The African School of Fundamental Physics and Applications

# Integrating Scientific Computing into Math and Science Classes


Dave Biersach
Brookhaven National Laboratory
dbiersach@bnl.gov



**Session 02**
Calculus and
Monte Carlo Methods

# Session **02** – Topics

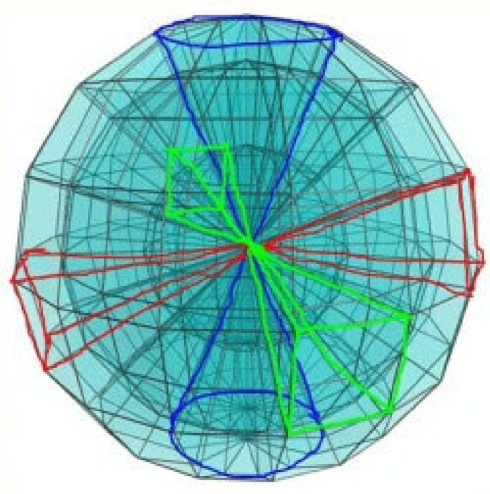
- Understand the principles behind the **Monte Carlo** method
- Compare the accuracy when estimating the area of a 2-D **unit** circle using two different sampling methods
  - Fixed sampling across a **uniform grid** of points
  - Variable sampling using a set of **random** points
- Appreciate the impact of minimizing **discrepancies** when using Monte Carlo estimation techniques
  - Pseudo-random number generator (**PRNG**) – **Permuted Congruential Generator**
  - Quasi-random number generator (**QRNG**) – **Halton Sequence**

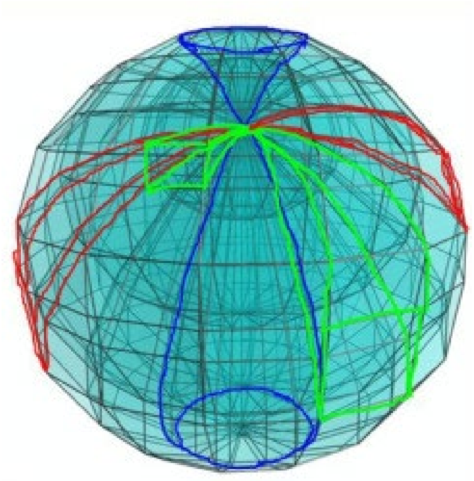
# Session **02** – Topics

- Use the Monte Carlo method to estimate the volume of a three-dimensional unit **sphere** using a QRNG
- Use the Monte Carlo method to estimate the content of an **n-ball** in dimensions from **1 to 12**
- Use the Monte Carlo method to estimate the probability that a *random variable* selected from a Gaussian **Standard Normal** distribution will fall within one standard deviation away from its **mean**

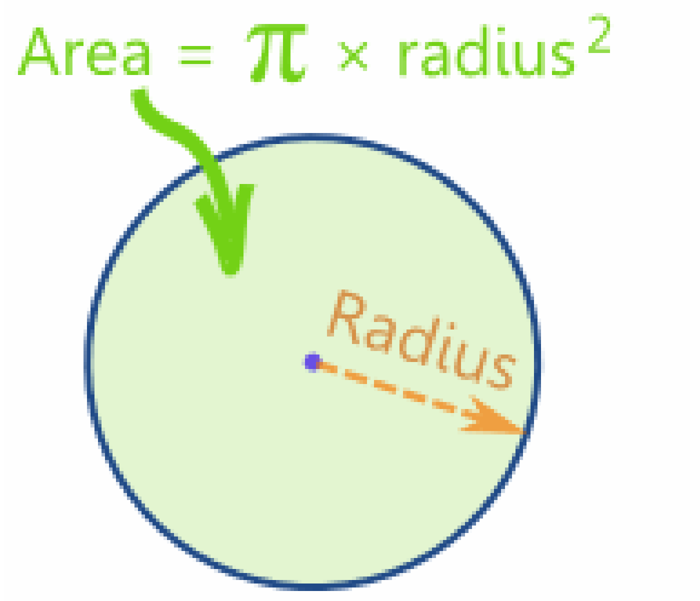
# An *Interesting* Question

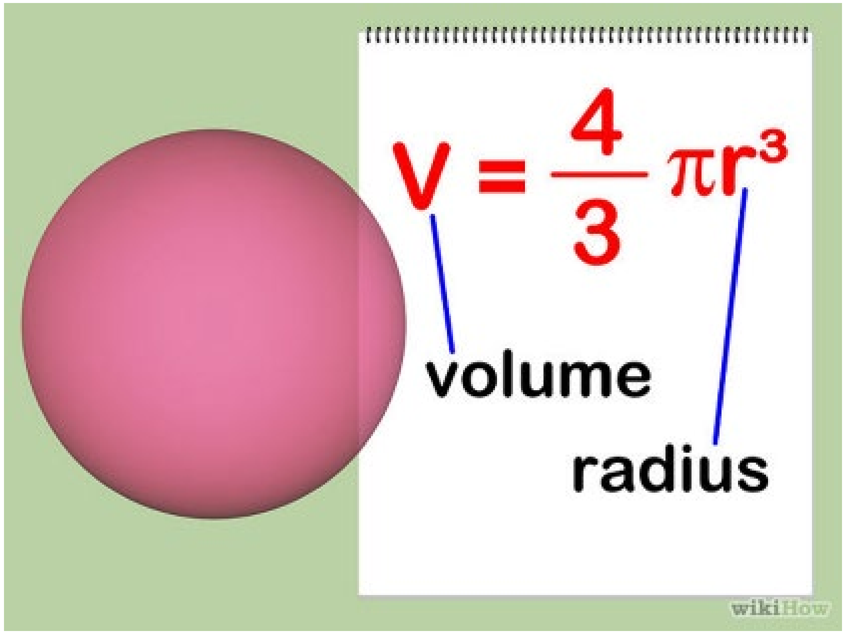
- What is the ***volume*** of a **four-dimensional** **unit** hypersphere?
  - What does a 4D sphere "look" like?
  - What is a "unit" sphere?
  - Where do I even start?
- Break down complex questions into simpler steps:
  - How can we calculate the area of a 2D circle?
  - How can we calculate the volume of a 3D sphere?
  - How do we move from 3D to 4D?



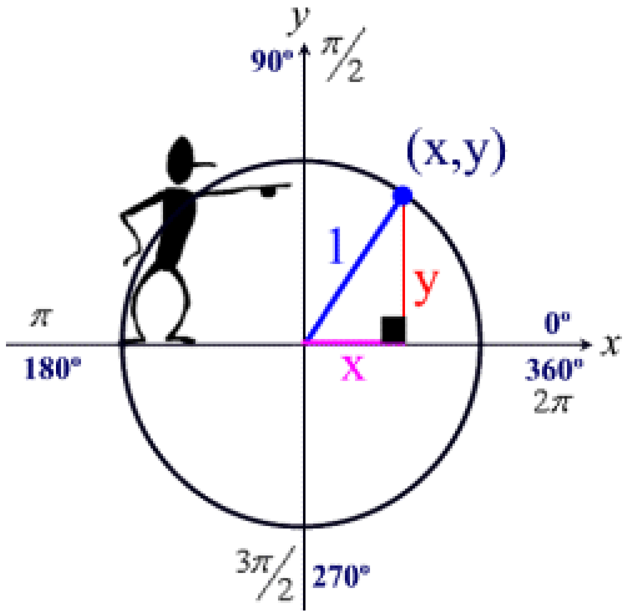


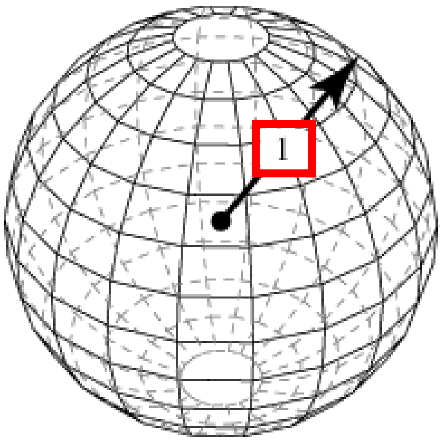
# Area and Volume



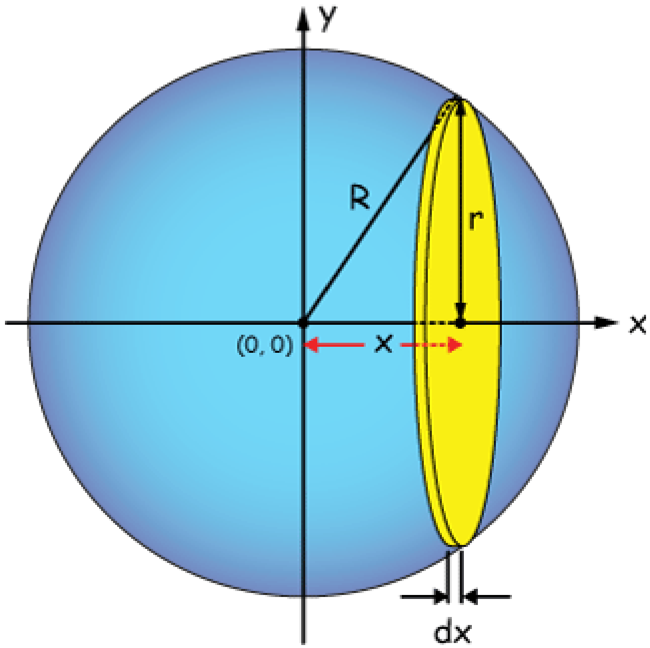



# A **Unit** Circle and **Unit** Sphere

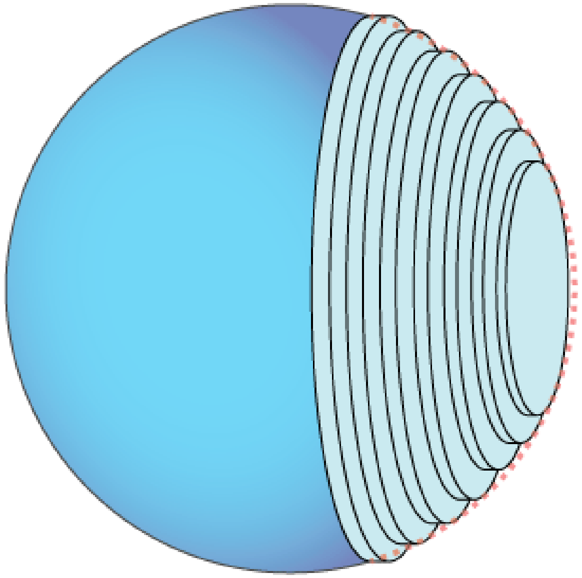


# 2-D Area → 3-D Volume

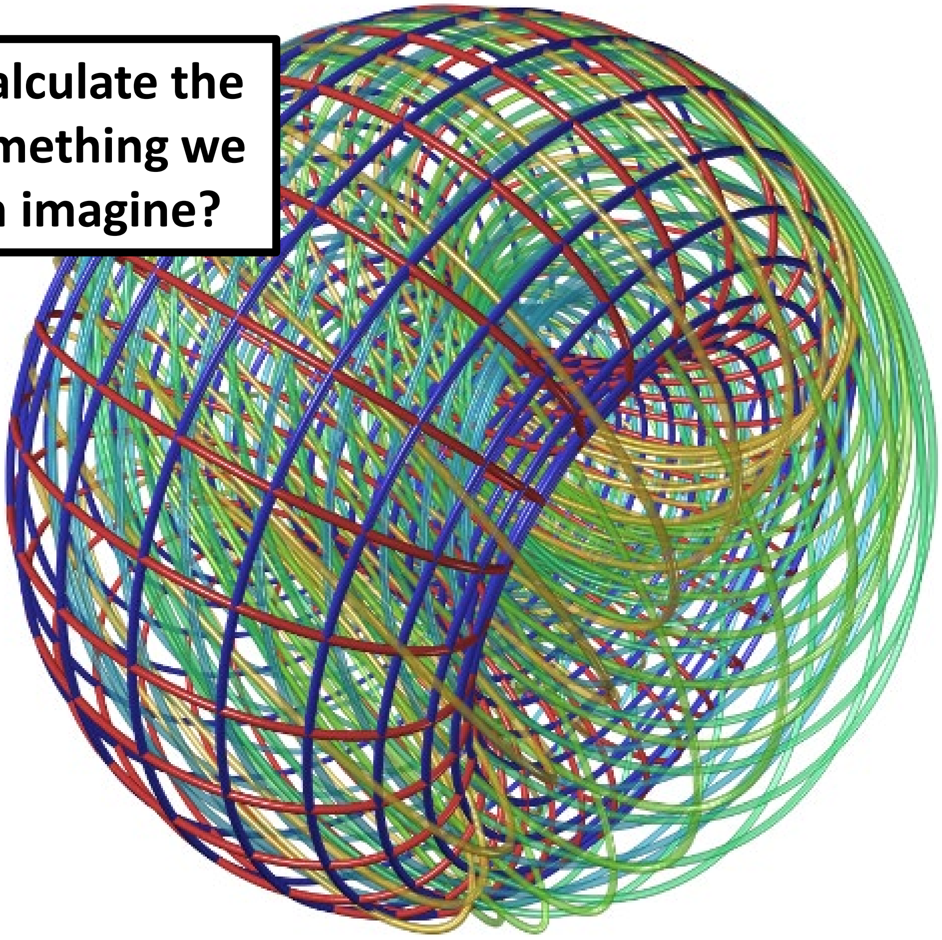



Volume of the disk:
$\pi r^2 \cdot dx = \pi(R^2 - x^2) \cdot dx$

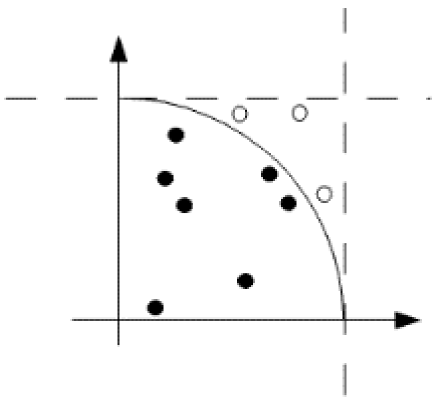


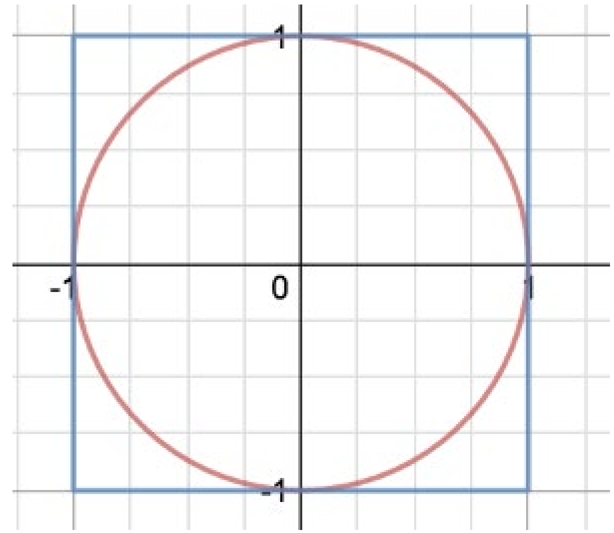
# A 4-D Hypersphere

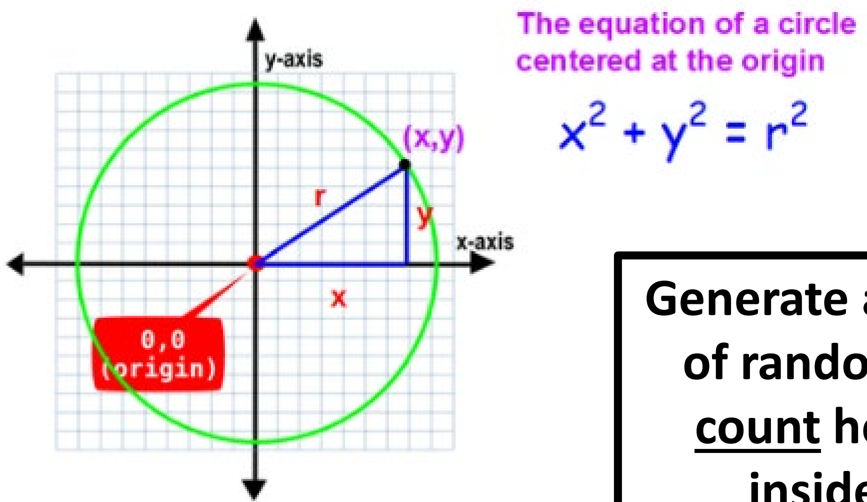
**How do we calculate the volume of something we can not even imagine?**

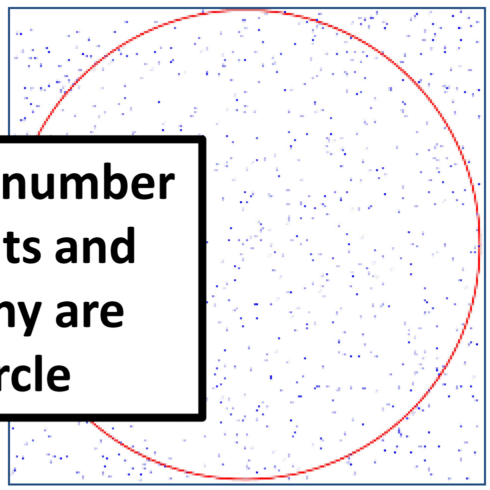


# Area as a “Ratio” of Inside vs. Total Dots



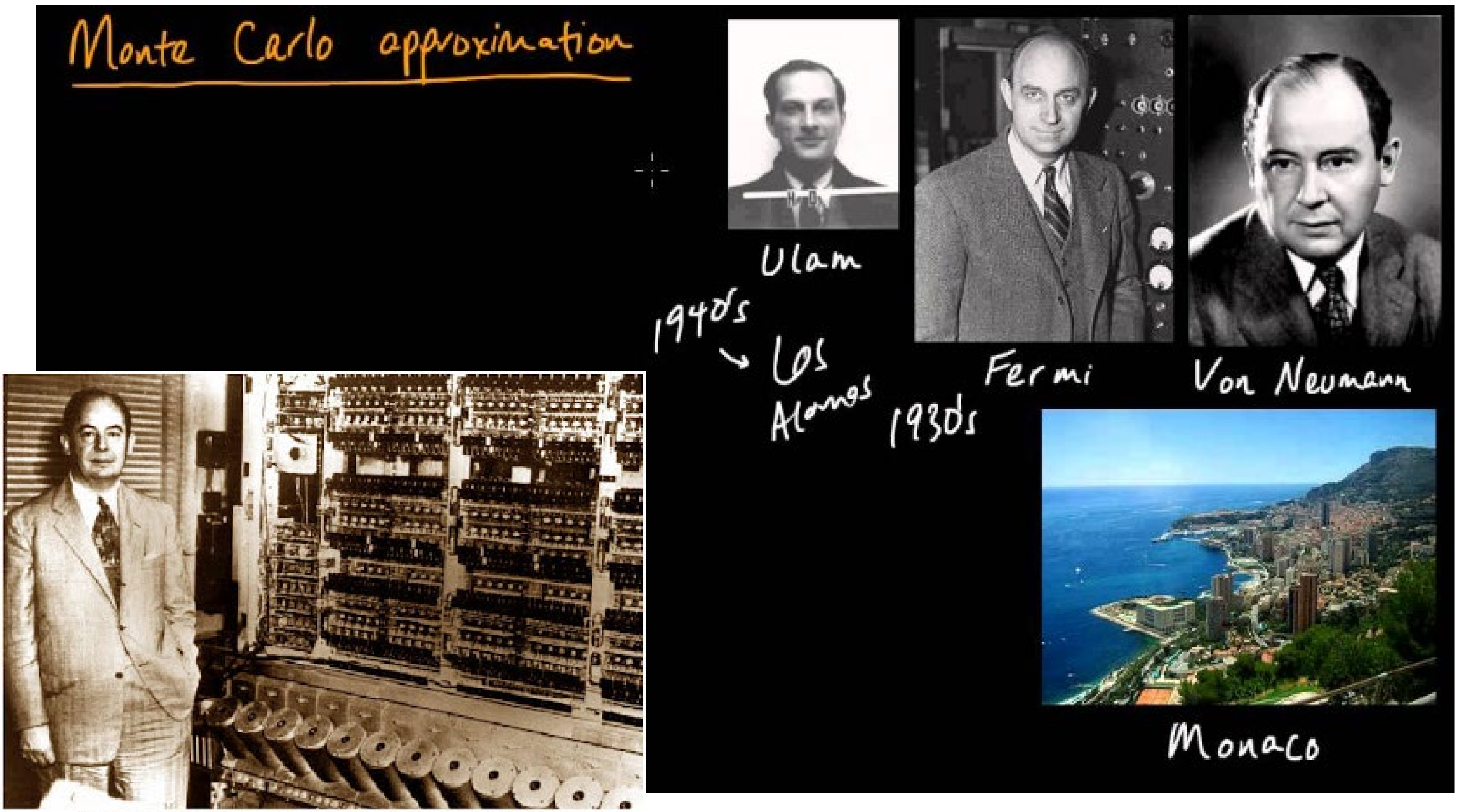


**Generate a large number of random points and <u>count</u> how many are inside the circle**

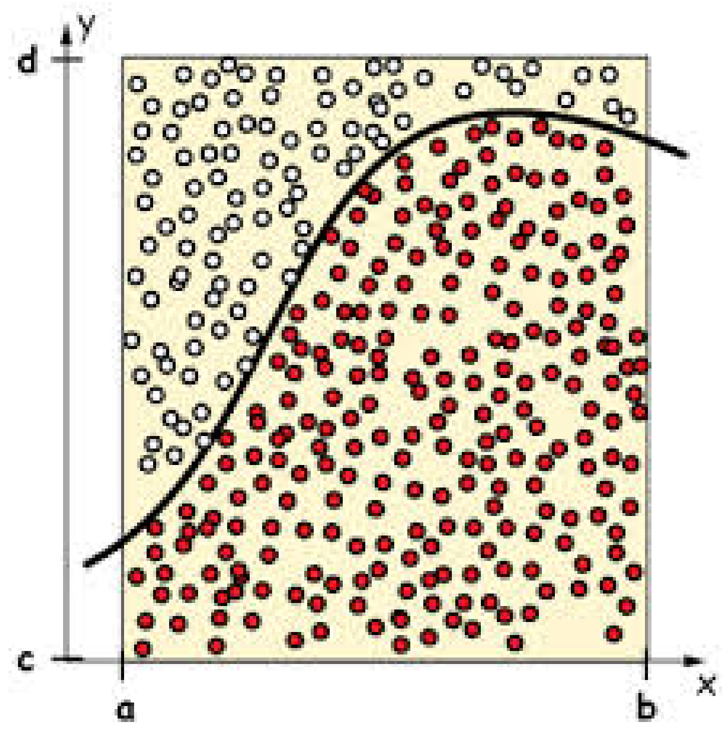
# The Monte Carlo Method



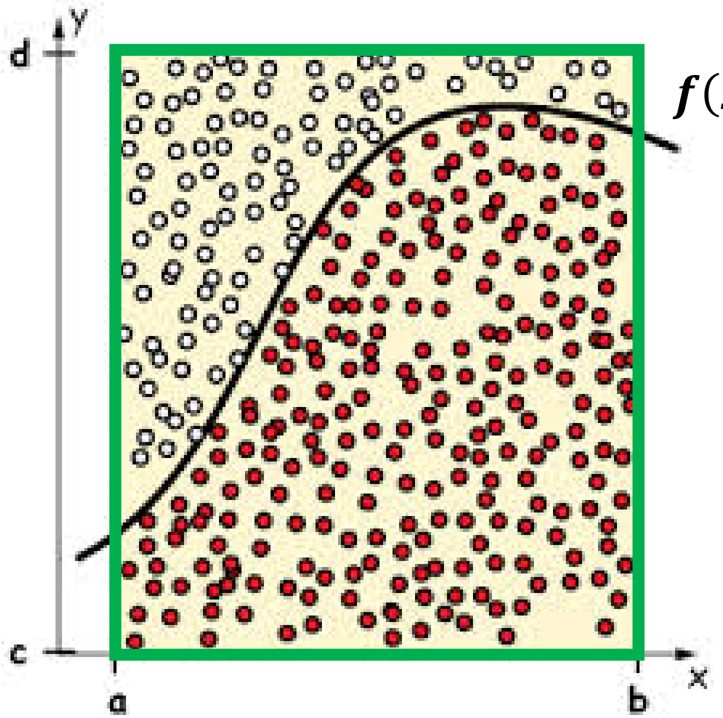

Johnny von Neumann [1903-1957] alongside the Maniac
computer at the Institute for Advanced Studies, Princeton.

# The Monte Carlo Method

- With Monte Carlo, we **randomly sample** points within a bounded space and count how many are ***inside*** the curve
- The ratio of **inside** dots (those under the curve) vs. **total** dots leads to an estimate of the *integral*
- Monte Carlo is **non-deterministic** when a random number generator is used to create the sample points



# The Monte Carlo Method




$f(x)$

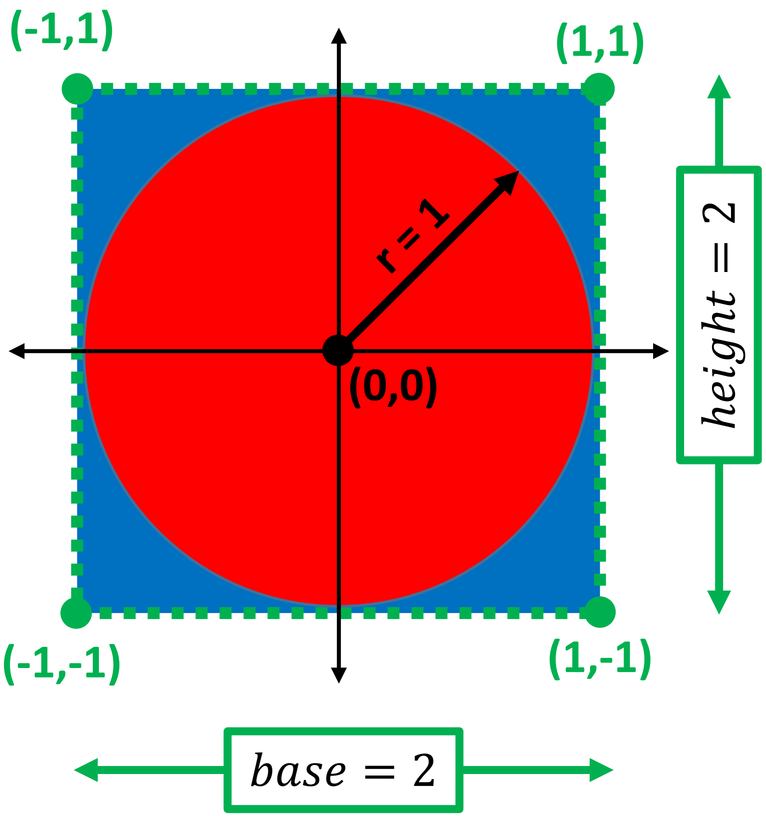
We don't know this area

$$\frac{dots_{inside}}{dots_{total}} = \frac{area_{curve}}{area_{sample}}$$

$$area_{sample} = (b - a) \times (d - c)$$

$$area_{curve} = area_{sample} \times \frac{dots_{inside}}{dots_{total}}$$

# The Monte Carlo Method




We don't know this area

$$\frac{dots_{inside}}{dots_{total}} = \frac{area_{circle}}{area_{sample}}$$

$$area_{sample} = base \times height$$
$$= 2 \times 2$$
$$= 4$$

$$area_{circle} = 4 \times \frac{dots_{inside}}{dots_{total}}$$

# **Run** mc_circle_prng.ipynb – **Cells 1…3**

**Import needed packages/modules**

```
[1]  # Cell 1
     import matplotlib.pyplot as plt
     import numpy as np
```

① 

**Set the total number of random $dots$ (samples) to take**

```
[2]  # Cell 2
     total_dots = 320 * 320 # 102_400
     print(f"{total_dots = :,}")
```

②

```
total_dots = 102,400
```

**Set the numpy PRNG seed to 2020 and take $n$ random samples of 2D Cartesian points $(x, y)$**

1. Use the built-in Python `uniform` distribution which returns a random float [0,1) ③
2. Subtract that float from 1, so the interval flips to (0,1] ensuring any points on the perimeter will now contribute to the area
3. Scale the result so it now falls in the interval [-1, 1]

```
[3]  # Cell 3
     rng = np.random.default_rng(seed=2020)
     x = (1 - rng.random(total_dots)) * 2 - 1
     y = (1 - rng.random(total_dots)) * 2 - 1
     print(x)
     print(y)
```

④

```
[ 0.06338491 -0.02868446 -0.72797656 ... -0.98667256 -0.12884392
  0.59351843]
[-0.16326076  0.21528812 -0.70365621 ... -0.38432811 -0.83694384
  0.52029149]
```

⑤

# Run mc_circle_prng.ipynb – Cells 4…5

**Create an array $d$ that contains the distance from the origin $(0, 0)$ for every point $(x, y)$**

Leverage the fact the exponentiation and addition operators are "vector aware" ①

```
[4] # Cell 4
    d = x**2 + y**2   ②
    print(d)
```

```
[0.03067172 0.04717177 1.02508193 ... 1.12123084 0.71707575 0.62296737]
```
③

**Create arrays of $(x, y)$ coordinates that are "on or inside" vs. "outside" the circle using the Pythagorean distance $d$**

Leverage the ability to `filter` numpy arrays using a conditional expression ④

```
[5] # Cell 5
    x_in = x[d <= 1.0]  # On or inside the circle   ⑤
    y_in = y[d <= 1.0]
    x_out = x[d > 1.0]  # Outside the circle   ⑥
    y_out = y[d > 1.0]
```

# **Run** mc_circle_prng.ipynb – **Cells 6...7**

**Calculate the absolute percent error in the area estimation**

1. The actual/expected area of a unit circle is exactly $\pi$ ①
2. The observed/estimated area using the Monte Carlo formulation $= 4 \times \frac{dots_{inside}}{dots_{total}}$ ②

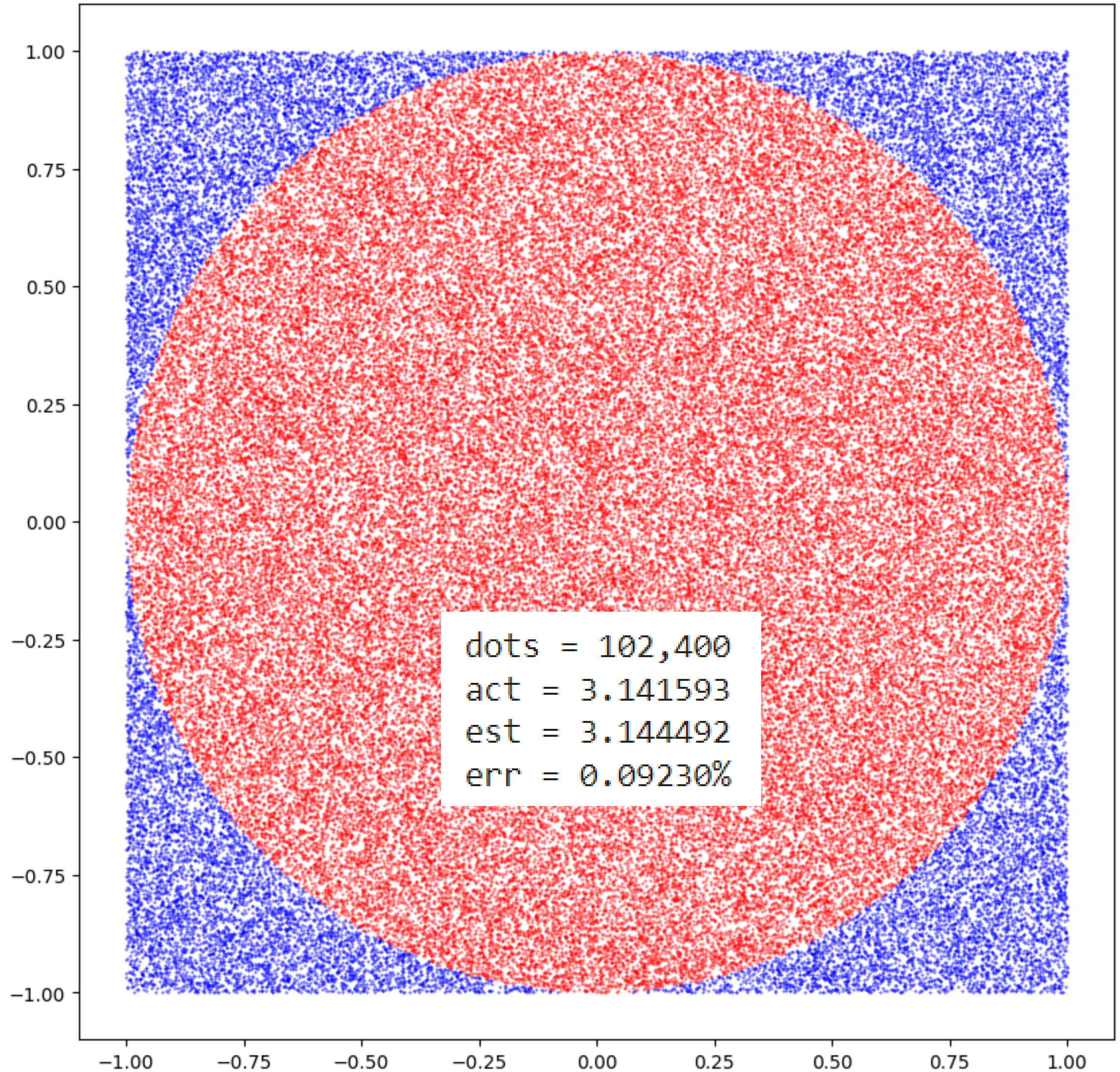
```
[6]  # Cell 6
     act = np.pi
     est = 4 * np.count_nonzero(d <= 1.0) / total_dots     ③
     err = np.abs((est - act) / act)     ④

     print(f"dots = {total_dots:,}")
     print(f"act = {act:.6f}")
     print(f"est = {est:.6f}")
     print(f"err = {err:.5%}")
```

```
dots = 102,400
act = 3.141593     ⑤
est = 3.144492
err = 0.09230%
```

**Display the scatter plot of the Monte Carlo estimation**

```
[7]  # Cell 7
     plt.scatter(x_in, y_in, color="red", marker=".", s=0.5)     ⑥
     plt.scatter(x_out, y_out, color="blue", marker=".", s=0.5)
     plt.gcf().set_size_inches(10, 10)
     plt.gca().set_aspect("equal")
     plt.show()
```

# **Check** mc_circle_prng.ipynb – **Cell 7**



# Run mc_circle_grid.ipynb – **Cells 1…2**

**Import needed packages/modules**

```
[1]  # Cell 1  ①
     import matplotlib.pyplot as plt
     import numpy as np
```

**Set the number of grid intervals along each side ($x$ and $y$) of the sample area**

```
[2]  # Cell 2
     side_dots = 320
     total_dots = side_dots**2  ②
     print(f"{total_dots = :,}")
```

```
total_dots = 102,400  ③
```

# Run mc_circle_grid.ipynb – Cell 3

**Create linear spaces (number of # intervals = `side_dots`) spanning each dimension**

1. Set $-1 \leq x \leq 1$ and $-1 \leq y \leq 1$ ①
2. Create a `meshgrid` at every pairing of the $x$ and $y$ values ②

```
# Cell 3
x = np.linspace(-1, 1, side_dots)
y = np.linspace(-1, 1, side_dots)

xv, yv = np.meshgrid(x, y)
x = xv.flatten()
y = yv.flatten()

print(x)
print(y)
```

③ ④ ⑤

```
[-1.         -0.99373041 -0.98746082 ...  0.98746082  0.99373041
  1.        ]
[-1. -1. -1. ...  1.  1.  1.]
```

⑥

# **Run** mc_circle_grid.ipynb – **Cells 4…5**

**Create an array $d$ that contains the distance from the origin $(0, 0)$ for every point $(x, y)$**

Leverage the fact the exponentiation and addition operators are "vector aware"

```
[4] # Cell 4   ← ①
    d = x**2 + y**2
    print(d)
```

```
[2.         1.98750012 1.97507886 ... 1.97507886 1.98750012 2.        ]
```

**Create arrays of $(x, y)$ coordinates that are "on or inside" vs. "outside" the circle using the Pythagorean distance $d$**

Leverage the ability to `filter` numpy arrays using a conditional expression

```
[6] # Cell 5   ← ②
    x_in = x[d <= 1.0]  # On or inside the circle
    y_in = y[d <= 1.0]
    x_out = x[d > 1.0]  # Outside the circle
    y_out = y[d > 1.0]
```

# Run mc_circle_grid.ipynb – Cells 6…7

**Calculate the absolute percent error in the area estimation**

1. The actual/expected area of a unit circle is exactly $\pi$
2. The observed/estimated area using the uniform grid method $= 4 \times \frac{dots_{\ inside}}{dots_{\ total}}$

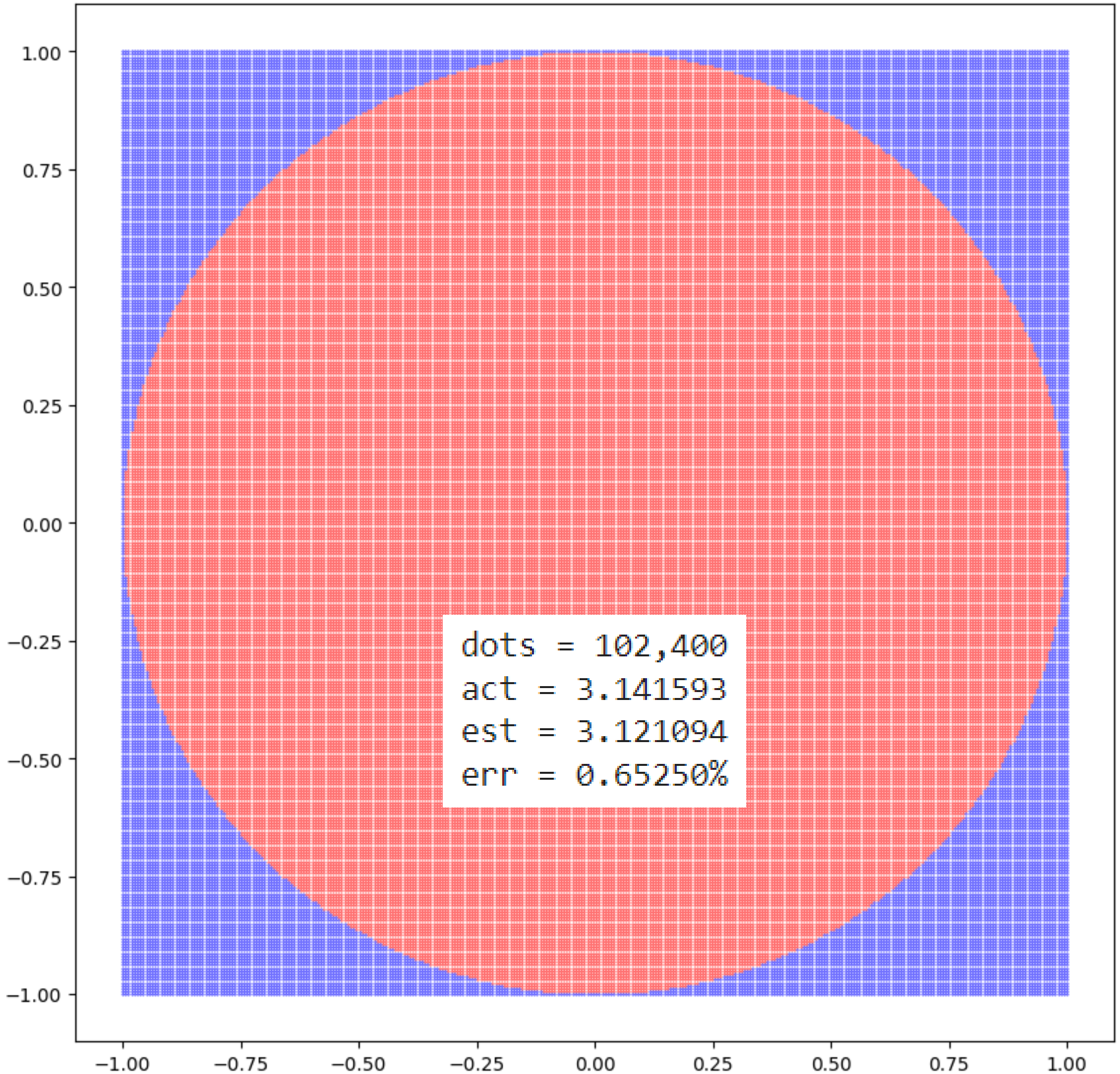
```
[7]  # Cell 6
     act = np.pi
     est = 4 * np.count_nonzero(d <= 1.0) / total_dots      ←  ①
     err = np.abs((est - act) / act)

     print(f"dots = {total_dots:,}")
     print(f"act = {act:.6f}")
     print(f"est = {est:.6f}")
     print(f"err = {err:.5%}")
```

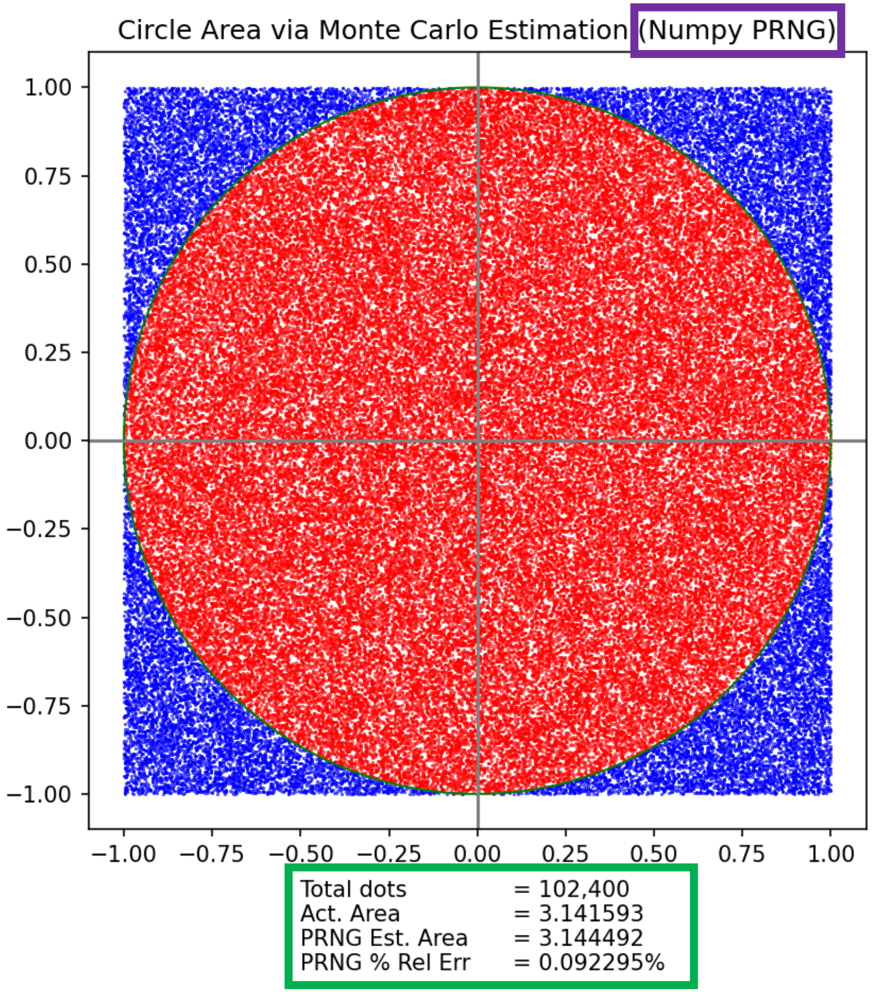
```
dots = 102,400
act = 3.141593
est = 3.121094
err = 0.65250%
```

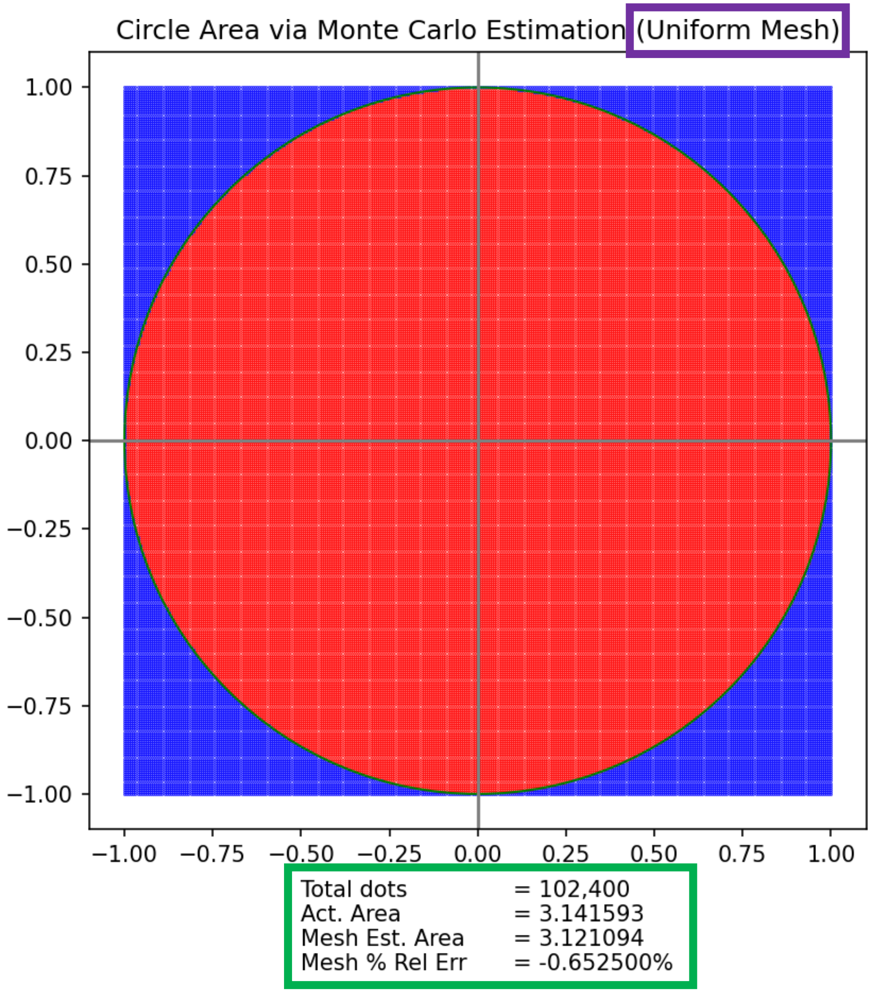
**Display the scatter plot of the Monte Carlo estimation**

```
[8]  # Cell 7
     plt.scatter(x_in, y_in, color="red", marker=".", s=0.5)      ←  ②
     plt.scatter(x_out, y_out, color="blue", marker=".", s=0.5)
     plt.gcf().set_size_inches(10, 10)
     plt.gca().set_aspect("equal")
     plt.show()
```

# **Check** mc_circle_grid.ipynb






Taking random samples was **607**% more accurate than using a uniform mesh!

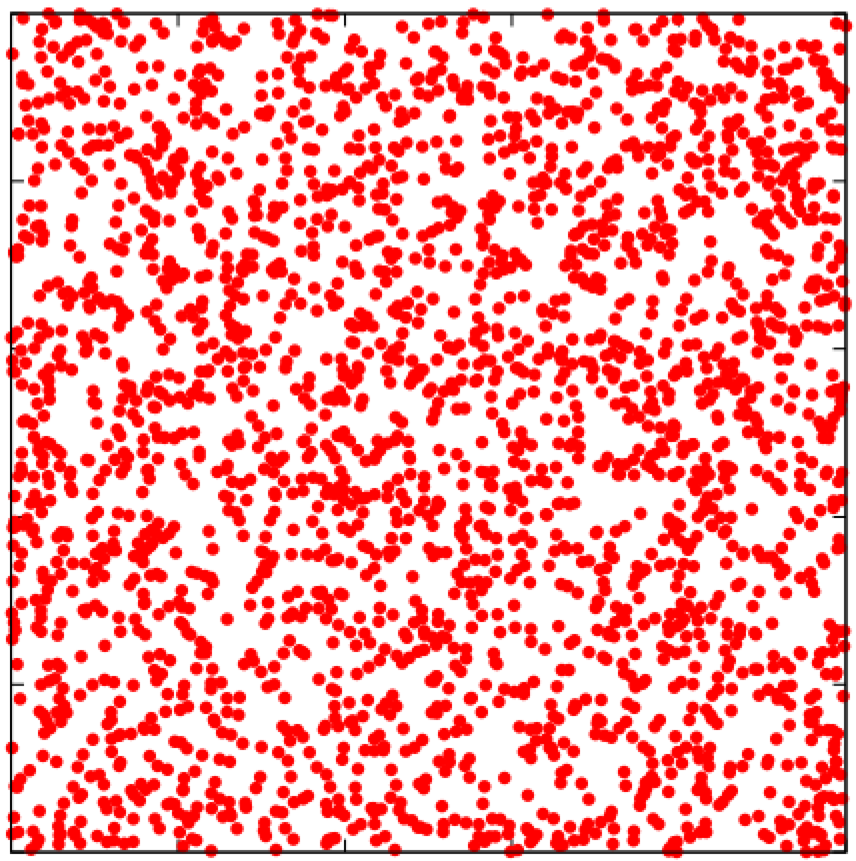
# Monte Carlo Questions

- The random Monte Carlo approach and the version based upon taking uniformly spaced samples along a Cartesian (orthogonal) grid **used the same number of samples**
- The **MC** approach resulted in **607% reduction in relative error** compared to the simple grid method – why?
- What is the **underlying issue** that can force a uniformly spaced sampling approach to miscount the dots inside vs. outside the *circle*?
- Consider an individual mesh square that overlaps the perimeter of the circle - how does the rigid placement of the corners of each **square** affect the ***accuracy*** of the estimate of the **curve**?

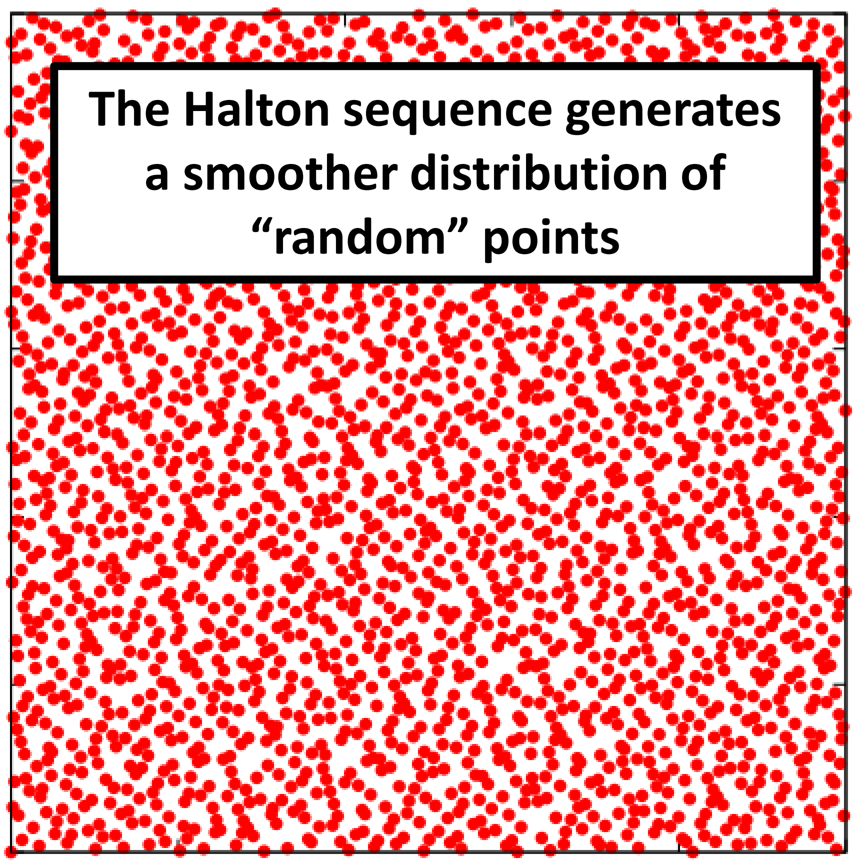
# Comparing "Random" Number Generators

A quasi-random number generator

**Standard PRNG**



**Halton QRNG**



# The Halton Sequence


**Mathematical Proceedings of the Cambridge Philosophical Society**

Mathematical Proceedings of the Cambridge Philosophical Society / Volume 61 / Issue 02 / April 1965, pp 497-498



DOI: http://dx.doi.org/10.1017/S0305004100004059 (About DOI), Published online: 24 October 2008

Table of Contents - 1965 - Volume 61, Issue 02


```
def halton(n, p):
    h, f = 0, 1
    while n > 0:
        f = f / p
        h += (n % p) * f
        n = int(n / p)
    return h
```

Buy This Article $45.00 / £30.00 | Rent This Article Now for 24 Hours $5.99 / £3.99 / €4.49 | Request Permissions | < Previous Abstract | Ne

Research Article

## On the relative merits of correlated and importance sampling for Monte Carlo integration


John H. Halton[a1]

[a1] Brookhaven National Laboratory, Upton, New York

# Run mc_circle_halton.ipynb – Cells 1…3

**Import needed packages/modules**

```
[1]  # Cell 1
     import matplotlib.pyplot as plt
     import numpy as np
     from numba import int64, float64, vectorize     ①
```

**Declare a numba accelerated function that computes the Halton QRNG**

1. The parameter $n$ is an integer of any size
2. The parameter $p$ is a prime number ②

```
[2]  # Cell 2
     @vectorize([float64(int64, int64)], nopython=True)     ③
     def halton(n, p):
         h, f = 0, 1
         while n > 0:
             f = f / p
             h += (n % p) * f
             n = int(n / p)
         return h
```

**Set the number of random $dots$ (samples) to take**

```
[3]  # Cell 3
     total_dots = 25_600     ④
```

# Run mc_circle_halton.ipynb – **Cells 4...5**

**Take $n$ "random" samples of 2D Cartesian points $(x, y)$ using the Halton sequence** ①

1. The Halton QRNG returns a random float [0,1)
2. Subtract that float from 1, so the interval flips to (0,1] ensuring any points on the perimeter will now contribute to the area
3. Scale the result so it now falls in the interval [-1, 1]

```
[4] # Cell 4
    x = (1 - halton(np.arange(total_dots), 2)) * 2 - 1
    y = (1 - halton(np.arange(total_dots), 3)) * 2 - 1
    print(x)
    print(y)
```
②

```
[ 1.          0.          0.5        ... -0.49822998  0.00177002
 -0.99822998]
[ 1.          0.33333333 -0.33333333 ...  0.32263036 -0.34403631
  0.7670748 ]
```
③

**Create an array $d$ that contains the distance from the origin $(0, 0)$ for every point $(x, y)$**

Leverage the fact the exponentiation and addition operators are "vector aware" ④

```
[5] # Cell 5
    d = x**2 + y**2
    print(d)
```
⑤

```
[2.         0.11111111 0.36111111 ... 0.35232346 0.11836411 1.58486685]
```
⑥

# **Run** mc_circle_halton.ipynb – **Cells 6...7**

**Create arrays of $(x, y)$ coordinates that are "on or inside" vs. "outside" the circle using the Pythagorean distance $d$**

Leverage the ability to `filter` numpy arrays using a conditional expression

```
[6] # Cell 6
    x_in = x[d <= 1.0] # On or inside the circle
    y_in = y[d <= 1.0]
    x_out = x[d > 1.0] # Outside the circle
    y_out = y[d > 1.0]
```

①

**Calculate the absolute percent error in the area estimation**

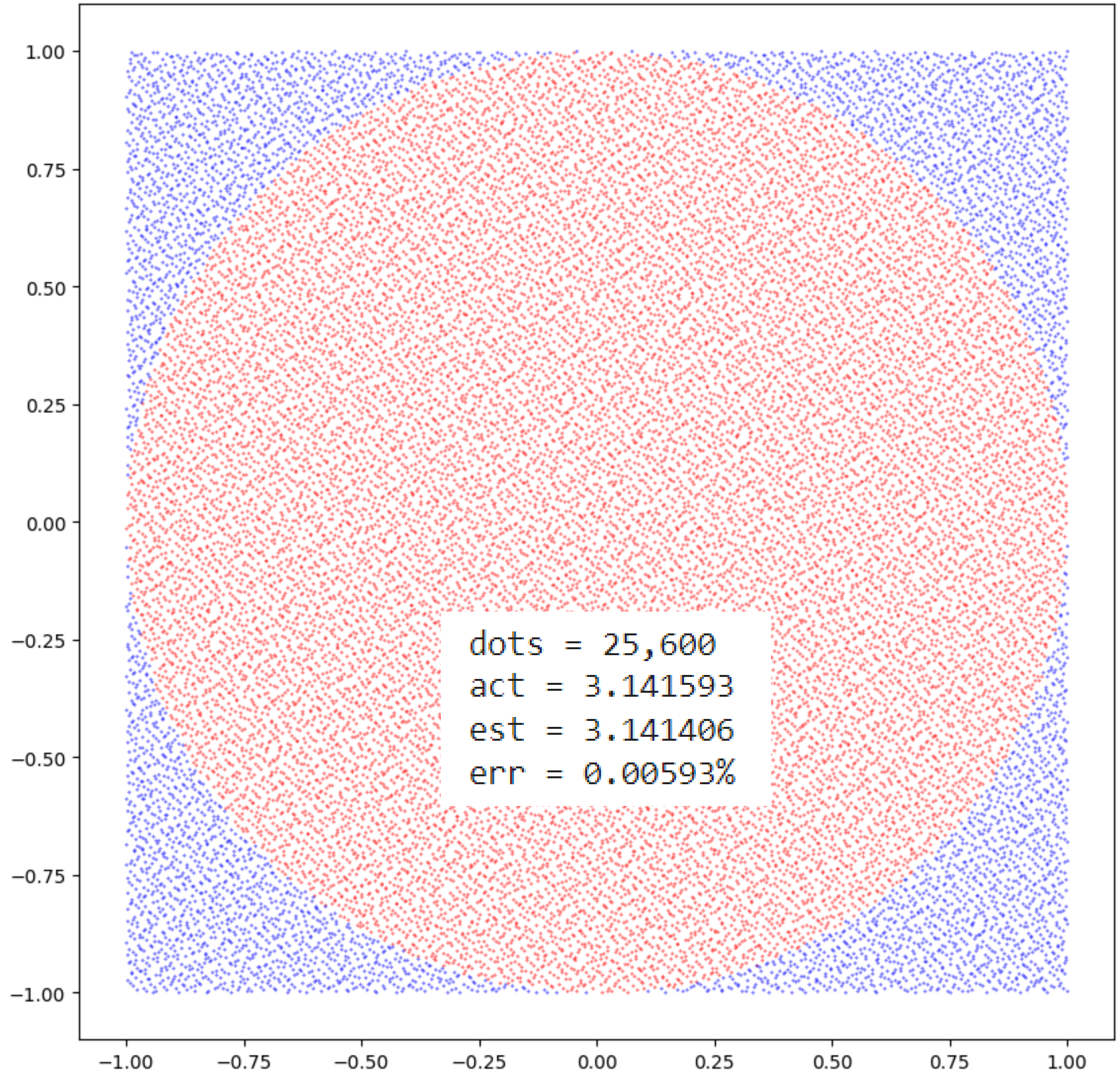
1. The actual/expected area of a unit circle is exactly $\pi$
2. The observed/estimated area using the Monte Carlo formulation $= 4 \times \frac{dots_{\ inside}}{dots_{\ total}}$

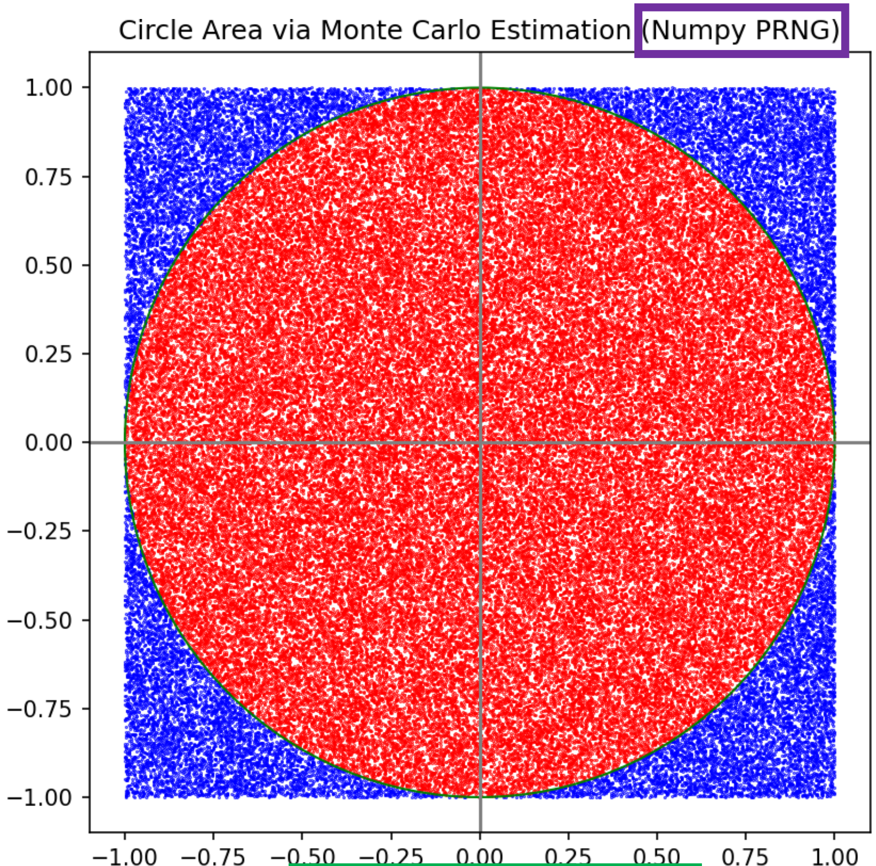
```
[7] # Cell 7
    act = np.pi
    est = 4 * np.count_nonzero(d <= 1.0) / total_dots
    err = np.abs((est - act) / act)

    print(f"dots = {total_dots:,}")
    print(f"act = {act:.6f}")
    print(f"est = {est:.6f}")
    print(f"err = {err:.5%}")
```

②

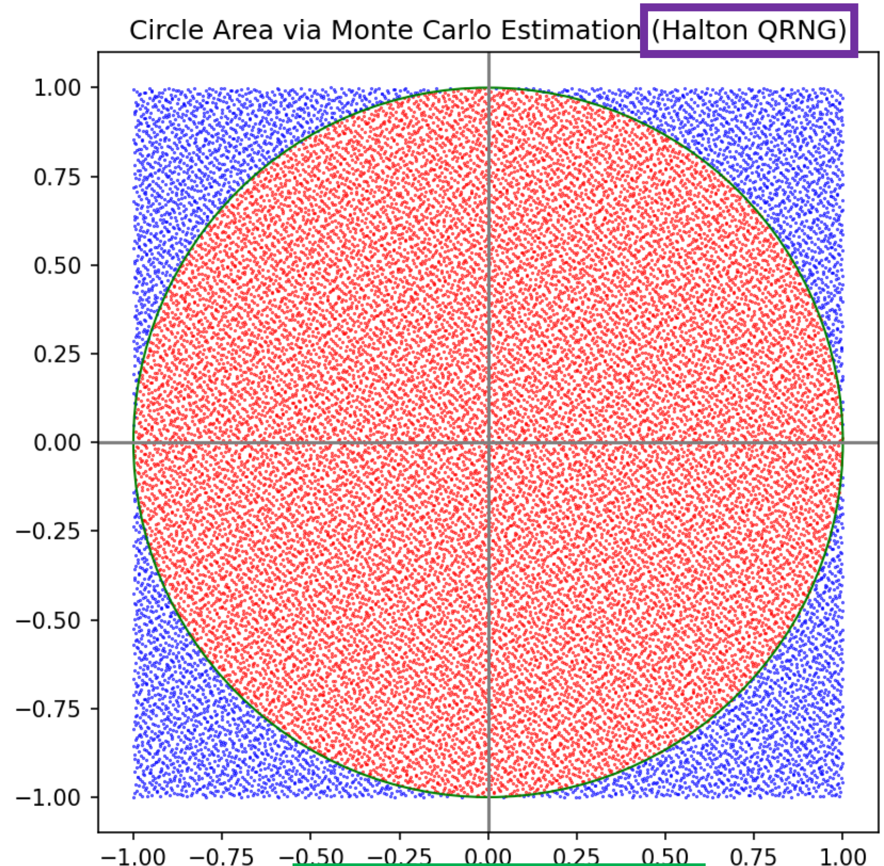
```
dots = 25,600
act = 3.141593
est = 3.141406
err = 0.00593%
```

# **Check** mc_circle_halton.ipynb – **Cell 8**






Total dots = 102,400
Act. Area = 3.141593
PRNG Est. Area = 3.144492
PRNG % Rel Err = 0.092295%

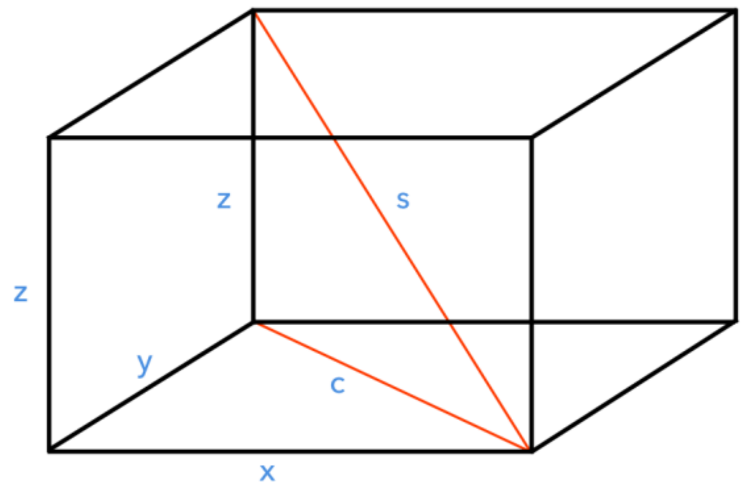



Total dots = 25,600
Act. Area = 3.141593
QRNG Est. Area = 3.141406
QRNG % Rel Err = -0.005933%

The Halton QRNG MC was **1,456**% more accurate than the PRNG MC while needing **300%** fewer samples!

# Moving to Higher Dimensions

The Pythagorean Distance is a **metric** that is true in all **orthogonal** spaces of any dimension




$c^2 = x^2 + y^2$

$s^2 = c^2 + z^2$

$$\therefore s^2 = x^2 + y^2 + z^2$$

# Run mc_sphere.ipynb – Cells 1…3

**Import needed packages/modules**

```
[1]  # Cell 1
     import matplotlib.pyplot as plt
     import numpy as np
     from numba import float64, int64, vectorize
```
①

**Declare a numba accelerated function that computes the Halton QRNG**

1. The parameter $n$ is an integer of any size
2. The parameter $p$ is a prime number

```
[2]  # Cell 2
     @vectorize([float64(int64, int64)], nopython=True)
     def halton(n, p):
         h, f = 0, 1
         while n > 0:
             f = f / p
             h += (n % p) * f
             n = int(n / p)
         return h
```
②

**Set the total number of dots (samples) to take**

```
[3]  # Cell 3
     total_dots = 125_000
```
③

# Run mc_sphere.ipynb – Cells 4…6

**Create `total_dots` samples of 3D Cartesian points $(x, y, z)$ using the Halton sequence**

1. The Halton QRNG returns a random float [0,1)
2. Subtract that float from 1, so the interval flips to (0,1] ensuring any points on the perimeter will now contribute to the volume
3. Scale the result so it now falls in the interval [-1, 1]

```
[4]  # Cell 4
     x = (1 - halton(np.arange(total_dots), 2)) * 2 - 1
     y = (1 - halton(np.arange(total_dots), 3)) * 2 - 1    ← ①
     z = (1 - halton(np.arange(total_dots), 5)) * 2 - 1
```

**Create an array $d$ that contains the distance from the origin $(0, 0)$ for every point $(x, y, z)$**

Leverage the fact the exponentiation and addition operators are "vector aware"

```
[5]  # Cell 5
     d = x**2 + y**2 + z**2    ← ②
```

**Create arrays of $(x, y)$ coordinates that are "on or inside" vs. "outside" the sphere using the Pythagorean distance $d$**

Leverage the ability to `filter` numpy arrays using a conditional expression

```
[6]  # Cell 6

     # On the surface (or inside) the sphere
     x_in = x[d <= 1.0]
     y_in = y[d <= 1.0]
     z_in = z[d <= 1.0]    ← ③

     # Outside the sphere
     x_out = x[d > 1.0]
     y_out = y[d > 1.0]
     z_out = z[d > 1.0]
```

# **Run** mc_sphere.ipynb – **Cells 7…8**

**Calculate the absolute percent error in the area estimation**

1. The actual/expected volume of a unit sphere is exactly $\frac{4}{3}\pi$
2. The observed/estimated volume using the Monte Carlo formulation $= 8 \times \frac{dots_{\ inside}}{dots_{\ total}}$

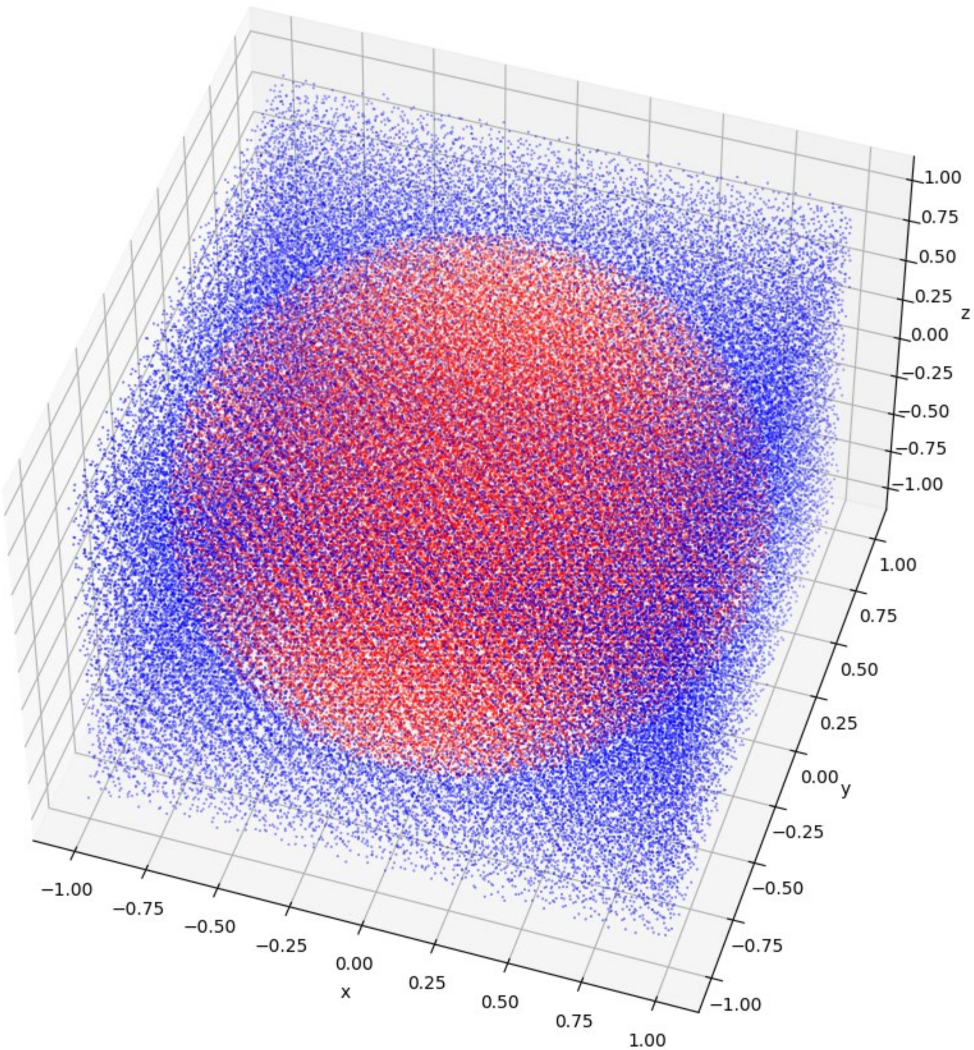
```
[7] # Cell 7
    act = 4 / 3 * np.pi
    est = 8 * np.count_nonzero(d <= 1.0) / total_dots    ← ①
    err = np.abs((est - act) / act)

    print(f"dots = {total_dots:,}")
    print(f"act = {act:.6f}")
    print(f"est = {est:.6f}")
    print(f"err = {err:.5%}")
```

```
dots = 125,000
act = 4.188790    ← ②
est = 4.188416
err = 0.00893%
```

**Display the scatter plot of the Monte Carlo estimation**

```
[8] # Cell 8
    ax = plt.axes(projection="3d")    ← ③
    ax.view_init(azim=-72, elev=48)
    ax.scatter(x_in, y_in, z_in, color="red", marker=".", s=0.1)    ← ④
    ax.scatter(x_out, y_out, z_out, color="blue", marker=".", s=0.1)
    plt.gcf().set_size_inches(10, 10)
    plt.gca().set_aspect("equal")
    plt.show()
```

# **Check** mc_sphere.ipynb




We just estimated the volume of a unit sphere to within **0.009%** without a stitch of *calculus* and using nothing but <u>random</u> numbers!

```
Total dots         = 125,000
Act. Volume        = 4.188790
PRNG Est. Volume   = 4.195392
PRNG % Rel Err     = 0.157606%

Total dots         = 125,000
Act. Volume        = 4.188790
QRNG Est. Volume   = 4.188416
QRNG % Rel Err     = -0.008933%
```

QRNG is **1,664%** more accurate than the PRNG

# Run mc_hypersphere.ipynb – Cells 1…3

**Import needed packages/modules**

```
[1]  # Cell 1
     import matplotlib.pyplot as plt
     import numpy as np
     from numba import int64, float64, vectorize      ①
```

**Declare a numba accelerated function that computes the Halton QRNG**

1. The parameter $n$ is an integer of any size
2. The parameter $p$ is a prime number

```
[2]  # Cell 2                                              ②
     @vectorize([float64(int64, int64)], nopython=True)
     def halton(n, p):
         h, f = 0, 1
         while n > 0:
             f = f / p
             h += (n % p) * f
             n = int(n / p)
         return h
```

**Set the total number of samples to take**

```
[3]  # Cell 3
     total_dots = 6_250_000      ③
```

# Run mc_hypersphere.ipynb – Cells 4...5

**Create `total_dots` samples of 4D Cartesian points $(x, y, z, w)$ using the Halton sequence**

1. The Halton QRNG returns a random float [0,1)
2. Subtract that float from 1, so the interval flips to (0,1] ensuring any points on the perimeter will now contribute to the "content"
3. Scale the result so it now falls in the interval [-1, 1]

```
[4]  # Cell 4
     x = (1 - halton(np.arange(total_dots), 2)) * 2 - 1
     y = (1 - halton(np.arange(total_dots), 3)) * 2 - 1
     z = (1 - halton(np.arange(total_dots), 5)) * 2 - 1
     w = (1 - halton(np.arange(total_dots), 7)) * 2 - 1   ← ①
```

**Create an array $d$ that contains the distance from the origin $(0, 0, 0, 0)$ for every point $(x, y, z, w)$**

Leverage the fact the exponentiation and addition operators are "vector aware"

```
[5]  # Cell 5
     d = x**2 + y**2 + z**2 + w**2   ← ②
```

# Run mc_hypersphere.ipynb – Cell 6

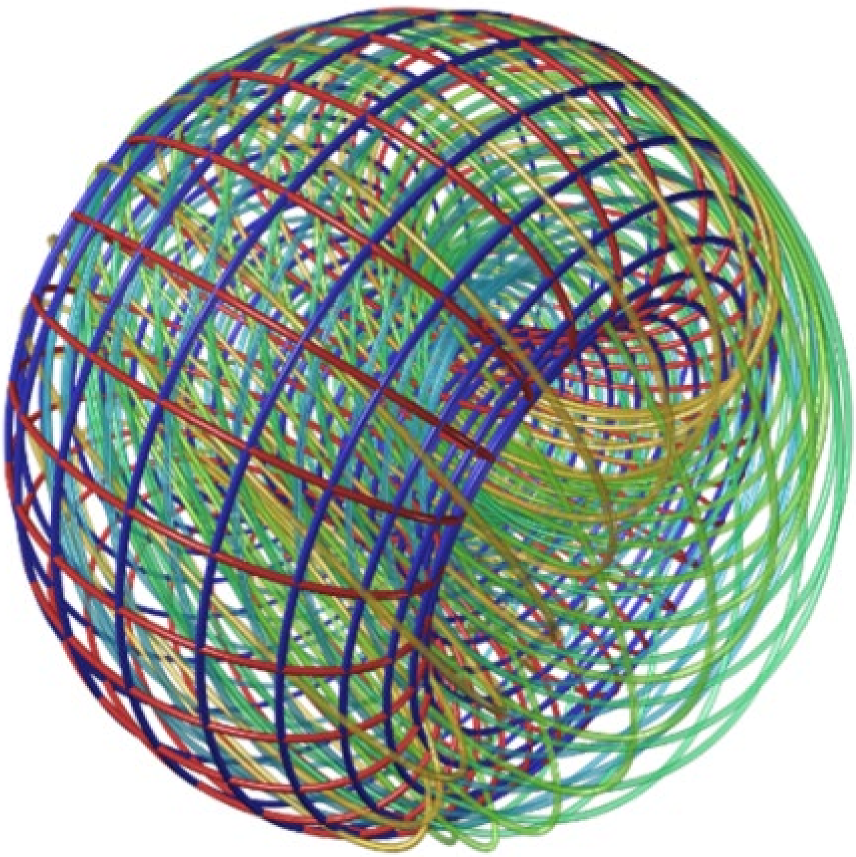
**Calculate the absolute percent error in the content estimation**

1. The actual/expected content of a unit hypersphere is exactly $\frac{\pi^2}{2}$
2. The observed/estimated content using the Monte Carlo formulation $= 16 \times \frac{dots_{\ inside}}{dots_{\ total}}$ ←①

```
[6] # Cell 7
    act = np.pi**2 / 2
    est = 16 * np.count_nonzero(d <= 1.0) / total_dots   ←②
    err = np.abs((est - act) / act)

    print(f"dots = {total_dots:,}")
    print(f"act = {act:.6f}")
    print(f"est = {est:.6f}")
    print(f"err = {err:.5%}")
```

```
dots = 6,250,000
act = 4.934802   ←③
est = 4.934543
err = 0.00525%
```

# An *Interesting* Question



**What is the volume of a 4-D unit hypersphere?**

Act. Volume = 4.934802

Est. Volume = 4.934543

% Rel Err = -0.005245%

$$= \frac{\pi^2}{2}$$

**Yes, we can calculate the volume of something we can not even *imagine!***

# A Recurrence Relation

*The **content** of an **n-ball** is proportional to the unit ball for that dimension*

$$V_n(R) = V_n(1)R^n$$

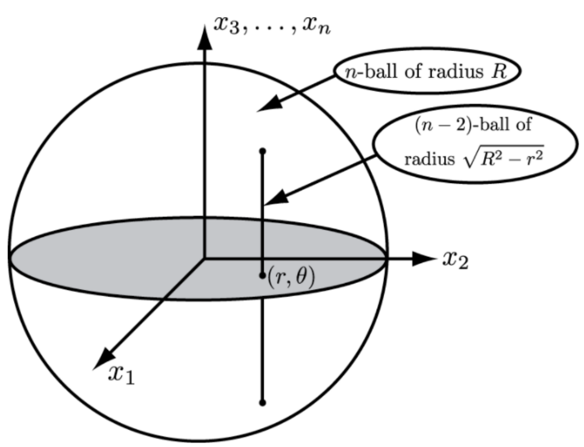
We can compute $V_n(1)$ by integrating the $n-2$ ball over a **unit** disk using polar coordinates

$$V_n(1) = \int_0^1 \int_0^{2\pi} V_{n-2}(1)\left(\sqrt{1-r^2}\right)^{n-2} r\, d\theta\, dr$$

$$= V_{n-2}(1) \int_0^1 r(1-r^2)^{\frac{n-2}{2}}\, \theta\Big|_0^{2\pi} dr$$

$$= 2\pi V_{n-2}(1) \int_0^1 r(1-r^2)^{\frac{n-2}{2}}\, dr$$

$$V_n(1) = \frac{2\pi}{n} V_{n-2}(1)$$



# A Recurrence Relation

$$V_n(1) = \frac{2\pi}{n} V_{n-2}(1)$$

$$V_n(R) = V_n(1) R^n$$

By definition

$$V_o(1) = 1$$

$$V_o(R) = 1$$

$1 - (-1) = 2$

$$V_1(1) = 2$$

$$V_1(R) = 2R$$

$$V_2(1) = \frac{2\pi}{2}(1) = \pi$$

$$V_2(R) = \pi R^2$$

$$V_3(1) = \frac{2\pi}{3}(2) = \frac{4}{3}\pi$$

$$V_3(R) = \frac{4}{3}\pi R^3$$

$$V_4(1) = \frac{2\pi}{4}(\pi) = \frac{\pi^2}{2}$$

$$V_4(R) = \frac{\pi^2}{2} R^4$$

# Volume via the Gamma Function

$$V_n(R) = \frac{\pi^{\frac{n}{2}} R^n}{\Gamma\left(\frac{n}{2}+1\right)}$$

$\Gamma(n) = (n-1)!$

$n! = \Gamma(n+1)$

$$V_0(R) = \frac{\pi^{\frac{0}{2}} R^0}{\Gamma\left(\frac{0}{2}+1\right)} = \frac{1}{(1-1)!} = 1$$

$$V_2(R) = \frac{\pi R^2}{\Gamma\left(\frac{2}{2}+1\right)} = \frac{\pi R^2}{\Gamma(2)} = \frac{\pi R^2}{(2-1)!} = \boxed{\pi R^2}$$

$$V_3(R) = \frac{\pi^{\frac{3}{2}} R^3}{\Gamma\left(\frac{3}{2}+1\right)} = \frac{\pi R^3}{\Gamma\left(\frac{5}{3}\right)} = \frac{\pi^{\frac{3}{2}} R^3}{\left(\frac{3\sqrt{\pi}}{4}\right)} = \pi^{\frac{3}{2}} R^3 \left(\frac{4}{3\sqrt{\pi}}\right) = \boxed{\frac{4}{3}\pi R^3}$$

$$V_4(R) = \frac{\pi^{\frac{4}{2}} R^4}{\Gamma\left(\frac{4}{2}+1\right)} = \frac{\pi^2 R^2}{\Gamma(3)} = \frac{\pi^2 R^2}{(3-1)!} = \boxed{\frac{\pi^2 R^4}{2}}$$

# Volume via the Gamma Function

$$V_n(R) = \frac{\pi^{\frac{n}{2}} R^n}{\Gamma\left(\frac{n}{2} + 1\right)}$$

Because we can evaluate $\boldsymbol{\Gamma(x)}$ at every point in $\mathbb{R}$ we can now determine the volume of a unit hypersphere in *any* dimension

$$V_{\mathbf{7.89}}(5.12) = \frac{\pi^{\frac{7.89}{2}} 5.12^{7.89}}{\Gamma\left(\frac{7.89}{2} + 1\right)} = \mathbf{1{,}633{,}106.2809}$$

As the Gamma function can extends its domain to include $\boldsymbol{n} \in \mathbb{R}$, we can use this analytic solution to compute the volume of hyperspheres having **fractional** (non-integer) dimensions!

# **Run** mc_high_dimensions.ipynb – **Cells 1…3**

**Import needed packages/modules**

```
[1] # Cell 1
    import matplotlib.pyplot as plt
    import numpy as np
    import sympy
    from matplotlib.ticker import AutoMinorLocator, MultipleLocator
    from numba import float64, int64, vectorize
    from scipy.signal import find_peaks
    from scipy.special import gamma
```

①

**Declare a numba accelerated function that computes the Halton QRNG**

1. The parameter $n$ is an integer of any size
2. The parameter $p$ is a prime number

```
[2] # Cell 2
    @vectorize([float64(int64, int64)], nopython=True)
    def halton(n, p):
        h, f = 0, 1
        while n > 0:
            f = f / p
            h += (n % p) * f
            n = int(n / p)
        return h
```

②

**Set the total number of samples to take**

```
[3] # Cell 3
    total_dots = 6_250_000
```

③

# **Run** mc_high_dimensions.ipynb – **Cell 4**

**Estimate the content of `n-balls` from dimension 1 to 13**

1. Use `sympy` to provide the Halton generator the correct prime number for each successive dimension
2. We only need to keep a single accumulating $d$ value to represent the distance to origin for a point as we add for each dimension
3. The Monte Carlo sample space multiplier grows by $2^{dimension}$ ①

```
[4]  # Cell 4
     dimensions = 13
     d = np.zeros(total_dots)
     est = np.zeros(dimensions)
     est[0] = 1  # By definition
     est[1] = 2  # The 1-D line in the interval [-1,1] has "area" (length) 2        ②

     for dim in np.arange(1, dimensions):        ③
         print(f"Calculating the volume of a unit {dim:>2}-ball . . .")
         v = halton(np.arange(total_dots), sympy.prime(dim)) * 2 - 1        ④
         d += v**2
         est[dim] = 2**dim * np.count_nonzero(d <= 1.0) / total_dots        ⑤
```

```
Calculating the volume of a unit  1-ball . . .
Calculating the volume of a unit  2-ball . . .
Calculating the volume of a unit  3-ball . . .
Calculating the volume of a unit  4-ball . . .
Calculating the volume of a unit  5-ball . . .
Calculating the volume of a unit  6-ball . . .        ⑥
Calculating the volume of a unit  7-ball . . .
Calculating the volume of a unit  8-ball . . .
Calculating the volume of a unit  9-ball . . .
Calculating the volume of a unit 10-ball . . .
Calculating the volume of a unit 11-ball . . .
Calculating the volume of a unit 12-ball . . .
```

# Run mc_high_dimensions.ipynb – **Cell 5**

**Using the analytic solution, calculate the dimension and content for the <u>largest</u> unit `n-ball`**

$$V_n(R) = \frac{\pi^{\frac{n}{2}} R^n}{\Gamma\left(\frac{n}{2}+1\right)}$$ ①

```
[5]  # Cell 5
     act_x = np.linspace(0, dimensions - 1, 1000)                       ②
     act_y = np.power(np.pi, act_x / 2) / gamma(act_x / 2 + 1)          ③
     m = find_peaks(act_y)[0][0]                                        ④
     print(f"max dim = {act_x[m]:.6f}")
     print(f"max vol = {act_y[m]:.6f}")
```

```
max dim = 5.261261                                                      ⑤
max vol = 5.277764
```

# Run mc_high_dimensions.ipynb – **Cell 6**

**Plot the estimated and actual `n-ball` content vs. dimension**

```
[6]  # Cell 6
     plt.figure(figsize=(12, 8))
     plt.plot(np.arange(dimensions), est, color="blue", label="Estimated")  ←  ①
     plt.plot(act_x, act_y, color="red", label="Actual")  ←  ②
     plt.scatter(act_x[m], act_y[m], marker="o", color="green")
     plt.vlines(act_x[m], 0, act_y[m], color="green")
     plt.title("Volume of n-Dimensional Hyperspheres")
     plt.xlabel("Dimension")
     plt.ylabel("Volume")
     ax = plt.gca()
     ax.xaxis.set_major_locator(MultipleLocator(1))
     ax.xaxis.set_minor_locator(MultipleLocator(0.5))  ←  ③
     ax.yaxis.set_minor_locator(AutoMinorLocator())
     ax.legend(loc="upper right")  ←  ④
     ax.grid("on")
     plt.show()
```

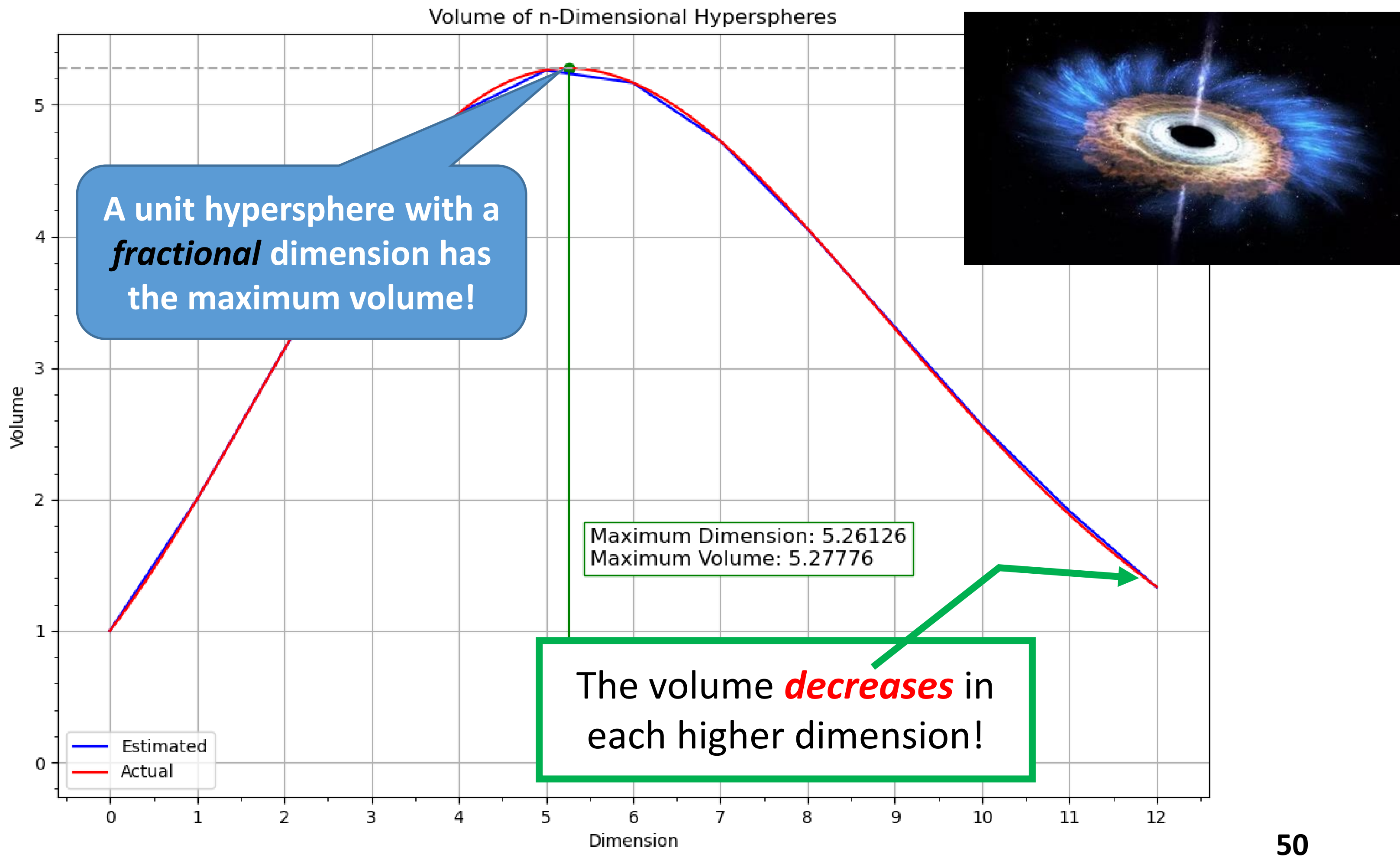
# Monte Carlo Estimation of n-Ball Content

**What lurks beyond the 4th dimension?**

```
Calculating the volume of a unit  1-ball . . .
Calculating the volume of a unit  2-ball . . .
Calculating the volume of a unit  3-ball . . .
Calculating the volume of a unit  4-ball . . .
Calculating the volume of a unit  5-ball . . .
Calculating the volume of a unit  6-ball . . .
Calculating the volume of a unit  7-ball . . .
Calculating the volume of a unit  8-ball . . .
Calculating the volume of a unit  9-ball . . .
Calculating the volume of a unit 10-ball . . .
Calculating the volume of a unit 11-ball . . .
Calculating the volume of a unit 12-ball . . .
```

# Monte Carlo Estimation of n-Ball Content

**What lurks beyond the 4$^{th}$ dimension?**




A unit hypersphere with a ***fractional*** dimension has the maximum volume!

The volume ***decreases*** in each higher dimension!

# The Power Of Monte Carlo Integration

$$\begin{aligned}
\mathbf{F}^{(n)} &= \frac{\mu}{8\pi} \int_{r_1}^{r_2} \int_{s_1}^{s_2} \int_{y_1}^{y_2} \left( \frac{2}{R_a^3} + \frac{3a^2}{R_a^5} \right) \{ (\mathbf{R} \times \mathbf{b}) (\mathbf{t} \cdot \mathbf{n}) + \mathbf{t} [(\mathbf{R} \times \mathbf{b}) \cdot \mathbf{n}] \} \\
&\quad \times \left( \frac{r - r_1}{r_2 - r_1} \frac{s - s_1}{s_2 - s_1} \right) \mathrm{d}s \, \mathrm{d}r \, \mathrm{d}y \\
&- \frac{\mu}{4\pi (1 - \nu)} \int_{r_1}^{r_2} \int_{s_1}^{s_2} \int_{y_1}^{y_2} \left( \frac{1}{R_a^3} + \frac{3a^2}{R_a^5} \right) [(\mathbf{R} \times \mathbf{b}) \cdot \mathbf{t}] \, \mathbf{n} \left( \frac{r - r_1}{r_2 - r_1} \frac{s - s_1}{s_2 - s_1} \right) \mathrm{d}s \, \mathrm{d}r \, \mathrm{d}y \\
&+ \frac{\mu}{4\pi (1 - \nu)} \int_{r_1}^{r_2} \int_{s_1}^{s_2} \int_{y_1}^{y_2} \frac{1}{R_a^3} \{ (\mathbf{b} \times \mathbf{t}) (\mathbf{R} \cdot \mathbf{n}) + \mathbf{R} [(\mathbf{b} \times \mathbf{t}) \cdot \mathbf{n}] \} \left( \frac{r - r_1}{r_2 - r_1} \frac{s - s_1}{s_2 - s_1} \right) \mathrm{d}s \, \mathrm{d}r \, \mathrm{d}y \\
&- \frac{\mu}{4\pi (1 - \nu)} \int_{r_1}^{r_2} \int_{s_1}^{s_2} \int_{y_1}^{y_2} \frac{3}{R_a^5} [(\mathbf{R} \times \mathbf{b}) \cdot \mathbf{t}] (\mathbf{R} \cdot \mathbf{n}) \, \mathbf{R} \left( \frac{r - r_1}{r_2 - r_1} \frac{s - s_1}{s_2 - s_1} \right) \mathrm{d}s \, \mathrm{d}r \, \mathrm{d}y.
\end{aligned}$$

# More Monte Carlo Integration

We can use the principles of Monte Carlo sampling to estimate the area under **other types of curves**

1. We must determine which dots are "inside" (underneath) versus "outside" (above) the **curve**
2. We must determine the *bounds* (**area**) of the sample space
3. We need to determine the number of samples (dots) required to achieve the desired **accuracy**

**We don't know this area**

$$\frac{dots_{inside}}{dots_{total}} = \frac{area_{\boldsymbol{curve}}}{area_{sample}}$$

# The Quadrature of a Parabola

- Use the Monte Carlo method to estimate and display the area under the parabola $y = 4 - x^2$
- Pick $40{,}000$ random points within a sample area bounded by $-2 \leq x \leq 2$ and $0 < y \leq 5$
- Plot sampled points **under** the curve **red** and sample points **above** the curve **blue**
- From calculus, we know the exact area is **32/3**
- Print the actual area, the estimated area, and the absolute percentage error (APE) of the estimate

# Run mc_parabola.ipynb – **Cells 1…3**

**Import needed packages/modules**

```
[1]  # Cell 1
     import matplotlib.pyplot as plt
     import numpy as np
     from numba import int64, float64, vectorize
```

①

**Declare a numba accelerated function that computes the Halton QRNG**

1. The parameter $n$ is an integer of any size
2. The parameter $p$ is a prime number

```
[2]  # Cell 2
     @vectorize([float64(int64, int64)], nopython=True)
     def halton(n, p):
         h, f = 0, 1
         while n > 0:
             f = f / p
             h += (n % p) * f
             n = int(n / p)
         return h
```

②

**Set the number of random $dots$ (samples) to take**

```
[3]  # Cell 3
     total_dots = 40_600
```

③

# Run mc_parabola.ipynb – Cells 4…5

**Take $n$ "random" samples of 2D Cartesian points $(x, y)$ using the Halton sequence**

1. Scale the results so $-2 \leq x_{rng} \leq 2$ and $0 \leq y_{rng} \leq 5$
2. The sample area is thus $(-2...2) \times (0..5) = 20$ ①

```
[4]  # Cell 4
     x = (1 - halton(np.arange(total_dots), 2)) * 4 - 2
     y = (1 - halton(np.arange(total_dots), 3)) * 5
     print(x)
     print(y)
```
②

```
[ 2.          0.          1.          ... -0.64801025  0.35198975
 -1.64801025]
[5.         3.33333333 1.66666667 ... 2.02086403 0.35419736 4.61345662]
```
③

**Create an array $d$ containing $y_{rnd} - f(x_{rnd})$** ④

```
[5]  # Cell 5
     d = y - (4 - x**2)
     print(d)
```
⑤

```
[ 5.         -0.66666667 -1.33333333 ... -1.55921868 -3.52190586
  3.32939442]
```
⑥

# Run mc_parabola.ipynb – Cells 6...7

**Create arrays of $(x, y)$ coordinates that are "above" or "on or below" the parabola**

Here $f(x) = 4 - x^2$ so if $d > 0$ then the sample point is "above" the curve

```
[6]  # Cell 6
     x_in = x[d <= 0.0]
     y_in = y[d <= 0.0]
     x_out = x[d > 0.0]
     y_out = y[d > 0.0]
```

①

**Calculate the absolute percent error in the area estimation**

1. The actual/expected definite integral is $\frac{32}{3} = 10.6666...$ ②
2. The observed/estimated area using the Monte Carlo formulation $= 20 \times \frac{dots_{\ inside}}{dots_{\ total}}$ ③

```
[7]  # Cell 7
     act = 32 / 3
     est = 20 * np.count_nonzero(d <= 0.0) / total_dots
     err = np.abs((est - act) / act)

     print(f"dots = {total_dots:,}")
     print(f"act = {act:.6f}")
     print(f"est = {est:.6f}")
     print(f"err = {err:.5%}")
```

④

```
dots = 40,600
act = 10.666667
est = 10.670936
err = 0.04002%
```

⑤

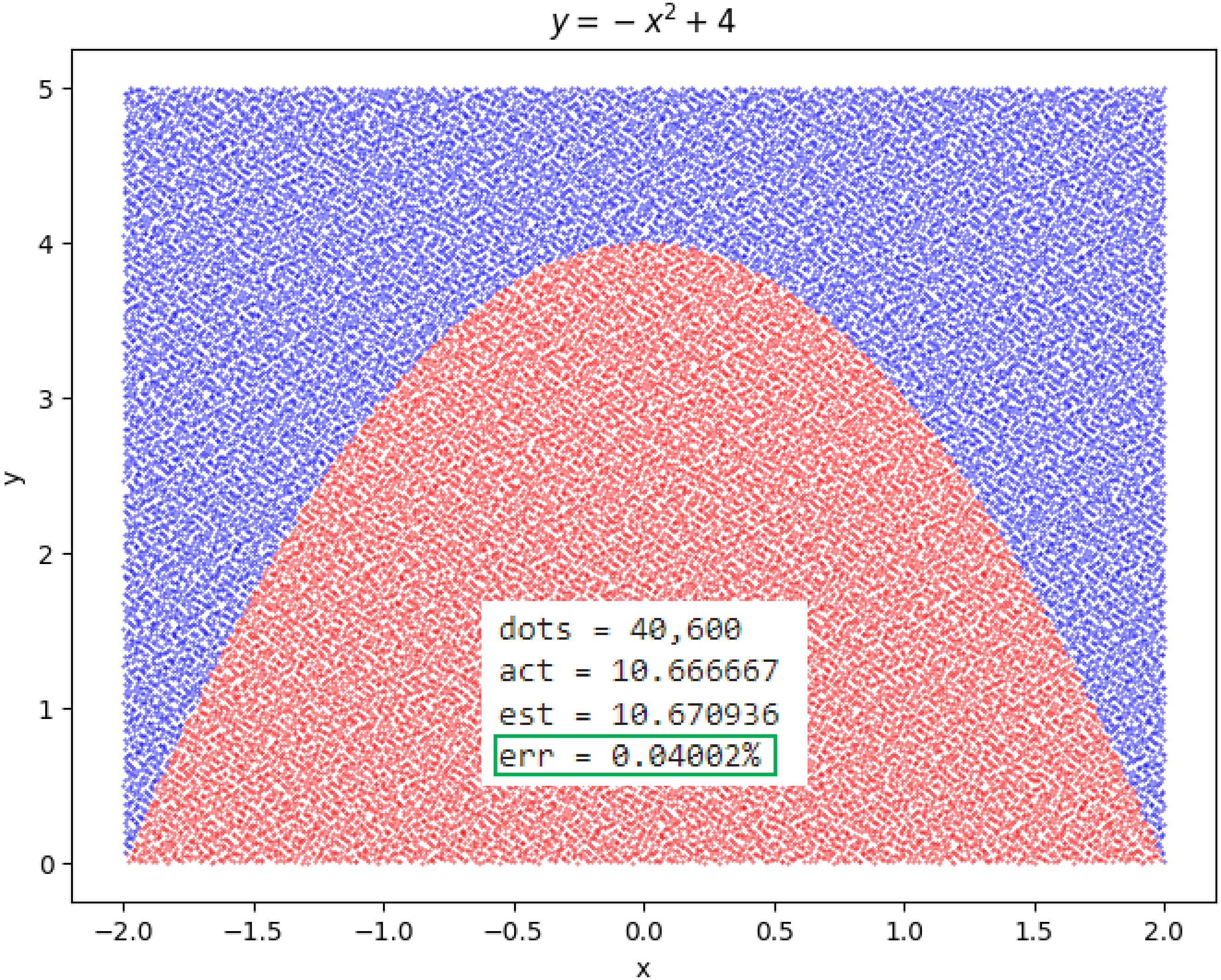
# Run mc_parabola.ipynb – Cells 8

**Display the scatter plot of the Monte Carlo estimation**

```
[8] # Cell 8
    plt.figure(figsize=(8, 6))
    plt.scatter(x_in, y_in, color="red", marker=".", s=0.5)
    plt.scatter(x_out, y_out, color="blue", marker=".", s=0.5)
    plt.title("$y=-x^2+4$")
    plt.xlabel("x")
    plt.ylabel("y")
    plt.show()
```

①

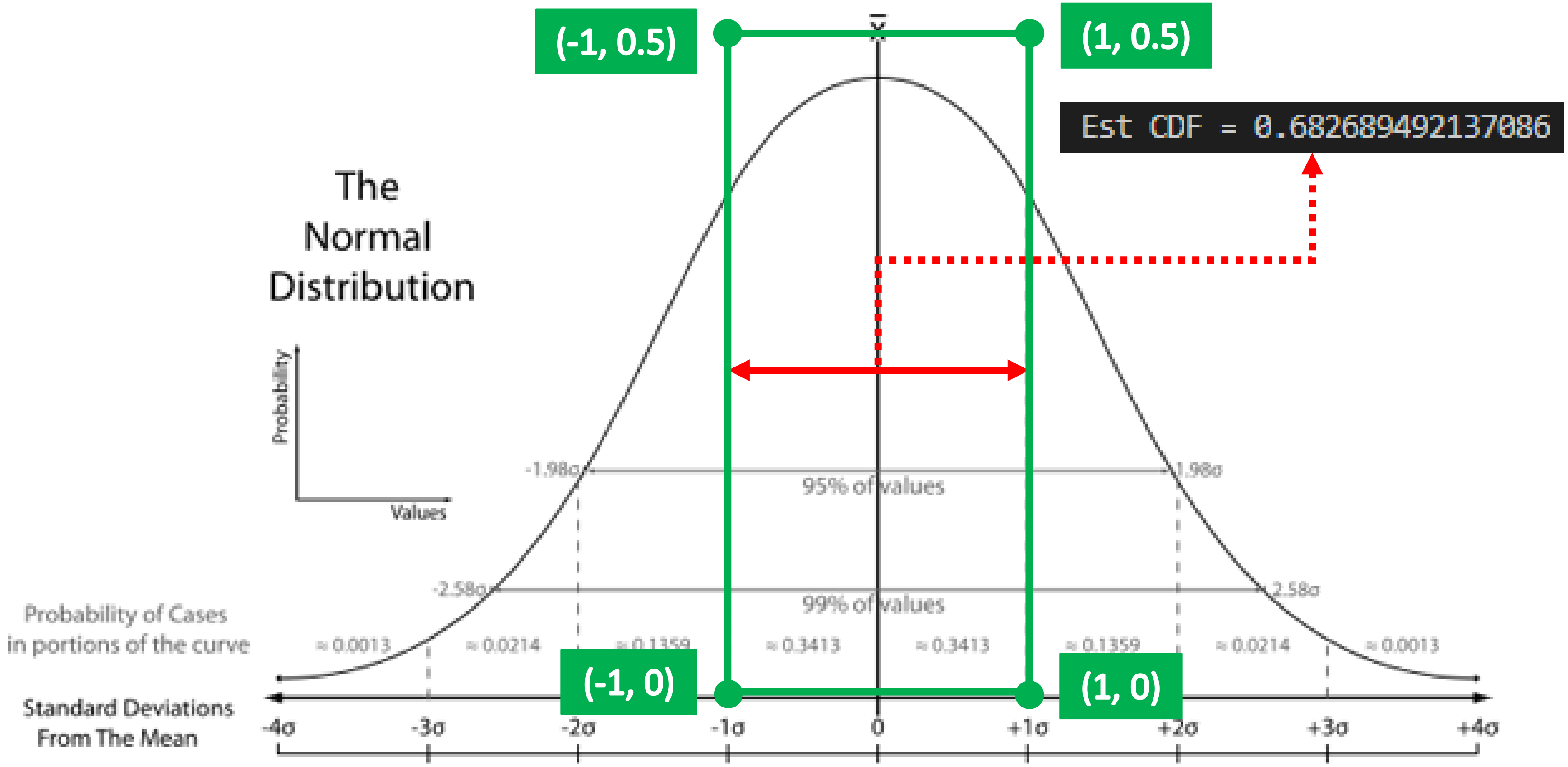
# **Check** mc_parabola.ipynb – **Cells 8**



# Cumulative Distribution Function

Estimate the probability that a normally distributed random variable will fall within $\pm$ the **first** standard deviation ($\sigma$) of its mean ($\mu$)

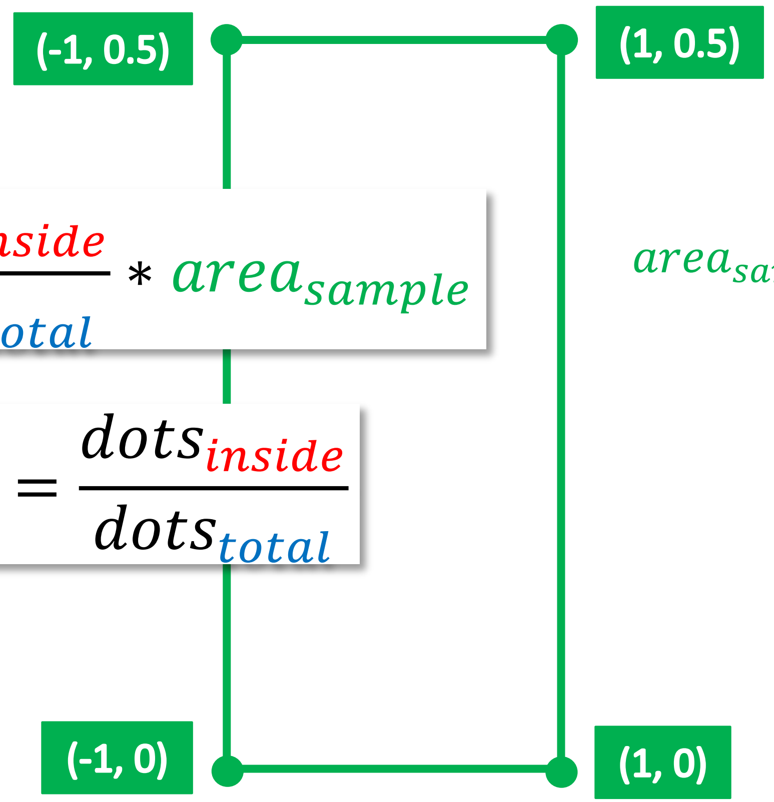
$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-(x-\mu)^2}{2\sigma^2}}$$



# Cumulative Distribution Function

Estimate the probability that a normally distributed random variable will fall within $\pm$ the first standard deviation ($\sigma$) of its mean ($\mu$)

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-(x-\mu)^2}{2\sigma^2}}$$




$$area_{\mathbf{curve}} = \frac{dots_{inside}}{dots_{total}} * area_{sample}$$

$$area_{sample} = 2 * 0.5 = 1$$

$$area_{\mathbf{std\ normal}} = \frac{dots_{inside}}{dots_{total}}$$

# **Run** mc_std_normal.ipynb – **Cells 1…3**

**Import needed packages/modules**

```
[1]  # Cell 1
     import matplotlib.pyplot as plt
     import numpy as np
     from numba import int64, float64, vectorize
```

①

**Declare a numba accelerated function that computes the Halton QRNG**

1. The parameter $n$ is an integer of any size
2. The parameter $p$ is a prime number

```
[2]  # Cell 2
     @vectorize([float64(int64, int64)], nopython=True)
     def halton(n, p):
         h, f = 0, 1
         while n > 0:
             f = f / p
             h += (n % p) * f
             n = int(n / p)
         return h
```

②

**Set the number of random $dots$ (samples) to take**

```
[3]  # Cell 3
     total_dots = 30_000
```

③

# **Run** mc_std_normal.ipynb – **Cells 4...5**

**Take $n$ "random" samples of 2D Cartesian points $(x, y)$ using the Halton sequence**

1. Scale the results so $-1 \leq x_{rng} \leq 1$ and $0 \leq y_{rng} \leq 0.5$
2. The sample area is thus $(-1...1) \times (0...\frac{1}{2}) = 1$

①

```
[4] # Cell 4
    x = (1 - halton(np.arange(total_dots), 2)) * 2.0 - 1.0
    y = (1 - halton(np.arange(total_dots), 3)) * 0.5
    print(x)
    print(y)
```

②

```
[ 1.          0.          0.5        ... -0.41156006  0.08843994
 -0.91156006]
[0.5        0.33333333 0.16666667 ... 0.49120222 0.32453556 0.15786889]
```

**Create an array $d$ containing $y_{rnd} - f(x_{rnd})$**

Here $f(x) \equiv$ the Gaussian Standard Normal PDF

```
[5] # Cell 5

    def f(x):
        # Standard Normal PDF
        return 1.0 / np.sqrt(2.0 * np.pi) * np.exp(-np.power(x, 2) / 2.0)

    d = y - f(x)
    print(d)
```

③ ④

```
[ 0.25802928 -0.06560895 -0.18539866 ...  0.12465553 -0.07284958
 -0.10544475]
```

# Run mc_std_normal.ipynb – **Cells 6...7**

**Create arrays of $(x, y)$ coordinates that are "above" or "on or below" the curve**

if $d > 0$ then the sample point is "above" the curve

```
[6]  # Cell 6
     x_in = x[d <= 0.0]
     y_in = y[d <= 0.0]
     x_out = x[d > 0.0]
     y_out = y[d > 0.0]
```

①

**Calculate the absolute percent error in the area estimation**

1. The actual/expected definite *non-analytic* integral is $0.682689492...$
2. The observed/estimated area using the Monte Carlo formulation $= 1 \times \frac{dots_{\ inside}}{dots_{\ total}}$

```
[7]  # Cell 7
     act = 0.682689492
     est = 1 * np.count_nonzero(d <= 0.0) / total_dots
     err = np.abs((est - act) / act)

     print(f"dots = {total_dots:,}")
     print(f"act = {act:.6f}")
     print(f"est = {est:.6f}")
     print(f"err = {err:.5%}")
```

②

```
dots = 30,000
act = 0.682689
est = 0.682667
err = 0.00334%
```

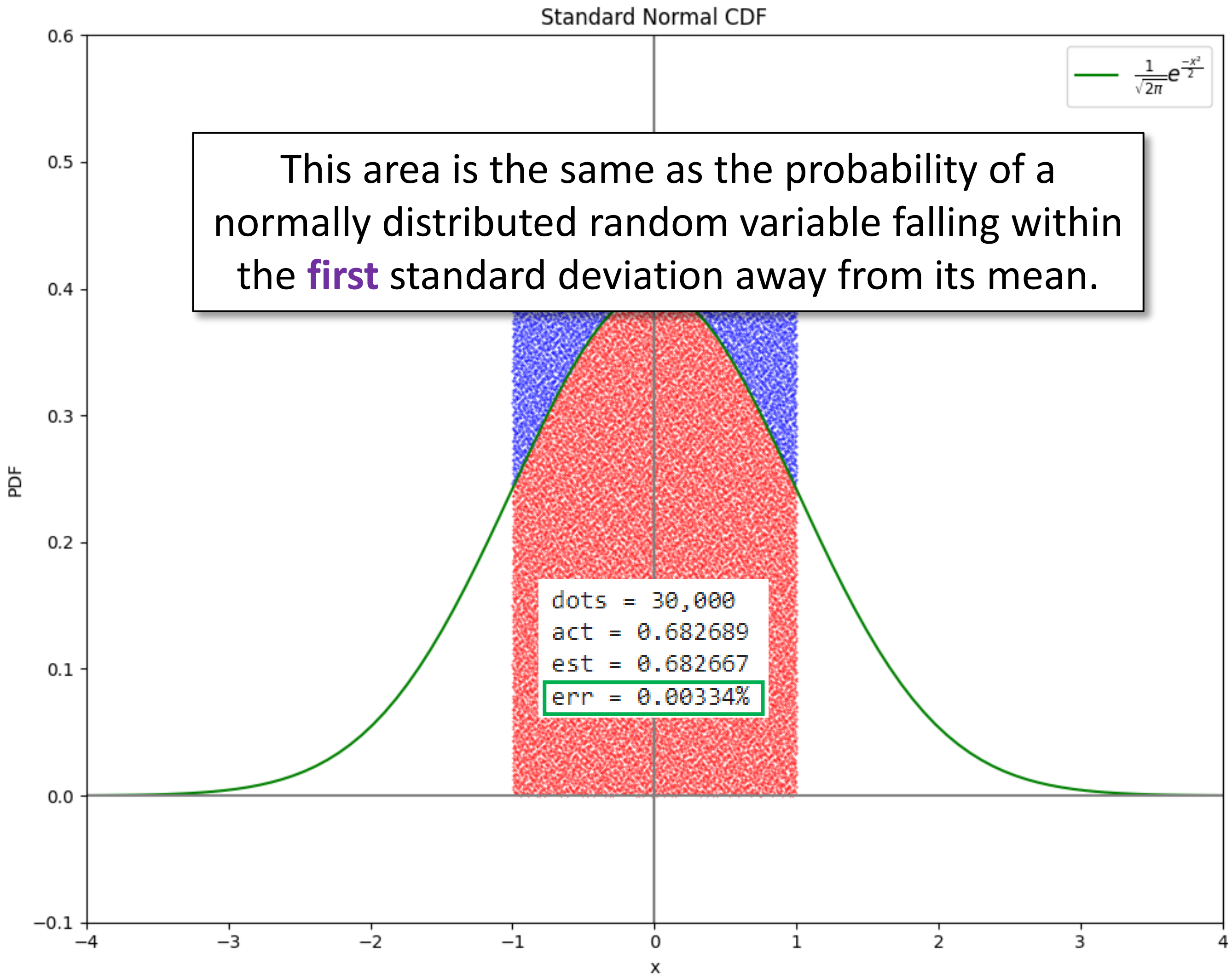
# Run mc_std_normal.ipynb – **Cell 8**

**Display the scatter plot of the Monte Carlo estimation**

Include a line graph of the Std Normal PDF to highlight the integrand

```
[8]  # Cell 8

     act_x = np.linspace(-4, 4, 100)        ①
     act_y = f(act_x)

     plt.figure(figsize=(10, 8))
     plt.scatter(x_in, y_in, color="red", marker=".", s=0.5)        ②
     plt.scatter(x_out, y_out, color="blue", marker=".", s=0.5)
     plt.plot(
         act_x, act_y, color="green",        ③
         label=r"$\frac{1}{\sqrt{2\pi}}e^{\frac{-x^2}{2}}$"
     )
     plt.title("Standard Normal CDF")
     plt.axhline(0, color="gray")
     plt.axvline(0, color="gray")
     plt.xlim(-4.0, 4.0)
     plt.ylim(-0.1, 0.6)
     plt.xlabel("x")
     plt.ylabel("PDF")
     plt.legend(loc="upper right", fontsize="12")
     plt.tight_layout()
     plt.show()
```

# **Check** mc_std_normal.ipynb – **Cell 8**



# Session **02** – Now You Know…

- Monte Carlo integration uses **random sampling**
  - The method calculates the ratio of the points below the curve to the total number of points – **the final ratio is the "area"**
  - It may require <u>billions of samples</u> to provide a reasonable estimate
  - It may be the *only way* to take the integral of a very complex function
- What you are taught cannot be the limit of your knowledge
  - The volume of a 4-D unit hypersphere $= \frac{\pi^2}{2}$
  - In infinite dimensions the volume of **all** hyperspheres is zero!
  - A *<u>fractional</u>* **5-dimensional** unit sphere has **maximum** volume
  - Mother Nature *never* said dimensions must be integers!