Note: please compile the relevant .c source file with “gcc <*source.c*> ranlxd.c -lm”.

Task 1:

Which random number generators can be considered good/bad according to this set of tests (name a few).

See files *dieharder_** in the *Task 1* folder for the raw results.

One of the worst random number generators I tested with DieHarder was RANDU.
RANDU is an example of a linear congruential generator. It is defined by the recurrence $V_{j+1} = (65539 * V_j) \bmod 2^{31}$, where V_0 is an odd number.
In DieHarder it fails the majority of tests, and for the tests it does pass many of them have a “weak” grade.
Numbers generated by RANDU are highly correlated, indeed when generating triplets for points in 3D space all points will fall into one of fifteen planes[1].

MT19937 is one of the best random number generators I tested with DieHarder.
It is a version of the Mersenne Twister random number generator.
It has a very large period (2^{19937}-1) and is 623-dimensionally equidistributed. This makes it suited for use in scientific simulations[2].
It passed all DieHarder tests, though it received a “weak” grade in two of them.

One random number generator that seems good is RANLUX.
RANLUX uses a subtract with carry Fibonacci recurrence, but chooses sequences of numbers separated by several correlation times. These resulting numbers are highly decorrelated[3].
It failed two tests but passed all others (with a weak score in five).

Explain one of the tests in more detail.

Dieharder uses a modified birthday spacings test[4] that generates n birthdays across an m day year. Knuth describes the algorithm for a normal birthday spacings test as follows[5]:

1. Generate birthdays $Y_1,..,Y_n$ where $0 \le Y_k \le m$
2. Sort the birthdays into nondecreasing order $Y_1 \le \ldots \le Y_n$
3. Define n spacings such that $S_1 = Y_2 - Y_1$, …, $S_{n-1} = Y_n - Y_{n-1}$, $S_n = Y_1 + m - Y_n$
4. Sort the spacings into order $S_1 \le \ldots \le S_n$
5. Let R be the number of equal spacings, such that it is the number of indices j where $1 \le j \le n$ and $S_j = S_{j-1}$.

---

[1] http://physics.ucsc.edu/~peter/115/randu.pdf pages 1 and 2
[2] https://software.intel.com/en-us/node/590405
[3] http://luscher.web.cern.ch/luscher/ranlux/
[4] DieHarder help option, “dieharder -d 0 -h”
[5] Donald Knuth, The Art of Computer Programming, Volume 2, 3rd Edition, pages 71-72

The test is repeated a number of times (100 by default in DieHarder) and a chi-square test is performed. Repeated intervals should have a Poisson distribution if the generator is random enough.
This test is notable because lagged Fibonacci generators generally fail this test even if they pass other tests.

Task 3:

Write a program which uses a random number generator (i.e. continuous RV with uniform distribution on [0,1)) to produce a binomial random variable.

See file *binomial_rv.c*

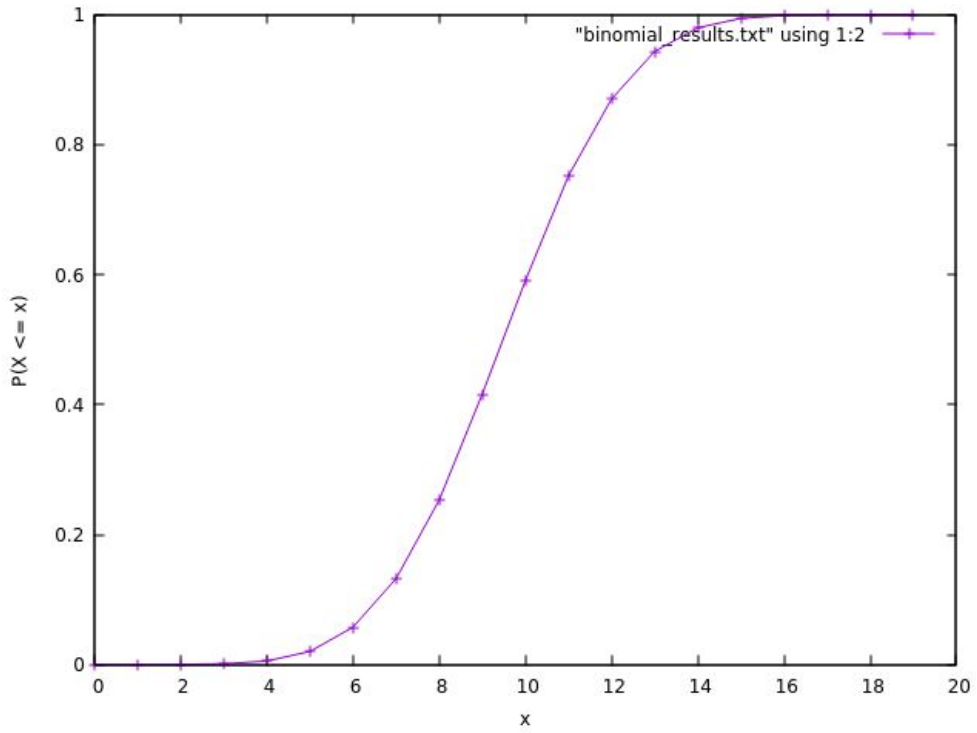
Rather than use the inverse transform method to generate binomial random variables I instead used a method that counts the number of successes in *n* independent trials[6].
RANLUX was used to generate random variables in the range [0, 1). Each of these values was checked if it was < *p*, and the trial was recorded as a success if so.

Choose values for *n* and *p* and plot the empirical CDF for your binomial distribution. Does it approach the expected CDF?

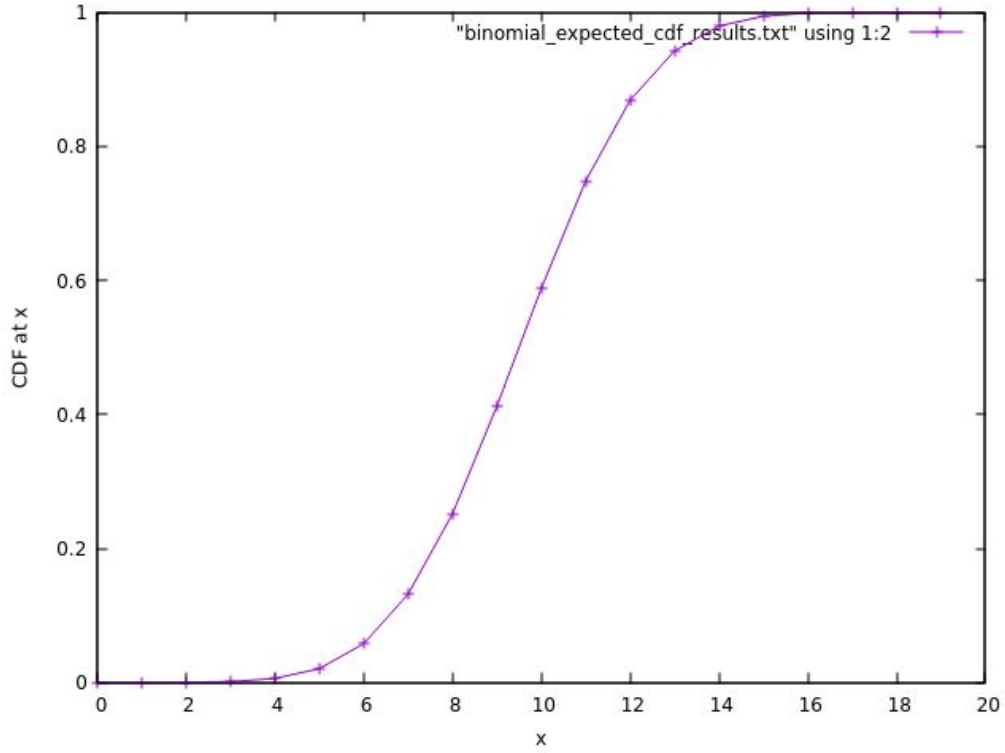
See files *binomial_cdf_calc.c*, *binomial_empirical_cdf_results.txt*, *binomial_empirical_cdf.png*, *binomial_expected_cdf_results.txt* and *binomial_expected_cdf.png*,

To generate the empirical CDF values I generated 10'000 binomial random numbers with p=0.5 and n=20. I counted the occurrence of each value in the range [0, 19] and then plotted the CDF using these values.
I wrote a program to generate the expected values using the binomial CDF[7], then plotted the expected CDF values using these values
The two plots can be seen below or in the specified .txt and .png files. As can be seen my empirical results approach the expected results.

[6] Sheldon Ross, Simulation, 3rd Edition, Page 52
[7] http://www.itl.nist.gov/div898/handbook/eda/section3/eda366i.htm



Binomial empirical CDF with n=20 and p=0.5



Binomial expected CDF with n=20 and p=0.5

Task 4:

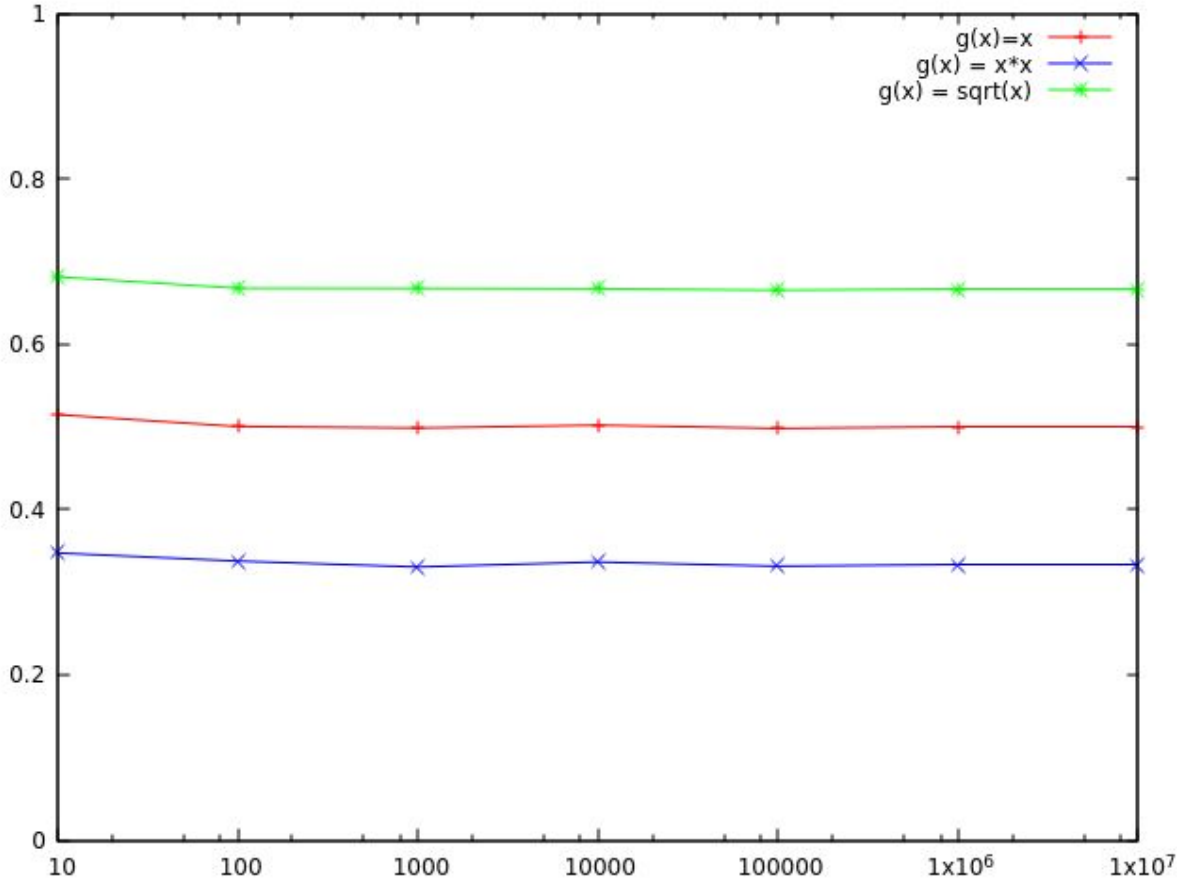
Estimate the integrals of the functions g(x) = x, g(x) = x * x and g(x) = sqrt(x) by averaging over N samples.

See files *integral_estimation.c* and *integral_estimation_results.txt*.

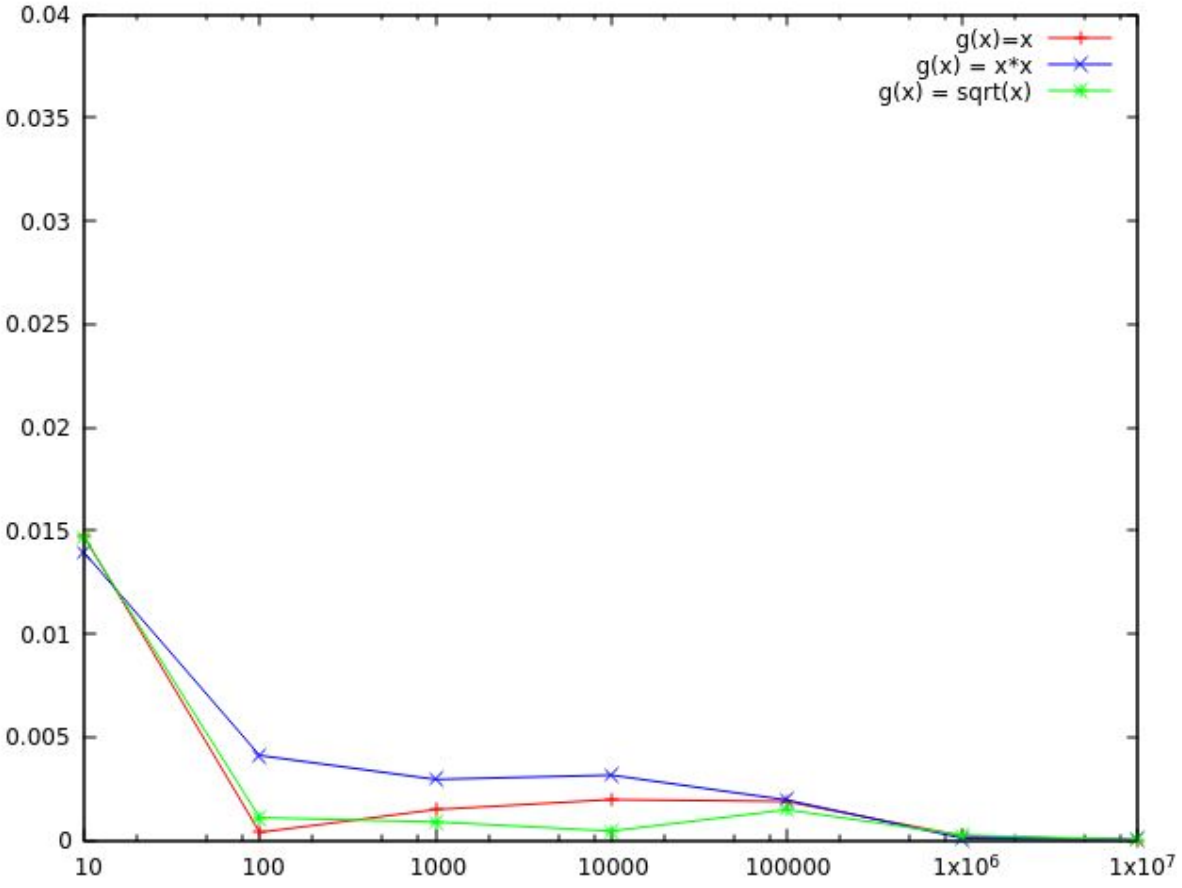
Produce a plot of the results vs. N

See file *integral_estimation_result.png*, *integral_estimation_result.txt*, *integral_estimation_absolute_error.png* and *integral_estimation_absolute_error.png*.
The image files are included below.

In addition to plotting the results vs. N I also plotted the absolute error vs. N.
It's interesting to note that at some as N increases the absolute error sometimes increases, or decreases rapidly. I believe this is caused by the random numbers picked during that particular run.



N vs result



N vs absolute error

How large does N have to be to reproduce the analytic result to 2, 3, or 4 digits, respectively?

Function	**Reproduced digits**	**N**
g(x) = x	1 digit	10
	2 digits	1’000
	3 digits	1’000’000
	4 digits	10’000’000
g(x) = x^2	1 digit	10
	2 digits	1’000
	3 digits	100’000
	4 digits	10’000’000
g(x) = sqrt(x)	1 digit	10
	2 digits	10’000

	3 digits	100’000
	4 digits	10’000’000

Can you estimate how large N would have to be to obtain 8 digit accuracy?

Ideally the error is proportional to *(1 / sqrt(N))*[8].
Therefore to decrease the error by a factor of 10 (reproduce 1 more digit) N must be 100 times larger.
Combining this with my above result that reproducing one digit is achieved with N=10, I estimate that N must be around the order of 10^{15} to reproduce 8 digits.
However it is important to note that the error can vary depending on the exact random numbers are that generated and the exact function that is being integrated.

Estimate Var[g(x)] = E[(g(x)−μ)2]. How is the variance related to the accuracy of your previous estimates for E[g(x)]?

See files *variance_estimation.c* and *variance_estimation_results.txt.*

I slightly modified my integral estimation program to instead estimate the variance using $Var[g(x)] = E[g(x)^2] - \mu^2$ [9].
The variance was estimated using different values of N. For the purposes of space I will not include all results in the below table, but only N=10, 10’000 and 10’000’000. The full results can be found in *variance_estimation_results.txt.*
From the results we can see that as N increases the estimated variance becomes smaller and approaches the actual variance for these choices for *g(x)*.
As the variance becomes smaller the estimated integration result becomes closer to the analytical result.

Function	**N**	**Estimated variance**
g(x) = x Actual variance: 1/12 (0.0833333…)	10	0.048718
	10’000	0.083047
	10’000’000	0.083323
$g(x) = x^2$ Actual variance: 4/45 (0.0888888…)	10	0.068358
	10’000	0.089061
	10’000’000	0.088912

8
http://pitp.physics.ubc.ca/confs/sherbrooke/archives/pitpcifar_sherbrooke2008_intro_mc_troyer.pdf page 8
9 Sheldon Ross, Simulation, 3rd Edition, Page 13

g(x) = sqrt(x) Actual variance: 1/18 (0.0555555…)	10	0.012841
	10’000	0.055096
	10’000’000	0.055522

One may be tempted to estimate π numerically by choosing g(x) = $\frac{2}{\sqrt{1-x^2}}$.
Why is this not such a great idea? (Hint: what happens to the variance?)

See files *variance_estimation.c* and *variance_estimation_results.txt.*

I modified my *variance_estimation.c* function to use this choice of *g(x)*. The results show that the estimated variance starts out high, decreases, but then starts increasing again. As the variance increases and is large this choice of *g(x)* will not give a good estimate of π.

N	**Estimated variance**
10	16.742633
100	13.874154
1’000	5.416537
10’000	9.822583
100’000	17.193613
1’000’000	18.946556
10’000’000	24.243550