Supporting textbook chapters for week 10: Chapters 10.1 and 10.2

Week 10 topics:

- Random number generation
- Monte Carlo integration

Intro to Random Numbers

Why we need random numbers:

- For randomly sampling a domain (today)
- Monte Carlo integration (today)
- Monte Carlo simulations (next week): including physical processes like diffusion, radioactive decay, Brownian motion
- Stochastic algorithms (we'll see some next week)
- Cryptography

What is a useful random sequence of numbers?

- Follows some desired distribution
- Unpredictable on a number-by-number basis
- Fast to generate (we may need billions of them)
- Long period (we may need billions of them)
- Uncorrelated

Problems with *actually* random numbers:

- generally slow, expensive to generate,
- hard/impossible to reproduce for debugging
- Often hard to characterize underlying distribution

Q: How can a computer generate random numbers?

A: It can't, assuming it's a classical (not quantum) computer!

The classical computer can't do anything randomly. So there are 2 options:

- find physical process (e.g. quantum) that actually is random, have computer store info from that to provide a random number
- Use an algorithm for generating a sequence of numbers that approximates the properties of random numbers. This is called a "Pseudorandom Number Generator" (PRNG) or a "Deterministic Random Bit Generator" (DRBG).

Common Tests for Randomness

- Making sure numbers aren't correlated,
- Making sure higher-order moments of distributions have desired properties
- Other tests...

Correlations

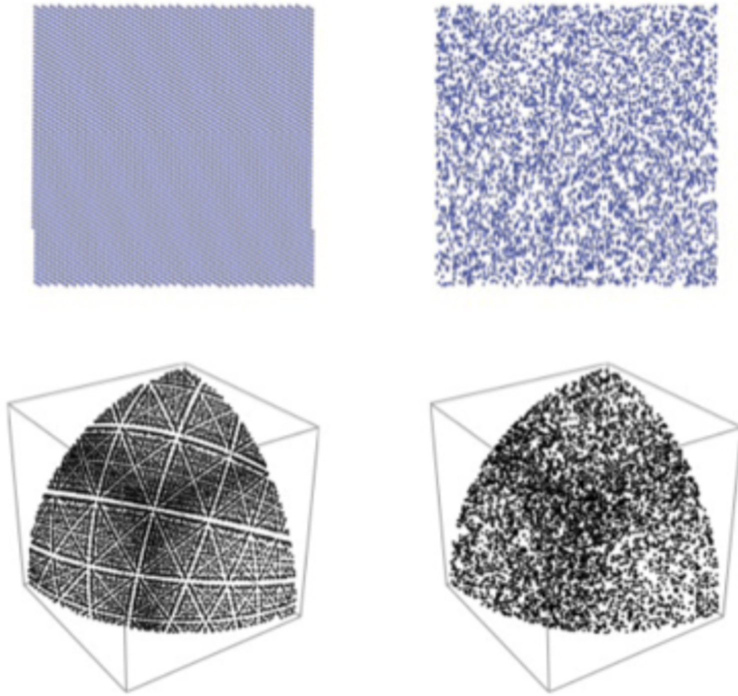
Simple pairwise correlations:

$$\epsilon(N,n) = \frac{1}{N}\sum_{i=1}^{N} x_i x_{i+n} - \mathrm{E}[x^2]$$

- $N =$ number of data points
- $n =$ correlation "distance"
- $\mathrm{E}[X]$, the expected value of X.

We want to avoid correlations between pairs of numbers.

Left: bad PRNG. right: Mersenne Twister. From Katzgrabber, "Random Numbers in Scientific Computing: an Introduction" (arXiv: 1005.4117)



Moments

k^{th} moment of $\mu(N,k)$ (sequence of N elements) : $\mu(N,k) = \mathrm{E}[x^k]$

We want to ensure moments of random number distributions also have desired properties.

Other tests

- Overlapping permutations. e.g. analyze orders of 5 consecutive random numbers. There are $5!(= 5 \times 4 \times \ldots)$ possible permutations. They should occur with equal probability.
- ...

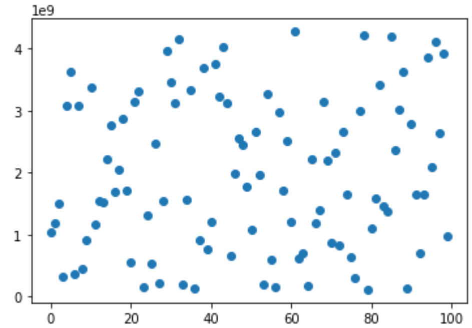
Linear Congruential Generator

- Sequences produced by a PRNG seem random, but are reproducible if you start with same "seed"
- e.g. (actually a bad choice for a PRNG, but good for illustration): LC-RNG
  - $x_{i+1} = (ax_i + c) \mod m.$
  - In Python, produce it with: `x[i+1] = (a*x[i] + c) % m`
  - x_0 is the seed, m is a large integer which determines the period. For good results:
    - $a-1$ is a multiple of p for every prime divisor p of m (e.g., $a-1$ is multiple of 2 and 3 if m is multiple of 2 and 3),
    - c relatively prime to m.
- How does computer pick seed x_0? Taking system time is common (dangerous in parallel because all processors could use the same time-seed, though)

In [10]:
```python
# Newman's lcg.py
from pylab import plot, show

N = 100
a = 1664525
c = 1013904223
m = 4294967296
x = 11
results = []

for i in range(N):
    x = (a*x+c) % m
    results.append(x)
plot(results, "o")
show()
```



Benefits:

- good for testing code, since you can supply the same 'seed' (for reproducible outcome). e.g. the following code will always produce the same `x` (that is, 0.03738057695923325).

  `random.seed(4219)`

  `x = random.random()`
  - *This is actually true for most PRNGs, not just this linear congruential*
  - *The basic default behaviour of PRNGs is to rescale results over $[0, 1)$, hence the non-integer value for `x` above.*
- easy to generate many different sequences, just pick many different seeds.

Randoms in python

Better methods?

- We want to avoid correlations between pairs of numbers
- Can do lots of test to show if PRNGs producing right "statistics" of random numbers!
- Python uses a Mersenne twister

Functions in `random.py` most likely to use (assuming `import random`):

- `random()` : gives a random float uniformly distributed in a the range $[0, 1)$ (all values have equal probability of being selected),
- `randrange(m, n)` : Gives a random integer from `m` to `n-1` , inclusive.
- If you need a uniformly distributed random float outside the range $[0, 1)$, say in range $[a, b)$, then just multiply your answer by $(b - a)$ and shift the argument. For example:

  `num = random()`

  `shiftnum = (b-a)*num + a`

More resources (**you may find useful for lab!**):

[https://numpy.org/doc/stable/reference/random/index.html (https://numpy.org/doc/stable/reference/random/index.html)](https://numpy.org/doc/stable/reference/random/index.html)

[https://docs.python.org/3/library/random.html (https://docs.python.org/3/library/random.html)](https://docs.python.org/3/library/random.html)

Non-Uniform distributions

What if you need a random number from a non-uniform distribution?

- Get a uniformly distributed random number, then use a transformation to make it seem like it comes from a non-uniform distribution.
- Consider source of random floats z from a distribution with probability density function $q(z)$, i.e., the probability of generating a number in the interval z to $z + \mathrm{d}z$ is:

$$q(z)\mathrm{d}z$$

- For a uniform distribution over $[0, 1)$, $q(z) = 1$ because for all $\mathrm{d}z$, equal probability of number being chosen.

- Now consider transformation of z into new variable, say x, using:
$$x = x(z)$$
- Then x is also a random number but will have some other probability distribution, call it $p(x)$.
- The probability of generating a value of x between x and $x + \mathrm{d}x$ is by definition equal to the probability of generating a value of z between the corresponding z and $z + \mathrm{d}z$:
$$p(x)\mathrm{d}x = q(z)\mathrm{d}z, \quad \text{where } x = x(z)$$

- Goal: find a function $x(z)$ so that x has the distribution we want.
- Then we can use `random()` to get a uniformly distributed random number z and transform it to x using:
$$q(z) = 1 \quad \text{over} \quad [0, 1)$$
$$q(z)\mathrm{d}z = p(x)\mathrm{d}x$$
$$\Rightarrow \int_0^z 1\mathrm{d}z' = z = \int_0^{x(z)} p(x')\mathrm{d}x'.$$
- Plug in your $p(x)$ for the probability distribution you need and integrate to find $z(x)$ (if you can!)
- Even then: might not be possible to solve for $x(z)$.

Example: exponential distribution
$$q(z) = 1 \quad \text{over} \quad [0, 1)$$
$$p(x) = a\exp(-ax) \quad \text{over} \quad [0, \infty)$$
$$\Rightarrow z = \int_0^{x(z)} a\exp(-ax')\mathrm{d}x' = 1 - \exp(-ax)$$
$$\Rightarrow x = -\frac{\ln(1-z)}{a}.$$

- Draw a number z in $[0, 1)$,
- $x(z)$ has the desired distribution.

Intro to "Monte Carlo Integration"

- Sounds great in theory. Would never work in practice without computers.
- 3 Monte Carlo techniques you will use in the lab:
  - "hit or miss" or "standard" Monte Carlo
  - "mean value" Monte Carlo
  - "importance sampling" Monte Carlo

You've already learned a bunch of different methods for integrating, why introduce another one? (Especially since its convergence/error properties are worse than the other methods):

Reason 1: Good for pathological functions

Or just fast-varying functions.

Reason 2: MUCH faster for multi-dimensional integrals.

The "curse of dimensionality":

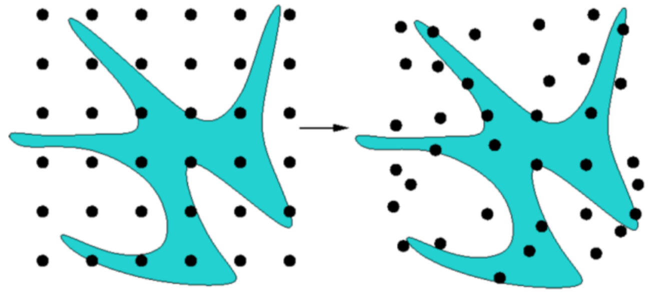
- For a dimension-d integral, you need $O(n^d)$ grid points.
  - E.g. with trapezoid, Simpson or Gaussian integration: for $n = 10$ points along one axis, a 10-d integral need 10^{10} grid points!
- Alternative way to look at it: if you can afford N points, your grid has side length $O(N^{1/d})$.
  - For trapezoid integration, error $\epsilon = O(h^2) \propto 1/N^{2/d}$.
  - E.g., for a 10-d integral, $\epsilon \propto 1/N^{1/5}$.
  - **Monte Carlo: $\boxed{\epsilon \propto 1/N^{1/2}}$, regardless of d.**

Reason 3: much easier to implement in complicated domains

i.e., complicated boundaries of integration.

Implementation of Monte Carlo integration

Use random numbers to pick points at which to evaluate integrand.



- Simple and flexible.
- Can "tune" it to focus on important parts.

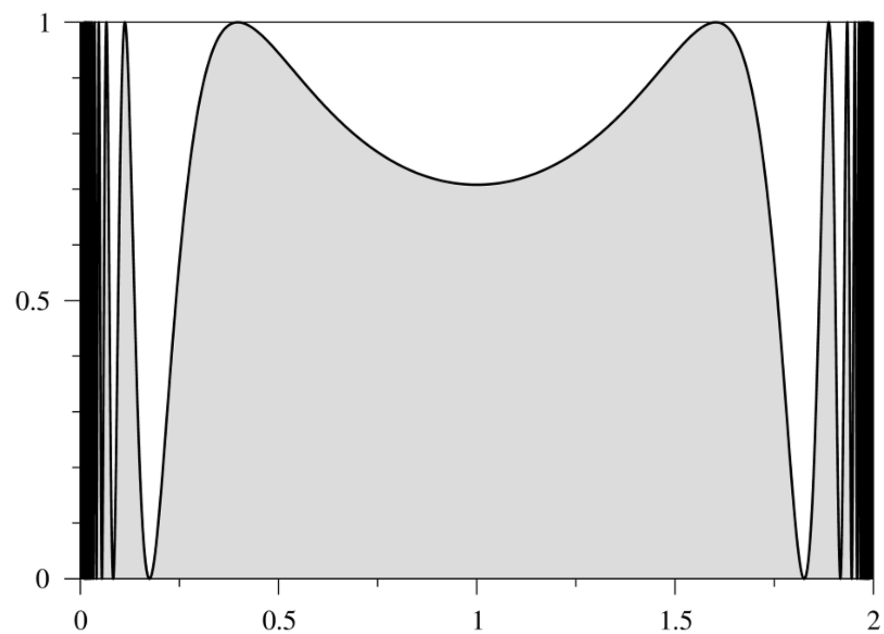
Hit-or-miss MC

- If your function "fits" in a finite region where we want to integrate from $x = 0$ to $x = 2$:

$$f(x) = \sin^2\left[\frac{1}{(2-x)x}\right]$$

- function fits in box of height 1, width 2.
- Define area of box: A ($= 2$ here; *this is important! It is the piece of info we will leverage*).
- Integral of function is shaded area in the box (call it I).

Newman, fig 10-4



- Probability that your random point falls in the shaded region is $p = I/A$.
- Algorithm:
  1. Randomly pick N locations (x, y) in the box (lots of them).
  2. Count the number of locations that are in the shaded region (call the count k).
  3. The fraction of points in the shaded region is k/N. This approximates the probability p. Solve for I:

$$P = \frac{I}{A} \approx \frac{k}{N} \Rightarrow I \approx \frac{kA}{N}.$$

Can estimate the error on the integral (text gives derivation on page 467 from probability theory):

- The *Expected Error* (standard deviation):

$$\sigma = \sqrt{\frac{(A-I)I}{N}}.$$

- Notice it varies as $N^{-1/2}$. This is **very slow**!
- Compare:
  - Trapezoid Rule: error varies as N^{-2},
  - Simpson's Rule: error varies as N^{-4}.
- This is why you only use Monte Carlo integration if you absolutely have to.

Example: exercise 10.5(a) from the text.

Write a program to evaluate

$$I = \int_0^2 \sin^2\left[\frac{1}{(2-x)x}\right]\mathrm{d}x$$

using the "hit-or-miss" method.

- Use $N = 10^4$ points.
- Also evaluate the error on your method.

In [5]:
```
import numpy as np

def f(x):
    return np.sin(1/((2-x)*x))**2  # the function to integrate

# define parameters
N = 10000
k = 0
a = 0.
b = 2.

# loop over samples; in loop, check wether point is above|below curve
for i in range(N):
    x_sampl = a + (b-a)*np.random.random()
    y_sampl = np.random.random()
    if y_sampl <= f(x_sampl):
        k += 1
# compute fraction of points below and integral.
A = (b-a)*1.
Int = A*k/N
print("I = {0:.6e}".format(Int))

# compute error
sigma_HM = np.sqrt(Int*(A-Int)/N)  # HM stands for hit-or-miss
print('error for hit-or-miss = {0:.6e}'.format(sigma_HM))
```

```
I = 1.441800e+00
error for hit-or-miss = 8.971136e-03
```

Mean value MC

- Use the definition of an average (or mean value):

$$I = \int_a^b f(x)\mathrm{d}x,$$
$$\langle f\rangle = \frac{1}{b-a}\int_a^b f(x)\mathrm{d}x = \frac{I}{b-a}$$
$$\Rightarrow\ I = (b-a)\langle f\rangle$$

- Use random numbers for x to estimate $\langle f\rangle$. Evaluate f at N random x's, then calculate:

$$\langle f\rangle \approx \frac{1}{N}\sum_{i=1}^{N} f(x_i)\ \Rightarrow\ I \approx \frac{b-a}{N}\sum_{i=1}^{N} f(x_i).$$

- Different from "hit-or-miss": back then we chose N random points over (x, y) instead of just x here.

Error estimate.

- Can estimate the error on the integral (text gives derivation on pages 468-469 from probability theory): "Expected Error":

$$\sigma = (b-a)\sqrt{\frac{\operatorname{var} f}{N}}$$
$$\operatorname{var} f = \left\langle f^2\right\rangle - \langle f\rangle^2.$$

- Notice it also varies as $N^{-1/2}$. However, it turns out the leading constant is smaller than with the hit or miss method. (We won't go into the mathematical details of why.)

Example: exercise 10.5(b) from the text.

Write a program to evaluate

$$I = \int_0^2 \sin^2\left[\frac{1}{(2-x)x}\right]\mathrm{d}x$$

using the mean value method.

- Use $N = 10^4$ points.
- Also evaluate the error on your method.

In [6]:
```python
#import numpy as np

#def f(x):
#    return np.sin(1/((2-a)*x))**2

#N = 10000
#a = 0.
#b = 2.

k = 0  # will contain the average
k2 = 0  # will be used for variance

for i in range(N):
    x = (b-a)*np.random.random()
    k += f(x)
    k2 += f(x)**2

I = k * (b-a) / N
print(I)

# error
var = k2/N - (k/N)**2  # variance <f**2> - <f>**2
sigma_MV = (b-a)*np.sqrt(var/N)
print('error = ', sigma_MV)
print('recall error in hit-or-miss = ', sigma_HM)
```

```
1.447445892696602
error =  0.005331635473119352
recall error in hit-or-miss =  0.008971135714055384
```

Importance sampling MC

- Good to use when your integrand contains a divergence
- Want to place more points in region where the integrand is large to better estimate the integral
- When you want to integrate out to infinity, give less weight to points in densely-populated regions to not bias final result
- Illustrative example (obviously a bad one for Monte-Carlo, but good for making my point):

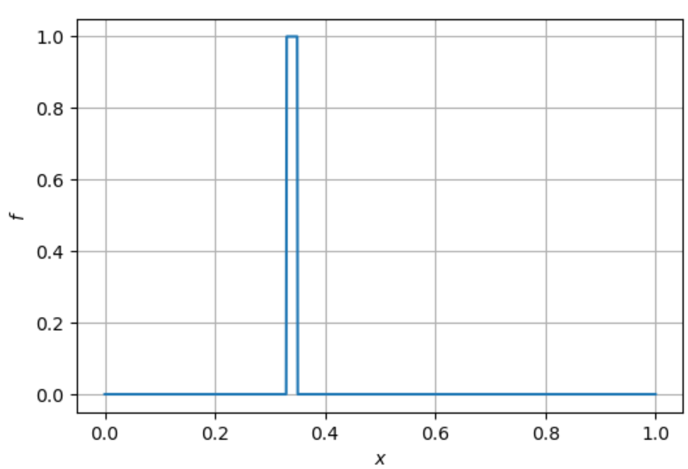
$$f(x) = 1 \quad \text{for} \quad c < x < d, \qquad f(x) = 0 \quad \text{otherwise.}$$

In [4]:
```python
import matplotlib.pyplot as plt
x = np.linspace(0, 1, 1000)
f = 0.*x

c = 0.33
d = 0.35
for i, xs in enumerate(x):
    if c < xs < d:
        f[i] = 1.

plt.figure(dpi=100)
plt.plot(x, f)
plt.grid()
plt.xlabel('$x$')
plt.ylabel('$f$')
```

Out[4]: `Text(0, 0.5, '$f$')`



- Easy to miss the region between c and d with uniformly sampled points
- evaluating the integral many times using Mean Value or Hit/Miss MC (with different randomly sampled points) can give very different answers, much larger than the expected error
- Solution: sample "important" regions more frequently. I.e., come up with a non-uniformly distributed set of random numbers. This is called "Importance Sampling".

- Text (p. 473) shows that using a weight function $w(x)$, you can always write:

$$I = \int_a^b f(x)\mathrm{d}x = \underbrace{\left\langle \frac{f(x)}{w(x)} \right\rangle_w}_{\substack{weighted\\ average}} \int_a^b w(x)\mathrm{d}x.$$

- Weighted average: $\langle X \rangle_w =$ average of X over set of points that sample "heavily-weighted" region more frequently, following w.
  - But then, $\langle f/w \rangle_w$ means that the more we sample a region, the less weight points in that region have in the final average.
- Goal: find a weight function that gets rid of pathologies in integrand $f(x)$. E.g., if $f(x)$ has a divergence, factor the divergence out and hence get a sum (in the $\langle\rangle$) that is well behaved (i.e. doesn't vary much each time you do the integral).

Example:

$$I = \int_0^1 \frac{x^{-1/2}}{1+\exp(x)}\mathrm{d}x,$$

diverges as $x \to 0$ because of numerator.

- Fine, let $w(x) = numerator$. Then

$$\left\langle \frac{f(x)}{w(x)} \right\rangle_w = \frac{1}{N}\sum_{i=1}^{N} \frac{f(x_i)}{w(x_i)} = \frac{1}{N}\sum_{i=1}^{N} \frac{1}{1+\exp(x_i)},$$

  which is much better behaved than

$$\langle f(x) \rangle = \frac{1}{N}\sum_{i=1}^{N} \frac{x^{-1/2}}{1+\exp(x_i)}.$$

- $\langle\rangle_w$ isn't $\langle\rangle$: it is a *weighted* average, numbers aren't drawn uniformly in $[0,1)$. The *weights* define how often you draw a sample
- In practice: when you've chosen your weight function, you then need to make sure to randomly sample points from the non-uniform distribution:

$$p(x) = \frac{w(x)}{\int_a^b w(x)\mathrm{d}x}$$

  Use the transformation method described earlier in this lecture to take a uniformly distribution random z and find the corresponding x for this distribution.
- "Expected error":

$$\sigma = \sqrt{\frac{\mathrm{var}(f/w)}{N}} \int_a^b w(x)\mathrm{d}x.$$

  Yes, it also varies as $N^{-1/2}$. If you do the integral many times, your values should mostly fall within the expected error.

Summary

Pseudo-random number generators

- Computers can't generate purely random numbers, but we can fool ourselves in creating algorithms that mimic random processes
- Need statistical tests for randomness of PRNGs (correlations, moments, others...)
- Linear congruential random number generator: OK-ish, need long period, some tricks
- Python and most libraries use a Mersenne twister, which is better.
- Need a non-uniform distribution? (i.e., higher probability to draw a number around or above certain value?) Use a mathematical transform to map a uniform distribution in $[0,1)$ onto desired distribution.

Monte Carlo integration

- General idea: shoot randomly at a domain (sometimes non-uniformly), tally results.
- Convergence in $N^{-1/2}$, with N the total number of evaluations, is the general rule.
- Good for higher-dimensional integrals, complicated geometries, pathological functions (in other words: when you are desperate).
- Hit-or-miss MC: shoot randomly in dimension-$(d+1)$ domain, count when you hit and when you miss, gives integral.
- Mean-value MC: shoot randomly in dimension-d domain (x if 1D) evaluate $f(x_1,\dots)$, integral is mean value divided by domain size.
- Importance sampling: in case of pathology (divergence, etc), factor it out, perform *weighted average*, and use properties of non-uniform distributions to draw numbers and assign weight.