# Pseudo-random Number Generation

# Random Numbers

- All the code we've written is deterministic - it does the same thing every time.

- Some things we might want to simulate are apparently not so - nuclear decay, "noise" on signals, ages of a person you bump into.

- Sometimes we also just want to generate "different values", to explore the phase space of a problem.

- So, how do we generate "random variation"?

# Pseudo Random Numbers

- Most of the time, we don't really care about having truly "random" sequences.
- What we want are sequences which have statistical properties of a random sequence:
    - each value is hard to predict from the ones before it (**uncorrelated**)
    - the probability of any value is the same (**uniform**)
- We can imagine a *pseudorandom* number generator, which produces such a sequence, but deterministically, from some kind of internal state.

# A (bad) example PRNG

0**54**1

**54**$^2$ = 2916

2**91**6

**91**$^2$ = 8281

8**28**1

**28**$^2$ = 0784

0**78**4

**78**$^2$ = 6084

6**08**4

**08**$^2$ = 0064

0**06**4

**06**$^2$ = …

…

- John Von Neumann wrote this PRNG in 1949.
- We start with a "seed" value: here 0541.
- Each value in the sequence is the square of the middle digits of the value before it.

# PRNGs - flaws

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0**00**0 | 0**00**1 | 0**00**2 | 9**00**8 | | 1**45**3 | | 1**24**3 |
| 0**00**0 | 0**00**0 | 0**00**0 | 0**00**0 | | 2**02**5 | | 0**57**6 |
| 0**00**0 | 0**00**0 | 0**00**0 | 0**00**0 | | 0**00**4 | | 3**24**9 |
| 0**00**0 | 0**00**0 | 0**00**0 | 0**00**0 | **...** | 0**00**0 | **...** | 0**57**6 |
| 0**00**0 | 0**00**0 | 0**00**0 | 0**00**0 | | 0**00**0 | | 3**24**9 |
| 0**00**0 | 0**00**0 | 0**00**0 | 0**00**0 | | 0**00**0 | | 0**57**6 |
| ... | ... | ... | ... | | ... | | ... |

- Badly designed PRNGs can have unexpected behaviour for some seed values.

- Here, starting with any sequence containing 00, 45, 24 (or 57) generates a very predictable ("non-random") sequence!

# PRNGs - flaws

- Even much more sophisticated PRNGs, which don't have obvious flaws, can have more subtle issues.
- For example: IBM's **RANDU** PRNG, which was widely used in the 1960s.
- Although the output "looked random", it actually had some extremely strong correlations, which made the output untrustworthy.
- Many millions of hours of simulations had to be redone once this was discovered!
- (Plus, obviously, no PRNG can ever be *truly* uncorrelated, they just approximate this state.)

# PRNGs - advantages

- Because a PRNG generates a sequence from an internal state, which is set by a seed value…
- …the same seed will always give the same sequence.
- So, we can directly compare the result of changes to code, by running it with the same seed as before.

# PRNGs - advantages

- The alternative to PRNGs is to try to "collect" randomness from the real world.
- "Hardware Random Number Generators" exist, but must gather truly random sources - radioactive decay, or shot noise in electronic circuits - which limits the rate at which they can work.
- PRNGs are usually faster than HRNGs; and you can always rely on them being available!

# PRNGs in C

- The C Standard Library provides some functions for generation of pseudorandom sequences.
- These functions are prototyped in `stdlib.h`, so you need to `#include` it to use them.
- `void srand(unsigned int seed)`
  - set seed for PRNG to provided seed value (default is `0`)
- `unsigned int rand(void)`
  - get next value from PRNG (integer in range `0` to `RAND_MAX`)
- `RAND_MAX`
  - `#define`d in `stdlib.h`: the largest value that `rand()` will ever return.

# Example - continuous uniform variates

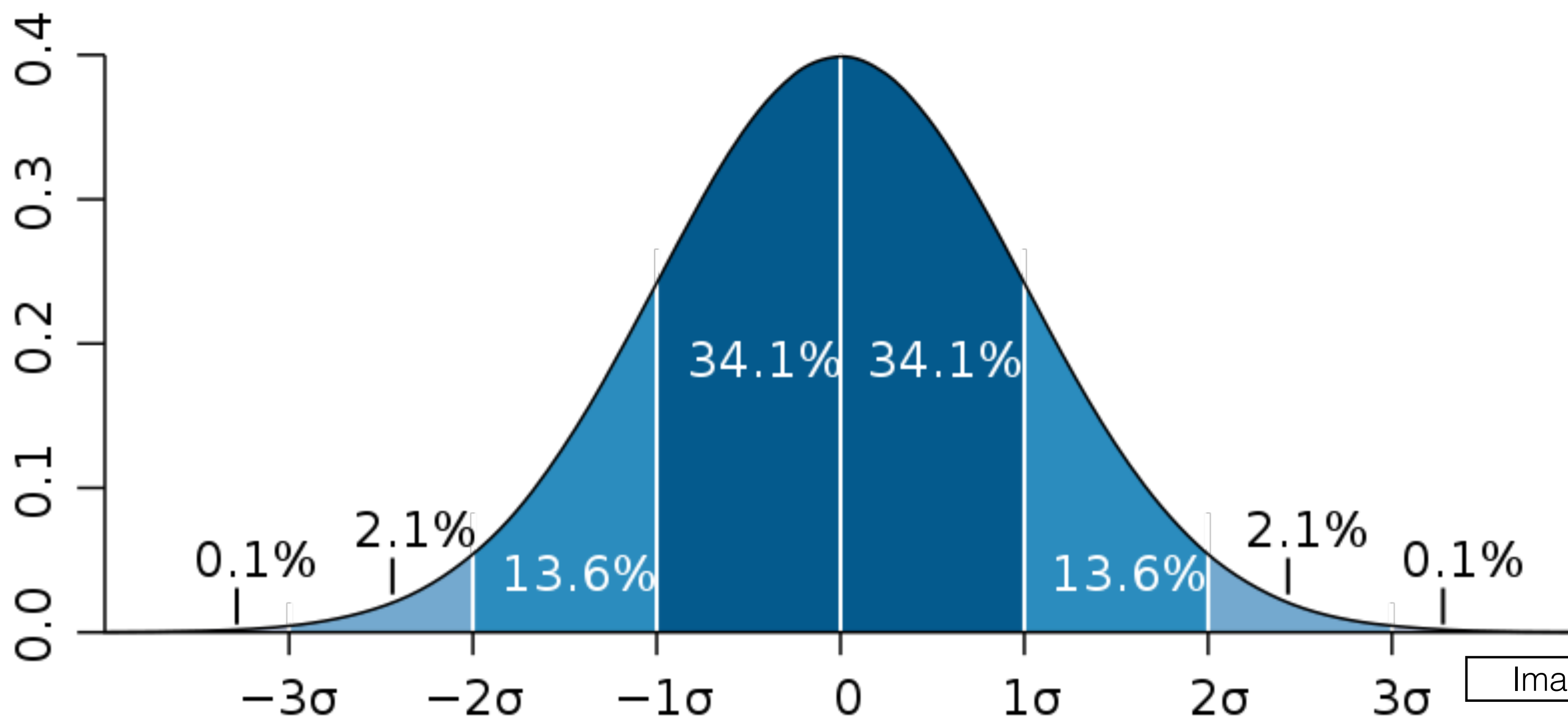
```
double zero_to_one(void) {
 return rand()/ (double) RAND_MAX;
}
```

`rand()` will only generate values from
**0** to **RAND_MAX**
If we re-scale those values by dividing by **RAND_MAX**,
then we will clearly get a number between **0** and **1**.
(Remember to cast to `double`, to avoid integer division)

# Example (approximation of Gaussian / Normally distributed variates)

- Sometimes we want a *non-uniform* distribution.

- A common distribution we might want is the Gaussian distribution (the Normal distribution with $\mu = 0$, $\sigma = 1$).






There are efficient, exact, methods to produce arbitrary probability distributions, but these are not in scope for this course.

# Example (approximation of Gaussian / Normally distributed variates)

- The **Central Limit Theorem** states that the distribution of *sums of a large number of samples from any distribution* itself approximates the Gaussian distribution, as the number of samples summed over tends to infinity.

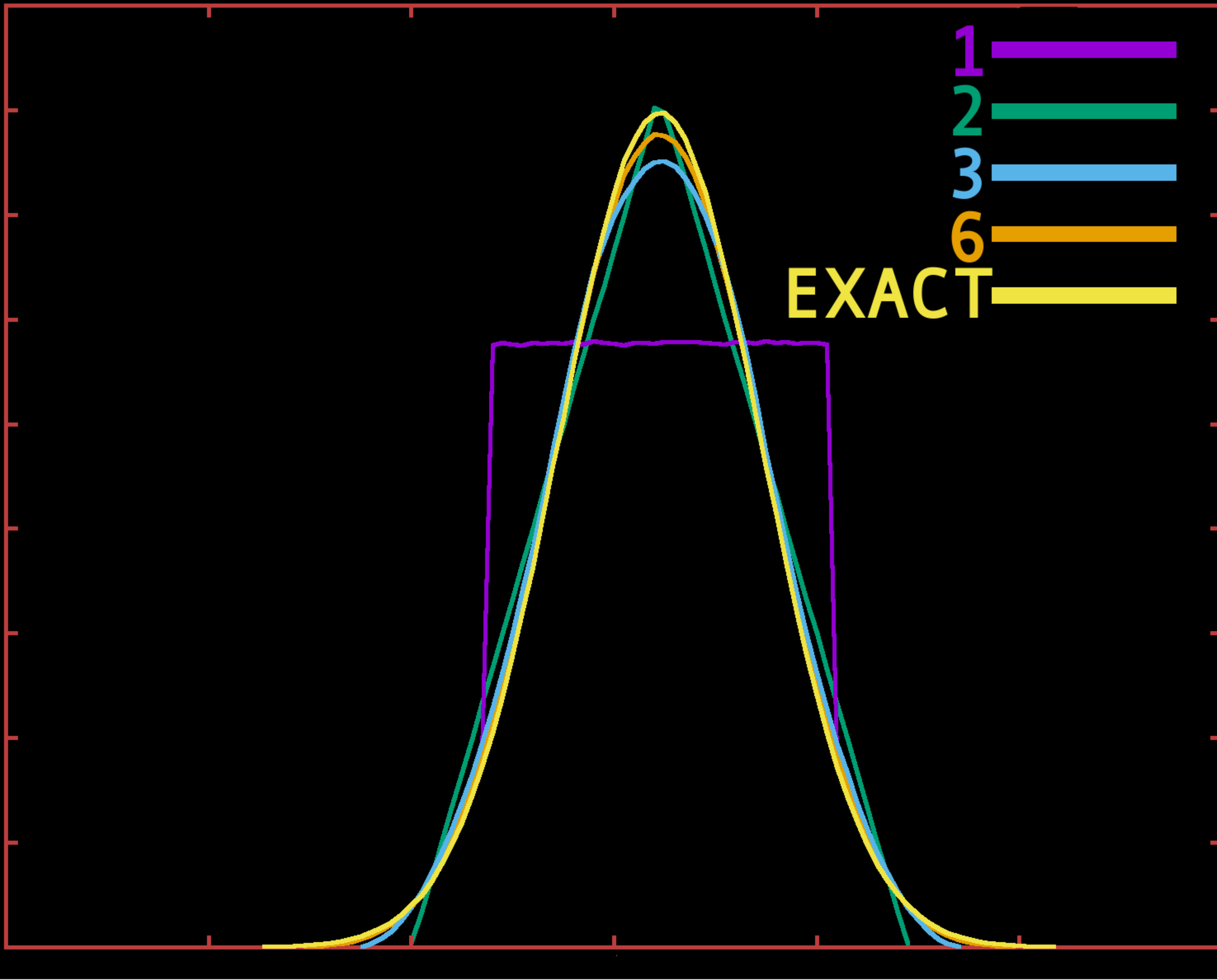
- So, if we average over a large number of samples from a uniform distribution - for example, calls to `rand()`, and return the result…

- …the resulting sequence will be an approximation to the distribution we want.

# Example (approximation of Gaussian / Normally distributed variates)

```
#include <math.h>

//number of uniform deviates to add for our approximation. Larger is better, but slower
#define GAUSS_PREC 6

double approx_gauss(void) {

  double variate=0.0;

  for(int i=0;i<GAUSS_PREC;i++) {

    variate += rand() /(double) RAND_MAX;

    }

    //rescale variance and mean appropriately for Gaussian
    // distribution is still Normal without this, just centred on GAUSS_PREC/2
    //  with variance GAUSS_PREC / 12.0

    return (variate - 0.5*GAUSS_PREC) / sqrt(GAUSS_PREC/12.0);

  }
```

This method takes GAUSS_PREC uniform variates to make 1 approximate Gaussian variate. More efficient, exact - such as the Box-Müller and Marsaglia's Polar - methods exist, but are not in scope for this course.



1
2
3
6
EXACT

# Example - seeding rand from system time.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(void) {
 /* time(TIME) returns the current time as seconds
  since the "Unix Epoch" (midnight 1 Jan 1970).
 TIME can be NULL (or a pointer to a place to store
 the result, instead of returning it) */

 srand(time(NULL));

 //print 100 pseudo random numbers in sequence
 for(int i=0; i<100; i++) {
  printf("%d\n", rand() );
 }

 return 0;
}
```

`time.h` **header for time related functions in libc**

`time(NULL)` **returns a number which will be different every second, "good enough" for the seed to be different on each execution**

# Limitations of rand()

- `rand()` is convenient, being in the C Standard Library, but it has limitations.

- As `srand()` takes an unsigned int as a seed, there are only `UINT_MAX` possible sequences that rand can produce (one per seed).

- The C Standard doesn't require `rand` to have a particularly long sequence length (for portability). In some versions of the library, `rand` can have quite short repeat lengths, or fail some statistical tests.

# Beyond rand()…

- For serious work requiring pseudorandom sequences with good statistical properties, you should use a dedicated PRNG from a library designed for this.
- Mersenne Twister and WELL, for example, are two widely used and very good PRNGs which you should consider.
- (For cryptography, even these are not sufficient - consider special cryptographically strong PRNGs, or actually taking random numbers from outside machine.)

# References

- Mersenne Twister
- http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
- WELL
- http://www.iro.umontreal.ca/~panneton/WELLRNG.html
- Example use for PRNGs:
- http://moodle2.gla.ac.uk/mod/resource/view.php?id=149996

# An aside on pointers.

- You can have pointers to anything.
- This includes *functions*.
  - functions exist in memory.
  - And they have a type (their signature).

```
int myfunc (int y, double z); // a function prototype

int (*myfunc_ptr) (int, double);
//myfunc_ptr is a pointer to functions with myfunc's signature.
// it *CANNOT* safely point to functions with a different signature.

myfunc_ptr = myfunc; //point myfunc_ptr at myfunc
myfunc_ptr(1,2.0); //the same as myfunc(1,2.0)
```

# Using function pointers to make a generic test

```
need the ( ) around the * so compiler knows we mean "pointer to function
which returns int" and not "function which returns pointer to int"
```

```
double mean( int (*f) (void) ) {
 //f here is a "pointer to a function which takes void
 //                                       and returns double"
 double accum = 0;
 for (int i=0; i<50; i++) {
  accum += f(); //you can call the function pointer with () as normal
  }
  return accum/50;
}
```

```
a function name without the () is a pointer to it
(just like the array name without the [] is a pointer)
```

```
double mu = mean(rand); //return mean of 50 invocations of rand()
```