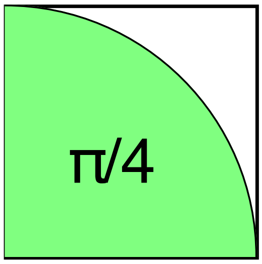
Monte Carlo Integration and Random Numbers

Higher dimensional integration

- Simpson rule with M evaluations in
  - one dimension the error is order M^{-4}
  - d dimensions the error is order $M^{-4/d}$
- In general an order-n scheme in one dimensions is order-n/d in d dimensions
- The phase space of physical N-body problems are usually very high-dimensional
  - classical mechanics: $d=6N$ (positions and velocities)
  - classical spin problem: $d=2N$ (two angles)
  - quantum spin-S problem: $d=(2S+1)^N$

Throwing stones into a pond

- How can we estimate the size of a pond with stones?
- How can we calculate π by throwing stones?
- Let us take a square surrounding the area we want to measure:



- Choose M random points and count how many lie in the interesting area
- Again we have a Mathematica notebook for this problem

Monte Carlo integration

- Consider an integral $$\langle f \rangle = \int_\Omega f(x)dx \Big/ \int_\Omega dx$$
- Instead of evaluating it at equally spaced points evaluate it at M points x_i chosen randomly in Ω:

$$\langle f \rangle \approx \frac{1}{M} \sum_{i=1}^{M} f(\vec{x}_i)$$

- This is a Monte Carlo estimate for the integral
- The error is statistical:

$$\Delta = \sqrt{\frac{\mathrm{Var}\, f}{M-1}} \propto M^{-1/2}$$

$$\mathrm{Var}\, f = \langle f^2 \rangle - \langle f \rangle^2$$

- In d>8 dimensions Monte Carlo is better than Simpson!

Importance sampling

- Simple sampling as discussed before is slow if the variance is big (function large in some regions, small in others)
- Then importance sampling is better. We choose points not uniformly but with probability p(x):

$$\langle f \rangle = \left\langle \frac{f}{p} \right\rangle_p := \int_\Omega \frac{f(x)}{p(x)} p(x) dx \Big/ \int_\Omega dx$$

- The error is now determined by the variance of *f/p*
- We want to choose *p* similar to *f* and such that p-distributed random numbers are easily available
- Example can also be found on the Mathematica file

$$f(x) = \exp(-x^2) \qquad p(x) = \exp(-x)$$

Random numbers

Random numbers

- Real random numbers are hard to obtain
  - cosmic radiation
  - atmospheric noise
  - used mainly for one-time encryption keys
  - New: Swss-made quantum random number generators
    - IDquantique in Geneva
- Pseudo random numbers
  - Get random numbers by an algorithm
  - generating random numbers algorithmically is a sin!
    - not random at all
    - completely deterministic
  - maybe they look random enough for our purposes (as long as the algorithm is not known)

Random numbers

- Real random numbers are hard to obtain
  - cosmic radiation
  - atmospheric noise
  - used mainly for one-time encryption keys
  - New: Swss-made quantum random number generators
    - IDquantique in Geneva
- Pseudo random numbers
  - Get random numbers by an algorithm
  - generating random numbers algorithmically is a sin!
    - not random at all
    - completely deterministic
  - maybe they look random enough for our purposes (as long as the algorithm is not known)

- Never trust pseudo random numbers however!

Linear congruential generators

- are of the simple form $x_{n+1}=f(x_n)$, with f usually a linear function
- A good choice is the GGL generator

$$x_{n+1} = (ax_n + c) \bmod m$$

  with $a = 16807$, $c = 0$, $m = 2^{31}-1$, $x_0=667790$
- quality depends sensitively on a,c,m and the seed value x_0

- Periodicity is a problem with such 32-bit generators
  - The sequence repeats identically after $2^{31}-1$ iterations
  - With modern computers that is just a few seconds!
  - Nowadays such 32-bit generators should not be used!

Lagged Fibonacci generators

- $$x_n = x_{n-p} \otimes x_{n-q} \bmod m$$
- Good choices for 64-bit floating point numbers (m=1)
  - (55,24,+)
  - (607,273,+)
  - (2281,1252,+)
  - (9689,5502,+)
  - (44497,23463,+)

- Seed blocks usually generated by linear congruential
- Has very long periods since large block of seeds
- no data dependencies for min(p,q) iterations
  - can be vectorized on vector CPUs
  - can be pipelined on scalar CPUs

Are these numbers really random?

Are these numbers really random?

- No!

Are these numbers really random?

- No!
- Are they random enough?
  - Maybe?

Are these numbers really random?

- No!
- Are they random enough?
  - Maybe?
- How can we test?
  - Statistical tests for distribution
  - Statistical tests for short time correlations
  - Statistical tests for long time correlations
  - …

Are these numbers really random?

- No!
- Are they random enough?
  - Maybe?
- How can we test?
  - Statistical tests for distribution
  - Statistical tests for short time correlations
  - Statistical tests for long time correlations
  - …
- Are these tests enough?
  - No! Your calculation could depend in a subtle way on hidden correlations!

Are these numbers really random?

- No!
- Are they random enough?
  - Maybe?
- How can we test?
  - Statistical tests for distribution
  - Statistical tests for short time correlations
  - Statistical tests for long time correlations
  - …
- Are these tests enough?
  - No! Your calculation could depend in a subtle way on hidden correlations!
- What is the ultimate test?
  - Run your simulation with various random number generators and compare the results

Easiest: graphical

- Before discussing statistical tests there is a simple first tool:
  - Create random pairs (x,y) and plot them
  - Create random triples (x,y,z) and plot them
- Can you see correlations?
- A Mathematica Notebook for these plots is on the web page of this course

Non-uniform random numbers

- we found ways to generate pseudo random numbers u in the interval $[0,1[$
- How do we get other uniform distributions?
  - uniform in $[a,b[$: $a+(b-a)\,u$
- Other distributions:
  - inversion of integrated distribution
  - acceptance-rejection method

The probability density function of a distribution

- The probability density function $p(x)$ Gives the probability of finding a number in an infinitesimal interval dx around x

- The probability of finding a number x in an interval $[a,b[$ is

$$P[a \leq x < b] = \int_a^b p(x)dx$$

- The integrated probability function $P(x)$ is the integral of $p(x)$

$$P(x) = \int_{-\infty}^{x} p(t)dt$$

Non-uniform distributions

- How can we get a random number x distributed with $f(x)$ in the interval $[a,b[$ from a uniform random number u?
- Look at probabilities:

$$P[x < y] = \int_a^y f(t)dt =: F(y) \equiv P[u < F(y)]$$

$$\Rightarrow u = F(x)$$

$$\Rightarrow x = F^{-1}(u)$$

- This method is feasible if the integral can be inverted easily
  - exponential distribution $f(x)=\lambda \exp(-\lambda x)$
  - can be obtained from uniform by $x=-1/\lambda \ln(1-u)$

Normally distributed numbers

- The normal distribution

$$f(x) = \frac{1}{\sqrt{2\pi}} \exp\left(-x^2 / 2\right)$$

  can be easily integrated in 2 dimensions

- We can obtain two normally distributed numbers from two uniform ones (Box-Muller method)

$$n_1 = \sqrt{-2\ln(1-u_1)}\,\sin u_2$$
$$n_2 = \sqrt{-2\ln(1-u_1)}\,\cos u_2$$

Uniform random numbers on *N*-sphere

- random points ***s*** on the surface of an *N*-sphere
- using acceptance-rejectance
  - get uniform random vector ***x*** with each component in [-1,1[
  - if norm is greater then one choose new one
  - normalize length to one

- using Box-Muller
  - start with uniform random vector ***x***
  - use Box-Muller to get normally distributed vector ***n***
  - normalize the length to one: the angles are uniformly distributed

- first method better only for very small *N*

Acceptance-rejection method

- If the integral of the distribution function *f* cannot be inverted easily
- Look for a simpler distribution *h* that bounds f:

$$f(x) < \lambda h(x)$$

- Repeat
  - Choose one h-distributed number *x*
  - Choose a uniform number *u*
- Until $u < f(x)/\lambda h(x)$
- Needs a good guess *h*
- Where that is not possible numerical inversion of integral might be faster!

The Boost random library

- Has been accepted into the next revision of the C++ standard
- For now get it from Boost: http://www.boost.org/
- It contains
  - Random number generators
  - Distribution functions

Generators in the Boost random library

- All generators have members such as:

```
class RNG {
  public:
    typedef … result_type;  // can be int, double,…
    RNG();

    void seed();  // the default seed
    template <class Iterator>
    Iterator seed(Iterator first, Iterator last);
    // seed from a range of unsigned int

    result_type min() const;
    result_type max() const;

    result_type operator();  // get the next random number
};
```

- They can be uniform floating point or integer generators with range between `min()` and `max()`

Useful and good generators

- `#include <boost/random.hpp>`

```
// Mersenne-twisters (modern, improved lagged Fibonacci
generators)
boost::mt11213b rng1;
boost::mt19937 rng2;

// standard lagged Fibonacci generators
boost::lagged_fibonacci607 rng3;
boost::lagged_fibonacci1279 rng4;
boost::lagged_fibonacci2281 rng5;

// linear congruential generators
boost::minstd_rand0 rng6;
boost::minstd_rand rng7;
```

- Read the documentation for more generators and details

Distributions in the Boost random library

- Uniform distributions
  - Integer: `boost::uniform_int<int> dist1(a,b)`
  - Floating point: `boost::uniform_real<double> dist2(a,b)`

- Exponential distribution

$$p(x) = \frac{1}{\lambda}\exp(-\lambda x)$$

  - `boost::exponential_distribution<double> dist3(lambda)`

- Normal distribution

$$f(x) = \frac{1}{\sqrt{2\pi}}\exp\left(-(x-\mu)^2 / 2\sigma^2\right)$$

  - `boost::normal_distribution<double> dist4(mu,sigma)`

- Read the documentation for more distributions and details

Combining generators with distributions

- Is done using `boost::variate_generator`

```
// define the distribution
boost::normal_distribution<double> dist(0.,1.);

// define the random number generator engine
boost::mt19937 engine;

// create a normally distributed generator
boost::variate_generator<boost::mt19937&,
  boost::normal_distribution<double> >
  rng(engine,dist);

// use it
 for (int i=0;i<100;++i)
  std::cout << rng() << "\n";
```

- Read the documentation for more details