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: $d=(2S+1)^N$
Random numbers

Throwing stones into a pond

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

\[
\frac{\pi}{4}
\]

- 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

\[
\langle f \rangle = \frac{\int_{W} f(x) \, dx}{\int_{W} dx}
\]

- Instead of evaluating it at equally spaced points evaluate it at \( M \) points \( x_i \), chosen randomly in \( W \):

\[
\langle f \rangle \approx \frac{1}{M} \sum_{i=1}^{M} f(x_i)
\]

- This is a Monte Carlo estimate for the integral

- The error is statistical:

\[
\Delta = \frac{\sqrt{\text{Var} f}}{\sqrt{M-1}} \approx M^{-1/2}
\]

\[
\text{Var} f = \langle f^2 \rangle - \langle f \rangle^2
\]

- In \( d > 8 \) dimensions Monte Carlo is better than Simpson!
**Sharply peaked functions**

- In many cases a function is large only in a tiny region
- Lots of time wasted in regions where the function is small
- The sampling error is large since the variance is large

---

**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 \( p(x) \):

\[
\left\langle f \right\rangle = \left( \frac{f}{p} \right)_p = \frac{\int f(x) p(x) dx}{\int p(x) dx} \Omega
\]

- 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

\[
f(x) = \exp(-x^2) \quad p(x) = \exp(-x)
\]
Generating random numbers

- Real random numbers are hard to obtain
  - classical or thermal chaos (atmospheric noise)
  - quantum mechanics
- Commercial products: quantum random number generators
  - based on photons and semi-transparent mirror
  - 4 Mbit/s from a USB device, too slow for most MC simulations

http://www.idquantique.com/

Pseudo Random numbers

- Are generated by an algorithm
- Not random at all, but completely deterministic
- Look nearly random however when algorithm is not known and may be good enough for our purposes
- 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

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

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**

- $x_n = x_{n-p} \otimes x_{n-q} \mod m$
- 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
More advanced generators

- As well-established generators fail new tests, better and better generators get developed

  - Mersenne twister (Matsumoto & Nishimura, 1997)
    http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html

  - Well generator (Panneton and L’Ecuyer, 2004)
    http://www.iro.umontreal.ca/~panneton/WELLRNG.html

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

Some simple RNG tests

- Graphical correlations test:
  - Create random n-tuplets \((x_1,x_2, ..., x_n)\) and plot them (see Mathematica notebook).

- Correlations test:
  \[
  \frac{1}{N} \sum_{i=1}^{N} x_ix_{i+n} = \langle x \rangle^2 + \mathcal{O}(N^{-1/2}) \quad \forall n
  \]

- Moments test:
  \[
  \left| \frac{1}{N} \sum_{i=1}^{N} x_i^k - \frac{1}{k-1} \right| \sim \mathcal{O}(N^{-1/2})
  \]

- Best known (free) testsuite: Diehard from George Marsaglia.
  See: http://stat.fsu.edu/pub/diehard/.
Marsaglia’s diehard tests

- Birthday spacings: Choose random points on a large interval. The spacings between the points should be asymptotically Poisson distributed. The name is based on the birthday paradox.
- Overlapping permutations: Analyze sequences of five consecutive random numbers. The 120 possible orderings should occur with statistically equal probability.
- Ranks of matrices: Select some number of bits from some number of random numbers to form a matrix over \( \{0,1\} \), then determine the rank of the matrix. Count the ranks.
- Monkey tests: Treat sequences of some number of bits as "words". Count the overlapping words in a stream. The number of "words" that don't appear should follow a known distribution. The name is based on the infinite monkey theorem.
- Count the 1s: Count the 1 bits in each of either successive or chosen bytes. Convert the counts to "letters", and count the occurrences of five-letter "words".
- Parking lot test: Randomly place unit circles in a 100 x 100 square. If the circle overlaps an existing one, try again. After 12,000 tries, the number of successfully "parked" circles should follow a certain normal distribution.

Marsaglia’s diehard tests (cont.)

- Minimum distance test: Randomly place 8,000 points in a 10,000 x 10,000 square, then find the minimum distance between the pairs. The square of this distance should be exponentially distributed with a certain mean.
- Random spheres test: Randomly choose 4,000 points in a cube of edge 1,000. Center a sphere on each point, whose radius is the minimum distance to another point. The smallest sphere's volume should be exponentially distributed with a certain mean.
- The squeeze test: Multiply 231 by random floats on \([0,1)\) until you reach 1. Repeat this 100,000 times. The number of floats needed to reach 1 should follow a certain distribution.
- Overlapping sums test: Generate a long sequence of random floats on \([0,1)\). Add sequences of 100 consecutive floats. The sums should be normally distributed with characteristic mean and sigma.
- Runs test: Generate a long sequence of random floats on \([0,1)\). Count ascending and descending runs. The counts should follow a certain distribution.
- The craps test: Play 200,000 games of craps, counting the wins and the number of throws per game. Each count should follow a certain distribution.
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:

\[
P[x < y] = \int_a^b f(x) \, dx = F(y) = P[u < F(y)]
\]

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

- 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 \( x = -\lambda \ln(1-u) \)

Normally distributed numbers

- The normal distribution

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

can be easily integrated in 2 dimensions

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

\[
n_1 = \sqrt{-2 \ln(1-u)} \sin u_2
\]
\[
n_2 = \sqrt{-2 \ln(1-u)} \cos u_2
\]
Random numbers

Uniform random numbers on $N$-sphere

- random points $s$ on the surface of an $N$-sphere
- using acceptance-rejection
  - get uniform random vector $x$ with each component in $[-1,1]$
  - if norm is greater than 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 $N$

Rejection method (von Neumann)

Look for a simple distribution $h$ that bounds $f$: $f(x) < \lambda h(x)$

- Choose an $h$-distributed number $x$
- Choose a uniform random number $0 \leq u < 1$
- Accept $x$ if $u < f(x) / \lambda h(x)$,
  otherwise reject $x$ and get a new pair $(x,u)$

- Needs a good guess $h$ to be efficient
The Boost random library

- Has become part of the C++03 standard and in a modified form in C++11

- For now get it from Boost: [http://www.boost.org/](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

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

  - 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`

```cpp
// 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() << 
```

◆ Read the documentation for more details