11. Monte Carlo Sampling

Dr. Hamish Carr

UNIVERSITY OF LEEDS

Random Generators

- Random numbers are fair and unpredictable
- Computer provides pseudo-random number
  - Functions in <stdlib.h>
  - random() returns 0 .. RAND_MAX
  - divide by RAND_MAX to get [0,1]
  - then scale & translate

Example

```
// generates a random value
GLfloat RandomRange(GLfloat min, GLfloat max)
     { // RandomRange()
     // compute range for scaling
     GLfloat range = max - min;
     // generate pseudorandom number in [0,RAND_MAX]
     GLfloat randomNumber = (GLfloat) random();
     // convert to [0,1]
     randomNumber /= (GLfloat) RAND_MAX;
     // scale
     randomNumber *= range;
     // translate
     randomNumber + min;
     // return result
     return randomNumber;
     } // RandomRange()
```

Expected Value

$$\bar{X} = E(X) = \sum_{s \in S} p(s) X(s)$$

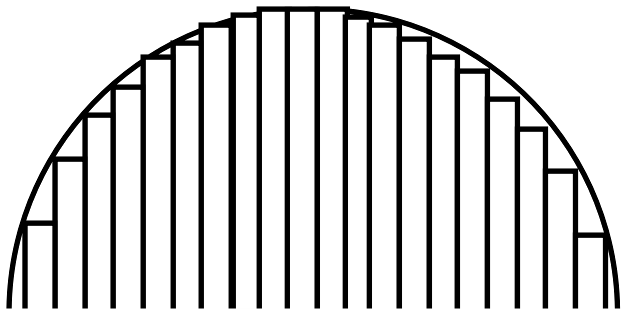
- Given a “random variable” X
- What value do we expect?
  - Multiply probability of value by value
  - Then sum over all values
- Also known as the mean or the average

Variance

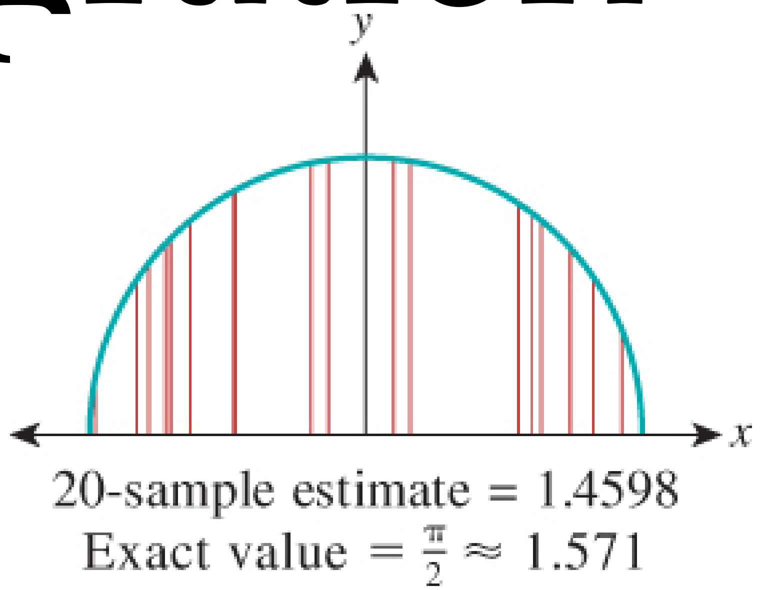
$$Var[X] = E[(X - \bar{X})^2]$$

- How tightly the value clusters
  - Take the difference from the mean
  - Square it & add them together
- It's square root is the standard deviation

Sampled Integration



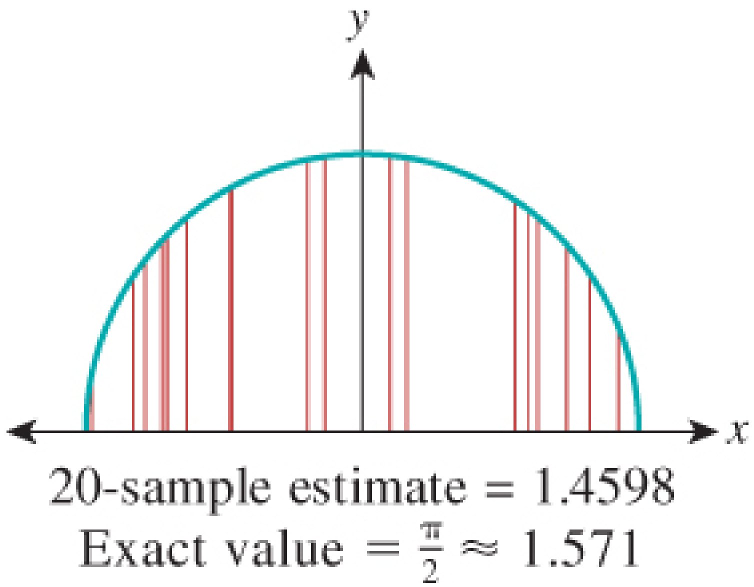
Regular Intervals



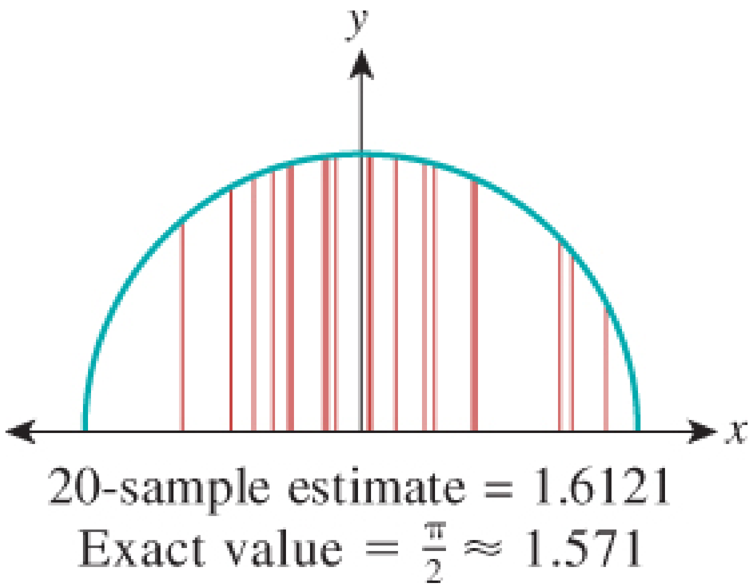
Uniform Sampling

- Instead of regular intervals for integration
- Use random samples
    - The *variance* of the estimate is better
    - I.e. it converges faster

Importance Sampling



Uniform Sampling



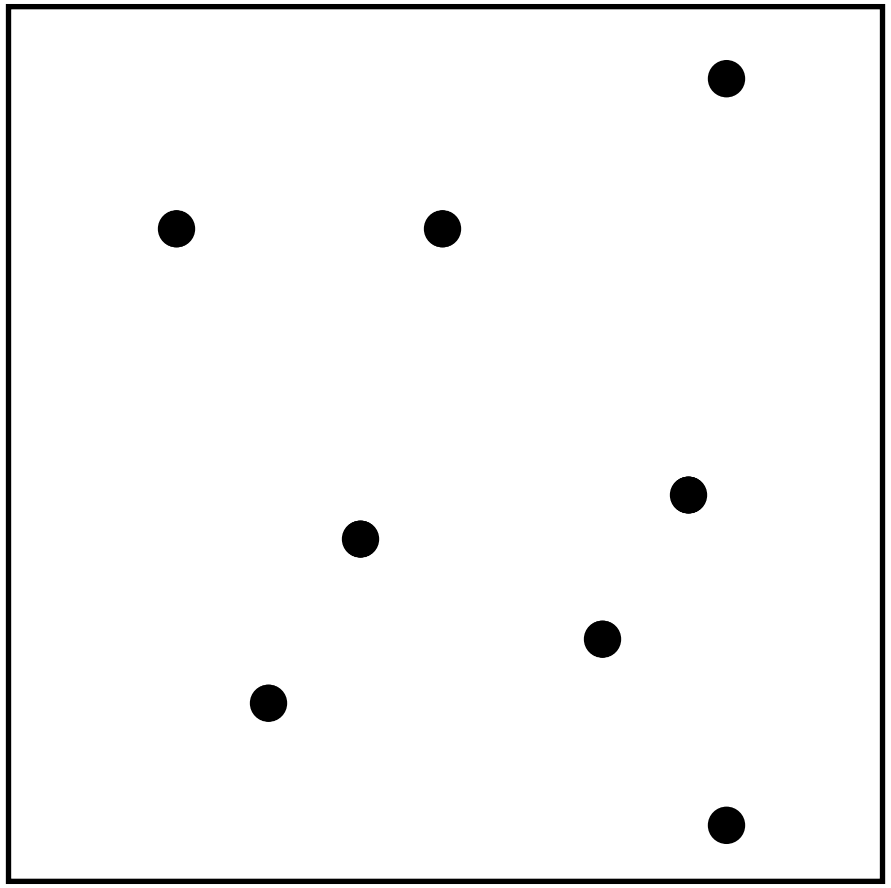
Importance Sampling

- Large values dominate the variance
  - i.e. they are more important
- So choose them more frequently
  - And weight samples for right result

Moving to 3D

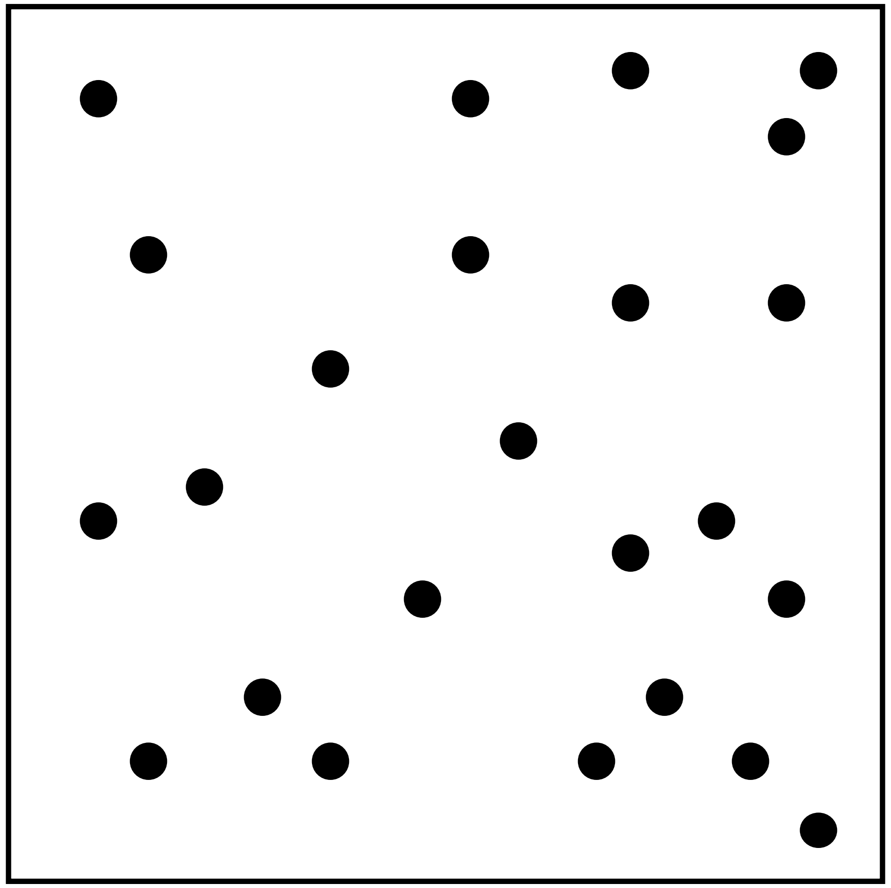
- This is easy for functions $f: \mathbb{R} \rightarrow \mathbb{R}$
- But gets harder for higher dimensions
- So let's look at another aspect of this:
  - Monte Carlo Sampling

Random (x,y) points



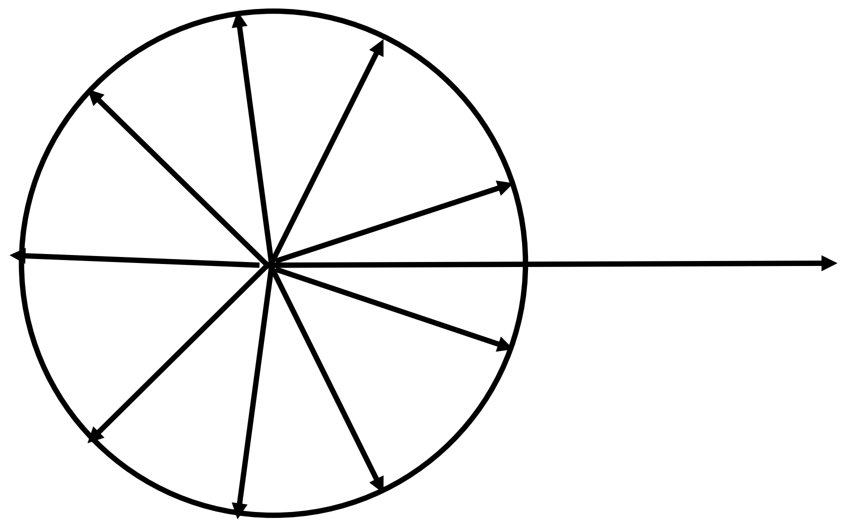
- Generate random x, random y

Directional Bias



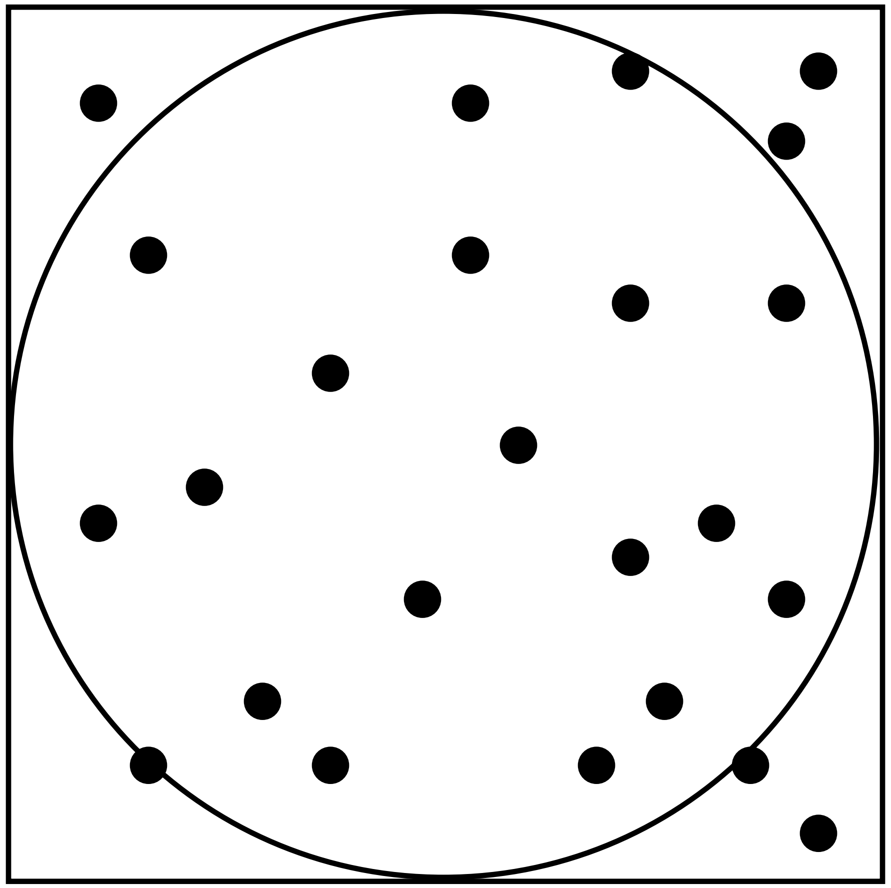
- Directions not equally likely
- Diagonals more likely

Circular Distribution



- All directions equally probable
- Could use random angle theta
- Breaks down in 3D (texture distortion)

Monte Carlo Method



- Generate points in a square
- *Discard* points outside the circle

Monte Carlo Code

```
Vector2D 2DMonteCarloVector()
  { // 2DMonteCarloVector()
  // loop until we get a valid one
  while (true)
    { // while loop
    // randomise x and y
    Vector2D aVector;
    aVector.x = RandomRange(-1.0, 1.0);
    aVector.y = RandomRange(-1.0, 1.0);
    // compute length & compare
    float length = aVector.Length();
    // return if it's good
    if (length <= 1.0)
      return aPoint;
    } // while loop
  // this should never be called - but it makes the compiler happy
  return Vector2D(0.0, 0.0);
  } // 2DMonteCarloVector()
```

Refinements

- Avoid taking the square root
- Discard points inside as well
  - avoids degenerate cases
- Normalize vectors to unit
  - Perfect for raytracing
  - Unless you want a vector not a direction

3D Monte Carlo Code

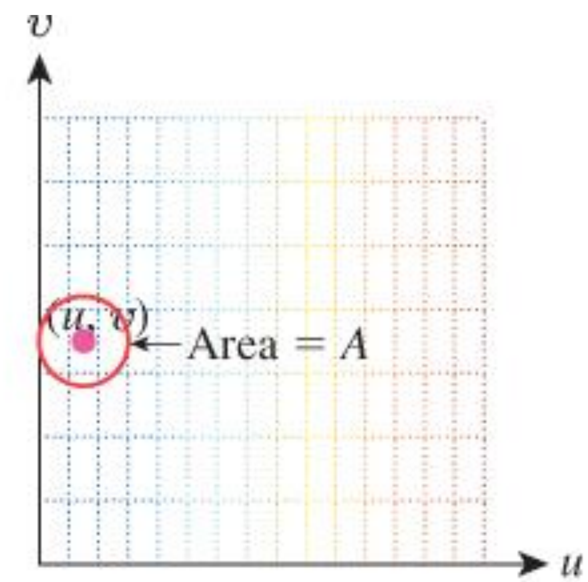
```
Vector3D 3DMonteCarloDirection()
  { // 3DMonteCarloDirection()
  // loop until we get a valid one
  while (true)
    { // while loop
    // randomise x, y, z
    Vector3D aVector;
    aVector.x = RandomRange(-1.0, 1.0);
    aVector.y = RandomRange(-1.0, 1.0);
    aVector.z = RandomRange(-1.0, 1.0);
    // compute length & compare
    float length = aVector.Length();
    // if the length is bad, do another loop
    if ((length > 1.0) || (length < 0.1))
    // otherwise, return the normalised version
    return aVector.UnitNormal();
    } // while loop
  // this should never be called - but it makes the compiler happy
  return Vector3D(0.0, 0.0, 0.0);
  } // 3DMonteCarloDirection()
```

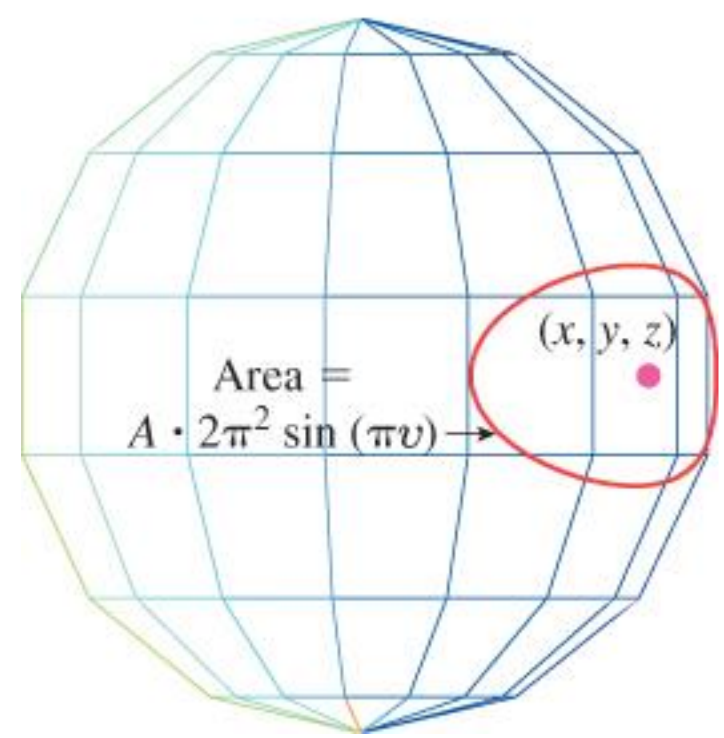
Downsides

- We're wasting samples
  - On the region outside the circle
- We want an efficient solution
  - That doesn't waste random numbers
- We re-parameterise the computation

Weighting Factor

- This turns out to be equal-area mapping
- An idea long understood in cartography
- Weight the sample by it's local distortion
- Known as the Jacobian
- Related to the slope





Random Hemisphere

- We weight the probability by this area
- It turns out to be a sine function
- And there's a short cut

$$f : [0,1] \times [0,1] \to H : (u, v) \mapsto (\cos(2\pi u), v, \sin(2\pi u)) \qquad (30.46)$$

Importance Sampling

f(x) is a function on [a,b]

X is a random variable with distribution g

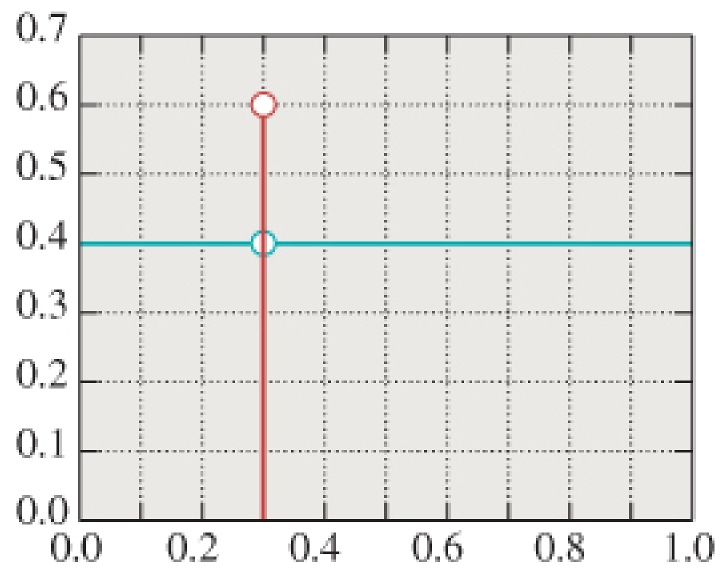
$\frac{f(X)}{g(X)}$ gives the expected value $\int_a^b f(x)dx$

- Large values have more influence on variance
- So we sample them more frequently
  - Increases contribution to the integral
  - So we counterweight their contribution
  - Net result: same mean, smaller variance

In practice

- Reflection involves estimating this integral:
  - $\int_{\omega \in \boldsymbol{S}^2_+(P)} L(R(P,\omega_i), -\omega_i) f_r(P,\omega_i,\omega_o)\omega_i \cdot \vec{n}(P) d\omega_i$
- We know f_r, but not L
- However, if f_r is small, the contribution is too
- Better, it's proportional to:
  - $f_r(P,\omega_i,\omega_o)\omega_i \cdot \vec{n}(P)$
  - The cosine weighted BRDF

Mixed Probabilities



- In practice, we need impulse or diffuse
- We divide our reflections between the two
  - e.g. 60% chance of an impulse
  - 40% chance of diffuse

Summary

- Monte Carlo sampling uses random numbers
- Cheaper and faster than uniform sampling
- Especially if we weight by importance
- And this means more complex maths
- Soluble for simple spherical distribution
- Other cases: generate a random vector
  - Then test BRDF to see if it's extinguished