Centrum Wiskunde & Informatica

# Low discrepancy sampling

WISM454 Laboratory Class Scientific Computing, Jan-Willem Buurlage

March 18, 2019

## Numerical integration

## Recap of last week

- 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 $\epsilon$ we need

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

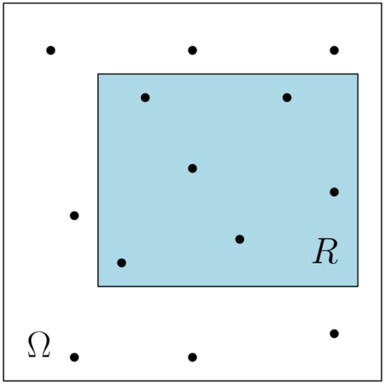
- Monte Carlo methods
  - Hit-or-miss
  - Simple sampling
  - To achieve error $\epsilon$ we need

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

## Discrepancy

- With Monte Carlo methods, sample points are selected randomly, is this optimal?
- Intuitively, the discrepancy of a sequence is a measure of the gaps that a sequence leaves
- Sampling for low discrepancy is the subject of today

## Discrepancy



- We estimate the area of $R$ by hit-or-miss sampling with sequence of points

## Discrepancy definition

- Let $\Omega = [0,1]^d$. For some sampling sequence $\{\mathbf{x}_j\}$, what is the largest error in estimating rectangular volumes?
- $R = [a_1, b_1] \times \ldots \times [a_d, b_d]$, volume is

$$V(R) = \prod_{i=1}^{d} (b_i - a_i).$$

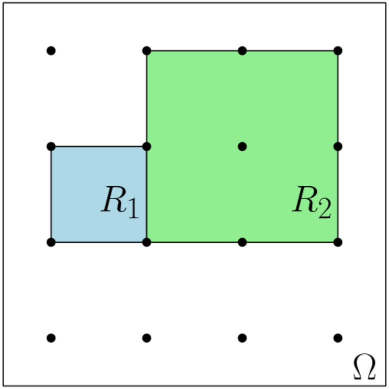
- Simple sampling with first $n$ elements of the sequence gives:

$$\tilde{V}_n(R) = |\{j \leq n \,|\, \mathbf{x}_j \in R\}|.$$

- Discrepency $D$ defined as

$$D_n = \sup_{\text{rectangles } R} |\tilde{V}_n(R) - V(R)|.$$

## Discrepancy Example (uniform)



- $V(R_1) = \frac{1}{16}$, $V(R_2) = \frac{1}{4}$, $\tilde{V}(R_1) = 0$, $\tilde{V}(R_2) = \frac{1}{16}$.

## Discrepency for first $n$ points

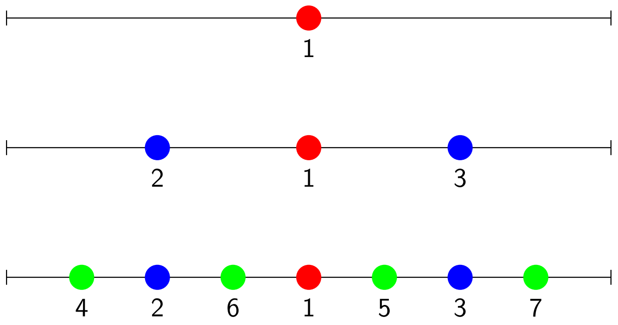
- We want a sequence that has low discrepency for all $n$
- Instead of a random sequence, we can start with something uniform, and then start filling in the gaps
- There are various deterministic sequences that obtain low discrepancy

## Van der Corput sequence

- Exercise 2.13
- $\pi(b_{n-1} \dots b_0) = 0.b_0 b_1 \dots b_{n-1}$.
- The sequence $\{\pi(1), \pi(2), \pi(3), \dots\}$ is the van der Corput sequence.
- Example of a deterministic uniform distribution
- This coincides with the 'uniform distribution then fill up gaps' for $d = 1$!

## Example van der Corput sequence

- First elements are $\frac{1}{2}, \frac{1}{4}, \frac{3}{4}, \frac{1}{8}, \frac{5}{8}, \frac{3}{8}, \frac{7}{8}, \ldots$



## Sampling for low discrepancy

- We want to extend this idea to $d > 1$.
- Prime number $p$, base-$p$ expansions. Change of notation:

$$\pi_2\big((b_{n-1}\dots b_0)_2\big) = (0.b_0 b_1 \dots b_{n-1})_2.$$

- This is for binary representation, but we can do this for arbitrary base $p$:

$$\pi_p\big((a_{n-1}\dots a_0)_p\big) = (0.a_0 a_1 \dots a_{n-1})_p.$$

- More explicitely:

$$\pi_p\left(\sum_{i=0}^{n-1} a_i p^i\right) = \sum_{i=0}^{n-1} a_i p^{-i-1}.$$

## Halton sequence

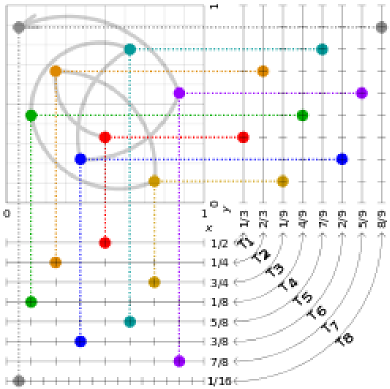
- Let $p_1, \ldots, p_d$ be the first $d$ primes (i.e. $2, 3, 5, 7, 11, \ldots$).
- Halton sequence is:

$$\mathbf{x}_j = \left(\pi_{p_1}(j), \pi_{p_2}(j), \ldots, \pi_{p_d}(j)\right)^T.$$

- Note that this is different from the 'uniform then fill gaps' idea!

## Example Halton sequence

$$\left\{\left(\frac{1}{2},\frac{1}{3}\right),\left(\frac{1}{4},\frac{2}{3}\right),\left(\frac{3}{4},\frac{1}{9}\right),\left(\frac{1}{8},\frac{4}{9}\right),\left(\frac{5}{8},\frac{7}{9}\right),\left(\frac{3}{8},\frac{2}{9}\right),\left(\frac{7}{8},\frac{5}{9}\right),\left(\frac{1}{16},\frac{8}{9}\right),\left(\frac{9}{16},\frac{1}{27}\right),\ldots\right\}$$



## Halton discrepancy

- As we have seen, for Monte Carlo the (expected) error (and discrepency) is of

$$\mathcal{O}\left(\frac{1}{\sqrt{n}}\right).$$

- For Halton we instead have (deterministically)

$$\mathcal{O}\left(\frac{\log^d(n)}{n}\right).$$

- This is almost a quadratic improvement!

## Exercise 3.8

- Implement the Halton sequence in *d*-dimensions:
  - How does this tie into your RNG code?
- Find the volume of the *d*-dimensional sphere using
  1. Random sequence
  2. Halton sequence
- Plot the error for both methods

# C++

## Copy-versus-move

- Although potentially expensive, making copies is sometimes unavoidable.

```
std::string id(std::string x) {
    return x;
}

auto s = std::string("Lorem ipsum"); // construct
s = id(s);                           // copy s into id
                                     // copy(?) back to s
auto u = std::string("Sit amet");    // construct
u = s;                               // copy s into u
```

## Copy constructors (1) and copy assignment (2)

```
class X {
    ...
    X(const X& other) : ... { ... }    // (1)
    X& operator=(const X& other) ...;  // (2)
};

X a;
X b(a);     // (1)
auto b = a; // (2)
```

## Example of copy constructor

```
class List {
    ...
    List(const List& other) : xs_(other.xs_) {}
    // ... calls copy constructor of std::vector<T>!

  private:
    std::vector<T> xs_;
};
```

## Moves

- Sometimes copies can be avoided (perhaps because the original is no longer needed).

```
std::string id(std::string x) {
    return x;
    // x is no longer used here...
}
```

- This is indicated using a so-called rvalue reference `T&&`. Such references are free to move from, meaning that it is OK to steal their resources and leave them empty.

## std::move

- An rvalue reference can be created using std::move.

```
auto xs = std::vector<int>(10000000);
auto ys = std::move(xs);
```

- std::vector instances hold (a pointer to) chunk of heap memory
- ys = xs will copy this chunk of memory to ys, leaving two copies
- ys = std::move(xs) sets pointer of ys to xs resource, and e.g. sets xs resource to 'nullptr'. No copy!

## Move constructors

```
class List {
    ...
    List(List&& other) : xs_(std::move(other.xs_)) {}
    // ... calls move constructor of std::vector<T>!

  private:
    std::vector<T> xs_;
};
```

## Copy versus move

```
std::vector<T>(const std::vector<T>& other) {
    this->resize(other.size);
    std::copy(other.begin(), other.end(), this->begin());
}

std::vector<T>(std::vector<T>&& other) {
    this->data_ = other.data_;
    this->size_ = other.size_;
    other.data_ = nullptr;
    other.size_ = 0;
}
```

## Overloading

- It is allowed in C++ (but not C) to have the same name for functions with different arguments.

```
int f(int x);
float f(float x);
int f(int x, float y);
float f(int x, float y); // ... ERROR!
```

## Operators

- Function overloading is especially useful for operators.

```
struct complex {
    complex(double re_, double im_) : re(re_), im(im_) {}
    double re;
    double im;
};
```

- For `complex` values `x, y` we want to be able to write:

```
x + y; x += y; x * y;
```

## Operator overloading

```cpp
complex operator+(complex alpha, complex beta) {
    complex gamma;
    gamma.re = alpha.re + beta.re;
    gamma.im = alpha.im + beta.im;
    return gamma;
}

// shorter...
complex operator+(complex alpha, complex beta) {
    return {alpha.re + beta.re, alpha.im + beta.im};
}
```

## Operator overloading (II)

- Operators can also be member functions

```
struct complex {
    ...
    void operator+=(complex other) {
        re += other.re;
        im += other.im;
    }

    complex operator-() {
        return {-re, -im};
    }
};
```

- Up to taste. I typically write `..=` and unary ops as member functions, and other ops as non-member functions.

## Operator overloading (III)

- Operators give a lot of freedom

```
// add a double to a complex
complex operator+(complex alpha, double x) {
    return {alpha.re + x, alpha.im};
}

// multiply with a scalar
complex operator*(double x, complex alpha) {
    return {x * alpha.re, x * alpha.im};
}
```

- Unfortunately, a lot of repetition is (currently) unavoidable in C++ when building complete numeric types.

## User-defined literals

- You can 'invent your own language' (DSL) by using user-defined literals.

```
constexpr complex operator ""i(double x) {
    return {0, x};
}

auto x = 3.0 + 4.0i;
```

- I use this for annotating e.g. dimensions, units, ...

```
auto h = convolve<3_D>(f, g);
```

## Conclusion

- Copying can sometimes be avoided
- Move semantics rely on rvalue references `T&&`
  - Copy constructors
  - Move constructors
  - Cast using `std::move`.
- Overloading and operators lead to generic and readable code
  - Unary and binary operations
  - Choice between non-member or member function
  - User defined literals can make code more readable

## Numerical integration library

- Required features of your numerical integration library:
  - quadrature formula
  - MC hit-or-miss
  - MC simple sampling
  - low-discrepency sampling
- All in higher dimensions as well!
- Should work for a 'black-box' `std::function<T(T...)>`, with a RNG generator of choice (from your RNG library).
- Gather information about the performance in some intermediate format (e.g. CSV, binary, ...). Plot using application of your choice (MATLAB, matplotlib, ...)

## Example

```cpp
auto f = std::function([](double x)
    { return sqrt(1 - x * x); });
auto x = integrate_trapezoid(f, a, b, steps);
auto y = integrate_mc_hitmiss(f, rng, a, b, samples);
auto z = integrate_mc_sampling(f, rng, a, b, samples);

// so e.g.
template <typename T>
T integrate_mc_sampling(std::function<T(T)> f,
    lcsc::rng_engine<uint32_t>& gen, T a, T b,
    uint32_t samples = 100);
```