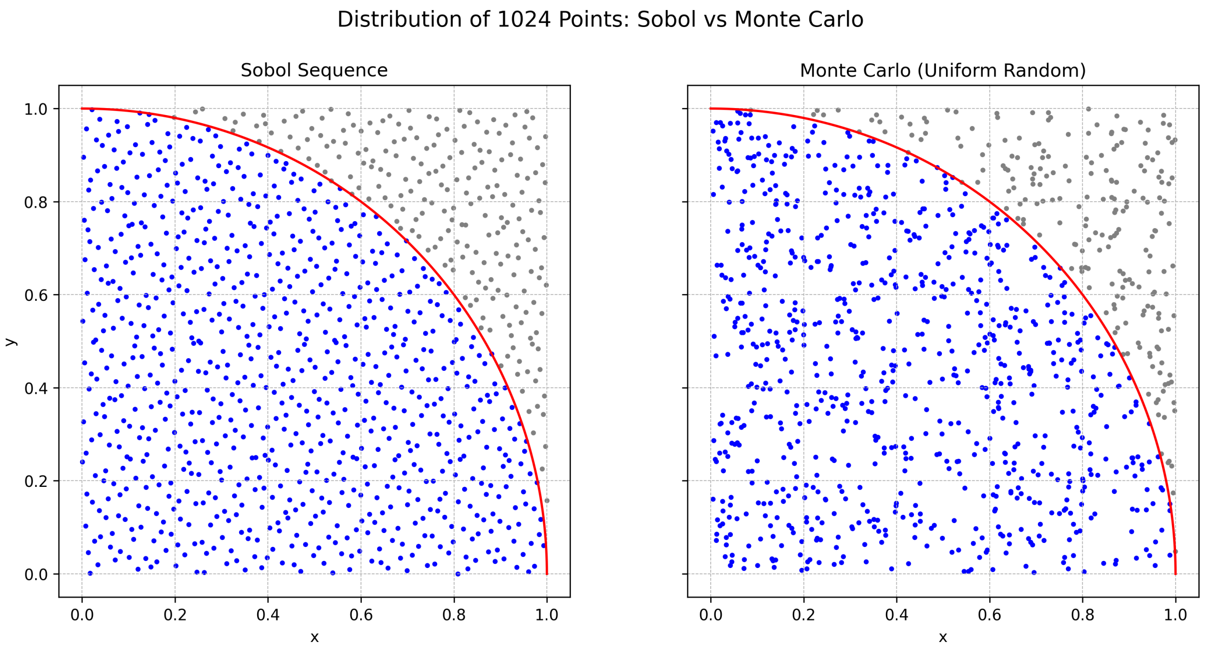
$$\pi = 4r$$

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import qmc

# Settings
max_power = 10  # up to 2^10 = 1024 points

# --- Sobol sequence ---
sobol_sampler = qmc.Sobol(d=2, scramble=True)
sobol_points = sobol_sampler.random_base2(m=max_power)
#sobol_points = 2 * sobol_points - 1
sobol_radii = np.sum(sobol_points**2, axis=1)

# --- Monte Carlo (uniform random) ---
np.random.seed(42)
random_points = np.random.rand(2**max_power, 2)
random_radii = np.sum(random_points**2, axis=1)

# Store estimates
ns = []
pi_sobol = []
pi_random = []

# Compute estimates for each power of 2
for m in range(1, max_power + 1):
    n = 2**m
    ns.append(n)

    inside_sobol = np.sum(sobol_radii[:n] <= 1)
    inside_random = np.sum(random_radii[:n] <= 1)

    pi_sobol.append(4 * inside_sobol / n)
    pi_random.append(4 * inside_random / n)
```

```python
# --- Plot ---
plt.figure(figsize=(10, 5))
plt.plot(ns, pi_sobol, marker='o', label='Sobol Estimate')
plt.plot(ns, pi_random, marker='x', linestyle='--', label='Monte Carlo Estimate', \
    alpha=0.7)
plt.axhline(np.pi, color='red', linestyle=':', label=r'True $\pi$')
plt.xscale('log', base=2)
plt.xlabel("Number of Points (log scale)")
plt.ylabel(r"Estimated $\pi$")
plt.title(r"Comparison: Sobol vs Monte Carlo Estimation of $\pi$")
plt.grid(True, linestyle='--', linewidth=0.5)
plt.legend()
plt.tight_layout()
plt.show()
#plt.savefig("pi_convergence.png")
```


Comparison: Sobol vs Monte Carlo Estimation of π
Sobol Estimate
Monte Carlo Estimate
True π
Estimated π
Number of Points (log scale)
4.00
3.75
3.50
3.25
3.00
2.75
2.50
2.25
2.00
2^2
2^4
2^6
2^8
2^{10}

```
# Plotting
fig, axs = plt.subplots(1, 2, figsize=(12, 6), sharex=True, sharey=True)

# Circle coordinates
theta = np.linspace(0, np.pi/2, 500)
circle_x = np.cos(theta)
circle_y = np.sin(theta)

# Sobol sequence
# axs[0].scatter(sobol_points[:, 0], sobol_points[:, 1], s=5, color='blue')
axs[0].scatter(sobol_points[sobol_radii > 1, 0], sobol_points[sobol_radii > 1, 1],
                                        s=5, color='gray', label='Outside Circle')
axs[0].scatter(sobol_points[sobol_radii <= 1, 0], sobol_points[sobol_radii <= 1, 1],
                                        s=5, color='blue', label='Inside Circle')
axs[0].plot(circle_x, circle_y, color='red', linewidth=1.5, label='Unit Circle')
axs[0].set_title("Sobol Sequence")
axs[0].set_xlabel("x")
axs[0].set_ylabel("y")
axs[0].axis("square")
axs[0].grid(True, linestyle='--', linewidth=0.5)

# Random sampling
#axs[1].scatter(random_points[:, 0], random_points[:, 1], s=5, color='orange')
axs[1].scatter(random_points[random_radii > 1, 0], random_points[random_radii > 1, 1],
                                        s=5, color='gray', label='Outside Circle')
axs[1].scatter(random_points[random_radii <= 1, 0], random_points[random_radii <= 1, 1],
                                        s=5, color='blue', label='Inside Circle')
axs[1].plot(circle_x, circle_y, color='red', linewidth=1.5, label='Unit Circle')
axs[1].set_title("Monte Carlo (Uniform Random)")
axs[1].set_xlabel("x")
axs[1].axis("square")
axs[1].grid(True, linestyle='--', linewidth=0.5)

plt.suptitle(f"Distribution of {n} Points: Sobol vs Monte Carlo", fontsize=14)
plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()
#plt.savefig("pi_points.png")
```



Distribution of 1024 Points: Sobol vs Monte Carlo
Sobol Sequence
Monte Carlo (Uniform Random)
y
x
x
0.0
0.2
0.4
0.6
0.8
1.0