H2b Variational Monte Carlo

Ole Fjeldså & Gustaf Jonasson Johansson

December 13, 2024

Task №	Points	Avail. points
Σ		

In this report energy and distance is referred to as atomic units [a.u.]. In the case of length, the atomic unit is the Bohr radius $a_0 = 0.529\text{Å}$, while the atomic unit for energy is the Hartree energy $E_H = 27.2eV$.

The code we used is presented in Appendix A.

Problem 1

The trial wave-function

$$\Psi_T(\mathbf{r}_1, \mathbf{r}_2) = exp[-2r_1]\ exp[-2r_2]\ exp\left[\frac{r_{12}}{2(1+\alpha r_{12})}\right] \tag{1}$$

Central field approximation

$$Z^3 4r^2 e^{-2Zr} \tag{2}$$

Figure 1 shows the simulated density function for finding an electron at distance r a.u. away from the centre of the helium nucleus. As one can observe from the figure the simulated probability closely matches the central field approximation (2) with the variationally optimized value $Z = \frac{27}{16}$.

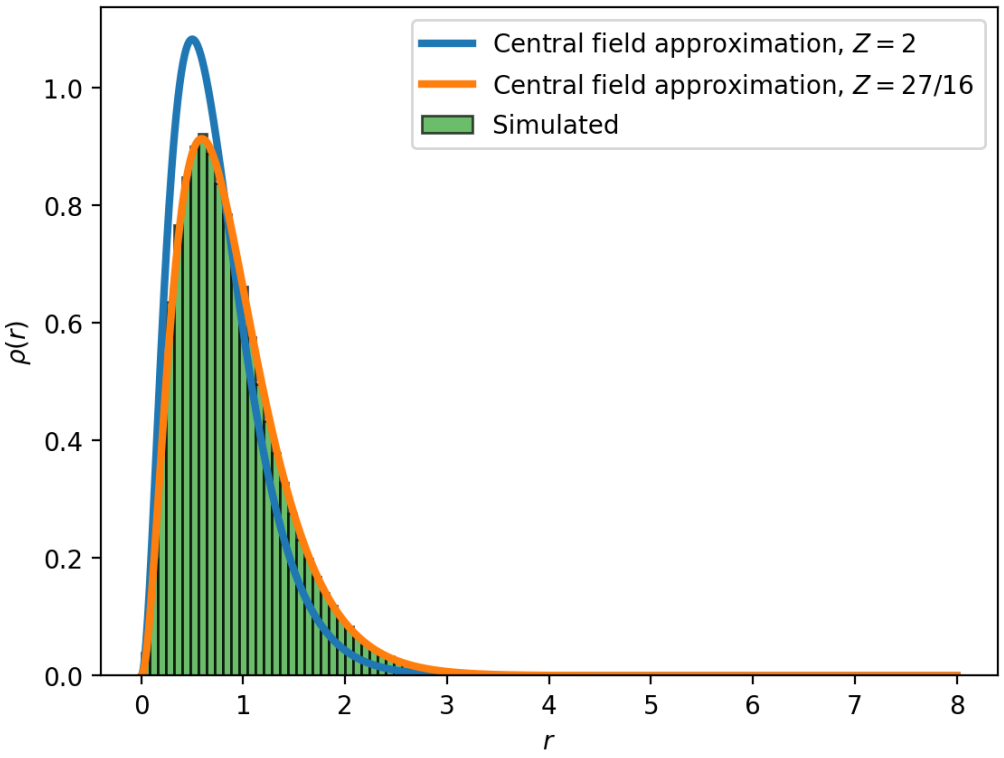

Figure 1: The probability of finding an electron at distance r from origo. The green histogram is from a simulated run, and the blue and orange lines are central field approximations. The simulation was run for 10^6 time steps.

The left panel of Figure 2 shows the simulated probabilities of the two helium electrons to have angle $\theta \in (0, \pi)$. The line $\frac{\sin(\theta)}{2}$ represents the expected density function of for completely uncorrelated electrons. The right panel of Figure 2 show the cosine of the simulated angle distribution, again if the two electrons moved independently the distribution would closely follow the line $P(x) = \frac{1}{2}$.

The update rule for the electron positions is $\left(\frac{\Psi_T(\mathbf{r}_1', \mathbf{r}_2')}{\Psi_T(\mathbf{r}_1, \mathbf{r}_2)}\right)^2 > R$ where $\mathbf{r}_i$ are the old electron positions, $\mathbf{r}_i'$ are the new positions and $R \in (0, 1)$ is a uniformly sampled random number. Since equation (1) depends on both $\mathbf{r}_1$ and $\mathbf{r}_2$ the two are correlated, skewing the distribution towards larger θ.

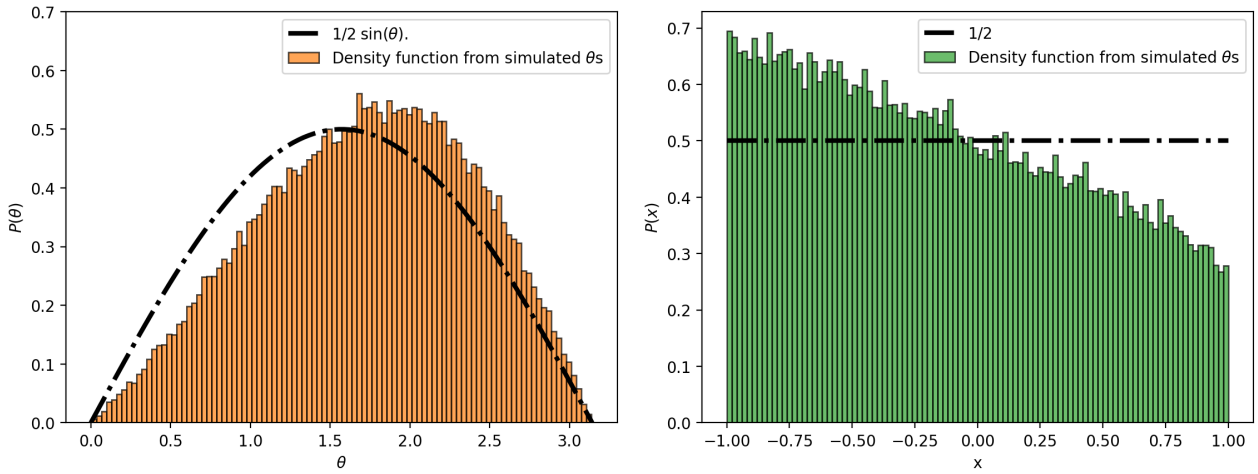

Figure 2: The left panel shows the probability density function for the angle between the two electrons. The orange histogram is from a simulated run, and the black dotted line is the function $p(\theta) = \frac{1}{2} sin(\theta)$ from 0 to π. The right panel shows the probability density for $cos(\theta)$, where θ is the angle between the two electrons. The black line shows what a uniform distribution looks like. The histogram would follow the black line if the electrons were uncorrelated. The simulation was run for 10^6 time steps.

Figure 3 shows the energy E_L and its time average over one million time steps. As we can see from the figure the average hovers just above -3 a.u.. The mean of the time average after it has stabilized is -2.878886 a.u..

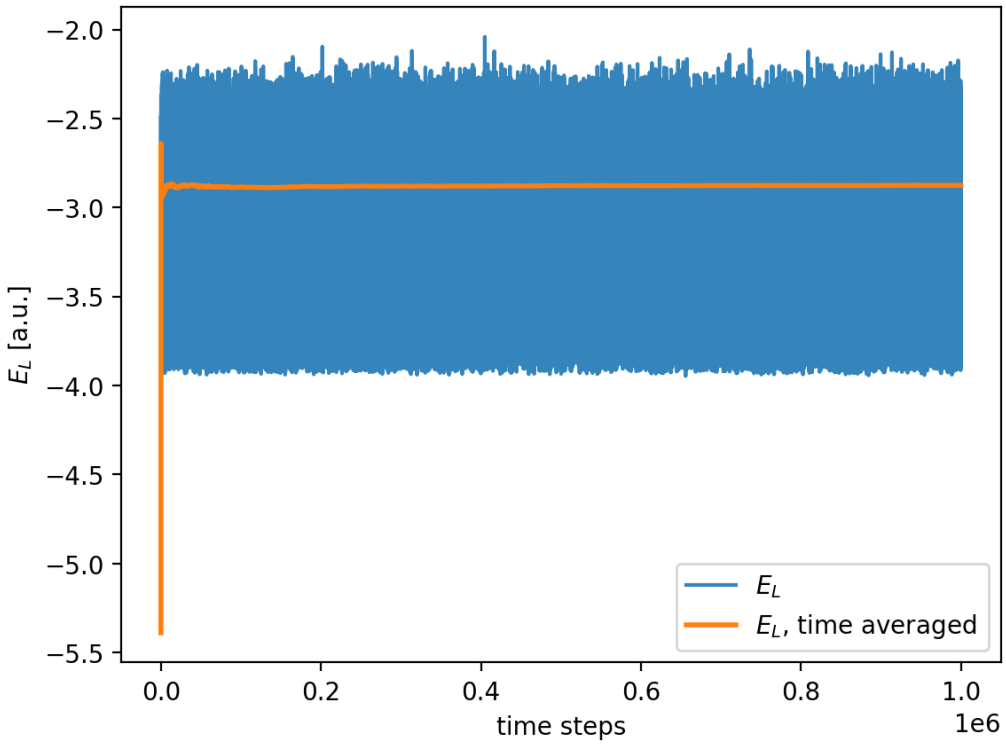

Figure 3: Energy E_L and the time average of E_L over a simulation of one million time steps. The time average energy converges to -2.87575 a.u., slightly above the ground state for helium at -2.90372 a.u..

Problem 2

Here, we started by initializing the electrons to unlikely positions, $r_1 = (75, 0, 0)$, and $r_2 = (0, -75, 0)$. Figure 4 shows the energy plotted against time. The gray area represents what we estimate to be the equilibration phase, roughly 500 time steps in this case.

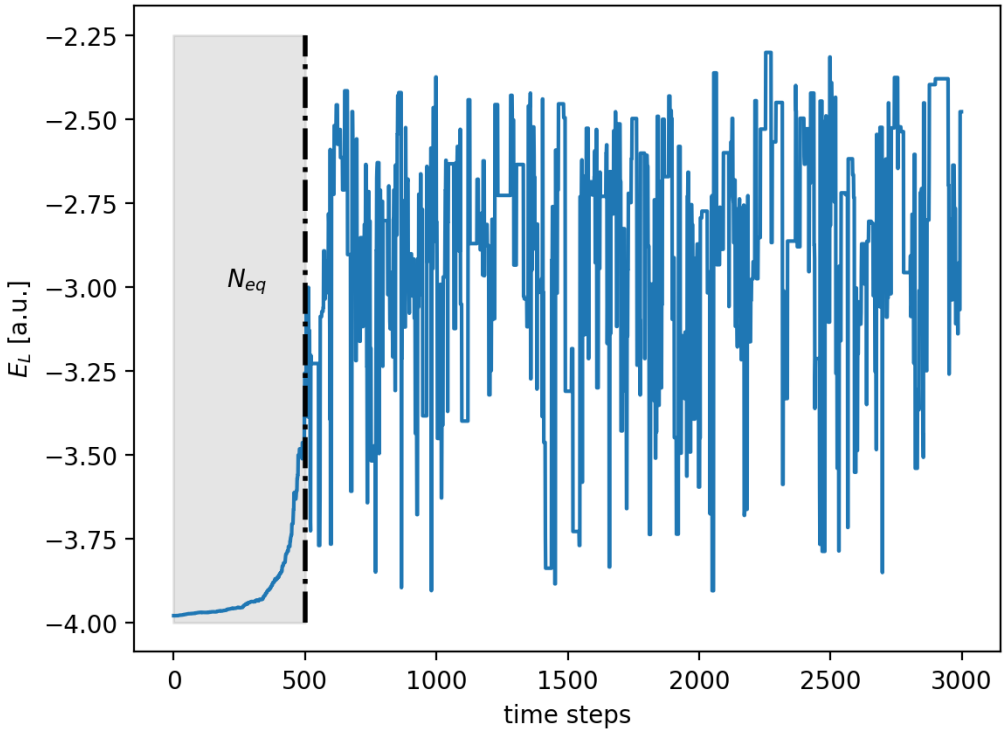

Figure 4: Local energy plotted against time. The gray area represents the equilibration phase. The initial positions of the electrons were $r_1 = (75, 0, 0)$, and $r_2 = (0, -75, 0)$. Here, we estimate that the equilibration phase is roughly 500 time steps.

Next, we estimated the statistical inefficiency using autocorrelation and block averaging. The left panel of Figure 5 shows the results for autocorrelation. We can see that $\Phi(k)$ appears to decay exponentially. The closest we got to $e^{-2} \approx 0.135$ was 0.133 at a time lag of 18, and so this is what we estimate the statistical inefficiency to be.

The right panel of Figure 5 shows the results obtained using block averaging. The figure shows the results for block sizes 1 to 300, and the orange line is the average statistical inefficiency for block sizes 100 to 2000. 100 was picked because the statistical inefficiency appears to have roughly converged at this point. We obtained a statistical inefficiency of 18.229, which is very close to the number we obtained when using autocorrelation.

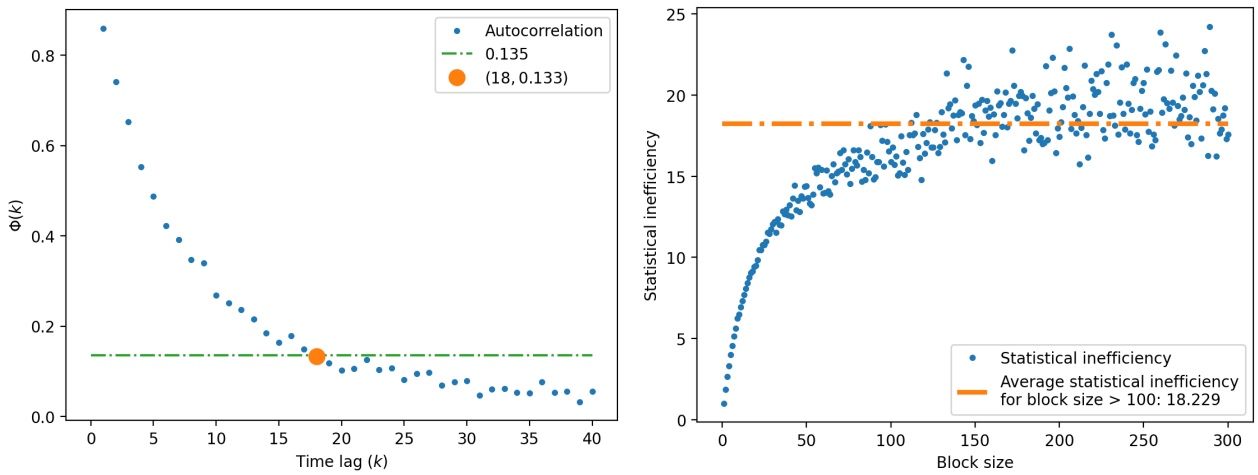

Figure 5: The left panel shows autocorrelation plotted against time lag (k). The green dashed line is plotted at e^{-2}. The closest point to this line corresponds to a time lag of 18. The right panel shows statistical inefficiency, based on block averaging, plotted against block size. The orange dashed line represents the average statistical inefficiency for block sizes 100 to 2000. The average statistical inefficiency is 18.229. Here, only block sizes up to 300 are plotted. Every data point is an average of 10 runs with 10^4 time steps and an equilibration phase of 10^3 time steps.

Problem 3

Here, we ran multiple simulations for different α values between 0.05 and 0.25, as shown in Figure 6. The optimal energy value is the lowest. The lowest value we achieved was roughly -2.880 a.u., and the corresponding α is roughly 0.15.

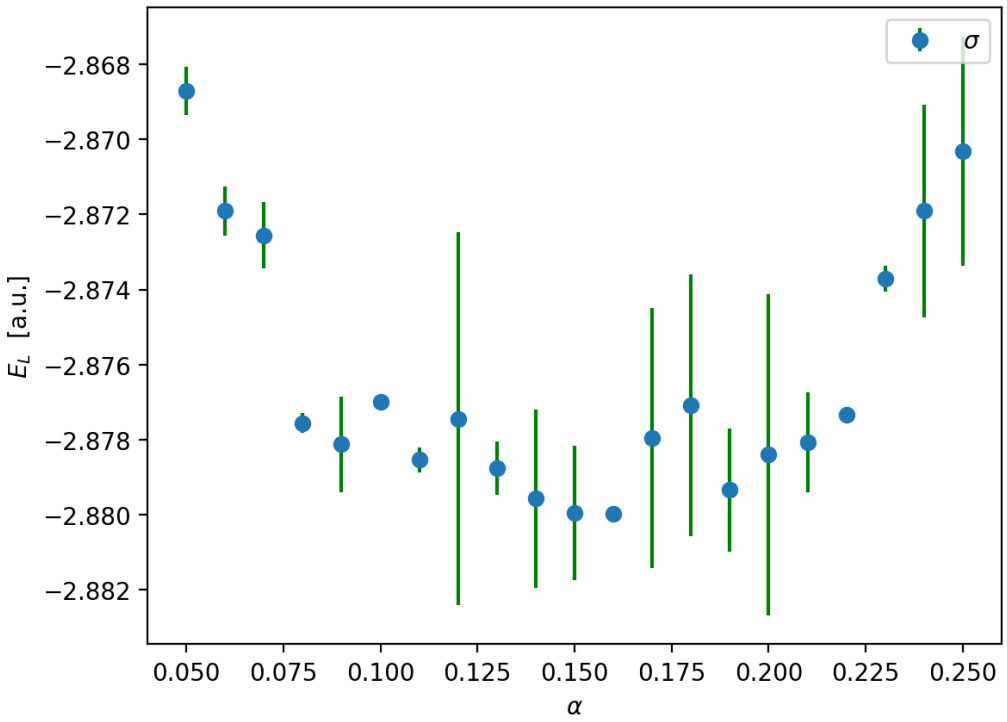

Figure 6: Time averaged energy, and corresponding error bars, for α values between 0.05 and 0.25. The lowest energy appears to be achieved at α values around 0.15. Each data point is an average of 200 simulations with 10^5 time steps and an equilibration phase of 10^3 time steps.

Problem 4

Here, we tested the following β values: 0.6, 0.7, 0.8, 0.9, 1. For each β, we ran 200 simulations with $2 * 10^5$ time steps and an equilibration phase of 500 time steps. In each run, α was updated every 20 time steps. Then, we computed the average α for each β. The results are summarised in Table 1.

β	α
0.6	0.1467
0.7	0.1427
0.8	0.1471
0.9	0.1357
1	0.1236

Table 1: The β values and their corresponding α values.

As shown in the table, we obtain roughly the same optimal α value compared to task 3.

Problem 5

Figure 7 shows a run with the calibrated $\alpha = 0.15$, The time averaged energy converge to -2.868892 a.u. placing it right in between the Hartree value -2.862 a.u. and the experimental number -2.9033 a.u..

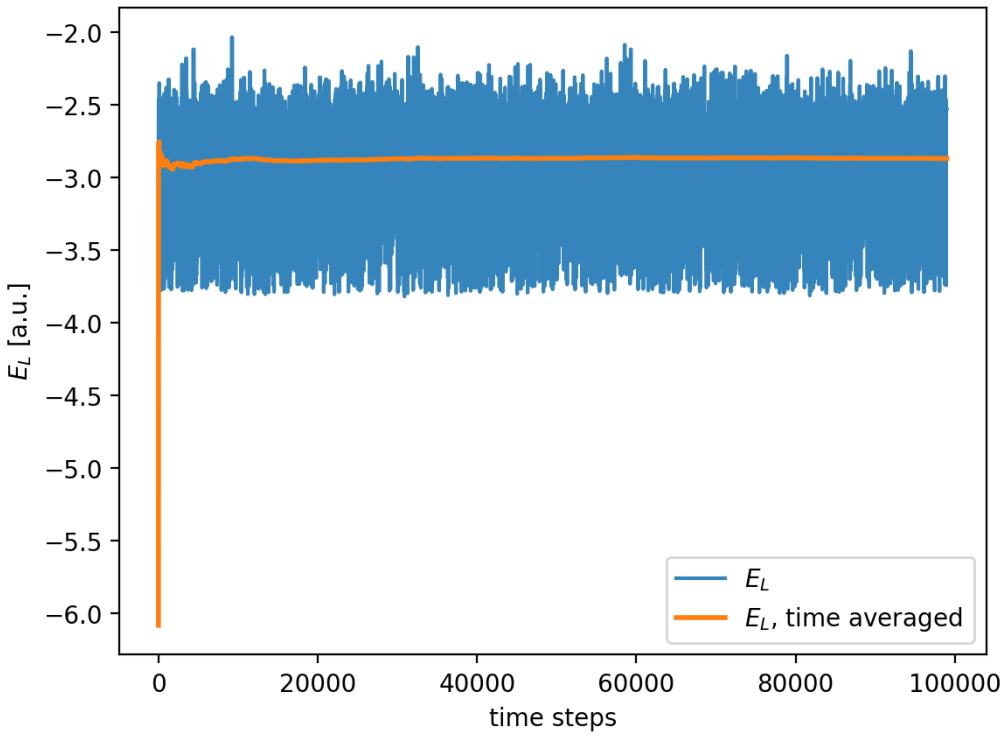

Figure 7: Energy E_L and time averaged energy plotted over 100000 time steps. The time averaged energy has a mean value of -2.868892 a.u. after the initial phase.

A Source Code

A.1 The main markov chain monte carlo method :

```
result_mcmc variational_mcmc(double r1[3], double r2[3], int n, int n_eq, double alpha,
    double delta, bool adjust_alpha, double a, double beta, bool print,
    bool write_file, int time_lag)
{
    gsl_rng* k = get_rand();
    int accepted = 0;
    double energy_accum = 0;
    double d_ln_wave_accum = 0;
    double energy_wave_accum = 0;
    double* energies = (double*) malloc(sizeof(double) * (n - n_eq));
    double temp_r1[3];
    double temp_r2[3];
    char buffer[50];
    sprintf(buffer, "data/data_n%d_neq%d_alpha_%.3f.csv", n, n_eq, alpha);
    FILE* file;
    if (write_file == true)
    {
        file = fopen(buffer, "w+");
    }
    for (int t = 0; t < n; t++)
    {
        // One step of the MCMC
        memcpy(temp_r1, r1, sizeof(temp_r1));
        memcpy(temp_r2, r2, sizeof(temp_r2));
        result_t result = variational_mcmc_one_step(temp_r1, temp_r2, delta, k, alpha);
        if (result.accepted == true)
        {
            accepted++;
            memcpy(r1, temp_r1, sizeof(temp_r1));
            memcpy(r2, temp_r2, sizeof(temp_r2));
        }
        // Continuing to the next timestep before calculating quantities and
        // writing to file if we are in the equilibration phase.
        if (t >= n_eq)
        {
            double energy = get_energy(r1, r2, alpha);
            double d_ln_wave = d_wave(r1, r2, alpha);
            energies[t - n_eq] = energy;// / (t - n_eq);
            energy_accum += energy;
            d_ln_wave_accum += d_ln_wave;
            energy_wave_accum += energy * d_ln_wave;
            if (write_file == true)
            {
                fprintf(file, "%f,%f,%f,%f,%f,%f,%f,%f\n", r1[0], r1[1], r1[2], r2[0],
                        r2[1], r2[2], alpha, energy);
            }
            // Adjusting alpha using gradient descent.
            if(adjust_alpha == true && t % 20 == 0)
            {
                int p = t - n_eq + 1;
                double step = a * pow(p, - beta);
                double d_alpha = 2 * ((energy_wave_accum / p) - (energy_accum / p) * (d_ln_wave_accum / p));
                alpha -= step * d_alpha;
            }
        }
    }
    double block_avg = block_average(energies, n - n_eq, time_lag);
    double autocor = autocorrelation(energies, n - n_eq, time_lag);
    if (print == true)
    {
        printf("Fraction accepted: %.5f\nAutocorrelation: %.5f\nBlock average: %.5f\nAlpha: %.5f\n",
            (float) accepted / n, autocor, block_avg, alpha);
    }
    result_mcmc result;
    result.alpha = alpha;
    result.autocorrelation = autocor;
    result.block_average = block_avg;
    result.avg_energy = energy_accum / (n - n_eq);
    if (write_file == true)
    {
        fclose(file);
    }
    return result;
}
```

A.2 The main method for task 1 :

```
void task1(void)
{
    double r1[] = {0.5, 0, 0};
    double r2[] = {0, -0.5, 0};
    double alpha = 0.1;
    double delta = 2;
    int n = 1000000;
    int n_eq = 0;
    result_mcmc result = variational_mcmc(r1, r2, n, n_eq,
        alpha, delta, false, 1, 0.9, true, true, 1000);
}
```

A.3 The main method for task 2 :

```
void task2(void)
{
    double r1[] = {75, 0, 0};
    double r2[] = {0, -75, 0};
    double alpha = 0.1;
    double delta = 2;
    int n = 5000;
    int n_eq = 0;
    result_mcmc result = variational_mcmc(r1, r2, n, n_eq, alpha,
        delta, false, 1, 0.9, true, true, 1000);

    n = 10000;
    n_eq = 1000;
    int num_block_sizes = 2000;
    int n_runs = 10;

    FILE* file = fopen("data/task2b.csv", "w+");

    for(int i = 1; i <= num_block_sizes; ++i){

        double autocor_accum = 0;
        double block_avg_accum = 0;
        for (int j = 0; j < n_runs; j++)
        {
            result_mcmc result = variational_mcmc(r1, r2, n, n_eq,
                alpha, delta, false, 1, 0.9, false, false, i);
            autocor_accum += result.autocorrelation;
            block_avg_accum += result.block_average;
        }
        fprintf(file, "%f,%f,%d\n",
            autocor_accum / n_runs, block_avg_accum / n_runs, i);
    }
    fclose(file);
}
```

A.4 The main method for task 3 :

```
void task3(void)
{
    double r1[] = {1, 0, 0};
    double r2[] = {0, 1, 0};
    double alpha0 = 0.05;
    double increment = 0.01;
    int n_alphas = 21;
    double alphas[n_alphas];
    double delta = 2;
    int n = 100000;
    int n_eq = 1000;
    int n_runs = 200;
    FILE* file = fopen("data/task3.csv", "w+");

    // Generating alpha values.
    for (int i = 0; i < n_alphas; i++)
    {
        alphas[i] = alpha0;
        alpha0 += increment;
    }
    // Calculating the statistical inefficiency for each alpha value.
    for (int i = 0; i < n_alphas; i++)
    {
        double autocorrelation_accum = 0;
        double block_avg_accum = 0;
        double energy_accum = 0;
        for (int j = 0; j < n_runs; j++)
        {
            result_mcmc result = variational_mcmc(r1, r2, n, n_eq,
                alphas[i], delta, false, 1, 1, false, false, 1000);
            autocorrelation_accum += result.autocorrelation;
            block_avg_accum += result.block_average;
            energy_accum += result.avg_energy;
        }
        autocorrelation_accum /= n_runs;
        block_avg_accum /= n_runs;
        energy_accum /= n_runs;
        printf("alpha = %.4f, avg energy = %.4f, autocor = %.4f,
            block avg = %.4f\n", alphas[i], energy_accum,
            autocorrelation_accum, block_avg_accum);
        fprintf(file, "%f, %f, %f, %f\n", alphas[i], energy_accum,
            autocorrelation_accum, block_avg_accum);
    }
    fclose(file);
}
```

A.5 The main method for task 4 :

```
void task4(void)
{
    double r1[] = {0.5, 0, 0};
    double r2[] = {0, -0.5, 0};
    double alpha = 0.1;
    double delta = 2;
    int n = 200000;
    int n_eq = 500;
    int n_runs = 200;
    double betas[] = {0.6, 0.7, 0.8, 0.9, 1};
    for (int i = 0; i < sizeof(betas) / sizeof(double); i++)
    {
        double alpha_converge = 0;
        for (int j = 0; j < n_runs; j++)
        {
            result_mcmc result = variational_mcmc(r1, r2, n, n_eq, alpha, delta,
                                                  true, 1, betas[i], false, false, 1000);
            alpha_converge += result.alpha;
        }
        printf("beta = %.4f: alpha = %.4f\n", betas[i],
            alpha_converge / n_runs);
    }
}
```

A.6 The main method for task 5 :

```
void task5(void)
{
    double r1[] = {0.5, 0, 0};
    double r2[] = {0, -0.5, 0};
    double alpha = 0.15;
    double delta = 2;
    int n = 100000;
    int n_eq = 1000;
    result_mcmc result = variational_mcmc(r1, r2, n, n_eq,
        alpha, delta, false, 1, 0.9, true, true, 1000);
}
```

A.7 One step of the MCMC algorithm:

```
result_t variational_mcmc_one_step(double* r1, double* r2, double delta, gsl_rng* k, double alpha)
{
    result_t result;
    result.accepted = 0;
    result.wave = wave(r1, r2, alpha);
    displace_electron(r1, delta, k);
    displace_electron(r2, delta, k);
    double w2 = wave(r1, r2, alpha);
    double r = gsl_rng_uniform(k);
    if (r < (w2 * w2) / (result.wave * result.wave))
    {
        result.wave = w2;
        result.accepted = 1;
    }
    return result;
}
```

A.8 Autocorrelation and block average:

```
double autocorrelation(double *data, int data_len, int time_lag_ind)
{
    double mean = average(data, data_len);
    double var = variance(data, data_len);
    double cov = 0;
    for (int idx = 0; idx < data_len - time_lag_ind; idx++)
    {
        // Covariance
        cov += (data[idx] - mean) * (data[idx + time_lag_ind] - mean);
    }
    // Covariance / variance = correlation
    return cov / (var * (data_len - time_lag_ind));
}

double block_average(double *data, int data_len, int block_size)
{
    int n_blocks = data_len / block_size;
    double* blocks = (double*) malloc(sizeof(double) * n_blocks);
    memset(blocks, 0, n_blocks * sizeof(double));
    for (int jdx = 0; jdx < n_blocks; jdx++)
    {
        for (int idx = 0; idx < block_size; idx++)
        {
            blocks[jdx] += data[idx + jdx * block_size];
        }
        blocks[jdx] /= block_size;
    }
    double var = variance(data, data_len);
    double block_var = variance(blocks, n_blocks);
    free(blocks);
    return block_size * block_var / var;
}
```

A.9 Misc. methods used in MCMC:

```
void displace_electron(double* r, double delta, gsl_rng* k)
{
    for (size_t i = 0; i < 3; i++)
    {
        r[i] += (gsl_rng_uniform(k) - 0.5) * delta;
    }
}

double wave(double* r1, double* r2, double alpha)
{
    double r1_len = vector_norm(r1, 3);
    double r2_len = vector_norm(r2, 3);
    double r12_len = distance_between_vectors(r1, r2, 3);
    return exp(- 2 * r1_len) * exp(- 2 * r2_len) *
        exp(r12_len / (2 + 2 * alpha * r12_len));
}

double d_wave(double* r1, double* r2, double alpha)
{
    double r12_len = distance_between_vectors(r1, r2, 3);
    return - 2 * (r12_len * r12_len) / pow(2 * r12_len * alpha + 2, 2);
}

double get_energy(double* r1, double* r2, double alpha)
{
    double r12_len = distance_between_vectors(r1, r2, 3);
    double r_norm_diff[] = {0, 0, 0};
    double r_diff[] = {0, 0, 0};
    double r1_norm[3];
    double r2_norm[3];
    memcpy(r1_norm, r1, sizeof(r1_norm));
    memcpy(r2_norm, r2, sizeof(r2_norm));
    double denominator = 1 + alpha * r12_len;
    normalize_vector(r1_norm, 3);
    normalize_vector(r2_norm, 3);
    elementwise_subtraction(r_norm_diff, r1_norm, r2_norm, 3);
    elementwise_subtraction(r_diff, r1, r2, 3);
    return - 4 + (dot_product(r_norm_diff, r_diff, 3)) /
        (r12_len * pow(denominator, 2)) - 1 /
        (r12_len * pow(denominator, 3)) - 1 /
        (4 * pow(denominator, 4)) + 1 / r12_len;
}

gsl_rng* get_rand(void){
    const gsl_rng_type* T;
    gsl_rng* r;
    gsl_rng_env_setup();
    T = gsl_rng_default;
    r = gsl_rng_alloc(T);
    time_t seed = time(NULL);
    gsl_rng_set(r, seed);
    return r;
}
```

A.10 Structs used in codebase:

```
typedef struct
{
    bool accepted;
    double wave;
} result_t;

typedef struct
{
    double autocorrelation;
    double block_average;
    double alpha;
    double avg_energy;
} result_mcmc;
```