Centrum Wiskunde & Informatica

Monte Carlo methods

WISM454 Laboratory Class Scientific Computing, Jan-Willem Buurlage

March 11, 2019

Organization

Today

- If you wish, you can e-mail me a zip file with your source files (do not include anything other than *.hpp, *.cpp, CMakeLists.txt files).
- I can give you initial feedback on your code and library design (not for a grade). You can use this to improve your code before the report.
- Only today!

Report I

- *Deadline:* April 17th
- Content of report
  1. Theory.
     - A description of LCRNGs and the influence of parameters.
     - Distributions, inversion and rejection.
     - A discussion on statistical properties and tests.
     - Numerical integration, Monte Carlo.
  2. Software library. Describe your software library, with discussions on usage and design choices. After reading this section, the reader should be able to understand how to use every aspect of your library.
  3. Numerical results.

How it is graded

- You will receive an independent grade for the following:
  - Writing
  - Presentation
  - Experiments
  - Features
  - Code
- They all count for 20%.
- Scientific writing skills are important for any researcher. I expect the reports to be written with care. Try to avoid spending 99% of your time on your code and rushing the report.

Remarks on report

- Do not include your source code in the report, but instead e-mail me a zip file with your source files (do not include anything other than *.hpp, *.cpp, CMakeLists.txt files).
- Describe the overall structure of your code on a high-level. For example, discuss the different classes (such as `lcrng`) you have and how they relate to each other.
- All exercises listed in the schedule, except those marked hand-in or optional, are expected to be treated in your reports. Do not refer to them (or the lecture notes in general) explicitly, but rather make them part of the story.
- Provide benchmark results, and the results of statistical tests when run on the RNG engines you provide. Be economical with your plots and tables. Make sure that every result you include is indispensable for the story that you want to tell about the RNGs and your library. After reading this, the reader should be able to make an informed decision on which parts of your library they want to use for his application.

Monte Carlo Methods (I)

Numerical computation of integrals

- Input: integration domain $\Omega \subseteq \mathbb{R}^d$ and black box access to

$$f : \mathbb{R}^d \to \mathbb{R}$$

- Output: approximation of

$$\int_\Omega f(\mathbf{x}) d\mathbf{x}$$

Three strategies

Main idea:

$$\int_\Omega f(\mathbf{x})d\mathbf{x} \approx \sum_{i\in I} w_i f(\mathbf{x}_i)$$

How to find weights $w_i \in \mathbb{R}$ and integration points $\mathbf{x}_i \in \mathbb{R}^d$?

- Quadrature formulas ([LN] 3.1)
- Monte Carlo ([LN] 3.2)
- Low-discrepancy sampling ([LN] 3.3)

Quadrature formulas

- Assume for now:
  - 1-dimensional $f : \mathbb{R} \to \mathbb{R}$
  - twice continuously differentiable
- The trapezoidal rule is

$$\int_a^b f(x)dx \approx \frac{b-a}{2}(f(a)+f(b))$$

- With remainder (i.e. error)

$$R = -\frac{(b-a)^3}{12} f''(\eta) \qquad \text{for a } \eta \in [a, b]$$

Quadrature formulas

- Divide $[a, b]$ into k intervals of size $h = \frac{b-a}{k}$.
- The $k + 1$ integration points are now $\{a, a + h, a + 2h, ..., b\}$

$$\int_a^b f(x)dx \approx h\left(\frac{1}{2}f(a) + \left(\sum_{j=1}^{k-1} f(a + j \cdot h)\right) + \frac{1}{2}f(b)\right)$$

- Remainder satisfies

$$R = \frac{b-a}{12}h^2\left[f'(b) - f'(a)\right] + \mathcal{O}(h^3) \quad \text{for } h \to 0$$

and is bounded by

$$|R| \leq \frac{b-a}{12}h^2 \max_{x\in[a,b]} |f''(x)|$$

Accuracy

- For k intervals of size h

$$|R| \leq \frac{b-a}{12} h^2 \max_{x \in [a,b]} |f''(x)|$$

- For a fixed function f and fixed integration domain $[a,b]$ we have

$$|R| = \mathcal{O}(h^2) = \mathcal{O}\left(\frac{1}{k^2}\right)$$

- To get accuracy $|R| \leq \epsilon$, we require $k \propto \frac{1}{\sqrt{\epsilon}}$.

Higher dimensions

- For a function $f : \mathbb{R}^2 \to \mathbb{R}$ we want to compute

$$\int_{a_1}^{b_1} \int_{a_2}^{b_2} f(x,y)\, dy\, dx$$

- We can write this as

$$\int_{a_1}^{b_1} F(x)\, dx \quad \text{with} \quad F(x) = \int_{a_2}^{b_2} f(x,y)\, dy$$

- Define stepsizes $h_1 = \frac{b_1 - a_1}{k_1}$ and $h_2 = \frac{b_2 - a_2}{k_2}$
- To compute the x integral we have to compute

$$F(a_1), F(a_1 + h_1), ...$$

- Computing $F(x_i)$ is a one-dimensional integral (over y).

Accuracy in higher dimensions

- When taking k points in each direction, the error of a d-dimensional integral is
$$|R| = \mathcal{O}(h^2) = \mathcal{O}(\frac{1}{k^2})$$
The total number of points needed is $n = k^d$.
- To achieve error $|R| \leq \epsilon$ we need
$$n \propto \left(\frac{1}{\sqrt{\epsilon}}\right)^d$$
- Curse of dimensionality

Monte Carlo

- Hit-or-miss
- Simple sampling

You will implement both methods

Hit-or-miss

- Assume $f : [0,1] \to [0,1]$

$$I(f) = \int_0^1 f(x)\, dx$$

- Generate uniform random 'shots'

$$(x_1, y_1), (x_2, y_2), \dots$$

in $[0,1] \times [0,1]$.

- Count the number of shots that hit the area below the graph of f.

$$N_f(n) = \#\{i \leq n \mid y_i \leq f(x_i)\}$$

Approximate the integral by $N_f(n)/n$.

Hit-or-miss

- Define the random variable B_f on the probability space $[0,1] \times [0,1]$

$$B_f(x,y) = \begin{cases} 1 & y \leq f(x) \\ 0 & y > f(x) \end{cases}$$

  B_f is a Bernoulli variable

$$\mathbb{E}(B_f) = \mathbb{P}(B_f) = I(f)$$

- By the law of large numbers

$$\overline{B_n} \equiv \frac{1}{n}\sum_{i=1}^{n} b_i$$

  converges to $\mathbb{E}(B_f)$ as $n \to \infty$ where b_i are realisations of B_f.

Hit-or-miss accuracy

$$\mathbb{E}(B_f) = I(f)$$

$$\overline{B_n} \equiv \frac{1}{n}\sum_{i=1}^{n} b_i = \frac{N_f(n)}{n}$$

- The expected error:

$$\mathbb{E}(\ |\overline{B_n} - \mathbb{E}(B_f)|\ ) \leq \frac{1}{\sqrt{n}}\sqrt{\mathrm{Var}(B_f)}$$

- Bernoulli variables: $\mathrm{Var}(B_f) = \mathbb{P}(B_f)(1 - \mathbb{P}(B_f))$

$$\mathbb{E}\left(\left|\frac{N_f(n)}{n} - I(f)\right|\right) \leq \sqrt{\frac{I(f)(1 - I(f))}{n}} \leq \frac{1}{2\sqrt{n}}$$

Multi-dimensional hit-or-miss

- Assume $f : [0,1]^d \to [0,1]$
- Generate uniform random shots of the form $(x_1, ..., x_d, y)$.
- Count the number of shots for which $y < f(x_1, ..., x_d)$.
- What is the expected error?

Simple sampling

$$\int_a^b f(x)\, dx$$

- Let Q be a random variable that is uniform over $[a, b]$.
- The density function q of Q is constant: $q(x) = \frac{1}{b-a}$ for $x \in [a, b]$.
- Consider the random variable $f(Q)$ then

$$\int_a^b f(x)\, dx = (b-a)\ \mathbb{E}\left(f(Q)\right)$$

Simple sampling accuracy

- By the law of large numbers

$$\overline{f(Q)_n} = \frac{1}{n}\sum_{i=1}^{n} f(q_i) \;\to\; \mathbb{E}(f(Q))$$

  where q_i are realisations of Q.

- Expected error

$$\mathbb{E}\left(\left|\overline{f(Q)_n} - \mathbb{E}(f(Q))\right|\right) \le \frac{1}{\sqrt{n}}\sqrt{\mathrm{Var}(f(Q))}$$

- Estimate for $\mathrm{Var}(f(Q))$:

$$\frac{1}{n-1}\sum_{i=1}^{n}\left[f(q_i) - \overline{f(Q)_n}\right]^2$$

Multi-dimensional simple sampling

- Let Q be uniform over $\mathbf{J} = [a_1, b_1] \times ... \times [a_d, b_d]$

$$\int_{\mathbf{J}} f(\mathbf{x})\, d\mathbf{x} = (b_1 - a_1) \cdots (b_d - a_d)\, \mathbb{E}(f(Q))$$

- The error is bounded by

$$\frac{1}{\sqrt{n}} \sqrt{\mathrm{Var}(f(Q))}$$

Multi-dimensional simple sampling

What if the integration domain Ω is not a rectangle?

$$\int_{\Omega} f(\mathbf{x})\, d\mathbf{x} = \int_{a_1}^{b_1} \int_{a_2(x_1)}^{b_2(x_1)} \cdots \int_{a_d(x_1,\ldots,x_{d-1})}^{b_d(x_1,\ldots,x_{d-1})} f(\mathbf{x})\, dx_d \cdots dx_1$$

Use repeated 1-dimensional simple sampling, then

$$\int_{\Omega} f(\mathbf{x})\, d\mathbf{x} \approx \frac{1}{|\mathcal{I}|} \sum_{\mathbf{x}\in\mathcal{I}} w(\mathbf{x}) f(\mathbf{x})$$

Conclusions

- Quadrature methods
  - Trapezoidal rule
  - Error depends on $\frac{1}{k^2} \max_x |f''(x)|$
  - $n = k^d$ points in d dimensions
  - To achieve error ϵ we need

$$n \propto \left(\frac{1}{\epsilon}\right)^{d/2}$$

- Monte Carlo methods
  - Hit-or-miss
  - Simple sampling
  - To achieve error ϵ we need

$$n \propto \left(\frac{1}{\epsilon}\right)^{2}$$

C++

Functions as objects

- In C++ we can store functions using `std::function`.

```cpp
// Example
int myfunction(float x, int y) {
    return ...;
}

std::function<int(float,int)> f = myfunction;

int z1 = myfunction(3.14, 15);
int z2 = f(3.14, 15);

// General syntax
std::function<result_type()> g;
std::function<result_type(argument_type,...)> h;
```

Functions as objects

Why?

- Pass a function as an argument to another function
  - `integrate( f, ...  )`
- Store a function
  - Common examples are callbacks or event handlers.

```
void create_button(int x, int y,
                   std::function<void()> onclick);

void myfunction() {
    std::cout << "Button clicked!" << std::endl;
}

create_button(100, 200, myfunction);
```

Functions as objects

- You have seen other examples:
  - `std::accumulate`
  - `std::transform`
  - `std::generate`
  - `std::any_of`
  - ...

```
int square(int x) {
    return x * x;
}

// xs <- [1,2,3,4,5]
std::transform(xs.begin(), xs.end(), xs.begin(),
               square);
// xs <- [1,4,9,16,25]
```

Functions as objects

- Good for Monte Carlo integration

```
class mcintegrator {
    ...
    float integrate(std::function<float(float)> f,
                    float a, float b, int n) {
        return ...;
    }
    ...
    // f : T -> T
    template <typename T>
    T integrate(std::function<T(T)> f,
                T a, T b, int n) {
        return ...;
    }
};
```

Functions as objects

- Function object can be empty

```
int myfunction() { return 3; }

std::function<int()> f;
std::function<int()> g = myfunction;

if (f)
    f(); // not called

if (g)
    g(); // called

g = nullptr;

if (g)
    g(); // not called
```

Functions as objects

- std::function can store any callable object

```
class myclass {
  public:
    int operator()(int x) {
        return z + x;
    }
    int z;
};

myclass a;
a.z = 5;
int y = a(3); // y <- 8

std::function<int(int)> f = a;
int z = f(4);
```

Anonymous functions

- Anonymous functions, also known as lambda functions

```
int square(int x) {
    return x * x;
}
// xs <- [1,2,3,4,5]
std::transform(xs.begin(), xs.end(), xs.begin(),
               square);
// xs <- [1,4,9,16,25]
std::transform(xs.begin(), xs.end(), xs.begin(),
               [](int x) { return x+1; } );
// xs <- [2,5,10,17,26]
std::function<int(int)> f = square;
std::function<int(int)> g = [](int x) { return x+1; };
```

Lambda function syntax

```
// simple version
[] (parameters) { body }

// (almost) full version
[captures] (parameters) -> return_type { body }
```

Capturing variables

```
std::vector<int> xs = {1,2,3,4,5};
int a = 3;

// Does *not* compile!
std::transform(xs.begin(), xs.end(), xs.begin(),
               [](int x) { return x+a; } );

// This works
std::transform(xs.begin(), xs.end(), xs.begin(),
               [a](int x) { return x+a; } );
```

- The variable a is captured by the lambda function

Capturing variables

```
std::vector<int> xs(100, 0);
int a = 5, b = 0, c = 0;

// xs <- {0, 0, ..., 0}
std::generate(xs.begin(), xs.end(),
            [a, &b]() {
                b++;
                return a + b;
            } );
// xs <- {6, 7, ..., 105}
// a  <- 5
// b  <- 100
// c  <- 0
```

- The variable `a` is captured by value
- The variable `b` is captured by reference
- The variable `c` is not captured

Capturing variables

```
std::vector<int> xs(100, 0);
int a = 5, b = 0, c = 0;

// xs <- {0, 0, ..., 0}
std::generate(xs.begin(), xs.end(),
              [&]() {
                  b++;
                  return a + b;
              } );
// xs <- {6, 7, ..., 105}
// a  <- 5
// b  <- 100
// c  <- 0
```

- The variable a is captured by reference
- The variable b is captured by reference
- The variable c is not captured

Capturing variables

- `[a]` capture a by value
- `[&a]` capture a by reference
- `[&]` captures all variables used in the lambda by reference
- `[=]` captures all variables used in the lambda by value
- `[&, a]` captures variables like with `[&]`, but a by value
- `[=, &a]` captures variables like with `[=]`, but a by reference

Capturing variables

```cpp
int a = 5, b = 2, c = 0;
std::function<int()> f = [&, a]() { b++; return a + b; };
// a <- 5 , b <- 2 , c <- 0
c = f();
```

Capturing variables

```
int a = 5, b = 2, c = 0;
std::function<int()> f = [&, a]() { b++; return a + b; };
// a <- 5 , b <- 2 , c <- 0
c = f();

// a <- 5  , b <- 3 , c <- 8
c = f();
a = 20;
```

Capturing variables

```
int a = 5, b = 2, c = 0;
std::function<int()> f = [&, a]() { b++; return a + b; };
// a <- 5 , b <- 2 , c <- 0
c = f();

// a <- 5  , b <- 3 , c <- 8
c = f();
a = 20;

// a <- 20 , b <- 4 , c <- 9
c = f();
b = 100;
```

Capturing variables

```
int a = 5, b = 2, c = 0;
std::function<int()> f = [&, a]() { b++; return a + b; };
// a <- 5 , b <- 2 , c <- 0
c = f();

// a <- 5  , b <- 3 , c <- 8
c = f();
a = 20;

// a <- 20 , b <- 4 , c <- 9
c = f();
b = 100;

// a <- 20 , b <- 100 , c <- 10
c = f();
```

Capturing variables

```
int a = 5, b = 2, c = 0;
std::function<int()> f = [&, a]() { b++; return a + b; };
// a <- 5 , b <- 2 , c <- 0
c = f();

// a <- 5  , b <- 3 , c <- 8
c = f();
a = 20;

// a <- 20 , b <- 4 , c <- 9
c = f();
b = 100;

// a <- 20 , b <- 100 , c <- 10
c = f();

// a <- 20 , b <- 101 , c <- 106
```

Comparison with ‘old C++’

```
int myfunction(float x, int y) { return 3; }

int (*oldf)(float,int)                    = myfunction; // Old
std::function<int(float,int)> newf = myfunction; // New

int a = *oldf(3.1, 4);
int b =  newf(3.1, 4);

// Old
int func1( int (*f)(float,int) ) {...}
// New
int func2( std::function<int(float,int)> f ) {...}
```

- Old function pointers can not store arbitrary callable objects
- They can store lambda functions but only without captures

Values and references

- `std::function` object can store data so passing by reference makes sense
- When used with lambdas, passing by value makes sense because of move semantics

Smart pointers

- Regular pointers
- Unique pointers
- Shared pointers

Regular pointers

- Pointers can be used for objects on the heap

```
int* x = new int;

*x = 5;

delete x;
```

Regular pointers

- Pointers are “dangerous”

```
int myfunction (...) {
    int* x = new int;
    *x = 3;
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    delete x;
    return result;
}
```

Regular pointers

- Pointers are “dangerous”

```
int myfunction (...) {
    int* x = new int;
    *x = 3;
    ...
    ...
    if (error) {
        return 0;
    }
    ...
    ...
    ...
    delete x;
    return result;
}
```

Unique pointers

- `std::unique_ptr` is a smart pointer
- Takes care of deleting the object at the right time

```
int myfunction (...) {
    std::unique_ptr<int> x = std::make_unique<int>();
    *x = 3;
    ...
    ...
    if (error) {
        return 0;
    }
    ...
    ...
    return result;
}
```

Unique pointers

- std::unique_ptr takes care of ownership

```
int myfunction (std::unique_ptr<int> x) {
    ...
}

std::unique_ptr<int> a = std::make_unique<int>();
*a = 3;
// Now we pass the ownership to myfunction
myfunction(std::move(a));

// Here a is no longer valid
if (a)
    std::cout << "a is valid" << std::endl;
else
    std::cout << "a is not valid" << std::endl;
```

Shared pointers

- You can not pass `std::unique_ptr` to different functions
- For this we have `std::shared_ptr`

```
class rng;   // base class in your library
class lcrng; // subclass in your library

// Create random number generator
std::shared_ptr<rng> park_miller =
        std::make_shared<lcrng>(16807, ...);

int x = park_miller->next(); // Use the shared pointer

// Pass the shared pointer to other functions
output_random_numbers(park_miller);
// Use it to create your Monte Carlo class
mcintegrator mc(park_miller);
mc.integrate(myfunction);
```

Conclusion

- `std::function`
  - store functions
  - functions as arguments
- Lambda functions
  - easy way of passing small functions to other functions
  - captures
- Smart pointers
  - safe way of dealing with pointers