Random Number Generation
A Practitioner's Overview

Prof. Michael Mascagni

Department of Computer Science
Department of Mathematics
Department of Scientific Computing
Graduate Program in Molecular Biophysics
Florida State University, Tallahassee, FL 32306 **USA**
AND
Applied and Computational Mathematics Division, ITL
National Institute of Standards and Technology, Gaithersburg, MD 20899-8910 **USA**
E-mail: mascagni@fsu.edu or mascagni@math.ethz.ch
or mascagni@nist.gov
URL: http://www.cs.fsu.edu/~mascagni

Research supported by ARO, DOE, NASA, NATO, NIST, and NSF
with equipment donated by Intel and Nvidia

Outline of the Talk

Types of random numbers and Monte Carlo Methods

Pseudorandom number generation
- Types of pseudorandom numbers
- Properties of these pseudorandom numbers
- Parallelization of pseudorandom number generators
- New directions for `SPRNG`

Quasirandom number generation
- The Koksma-Hlawka inequality
- Discrepancy
- The van der Corput sequence
- Methods of quasirandom number generation
- Randomization

Conclusions and Future Work

Outline of the Talk

Types of random numbers and Monte Carlo Methods

Pseudorandom number generation
- Types of pseudorandom numbers
- Properties of these pseudorandom numbers
- Parallelization of pseudorandom number generators
- New directions for `SPRNG`

Quasirandom number generation
- The Koksma-Hlawka inequality
- Discrepancy
- The van der Corput sequence
- Methods of quasirandom number generation
- Randomization

Conclusions and Future Work

Outline of the Talk

- Types of random numbers and Monte Carlo Methods
- Pseudorandom number generation
  - Types of pseudorandom numbers
  - Properties of these pseudorandom numbers
  - Parallelization of pseudorandom number generators
  - New directions for `SPRNG`
- Quasirandom number generation
  - The Koksma-Hlawka inequality
  - Discrepancy
  - The van der Corput sequence
  - Methods of quasirandom number generation
  - Randomization
- Conclusions and Future Work

Outline of the Talk

- Types of random numbers and Monte Carlo Methods
- Pseudorandom number generation
  - Types of pseudorandom numbers
  - Properties of these pseudorandom numbers
  - Parallelization of pseudorandom number generators
  - New directions for `SPRNG`
- Quasirandom number generation
  - The Koksma-Hlawka inequality
  - Discrepancy
  - The van der Corput sequence
  - Methods of quasirandom number generation
  - Randomization
- Conclusions and Future Work

Monte Carlo Methods: Numerical Experimental that Use Random Numbers

- A Monte Carlo method is any process that consumes random numbers

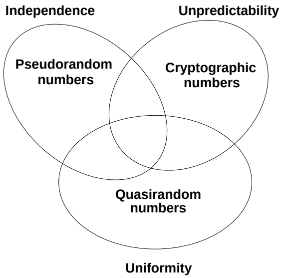
1. Each calculation is a numerical experiment
   - Subject to known and unknown sources of error
   - Should be reproducible by peers
   - Should be easy to run anew with results that can be combined to reduce the variance
2. Sources of errors must be controllable/isolatable
   - Programming/science errors under your control
   - Make possible RNG errors approachable
3. Reproducibility
   - Must be able to rerun a calculation with the same numbers
   - Across different machines (modulo arithmetic issues)
   - Parallel and distributed computers?

What are Random Numbers Used For?

1. Random numbers are used extensively in simulation, statistics, and in *Monte Carlo* computations
    - Simulation: use random numbers to “randomly pick" event outcomes based on statistical or experiential data
    - Statistics: use random numbers to generate data with a particular distribution to calculate statistical properties (when analytic techniques fail)
2. There are many Monte Carlo applications of great interest
    - Numerical quadrature “all Monte Carlo is integration"
    - Quantum mechanics: Solving Schrödinger's equation with Green's function Monte Carlo via random walks
    - Mathematics: Using the Feynman-Kac/path integral methods to solve partial differential equations with random walks
    - Defense: neutronics, nuclear weapons design
    - Finance: options, mortgage-backed securities

What are Random Numbers Used For?

3. There are many types of random numbers
   - "Real" random numbers: uses a 'physical source' of randomness
   - Pseudorandom numbers: deterministic sequence that passes tests of randomness
   - Quasirandom numbers: well distributed (low discrepancy) points



Pseudorandom Numbers

- Pseudorandom numbers mimic the properties of 'real' random numbers

A. Pass statistical tests

B. Reduce error is $O(N^{-\frac{1}{2}})$ in Monte Carlo

- Some common pseudorandom number generators (RNG):

1. Linear congruential: $x_n = ax_{n-1} + c \pmod{m}$
2. Implicit inversive congruential: $x_n = a\overline{x_{n-1}} + c \pmod{p}$
3. Explicit inversive congruential: $x_n = a\overline{n} + c \pmod{p}$
4. Shift register: $y_n = y_{n-s} + y_{n-r} \pmod{2}$, $r > s$
5. Additive lagged-Fibonacci: $z_n = z_{n-s} + z_{n-r} \pmod{2^k}$, $r > s$
6. Combined: $w_n = y_n + z_n \pmod{p}$
7. Multiplicative lagged-Fibonacci: $x_n = x_{n-s} \times x_{n-r} \pmod{2^k}$, $r > s$

Pseudorandom Numbers

- Some properties of pseudorandom number generators, integers: $\{x_n\}$ from modulo m recursion, and $U[0,1]$, $z_n = \frac{x_n}{m}$

A. Should be a purely periodic sequence (e.g.: DES and IDEA are not provably periodic)

B. Period length: $\mathrm{Per}(x_n)$ should be large

C. Cost per bit should be moderate (not cryptography)

D. Should be based on theoretically solid and empirically tested recursions

E. Should be a totally reproducible sequence

Pseudorandom Numbers

- Some common facts (rules of thumb) about pseudorandom number generators:

1. Recursions modulo a power-of-two are cheap, but have simple structure
2. Recursions modulo a prime are more costly, but have higher quality: use Mersenne primes: $2^p - 1$, where
3. Shift-registers (Mersenne Twisters) are efficient and have good quality
4. Lagged-Fibonacci generators are efficient, but have some structural flaws
5. Combining generators is 'provably good'
6. Modular inversion is very costly
7. All linear recursions 'fall in the planes'
8. Inversive (nonlinear) recursions 'fall on hyperbolas'

Periods of Pseudorandom Number Generators

1. Linear congruential: $x_n = ax_{n-1} + c \pmod{m}$, $\mathrm{Per}(x_n) = m - 1$, m prime, with m a power-of-two,
2. Implicit inversive congruential: $x_n = a\overline{x_{n-1}} + c \pmod{p}$, $\mathrm{Per}(x_n) = p$
3. Explicit inversive congruential: $x_n = a\overline{n} + c \pmod{p}$, $\mathrm{Per}(x_n) = p$
4. Shift register: $y_n = y_{n-s} + y_{n-r} \pmod{2}$, $r > s$, $\mathrm{Per}(y_n) = 2^r - 1$
5. Additive lagged-Fibonacci: $z_n = z_{n-s} + z_{n-r} \pmod{2^k}$, $r > s$, $\mathrm{Per}(z_n) = (2^r - 1)2^{k-1}$
6. Combined: $w_n = y_n + z_n \pmod{p}$, $\mathrm{Per}(w_n) = \mathrm{lcm}(\mathrm{Per}(y_n), \mathrm{Per}(z_n))$
7. Multiplicative lagged-Fibonacci: $x_n = x_{n-s} \times x_{n-r} \pmod{2^k}$, $r > s$, $\mathrm{Per}(x_n) = (2^r - 1)2^{k-3}$

Combining RNGs

- There are many methods to combine two streams of random numbers, $\{x_n\}$ and $\{y_n\}$, where the y_n's modulo m_y:

1. Addition modulo one: $z_n = \frac{x_n}{m_x} + \frac{y_n}{m_y} \pmod 1$
2. Addition modulo either m_x or m_y
3. Multiplication and reduction modulo either m_x or m_y
4. Exclusive "or-ing"

- Rigorously provable that linear combinations produce combined streams that are "no worse" than the worst
- Tony Warnock: all the above methods seem to do about the same

Splitting RNGs for Use In Parallel

- We consider splitting a single PRNG:
  - Assume $\{x_n\}$ has $\text{Per}(x_n)$
  - Has the fast-leap ahead property: leaping L ahead costs no more than generating $O(\log_2(L))$ numbers
- Then we associate a single block of length L to each parallel subsequence:

1. Blocking:
   - First block: $\{x_0, x_1, \ldots, x_{L-1}\}$
   - Second : $\{x_L, x_{L+1}, \ldots, x_{2L-1}\}$
   - ith block: $\{x_{(i-1)L}, x_{(i-1)L+1}, \ldots, x_{iL-1}\}$
2. The Leap Frog Technique: define the leap ahead of $\ell = \left\lfloor \frac{\text{Per}(x_i)}{L} \right\rfloor$:
   - First block: $\{x_0, x_\ell, x_{2\ell}, \ldots, x_{(L-1)\ell}\}$
   - Second block: $\{x_1, x_{1+\ell}, x_{1+2\ell}, \ldots, x_{1+(L-1)\ell}\}$
   - ith block: $\{x_i, x_{i+\ell}, x_{i+2\ell}, \ldots, x_{i+(L-1)\ell}\}$

Splitting RNGs for Use In Parallel

3. The Lehmer Tree, designed for splitting LCGs:
   - Define a right and left generator: $R(x)$ and $L(x)$
   - The right generator is used within a process
   - The left generator is used to spawn a new PRNG stream
   - Note: $L(x) = R^W(x)$ for some W for **all** x for an LCG
   - Thus, spawning is just jumping a fixed, W, amount in the sequence
4. Recursive Halving Leap-Ahead, use fixed points or fixed leap aheads:
   - First split leap ahead: $\left\lfloor \frac{\mathrm{Per}(x_i)}{2} \right\rfloor$
   - ith split leap ahead: $\left\lfloor \frac{\mathrm{Per}(x_i)}{2^{l+1}} \right\rfloor$
   - This permits effective user of all remaining numbers in $\{x_n\}$ without the need for *a priori* bounds on the stream
     length L

Generic Problems Parallelizing via Splitting

1. Splitting for parallelization is not scalable:
   - It usually costs $O(\log_2(\text{Per}(x_i)))$ bit operations to generate a random number
   - For parallel use, a given computation that requires L random numbers per process with P processes must have $\text{Per}(x_i) = O((LP)^e)$
   - Rule of thumb: never use more than $\sqrt{\text{Per}(x_i)}$ of a sequence $\rightarrow e = 2$
   - Thus cost per random number is not constant with number of processors!!

Generic Problems Parallelizing via Splitting

2. Correlations within sequences are generic!!
   - Certain offsets within any modular recursion will lead to extremely high correlations
   - Splitting in any way converts auto-correlations to cross-correlations between sequences
   - Therefore, splitting generically leads to interprocessor correlations in PRNGs

New Results in Parallel RNGs: Using Distinct Parameterized Streams in Parallel

1. Default generator: additive lagged-Fibonacci,
   $x_n = x_{n-s} + x_{n-r} \pmod{2^k}$, $r > s$
   - Very efficient: 1 add & pointer update/number
   - Good empirical quality
   - Very easy to produce distinct parallel streams
2. Alternative generator #1: prime modulus LCG,
   $x_n = a x_{n-1} + c \pmod{m}$
   - Choice: Prime modulus (quality considerations)
   - Parameterize the multiplier
   - Less efficient than lagged-Fibonacci
   - Provably good quality
   - Multiprecise arithmetic in initialization

New Results in Parallel RNGs: Using Distinct Parameterized Streams in Parallel

3. Alternative generator #2: power-of-two modulus LCG, $x_n = ax_{n-1} + c \pmod{2^k}$
    - Choice: Power-of-two modulus (efficiency considerations)
    - Parameterize the prime additive constant
    - Less efficient than lagged-Fibonacci
    - Provably good quality
    - Must compute as many primes as streams

Parameterization Based on Seeding

- Consider the Lagged-Fibonacci generator:

$$x_n = x_{n-5} + x_{n-17} \pmod{2^{32}} \text{ or in general:}$$

$$x_n = x_{n-s} + x_{n-r} \pmod{2^k},\ r > s$$

- The seed is 17 32-bit integers; 544 bits, longest possible period for this linear generator is $2^{17\times 32} - 1 = 2^{544} - 1$
- Maximal period is $\mathrm{Per}(x_n) = (2^{17} - 1) \times 2^{31}$
- Period is maximal $\iff$ at least one of the 17 32-bit integers is odd
- This seeding failure results in only even "random numbers"
- Are $(2^{17} - 1) \times 2^{31\times 17}$ seeds with full period
- Thus there are the following number of full-period equivalence classes (ECs):

$$E = \frac{(2^{17} - 1) \times 2^{31\times 17}}{(2^{17} - 1) \times 2^{31}} = 2^{31\times 16} = 2^{496}$$

The Equivalence Class Structure

With the "standard" l.s.b., b_0:

m.s.b. b_{k-1}	b_{k-2}	$\dots$	b_1	l.s.b. b_0	
$\square$	$\square$	$\dots$	0	0	x_{r-1}
0	$\square$	$\dots$	$\square$	0	x_{r-2}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	
$\square$	0	$\dots$	$\square$	0	x_1
$\square$	$\square$	$\dots$	$\square$	1	x_0

or a special b_0 (adjoining 1's):

m.s.b. b_{k-1}	b_{k-2}	$\dots$	b_1	l.s.b. b_0	
$\square$	$\square$	$\dots$	$\square$	$b_{0\,n-1}$	x_{r-1}
$\square$	$\square$	$\dots$	$\square$	$b_{0\,n-2}$	x_{r-2}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	
$\square$	$\square$	$\dots$	$\square$	$b_{0\,1}$	x_1
0	0	$\dots$	0	$b_{0\,0}$	x_0

Parameterization of Prime Modulus LCGs

- Consider only $x_n = ax_{n-1} \pmod{m}$, with m prime has maximal period when a is a primitive root modulo m
- If α and a are primitive roots modulo m then $\exists\, l$ s.t. $\gcd(l, m-1) = 1$ and $\alpha \equiv a^l \pmod{m}$
- If $m = 2^{2^n} + 1$ (Fermat prime) then all odd powers of α are primitive elements also
- If $m = 2q + 1$ with q also prime (Sophie-Germain prime) then all odd powers (save the qth) of α are primitive elements

Parameterization of Prime Modulus LCGs

- Consider $x_n = a x_{n-1} \pmod{m}$ and $y_n = a^l y_{n-1} \pmod{m}$ and define the full-period exponential-sum

$$C(j, l) = \sum_{n=0}^{m-1} e^{\frac{2\pi i}{m}(x_n - y_{n-j})}$$

then the Riemann hypothesis over finite-fields implies $|C(j, l)| \leq (l-1)\sqrt{m}$

Parameterization of Prime Modulus LCGs

- Mersenne modulus: relatively easy to do modular multiplication
- With Mersenne prime modulus, $m = 2^p - 1$ must compute $\phi_{m-1}^{-1}(k)$, the kth number relatively prime to $m - 1$ item Can compute $\phi_{m-1}(x)$ with a variant of the Meissel-Lehmer algorithm fairly quickly:
  - Use partial sieve functions to trade off memory for more than 2^j operations, $j = \#$ of factors of
  - Have fast implementation for $p = 31, 61, 127, 521, 607$

Parameterization of Power-of-Two Modulus LCGs

- $x_n = ax_{n-1} + c_i \pmod{2^k}$, here the c_i's are distinct primes
- Can prove (Percus and Kalos) that streams have good spectral test properties among themselves
- Best to choose $c_i \approx \sqrt{2^k} = 2^{k/2}$
- Must enumerate the primes, uniquely, not necessarily exhaustively to get a unique parameterization
- Note: in $0 \leq i < m$ there are $\approx \frac{m}{\log_2 m}$ primes via the prime number theorem, thus if $m \approx 2^k$ streams are required, then must exhaust all the primes modulo $\approx 2^{k+\log_2 k} = 2^k k = m \log_2 m$
- Must compute distinct primes on the fly either with table or something like Meissel-Lehmer algorithm

Parameterization of MLFGs

1. Recall the MLFG recurrence:
   $x_n = x_{n-s} \times x_{n-r} \pmod{2^k}$, $r > s$
2. One of the r seed elements is even $\rightarrow$ eventually all become even
3. Restrict to only odd numbers in the MLFG seeds
4. Allows the following parameterization for odd integers modulo a power-of-two
   $x_n = (-1)^{y_n} 3^{z_n} \pmod{2^k}$, where $y_n \in \{0, 1\}$ and where
   - $y_n = y_{n-s} + y_{n-r} \pmod{2}$
   - $z_n = z_{n-s} + z_{n-r} \pmod{2^{k-2}}$
5. Last recurrence means we can us ALFG parameterization, z_n, and map to MLFGs via modular exponentiation

Quality Issues in Serial and Parallel PRNGs

- Empirical tests (more later)
- Provable measures of quality:

1. Full- and partial-period discrepancy (Niederreiter) test equidistribution of overlapping k-tuples
2. Also full- ($k = \text{Per}(x_n)$) and partial-period exponential sums:

$$C(j,k) = \sum_{n=0}^{k-1} e^{\frac{2\pi}{m}(x_n - x_{n-j})}$$

Quality Issues in Serial and Parallel PRNGs

- For LCGs and SRGs full-period and partial-period results are similar
  - ▷ $|C(\cdot, \mathrm{Per}(x_n))| < O(\sqrt{\mathrm{Per}(x_n)})$
  - ▷ $|C(\cdot, j)| < O(\sqrt{\mathrm{Per}(x_n)})$
- Additive lagged-Fibonacci generators have poor provable results, yet empirical evidence suggests

Parallel Neutronics: A Difficult Example

1. The structure of parallel neutronics
   - Use a parallel queue to hold unfinished work
   - Each processor follows a distinct neutron
   - Fission event places a new neutron(s) in queue with initial conditions
2. Problems and solutions
   - Reproducibility: each neutron is queued with a new generator (and with the next generator)
   - Using the binary tree mapping prevents generator reuse, even with extensive branching
   - A global seed reorders the generators to obtain a statistically significant new but reproducible result

Many Parameterized Streams Facilitate Implementation/Use

1. Advantages of using parameterized generators
    - Each unique parameter value gives an "independent" stream
    - Each stream is uniquely numbered
    - Numbering allows for absolute reproducibility, even with MIMD queuing
    - Effective serial implementation + enumeration yield a portable scalable implementation
    - Provides theoretical testing basis

Many Parameterized Streams Facilitate Implementation/Use

2. Implementation details
   - Generators mapped canonically to a binary tree
   - Extended seed data structure contains current seed and next generator
   - Spawning uses new next generator as starting point: assures no reuse of generators
3. All these ideas in the **S**calable **P**arallel **R**andom **N**umber **G**enerators (`SPRNG`) library: `http://sprng.org`

Many Different Generators and A Unified Interface

1. Advantages of having more than one generator
   - An application exists that stumbles on a given generator
   - Generators based on different recursions allow comparison to rule out spurious results
   - Makes the generators real experimental tools
2. Two interfaces to the `SPRNG` library: simple and default
   - Initialization returns a pointer to the generator state: `init_SPRNG()`
   - Single call for new random number: `SPRNG()`
   - Generator type chosen with parameters in `init_SPRNG()`
   - Makes changing generator very easy
   - Can use more than one generator *type* in code
   - Parallel structure is extensible to new generators through dummy routines

New Directions for SPRNG

- SPRNG was originally designed for distributed-memory multiprocessors
- HPC architectures are increasingly based on commodity chips with architectural variations
  1. Microprocessors with more than one processor core (multicore)
  2. Microprocessors with accelerators, most popular being GPGPUs (video games)
  3. Intel Phi
- We will consider only two of these:
  1. Multicore support using OpenMP
  2. GPU support using CUDA (Nvidia) and/or OpenCL (standard)
  3. OpenACC?

SPRNG Update Overview

- SPRNG uses independent full-period cycles for each processor
  1. Organizes the independent use of generators without communication
  2. Permits reproducibility
  3. Initialization of new full-period generators is slow for some generators
- A possible solution
  1. Keep the independent full-period cycles for “top-level" generators
  2. Within these (multicore processor/GPU) use cycle splitting to service threads

Experience with Multicore

- We have implemented an OpenMP version of SPRNG for multicore using these ideas
- OpenMP is now built into the main compilers, so it is easy to access
- Our experience has been
  1. It works as expected giving one access to Monte Carlo on all the cores
  2. Permits reproducibility but with some work: must know the number of threads
  3. Near perfect parallelization is expected and seen
  4. Comparison with independent spawning vs. cycle splitting is not as dramatic as expected
- Backward reproducibility is something that we can provide, but forward reproducibility is trickier
- This version is a prototype, but will be used for the eventual creation of the multicore version of SPRNG
- Work with Drs. Haohai Yu and Yue Qiu

Experience with SPRNG on GPGPUs via CUDA

- SPRNG for the GPU will be simple in principal, but harder for users
  1. The same technique that was used for multicore will work for GPUs with many of the same issues
  2. The concept of reproducibility will have to modified as well
  3. Successful exploitation of GPU threads will require that SPRNG calls be made to insure that the data and the execution are on the GPU
- The software development may not be the hardest aspect of this work
  1. Clear documentation with descriptions of common coding errors will be essential for success
  2. An extensive library of examples will be necessary to provide most users with code close to their own to help use the GPU efficiently for Monte Carlo
- Currently have Dr. Qiu's codes for SGMLCGs and MLFGs

Other SPRNG Opportunities

- SPRNG for use on Intel Phi
  1. Rakesh is doing some work on the Phi, but I am unsure what has been accomplished
  2. Rakesh owes us another presentation on using our Phi!!
- Work on the website using sprng.kissr.com as a local sandbox
  1. Creating a clean version of the v4.4 code
  2. Making the website more user friendly
  3. I need volunteers for this to get access to login.shodor.org and also the kissr website on Dropbox

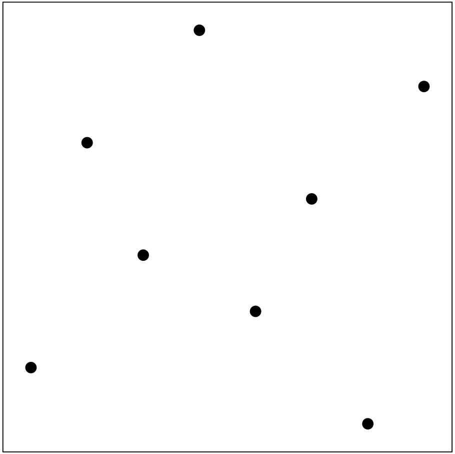
Quasirandom Numbers

- Many problems require uniformity, not randomness: "quasirandom" numbers are highly uniform deterministic sequences with small *star discrepancy*
- **Definition**: The *star discrepancy* D_N^* of $x_1, \dots, x_N$:

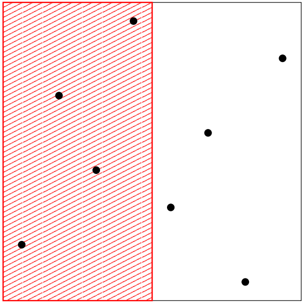
$$\begin{aligned} D_N^* &= D_N^*(x_1, \dots, x_N) \\ &= \sup_{0 \le u \le 1} \left| \frac{1}{N} \sum_{n=1}^{N} \chi_{[0,u)}(x_n) - u \right|, \end{aligned}$$

where χ is the characteristic function

Star Discrepancy in 2D

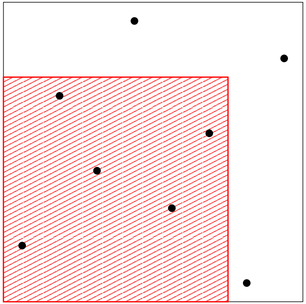


Star Discrepancy in 2D



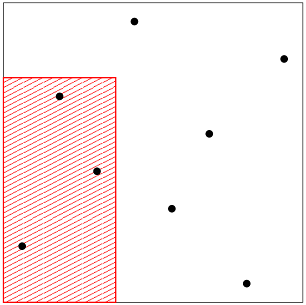
- $\left|\frac{1}{2} - \frac{4}{8}\right| = 0$
-
-
-

Star Discrepancy in 2D



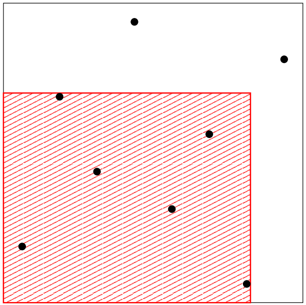
- $\left|\frac{1}{2} - \frac{4}{8}\right| = 0$
- $\left|\frac{9}{16} - \frac{5}{8}\right| = \frac{1}{16} = 0.0625$
-
-

Star Discrepancy in 2D



- $\left|\frac{1}{2} - \frac{4}{8}\right| = 0$
- $\left|\frac{9}{16} - \frac{5}{8}\right| = \frac{1}{16} = 0.0625$
- $\left|\frac{9}{32} - \frac{3}{8}\right| = \frac{3}{32} = 0.09375$
-

Star Discrepancy in 2D



- $\left|\frac{1}{2} - \frac{4}{8}\right| = 0$
- $\left|\frac{9}{16} - \frac{5}{8}\right| = \frac{1}{16} \quad = 0.0625$
- $\left|\frac{9}{32} - \frac{3}{8}\right| = \frac{3}{32} \quad = 0.09375$
- $\left|\frac{143}{256} - \frac{6}{8}\right| = \frac{49}{256} \approx 0.19140625$

Quasirandom Numbers

- **Theorem** (Koksma, 1942): if $f(x)$ has bounded variation $V(f)$ on $[0,1]$ and $x_1, \dots, x_N \in [0,1]$ with star discrepancy D_N^*, then:

$$\left| \frac{1}{N} \sum_{n=1}^{N} f(x_n) - \int_0^1 f(x)\, dx \right| \le V(f) D_N^*,$$

  this is the Koksma-Hlawka inequality
- Note: Many different types of discrepancies are definable

Discrepancy Facts

- Real random numbers have (the law of the iterated logarithm):

$$D_N^* = O(N^{-1/2}(\log\log N)^{-1/2})$$

- Klaus F. Roth (Fields medalist in 1958) proved the following lower bound in 1954 for the star discrepancy of N points in s dimensions:

$$D_N^* \geq O(N^{-1}(\log N)^{\frac{s-1}{2}})$$

- Sequences (indefinite length) and point sets have different "best discrepancies" at present
  - Sequence: $D_N^* \leq O(N^{-1}(\log N)^{s-1})$
  - Point set: $D_N^* \leq O(N^{-1}(\log N)^{s-2})$

Some Types of Quasirandom Numbers

- Must choose point sets (finite $\#$) or sequences (infinite $\#$) with small D_N^*
- Often used is the *van der Corput sequence* in base b: $x_n = \Phi_b(n-1), n = 1, 2, \dots,$ where for $b \in \mathbb{Z}, b \geq 2$:

$$\Phi_b\left(\sum_{j=0}^{\infty} a_j b^j\right) = \sum_{j=0}^{\infty} a_j b^{-j-1} \quad \text{with}$$

$$a_j \in \{0, 1, \dots, b-1\}$$

Some Types of Quasirandom Numbers

- For the van der Corput sequence

$$ND_N^* \leq \frac{\log N}{3 \log 2} + O(1)$$

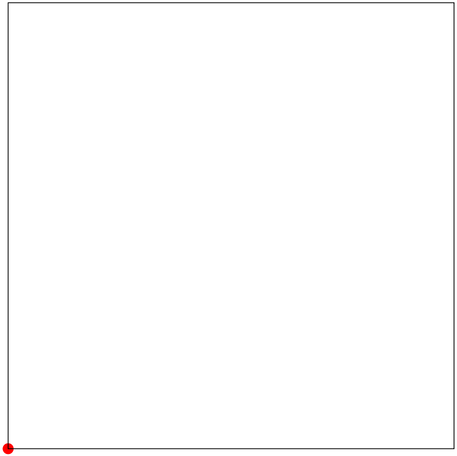
- With $b = 2$, we get $\{\frac{1}{2}, \frac{1}{4}, \frac{3}{4}, \frac{1}{8}, \frac{5}{8}, \frac{3}{8}, \frac{7}{8} \dots\}$
- With $b = 3$, we get $\{\frac{1}{3}, \frac{2}{3}, \frac{1}{9}, \frac{4}{9}, \frac{7}{9}, \frac{2}{9}, \frac{5}{9}, \frac{8}{9} \dots\}$

Some Types of Quasirandom Numbers

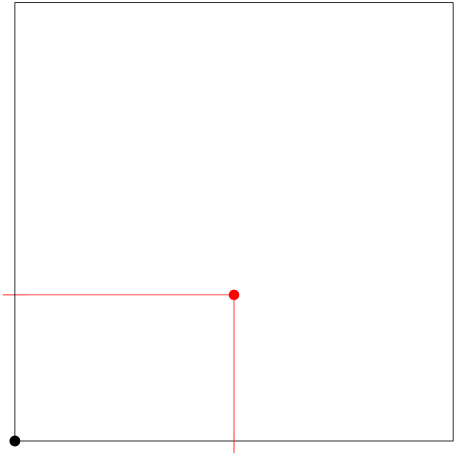
- Other small D_N^* points sets and sequences:

1. Halton sequence: $\mathbf{x}_n = (\Phi_{b_1}(n-1), \dots, \Phi_{b_s}(n-1))$, $n = 1, 2, \dots,$
   $D_N^* = O\left(N^{-1}(\log N)^s\right)$ if $b_1, \dots, b_s$ pairwise relatively prime
2. Hammersley point set: $\mathbf{x}_n = \left(\frac{n-1}{N}, \Phi_{b_1}(n-1), \dots, \Phi_{b_{s-1}}(n-1)\right)$, $n = 1, 2, \dots, N$,
   $D_N^* = O\left(N^{-1}(\log N)^{s-1}\right)$ if $b_1, \dots, b_{s-1}$ are pairwise relatively prime

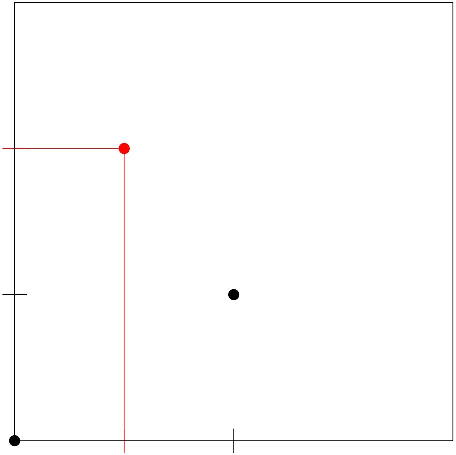
Halton sequence: example



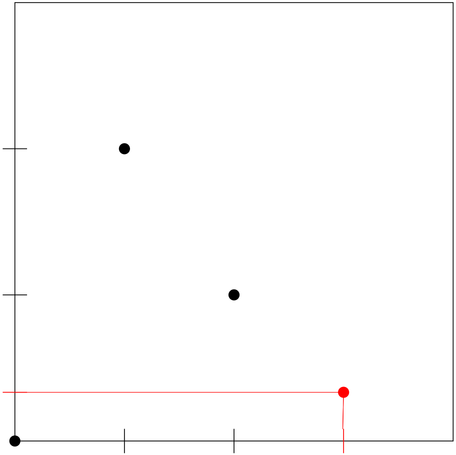
Halton sequence: example



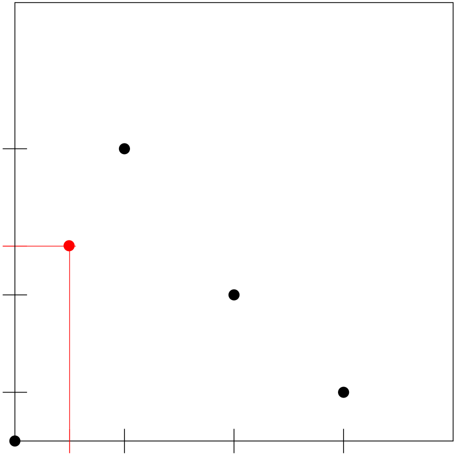
Halton sequence: example



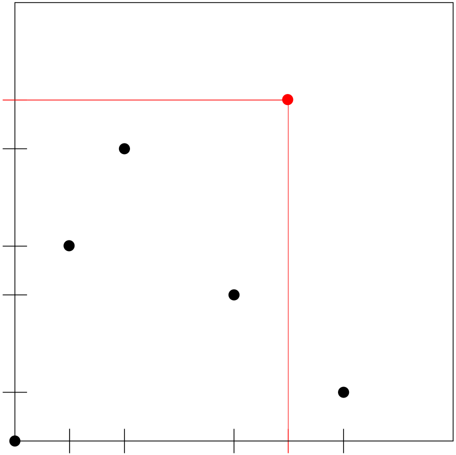
Halton sequence: example



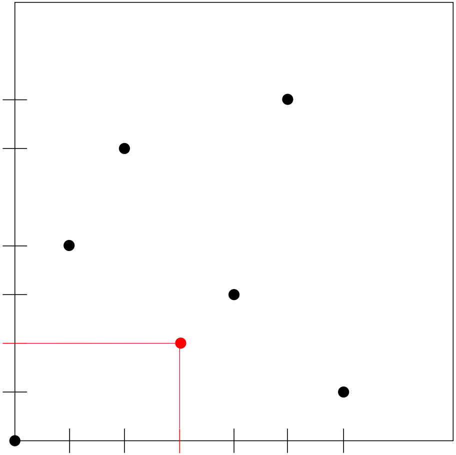
Halton sequence: example



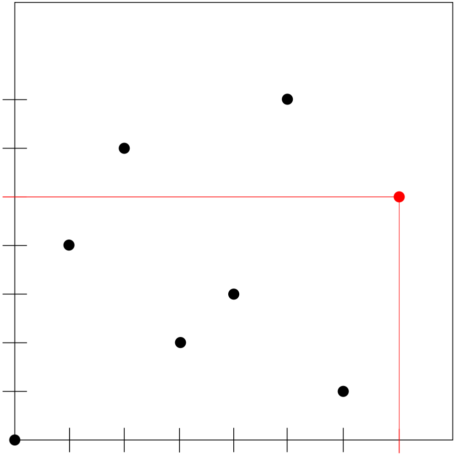
Halton sequence: example



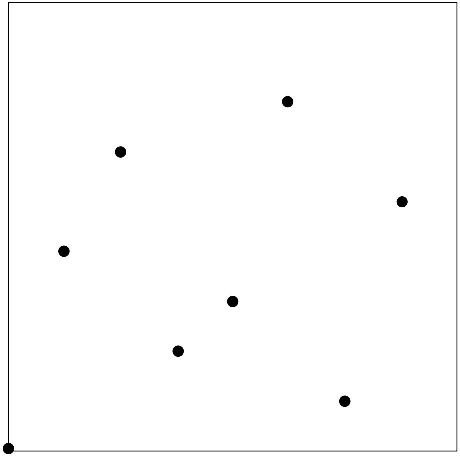
Halton sequence: example



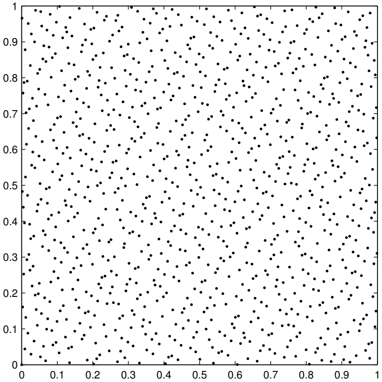
Halton sequence: example



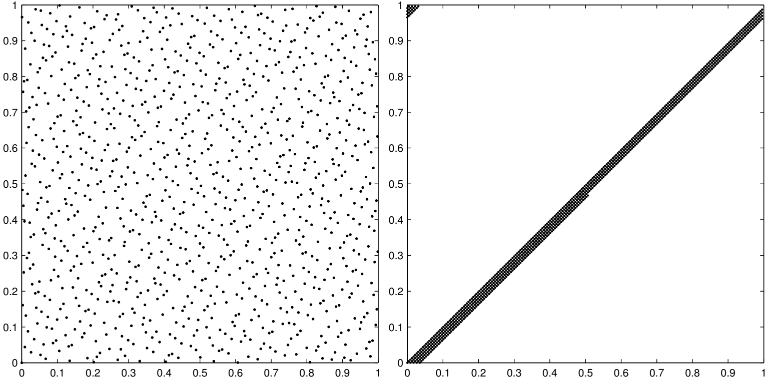
Halton sequence: example



Good Halton points vs poor Halton points



Good Halton points vs poor Halton points



Some Types of Quasirandom Numbers

3. Ergodic dynamics: $\mathbf{x}_n = \{n\alpha\}$, where $\alpha = (\alpha_1, \dots, \alpha_s)$ is irrational and $\alpha_1, \dots, \alpha_s$ are linearly independent over the rationals then for almost all $\alpha \in \mathbb{R}^s$, $D_N^* = O(N^{-1}(\log N)^{s+1+\epsilon})$ for all $\epsilon > 0$
4. Other methods of generation
   - Method of good lattice points (Sloan and Joe)
   - Soboĺ sequences
   - Faure sequences (more later)
   - Niederreiter sequences

Continued-Fractions and Irrationals

Infinite continued-fraction expansion for choosing good irrationals:

$$r = a_0 + \frac{1}{a_1 + \frac{1}{a_2 + \dots}}$$

$a_i \leq K \longrightarrow$ sequence is a low-discrepancy sequence
Choose all $a_i = 1$. Then

$$r = 1 + \frac{1}{1 + \frac{1}{1 + \dots}}.$$

is the golden ratio.

$$0.618,\ 0.236,\ 0.854,\ 0.472,\ 0.090,\ \dots$$

Irrational sequence in more dimensions is not a low-discrepancy sequence.

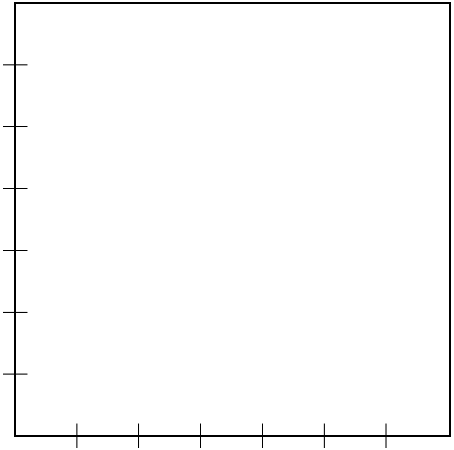
Lattice

- Fixed N
- Generator vector $\vec{g} = (g_1, \ldots, g_d) \in \mathbb{Z}^d$.

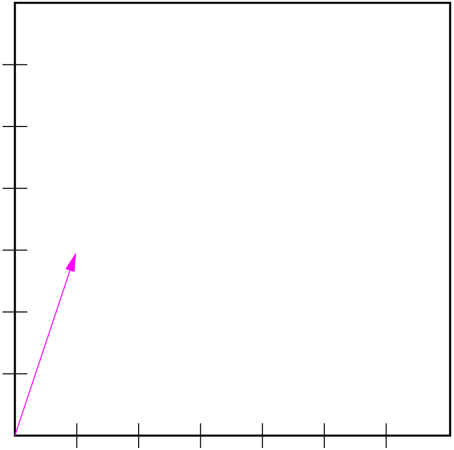
We define a rank-1 lattice as

$$P_{\text{lattice}} := \left\{ \vec{x}_i = \frac{i\vec{g}}{N} \bmod 1 \right\}.$$

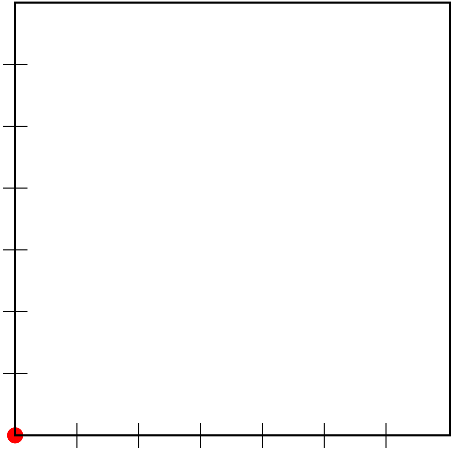
An example lattice



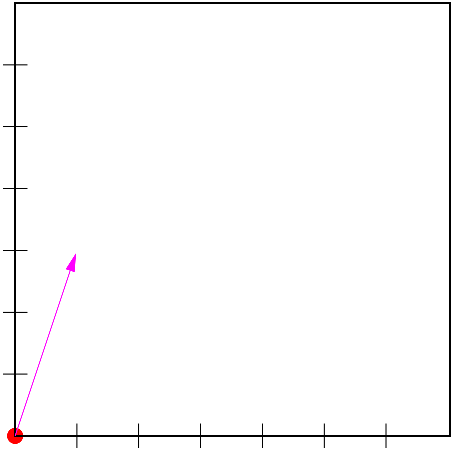
An example lattice



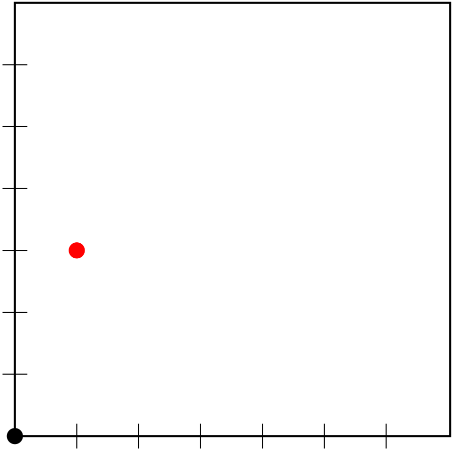
An example lattice



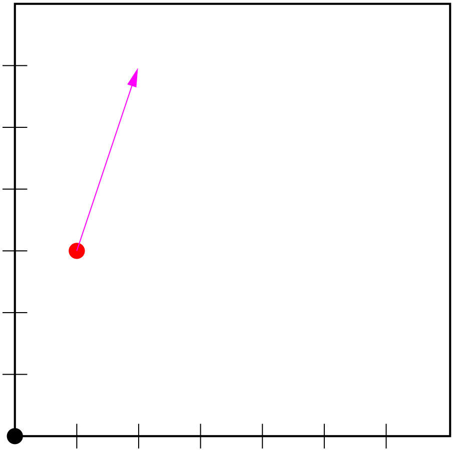
An example lattice



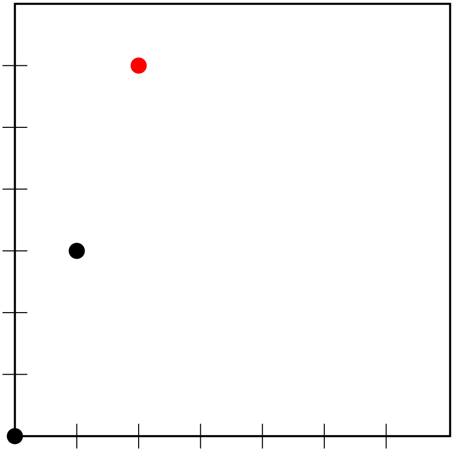
An example lattice



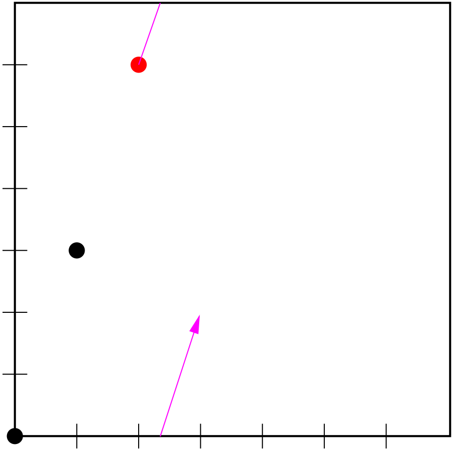
An example lattice



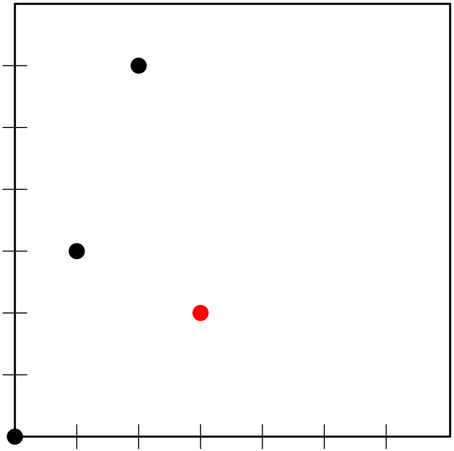
An example lattice



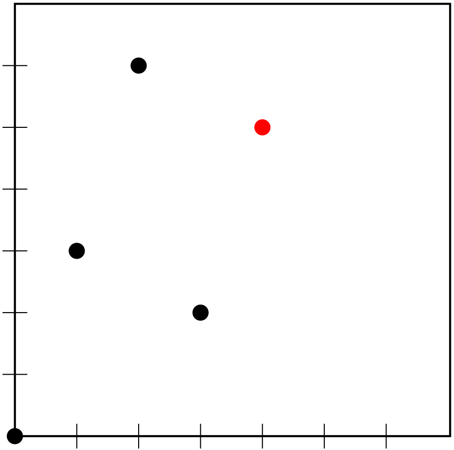
An example lattice



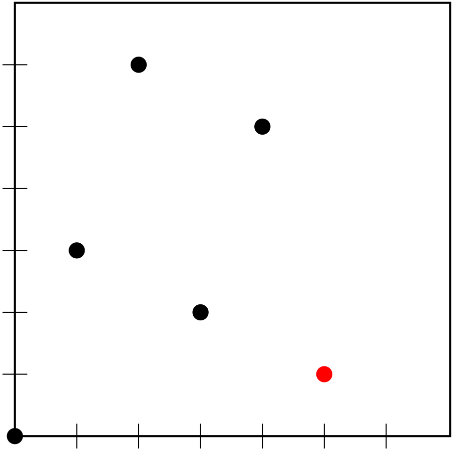
An example lattice



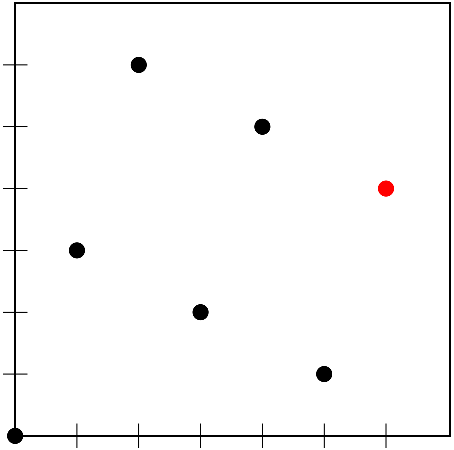
An example lattice



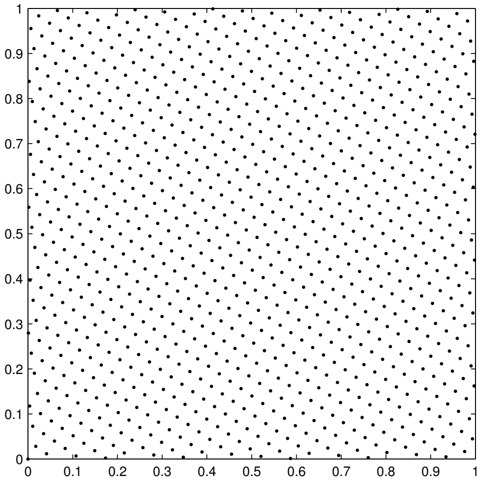
An example lattice



An example lattice



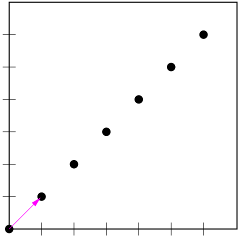
Lattice with 1031 points



Lattice

- After N points the sequence repeats itself,
- Projection on each axe gives the set $\{\frac{0}{N}, \frac{1}{N}, \dots, \frac{N-1}{N}\}$.

Not every generator gives a good point set. E.g. $g_1 = g_2 = \cdots = g_d = 1$, gives $\{(\frac{i}{N}, \dots, \frac{i}{N})\}$.



Some Types of Quasirandom Numbers

1. Another interpretation of the v.d. Corput sequence:
   - Define the ith ℓ-bit "direction number" as: $v_i = 2^i$ (think of this as a bit vector)
   - Represent $n-1$ via its base-2 representation $n-1 = b_{\ell-1}b_{\ell-2}\dots b_1 b_0$
   - Thus we have
   $$\Phi_2(n-1) = 2^{-\ell} \bigoplus_{i=0,\ b_i=1}^{i=\ell-1} v_i$$
2. The Soboĺ sequence works the same!!
   - Use recursions with a primitive binary polynomial define the (dense) v_i
   - The Soboĺ sequence is defined as:
   $$s_n = 2^{-\ell} \bigoplus_{i=0,\ b_i=1}^{i=\ell-1} v_i$$
   - Use Gray-code ordering for speed

Some Types of Quasirandom Numbers

- (t, m, s)-nets and (t, s)-sequences and generalized Niederreiter sequences

1. Let $b \geq 2$, $s > 1$ and $0 \leq t \leq m \in \mathbb{Z}$ then a b-ary box, $J \subset [0, 1)^s$, is given by

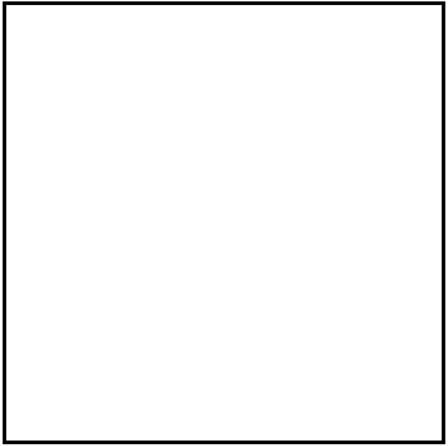
$$J = \prod_{i=1}^{s} [\frac{a_i}{b^{d_i}}, \frac{a_i + 1}{b^{d_i}})$$

where $d_i \geq 0$ and the a_i are b-ary digits, note that $|J| = b^{-\sum_{i=1}^{s} d_i}$

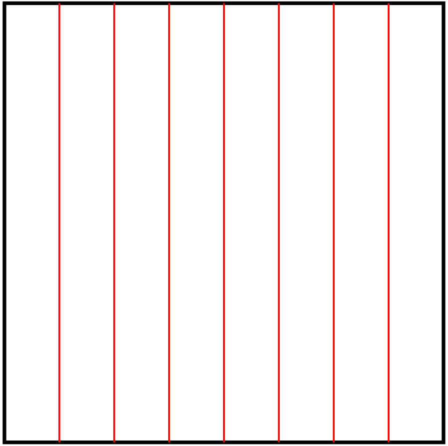
Some Types of Quasirandom Numbers

2. A set of b^m points is a (t, m, s)-net if each b-ary box of volume b^{t-m} has exactly b^t points in it
3. Such (t, m, s)-nets can be obtained via Generalized Niederreiter sequences, in dimension j of s: $y_i^{(j)}(n) = C^{(j)} a_i(n)$, where n has the b-ary representation $n = \sum_{k=0}^{\infty} a_k(n) b^k$ and $x_i^{(j)}(n) = \sum_{k=1}^{m} y_k^{(j)}(n) q^{-k}$

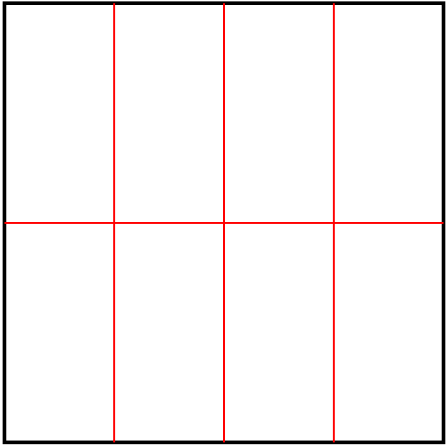
Nets: Example



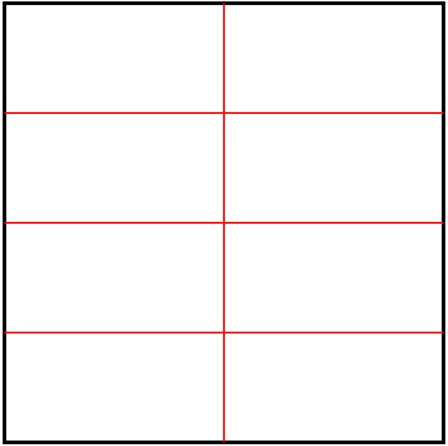
Nets: Example



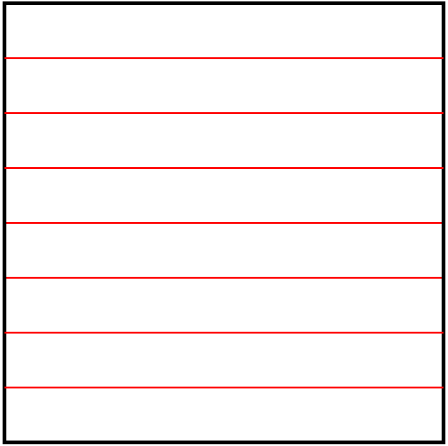
Nets: Example



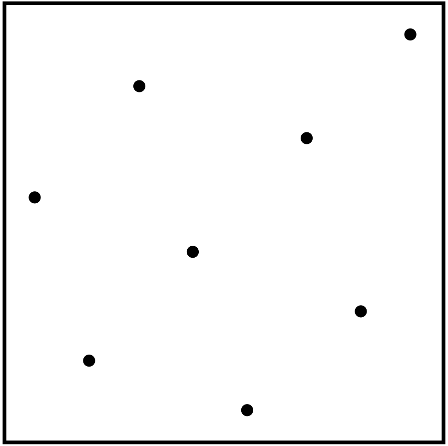
Nets: Example



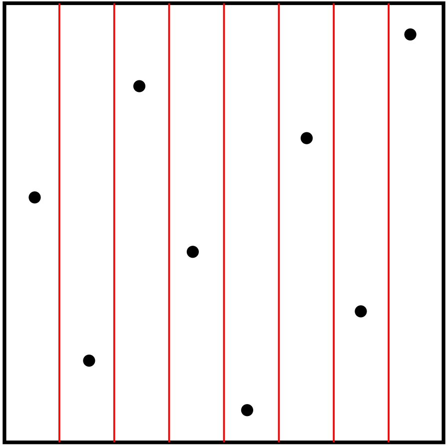
Nets: Example



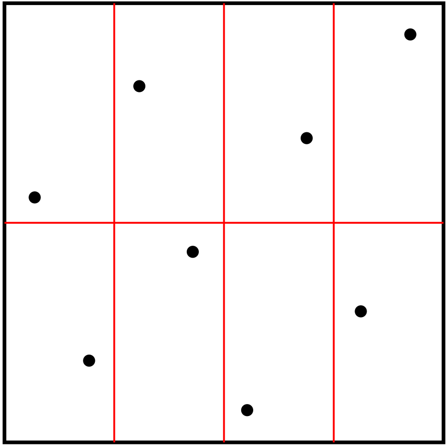
Nets: Example



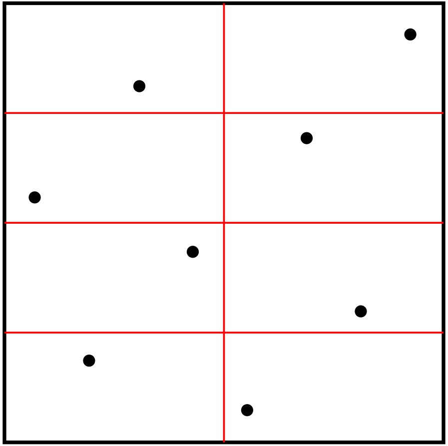
Nets: Example



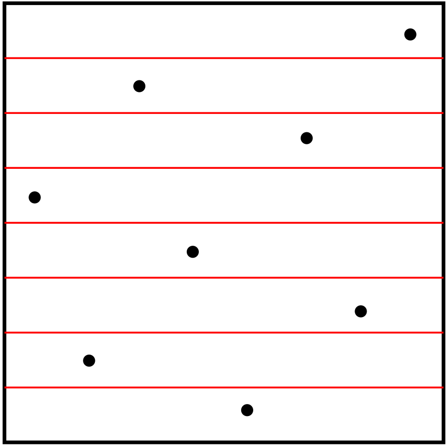
Nets: Example



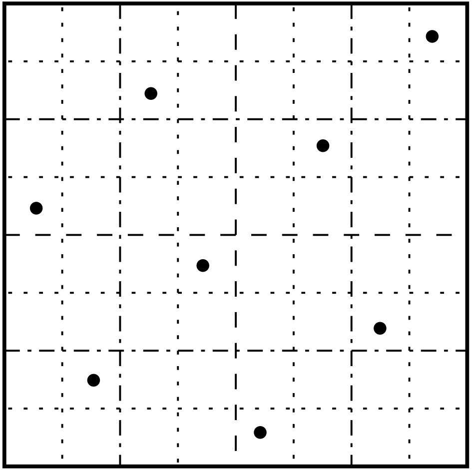
Nets: Example

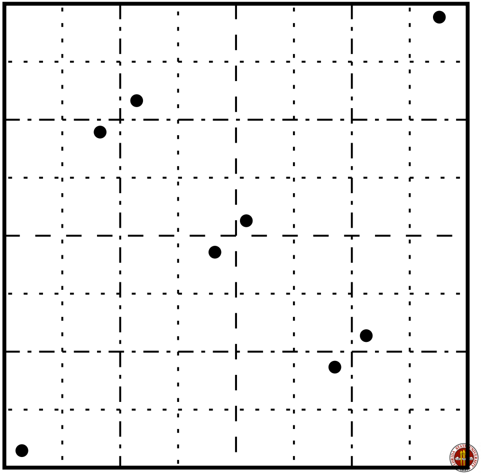


Nets: Example



Good vs poor net



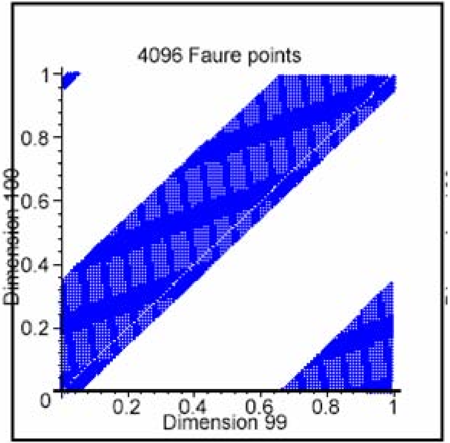


Randomization of the Faure Sequence

1. A problem with all QRNs is that the Koksma-Hlawka inequality provides no practical error estimate
2. A solution is to randomize the QRNs and then consider each randomized sequence as providing an independent sample for constructing confidence intervals
3. Consider the s-dimensional Faure series is:
   $(\phi_p(C^{(0)}(n)), \phi_p(C^{(1)}(n)), \dots, \phi_p(P^{s-1}(n)))$
   - $p > s$ is prime
   - $C^{(j-1)}$ is the generator matrix for dimension $1 \leq j \leq s$
   - For Faure $C^{(j)} = P^{j-1}$ is the Pascal matrix: $P^{j-1}_{r,k} = \binom{r-1}{k-1}(j-1)^{(r-k)} \pmod{p}$

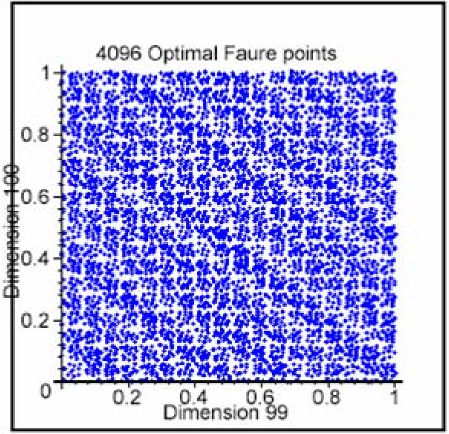
Another Reason for Randomization

QRNs have inherently bad low-dimensional projections



Another Reason for Randomization

Randomization (scrambling) helps



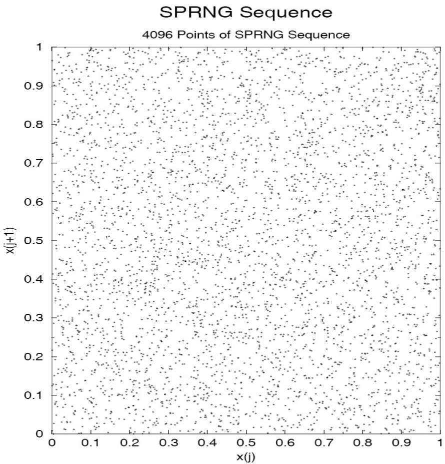
General Randomization Techniques

1. Random shifting: $z_n = x_n + r \pmod 1$
   - $x_n \in [0, 1]^s$ is the original QRN
   - $r \in [0, 1]^s$ is a random point
   - $z_n \in [0, 1]^s$ scrambled point
2. Digit permutation
   - Nested scrambling (Owen)
   - Single digit scrambling like linear scrambling
3. Randomization of the generator matrices, i.e. Tezuka's `GFaure`, $C^{(j)} = A^{(j)} P^{j-1}$ where A^j is a random nonsingular lower-triangular matrix modulo p

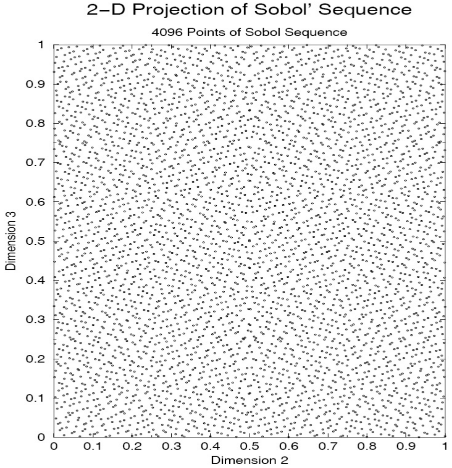
Derandomization and Applications

1. Given that a randomization leads to a family of QRNs, is there a best?
   - Must make the family small enough to exhaust over, so one uses a small family of permutations like the linear scramblings
   - The must be a quality criterion that is indicative and cheap to evaluate
2. Applications of randomization: tractable error bounds, parallel QRNs
3. Applications of derandomization: finding more rapidly converging families of QRNs

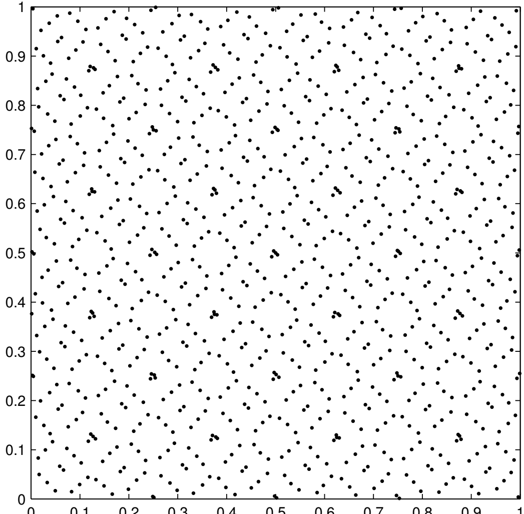
A Picture is Worth a 1000 Words: 4K Pseudorandom Pairs

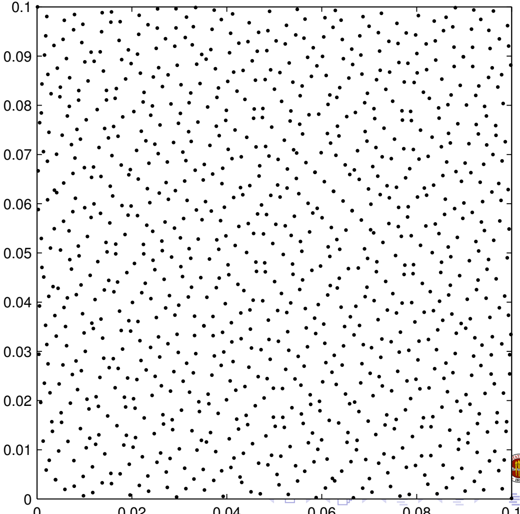


A Picture is Worth a 1000 Words: 4K Quasirandom Pairs



Sobol′ sequence





A New Hierarchy

1. New nomenclature thanks to Random123
   - Recurrence-based generators (most PRNGs)
   - Counter-based generators (ALL QRNGs)
   - Substreaming (cycle splitting)
   - Multistreaming (parameterization)
2. Permutations or bijections or scrambling
   - QRNGs all use scrambling now
   - PRNGs in Random123 depend on bijections
   - The study of random permutations and efficient implementation will be a major new thrust in RNG research

Conclusions and Future Work

1. `SPRNG` is at version 4 requires: some cleaning at the moment
2. Splitting `SPRNG` up
   - Base `SPRNG` in `C++`
   - Module for `MPI`
   - Module for `Fortran` users
   - Testing module
3. Other `SPRNG` updates
   - Spawn-intensive/small-memory footprint generators: MLFGs
   - "`QPRNG`"
   - Update test suite with `TestU01`

For Further Reading I

- [Y. Li and M. Mascagni (2005)]
  Grid-based Quasi-Monte Carlo Applications,
  Monte Carlo Methods and Applications, **11**: 39–55.
- [H. Chi, M. Mascagni and T. Warnock (2005)]
  On the Optimal Halton Sequence,
  Mathematics and Computers in Simulation, **70(1)**: 9–21.
- [M. Mascagni and H. Chi (2004)]
  Parallel Linear Congruential Generators with Sophie-Germain Moduli,
  Parallel Computing, **30**: 1217–1231.
- [M. Mascagni and A. Srinivasan (2004)]
  Parameterizing Parallel Multiplicative Lagged-Fibonacci Generators,
  Parallel Computing, **30**: 899–916.

For Further Reading II

[M. Mascagni and A. Srinivasan (2000)]
Algorithm 806: SPRNG: A Scalable Library for Pseudorandom Number Generation,
ACM Transactions on Mathematical Software, **26**: 436–461.