Monte Carlo methods in computational physics

Casey Daniel and Chris Deimert
February 14, 2015

1 Introduction

In this report, we explore a number of Monte Carlo numerical methods. Monte Carlo methods use pseudorandom numbers to explore complicated systems. These methods tend to converge slowly for simple problems, but can be very efficient for complex problems: especially those with a large number of variables.

In Section 2, we look at methods for generating pseudo-random numbers. These are at the core of any Monte Carlo method. In Section 3, we apply Monte Carlo methods to a real physics problem: photons moving through a slab of material. In Section 4, we look at a classic Markov Chain example: a string of sunny and rainy days in which the weather of the next day depends on the weather of the current day. In Section 5, we use another Markov Chain method: the Metropolis-Hastings random walk. This allows us to generate samples from complicated probability distribution functions. In Section 6, we use the simulated annealing algorithm to find the global maxima of functions. The simulated annealing algorithm is a robust method for finding global maxima of complicated functions.

2 Pseudo random numbers

Monte Carlo methods typically rely heavily on our ability to generate a large quantity of random numbers. We cannot, of course, generate truly random numbers on a deterministic computer, so we have to approximate them with pseudo-random numbers (PRN's). Having a reliable PRN generator is thus a key prerequisite to any Monte Carlo method. In this section, we will study the linear congruential method (LCM) for generating PRN's. The LCM generates a sequence of pseudo-random integers with the nth integer given by

$$I_n = (AI_n + C) \mod M \tag{1}$$

(I_0 is called the "seed".) We can then generate a sequence $x_n = I_n/M$ of pseudo-random real numbers with $0 < x_n < 1$.

A Fortran 90 module was created to implement the LCM generator and can be seen in Section 8.1. This code was tested in a number of ways, the code for which is in Section 8.2 and Section 8.3.)

The first, simplest test used $I_0 = 3$, $A = 7$, $C = 0$, and $M = 10$. The result is a repeating sequence of numbers:

$$x = 0.1, 0.7, 0.9, 0.3, 0.1, 0.7, 0.9, 0.3, \ldots \tag{2}$$

This sequence repeats after only 4 numbers, demonstrating why small values of M are a poor choice. The sequence will repeat after at most M numbers, so we must select a high value of M in order to obtain a long sequence (though high values of M do not *guarantee* a long sequence).

One useful way to test a pseudo-random number generator is to look at the correlation between each generated number and the number that preceded it. These can be plotted in a 2D scatter plot: patterns on the scatter plot imply that the pseudo-random number generator is actually predictable and thus not very random. The correlation plots for three different sets of coefficients are shown in Figures 1, 2, and 3. Note that a seed of $I_0 = 1$ was used in each case. It is found that the third combination of coefficients provides the best pseudo-random number generator.

Another test of pseudo-random number generators that can be performed is a χ^2 test. This is used to determine

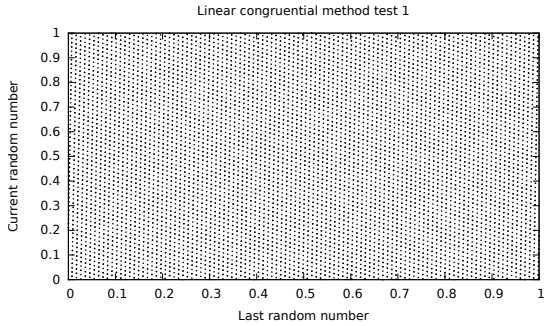

Figure 1: The correlation plot for $A = 106$, $C = 1283$, and $M = 6075$. It is seen that the correlation plot is strongly patterned, meaning that this is a relatively predictable sequence of numbers. Thus, this choice of LCM coefficients is a poor one.

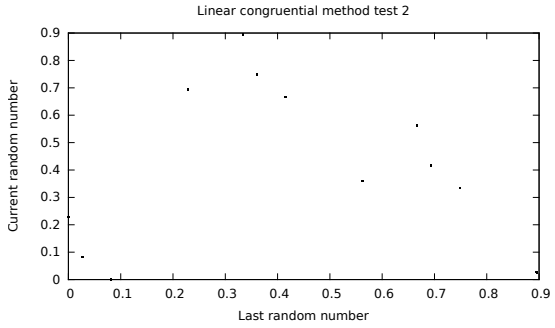

Figure 2: The correlation plot for $A = 107$, $C = 1283$, and $M = 6075$. It is seen that the correlation plot is even more strongly patterned than the last one. Thus, this is a relatively predictable sequence of numbers and this choice of LCM coefficients is poor. This also demonstrates how small changes in the coefficients lead to drastic changes in the effectiveness of the LCM generator. (The coefficients here are the same as the last set except that A has been increased by 1).

our statistical confidence in the fact that the pseudo-random numbers are evenly generated.

Using the LCM generator with a seed of $I_0 = 1$ and coefficients from the third test above, 100 numbers were generated. Similarly, 100 numbers were generated using the gfortran generator with a seed based on the computer time. The distribution of results are shown below.

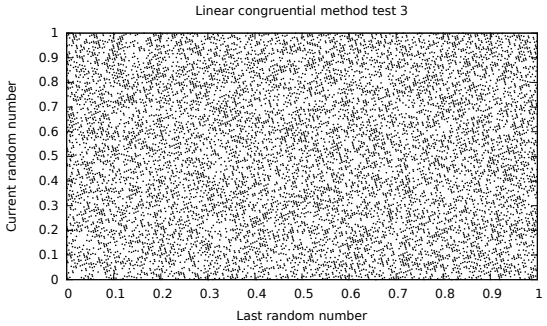

Figure 3: The correlation plot for $A = 1103515245$, $C = 12345$, and $M = 32768$. It is seen that the correlation plot is very weakly patterned, meaning that this is a relatively unpredictable sequence of numbers. Thus, this choice of LCM coefficients is a good one.

Interval	Upper limit	LCM	gfortran	Exp.
1	0.1	6	6	10
2	0.2	11	16	10
3	0.3	12	9	10
4	0.4	9	6	10
5	0.5	10	11	10
6	0.6	9	5	10
7	0.7	8	12	10
8	0.8	10	9	10
9	0.9	11	13	10
10	1.0	14	13	10

The χ^2 values for the LCM and gfortran generators are:

$$\chi^2_{\text{LCM}} = 6.80 \tag{3}$$

$$\chi^2_{\text{gfortran}} = 8.40 \tag{4}$$

These were calculated using

$$\chi^2 = \sum_{i=1}^{10} \frac{(O_i - E_i)^2}{E_i} \tag{5}$$

where O_i is the observed value in the ith row of the table above, and E_i is the expeced value.

It should be noted that the χ^2 values change significantly depending on the seed used, though the ones given here are fairly typical. The LCM χ^2 varies from about 2 to about 10, and the gfortran χ^2 varies from about 3 to about 18.

In this case, our null hypothesis is that the generated random numbers which are independent and uniformly distributed between 0 and 1. We have 9 degrees of freedom in this case, which means that the critical χ^2 value for 95% confidence is $\chi^2_{\text{crit}} = 16.92$. Thus we cannot confidently reject the hypothesis that the pseudo-random number generators are uniformly distributed and independent.

It should be noted that if the number of trials is increased from 100, the χ^2 for the LCM decreases while the χ^2 for the gfortran remains approximately the same. We would expect the χ^2 to decrease for truly random numbers (from the law of large numbers), so this indicates a problem with the gfortran generator.

A final test we can perform is the auto-correlation test for independence. The autocorrelation A_k of a sequence of random variables represents the correlation between the tth number and the $(t+k)$th random number. It is given by

$$A_k = \frac{\sum_{t=1}^{N-k} \big(x_t - \langle x \rangle\big)\big(x_{t+k} - \langle x \rangle\big)}{\sum_{t=1}^{N-k} \big(x_t - \langle x \rangle\big)^2} \tag{6}$$

The autocorrelation sequences were calculated for both the LCM random generator (with the third set of coefficients) and the gfortran random number generator. These are plotted in Figure 4 and Figure 5.

We want the autocorrelation to be as close to zero as possible for a pseudo-random number generator. Small autocorrelations indicate that the random numbers are independent of each other. It is seen that both generators are pretty good in terms of independence, but the LCM performs slightly better. The gfortran autocorrelation sequence jumps closer to 1 towards the tail end, indicating a lack of independence.

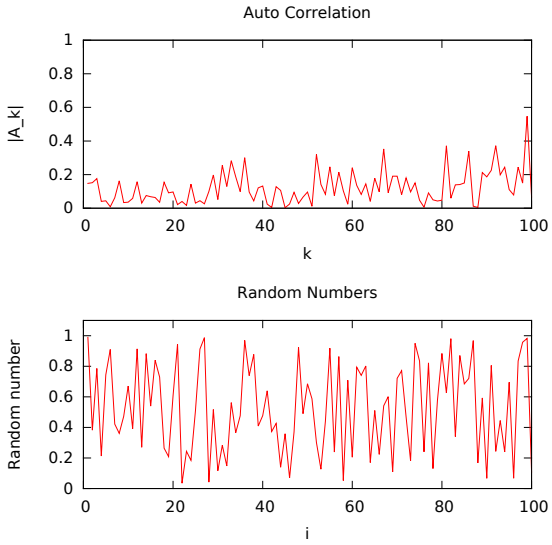

Figure 4: The autocorrelation sequence for an LCM generator with $A = 1103515245$, $C = 12345$, $M = 32768$, and $I_0 = 1$.

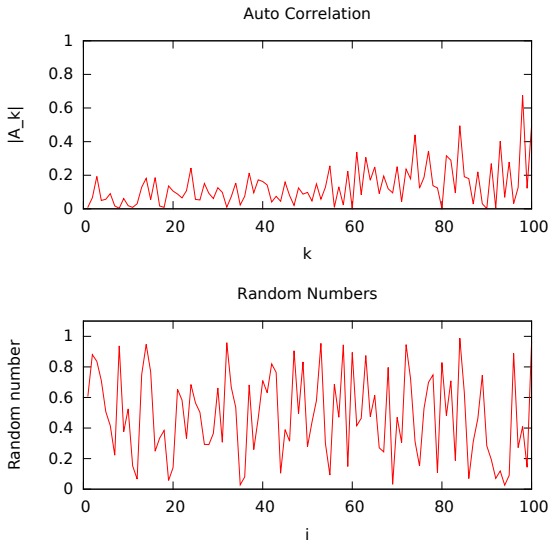

Figure 5: The autocorrelation sequence for the standard gfortran pseudo-random number generator.

3 Radiative transfer

In this section we apply Monte Carlo methods to the problem of radiative transfer. Basically, the process is:

- Generate a number of photons at the bottom of a slab.
- Move each photon forward in a random direction by a random distance. At each step, there is a probability that the photon is absorbed.
- Repeat until all the photons have either exited the slab or been absorbed.

Most random number generators only create numbers uniformly distributed between 0 and 1, so we need to determine how to generate the required random variables in terms of a uniformly distributed random variable U.

First, we look at the optical depth, τ. It is exponentially distributed with probability distribution function [1]:

$$f_\tau(\tau') = e^{-\tau'} \tag{7}$$

The cumulative distribution function is then

$$F_\tau(\tau') = \int_0^{\tau'} f_\tau(s)\,ds \tag{8}$$

$$F_\tau(\tau') = 1 - e^{-\tau'} \tag{9}$$

Solving for τ', we get

$$\tau' = -\ln\left(1 - F_\tau(\tau')\right) \tag{10}$$

Finally, using the fundamental principle, we find that we can generate optical depths using

$$\tau = -\ln(1-U) \tag{11}$$

Note that the actual distance moved by the photon is not τ, but $L = \frac{\tau z_{\max}}{\tau_{\max}}$, where $z_{\max}$ is the thickness of the slab, and $\tau_{\max}$ is 20 in this case. This is shown by:

$$\tau = \int_0^L n\sigma\,ds = Ln\sigma \tag{12}$$

and

$$\tau_{\max} = \int_0^{z_{\max}} n\sigma\,ds = z_{\max} n\sigma \tag{13}$$

Dividing the first equation by the second gives

$$L = \frac{\tau z_{\max}}{\tau_{\max}} \tag{14}$$

When generating random directions, we want to ensure that each differential solid angle $d\Omega = \sin\theta\,d\theta\,d\phi$ is equally likely. Thus is made easier by defining $\mu = \cos\theta$ so that $d\Omega = d\mu d\phi$. Then, μ is a random variable distributed uniformly on $[-1,1]$, and ϕ is a random variable distributed uniformly on $[0,2\pi]$. So it is trivial to generate μ and ϕ in terms of the unit, uniform random variable U:

$$\mu = 2U - 1 \tag{15}$$

$$\phi = 2\pi U \tag{16}$$

To calculate θ we simply use

$$\phi = \cos^{-1}(\mu) = \cos^{-1}(2U-1) \tag{17}$$

To test the effectiveness of the fundamental principle, a large number of unit uniform random numbers (U) were generated and used to create distributions of τ, ϕ, μ, and θ. These histograms are shown in Figures 6, 7, 8, 9. In each case, 10^6 trials were run, and the histograms were normalized so that they correspond to an estimated probability distribution function. In each case, the histogram is exactly as expected, confirming our use of the fundamental principle.

The final piece needed to perform this Monte Carlo experiment is the absorption probability, which is given in terms of the absorption and scattering cross sections:

$$P_s = \frac{\sigma_s}{\sigma_s + \sigma_a} \tag{18}$$

[1] A note on notation: the prime is used here to distinguish between a random variable (τ) and the value it takes on (τ')

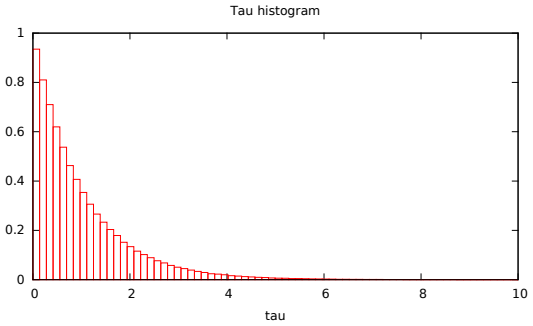

Figure 6: Distribution of τ generated using a unit uniform random variable U with the fundamental principle.

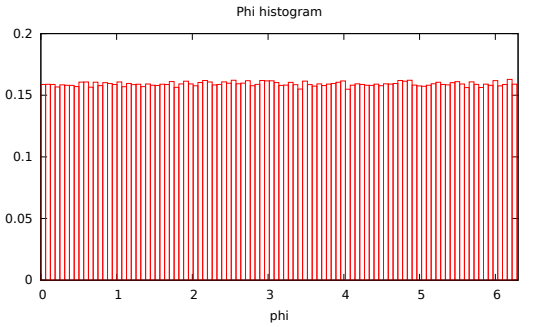

Figure 7: Distribution of ϕ generated using a unit uniform random variable U with the fundamental principle.

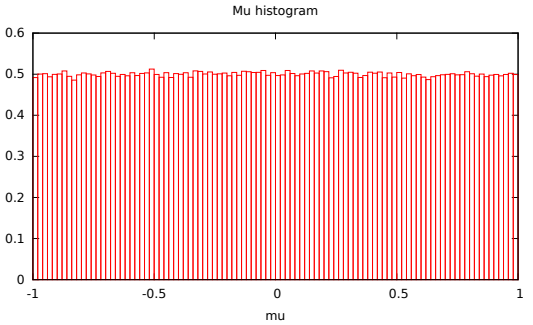

Figure 8: Distribution of μ generated using a unit uniform random variable U with the fundamental principle.

Note that P_s is between 0 and 1, as expected for a probability. $P_s \approx 0$ for $\sigma_a \gg \sigma_s$, and $P_s \approx 1$ for $\sigma_a \ll \sigma_s$.

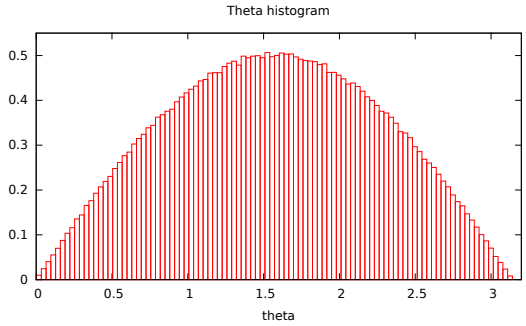

Figure 9: Distribution of θ generated using a unit uniform random variable U with the fundamental principle.

To determine whether a photon is absorbed on a given iteration, we generate a unit uniform random number U. The photon scatters if $U < P_s$ and it is absorbed if $U > P_s$.

With this setup, a Monte Carlo radiative transfer experiment was run with 10^6 photons, $\tau_{\max} = 10$, $z_{\max} = 1$ and $P_s = 1$ (no absorption). On this particular run, 101306 photons were transmitted (exited at $z = z_{\max}$) and 898694 were reflected (exited at $z = 0$). It took 532 steps before all photons were either reflected or transmitted.

The orientation of each transmitted photon was captured and put into bins. This allows us to determine the normalized intensity distribution as a function of θ and compare it to the expected theoretical curve. It is seen in Figure 10 that there is excellent agreement between the theoretical intensity distribution and the distribution calculated using Monte Carlo.

A second experiment was performed in which there was 50% probability of a photon being absorbed at each step ($P_s = P_a = 0.5$). As before, $\tau_{\max} = 10$ and $z_{\max} = 1$. This time, a much larger number of photons was used: 10^8. This was necessary to ensure that enough photons made it through the slab to make an accurate intensity distribution. Even with 10^8 incident photons, only 2768 photons were transmitted. The vast majority (82840816) were absorbed, and the rest (17156416) were reflected. It took only 28 iterations before all photons were either absorbed, reflected, or transmitted.

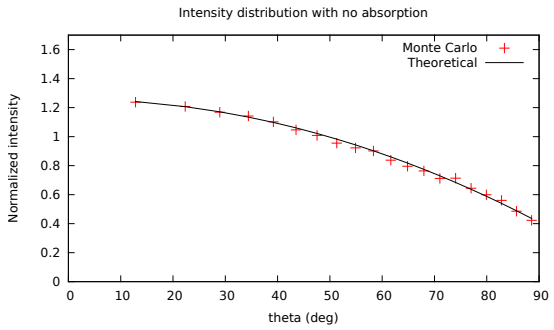

Figure 10: Normalized intensity distribution for a radiative transfer experiment with no absorption.

The intensity distribution for transmitted photons is shown in Figure 11. Not only does the absorption drastically reduce the number of transmitted photons, but it can be seen that absorption also changes the intensity distribution.

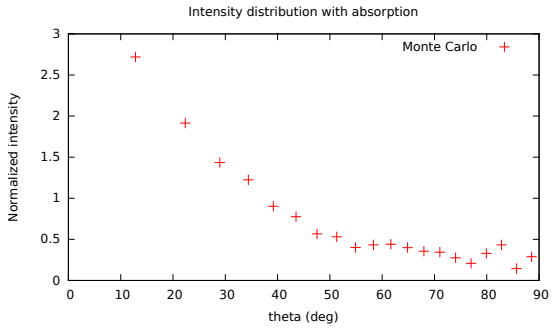

Figure 11: Normalized intensity distribution for a radiative transfer experiment with absorption ($P_s = 0.5$).

4 Markov Chain Monte Carlo

Markov Chains are random processes in which the next state depends only on the current state. As another example of a Monte Carlo, we perform the classic Markov Chain example: sunny and rainy days. In this example, the random process is the weather on day n, and we have two possible states for each day: sunny (S) and rainy (R).

The transition probabilities are:

$$P[W_{n+1} = S|W_n = R] = 0.5 \tag{19}$$

$$P[W_{n+1} = R|W_n = R] = 0.5 \tag{20}$$

$$P[W_{n+1} = S|W_n = S] = 0.9 \tag{21}$$

$$P[W_{n+1} = R|W_n = S] = 0.1 \tag{22}$$

or, in matrix form

$$P = \begin{bmatrix} 0.9 & 0.5 \\ 0.1 & 0.5 \end{bmatrix} \tag{23}$$

Using this matrix, we can find the probabilities of the next day's weather, given the curent day's weather. For example, if today is sunny, then the probabilities for tomorrow are

$$\begin{bmatrix} 0.9 & 0.5 \\ 0.1 & 0.5 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.9 \\ 0.1 \end{bmatrix} \tag{24}$$

So given that today is sunny, it is much more likely that tomorrow will be sunny. Applying the matrix again, we can find the probabilities for two days from now:

$$\begin{bmatrix} 0.9 & 0.5 \\ 0.1 & 0.5 \end{bmatrix} \begin{bmatrix} 0.9 \\ 0.1 \end{bmatrix} = \begin{bmatrix} 0.86 \\ 0.14 \end{bmatrix} \tag{25}$$

Again, it is more likely that the day will be sunny.

With Fortran 95 code, we applied the probability matrix repeatedly to two different starting states. The results are plotted in Figure 12 and Figure 13. It is seen that the probabilities converge towards:

$$P[W_\infty = S] = 0.833333 \tag{26}$$

$$P[W_\infty = R] = 0.166667 \tag{27}$$

and the probability matrix after 100 iterations is

$$P^{100} = \begin{bmatrix} 0.833333 & 0.833333 \\ 0.166667 & 1.66667 \end{bmatrix} \tag{28}$$

The steady state probabilities are given by:

$$\begin{bmatrix} 0.9 & 0.5 \\ 0.1 & 0.5 \end{bmatrix} \begin{bmatrix} P[W_\infty = S] \\ P[W_\infty = R] \end{bmatrix} = \begin{bmatrix} P[W_\infty = S] \\ P[W_\infty = R] \end{bmatrix} \tag{29}$$

$$\begin{bmatrix} -0.1 & 0.5 \\ 0.1 & -0.5 \end{bmatrix} \begin{bmatrix} P[W_\infty = S] \\ P[W_\infty = R] \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{30}$$

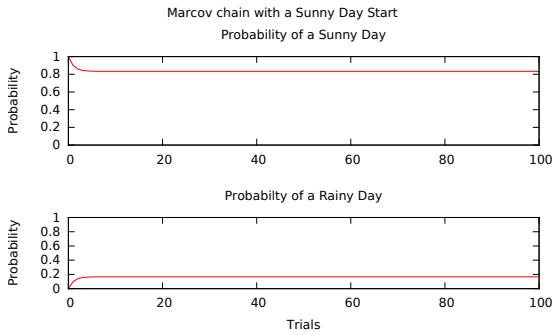

Figure 12: Probabilities of sunny and rainy days, given that the first day was a sunny one.

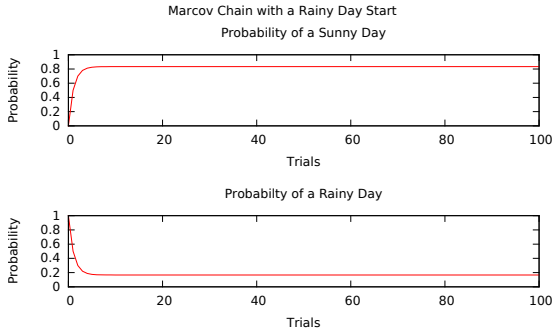

Figure 13: Probabilities of sunny and rainy days, given that the first day was a rainy one.

which gives

$$P[W_\infty = S] = 5P[W_\infty = R] \tag{31}$$

Choosing values which add up to 1, we get

$$P[W_\infty = S] = \frac{5}{6} \approx 8.33333 \tag{32}$$

$$P[W_\infty = R] = \frac{1}{6} \approx 0.16667 \tag{33}$$

which agrees with the predicted results above.

From the numerical results above, we can also guess that

$$P^\infty = \begin{bmatrix} 5/6 & 5/6 \\ 1/6 & 1/6 \end{bmatrix} \tag{34}$$

and note that

$$\begin{bmatrix} 5/6 & 5/6 \\ 1/6 & 1/6 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} 5/6 \\ 1/6 \end{bmatrix} \tag{35}$$

for any a, b which add up to 1. Thus, regardless of the starting condition, the probabilities will always come out to $5/6$ and $1/6$ in the steady state. This was confirmed by a number of manual runs of the code.

5 Metropolis-Hastings sampling

As another example of a Markov-Chain Monte Carlo method, we will use the Metropolis-Hastings random walk. This can be used to generate samples from complicated probability distributions. In this case, we will be generating random samples from a distribution proportional to:

$$P(x) = \frac{1}{2\sqrt{pi}} \left(\sin 5x + \sin 2x + 2\right) e^{-x^2} \tag{36}$$

For this implementation of the Metropolis-Hastings algorithm, we will use a normalized distribution centred on the current state. The proposal distributions are then:

$$q\left(x_n|x^*\right) = q\left(x^*|x_n\right) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left[\frac{(x^* - x_n)^2}{2\sigma^2}\right] \tag{37}$$

Our Metropolis-Hastings acceptance probability is

$$\min\left(1, \frac{p(x^*)q(x^*|x_n)}{p(x_n)q(x_n|x^*)}\right) \tag{38}$$

However, since $q(x^*|x_n) = q(x_n|x^*)$, this reduces to

$$\min\left(1, \frac{p(x^*)}{p(x_n)}\right) \tag{39}$$

which is the Metropolis criterion.

In general, the inclusion of the ratio allows for non-symmetric proposal probabilities. If the proposal probability distribution is symmetric (as it is in the case of a normal or uniform distribution centred around the current point), then this ratio is unity and it has no effect.

The key to the Metropolis-Hastings algorithm is the accept-reject step. We generate a proposed point x^* using the distribution $q(x^*|x_n)$. Then we generate a random variable r which is uniformly distributed between 0 and 1. Then the next point x_{n+1} is given by

$$x_{n+1} = \begin{cases} x^* & \text{if } r < \min\left(1, \frac{p(x^*)}{p(x_n)}\right) \\ x_n & \text{otherwise} \end{cases} \tag{40}$$

Fortran 90 code was created based on this algorithm to sample the distribution $P(x)$ from Equation (36) above. This was done using a starting point $x_0 = 1$, and different standard deviation values for the proposal distribution: $\sigma = 0.025, 1.0, 50.0$. 1000 iterations of the algorithm were run—the results are shown in Figures 14, 15, and 16. For $\sigma = 0.025$, 2.60% of points were accepted, for $\sigma = 1$, 50.0% of points were selected, and for $\sigma = 50$, 98.0% of points were selected.

It can be seen that $\sigma = 1$ gives the best sampling of the distribution. This is not surprising, since $P(x)$ is proportional to e^{-x^2}, which has a characteristic length of 1.

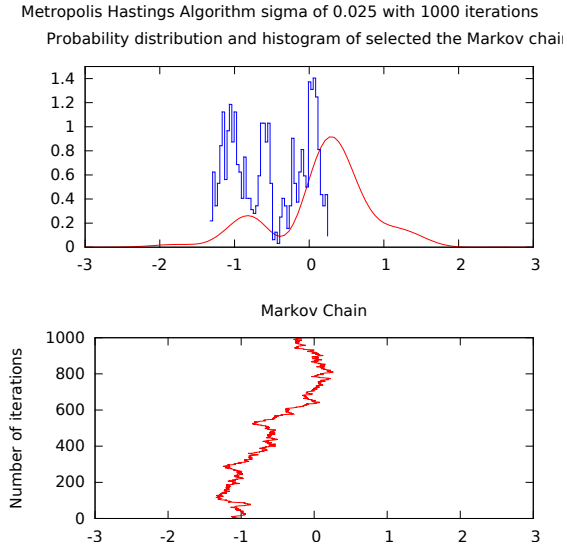

Figure 14: Metropolis sampling with $\sigma = 0.025$ and 1000 iterations.

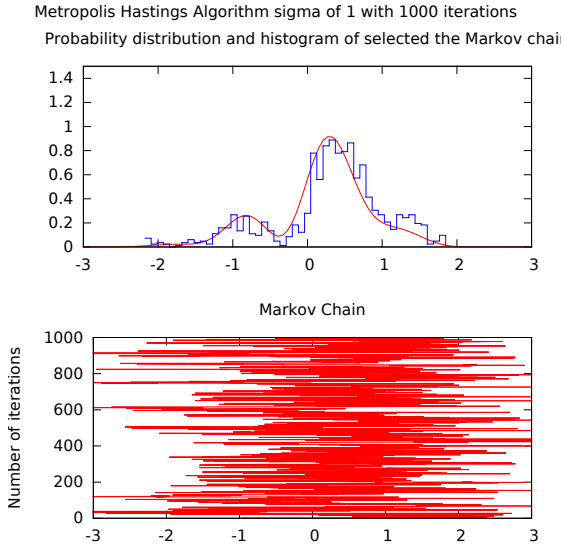

Figure 15: Metropolis sampling with $\sigma = 1$ and 1000 iterations.

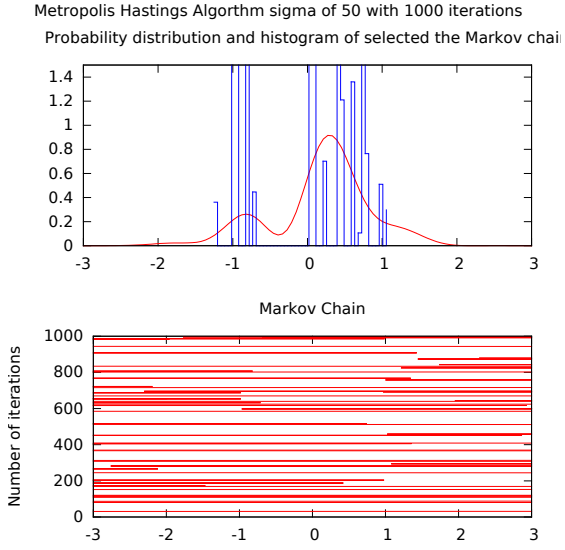

Figure 16: Metropolis sampling with $\sigma = 50$ and 1000 iterations.

To improve the sampling, the algorithm was used with the same three σ values, but with 50000 iterations. The

results are shown in Figures 17, 18, and 19. Overall, the sampling is better in all three cases, however, the $\sigma = 1$ test is by far the best.

The acceptance rates for 50000 points were nearly identical to the acceptance rates for 1000 iterations. For $\sigma = 0.025$, 2.12% of points were accepted, for $\sigma = 1$, 50.2% of points were selected, and for $\sigma = 50$, 98.6% of points were selected. This shows that the acceptance rate is more strongly controlled by the selection of points than it is by the number of iterations.

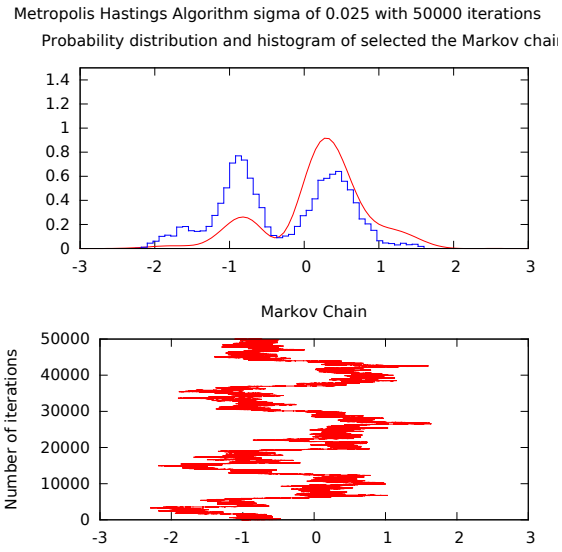

Figure 17: Metropolis sampling with $\sigma = 0.025$ and 50000 iterations.

More tests were run to explore the use of a burn-in phase. As a control, three tests were run using $x_0 = -3$, $\sigma = 0.2$ and 1000 iterations. These results are seen in Figures 20, 21, and 22. The acceptance rates were 15.4%, 16.5%, and 14.7%, respectively.

Three more tests were run using $x_0 = -3$, $\sigma = 0.2$ and 1000 iterations, but this time the first 200 iterations were ignored. These results are seen in Figures 23, 24, and 25. The acceptance rates were 14.4%, 16.5%, and 15.0%, respectively. It is seen that even though fewer total points are used, the sampling is better when the first 200 iterations are ignored.

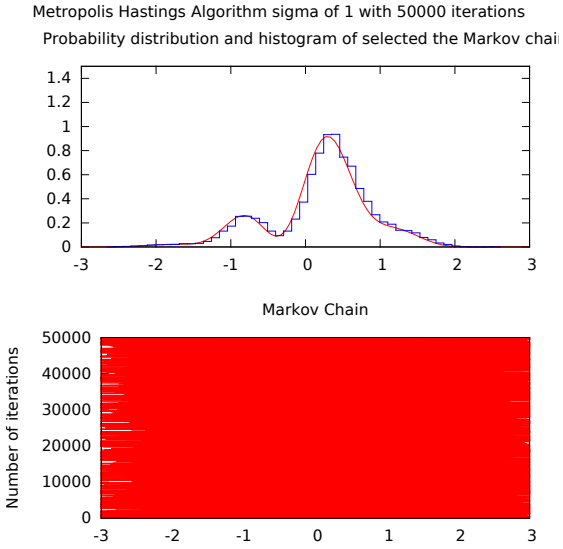

Figure 18: Metropolis sampling with $\sigma = 1$ and 50000 iterations.

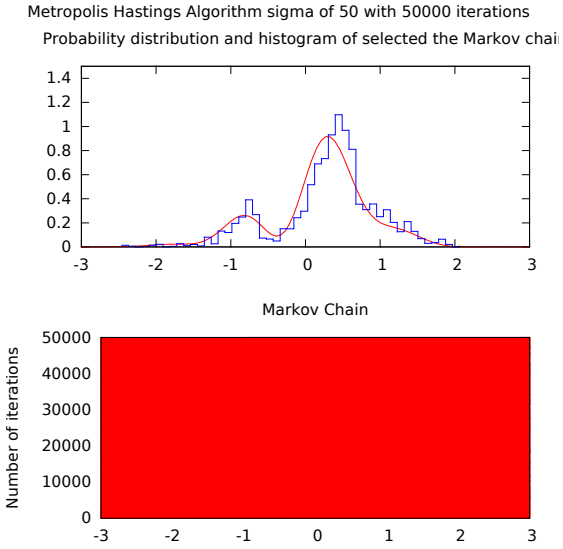

Figure 19: Metropolis sampling with $\sigma = 50$ and 50000 iterations.

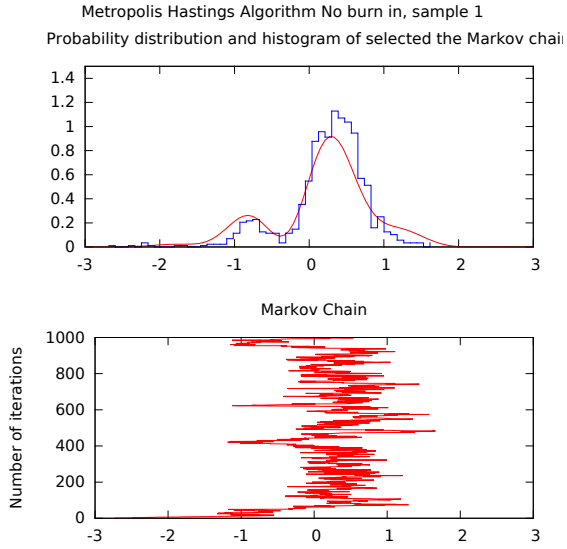

Figure 20: Metropolis sampling with $\sigma = 0.2$, 1000 iterations, and no burn-in.

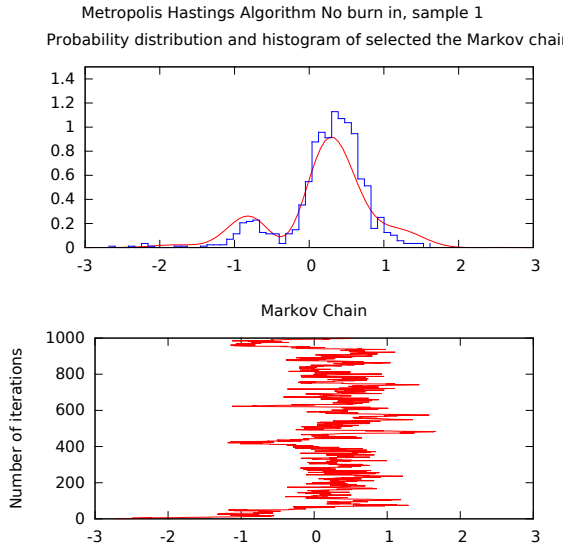

Figure 21: Metropolis sampling with $\sigma = 0.2$, 1000 iterations, and no burn-in.

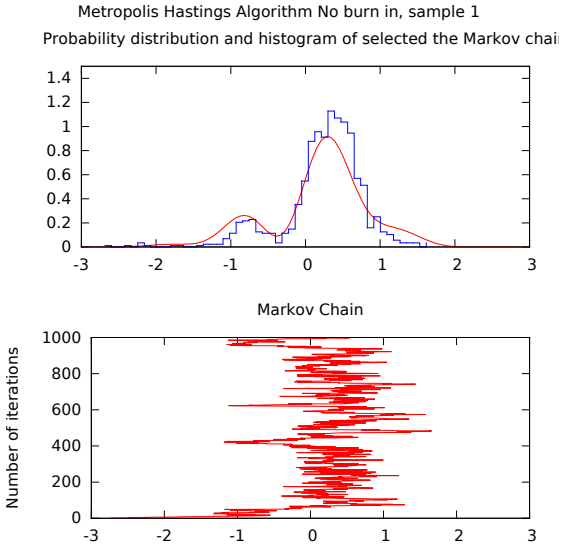

Figure 22: Metropolis sampling with $\sigma = 0.2$, 1000 iterations, and no burn-in.

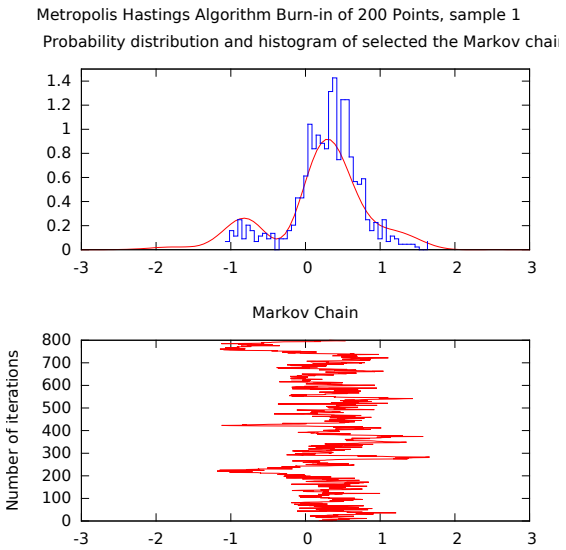

Figure 23: Metropolis sampling with $\sigma = 0.2$, 1000 iterations, and 200 iteration burn-in.

6 Simulated annealing

Simulated annealing is used to find the global maxima of complicated functions. It is more robust than traditional hill-climbing techniques at the cost of execution speed. If $P(x)$ is the function to be maximized, then the

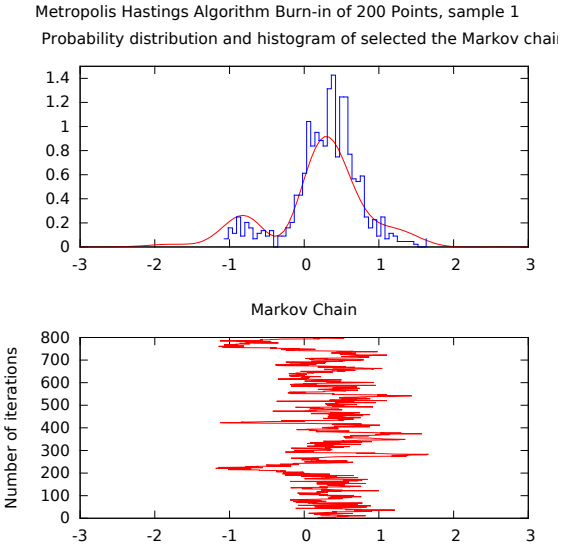

Figure 24: Metropolis sampling with $\sigma = 0.2$, 1000 iterations, and 200 iteration burn-in.

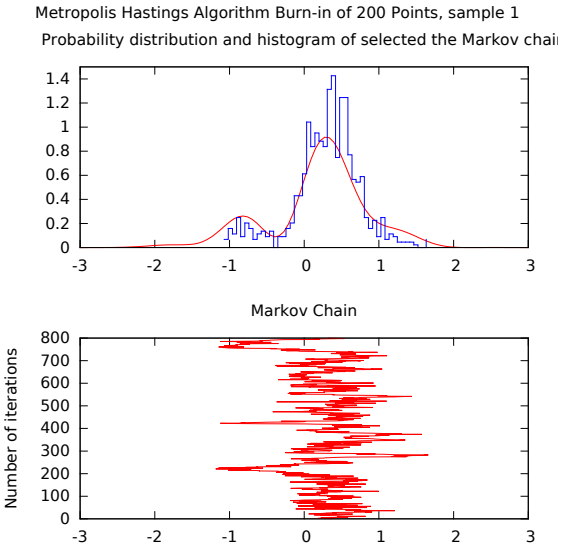

Figure 25: Metropolis sampling with $\sigma = 0.2$, 1000 iterations, and 200 iteration burn-in.

transition probability is

$$A\left(x_n \to x^*\right) = \min\left(1, \left(\frac{P\left(x^*\right)}{P\left(x_n\right)}\right)^{1/T_n}\right) \tag{41}$$

The temperature, T_n, is a key part of this algorithm. When the temperature is set to 1, then this is simply the Metropolis random walk algorithm. When T_n is large (e.g. 100), then the transition probabilities will tend to be high, regardless of $P(x^*)$ and $P(x_n)$. As the temperature "cools" down to near unity, the transition probability approaches $P(x^*)/P(x_n)$ from the Metropolis algorithm. As the temperature "cools" further towards 0, the transition probability approaches a step function: it is 1 if $P(x^*) > P(x_n)$ and 0 otherwise. So for high temperatures, the walk is essentially random, exploring the whole space. For small temperatures, the walk approaches pure hill-climbing algorithms, narrowing in on local maxima.

Note that if $P(E) = e^{-E}$, then

$$A\left(x_n \to x^*\right) = \min\left(1, e^{(E_n - E^*)/T_n}\right) \tag{42}$$

Making the replacement $T_n \to k_B T$ demonstrates why this algorithm is called "simulated annealing." In statistical mechanics, the probability of a transition is proportional to the exponential of the energy difference over $k_B T$.

The simulated annealing algorithm was implemented by Goffe et al.; this code can be seen in Section 8.10. This code was used to find the maxima of three functions. We used all the recommended settings for the input parameters, suggested within the Goffe et al. code.

The first function

$$f(x,y) = e^{-\left(x^2+y^2\right)} \tag{43}$$

is plotted in Figure 26. There is a single maximum at $(x,y) = (0,0)$.

The simulated annealing found the maximum at

$$(x,y) = \left(1.7235 \times 10^{-6}, -4.9737 \times 10^{-6}\right) \tag{44}$$

which is quite close to the actual value. The run time, however, was about 43.1 ms, which is quite significant for such a simple task.

The second function

$$f(x,y) = e^{-\left(x^2+y^2\right)} 2e^{-(x-1.7)^2-(y-1.7)^2} \tag{45}$$

is plotted in Figure 27. There is a local maximum at $(x,y) \approx (0,0)$ and a global maximum at $(x,y) \approx (1.7,1.7)$.

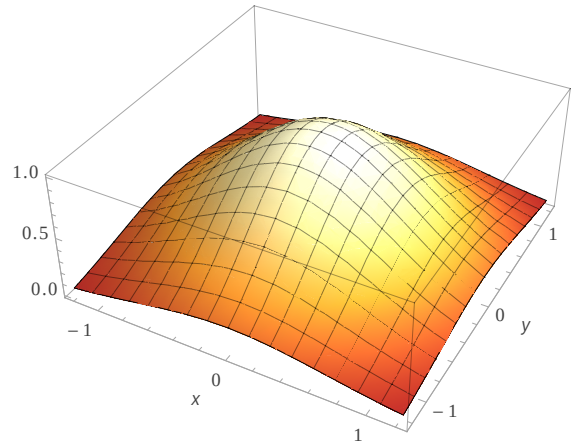

Figure 26: First function maximized by simulated annealing: $f(x,y) = e^{-(x^2+y^2)}$.

The simulated annealing found the maximum at

$$(x,y) = (1.6973, 1.6973) \tag{46}$$

which is quite close to the actual global maximum. So the algorithm did not get fooled by the local maximum like a basic hill-climbing algorithm might. Again, though, the run time was quite significant at around 50.5 ms.

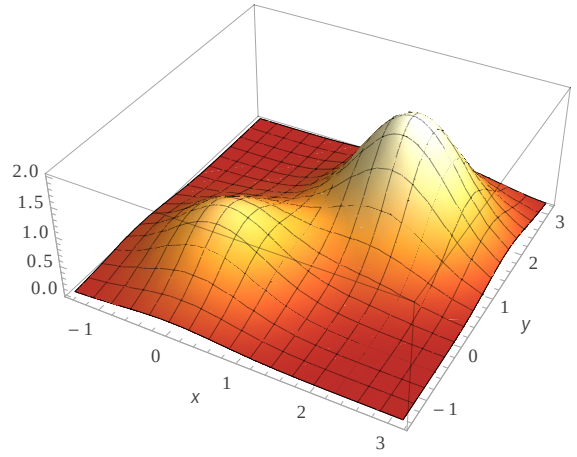

Figure 27: Second function maximized by simulated annealing: $f(x,y) = e^{-(x^2+y^2)} 2e^{-(x-1.7)^2-(y-1.7)^2}$.

The final function

$$f(x,y) = (1-x)^2 + 100(y-x^2)^2 \tag{47}$$

is plotted in Figure 28. This time, the simulated annealing algorithm was used to find a global minimum (just a matter of taking the negative of the function). There is one minimum of this function at $(x,y) = (1,1)$.

The simulated annealing found the minimum at

$$(x,y) = (0.99999, 0.99999) \tag{48}$$

which is quite close to the actual global minimum. So the algorithm was again successful. Again, though, the run time was quite significant at around 39.5 ms.

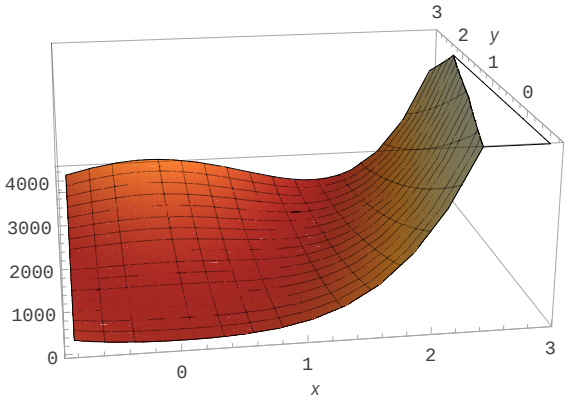

Figure 28: Second function maximized by simulated annealing: $f(x,y) = (1-x)^2 + 100(y-x^2)^2$.

Overall, this algorithm is quite robust, but it is computationally expensive. Simply put, finding the global maximum of a complicated function is going to be hard work.

7 Conclusions

In Section 2, we looked at methods for generating pseudo-random numbers. We explored a number of different tests for determining the effectiveness of a pseudo-random number generator. In Section 3, we applied Monte Carlo methods to a real physics problem: photons

moving through a slab of material. Using this method, we were able to generate the intensity distribution as a function of exit angle for a complicated system which is difficult to deal with analytically. In Section 4, we looked at a classic Markov Chain example: a string of sunny and rainy days in which the weather of the next day depends on the weather of the current day. We saw that the system very quickly converges to a probability state which is independent of the original state. In Section 5, we used another Markov Chain method: the Metropolis-Hastings random walk. With this, we were able to generate samples from a very complicated probability distribution function which would have been otherwise difficult to sample from. In Section 6, we used the simulated annealing algorithm to find the global maxima of three functions. The algorithm was robust and was not fooled by local maxima, but its runtimes were quite long.

8 Code

8.1 Pseudo random numbers module

```
module random_numbers_module
    implicit none

    private

    integer(16) :: lcm_seed, lcm_a, lcm_c, lcm_m

    public :: set_lcm_seed
    public :: set_lcm_params
    public :: lcm_random_number
    public :: AutoCorrelation

contains

    subroutine set_lcm_seed(seed_val)
        integer(16), intent(in) :: seed_val
        lcm_seed = seed_val
    end subroutine

    subroutine set_lcm_params(a_val, c_val, m_val)
        integer(16), intent(in) :: a_val, c_val, m_val
        lcm_a = a_val
        lcm_c = c_val
        lcm_m = m_val
    end subroutine

    real(8) function lcm_random_number()
        lcm_seed = mod(lcm_a*lcm_seed + lcm_c, lcm_m)
        lcm_random_number = lcm_seed/real(lcm_m, kind = 8)
    end function

    subroutine AutoCorrelation(x, k, AC)
      real(8), dimension(:), intent(in) :: x
      integer, intent(in) :: k
      real(8), intent(out) :: AC
      real(8) :: xbar, denom, numerator
      integer :: i, limit

      denom = 0
      numerator = 0
      limit = size(x)-k

      xbar = sum(x)/size(x)

```

```
45       do i = 1, limit
46          numerator = numerator + ((x(i)-xbar)*(x(i+k)-xbar))
47          denom = denom + ((x(i)-xbar)**2)
48       enddo
49
50       AC = numerator/denom
51
52     end subroutine
53
54 end module
```

8.2 Pseudo random numbers main code

```
1  program Pseudo_RNGs
2      use Useful_stuff_module
3      use Random_numbers_module
4      use Ouyed_random_number_module
5
6      implicit none
7      integer(16), dimension(3) :: a, c, m, seed
8      integer :: i, simple_test_file
9
10     simple_test_file = new_file_unit()
11     open( unit = simple_test_file, file = "./Data/Simple_LCM_test.txt", action = 'write' )
12
13     call set_lcm_seed(seed_val = 3_16)
14     call set_lcm_params(a_val = 7_16, c_val = 0_16, m_val = 10_16)
15
16     do i = 1, 50
17         write(simple_test_file,*) lcm_random_number()
18     end do
19
20     close(simple_test_file)
21
22     a(1) = 106
23     c(1) = 1283
24     m(1) = 6075
25     seed(1) = 1
26
27     a(2) = 107
28     c(2) = 1283
29     m(2) = 6075
30     seed(2) = 1
31
32     a(3) = 1103515245
33     c(3) = 12345
34     m(3) = 32768
35     seed(3) = 1
```

```
36
37      do i = 1, 3
38          call lcm_test(a = a(i), c = c(i), m = m(i), seed = seed(i), test_num = i)
39      end do
40
41      call lcm_chi2_test(a = a(3), c = c(3), m = m(3), seed = seed(3))
42
43      call auto_correlation_calc(a = a(3), c = c(3), m = m(3), seed = seed(3), num = 110)
44
45  contains
46
47      subroutine lcm_test(a, c, m, seed, test_num)
48          integer(16), intent(in) :: a, c, m, seed
49          integer, intent(in) :: test_num
50          character(len = 1024) :: filename
51          integer :: lcm_file, i, j
52          real :: last_num, new_num
53
54          write(filename, "(a16,i1.1,a4)") "./Data/LCM_test_", test_num, ".txt"
55          lcm_file = new_file_unit()
56          open( unit = lcm_file , file = filename, action = 'write')
57
58          call set_lcm_params(a_val = a, c_val = c, m_val = m)
59          call set_lcm_seed(seed_val = seed)
60
61          last_num = seed/real(m, kind = 8)
62          do i = 1, 10000
63              new_num = lcm_random_number()
64              write(lcm_file, *) last_num, new_num
65              last_num = new_num
66          end do
67
68          close(lcm_file)
69
70      end subroutine
71
72      subroutine lcm_chi2_test(a, c, m, seed)
73          integer(16), intent(in) :: a, c, m, seed
74          real(8), dimension(10) :: lcm_obs, lcm_exp
75          real(8), dimension(10) :: gfort_obs, gfort_exp
76          real(8) :: lcm_rand, gfort_rand
77          real(8) :: lcm_chi2, gfort_chi2, AC
78          character(len = 1024) :: format_str
79          integer :: chi2_file, AC_LCM_file, AC_gfort_file
80          integer :: num_tests
81          integer :: i, j
82
83          chi2_file = new_file_unit()
84          open(unit = chi2_file, file = "./Data/LCM_chi2_test.txt", action = 'write')
85
```

```
86          call set_lcm_params(a_val = a, c_val = c, m_val = m)
87          call set_lcm_seed(seed_val = seed)
88
89          call init_random_seed()
90          lcm_obs = 0.0
91          gfort_obs = 0.0
92          num_tests = 100
93          do i = 1, num_tests
94              lcm_rand = lcm_random_number()
95              call random_number(gfort_rand)
96              do j = 1, 10
97                  if ((lcm_rand >= (j-1)*0.1) .and. (lcm_rand < j*0.1)) then
98                      lcm_obs(j) = lcm_obs(j) + 1.0
99                  end if
100                 if ((gfort_rand >= (j-1)*0.1) .and. (gfort_rand < j*0.1)) then
101                     gfort_obs(j) = gfort_obs(j) + 1.0
102                 end if
103             end do
104         end do
105         lcm_exp(:) = num_tests/10.0
106         gfort_exp(:) = num_tests/10.0
107
108         format_str = '(i2.1,a3,i2.1,a3,i2.1,a3)'
109         do i = 1, 10
110             write(chi2_file, format_str) nint(lcm_obs(i)), ' & ', nint(gfort_obs(i)), ' & ', nint(
                    lcm_exp(i)), ' \\'
111         end do
112
113
114         lcm_chi2 = 0.0
115         gfort_chi2 = 0.0
116         do i = 1, 10
117             lcm_chi2 = lcm_chi2 + ((lcm_obs(i) - lcm_exp(i))**2)/lcm_exp(i)
118             gfort_chi2 = gfort_chi2 + ((gfort_obs(i) - gfort_exp(i))**2)/gfort_exp(i)
119         end do
120
121         write(*,*) "-----------"
122         write(*,*) "LCM chi squared = ", lcm_chi2
123         write(*,*) "Gfortran chi squared = ", gfort_chi2
124         write(*,*) "-----------"
125
126         close(chi2_file)
127
128     end subroutine
129
130     subroutine auto_correlation_calc(a, c, m, seed, num)
131       integer(16), intent(in) :: a, c, m, seed
132       integer, intent(in) :: num
133       real(8), dimension(num):: gfortran_rand, lcm_rand
134       integer :: i
```

```
135       real(8) :: AC_LCM, AC_Gfort
136       integer :: AC_LCM_file, AC_Gfort_file
137
138       call set_lcm_params(a_val = a, c_val = c, m_val = m)
139       call set_lcm_seed(seed_val = seed)
140
141       do i = 1, num
142          lcm_rand(i) = lcm_random_number()
143          call random_number(gfortran_rand(i))
144       end do
145
146       AC_LCM_file = new_file_unit()
147       open(unit = AC_LCM_file, file = "./Data/Auto_correlation_LCM_test.txt", action = 'write')
148
149       AC_Gfort_file = new_file_unit()
150       open(unit = AC_Gfort_file, file = "./Data/Auto_correlation_gfortran_test.txt", action = '
              write')
151
152       do i = 1, num
153           call AutoCorrelation(lcm_rand, i, AC_LCM)
154           write(AC_LCM_file,*) i, abs(AC_LCM), lcm_rand(i)
155
156           call AutoCorrelation(gfortran_rand, i, AC_Gfort)
157           write(AC_Gfort_file,*) i, abs(AC_Gfort), gfortran_rand(i)
158       end do
159
160       close(AC_LCM_file)
161       close(AC_Gfort_file)
162
163     end subroutine
164
165 end program
```

8.3 Pseudo random numbers plotting code

```
1  reset
2  set terminal pdfcairo
3
4  set output "./Plots/LCM_test_1.pdf"
5
6  set title "Linear congruential method test 1"
7  set xlabel "Last random number"
8  set ylabel "Current random number"
9  unset key
10
11 plot "./Data/LCM_test_1.txt" with points pt 0 lc 0
12
13 set output "./Plots/LCM_test_2.pdf"
```

```
14
15  set title "Linear congruential method test 2"
16  set xlabel "Last random number"
17  set ylabel "Current random number"
18  unset key
19
20  plot "./Data/LCM_test_2.txt" with points pt 0 lc 0
21
22  set output "./Plots/LCM_test_3.pdf"
23
24  set title "Linear congruential method test 3"
25  set xlabel "Last random number"
26  set ylabel "Current random number"
27  unset key
28
29  plot "./Data/LCM_test_3.txt" with points pt 0 lc 0
30
31  set output "./Plots/LCM_auto_correlation.pdf"
32
33  set terminal pdfcairo size 4, 4
34
35  set multiplot layout 2,1
36
37  #First plot
38  set title "Random Numbers"
39  unset key
40  set origin 0,0
41  set size 1,0.5
42  set xrange[0:100]
43  set yrange[0:1.1]
44  set xlabel "i"
45  set ylabel "Random number"
46  plot "./Data/Auto_correlation_LCM_test.txt" using 1:3 with lines
47
48  set title "Auto Correlation"
49  unset key
50  set origin 0,0.5
51  set size 1,0.5
52  set xrange[0:100]
53  set yrange[0:1]
54  set xlabel "k"
55  set ylabel "|A_k|"
56  plot "./Data/Auto_correlation_LCM_test.txt" using 1:2 with lines
57
58  unset multiplot
59  #Second Plot
60  set output "./Plots/Gfortran_auto_correlation.pdf"
61
62  set multiplot layout 2,1
63  unset key
```

```
64
65  set title "Random Numbers"
66  unset key
67  set origin 0,0
68  set size 1,0.5
69  set xrange[0:100]
70  set yrange[0:1.1]
71  set xlabel "i"
72  set ylabel "Random number"
73  plot "./Data/Auto_correlation_gfortran_test.txt" using 1:3 with lines
74
75  set title "Auto Correlation"
76  unset key
77  set origin 0,0.5
78  set size 1,0.5
79  set xrange[0:100]
80  set yrange[0:1]
81  set xlabel "k"
82  set ylabel "|A_k|"
83  plot "./Data/Auto_correlation_gfortran_test.txt" using 1:2 with lines
84
85  unset multiplot
86  unset out
87
88  reset
```

8.4 Radiative transfer code

```
1   program Radiative_transfer
2       use Useful_stuff_module
3       use Random_numbers_module
4       use Ouyed_random_number_module
5
6       implicit none
7
8       type photon
9           real(8) :: phi, theta
10          real(8) :: x, y, z
11          integer :: state, life_time
12      end type
13
14      real(8), parameter :: pi = 4.0d0*atan(1.0d0)
15      integer, parameter :: ALIVE = 0
16      integer, parameter :: ABSORBED = 1
17      integer, parameter :: REFLECTED = 2
18      integer, parameter :: TRANSMITTED = 3
19
20      character(len=1024) :: filename
```

```
21
22      call generate_histograms()
23
24      write(filename, "(a34)") "./Data/Scattering_experiment_1.txt"
25      call scattering_experiment( tau_max = 10.0d0,       &
26                                  z_max = 1.0d0,          &
27                                  prob_abs = -0.1d0,      &
28                                  num_photons = 1000000,  &
29                                  output_file = filename, &
30                                  num_bins = 20 )
31
32      write(filename, "(a34)") "./Data/Scattering_experiment_2.txt"
33      call scattering_experiment( tau_max = 10.0d0,       &
34                                  z_max = 1.0d0,          &
35                                  prob_abs = 0.5d0,       &
36                                  num_photons = 100000000,&
37                                  output_file = filename, &
38                                  num_bins = 20 )
39
40      write(filename, "(a35)") "./Data/Single_photon_experiment.txt"
41      call single_photon_experiment( tau_max = 10.0d0,       &
42                                     z_max = 1.0d0,          &
43                                     prob_abs = -0.1d0,       &
44                                     output_file = filename )
45
46
47  contains
48      subroutine set_up_lcm_generator()
49          integer(16) :: a, c, m, seed
50
51          a = 1103515245
52          c = 12345
53          m = 32768
54          seed = 1
55
56          call set_lcm_params(a_val = a, c_val = c, m_val = m)
57          call set_lcm_seed(seed_val = seed)
58      end subroutine
59
60      real(8) function random_tau()
61          real(8) :: uni_rand
62          !uni_rand = lcm_random_number()
63          call random_number(uni_rand)
64          random_tau = -log(1.0-uni_rand)
65      end function
66
67      real(8) function random_mu()
68          real(8) :: uni_rand
69          !uni_rand = lcm_random_number()
70          call random_number(uni_rand)
```

```
71          random_mu = 1.0 - 2.0*uni_rand
72      end function
73
74      real(8) function random_theta()
75          random_theta = acos(random_mu())
76      end function
77
78      real(8) function random_phi()
79          real(8) :: uni_rand
80          !uni_rand = lcm_random_number()
81          call random_number(uni_rand)
82          random_phi = 2.0*pi*uni_rand
83      end function
84
85      subroutine init_photons(photons)
86          type (photon), dimension(:), intent(inout) :: photons
87          real(8) :: mu
88          integer :: i
89
90          do i = 1, size(photons)
91              mu = random_mu()
92              do while (mu < 0.0)
93                  mu = random_mu()  ! Ensure mu is positive initially
94              end do
95
96              photons(i)%theta = acos(mu)
97              photons(i)%phi = random_phi()
98              photons(i)%x = 0.0
99              photons(i)%y = 0.0
100             photons(i)%z = 0.0
101             photons(i)%state = ALIVE
102             photons(i)%life_time = 1
103         end do
104     end subroutine
105
106     integer function num_in_state(photons, state)
107         type (photon), dimension(:), intent(in) :: photons
108         integer, intent(in) :: state
109         integer :: i
110
111         num_in_state = 0
112         do i = 1, size(photons)
113             if (photons(i)%state == state) then
114                 num_in_state = num_in_state + 1
115             end if
116         end do
117     end function
118
119     subroutine step_forward(tau_max, z_max, prob_abs, photons)
120         real(8), intent(in) :: tau_max, z_max, prob_abs
```

```
        type (photon), dimension(:), intent(inout) :: photons
        integer :: num_photons, i
        real(8) :: tau, dist, uni_rand

        num_photons = size(photons)

        do i = 1, num_photons
            if (photons(i)%state == ALIVE) then
                dist = random_tau()*z_max/tau_max
                photons(i)%x = photons(i)%x + dist*sin(photons(i)%theta)*cos(photons(i)%phi)
                photons(i)%y = photons(i)%y + dist*sin(photons(i)%theta)*sin(photons(i)%phi)
                photons(i)%z = photons(i)%z + dist*cos(photons(i)%theta)

                photons(i)%life_time = photons(i)%life_time + 1

                call random_number(uni_rand)
                if (photons(i)%z > z_max) then
                    photons(i)%state = TRANSMITTED
                else if (photons(i)%z < 0.0) then
                    photons(i)%state = REFLECTED
                else if (uni_rand < prob_abs) then
                    photons(i)%state = ABSORBED
                else
                    photons(i)%theta = random_theta()
                    photons(i)%phi = random_phi()
                end if
            end if
        end do

    end subroutine

    subroutine generate_histograms()
        real(8), dimension(1000000) :: tau, mu, phi, theta
        real(8), dimension(100) :: tau_bins, mu_bins, phi_bins, theta_bins
        integer, dimension(100) :: tau_hist, mu_hist, phi_hist, theta_hist
        real(8), dimension(100) :: tau_pdf, mu_pdf, phi_pdf, theta_pdf
        integer :: tau_id, mu_id, phi_id, theta_id
        integer :: num_exp, num_bins
        real(8) :: norm_factor
        integer :: i

        open(unit = new_file_unit(tau_id), file = "./Data/Tau_histogram.txt", action = 'write')
        open(unit = new_file_unit(mu_id), file = "./Data/Mu_histogram.txt", action = 'write')
        open(unit = new_file_unit(phi_id), file = "./Data/Phi_histogram.txt", action = 'write')
        open(unit = new_file_unit(theta_id), file = "./Data/Theta_histogram.txt", action = 'write')

        !call set_up_lcm_generator()
        call init_random_seed()

        ! Run histogram tests
```

```
171
172         num_exp = size(tau)
173         do i = 1, num_exp
174             tau(i) = random_tau()
175             mu(i) = random_mu()
176             phi(i) = random_phi()
177             theta(i) = acos(mu(i))
178         end do
179
180         call normed_histogram(tau, tau_bins, tau_pdf)
181         call normed_histogram(mu, mu_bins, mu_pdf)
182         call normed_histogram(phi, phi_bins, phi_pdf)
183         call normed_histogram(theta, theta_bins, theta_pdf)
184
185         ! Calculate probability distribution functions
186
187         !tau_pdf  =  tau_hist / (real(num_exp)*( tau_bins(2) - tau_bins(1)))
188         !mu_pdf   =   mu_hist / (real(num_exp)*( mu_bins(2) -  mu_bins(1)))
189         !phi_pdf  =  phi_hist / (real(num_exp)*( phi_bins(2) - phi_bins(1)))
190         !theta_pdf = theta_hist / (real(num_exp)*(theta_bins(2) - theta_bins(1)))
191
192         num_bins = size(tau_hist)
193         do i = 1, num_bins
194             write(tau_id, *) tau_bins(i), tau_pdf(i)
195             write(mu_id, *) mu_bins(i), mu_pdf(i)
196             write(phi_id, *) phi_bins(i), phi_pdf(i)
197             write(theta_id, *) theta_bins(i), theta_pdf(i)
198         end do
199
200         close(tau_id)
201         close(mu_id)
202         close(phi_id)
203         close(theta_id)
204
205     end subroutine
206
207     subroutine scattering_experiment(tau_max, z_max, prob_abs, num_photons, num_bins, output_file)
208         real(8), intent(in) :: tau_max, z_max, prob_abs
209         character(len = 1024), intent(in) :: output_file
210         integer, intent(in) :: num_photons, num_bins
211         type (photon), dimension(num_photons) :: photons
212         real(8), dimension(num_bins) :: mu_bin_centres, theta_bin_centres, intensity,
                theor_intensity
213         integer, dimension(num_bins) :: mu_hist
214         real(8), dimension(:), allocatable :: transmitted_mus
215         real(8) :: mu_spacing
216         integer :: num_iter, max_num_iter, num_transmitted
217         integer :: file_id, i, j
218
219         open( unit = new_file_unit(file_id), file = output_file, action = 'write' )
```

```
220
221         max_num_iter = 100000
222
223         call init_photons(photons)
224
225         num_iter = 0
226         do while (num_in_state(photons, ALIVE) > 0)
227             call step_forward(tau_max, z_max, prob_abs, photons)
228             num_iter = num_iter + 1
229             if (num_iter > max_num_iter) then
230                 write(*,*) "WARNING in scattering_experiment: too many iterations!"
231                 exit
232             end if
233         end do
234
235         num_transmitted = num_in_state(photons, TRANSMITTED)
236         allocate(transmitted_mus(num_transmitted))
237
238         j = 1
239         do i = 1, num_photons
240             if (photons(i)%state == TRANSMITTED) then
241                 transmitted_mus(j) = cos(photons(i)%theta)
242                 j = j + 1
243             end if
244         end do
245
246         mu_hist = 0.0
247         mu_spacing = 1.0/num_bins
248         do i = 1, num_bins
249             mu_bin_centres(i) = (i - 0.5)*mu_spacing
250             theta_bin_centres(i) = acos(mu_bin_centres(i))*180.0/3.14159265
251             do j = 1, num_transmitted
252                 if (abs(transmitted_mus(j) - mu_bin_centres(i)) < 0.5*mu_spacing) then
253                     mu_hist(i) = mu_hist(i) + 1
254                 end if
255             end do
256             intensity(i) = mu_hist(i)*num_bins/(2.0*num_transmitted*mu_bin_centres(i))
257             theor_intensity(i) = 0.0244*(51.6 - 0.0043*(theta_bin_centres(i))**2)
258             write(file_id,*) theta_bin_centres(i), mu_hist(i), intensity(i), theor_intensity(i)
259         end do
260
261         !call histogram(transmitted_mus, mu_bin_centres, mu_hist)
262
263
264         !do i = 1, num_bins
265         !    write(file_id,*) acos(mu_bin_centres(i)), mu_hist(i)
266         !end do
267
268         !do i = 1, num_photons
269         !    write(file_id, *) photons(i)%x, photons(i)%y, photons(i)%z, &
```

```
270         !                   photons(i)%theta, photons(i)%phi,        &
271         !                   photons(i)%state, photons(i)%life_time
272         !end do
273
274         write(*,*) "================================"
275         write(*,*) "Scattering experiment:"
276         write(*,*) ""
277         write(*,*) "tau_max       = ", tau_max
278         write(*,*) "z_max         = ", z_max
279         write(*,*) "prob_abs      = ", prob_abs
280         write(*,*) "num_photons   = ", num_photons
281         write(*,*) "num_absorbed  = ", num_in_state(photons, ABSORBED)
282         write(*,*) "num_transmitted = ", num_in_state(photons, TRANSMITTED)
283         write(*,*) "num_reflected = ", num_in_state(photons, REFLECTED)
284         write(*,*) "num_iterations = ", num_iter
285         write(*,*) ""
286         write(*,*) "Output file = ", trim(output_file)
287         write(*,*) "================================"
288
289         close(file_id)
290         deallocate(transmitted_mus)
291
292     end subroutine
293
294     subroutine single_photon_experiment(tau_max, z_max, prob_abs, output_file)
295         real(8), intent(in) :: tau_max, z_max, prob_abs
296         character(len = 1024), intent(in) :: output_file
297         type (photon), dimension(1) :: photons
298         real(8), dimension(100000) :: x, y, z
299         integer :: file_id, num_iter, max_num_iter, i
300
301         open( unit = new_file_unit(file_id), file = output_file, action = 'write' )
302
303         max_num_iter = 100000
304
305         call init_photons(photons)
306         do while (photons(1)%life_time < 100)
307
308             call init_photons(photons)
309
310             do while (num_in_state(photons, ALIVE) > 0)
311                 x(photons(1)%life_time) = photons(1)%x
312                 y(photons(1)%life_time) = photons(1)%y
313                 z(photons(1)%life_time) = photons(1)%z
314
315                 call step_forward(tau_max, z_max, prob_abs, photons)
316                 if (photons(1)%life_time > max_num_iter) then
317                     write(*,*) "WARNING in scattering_experiment: too many iterations!"
318                     exit
319                 end if
```

```
320
321             end do
322
323             x(photons(1)%life_time) = photons(1)%x
324             y(photons(1)%life_time) = photons(1)%y
325             z(photons(1)%life_time) = photons(1)%z
326
327         end do
328
329         do i = 1, photons(1)%life_time
330             write(file_id, *) x(i), y(i), z(i), i
331         end do
332
333         write(*,*) "================================"
334         write(*,*) "Single photon experiment:"
335         write(*,*) ""
336         write(*,*) "tau_max     = ", tau_max
337         write(*,*) "z_max       = ", z_max
338         write(*,*) "prob_abs    = ", prob_abs
339         write(*,*) ""
340         write(*,*) "Output file = ", trim(output_file)
341         write(*,*) ""
342         write(*,*) "Photon life = ", photons(1)%life_time
343         write(*,*) "Photon state = ", photons(1)%state
344         write(*,*) ""
345         write(*,*) "================================"
346
347         close(file_id)
348
349     end subroutine
350
351 end program
```

8.5 Radiative transfer plotting code

```
1  reset
2  set terminal pdfcairo
3
4  set output "./Plots/Tau_histogram.pdf"
5
6  set title "Tau histogram"
7  set xlabel "tau"
8  set ylabel ""
9  set xrange[0:10]
10 set yrange[0:1]
11 unset key
12
13 plot "./Data/Tau_histogram.txt" with boxes
```

```
14
15  set output "./Plots/Mu_histogram.pdf"
16
17  set title "Mu histogram"
18  set xlabel "mu"
19  set ylabel ""
20  set xrange[-1:1]
21  set yrange[0:0.6]
22  unset key
23
24  plot "./Data/Mu_histogram.txt" with boxes
25
26  set output "./Plots/Phi_histogram.pdf"
27
28  set title "Phi histogram"
29  set xlabel "phi"
30  set ylabel ""
31  set xrange[0:6.3]
32  set yrange[0:0.2]
33  unset key
34
35  plot "./Data/Phi_histogram.txt" with boxes
36
37  set output "./Plots/Theta_histogram.pdf"
38
39  set title "Theta histogram"
40  set xlabel "theta"
41  set ylabel ""
42  set xrange[0:3.2]
43  set yrange[0:0.55]
44  unset key
45
46  plot "./Data/Theta_histogram.txt" with boxes
47
48  set output "./Plots/Scattering_experiment_1.pdf"
49
50  set title "Intensity distribution with no absorption"
51  set xlabel "theta (deg)"
52  set ylabel "Normalized intensity"
53  set xrange[0:90]
54  set yrange[0:1.7]
55  set key
56
57  plot \
58  "./Data/Scattering_experiment_1.txt" using 1:3 with points title "Monte Carlo", \
59  "./Data/Scattering_experiment_1.txt" using 1:4 with lines lc 0 title "Theoretical"
60
61  set output "./Plots/Scattering_experiment_2.pdf"
62
63  set title "Intensity distribution with absorption"
```

```
set xlabel "theta (deg)"
set ylabel "Normalized intensity"
set xrange[0:90]
set yrange[0:3.0]
set key

plot \
"./Data/Scattering_experiment_2.txt" using 1:3 with points title "Monte Carlo"

reset
```

8.6 Markov chain main code

```
program Marcov_chain
implicit none

call SunnyRainyCalculation()


contains

subroutine SunnyRainyCalculation()
  !This subroutine is meant to calculate the sunny ranny day senerio
  !There is a subroutine in the marcov_chain_module, however things needed
  !to be written to file, so here is the hardcoded situation for the lab
  real(kind=8),dimension(2,2)::prob_mat,pn
  real(kind=8),dimension(2)::init_vec,fin_vec
  integer::i,iterations=100
  prob_mat = reshape((/ 0.9d0, 0.1d0, 0.5d0, 0.5d0 /),shape(prob_mat))

  !for the first initial vec
  init_vec = reshape((/1,0/),shape(init_vec))
  open(unit=42, file="./Data/MarcovSunny1.txt", action="write")
  open(unit=43, file="./Data/MarcovRainy1.txt", action="write")
  !write the initial values
  write(42,*) init_vec(1), 0
  write(43,*) init_vec(2), 0
  write(*,*) prob_mat(1,1), prob_mat(1,2), prob_mat(2,1), prob_mat(2,2)

  !multiply the probability matrix
  !initial muliplication
  pn = prob_mat
  fin_vec=matmul(prob_mat,init_vec)
  write(42,*) fin_vec(1), 1
  write(43,*) fin_vec(2), 1
  write(*,*) pn(1,1), pn(1,2), pn(2,1), pn(2,2)
  !FINISH IT
  do i=2,iterations
```

```
36       pn=matmul(pn,prob_mat)
37       fin_vec=matmul(pn,init_vec)
38       !write(*,*) fin_vec
39       write(*,*) pn(1,1), pn(1,2), pn(2,1), pn(2,2)
40       write(42,*) fin_vec(1), i
41       write(43,*) fin_vec(2), i
42    enddo
43
44    close(42)
45    close(43)
46
47
48    !second set of initial values
49    init_vec = reshape((/0,1/),shape(init_vec))
50    open(unit=44, file="./Data/MarcovSunny2.txt", action="write")
51    open(unit=45, file="./Data/MarcovRainy2.txt", action="write")
52    !write the initial values
53    write(44,*) init_vec(1), 0
54    write(45,*) init_vec(2), 0
55
56    !multiply the probability matrix
57    !initial muliplication
58    pn = prob_mat
59    fin_vec=matmul(pn,init_vec)
60    write(44,*) fin_vec(1), 1
61    write(45,*) fin_vec(2), 1
62    !FINISH IT
63    do i=2,iterations
64       pn=matmul(pn,prob_mat)
65       fin_vec=matmul(pn,init_vec)
66       write(44,*) fin_vec(1), i
67       write(45,*) fin_vec(2), i
68    enddo
69
70    close(44)
71    close(45)
72
73 end subroutine
74
75 end program
```

8.7 Markov chain plotting code

```
1 set terminal pdfcairo
2 set output "./Plots/Markov_Sunny_Day.pdf"
3 #set terminal x11
4 set multiplot layout 2,1 title "Marcov chain with a Sunny Day Start"
5 unset key
```

```
set title "Probability of a Sunny Day"
set xrange [0:100]
set yrange [0:1]
set ylabel "Probability"
plot "./Data/MarcovSunny1.txt" using 2:1 with lines

set title "Probabilty of a Rainy Day"
set xrange [0:100]
set yrange [0:1]
set xlabel "Trials"
unset key
plot "./Data/MarcovRainy1.txt" using 2:1 with lines

unset multiplot
unset out

set output "./Plots/Markov_Rainy_Day.pdf"
#set terminal x11
set multiplot layout 2,1 title "Marcov Chain with a Rainy Day Start"
unset key
set title "Probability of a Sunny Day"
set xrange [0:100]
set yrange [0:1]
set ylabel "Probability"
plot "./Data/MarcovSunny2.txt" using 2:1 with lines

set title "Probabilty of a Rainy Day"
set xrange [0:100]
set yrange [0:1]
set xlabel "Trials"
unset key
plot "./Data/MarcovRainy2.txt" using 2:1 with lines

unset multiplot
unset out
```

8.8 Metropolis main code

```
program Metropolis
use Ouyed_random_number_module
use Useful_stuff_module

implicit none

integer::k, i, j, accepted_values, acc_vals3, acc_vals4, max_burn_iterations
real(kind=8)::sigma_burn
real(kind=8), dimension(3)::sigma
real(kind=8)::xn, x_star, unit_rand
```

```
integer, dimension(2,3)::file_id
integer, dimension(4)::num_iter
real(kind=8), allocatable,dimension(:,:)::x1,x2, x3, x4
real(kind=8) :: norm_factor1, norm_factor2
real(kind=8), dimension(3,50):: hist_data1, hist_data2, hist_data3, hist_data4
real(kind=8),dimension(3,50)::bin_center1, bin_center2, bin_center3, bin_center4

num_iter(1) = 1000
num_iter(2) = 50000
num_iter(3) = 1000
num_iter(4) = 800

!allocate the arrays
allocate(x1(3,num_iter(1)))
allocate(x2(3,num_iter(2)))
allocate(x3(3,num_iter(3)))
allocate(x4(3,num_iter(4)))

!Intialize them
sigma(1) = 0.025
sigma(2) = 1.0
sigma(3) = 50.0

!using an array of file ID to write to seperate data files for each senario
file_id(1,1) = 42
file_id(1,2) = 43
file_id(1,3) = 44
file_id(2,1) = 45
file_id(2,2) = 46
file_id(2,3) = 47

!OPEN ALL THE FILES!
open(unit=42, file="./Data/Metropolis_sigma-0.025_1000.txt", action = "write")
open(unit=43, file="./Data/Metropolis_sigma-1_1000.txt", action = "write")
open(unit=44, file="./Data/Metropolis_sigma-50_1000.txt", action = "write")
open(unit=45, file="./Data/Metropolis_sigma-0.025_50000.txt", action = "write")
open(unit=46, file="./Data/Metropolis_sigma-1_50000.txt", action = "write")
open(unit=47, file="./Data/Metropolis_sigma-50_50000.txt", action = "write")

!main loop
do k=1,2 !Change the max iterations
   do j=1,3 !new standard devation
      xn = -1.0
      accepted_values = 0.0
      do i=1,num_iter(k)
         !find x_star
         unit_rand = random_normal()
         x_star = sigma(j)*unit_rand + xn
         xn = select_new_point(x_star,xn)
         !icrement the accepted_values counter if needed
```

```
         if (abs(x_star - xn) .ge. 1e-20) then
              accepted_values = accepted_values + 1
         end if

         write(file_id(k,j),*) x_star, xn, i, accepted_values

         !write in the arrays
         if (k .eq. 1) then
            x1(j,i) = xn
         else if (k .eq. 2) then
            x2(j,i) = xn
         end if

      end do
      write(*,*) sigma(j), accepted_values/real(num_iter(k))
   end do
end do

!close the files
close(42)
close(43)
close(44)
close(45)
close(46)
close(47)

!reuse the file_id_array
file_id(1,1) = 48
file_id(1,2) = 49
file_id(1,3) = 50
file_id(2,1) = 51
file_id(2,2) = 52
file_id(2,3) = 53


!Open unit files
open(unit=48,file="./Data/hist_data_sigma-0.025_1000.txt",action="write")
open(unit=49,file="./Data/hist_data_sigma-1_1000.txt",action="write")
open(unit=50,file="./Data/hist_data_sigma-50_1000.txt",action="write")
open(unit=51,file="./Data/hist_data_sigma-0.025_50000.txt",action="write")
open(unit=52,file="./Data/hist_data_sigma-1_50000.txt",action="write")
open(unit=53,file="./Data/hist_data_sigma-50_50000.txt",action="write")

!deal with the historgrams
do i =1,3
   call normed_histogram(x1(i,:),bin_center1(i,:),hist_data1(i,:))
   call normed_histogram(x2(i,:),bin_center2(i,:),hist_data2(i,:))
   do j=1,size(bin_center1(i,:))
      write(file_id(1,i),*) bin_center1(i,j), hist_data1(i,j)
      write(file_id(2,i),*) bin_center2(i,j), hist_data2(i,j)
```

```
111     end do
112   end do
113
114   !close the files
115   close(48)
116   close(49)
117   close(50)
118   close(51)
119   close(52)
120   close(53)
121
122
123
124   !*****************************************************************************************************************

125   !Burn in section 1
126
127
128   !Intialize the sigma
129   sigma_burn = 0.2
130   max_burn_iterations = 1000
131
132   !using an array of file ID to write to seperate data files for each senario
133   file_id(1,1) = 54
134   file_id(1,2) = 55
135   file_id(1,3) = 56
136   file_id(2,1) = 57
137   file_id(2,2) = 58
138   file_id(2,3) = 59
139
140   !OPEN ALL THE FILES!
141   open(unit=54, file="./Data/Metropolis_Burn1_1.txt", action = "write")
142   open(unit=55, file="./Data/Metropolis_Burn1_2.txt", action = "write")
143   open(unit=56, file="./Data/Metropolis_Burn1_3.txt", action = "write")
144   open(unit=57, file="./Data/Metropolis_Burn2_1.txt", action = "write")
145   open(unit=58, file="./Data/Metropolis_Burn2_2.txt", action = "write")
146   open(unit=59, file="./Data/Metropolis_Burn2_3.txt", action = "write")
147
148   !main loop
149   do j=1,3 !new experiment
150     xn = -3.0
151     acc_vals3 = 0.0
152     acc_vals4 = 0.0
153     do i=1,max_burn_iterations
154       !find x_star
155       unit_rand = random_normal()
156       x_star = sigma_burn*unit_rand + xn
157       xn = select_new_point(x_star,xn)
158       !icrement the accepted_values counter if needed
159       write(file_id(1,j),*) x_star, xn, i
```

```
160
161      !write in the arrays
162      x3(j,i) = xn
163      if(i .gt. 200) then
164         x4(j,i-200)=xn
165         write(file_id(2,j),*) x_star, xn, i-200
166      end if
167
168      if (abs(x_star - xn) .ge. 1e-20) then
169        acc_vals3 = acc_vals3 + 1
170        if(i .gt. 200) then
171            acc_vals4 = acc_vals4 + 1
172        end if
173      end if
174
175    end do
176    write(*,*) sigma_burn, "3", acc_vals3/real(max_burn_iterations)
177    write(*,*) sigma_burn, "4", acc_vals4/real(max_burn_iterations-200)
178  end do
179
180  !close the files
181  close(54)
182  close(55)
183  close(56)
184  close(57)
185  close(58)
186  close(59)
187
188  !reuse the file_id_array
189  file_id(1,1) = 60
190  file_id(1,2) = 61
191  file_id(1,3) = 62
192  file_id(2,1) = 63
193  file_id(2,2) = 64
194  file_id(2,3) = 65
195
196  !Open unit files
197  open(unit=60,file="./Data/hist_data_burn1-1.txt",action="write")
198  open(unit=61,file="./Data/hist_data_burn1-2.txt",action="write")
199  open(unit=62,file="./Data/hist_data_burn1-3.txt",action="write")
200  open(unit=63,file="./Data/hist_data_burn2-1.txt",action="write")
201  open(unit=64,file="./Data/hist_data_burn2-2.txt",action="write")
202  open(unit=65,file="./Data/hist_data_burn2-3.txt",action="write")
203
204  !deal with the historgrams
205  do i =1,3
206     call normed_histogram(x3(i,:),bin_center3(i,:),hist_data3(i,:))
207     call normed_histogram(x4(i,:),bin_center4(i,:),hist_data4(i,:))
208     do j=1,size(bin_center3(i,:))
209        write(file_id(1,i),*) bin_center3(i,j), hist_data3(i,j)
```

```
      write(file_id(2,i),*) bin_center4(i,j), hist_data4(i,j)
   end do
end do

!close the files
close(60)
close(61)
close(62)
close(63)
close(64)
close(65)


!*****************************************************************************************************************

contains

real(kind=8) function prob_distribution(x)
 !The desired probability distribution
 real(kind=8),intent(in)::x
 real(kind=8)::PI,constant,hard_part, temp
 PI=3.14159265359
 constant = 1/(2.0*sqrt(PI))
 hard_part = (sin(5.0*x) + sin(2.0*x) + 2)*exp(-1.*(x**2))
 temp = constant * hard_part
 prob_distribution = temp
end function prob_distribution

real(kind=8) function select_new_point(x_star,xn)
 !This function determines if the new point is accepted or rejected
 !bassed on the acceptance probability. x_star is the new point, xn is the
 !current point
 real(kind=8), intent(in)::x_star,xn
 real(kind=8)::a, rand_number

 a = prob_distribution(x_star)/prob_distribution(xn)
 call random_number(rand_number)
 !write(*,*) a, rand_number, x_star, xn
 if (a .ge. 1) then
    select_new_point = x_star
 else if (a .gt. rand_number) then
    select_new_point = x_star
 else
    select_new_point = xn
 end if

end function select_new_point


end program
```

8.9 Metropolis plotting code

```
set terminal pdfcairo size 4,4
set output "./Plots/Metropolis_sigma-025-1000.pdf"

set multiplot layout 2,1 title "Metropolis Hastings Algorithm sigma of 0.025 with 1000 iterations "

unset key
set xrange [-3:3]
set yrange [0:1.5]
set ylabel " "
set style fill solid 2.0
set title "Probability distribution and histogram of selected the Markov chain"
plot (1/3.5449)*(sin(5*x)+sin(2*x)+2)*exp(-1*(x**2)), "./Data/hist_data_sigma-0.025_1000.txt" using
     1:2 linestyle 3 with steps

set xrange [-3:3]
set yrange [0:1000]
set title "Markov Chain"
set ylabel "Number of iterations"
plot "./Data/Metropolis_sigma-0.025_1000.txt" using 1:3 with lines

unset multiplot
unset out
reset

#second plot
set output "./Plots/Metropolis_sigma-1-1000.pdf"

set multiplot layout 2,1 title "Metropolis Hastings Algorithm sigma of 1 with 1000 iterations "

unset key
set xrange [-3:3]
set yrange [0:1.5]
set ylabel " "
set style fill solid 2.0
set title "Probability distribution and histogram of selected the Markov chain"
plot (1/3.5449)*(sin(5*x)+sin(2*x)+2)*exp(-1*(x**2)), "./Data/hist_data_sigma-1_1000.txt" using 1:2
     linestyle 3 with steps

set xrange [-3:3]
set yrange [0:1000]
set title "Markov Chain"
set ylabel "Number of iterations"
plot "./Data/Metropolis_sigma-1_1000.txt" using 1:3 with lines

unset multiplot
unset out
reset
```

```
46
47 #third plot
48 set output "./Plots/Metropolis_sigma-50-1000.pdf"
49
50 set multiplot layout 2,1 title "Metropolis Hastings Algorthm sigma of 50 with 1000 iterations "
51
52 unset key
53 set xrange [-3:3]
54 set yrange [0:1.5]
55 set ylabel " "
56 set style fill solid 2.0
57 set title "Probability distribution and histogram of selected the Markov chain"
58 plot (1/3.5449)*(sin(5*x)+sin(2*x)+2)*exp(-1*(x**2)), "./Data/hist_data_sigma-50_1000.txt" using
       1:2 linestyle 3 with steps
59
60 set xrange [-3:3]
61 set yrange [0:1000]
62 set title "Markov Chain"
63 set ylabel "Number of iterations"
64 plot "./Data/Metropolis_sigma-50_1000.txt" using 1:3 with lines
65
66 unset multiplot
67 unset out
68 reset
69
70 #4th plot
71 set output "./Plots/Metropolis_sigma-025-50000.pdf"
72
73 set multiplot layout 2,1 title "Metropolis Hastings Algorithm sigma of 0.025 with 50000 iterations"
74
75 unset key
76 set xrange [-3:3]
77 set yrange [0:1.5]
78 set ylabel " "
79 set style fill solid 2.0
80 set title "Probability distribution and histogram of selected the Markov chain"
81 plot (1/3.5449)*(sin(5*x)+sin(2*x)+2)*exp(-1*(x**2)), "./Data/hist_data_sigma-0.025_50000.txt"
       using 1:2 linestyle 3 with steps
82
83 set xrange [-3:3]
84 set yrange [0:50000]
85 set title "Markov Chain"
86 set ylabel "Number of iterations"
87 plot "./Data/Metropolis_sigma-0.025_50000.txt" using 1:3 with lines
88
89 unset multiplot
90 unset out
91 reset
92
93 #5th plot
```

```
94  set output "./Plots/Metropolis_sigma-1-50000.pdf"
95
96  set multiplot layout 2,1 title "Metropolis Hastings Algorithm sigma of 1 with 50000 iterations"
97
98  unset key
99  set xrange [-3:3]
100 set yrange [0:1.5]
101 set ylabel " "
102 set style fill solid 2.0
103 set title "Probability distribution and histogram of selected the Markov chain"
104 plot (1/3.5449)*(sin(5*x)+sin(2*x)+2)*exp(-1*(x**2)), "./Data/hist_data_sigma-1_50000.txt" using
        1:2 linestyle 3 with steps
105
106 set xrange [-3:3]
107 set yrange [0:50000]
108 set title "Markov Chain"
109 set ylabel "Number of iterations"
110 plot "./Data/Metropolis_sigma-1_50000.txt" using 1:3 with lines
111
112 unset multiplot
113 unset out
114 reset
115
116 #6th plot
117 set output "./Plots/Metropolis_sigma-50-50000.pdf"
118
119 set multiplot layout 2,1 title "Metropolis Hastings Algorithm sigma of 50 with 50000 iterations"
120
121 unset key
122 set xrange [-3:3]
123 set yrange [0:1.5]
124 set ylabel " "
125 set style fill solid 2.0
126 set title "Probability distribution and histogram of selected the Markov chain"
127 plot (1/3.5449)*(sin(5*x)+sin(2*x)+2)*exp(-1*(x**2)), "./Data/hist_data_sigma-50_50000.txt" using
        1:2 linestyle 3 with steps
128
129 set xrange [-3:3]
130 set yrange [0:50000]
131 set title "Markov Chain"
132 set ylabel "Number of iterations"
133 plot "./Data/Metropolis_sigma-50_50000.txt" using 1:3 with lines
134
135 unset multiplot
136 unset out
137 reset
138
139
140 #*****************************************************************************************************************
```

```
#7 plot
set output "./Plots/Metropolis_burn1-1.pdf"

set multiplot layout 2,1 title "Metropolis Hastings Algorithm No burn in, sample 1"

unset key
set xrange [-3:3]
set yrange [0:1.5]
set ylabel " "
set style fill solid 2.0
set title "Probability distribution and histogram of selected the Markov chain"
plot (1/3.5449)*(sin(5*x)+sin(2*x)+2)*exp(-1*(x**2)), "./Data/hist_data_burn1-1.txt" using 1:2
    linestyle 3 with steps

set xrange [-3:3]
set yrange [0:1000]
set title "Markov Chain"
set ylabel "Number of iterations"
plot "./Data/Metropolis_Burn1_1.txt" using 1:3 with lines

unset multiplot
unset out
reset

#8th plot
set output "./Plots/Metropolis_burn1-2.pdf"

set multiplot layout 2,1 title "Metrpolois Hastings Algorithm No burn in, sample 2"

unset key
set xrange [-3:3]
set yrange [0:1.5]
set ylabel " "
set style fill solid 2.0
set title "Probability distribution and histogram of selected the Markov chain"
plot (1/3.5449)*(sin(5*x)+sin(2*x)+2)*exp(-1*(x**2)), "./Data/hist_data_burn1-2.txt" using 1:2
    linestyle 3 with steps

set xrange [-3:3]
set yrange [0:1000]
set title "Markov Chain"
set ylabel "Number of iterations"
plot "./Data/Metropolis_Burn1_2.txt" using 1:3 with lines

unset multiplot
unset out
reset

#9th plot
set output "./Plots/Metropolis_burn1-3.pdf"
```

```
189
190 set multiplot layout 2,1 title "Metropolis Hastings Algorithm No burn in, sample 3"
191
192 unset key
193 set xrange [-3:3]
194 set yrange [0:1.5]
195 set ylabel " "
196 set style fill solid 2.0
197 set title "Probability distribution and histogram of selected the Markov chain"
198 plot (1/3.5449)*(sin(5*x)+sin(2*x)+2)*exp(-1*(x**2)), "./Data/hist_data_burn1-3.txt" using 1:2
        linestyle 3 with steps
199
200 set xrange [-3:3]
201 set yrange [0:1000]
202 set title "Markov Chain"
203 set ylabel "Number of iterations"
204 plot "./Data/Metropolis_Burn1_3.txt" using 1:3 with lines
205
206 unset multiplot
207 unset out
208 reset
209
210 #10th plot
211 set output "./Plots/Metropolis_burn2-1.pdf"
212
213 set multiplot layout 2,1 title "Metropolis Hastings Algorithm Burn-in of 200 Points, sample 1"
214
215 unset key
216 set xrange [-3:3]
217 set yrange [0:1.5]
218 set ylabel " "
219 set style fill solid 2.0
220 set title "Probability distribution and histogram of selected the Markov chain"
221 plot (1/3.5449)*(sin(5*x)+sin(2*x)+2)*exp(-1*(x**2)), "./Data/hist_data_burn2-1.txt" using 1:2
        linestyle 3 with steps
222
223 set xrange [-3:3]
224 set yrange [0:800]
225 set title "Markov Chain"
226 set ylabel "Number of iterations"
227 plot "./Data/Metropolis_Burn2_1.txt" using 1:3 with lines
228
229 unset multiplot
230 unset out
231 reset
232
233 #11 plot
234 set output "./Plots/Metropolis_burn2-2.pdf"
235
236 set multiplot layout 2,1 title "Metropolis Hastings Algorithm Burn-in of 200 Points, sample 2"
```

```
237
238 unset key
239 set xrange [-3:3]
240 set yrange [0:1.5]
241 set ylabel " "
242 set style fill solid 2.0
243 set title "Probability distribution and histogram of selected the Markov chain"
244 plot (1/3.5449)*(sin(5*x)+sin(2*x)+2)*exp(-1*(x**2)), "./Data/hist_data_burn2-2.txt" using 1:2
        linestyle 3 with steps
245
246 set xrange [-3:3]
247 set yrange [0:800]
248 set title "Markov Chain"
249 set ylabel "Number of iterations"
250 plot "./Data/Metropolis_Burn2_2.txt" using 1:3 with lines
251
252 unset multiplot
253 unset out
254 reset
255
256 #12th plot
257 set output "./Plots/Metropolis_burn2-3.pdf"
258
259 set multiplot layout 2,1 title "Metropolis Hastings Algorithm Burn-in of 200 Points, sample 3"
260
261 unset key
262 set xrange [-3:3]
263 set yrange [0:1.5]
264 set ylabel " "
265 set style fill solid 2.0
266 set title "Probability distribution and histogram of selected the Markov chain"
267 plot (1/3.5449)*(sin(5*x)+sin(2*x)+2)*exp(-1*(x**2)), "./Data/hist_data_burn2-3.txt" using 1:2
        linestyle 3 with steps
268
269 set xrange [-3:3]
270 set yrange [0:800]
271 set title "Markov Chain"
272 set ylabel "Number of iterations"
273 plot "./Data/Metropolis_Burn2_3.txt" using 1:3 with lines
274
275 unset multiplot
276 unset out
277 reset
```

8.10 Simulated annealing module code

```
1 ! This file is an example of the Corana et al. simulated annealing algorithm
2 ! for multimodal and robust optimization as implemented and modified by
```

```
!  Goffe, Ferrier and Rogers. Counting the above line
!  ABSTRACT as 1, the routine itself (SA), with its supplementary
!  routines, is on lines 232-990. A multimodal example from Judge et al.
!  (FCN) is on lines 150-231. The rest of this file (lines 1-149) is a
!  driver routine with values appropriate for the Judge example. Thus, this
!  example is ready to run.

!  To understand the algorithm, the documentation for SA on lines 236-
!  484 should be read along with the parts of the paper that describe
!  simulated annealing. Then the following lines will then aid the user
!  in becomming proficient with this implementation of simulated annealing.

!  Learning to use SA:
!      Use the sample function from Judge with the following suggestions
!  to get a feel for how SA works. When you've done this, you should be
!  ready to use it on most any function with a fair amount of expertise.
!    1. Run the program as is to make sure it runs okay. Take a look at
!       the intermediate output and see how it optimizes as temperature
!       (T) falls. Notice how the optimal point is reached and how
!       falling T reduces VM.
!    2. Look through the documentation to SA so the following makes a
!       bit of sense. In line with the paper, it shouldn't be that hard
!       to figure out. The core of the algorithm is described on pp. 68-70
!       and on pp. 94-95. Also see Corana et al. pp. 264-9.
!    3. To see how it selects points and makes decisions about uphill and
!       downhill moves, set IPRINT = 3 (very detailed intermediate output)
!       and MAXEVL = 100 (only 100 function evaluations to limit output).
!    4. To see the importance of different temperatures, try starting
!       with a very low one (say T = 10E-5). You'll see (i) it never
!       escapes from the local optima (in annealing terminology, it
!       quenches) & (ii) the step length (VM) will be quite small. This
!       is a key part of the algorithm: as temperature (T) falls, step
!       length does too. In a minor point here, note how VM is quickly
!       reset from its initial value. Thus, the input VM is not very
!       important. This is all the more reason to examine VM once the
!       algorithm is underway.
!    5. To see the effect of different parameters and their effect on
!       the speed of the algorithm, try RT = .95 & RT = .1. Notice the
!       vastly different speed for optimization. Also try NT = 20. Note
!       that this sample function is quite easy to optimize, so it will
!       tolerate big changes in these parameters. RT and NT are the
!       parameters one should adjust to modify the runtime of the
!       algorithm and its robustness.
!    6. Try constraining the algorithm with either LB or UB.


MODULE Simulated_anneal_module

! ABSTRACT:
!   Simulated annealing is a global optimization method that distinguishes
```

```
!   between different local optima. Starting from an initial point, the
!   algorithm takes a step and the function is evaluated. When minimizing a
!   function, any downhill step is accepted and the process repeats from this
!   new point. An uphill step may be accepted. Thus, it can escape from local
!   optima. This uphill decision is made by the Metropolis criteria. As the
!   optimization process proceeds, the length of the steps decline and the
!   algorithm closes in on the global optimum. Since the algorithm makes very
!   few assumptions regarding the function to be optimized, it is quite
!   robust with respect to non-quadratic surfaces. The degree of robustness
!   can be adjusted by the user. In fact, simulated annealing can be used as
!   a local optimizer for difficult functions.

!   This implementation of simulated annealing was used in "Global Optimizatio
!   of Statistical Functions with Simulated Annealing," Goffe, Ferrier and
!   Rogers, Journal of Econometrics, vol. 60, no. 1/2, Jan./Feb. 1994, pp.
!   65-100. Briefly, we found it competitive, if not superior, to multiple
!   restarts of conventional optimization routines for difficult optimization
!   problems.

!   For more information on this routine, contact its author:
!   Bill Goffe, bgoffe@whale.st.usm.edu

! This version in Fortran 90 has been prepared by Alan Miller.
! It is compatible with Lahey's ELF90 compiler.
! N.B. The 3 last arguments have been removed from subroutine sa. these
!      were work arrays and are now internal to the routine.
! e-mail: amiller @ bigpond.net.au
! URL  :  http://users.bigpond.net.au/amiller

! Latest revision of Fortran 90 version - 2 October 2013

IMPLICIT NONE

INTEGER, PARAMETER :: dp = SELECTED_REAL_KIND(14, 60)

! The following variables were in COMMON /raset1/
REAL, SAVE   :: u(97), cc, cd, cm
INTEGER, SAVE :: i97, j97


CONTAINS

SUBROUTINE sa(n, x, fcn, max, rt, eps, ns, nt, neps, maxevl, lb, ub, c, iprint, &
              iseed1, iseed2, t, vm, xopt, fopt, nacc, nfcnev, nobds, ier)

!  Version: 3.2
!  Date: 1/22/94.
!  Differences compared to Version 2.0:
!     1. If a trial is out of bounds, a point is randomly selected
!        from LB(i) to UB(i). Unlike in version 2.0, this trial is
```

```
103  !       evaluated and is counted in acceptances and rejections.
104  !       All corresponding documentation was changed as well.
105  !  Differences compared to Version 3.0:
106  !     1. If VM(i) > (UB(i) - LB(i)), VM is set to UB(i) - LB(i).
107  !        The idea is that if T is high relative to LB & UB, most
108  !        points will be accepted, causing VM to rise. But, in this
109  !        situation, VM has little meaning; particularly if VM is
110  !        larger than the acceptable region. Setting VM to this size
111  !        still allows all parts of the allowable region to be selected.
112  !  Differences compared to Version 3.1:
113  !     1. Test made to see if the initial temperature is positive.
114  !     2. WRITE statements prettied up.
115  !     3. References to paper updated.
116
117  !  Synopsis:
118  !  This routine implements the continuous simulated annealing global
119  !  optimization algorithm described in Corana et al.'s article "Minimizing
120  !  Multimodal Functions of Continuous Variables with the "Simulated Annealing"
121  !  Algorithm" in the September 1987 (vol. 13, no. 3, pp. 262-280) issue of
122  !  the ACM Transactions on Mathematical Software.
123
124  !  A very quick (perhaps too quick) overview of SA:
125  !     SA tries to find the global optimum of an N dimensional function.
126  !  It moves both up and downhill and as the optimization process
127  !  proceeds, it focuses on the most promising area.
128  !     To start, it randomly chooses a trial point within the step length
129  !  VM (a vector of length N) of the user selected starting point. The
130  !  function is evaluated at this trial point and its value is compared
131  !  to its value at the initial point.
132  !     In a maximization problem, all uphill moves are accepted and the
133  !  algorithm continues from that trial point. Downhill moves may be
134  !  accepted; the decision is made by the Metropolis criteria. It uses T
135  !  (temperature) and the size of the downhill move in a probabilistic
136  !  manner. The smaller T and the size of the downhill move are, the more
137  !  likely that move will be accepted. If the trial is accepted, the
138  !  algorithm moves on from that point. If it is rejected, another point
139  !  is chosen instead for a trial evaluation.
140  !     Each element of VM periodically adjusted so that half of all
141  !  function evaluations in that direction are accepted.
142  !     A fall in T is imposed upon the system with the RT variable by
143  !  T(i+1) = RT*T(i) where i is the ith iteration. Thus, as T declines,
144  !  downhill moves are less likely to be accepted and the percentage of
145  !  rejections rise. Given the scheme for the selection for VM, VM falls.
146  !  Thus, as T declines, VM falls and SA focuses upon the most promising
147  !  area for optimization.
148
149  !  The importance of the parameter T:
150  !     The parameter T is crucial in using SA successfully. It influences
151  !  VM, the step length over which the algorithm searches for optima. For
152  !  a small intial T, the step length may be too small; thus not enough
```

```
153  !  of the function might be evaluated to find the global optima. The user
154  !  should carefully examine VM in the intermediate output (set IPRINT =
155  !  1) to make sure that VM is appropriate. The relationship between the
156  !  initial temperature and the resulting step length is function
157  !  dependent.
158  !     To determine the starting temperature that is consistent with
159  !  optimizing a function, it is worthwhile to run a trial run first. Set
160  !  RT = 1.5 and T = 1.0. With RT > 1.0, the temperature increases and VM
161  !  rises as well. Then select the T that produces a large enough VM.
162  
163  !  For modifications to the algorithm and many details on its use,
164  !  (particularly for econometric applications) see Goffe, Ferrier
165  !  and Rogers, "Global Optimization of Statistical Functions with
166  !  Simulated Annealing," Journal of Econometrics, vol. 60, no. 1/2,
167  !  Jan./Feb. 1994, pp. 65-100.
168  !  For more information, contact
169  !              Bill Goffe
170  !              Department of Economics and International Business
171  !              University of Southern Mississippi
172  !              Hattiesburg, MS 39506-5072
173  !              (601) 266-4484 (office)
174  !              (601) 266-4920 (fax)
175  !              bgoffe@whale.st.usm.edu (Internet)
176  
177  !  As far as possible, the parameters here have the same name as in
178  !  the description of the algorithm on pp. 266-8 of Corana et al.
179  
180  !  In this description, SP is single precision, DP is double precision,
181  !  INT is integer, L is logical and (N) denotes an array of length n.
182  !  Thus, DP(N) denotes a double precision array of length n.
183  
184  !  Input Parameters:
185  !    Note: The suggested values generally come from Corana et al. To
186  !          drastically reduce runtime, see Goffe et al., pp. 90-1 for
187  !          suggestions on choosing the appropriate RT and NT.
188  !    N - Number of variables in the function to be optimized. (INT)
189  !    X - The starting values for the variables of the function to be
190  !        optimized. (DP(N))
191  !    FCN - Function to be maximized or minimized FCN(n, theta(n), h)
192  !    MAX - Denotes whether the function should be maximized or minimized.
193  !          A true value denotes maximization while a false value denotes
194  !          minimization. Intermediate output (see IPRINT) takes this into
195  !          account. (L)
196  !    RT - The temperature reduction factor. The value suggested by
197  !         Corana et al. is .85. See Goffe et al. for more advice. (DP)
198  !    EPS - Error tolerance for termination. If the final function
199  !          values from the last neps temperatures differ from the
200  !          corresponding value at the current temperature by less than
201  !          EPS and the final function value at the current temperature
202  !          differs from the current optimal function value by less than
```

```
!         EPS, execution terminates and IER = 0 is returned. (EP)
!    NS - Number of cycles. After NS*N function evaluations, each element of
!         VM is adjusted so that approximately half of all function evaluations
!         are accepted. The suggested value is 20. (INT)
!    NT - Number of iterations before temperature reduction. After
!         NT*NS*N function evaluations, temperature (T) is changed
!         by the factor RT. Value suggested by Corana et al. is
!         MAX(100, 5*N). See Goffe et al. for further advice. (INT)
!    NEPS - Number of final function values used to decide upon termi-
!           nation. See EPS. Suggested value is 4. (INT)
!    MAXEVL - The maximum number of function evaluations. If it is
!             exceeded, IER = 1. (INT)
!    LB - The lower bound for the allowable solution variables. (DP(N))
!    UB - The upper bound for the allowable solution variables. (DP(N))
!         If the algorithm chooses X(I) .LT. LB(I) or X(I) .GT. UB(I),
!         I = 1, N, a point is from inside is randomly selected. This
!         This focuses the algorithm on the region inside UB and LB.
!         Unless the user wishes to concentrate the search to a particular
!         region, UB and LB should be set to very large positive
!         and negative values, respectively. Note that the starting
!         vector X should be inside this region. Also note that LB and
!         UB are fixed in position, while VM is centered on the last
!         accepted trial set of variables that optimizes the function.
!    C - Vector that controls the step length adjustment. The suggested
!        value for all elements is 2.0. (DP(N))
!    IPRINT - controls printing inside SA. (INT)
!             Values: 0 - Nothing printed.
!                     1 - Function value for the starting value and
!                         summary results before each temperature
!                         reduction. This includes the optimal
!                         function value found so far, the total
!                         number of moves (broken up into uphill,
!                         downhill, accepted and rejected), the
!                         number of out of bounds trials, the
!                         number of new optima found at this
!                         temperature, the current optimal X and
!                         the step length VM. Note that there are
!                         N*NS*NT function evalutations before each
!                         temperature reduction. Finally, notice is
!                         is also given upon achieveing the termination
!                         criteria.
!                     2 - Each new step length (VM), the current optimal
!                         X (XOPT) and the current trial X (X). This
!                         gives the user some idea about how far X
!                         strays from XOPT as well as how VM is adapting
!                         to the function.
!                     3 - Each function evaluation, its acceptance or
!                         rejection and new optima. For many problems,
!                         this option will likely require a small tree
!                         if hard copy is used. This option is best
```

```
!                     used to learn about the algorithm. A small
!                     value for MAXEVL is thus recommended when
!                     using IPRINT = 3.
!           Suggested value: 1
!           Note: For a given value of IPRINT, the lower valued
!                 options (other than 0) are utilized.
!   ISEED1 - The first seed for the random number generator RANMAR.
!           0 <= ISEED1 <= 31328. (INT)
!   ISEED2 - The second seed for the random number generator RANMAR.
!           0 <= ISEED2 <= 30081. Different values for ISEED1
!           and ISEED2 will lead to an entirely different sequence
!           of trial points and decisions on downhill moves (when
!           maximizing). See Goffe et al. on how this can be used
!           to test the results of SA. (INT)

! Input/Output Parameters:
!   T - On input, the initial temperature. See Goffe et al. for advice.
!       On output, the final temperature. (DP)
!   VM - The step length vector. On input it should encompass the region of
!        interest given the starting value X. For point X(I), the next
!        trial point is selected is from X(I) - VM(I) to X(I) + VM(I).
!        Since VM is adjusted so that about half of all points are accepted,
!        the input value is not very important (i.e. is the value is off,
!        SA adjusts VM to the correct value). (DP(N))

! Output Parameters:
!   XOPT - The variables that optimize the function. (DP(N))
!   FOPT - The optimal value of the function. (DP)
!   NACC - The number of accepted function evaluations. (INT)
!   NFCNEV - The total number of function evaluations. In a minor
!           point, note that the first evaluation is not used in the
!           core of the algorithm; it simply initializes the
!           algorithm. (INT).
!   NOBDS - The total number of trial function evaluations that
!          would have been out of bounds of LB and UB. Note that
!          a trial point is randomly selected between LB and UB. (INT)
!   IER - The error return number. (INT)
!         Values: 0 - Normal return; termination criteria achieved.
!                 1 - Number of function evaluations (NFCNEV) is
!                     greater than the maximum number (MAXEVL).
!                 2 - The starting value (X) is not inside the
!                     bounds (LB and UB).
!                 3 - The initial temperature is not positive.
!                 99 - Should not be seen; only used internally.

! Work arrays that must be dimensioned in the calling routine:
!      RWK1 (DP(NEPS)) (FSTAR in SA)
!      RWK2 (DP(N))    (XP    "  " )
!      IWK  (INT(N))   (NACP  "  " )
! N.B. In the Fortran 90 version, these are automatic arrays.
```

```
303
304  !  Required Functions (included):
305  !    EXPREP - Replaces the function EXP to avoid under- and overflows.
306  !             It may have to be modified for non IBM-type main-
307  !             frames. (DP)
308  !    RMARIN - Initializes the random number generator RANMAR.
309  !    RANMAR - The actual random number generator. Note that
310  !             RMARIN must run first (SA does this). It produces uniform
311  !             random numbers on [0,1]. These routines are from
312  !             Usenet's comp.lang.fortran. For a reference, see
313  !             "Toward a Universal Random Number Generator"
314  !             by George Marsaglia and Arif Zaman, Florida State
315  !             University Report: FSU-SCRI-87-50 (1987).
316  !             It was later modified by F. James and published in
317  !             "A Review of Pseudo-random Number Generators." For
318  !             further information, contact stuart@ads.com. These
319  !             routines are designed to be portable on any machine
320  !             with a 24-bit or more mantissa. I have found it produces
321  !             identical results on a IBM 3081 and a Cray Y-MP.
322
323  !  Required Subroutines (included):
324  !    PRTVEC - Prints vectors.
325  !    PRT1 ... PRT10 - Prints intermediate output.
326  !    FCN - Function to be optimized. The form is
327  !             SUBROUTINE FCN(N, X, F)
328  !             IMPLICIT NONE
329  !             INTEGER, PARAMETER   :: dp = SELECTED_REAL_KIND(14, 60)
330  !             INTEGER, INTENT(IN)  :: N
331  !             REAL (dp), INTENT(IN) :: X(N)
332  !             REAL (dp), INTENT(OUT) :: F
333  !             ...
334  !             function code with F = F(X)
335  !             ...
336  !             RETURN
337  !             END
338  !  Note: This is the same form used in the multivariable
339  !        minimization algorithms in the IMSL edition 10 library.
340
341  !  Machine Specific Features:
342  !    1. EXPREP may have to be modified if used on non-IBM type main-
343  !       frames. Watch for under- and overflows in EXPREP.
344  !    2. Some FORMAT statements use G25.18; this may be excessive for
345  !       some machines.
346  !    3. RMARIN and RANMAR are designed to be protable; they should not
347  !       cause any problems.
348
349  !  Type all external variables.
350  REAL (dp), INTENT(IN)  :: lb(:), ub(:), c(:), eps, rt
351  REAL (dp), INTENT(IN OUT) :: x(:), t, vm(:)
352  REAL (dp), INTENT(OUT)  :: xopt(:), fopt
```

```
353  INTEGER, INTENT(IN)     :: n, ns, nt, neps, maxevl, iprint, iseed1, iseed2
354  INTEGER, INTENT(OUT)    :: nacc, nfcnev, nobds, ier
355  LOGICAL, INTENT(IN)     :: max
356  REAL (dp), EXTERNAL     :: fcn
357
358  ! Type all internal variables.
359  REAL (dp) :: f, fp, p, pp, ratio, xp(n), fstar(neps)
360  INTEGER  :: nup, ndown, nrej, nnew, lnobds, h, i, j, m, nacp(n)
361  LOGICAL  :: quit
362
363  INTERFACE
364     FUNCTION fcn(n, theta)
365         INTEGER, PARAMETER   :: dp = SELECTED_REAL_KIND(14, 60)
366         INTEGER, INTENT(IN)  :: n
367         REAL (dp), INTENT(IN) :: theta(:)
368     END FUNCTION fcn
369  END INTERFACE
370  ! Initialize the random number generator RANMAR.
371  CALL rmarin(iseed1, iseed2)
372  ! Set initial values.
373  nacc = 0
374  nobds = 0
375  nfcnev = 0
376  ier = 99
377
378  DO i = 1, n
379    xopt(i) = x(i)
380    nacp(i) = 0
381  END DO
382
383  fstar = 1.0D+20
384
385  ! If the initial temperature is not positive, notify the user and
386  ! return to the calling routine.
387  IF (t <= 0.0) THEN
388    WRITE(*,'(/, " THE INITIAL TEMPERATURE IS NOT POSITIVE. "/, &
389          &    "  reset the variable t. "/)')
390    ier = 3
391    RETURN
392  END IF
393
394  ! If the initial value is out of bounds, notify the user and return
395  ! to the calling routine.
396  DO i = 1, n
397    IF ((x(i) > ub(i)) .OR. (x(i) < lb(i))) THEN
398      CALL prt1()
399      ier = 2
400      RETURN
401    END IF
402  END DO
```

```
403
404  !  Evaluate the function with input X and return value as F.
405  f = fcn(n, x)
406
407  !  If the function is to be minimized, switch the sign of the function.
408  !  Note that all intermediate and final output switches the sign back
409  !  to eliminate any possible confusion for the user.
410  IF(.NOT. max) f = -f
411  nfcnev = nfcnev + 1
412  fopt = f
413  fstar(1) = f
414  IF(iprint >= 1) CALL prt2(max, n, x, f)
415
416  !  Start the main loop. Note that it terminates if (i) the algorithm
417  !  succesfully optimizes the function or (ii) there are too many
418  !  function evaluations (more than MAXEVL).
419  100 nup = 0
420  nrej = 0
421  nnew = 0
422  ndown = 0
423  lnobds = 0
424
425  DO m = 1, nt
426    DO j = 1, ns
427      DO h = 1, n
428
429  !  Generate XP, the trial value of X. Note use of VM to choose XP.
430        DO i = 1, n
431          IF (i == h) THEN
432            xp(i) = x(i) + (ranmar()*2. - 1.) * vm(i)
433          ELSE
434            xp(i) = x(i)
435          END IF
436
437  !  If XP is out of bounds, select a point in bounds for the trial.
438          IF((xp(i) < lb(i)) .OR. (xp(i) > ub(i))) THEN
439            xp(i) = lb(i) + (ub(i) - lb(i))*ranmar()
440            lnobds = lnobds + 1
441            nobds = nobds + 1
442            IF(iprint >= 3) CALL prt3(max, n, xp, x, f)
443          END IF
444        END DO
445
446  !  Evaluate the function with the trial point XP and return as FP.
447        fp = fcn(n, xp)
448        IF(.NOT. max) fp = -fp
449        nfcnev = nfcnev + 1
450        IF(iprint >= 3) CALL prt4(max, n, xp, x, fp, f)
451
452  !  If too many function evaluations occur, terminate the algorithm.
```

```
453       IF(nfcnev >= maxevl) THEN
454         CALL prt5()
455         IF (.NOT. max) fopt = -fopt
456         ier = 1
457         RETURN
458       END IF
459
460  !  Accept the new point if the function value increases.
461       IF(fp >= f) THEN
462         IF(iprint >= 3) THEN
463           WRITE(*,'(" POINT ACCEPTED")')
464         END IF
465         x(1:n) = xp(1:n)
466         f = fp
467         nacc = nacc + 1
468         nacp(h) = nacp(h) + 1
469         nup = nup + 1
470
471  !  If greater than any other point, record as new optimum.
472         IF (fp > fopt) THEN
473           IF(iprint >= 3) THEN
474             WRITE(*,'(" NEW OPTIMUM")')
475           END IF
476           xopt(1:n) = xp(1:n)
477           fopt = fp
478           nnew = nnew + 1
479         END IF
480
481  !  If the point is lower, use the Metropolis criteria to decide on
482  !  acceptance or rejection.
483       ELSE
484         p = exprep((fp - f)/t)
485         pp = ranmar()
486         IF (pp < p) THEN
487           IF(iprint >= 3) CALL prt6(max)
488           x(1:n) = xp(1:n)
489           f = fp
490           nacc = nacc + 1
491           nacp(h) = nacp(h) + 1
492           ndown = ndown + 1
493         ELSE
494           nrej = nrej + 1
495           IF(iprint >= 3) CALL prt7(max)
496         END IF
497       END IF
498
499     END DO
500   END DO
501
502  !  Adjust VM so that approximately half of all evaluations are accepted.
```

```
503   DO i = 1, n
504     ratio = DBLE(nacp(i)) /DBLE(ns)
505     IF (ratio > .6) THEN
506       vm(i) = vm(i)*(1. + c(i)*(ratio - .6)/.4)
507     ELSE IF (ratio < .4) THEN
508       vm(i) = vm(i)/(1. + c(i)*((.4 - ratio)/.4))
509     END IF
510     IF (vm(i) > (ub(i)-lb(i))) THEN
511       vm(i) = ub(i) - lb(i)
512     END IF
513   END DO
514
515   IF(iprint >= 2) THEN
516     CALL prt8(n, vm, xopt, x)
517   END IF
518
519   nacp(1:n) = 0
520
521 END DO
522
523 IF(iprint >= 1) THEN
524
525   CALL prt9(max,n,t,xopt,vm,fopt,nup,ndown,nrej,lnobds,nnew)
526 END IF
527
528 ! Check termination criteria.
529 quit = .false.
530 fstar(1) = f
531 IF ((fopt - fstar(1)) <= eps) quit = .true.
532 DO i = 1, neps
533   IF (ABS(f - fstar(i)) > eps) quit = .false.
534 END DO
535
536 ! Terminate SA if appropriate.
537 IF (quit) THEN
538   x(1:n) = xopt(1:n)
539   ier = 0
540   IF (.NOT. max) fopt = -fopt
541   IF(iprint >= 1) CALL prt10()
542   RETURN
543 END IF
544
545 ! If termination criteria is not met, prepare for another loop.
546 t = rt*t
547 DO i = neps, 2, -1
548   fstar(i) = fstar(i-1)
549 END DO
550 f = fopt
551 x(1:n) = xopt(1:n)
552
```

```
553  !  Loop again.
554
555  GO TO 100
556  END SUBROUTINE sa
557
558
559  FUNCTION exprep(rdum) RESULT(fn_val)
560  !  This function replaces exp to avoid under- and overflows and is
561  !  designed for IBM 370 type machines. It may be necessary to modify
562  !  it for other machines. Note that the maximum and minimum values of
563  !  EXPREP are such that they have no effect on the algorithm.
564
565  REAL (dp), INTENT(IN) :: rdum
566  REAL (dp)             :: fn_val
567
568  IF (rdum > 174._dp) THEN
569    fn_val = 3.69D+75
570  ELSE IF (rdum < -180._dp) THEN
571    fn_val = 0.0_dp
572  ELSE
573    fn_val = EXP(rdum)
574  END IF
575
576  RETURN
577  END FUNCTION exprep
578
579
580  SUBROUTINE rmarin(ij, kl)
581  !  This subroutine and the next function generate random numbers. See
582  !  the comments for SA for more information. The only changes from the
583  !  orginal code is that (1) the test to make sure that RMARIN runs first
584  !  was taken out since SA assures that this is done (this test didn't
585  !  compile under IBM's VS Fortran) and (2) typing ivec as integer was
586  !  taken out since ivec isn't used. With these exceptions, all following
587  !  lines are original.
588
589  ! This is the initialization routine for the random number generator
590  !    RANMAR()
591  ! NOTE: The seed variables can have values between: 0 <= IJ <= 31328
592  !                                                  0 <= KL <= 30081
593
594  INTEGER, INTENT(IN) :: ij, kl
595
596  INTEGER :: i, j, k, l, ii, jj, m
597  REAL    :: s, t
598
599  IF( ij < 0 .OR. ij > 31328 .OR. kl < 0 .OR. kl > 30081 ) THEN
600    WRITE(*, '(A)') ' The first random number seed must have a value ', &
601                 'between 0 AND 31328'
602    WRITE(*, '(A)') ' The second seed must have a value between 0 and 30081'
```

```
  STOP
END IF
i = MOD(ij/177, 177) + 2
j = MOD(ij   , 177) + 2
k = MOD(kl/169, 178) + 1
l = MOD(kl,    169)
DO ii = 1, 97
  s = 0.0
  t = 0.5
  DO jj = 1, 24
    m = MOD(MOD(i*j, 179)*k, 179)
    i = j
    j = k
    k = m
    l = MOD(53*l+1, 169)
    IF (MOD(l*m, 64) >= 32) THEN
      s = s + t
    END IF
    t = 0.5 * t
  END DO
  u(ii) = s
END DO
cc = 362436.0 / 16777216.0
cd = 7654321.0 / 16777216.0
cm = 16777213.0 /16777216.0
i97 = 97
j97 = 33
RETURN
END SUBROUTINE rmarin


FUNCTION ranmar() RESULT(fn_val)
REAL :: fn_val

! Local variable
REAL :: uni

uni = u(i97) - u(j97)
IF( uni < 0.0 ) uni = uni + 1.0
u(i97) = uni
i97 = i97 - 1
IF(i97 == 0) i97 = 97
j97 = j97 - 1
IF(j97 == 0) j97 = 97
cc = cc - cd
IF( cc < 0.0 ) cc = cc + cm
uni = uni - cc
IF( uni < 0.0 ) uni = uni + 1.0
fn_val = uni

```

```
653  RETURN
654  END FUNCTION ranmar
655
656
657  SUBROUTINE prt1()
658  !  This subroutine prints intermediate output, as does PRT2 through
659  !  PRT10. Note that if SA is minimizing the function, the sign of the
660  !  function value and the directions (up/down) are reversed in all
661  !  output to correspond with the actual function optimization. This
662  !  correction is because SA was written to maximize functions and
663  !  it minimizes by maximizing the negative a function.
664
665  WRITE(*, '(/, " THE STARTING VALUE (X) IS OUTSIDE THE BOUNDS "/, &
666       &    "  (lb AND ub). execution terminated without any"/, &
667       &    "  optimization. respecify x, ub OR lb so that "/, &
668       &    "  lb(i) < x(i) < ub(i), i = 1, n. "/)')
669
670  RETURN
671  END SUBROUTINE prt1
672
673
674  SUBROUTINE prt2(max, n, x, f)
675
676  REAL (dp), INTENT(IN) :: x(:), f
677  INTEGER, INTENT(IN) :: n
678  LOGICAL, INTENT(IN) :: max
679
680  WRITE(*, '(" ")')
681  CALL prtvec(x,n,'INITIAL X')
682  IF (max) THEN
683    WRITE(*, '(" INITIAL F: ",/, G25.18)') f
684  ELSE
685    WRITE(*, '(" INITIAL F: ",/, G25.18)') -f
686  END IF
687
688  RETURN
689  END SUBROUTINE prt2
690
691
692  SUBROUTINE prt3(max, n, xp, x, f)
693
694  REAL (dp), INTENT(IN) :: xp(:), x(:), f
695  INTEGER, INTENT(IN) :: n
696  LOGICAL, INTENT(IN) :: max
697
698  WRITE(*, '(" ")')
699  CALL prtvec(x, n, 'CURRENT X')
700  IF (max) THEN
701    WRITE(*, '(" CURRENT F: ", G25.18)') f
702  ELSE
```

```
703     WRITE(*, '(" CURRENT F: ", G25.18)') -f
704   END IF
705   CALL prtvec(xp, n, 'TRIAL X')
706   WRITE(*, '(" POINT REJECTED SINCE OUT OF BOUNDS")')
707
708   RETURN
709   END SUBROUTINE prt3
710
711
712   SUBROUTINE prt4(max, n, xp, x, fp, f)
713
714   REAL (dp), INTENT(IN) :: xp(:), x(:), fp, f
715   INTEGER, INTENT(IN) :: n
716   LOGICAL, INTENT(IN) :: max
717
718   WRITE(*,'(" ")')
719   CALL prtvec(x,n,'CURRENT X')
720   IF (max) THEN
721     WRITE(*,'(" CURRENT F: ",G25.18)') f
722     CALL prtvec(xp,n,'TRIAL X')
723     WRITE(*,'(" RESULTING F: ",G25.18)') fp
724   ELSE
725     WRITE(*,'(" CURRENT F: ",G25.18)') -f
726     CALL prtvec(xp,n,'TRIAL X')
727     WRITE(*,'(" RESULTING F: ",G25.18)') -fp
728   END IF
729
730   RETURN
731   END SUBROUTINE prt4
732
733
734   SUBROUTINE prt5()
735
736   WRITE(*, '(/, " TOO MANY FUNCTION EVALUATIONS; CONSIDER "/, &
737        &     "  increasing maxevl OR eps, OR decreasing "/, &
738        &     "  nt OR rt. these results are likely TO be "/, " poor.",/)')
739
740   RETURN
741   END SUBROUTINE prt5
742
743
744   SUBROUTINE prt6(max)
745   LOGICAL, INTENT(IN) :: max
746
747   IF (max) THEN
748     WRITE(*,'(" THOUGH LOWER, POINT ACCEPTED")')
749   ELSE
750     WRITE(*,'(" THOUGH HIGHER, POINT ACCEPTED")')
751   END IF
752
```

```
753  RETURN
754  END SUBROUTINE prt6
755
756
757  SUBROUTINE prt7(max)
758  LOGICAL, INTENT(IN) :: max
759
760  IF (max) THEN
761    WRITE(*,'(" LOWER POINT REJECTED")')
762  ELSE
763    WRITE(*,'(" HIGHER POINT REJECTED")')
764  END IF
765
766  RETURN
767  END SUBROUTINE prt7
768
769
770  SUBROUTINE prt8(n, vm, xopt, x)
771
772  REAL (dp), INTENT(IN) :: vm(:), xopt(:), x(:)
773  INTEGER, INTENT(IN) :: n
774
775  WRITE(*,'(/, " intermediate results after step length adjustment", /)')
776  CALL prtvec(vm, n, 'NEW STEP LENGTH (VM)')
777  CALL prtvec(xopt, n, 'CURRENT OPTIMAL X')
778  CALL prtvec(x, n, 'CURRENT X')
779  WRITE(*,'(" ")')
780
781  RETURN
782  END SUBROUTINE prt8
783
784
785  SUBROUTINE prt9(max, n, t, xopt, vm, fopt, nup, ndown, nrej, lnobds, nnew)
786
787  REAL (dp), INTENT(IN) :: xopt(:), vm(:), t, fopt
788  INTEGER, INTENT(IN) :: n, nup, ndown, nrej, lnobds, nnew
789  LOGICAL, INTENT(IN) :: max
790
791  ! Local variable
792  INTEGER :: totmov
793
794  totmov = nup + ndown + nrej
795
796  WRITE(*,'(/," intermediate results before next temperature reduction",/)')
797  WRITE(*,'(" CURRENT TEMPERATURE:         ",G12.5)') t
798  IF (max) THEN
799    WRITE(*, '(" MAX FUNCTION VALUE SO FAR: ",G25.18)') fopt
800    WRITE(*, '(" TOTAL MOVES:              ",I8)') totmov
801    WRITE(*, '("    UPHILL:                ",I8)') nup
802    WRITE(*, '("    ACCEPTED DOWNHILL:     ",I8)') ndown
```

```
803   WRITE(*, '("   REJECTED DOWNHILL:     ",I8)') nrej
804   WRITE(*, '(" OUT OF BOUNDS TRIALS:    ",I8)') lnobds
805   WRITE(*, '(" NEW MAXIMA THIS TEMPERATURE:",I8)') nnew
806  ELSE
807   WRITE(*, '(" MIN FUNCTION VALUE SO FAR: ",G25.18)') -fopt
808   WRITE(*, '(" TOTAL MOVES:             ",I8)') totmov
809   WRITE(*, '("   DOWNHILL:              ",I8)') nup
810   WRITE(*, '("   ACCEPTED UPHILL:       ",I8)') ndown
811   WRITE(*, '("   REJECTED UPHILL:       ",I8)') nrej
812   WRITE(*, '(" TRIALS OUT OF BOUNDS:    ",I8)') lnobds
813   WRITE(*, '(" NEW MINIMA THIS TEMPERATURE:",I8)') nnew
814  END IF
815  CALL prtvec(xopt, n, 'CURRENT OPTIMAL X')
816  CALL prtvec(vm, n, 'STEP LENGTH (VM)')
817  WRITE(*, '(" ")')
818
819  RETURN
820  END SUBROUTINE prt9
821
822
823  SUBROUTINE prt10()
824
825  WRITE(*, '(/, " SA ACHIEVED TERMINATION CRITERIA. IER = 0. ",/)')
826
827  RETURN
828  END SUBROUTINE prt10
829
830
831  SUBROUTINE prtvec(vector, ncols, name)
832  !  This subroutine prints the double precision vector named VECTOR.
833  !  Elements 1 thru NCOLS will be printed. NAME is a character variable
834  !  that describes VECTOR. Note that if NAME is given in the call to
835  !  PRTVEC, it must be enclosed in quotes. If there are more than 10
836  !  elements in VECTOR, 10 elements will be printed on each line.
837
838  INTEGER, INTENT(IN)        :: ncols
839  REAL (dp), INTENT(IN)      :: vector(ncols)
840  CHARACTER (LEN=*), INTENT(IN) :: name
841
842  INTEGER :: i, lines, ll
843
844  WRITE(*,1001) NAME
845
846  IF (ncols > 10) THEN
847    lines = INT(ncols/10.)
848
849    DO i = 1, lines
850      ll = 10*(i - 1)
851      WRITE(*,1000) vector(1+ll:10+ll)
852    END DO
```

```
853
854     WRITE(*,1000) vector(11+ll:ncols)
855   ELSE
856     WRITE(*,1000) vector(1:ncols)
857   END IF
858
859   1000 FORMAT( 10(g12.5, ' '))
860   1001 FORMAT(/, 25(' '), a)
861
862   RETURN
863   END SUBROUTINE prtvec
864
865   END MODULE
866
867   !
868   !
869   !PROGRAM simann
870   !
871   !USE simulated_anneal
872   !IMPLICIT NONE
873   !INTEGER, PARAMETER :: n = 2, neps = 4
874   !
875   !REAL (dp)  :: lb(n), ub(n), x(n), xopt(n), c(n), vm(n), t, eps, rt, fopt
876   !
877   !INTEGER    :: ns, nt, nfcnev, ier, iseed1, iseed2, i, maxevl, iprint, &
878   !              nacc, nobds
879   !
880   !LOGICAL    :: max
881   !
882   !! Set underflows to zero on IBM mainframes.
883   !!    CALL XUFLOW(0)
884   !
885   !! Set input parameters.
886   !max = .false.
887   !eps = 1.0D-6
888   !rt = .5
889   !iseed1 = 1
890   !iseed2 = 2
891   !ns = 20
892   !nt = 5
893   !maxevl = 100000
894   !iprint = 1
895   !DO i = 1, n
896   !  lb(i) = -1.0D25
897   !  ub(i) = 1.0D25
898   !  c(i) = 2.0
899   !END DO
900   !
901   !! Note start at local, but not global, optima of the Judge function.
902   !x(1) = 2.354471
```

```
!x(2) = -0.319186
!
!! Set input values of the input/output parameters.
!t = 5.0
!vm(1:n) = 1.0
!
!WRITE(*,1000) n, max, t, rt, eps, ns, nt, neps, maxevl, iprint, iseed1, iseed2
!
!CALL prtvec(x, n, 'STARTING VALUES')
!CALL prtvec(vm, n, 'INITIAL STEP LENGTH')
!CALL prtvec(lb, n, 'LOWER BOUND')
!CALL prtvec(ub, n, 'UPPER BOUND')
!CALL prtvec(c, n, 'C VECTOR')
!WRITE(*, '(/, " **** END OF DRIVER ROUTINE OUTPUT ****"/, &
!     &    "  ****  before CALL TO sa.          ****")')
!
!CALL sa(n, x, max, rt, eps, ns, nt, neps, maxevl, lb, ub, c, iprint, iseed1, &
!       iseed2, t, vm, xopt, fopt, nacc, nfcnev, nobds, ier)
!
!WRITE(*, '(/, " **** RESULTS AFTER SA ****  ")')
!CALL prtvec(xopt, n, 'SOLUTION')
!CALL prtvec(vm, n, 'FINAL STEP LENGTH')
!WRITE(*,1001) fopt, nfcnev, nacc, nobds, t, ier
!
!1000 FORMAT(/,' SIMULATED ANNEALING EXAMPLE',/,/, &
!            ' NUMBER OF PARAMETERS: ',i3,' MAXIMIZATION: ',l5, /, &
!            ' INITIAL TEMP: ', g8.2, ' RT: ',g8.2, ' EPS: ',g8.2, /, &
!            ' NS: ',i3, ' NT: ',i2, '  NEPS: ',i2, /, &
!            ' MAXEVL: ',i10, ' IPRINT: ',i1, ' ISEED1: ',i4, &
!            '   ISEED2: ',i4)
!1001 FORMAT(/,' OPTIMAL FUNCTION VALUE: ',g20.13 &
!           /,' NUMBER OF FUNCTION EVALUATIONS: ',i10, &
!           /,' NUMBER OF ACCEPTED EVALUATIONS: ',i10, &
!           /,' NUMBER OF OUT OF BOUND EVALUATIONS: ',i10, &
!           /,' FINAL TEMP: ', g20.13,' IER: ', i3)
!
!STOP
!END PROGRAM simann
!
!
!SUBROUTINE fcn(n, theta, h)
!! This subroutine is from the example in Judge et al., The Theory and
!! Practice of Econometrics, 2nd ed., pp. 956-7. There are two optima:
!! F(.864,1.23) = 16.0817 (the global minumum) and F(2.35,-.319) = 20.9805.
!
!IMPLICIT NONE
!INTEGER, PARAMETER   :: dp = SELECTED_REAL_KIND(14, 60)
!
!INTEGER, INTENT(IN)  :: n
!REAL (dp), INTENT(IN) :: theta(:)
```

```
!REAL (dp), INTENT(OUT) :: h
!
!! Local variables
!INTEGER  :: i
!REAL (dp) :: y(20), x2(20), x3(20)
!
!y(1) = 4.284
!y(2) = 4.149
!y(3) = 3.877
!y(4) = 0.533
!y(5) = 2.211
!y(6) = 2.389
!y(7) = 2.145
!y(8) = 3.231
!y(9) = 1.998
!y(10) = 1.379
!y(11) = 2.106
!y(12) = 1.428
!y(13) = 1.011
!y(14) = 2.179
!y(15) = 2.858
!y(16) = 1.388
!y(17) = 1.651
!y(18) = 1.593
!y(19) = 1.046
!y(20) = 2.152
!
!x2(1) = .286
!x2(2) = .973
!x2(3) = .384
!x2(4) = .276
!x2(5) = .973
!x2(6) = .543
!x2(7) = .957
!x2(8) = .948
!x2(9) = .543
!x2(10) = .797
!x2(11) = .936
!x2(12) = .889
!x2(13) = .006
!x2(14) = .828
!x2(15) = .399
!x2(16) = .617
!x2(17) = .939
!x2(18) = .784
!x2(19) = .072
!x2(20) = .889
!
!x3(1) = .645
!x3(2) = .585
```

```
!x3(3) = .310
!x3(4) = .058
!x3(5) = .455
!x3(6) = .779
!x3(7) = .259
!x3(8) = .202
!x3(9) = .028
!x3(10) = .099
!x3(11) = .142
!x3(12) = .296
!x3(13) = .175
!x3(14) = .180
!x3(15) = .842
!x3(16) = .039
!x3(17) = .103
!x3(18) = .620
!x3(19) = .158
!x3(20) = .704
!
!h = 0.0_dp
!DO i = 1, 20
!  h = (theta(1) + theta(n)*x2(i) + (theta(n)**2)*x3(i) - y(i))**2 + h
!END DO
!
!RETURN
!END SUBROUTINE fcn
```

8.11 Simulated annealing main code

```
program Simulated_anneal_tests

    use Simulated_anneal_module

    implicit none

    integer :: test_num
    real(8) :: temperature, f_opt
    real(8), dimension(2) :: step_vector, vect, vect_opt
    integer :: num_accepted, num_func_evals, num_out_bounds, error_code
    logical :: find_max

    real(8) :: time_begin, time_end

    do test_num = 0, 2
        write(*,*) "-------------------------------"
        write(*,*) "Begin test ", test_num
        write(*,*) "-------------------------------"
        step_vector = (/ 2.0, 2.0 /)
```

```
20          temperature = 1.0d0
21          vect = (/ -10.0d0, -10.0d0 /)
22          if (test_num == 2) then
23              find_max = .false.
24          else
25              find_max = .true.
26          end if
27
28          call cpu_time(time_begin)
29          call sa( n = 2,                          &
30                  x = vect,                        &
31                  fcn = func,                      &
32                  max = find_max,                  &
33                  rt = 0.85d0,                     &
34                  eps = 1.0d-06,                   &
35                  Ns = 20,                         &
36                  Nt = max(100, 5*2),              &
37                  Neps = 4,                        &
38                  maxevl = 10000000,               &
39                  lb = (/ -1.0d2, -1.0d2 /),       &
40                  ub = (/ +1.0d2, +1.0d2 /),       &
41                  c = (/ 2.0d0, 2.0d0 /),          &
42                  iprint = 1,                      &
43                  iseed1 = 12047,                  &
44                  iseed2 = 21013,                  &
45                  t = temperature,                 &
46                  vm = step_vector,                &
47                  xopt = vect_opt,                 &
48                  fopt = f_opt,                    &
49                  nacc = num_accepted,             &
50                  nfcnev = num_func_evals,         &
51                  nobds = num_out_bounds,          &
52                  ier = error_code )
53
54          call cpu_time(time_end)
55          write(*,*) "Execution time: ", time_end - time_begin
56          write(*,*) "-------------------------------"
57          write(*,*) "End test ", test_num
58          write(*,*) "-------------------------------"
59
60      end do
61
62  contains
63
64      real(8) function func(n, vect)
65
66          integer, intent(in) :: n
67          real(8), intent(in), dimension(:) :: vect
68
69          real(8) :: x, y
```

```
70
71          x = vect(1)
72          y = vect(2)
73
74          if (test_num == 0) then
75              func = exp( -(x**2 + y**2) )
76          else if (test_num == 1) then
77              func = exp( -(x**2 + y**2) ) + 2*exp( -( (x-1.7)**2 + (y-1.7)**2 ) )
78          else if (test_num == 2) then
79              func = (1.0 - x)**2 + 100.0*(y - x**2)**2
80          else
81              func = 0.0
82          end if
83
84      end function
85
86  end program
```