A Novel Approach to True Random Number Generation

Neel Mehta

5th year in PJAS

Introduction

- Random numbers have tremendous use in computing and science
  - **Cryptography – encryption algorithms, hashing**
  - Statistics – sampling, auditing, experimental design, data analysis
  - Computing – simulations, amortized sorting & searching
  - Physics – noise resonance studies
  - Mathematics – operations research
- Need high-quality random numbers to ensure security & accuracy

Problem

- Software alone can only generate *pseudo-*random numbers
- *Pseudo-random number*: contains patterns, lower entropy
- *Random number*: no patterns, high entropy
- Pseudo-random numbers are easier to predict
  - Can be exploited, compromising security
  - Can lead to inaccurate scientific results

Goals

- **Develop Python modules that provides truly random numbers**
- To compare the effectiveness of these modules against pseudo-random number generators
  - Effectiveness measures the randomness of the numbers generated
- Report which random number generators programmers should use under various circumstances

Software used

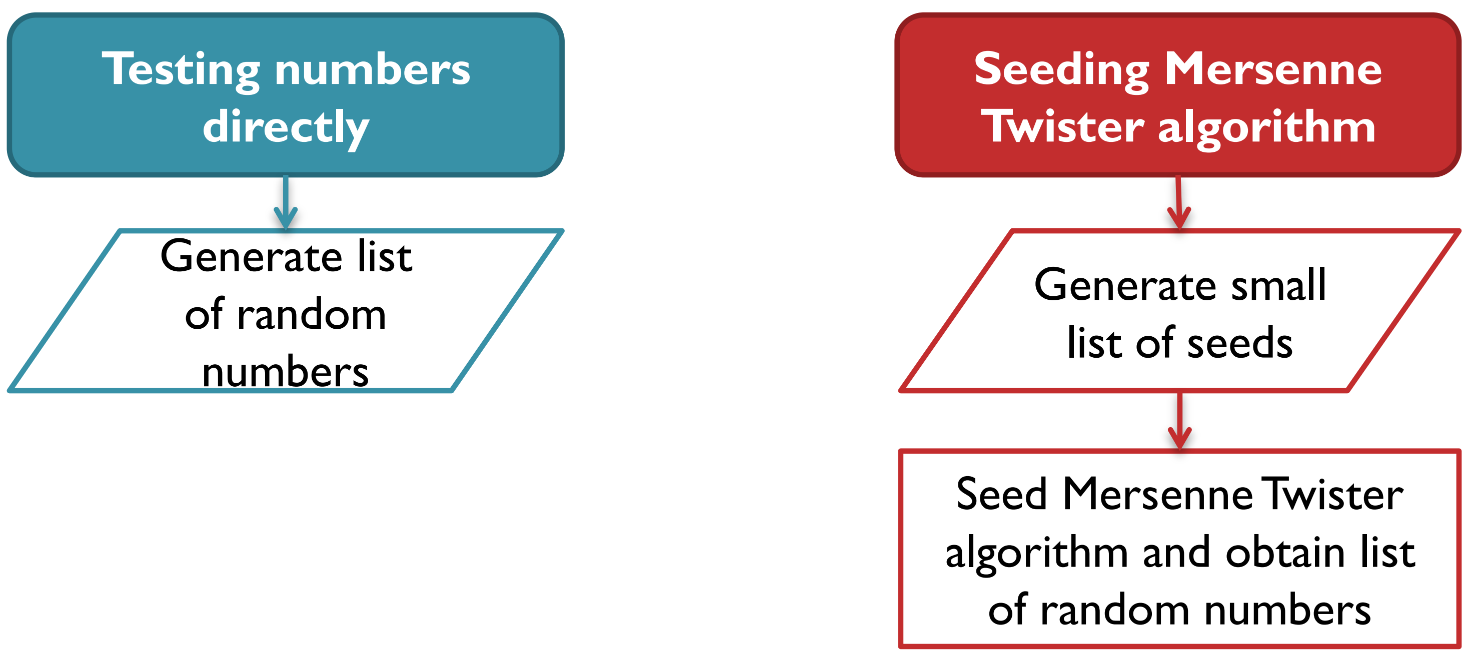
- Ubuntu 12.10
- Python 2.7.3
- Python Pip
  - Manages module installation & dependencies
- Python modules (must be downloaded):
  - **numpy**
    - Linear algebra, multi-dimensional arrays
  - **scipy**
    - Numerical integration, Fast Fourier transforms
  - BLAS
  - LAPACK
  - ATLAS

Engineering goals

- A number of modules must be created or implemented in Python:
- ***QuantumRandom* and *RandomDotOrg*** modules
  - Should be drop-in replacements for the *random* module
  - Should deliver truly random numbers
- **Mersenne Twister algorithm**
  - Uses a seed (source of entropy) to generate psedo-random numbers
- **NIST statistical test suite**
  - Statistical tests to determine randomness of numbers
- (Pseudo-)random number generators
  - Each should use a different technique to generate numbers
- Functions to interface with generators and test suite

Creating random numbers

- Goal: generate 10,000 random numbers to run tests on
- Two methods of creating random numbers:



Python's built-in PRNGs

- Built-in Python modules
- Used by most programmers
- Directly test the numbers generated

random

- Uses C implementation of rand() (pseudo-random number generator)

System Random

- Uses truly random numbers from hardware to generate pseudo-random numbers

Custom RNGs

- Interface with websites providing truly random numbers
- Directly test the numbers generated

Quantum Random

- Australian National University labs
- Measures quantum fluctuations in a vacuum

Random.org

- www.random.org
- Measures atmospheric noise

Seeding the Mersenne Twister algorithm (PRNG)

- The Mersenne Twister algorithm uses a *seed* value to generate pseudo-random numbers
  - Recognized as most efficient PRNG
- Find various sources of entropy to serve as seeds
- Test the numbers generated by the Mersenne Twister

Time
- Uses the milliseconds of the current system time as the seed

random
- Uses pseudo-random bits generated by the *random* module as the seed

SystemRandom
- Uses pseudo-random bits generated by the *SystemRandom* module as the seed

Binary conversion

- List of random numbers → binary string → bit sequence

To binary string

```
>>> dec2bin = lambda x: (dec2bin(x/2) + 
str(x%2)) if x else ''
```

Pad with zeroes

```
>>> padded_bin = 
raw_bin.zfill(length)
```

To bit sequence

```
>>> ss = [int(el) for el in binin]
```

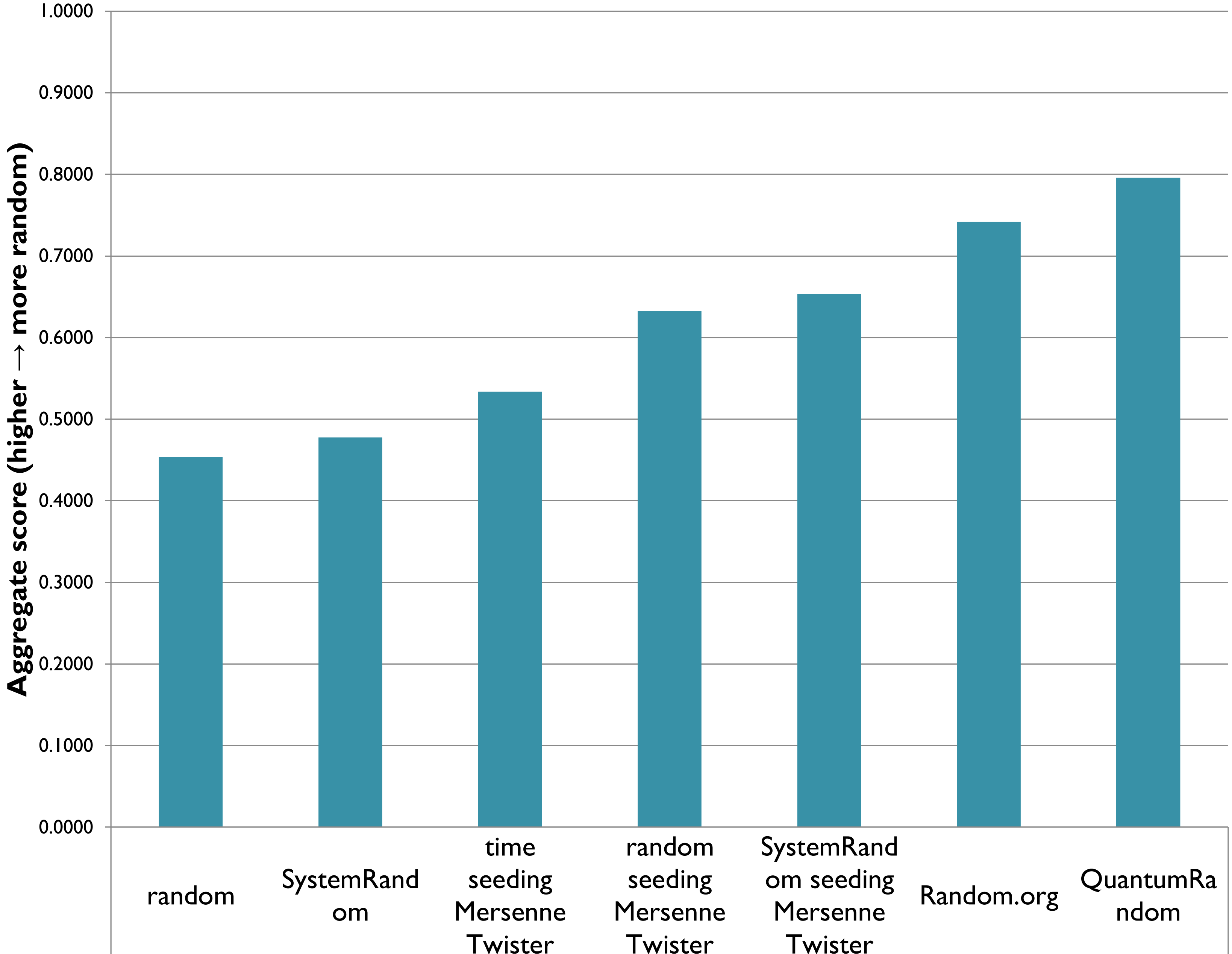
Randomness testing

- Run a battery of 13 statistical tests on bit sequences
  - Adapted from NIST's statistical test suite
- Tests analyze occurrences of 1's and 0's in a binary sequence
  - Compare these to expected values for a random sequence
- Tests return a score between 0 and 1
  - Higher score → more random (more entropy)

Randomness test battery

- Monobit frequency test
- Block frequency test
- Runs test
- Longest runs 10,000 test
- Binary matrix rank test
- Spectral test
- Non-overlapping template matching test
- Overlapping template matching test
- Maurer’s universal statistic test
- Cumulative sums test
- Cumulative sums test reverse
- Random excursions test
- Random excursions variant test

Aggregate randomness test battery scores



	random	SystemRandom	time seeding Mersenne Twister	random seeding Mersenne Twister	SystemRandom seeding Mersenne Twister	Random.org	QuantumRandom
Average	0.4537	0.4777	0.5338	0.6325	0.6533	0.7420	0.7960

Conclusions

- *random* and *SystemRandom*
  - Ineffective
- Mersenne Twister
  - More effective than built-in PRNGs
  - Using time as a seed is ineffective since time is a weak source of entropy
- The Mersenne Twister improves randomness of already-entropic data sets
  - Seeding with *random* yielded scores **39.4% higher** than *random* alone
  - Seeding with *SystemRandom* yielded scores **36.7% higher** than *SystemRandom* alone

Conclusions cont'd.

- Python's built-in modules and the Mersenne Twister only yield *pseudo*-random numbers
- *QuantumRandom* and *Random.org* yield high-quality truly random numbers
  - *QuantumRandom* scores **75.4% higher** than *random*
  - *Random.org* scores **63.5% higher** than *random*

Drop-in replacement

- Very easy for programmers to include these modules in their projects
- Trivial code changes
- Same API as built-in *random* module
  - Same *random* functions such as random(), randrange(), getrandbits(), etc.

random (built-in)

```
>>> import random
>>> num = random.random()
>>>
```

Quantum Random

```
>>> from quantumrandom import QuantumRandom
>>> random = QuantumRandom()
>>> num = random.random()
```

Random.org

```
>>> from randomdotorg import RandomDotOrg
>>> random = RandomDotOrg()
>>> num = random.random()
```

Benefits & Usage

- *QuantumRandom* and *Random.org* provide 60%+ stronger random numbers & are very easy to implement
- If internet is available, use *QuantumRandom*
  - *QuantumRandom* and *Random.org* require internet
- If internet is not available, use *SystemRandom* to seed the Mersenne Twister algorithm

Applications

- Truly random numbers are crucial in cryptography
- Need strong random number as key for encryption algorithms (e.g. md5, SHA-1)
  - Seed harder to guess → **more security**
- Using these modules internet programmers can easily boost security
  - Authentication, electronic commerce, *https* protocol secure communications

Extensions

- Develop similar modules for C#, Java, PHP, etc.
- Compare speeds of the PRNGs and RNGs
- Compare modules to hardware RNGs