Homework 1: Optimizations

Steven Stetzler

April 29, 2018

Pledge: On my honor as a student, I have neither given nor received unauthorized aid on this assignment.

1 Statement of Completion

I was successfully able to optimize the code provided in several ways. I used both by-hand optimizations and the O1, O2, and O3 compilation flags to optimize the provided code. I also made an attempt to change the algorithmic time complexity of this problem. I found that there were several by-hand optimizations that improved the run time of the code by almost a factor of 2 in some cases. However, I found that once one of the optimization flags is included (either O1, O2, or O3), the gap between the run time of the original and hand-optimized code closed quickly. Some by-hand optimizations make optimization with the O1, O2, and O3 flags harder and hindered performance when these flags were included. I also made an attempt to change the algorithmic time complexity of the code by first placing each point into a k-d tree data structure. However, in most cases, the memory overhead of this data structure caused performance to be significantly worse than the original implementation. In the case that the cutoff value C is reduced to below $C \sim 0.1$, k-d trees are found to provide a superior run time than any of the optimizations introduced by hand or when any of the O1, O2, or O3 compilation flags is provided.

2 Problem Description

We attempted to optimize a simple code that computes a specific made up function on pairs of points contained in a 3D grid of size $1 \times 1 \times 1$. Each point is specified by its coordinates in the x direction, the y direction, and the z direction and has an associated charge q. The made up function applied to the data was

$$E = \sum_{i<j} \frac{e^{r_{i,j} q_i} e^{r_{i,j} q_j}}{r_{i,j}} - \frac{1}{a} \quad \text{if } r_{i,j} \leq C \tag{1}$$

where $r_{i,j}$ is the euclidean distance computed as

$$r_{i,j} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2}$$

and C is a cutoff in distance that is chosen arbitrarily and $a = 3.2$. Within the confines of this problem, we require that $C \in (0, 1)$ in order for E to be non-zero. Since this function works on pairs of points, then each pair of points must be compared, causing a naive implementation of this function to have a time complexity of $\mathcal{O}(N^2)$. This function was originally implemented using the following C code snippet:

```
total_e = 0.0;
cut_count = 0;
for (i=1; i<=natom; ++i)
{
    for (j=1; j<=natom; ++j)
    {
        if ( j < i )
        {
```

```
            vec2 = pow((coords[0][i-1]-coords[0][j-1]),2.0)
                  + pow((coords[1][i-1]-coords[1][j-1]),2.0)
                  + pow((coords[2][i-1]-coords[2][j-1]),2.0);
            rij = sqrt(vec2);
            if ( rij <= cut )
            {
                ++cut_count;
                current_e = (exp(rij*q[i-1])*exp(rij*q[j-1]))/rij;
                total_e = total_e + current_e - 1.0/a;
            }
        }
    }
}
```

3 Approach, Results, and Analysis

I performed a set of several by-hand and automatic optimizations to this code to try to improve it's performance. Each version of the code is run using the same set of $N = 10000$ points generated using the `generate_input.c` program with a random seed of 1. Each version of the code is run using the following set of hardware on the `nibbler4` node of the University of Virginia Computer Science SLURM Cluster:

Processor: Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz

Cache: 32 kB L1d and L1i cache, 256 kB L2 cache, and 25,600 kB L3 cache.

Memory: 60 GB of RAM

A description of each of the optimizations I attempted follows with a summary of the performance of the optimization for $N = 10000$ and $C = 0.5$ when no optimization flag is included during compilation and when the O1, O2, or O3 flag is included. Each result quoted is the average run time among a set of 5 calls to a function with the specific optimization implemented. In all cases, the times quoted are the wall clock time spent by the program in the above code snippet alone. Additional time is spent by the program in reading the data from an external file, which is ignored throughout our analysis. The implementation of each optimization can be found at the end of this paper.

3.1 Compilation Flags

The original code was compiled using the O1, O2, and O3 compilation flags. To see how run time could be improved using automatic optimizations. The resulting run times are found below in Table 1.

	Run Time (s)
Optimization	`original`
None	2.368
O1	1.092
O2	0.916
O3	0.879

Table 1: Run time of the original implementation in comparison when no compilation flags are included and when the O1, O2, and O3 compilation flags are included.

We find that inclusion of the O1 flag already provides tremendous improvement to the run time of this program. Inclusion of the O2 and O3 flags provides marginal improvement over what is provided by O1, but it is improvement nonetheless. The O3 optimization flag provided the greatest improvement in performance, reducing run time by 64%. This provides a clear indication that when code is compiled the minimum optimization flag that should be included is the O3 flag.

Beyond these automatic optimizations, several by-hand optimizaitons can be performed to attempt to reduce the run time of the code, as described in the following sections.

3.2 Combine Exponents

Equation 1 can be modified slightly to take advantage of the fact that multiplication of two numbers with the same base can be reduced to a single exponentiation with the exponents of the two numbers added together. In math: $a^{b_1}a^{b_2} = a^{b_1+b_2}$. Modifying Eq. 1, we have

$$E = \sum_{i<j} \frac{e^{r_{i,j}q_i + r_{i,j}q_j}}{r_{i,j}} - \frac{1}{a} \quad \text{if } r_{i,j} \leq C \tag{2}$$

In code, the change is

```
current_e = (exp(rij*q[i-1] + rij*q[j-1]))/rij;
```

as implemented in the function `combine_exp`.

This should reduce the number of calls to the `exp` function from the `math.h` library by a factor of 2 and reduce the number of multiplications per pair of points within the cutoff distance from 3 to 2. The run time improvement is summarized in Table 2.

	Run Time (s)	
Optimization	`original`	`combine_exp`
None	2.368	2.215
O1	1.092	0.860
O2	0.916	0.688
O3	0.879	0.694

Table 2: Run time of the `combine_exp` function in comparison to the original implementation.

We find that this is a useful by-hand optimization as it decreases run time as compared to the original implementation for any level of automatic optimization.

3.3 Combine Exponents and Multiplication

Equation 2 can be modified further by using the distributive property of multiplication: $ab + ac = a(b + c)$. Modifying Eq. 2, we have

$$E = \sum_{i<j} \frac{e^{r_{i,j}(q_i+q_j)}}{r_{i,j}} - \frac{1}{a} \quad \text{if } r_{i,j} \leq C \tag{3}$$

In code, the change is

```
current_e = (exp(rij*(q[i-1] + q[j-1])))/rij;
```

as implemented in the function `combine_exp_combine_mult`.

For each pair of points within the cutoff distance, this should reduce the number of multiplications per pair of points from 2 to 1. The run time improvement is summarized in Table 3.

This optimization provides marginal improvement of the run time in comparison to the `combine_exp` optimization, indicating that removal of expensive function calls is more important than removal of base operations like multiplications.

3.4 Reducing `sqrt` Calls

We can also try to reduce the number of calls to the `sqrt` function of the `math.h` library by taking advantage of the fact that if two distances are equal, then the square of their distance is equal as well, so if we

	Run Time (s)	
Optimization	`original`	`combine_exp_combine_mult`
None	2.368	2.203
O1	1.092	0.853
O2	0.916	0.687
O3	0.879	0.693

Table 3: Run time of the `combine_exp_combine_mult` function in comparison to the original implementation.

are making the comparison between $\sqrt{x^2+y^2+z^2}$ and C, then an equivalent comparison can be made between $x^2+y^2+z^2$ and C^2. This changes the code in the following way, as implemented in the function `necessary_rij`:

```
cut_squared = pow(cut, 2);
for (i=1; i<=natom; ++i)
{
    for (j=1; j<=natom; ++j)
    {
        if ( j < i )
        {
            vec2 = pow((coords[0][i-1]-coords[0][j-1]),2.0)
                 + pow((coords[1][i-1]-coords[1][j-1]),2.0)
                 + pow((coords[2][i-1]-coords[2][j-1]),2.0);
            if ( vec2 <= cut_squared )
            {
                rij = sqrt(vec2);
                ++cut_count;
                current_e = (exp(rij*q[i-1])*exp(rij*q[j-1]))/rij;
                total_e = total_e + current_e - 1.0/a;
            }
        }
    }
}
```

By performing one exponentiation (computation of C^2), we can avoid unnecessary calls to the `sqrt` function for all pairs of points outside of the cut off distance. Note that while this should work in theory, performing this optimization actually changed the result of the program by changing the total E computed on the same set of data. This is likely because of limitations in the precision of floating point numbers. These limitations mean we cannot be guaranteed that $\sqrt{x^2}$ is that same as x where x is a floating point number. In this case of `necessary_rij`, there were 15 pairs of points for whom $r_{i,j}^2 > C^2$ but $r_{i,j} < C$. The performance of this optimization is shown in Table 4.

	Run Time (s)	
Optimization	`original`	`necessary_rij`
None	2.368	2.284
O1	1.092	0.858
O2	0.916	0.805
O3	0.879	0.806

Table 4: Run time of the `necessary_rij` function in comparison to the original implementation.

We find that this optimization provides a good amount of run time improvement at all levels of automatic optimization; however, since the end result computed (E) was different, this optimization cannot be guaranteed to be safe, and thus must be discarded from consideration.

3.5 Unnecessary Comparisons

We can also improve the code by removing the explicit check for $i < j$, as required in the summation in Eq. 1. Since this summation over pairs is implemented as two for loops, with i the index of the outer for loop and j the index of the inner for loop, then we can simply start the inner for loop at the index i instead. In code this becomes:

```
for (i=1; i<=natom; ++i)
{
    for (j=i; j<=natom; ++j)
    {
        ...
    }
}
```

which is implemented in the `no_if` function. This should remove N^2 unnecessary comparisons between i and j in our code. The effect of this optimization is shown in Table 5.

	Run Time (s)	
Optimization	`original`	`no_if`
None	2.368	2.269
O1	1.092	1.039
O2	0.916	0.886
O3	0.879	0.889

Table 5: Run time of the `no_if` function in comparison to the original implementation.

Again, we find some improvement in the run time of the program when this optimization is included.

3.6 Unnecessary Additions

In the original implementation, each index i and j is initialized to 1, ignoring the fact that computer scientists start counting at 0, since in array indexing the first element is at 0 offset from the beginning of the array. This has the effect that in the original implementation, many unnecessary additions are performed to correctly shift i and j to their correct values before indexing the data array. These additions can be removed by modifying the code to be

```
for (i=0; i<natom; ++i)
{
    for (j=0; j<natom; ++j)
    {
        if ( j < i )
        {
            vec2 = pow((coords[0][i]-coords[0][j]),2.0)
                 + pow((coords[1][i]-coords[1][j]),2.0)
                 + pow((coords[2][i]-coords[2][j]),2.0);
            rij = sqrt(vec2);
            if ( rij <= cut )
            {
                ++cut_count;
                current_e = (exp(rij*q[i])*exp(rij*q[j]))/rij;
                total_e = total_e + current_e - 1.0/a;
            }
        }
    }
}
```

which is implemented in the function `no_add_index`. This has the effect of removing 8 additions per pair of particles, which is in total $\sim 8N^2$ additions. A summary of the performance of this optimization is shown in Table 6.

	Run Time (s)	
Optimization	`original`	`no_add_index`
None	2.368	2.347
O1	1.092	1.073
O2	0.916	0.900
O3	0.879	0.906

Table 6: Run time of the `no_add_index` function in comparison to the original implementation.

This optimization had almost no effect on the run time of the program, which is to be expected as additions on modern processors can be started in parallel and cost very few processor cycles, meaning removing additions should have almost no effect on performance, as we saw.

3.7 Unnecessary Division

We can also modify Eq. 1 further to avoid unnecessary divisions for each pair of points. Notice that the constant being subtracted from the energy $1/a$ can be moved outside of the sum:

$$E = (\sum_{i<j} \frac{e^{r_{i,j}q_i} e^{r_{i,j}q_j}}{r_{i,j}}) - \frac{N_{pair}}{a} \quad \text{if } r_{i,j} \leq C \tag{4}$$

where N_{pair} is the number of pairs of points that lie within the cutoff distance of one another. In code, this is implemented in the function `pull_out_a` as

```
for (i=1; i<=natom; ++i)
{
    for (j=1; j<=natom; ++j)
    {
        if ( j < i )
        {
            ...
            if ( rij <= cut )
            {
                ++cut_count;
                current_e = (exp(rij*q[i-1])*exp(rij*q[j-1]))/rij;
                total_e = total_e + current_e;
            }
        }
    }
}
total_e = total_e - cut_count / a;
```

This should remove many unnecessary divisions from the program, one for each of the pairs of points that lies within the cutoff distance of one another. The improvement that this optimization provides is shown in Table 7.

Strangely, performance seems to have decreased marginally with the inclusion of this optimization. It is unclear why this optimization has no or negative impact on performance, however, one reason might be that the division of $1/a$ is easier to compute than the division of cut_count$/a$, however this seems unlikely as division complexity should seemingly scale with the bit width of the numbers involved, which should be the same between the two cases.

	Run Time (s)	
Optimization	`original`	`pull_out_a`
None	2.368	2.390
O1	1.092	1.074
O2	0.916	0.903
O3	0.879	0.910

Table 7: Run time of the `pull_out_a` function in comparison to the original implementation.

3.8 Distance Checking

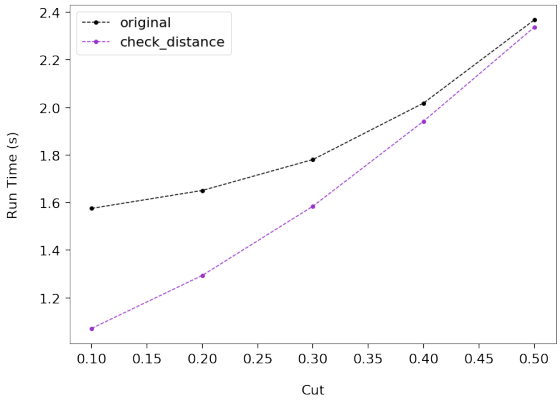
We can try to reduce the number of calls to the `pow` function of the `math.h` library by performing early checks of whether the distance of two points is within the distance cut off specified. Since our distance is computed as $r_{i,j} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i + z_j)^2}$ then if any of $x_i - x_j$, $y_i - y_j$, or $z_i - z_j$ are greater than the cutoff value, then we know that $r_{i,J}$ will be automatically greater than the cut off. This allows us to reduce the number of times that `pow` is called by exiting out of the for loop before `pow` is ever called. In code, this is implemented in the `check_distance` function as

```
for (i=1; i<=natom; ++i)
{
    for (j=1; j<=natom; ++j)
    {
        if ( j < i )
        {
            x = coords[0][i-1] - coords[0][j-1];
            if (x > cut) {
                continue;
            }
            y = coords[1][i-1] - coords[1][j-1];
            if (y > cut) {
                continue;
            }
            z = coords[2][i-1] - coords[2][j-1];
            if (z > cut) {
                continue;
            }
            vec2 = pow(x, 2.0) + pow(y, 2.0) + pow(z, 2.0);
            ...
        }
    }
}
```

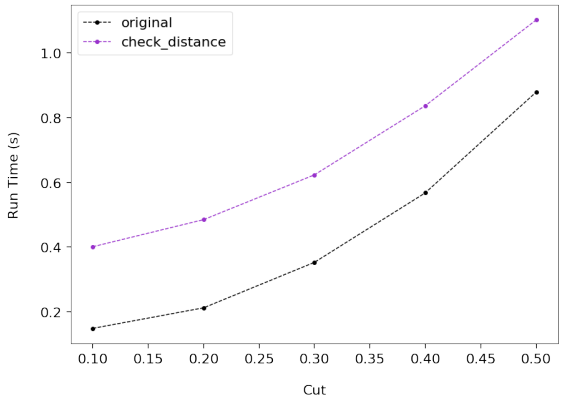
	Run Time (s)	
Optimization	`original`	`check_distance`
None	2.368	2.337
O1	1.092	1.173
O2	0.916	1.104
O3	0.879	1.102

Table 8: Run time of the `check_distance` function in comparison to the original implementation.

The performance of this optimization is shown in Table 8. This optimization provided a small increase in performance when no optimization flag is provided, and actually decreased performance when an optimization flag is provided. Perhaps this is due to the fact that this modification to the code introduces unnecessary branching in the code that is hard to resolve, slowing overall processing. We can check to see how the cutoff



(a) Run time without any automatic optimizations.



(b) Run time when the O3 flag is included.

Figure 1: Run time of the original implementation and the `check_distance` optimization as a function of the cutoff distance C.

distance C affects performance as well since wen we are checking the distance in advance of computing the true distance $r_{i,j}$, we will receive the most benefit when most particles are not within the cutoff distance of one another (when C is small). This is reflected in the scaling of the run time with the cutoff distances, as shown in Fig. 1a.

Interestingly, when optimization flags are included, the performance gain that `check_distance` achieves at low values of C over the original implementation is negligible or actually non-existent when the O1 or O2 flags are included. When the O3 flag is included, this optimization is strictly worse over all cut scales, as seen in Fig. 1b, which shows the run time scaling with C when the O3 flag is included during compilation. In this case, the `check_distance` optimization achieves worse performance than the original implementation and has similar scaling with the cutoff distance C as the original implementation. This implies that perhaps the inclusion of the optimization flags is performing a similar approach of early termination of the computation of Eq. 1, but with more optimizations on top of that one.

3.9 Replace `pow`

I altered the code to replace the use of `pow` from the `math.h` library with a hard coded multiplication. Since we are always raising numbers to the second power, we can replace a^2 with $a * a$. In code this was implemented in the function `no_pow` as

```
for (i=1; i<=natom; ++i)
{
    for (j=1; j<=natom; ++j)
    {
        if ( j < i )
        {
            x = coords[0][i-1]-coords[0][j-1];
            y = coords[1][i-1]-coords[1][j-1];
            z = coords[2][i-1]-coords[2][j-1];

            rij = sqrt(x*x + y*y + z*z);
            ...
        }
    }
}
```

Among all by-hand optimizations, this one provided the greatest performance boost, as shown in Table 9.

	Run Time (s)	
Optimization	`original`	`no_pow`
None	2.368	1.652
O1	1.092	1.077
O2	0.916	0.909
O3	0.879	0.905

Table 9: Run time of the `no_pow` function in comparison to the original implementation.

3.10 All Optimizations

I combined all of the useful optimizations into a single function called `all_opts` to analyze how multiple optimizations can improve performance when combined. I excluded from this function the optimizations `check_distance` and `necessary_rij`, as `check_distance` decreased performance when optimization flags were included and `necessary_rij` changed the result of the computation. I found that performance was improved pretty significantly, as shown in Table 10

	Run Time (s)	
Optimization	`original`	`all_opts`
None	2.368	1.147
O1	1.092	0.809
O2	0.916	0.664
O3	0.879	0.667

Table 10: Run time of the `all_opts` function in comparison to the original implementation.

Even with the inclusion of optimization flags O1, O2, and O3, all of the optimizations together were able to achieve better performance than the original implementation.

3.11 All Optimization Including `check_distance`

I also combined all of the by-hand optimizations including `check_distance` into a single function called `all_opts_and_check_dist` in order to see how the inclusion of a previously ineffective optimization might affect performance. The results are shown in Table 11.

	Run Time (s)	
Optimization	`original`	`all_opts_and_check_dist`
None	2.368	1.318
O1	1.092	0.902
O2	0.916	0.821
O3	0.879	0.823

Table 11: Run time of the `all_opts_and_check_dist` function in comparison to the original implementation.

The results are quite surprising in that the inclusion of the distance checking optimization caused performance to be significantly worse than excluding this optimization. Since we found that `check_distance` improved performance at low values of C, we can observe how this performance changes for `all_opts` and `all_opts_and_check_dist` when no optimization flags are included, which is shown in Fig. 2. Interestingly, we find that the inclusion of `check_distance` causes performance to be worse across all values of C in comparison to all of the other optimizations combined.

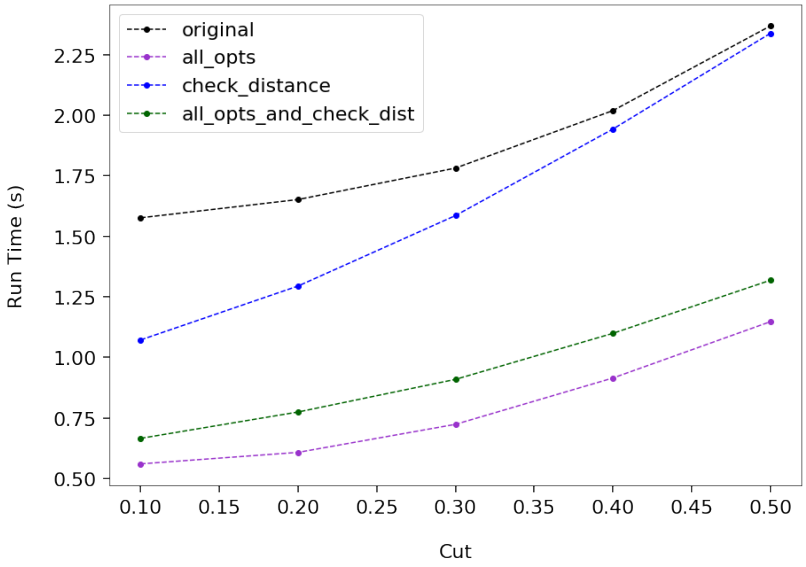


Figure 2: Run time of the original implementation, the `check_distance` optimization, `all_opts`, and `all_opts_and_check_dist` as a function of the cutoff distance C when no optimization flag is included.

3.12 K-D Trees

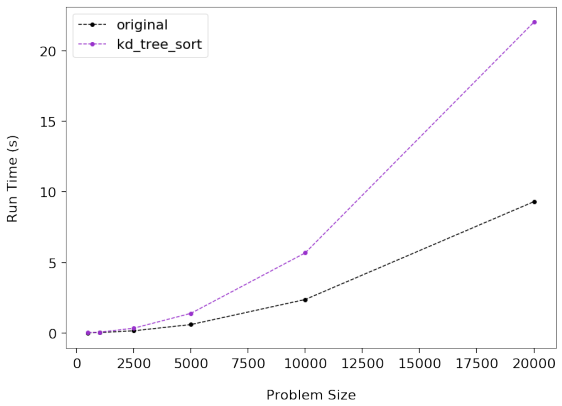
I attempted to change the algorithmic time complexity of finding pairs of points within the cutoff distance C but using a k nearest neighbor search on a k-d tree data structure that contains the data points. The code used to construct and search the k-d tree was provided by the kdtree software package, a C implementation of k-d trees. The optimization is implemented in the `kd_tree_sort` function. Search in a k-d tree containing N points is at best $\mathcal{O}(\log N)$ and at worst $\mathcal{O}(N)$. Considering that we are calling a search for every point within the data set, the search run time then scales as $\mathcal{O}(N \log N)$ in the best case and as $\mathcal{O}(N^2)$ in the worst case. The search time should also be proportional to k, the number of nearest neighbors, giving an overall time complexity of $\mathcal{O}(kN \log N)$ in the best case and as $\mathcal{O}(kN^2)$ in the worst case. So at worst, we would achieve a similar algorithmic time complexity as the original implementation ($\mathcal{O}(N^2)$) and at best, we would improve upon that. However, it is important to note that if $k \sim N$, then all benefits of using the k-d tree is lost, as the algorithmic time complexity becomes $\mathcal{O}(N^2 \log N)$ at best and $\mathcal{O}(N^3)$ at worst. This will happen if the cutoff distance C is large, in which case for any give point, most of the other points in the data set will be within the cutoff distance of it. For example, for $C = 0.5$, the k nearest neighbors of a give point will be distributed in a sphere of volume $4/3\pi(1/2)^3 \approx 0.52$. When we consider that the N points are distributed uniformly in a space of volume $1 \times 1 \times 1 = 1$, then we expect for any given point that it will have $k = 0.52N$ nearest neighbors. In this case, we expect that the k-d tree optimization will actually make the run time worse, since $k \sim N$. However, for a smaller cutoff distance, e.g. $C = 0.1$, following the same logic tells us that we should expect that $k = 0.004N$, which is significantly smaller than N. In this case, we can expect to achieve an improvement in run time when using the k-d tree optimization.

Listed in Table 12 is the run time achieved by the k-d tree optimization for $N = 100000$ and $C = 0.5$. We see from this a confirmation of our expectation, that the k-d tree optimization ruined the run time of our program. In Fig. 3a, we show how the run time scales with the number of points N for $C = 0.5$. Again, we see that the scaling of the k-d tree optimization is much worse than that of the original implementation, consistent with the $\mathcal{O}(N^2 \log N)$ or $\mathcal{O}(N^3)$ time complexity. In Fig. 3b, we show how run time scales when a smaller value for C is chosen. Here, we choose $C = 0.1$. This time however, we recover a better scaling of the run time with the size of the problem, one more consistent with $\mathcal{O}(N \log N)$ or $\mathcal{O}(N^2)$.

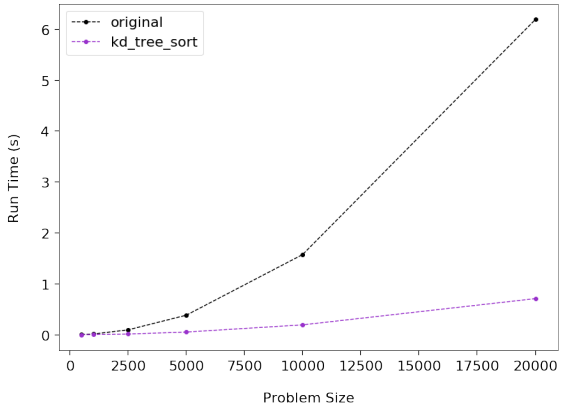
To take a closer look at how the cutoff distance C affects run time, we plot in Fig. 4a how the run time scales with the cutoff distance C for $N = 10000$. We observe that at small values of C, run time is dramatically improved using the k-d tree optimization. It is important to note however that we may have never actually improved the algorithmic complexity of the problem, just significantly reduced the pre-factor

	Run Time (s)	
Optimization	`original`	`kd_tree_sort`
None	2.368	5.674
O1	1.092	4.187
O2	0.916	4.164
O3	0.879	4.100

Table 12: Run time of the `kd_tree_sort` function in comparison to the original implementation.



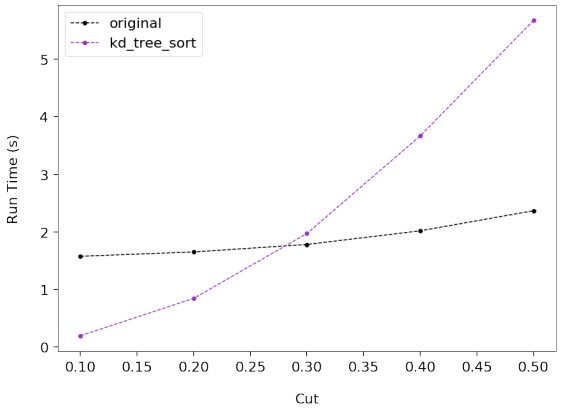
(a) Run time as a function of N for $C = 0.5$.



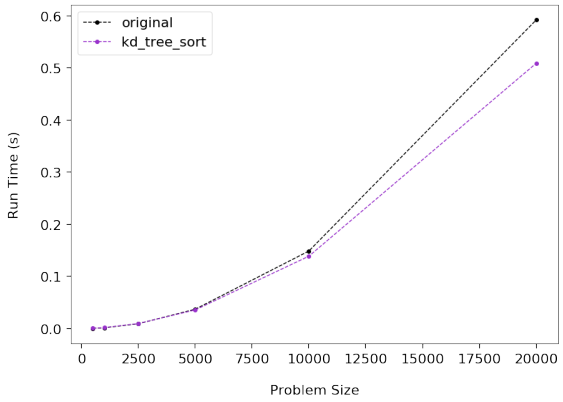
(b) Run time as a function of N for $C = 0.1$.

Figure 3: Run time of the original implementation and the `kd_tree_sort` implementation as a function of the problem size N when no optimization flag is included.

associated with the scaling of the problem since $k \propto N$ and $k \propto C^3$. By reducing C by a factor of 5, we have reduced the scaling of k to $k \sim N/125$ and thus the run time scaling to $\mathcal{O}(\frac{1}{125}N^2 \log N$ or $\mathcal{O}(\frac{1}{125}N^3$. If we introduce optimizations into our original implementation that can provide a similar reduction in the proportionality constant in front of the original code's time complexity of $\mathcal{O}(N^2)$, then we will see that k-d tree hasn't provided the promised time complexity after all. To do this, we included the O3 optimization flag during compilation of both the k-d tree optimization and the original code. The recovered scaling of the run time with N for $C = 0.1$ is shown in Fig. 4b. Here, we see that inclusion of the O3 flag reduces run time significantly for the original implementation, and by a factor similar to what the k-d optimization provides. However, the k-d tree optimization still provides a slightly better scaling over problem size.



(a) Run time as a function of the cutoff distances C for $N = 10000$.



(b) Run time as a function of the problem size N for $C = 0.1$ with the O3 flag.

Figure 4: Run time of the original implementation and the `kd_tree_sort` implementation as a function of the cutoff size C for $N = 10000$ (left) and the problem size N at $C = 0.5$ when the O3 optimization flag is included during compilation (right).

4 Conclusions

We applied several by-hand optimizations to our code and compiled the code with each of the optimization flags (O1, O2, and O3) in order to see how the run-time of this code can be improved. The average run times for all optimizations at all levels of compilation level optimization (O1, O2, and O3 flags) is shown in Fig. 5 for a problem size of $N = 10000$ and $C = 0.5$.

By analyzing Fig. 5, it can be seen that most by-hand optimizations have a small effect on the overall run time of the program; however, when all optimizations are included, there is a significant effect on the run time. One interesting observation is that the O1, O2, and O3 flags have different levels of effect on each version of the code, indicated by the difference in the heights of the red bars. The version of the code that achieved the best performance was the code that included all optimizations excluding `check_distance` with the O3 compilation flag.

In summary, this analysis shows the power of performing automatic optimization of code through the use of the O1, O2, and O3 compilation flags and through by-hand common sense optimizations that reduce the total number of operations that must be done in a program. Optimizations that had the largest impact on program performance were those that removed function calls to external libraries (removing calls to `pow` and `exp`). Optimizations that reduced the number of basic operations like addition, multiplication, and division had little effect on the overall performance of the the code. It is also important to note that when optimizing code, branching should be avoided as this makes it harder for the compiler to optimize further, as demonstrated by the increased run time of the `check_distance` optimization in comparison to the original code when any of the compilation flags is provided. An attempt was made to change the algorithmic time complexity of this problem through the use of k-d trees. This attempt seemed to be successful only when the value of the cutoff distance C was small ($C \sim 0.1$).

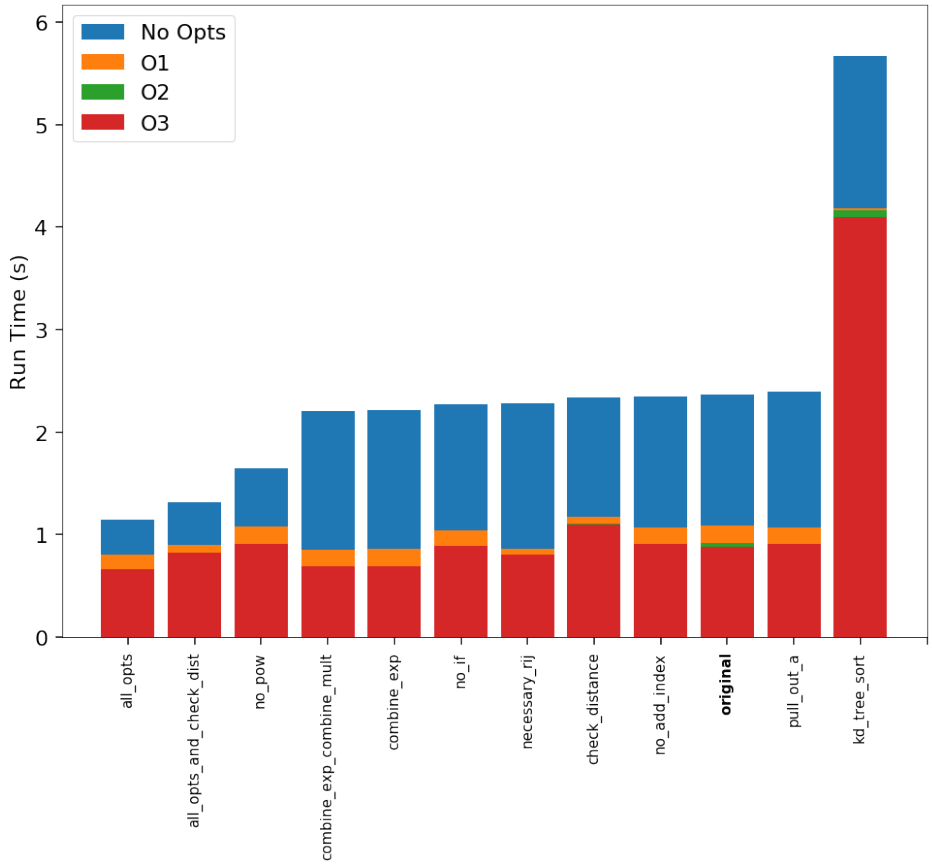


Figure 5: The run time for each of the by-hand optimizations and the original implementation (bolded) when no flags, the O1 flag, the O2 flag, and the O3 flag is included during compilation.

Here, I include the code I used to test each of the optimizations described in this paper.
Optimized Code: `opt.c` This file computes the average run time of several different functions. Each function contains the main block of code meant to be optimized with only one optimization applied to it. Testing in this way allows for optimizations to be separated from one another. The function `original_run` runs the program as originally provided.

```
/*************************************************************************
! Bad coding example 1
! !
! Shamefully written by Ross Walker (SDSC, 2006)
!
! This code reads a series of coordinates and charges from the file
! specified as argument $1 on the command line.
!
! This file should have the format:
!  I9
! 4F10.4  (repeated I9 times representing x,y,z,q)
!
! It then calculates the following fictional function:
!
!           exp(rij*qi)*exp(rij*qj) 1
!    E = Sum( ---------------------- - - ) (rij <= cut)
!        j<i          r(ij)          a
!
```

```
! where cut is a cut off value specified on the command line ($2),
! r(ij) is a function of the coordinates read in for each atom and
! a is a constant.
!
! The code prints out the number of atoms, the cut off, total number of
! atom pairs which were less than or equal to the distance cutoff, the
! value of E, the time take to generate the coordinates and the time
! taken to perform the calculation of E.
!
! All calculations are done in double precision.
!+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include "kdtree.h"
#include <string.h>

double **alloc_2D_double(int nrows, int ncolumns);
void double_2D_array_free(double **array);
/* Functions to test */
double original(double** coords, double* q, int natom, double cut);
double no_pow(double** coords, double* q, int natom, double cut);
double all_opts_and_check_dist(double** coords, double* q, int natom, double cut);
double kd_tree_sort(double** coords, double* q, int natom, double cut);
double check_distance(double** coords, double* q, int natom, double cut);
double all_opts(double** coords, double* q, int natom, double cut);
double necessary_rij(double** coords, double* q, int natom, double cut);
double pull_out_a(double** coords, double* q, int natom, double cut);
double no_add_index(double** coords, double* q, int natom, double cut);
double no_if(double** coords, double* q, int natom, double cut);
double combine_exp_combine_mult(double** coords, double* q, int natom, double cut);
double combine_exp(double** coords, double* q, int natom, double cut);
/* Testing function */
void test_func(double (*func_prt)(double**, double*, int, double), char* func_name, int argc,
    char* argv[]);
/* Original functionality */
void original_run(int argc, char* argv[]);

double TARGET_E;
double TARGET_CUT;

int main(int argc, char *argv[])
{
    original_run(argc, argv);
    test_func(&no_pow, "no_pow", argc, argv);
    test_func(&check_distance, "check_distance", argc, argv);
    test_func(&necessary_rij, "necessary_rij", argc, argv);
    test_func(&pull_out_a, "pull_out_a", argc, argv);
    test_func(&no_add_index, "no_add_index", argc, argv);
    test_func(&no_if, "no_if", argc, argv);
    test_func(&combine_exp_combine_mult, "combine_exp_combine_mult", argc, argv);
    test_func(&combine_exp, "combine_exp", argc, argv);
    test_func(&all_opts, "all_opts", argc, argv);
    test_func(&all_opts_and_check_dist, "all_opts_and_check_dist", argc, argv);
    test_func(&kd_tree_sort, "kd_tree_sort", argc, argv);
    exit(0);
```

```
}

double all_opts_and_check_dist(double** coords, double* q, int natom, double cut)
{
    clock_t time1, time2;

    double total_e, current_e, vec2, rij;
    double a;
    int cut_count;
    int i, j;
    double x, y, z;

    a = 3.2;

    time1 = clock();

    total_e = 0.0;
    cut_count = 0;

    for (i = 0; i < natom; ++i)
    {
        for (j = 0; j < i; ++j)
        {
            x = coords[0][i] - coords[0][j];
            if (x >= cut)
            {
                continue;
            }
            y = coords[1][i] - coords[1][j];
            if (y >= cut)
            {
                continue;
            }
            z = coords[2][i] - coords[2][j];
            if (z >= cut)
            {
                continue;
            }

            rij = sqrt(x*x + y*y + z*z);

            if ( rij <= cut )
            {
                ++cut_count;
                current_e = (exp(rij * (q[i] + q[j])))/rij;
                total_e = total_e + current_e;
            }
        }
    }

    total_e = total_e - cut_count / a;

    time2 = clock();

    if (fabs(total_e - TARGET_E) > 0.0001)
    {
        printf("Bad E! Got %14.10f\n", total_e);
    }
```

```
    if (cut_count != TARGET_CUT)
    {
        printf("Incorrect number of pairs! Got %d\n", cut_count);
    }

    return (double) (time2 - time1);
}

double kd_tree_sort(double** coords, double* q, int natom, double cut)
{
    clock_t time1, time2;

    double total_e, current_e, vec2, rij;
    double a;
    int cut_count;
    int i, j;

    a = 3.2;

    time1 = clock();

    void* kd = kd_create(3);
    double data[natom][4];
    double new_pt[natom][3];

    for (i = 0; i < natom; ++i)
    {
        new_pt[i][0] = coords[0][i];
        new_pt[i][1] = coords[1][i];
        new_pt[i][2] = coords[2][i];
        data[i][0] = new_pt[i][0];
        data[i][1] = new_pt[i][1];
        data[i][2] = new_pt[i][2];
        data[i][3] = (double) i;
        kd_insert(kd, new_pt[i], data[i]);
    }

    total_e = 0.0;
    cut_count = 0;

    void* result_set;
    double* near_pt_data;

    char* flags = (char*) malloc(natom * natom * sizeof(char));
    memset(flags, 0, natom * natom * sizeof(flags[0]));

    double pt[3];

    for (i = 0; i < natom; ++i)
    {
        pt[0] = coords[0][i];
        pt[1] = coords[1][i];
        pt[2] = coords[2][i];
        result_set = kd_nearest_range(kd, pt, cut);

        // Loop through, ignoring the first closest (self)
        do {
            near_pt_data = (double*) kd_res_item_data(result_set);
            j = (int) near_pt_data[3];
```

```
            if (i == j) {
                continue;
            }
            if (flags[i * natom + j] == 0) {
                flags[i * natom + j] = 1;
                flags[j * natom + i] = 1;

                vec2 = pow(pt[0] - near_pt_data[0], 2.0)
                    + pow(pt[1] - near_pt_data[1], 2.0)
                    + pow(pt[2] - near_pt_data[2], 2.0);
                rij = sqrt(vec2);

                ++cut_count;
                current_e = (exp(rij * q[i]) * exp(rij * q[j]))/rij;
                total_e = total_e + current_e - 1.0/a;
            }
        } while(kd_res_next(result_set));
    }

    time2 = clock();

    if (fabs(total_e - TARGET_E) > 0.0001)
    {
        printf("Bad E! Got %14.10f\n", total_e);
    }

    if (cut_count != TARGET_CUT)
    {
        printf("Incorrect number of pairs! Got %d\n", cut_count);
    }

    free(flags);
    kd_free(kd);

    return (double) (time2 - time1);
}

double check_distance(double** coords, double* q, int natom, double cut)
{
    clock_t time1, time2;

    double total_e, current_e, vec2, rij;
    double a;
    int cut_count;
    int i, j;
    double x;
    double y;
    double z;

    a = 3.2;

    time1 = clock();

    total_e = 0.0;
    cut_count = 0;
    for (i=1; i<=natom; ++i)
    {
        for (j=1; j<=natom; ++j)
        {
```

```
            if ( j < i )
            {
                x = coords[0][i-1] - coords[0][j-1];
                if (x > cut) {
                    continue;
                }
                y = coords[1][i-1] - coords[1][j-1];
                if (y > cut) {
                    continue;
                }
                z = coords[2][i-1] - coords[2][j-1];
                if (z > cut) {
                    continue;
                }
                vec2 = pow(x, 2.0) + pow(y, 2.0) + pow(z, 2.0);
                rij = sqrt(vec2);
                if ( rij <= cut )
                {
                    ++cut_count;
                    current_e = (exp(rij*q[i-1])*exp(rij*q[j-1]))/rij;
                    total_e = total_e + current_e - 1.0/a;
                }
            }
        }
    }

    time2 = clock();

    if (fabs(total_e - TARGET_E) > 0.0001)
    {
        printf("Bad E! Got %14.10f\n", total_e);
    }

    if (cut_count != TARGET_CUT)
    {
        printf("Incorrect number of pairs! Got %d\n", cut_count);
    }

    return (double) (time2 - time1);
}

double no_pow(double** coords, double* q, int natom, double cut)
{
    clock_t time1, time2;

    double total_e, current_e, vec2, rij, x, y, z;
    double a;
    int cut_count;
    int i, j;

    a = 3.2;

    time1 = clock();

    total_e = 0.0;
    cut_count = 0;
    for (i=1; i<=natom; ++i)
    {
        for (j=1; j<=natom; ++j)
```

```
        {
            if ( j < i )
            {
                x = coords[0][i-1] - coords[0][j-1];
                y = coords[1][i-1] - coords[1][j-1];
                z = coords[2][i-1] - coords[2][j-1];

                vec2 = x*x + y*y + z*z;
                rij = sqrt(vec2);
                if ( rij <= cut )
                {
                    ++cut_count;
                    current_e = (exp(rij*q[i-1])*exp(rij*q[j-1]))/rij;
                    total_e = total_e + current_e - 1.0/a;
                }
            }
        }
    }

    time2 = clock();

    if (fabs(total_e - TARGET_E) > 0.0001)
    {
        printf("Bad E! Got %14.10f\n", total_e);
    }

    if (cut_count != TARGET_CUT)
    {
        printf("Incorrect number of pairs! Got %d\n", cut_count);
    }

    return (double) (time2 - time1);
}


double all_opts(double** coords, double* q, int natom, double cut)
{
    clock_t time1, time2;

    double total_e, current_e, vec2, rij;
    double a;
    int cut_count;
    int i, j;
    double x, y, z;

    a = 3.2;

    time1 = clock();

    total_e = 0.0;
    cut_count = 0;

    for (i = 0; i < natom; ++i)
    {
        for (j = 0; j < i; ++j)
        {
            x = coords[0][i] - coords[0][j];
            y = coords[1][i] - coords[1][j];
            z = coords[2][i] - coords[2][j];
```

```
            rij = sqrt(x*x + y*y + z*z);

            if ( rij <= cut )
            {
                ++cut_count;
                current_e = (exp(rij * (q[i] + q[j])))/rij;
                total_e = total_e + current_e;
            }
        }
    }

    total_e = total_e - cut_count / a;

    time2 = clock();

    if (fabs(total_e - TARGET_E) > 0.0001)
    {
        printf("Bad E! Got %14.10f\n", total_e);
    }

    if (cut_count != TARGET_CUT)
    {
        printf("Incorrect number of pairs! Got %d\n", cut_count);
    }

    return (double) (time2 - time1);
}

double necessary_rij(double** coords, double* q, int natom, double cut)
{
    clock_t time1, time2;

    double total_e, current_e, vec2, rij;
    double a;
    int cut_count;
    int i, j;
    double cut2;

    a = 3.2;

    time1 = clock();

    total_e = 0.0;
    cut_count = 0;
    cut2 = cut * cut;

    for (i=1; i<=natom; ++i)
    {
        for (j=1; j<=natom; ++j)
        {
            if ( j < i )
            {
                vec2 = pow((coords[0][i-1]-coords[0][j-1]),2.0)
                      + pow((coords[1][i-1]-coords[1][j-1]),2.0)
                      + pow((coords[2][i-1]-coords[2][j-1]),2.0);
                if ( vec2 <= cut2 )
                {
                    rij = sqrt(vec2);
```

```
                ++cut_count;
                current_e = (exp(rij*q[i-1])*exp(rij*q[j-1]))/rij;
                total_e = total_e + current_e - 1.0/a;
            }
        }
    }
}

time2 = clock();

if (fabs(total_e - TARGET_E) > 0.0001)
{
    printf("Bad E! Got %14.10f\n", total_e);
}

if (cut_count != TARGET_CUT)
{
    printf("Incorrect number of pairs! Got %d\n", cut_count);
}

return (double) (time2 - time1);
}

double pull_out_a(double** coords, double* q, int natom, double cut)
{
clock_t time1, time2;

double total_e, current_e, vec2, rij;
double a;
int cut_count;
int i, j;

a = 3.2;

time1 = clock();

total_e = 0.0;
cut_count = 0;
for (i=1; i<=natom; ++i)
{
    for (j=1; j<=natom; ++j)
    {
        if ( j < i )
        {
            vec2 = pow((coords[0][i-1]-coords[0][j-1]),2.0)
                  + pow((coords[1][i-1]-coords[1][j-1]),2.0)
                  + pow((coords[2][i-1]-coords[2][j-1]),2.0);
            rij = sqrt(vec2);
            if ( rij <= cut )
            {
                ++cut_count;
                current_e = (exp(rij*q[i-1])*exp(rij*q[j-1]))/rij;
                total_e = total_e + current_e;
            }
        }
    }
}

total_e = total_e - (double) cut_count / a;
```

```
    time2 = clock();

    if (fabs(total_e - TARGET_E) > 0.0001)
    {
        printf("Bad E! Got %14.10f\n", total_e);
    }

    if (cut_count != TARGET_CUT)
    {
        printf("Incorrect number of pairs! Got %d\n", cut_count);
    }

    return (double) (time2 - time1);
}

double no_add_index(double** coords, double* q, int natom, double cut)
{
    clock_t time1, time2;

    double total_e, current_e, vec2, rij;
    double a;
    int cut_count;
    int i, j;

    a = 3.2;

    time1 = clock();

    total_e = 0.0;
    cut_count = 0;
    for (i = 0; i < natom; ++i)
    {
        for (j = 0; j <= natom; ++j)
        {
            if ( j < i )
            {
                vec2 = pow((coords[0][i] - coords[0][j]), 2.0)
                     + pow((coords[1][i] - coords[1][j]), 2.0)
                     + pow((coords[2][i] - coords[2][j]), 2.0);
                rij = sqrt(vec2);
                if ( rij <= cut )
                {
                    ++cut_count;
                    current_e = (exp(rij * q[i]) * exp(rij * q[j]))/rij;
                    total_e = total_e + current_e - 1.0/a;
                }
            }
        }
    }

    time2 = clock();

    if (fabs(total_e - TARGET_E) > 0.0001)
    {
        printf("Bad E! Got %14.10f\n", total_e);
    }

    if (cut_count != TARGET_CUT)
```

```
    {
        printf("Incorrect number of pairs! Got %d\n", cut_count);
    }

    return (double) (time2 - time1);
}

double no_if(double** coords, double* q, int natom, double cut)
{
    clock_t time1, time2;

    double total_e, current_e, vec2, rij;
    double a;
    int cut_count;
    int i, j;

    a = 3.2;

    time1 = clock();

    total_e = 0.0;
    cut_count = 0;
    for (i = 1; i <= natom; ++i)
    {
        for (j = 1; j < i; ++j)
        {
            vec2 = pow((coords[0][i-1]-coords[0][j-1]),2.0)
                  + pow((coords[1][i-1]-coords[1][j-1]),2.0)
                  + pow((coords[2][i-1]-coords[2][j-1]),2.0);
            rij = sqrt(vec2);
            if ( rij <= cut )
            {
                ++cut_count;
                current_e = (exp(rij*q[i-1])*exp(rij*q[j-1]))/rij;
                total_e = total_e + current_e - 1.0/a;
            }
        }
    }

    time2 = clock();

    if (fabs(total_e - TARGET_E) > 0.0001)
    {
        printf("Bad E! Got %14.10f\n", total_e);
    }

    if (cut_count != TARGET_CUT)
    {
        printf("Incorrect number of pairs! Got %d\n", cut_count);
    }

    return (double) (time2 - time1);
}

double combine_exp_combine_mult(double** coords, double* q, int natom, double cut)
{
    clock_t time1, time2;

    double total_e, current_e, vec2, rij;
```

```
    double a;
    int cut_count;
    int i, j;

    a = 3.2;

    time1 = clock();

    total_e = 0.0;
    cut_count = 0;
    for (i=1; i<=natom; ++i)
    {
        for (j=1; j<=natom; ++j)
        {
            if ( j < i )
            {
                vec2 = pow((coords[0][i-1]-coords[0][j-1]),2.0)
                      + pow((coords[1][i-1]-coords[1][j-1]),2.0)
                      + pow((coords[2][i-1]-coords[2][j-1]),2.0);
                rij = sqrt(vec2);
                if ( rij <= cut )
                {
                    ++cut_count;
                    current_e = (exp(rij * (q[i-1] + q[j-1])))/rij;
                    total_e = total_e + current_e - 1.0/a;
                }
            }
        }
    }

    time2 = clock();

    if (fabs(total_e - TARGET_E) > 0.0001)
    {
        printf("Bad E! Got %14.10f\n", total_e);
    }

    if (cut_count != TARGET_CUT)
    {
        printf("Incorrect number of pairs! Got %d\n", cut_count);
    }

    return (double) (time2 - time1);
}

double combine_exp(double** coords, double* q, int natom, double cut)
{
    clock_t time1, time2;

    double total_e, current_e, vec2, rij;
    double a;
    int cut_count;
    int i, j;

    a = 3.2;

    time1 = clock();

    total_e = 0.0;
```

```
    cut_count = 0;
    for (i=1; i<=natom; ++i)
    {
        for (j=1; j<=natom; ++j)
        {
            if ( j < i )
            {
                vec2 = pow((coords[0][i-1]-coords[0][j-1]),2.0)
                      + pow((coords[1][i-1]-coords[1][j-1]),2.0)
                      + pow((coords[2][i-1]-coords[2][j-1]),2.0);
                rij = sqrt(vec2);
                if ( rij <= cut )
                {
                    ++cut_count;
                    current_e = (exp(rij*q[i-1] + rij*q[j-1]))/rij;
                    total_e = total_e + current_e - 1.0/a;
                }
            }
        }
    }

    time2 = clock();

    if (fabs(total_e - TARGET_E) > 0.0001)
    {
        printf("Bad E! Got %14.10f\n", total_e);
    }

    if (cut_count != TARGET_CUT)
    {
        printf("Incorrect number of pairs! Got %d\n", cut_count);
    }

    return (double) (time2 - time1);
}

double original(double** coords, double* q, int natom, double cut)
{
    clock_t time1, time2;

    double total_e, current_e, vec2, rij;
    double a;
    int cut_count;
    int i, j;

    a = 3.2;

    time1 = clock();

    total_e = 0.0;
    cut_count = 0;
    for (i=1; i<=natom; ++i)
    {
        for (j=1; j<=natom; ++j)
        {
            if ( j < i )
            {
                vec2 = pow((coords[0][i-1]-coords[0][j-1]),2.0)
                      + pow((coords[1][i-1]-coords[1][j-1]),2.0)
```

```
                    + pow((coords[2][i-1]-coords[2][j-1]),2.0);
                rij = sqrt(vec2);
                if ( rij <= cut )
                {
                    ++cut_count;
                    current_e = (exp(rij*q[i-1])*exp(rij*q[j-1]))/rij;
                    total_e = total_e + current_e - 1.0/a;
                }
            }
        }
    }

    time2 = clock();

    if (fabs(total_e - TARGET_E) > 0.0001)
    {
        printf("Bad E! Got %14.10f\n", total_e);
    }

    if (cut_count != TARGET_CUT)
    {
        printf("Incorrect number of pairs! Got %d\n", cut_count);
    }

    return (double) (time2 - time1);
}

void test_func(double (*func_prt)(double**, double*, int, double), char* func_name, int argc,
     char* argv[])
{
    int n_test = 5;
    printf("\n\tTesting function %s\n", func_name);
    double avg_time = 0;
    int i;
    double test_time = 0;

    // NEW
    long natom;
    long cut_count;

    /* Timer variables */
    clock_t time0;

    double cut;    /* Cut off for Rij in distance units */
    double **coords;
    double *q;
    FILE *fptr;
    char *cptr;
    int fstatus;

    time0 = clock(); /*Start Time*/

    /* Step 1 - obtain the filename of the coord file and the value of
       cut from the command line.
               Argument 1 should be the filename of the coord file (char).
               Argument 2 should be the cut off (float). */
    /* Quit therefore if iarg does not equal 3 = executable name,
       filename, cut off */
    if (argc != 3)
```

```
{
    printf("ERROR: only %d command line options detected", argc-1);
    printf (" - need 2 options, filename and cutoff.\n");
    exit(1);
}
printf("\nCoordinates will be read from file: %s\n",argv[1]);

/* Step 2 - Open the coordinate file and read the first line to
   obtain the number of atoms */
if ((fptr=fopen(argv[1],"r"))==NULL)
{
    printf("ERROR: Could not open file called %s\n",argv[1]);
    exit(1);
}
else
{
    fstatus = fscanf(fptr, "%ld", &natom);
}

printf("Natom = %ld\n", natom);

cut = strtod(argv[2],&cptr);
printf("cut = %10.4f\n\n", cut);

/* Step 3 - Allocate the arrays to store the coordinate and charge
   data */
coords=alloc_2D_double(3,natom);
if ( coords==NULL )
{
    printf("Allocation error coords");
    exit(1);
}
q=(double *)malloc(natom*sizeof(double));
if ( q == NULL )
{
    printf("Allocation error q");
    exit(1);
}

/* Step 4 - read the coordinates and charges. */
for (i = 0; i<natom; ++i)
{
    fstatus = fscanf(fptr, "%lf %lf %lf %lf",&coords[0][i],
          &coords[1][i],&coords[2][i],&q[i]);
}

for (i = 0; i < n_test; i++) {

    test_time = (*func_prt)(coords, q, natom, cut);
    printf("Test %d: %f\n", i, test_time / (double) CLOCKS_PER_SEC);
    avg_time += test_time;

}

free(q);
double_2D_array_free(coords);

fclose(fptr);
```

```
    avg_time /= n_test;
    avg_time /= (double) CLOCKS_PER_SEC;
    printf("Average Time = %f\n", avg_time);
}

void original_run(int argc, char* argv[])
{
    long natom, i, j;
    long cut_count;

    /* Timer variables */
    clock_t time0, time1, time2;

    double cut;    /* Cut off for Rij in distance units */
    double **coords;
    double *q;
    double total_e, current_e, vec2, rij;
    double a;
    FILE *fptr;
    char *cptr;
    int fstatus;

    a = 3.2;

    time0 = clock(); /*Start Time*/
    printf("Value of system clock at start = %ld\n",time0);

    /* Step 1 - obtain the filename of the coord file and the value of
    *       cut from the command line.
    *                       Argument 1 should be the filename of the coord file (char).
    *                       Argument 2 should be the cut off (float). */
    /* Quit therefore if iarg does not equal 3 = executable name,
    *       filename, cut off */
    if (argc != 3)
    {
        printf("ERROR: only %d command line options detected", argc-1);
        printf (" - need 2 options, filename and cutoff.\n");
        exit(1);
    }
    printf("Coordinates will be read from file: %s\n",argv[1]);

    /* Step 2 - Open the coordinate file and read the first line to
    *       obtain the number of atoms */
    if ((fptr=fopen(argv[1],"r"))==NULL)
    {
        printf("ERROR: Could not open file called %s\n",argv[1]);
        exit(1);
    }
    else
    {
        fstatus = fscanf(fptr, "%ld", &natom);
    }

    printf("Natom = %ld\n", natom);

    cut = strtod(argv[2],&cptr);
    printf("cut = %10.4f\n", cut);

    /* Step 3 - Allocate the arrays to store the coordinate and charge
```

```
*       data */
coords=alloc_2D_double(3,natom);
if ( coords==NULL )
{
    printf("Allocation error coords");
    exit(1);
}
q=(double *)malloc(natom*sizeof(double));
if ( q == NULL )
{
    printf("Allocation error q");
    exit(1);
}

/* Step 4 - read the coordinates and charges. */
for (i = 0; i<natom; ++i)
{
    fstatus = fscanf(fptr, "%lf %lf %lf %lf",&coords[0][i],
        &coords[1][i],&coords[2][i],&q[i]);
}

time1 = clock(); /*time after file read*/
printf("Value of system clock after coord read = %ld\n",time1);


/* Step 5 - calculate the number of pairs and E. - this is the
*       majority of the work. */
total_e = 0.0;
cut_count = 0;
for (i=1; i<=natom; ++i)
{
    for (j=1; j<=natom; ++j)
    {
        if ( j < i )  /* Avoid double counting. */
        {
            vec2 = pow((coords[0][i-1]-coords[0][j-1]),2.0)
            + pow((coords[1][i-1]-coords[1][j-1]),2.0)
            + pow((coords[2][i-1]-coords[2][j-1]),2.0);
            /* X^2 + Y^2 + Z^2 */
            rij = sqrt(vec2);
            /* Check if this is below the cut off */
            if ( rij <= cut )
            {
                /* Increment the counter of pairs below cutoff */
                ++cut_count;
                current_e = (exp(rij*q[i-1])*exp(rij*q[j-1]))/rij;
                total_e = total_e + current_e - 1.0/a;
            }
        } /* if (j<i) */
    } /* for j=1 j<=natom */
} /* for i=1 i<=natom */

time2 = clock(); /* time after reading of file and calculation */
printf("Value of system clock after coord read and E calc = %ld\n",
    time2);

/* Step 6 - write out the results */
printf("                      Final Results\n");
printf("                      -------------\n");
```

```
    printf("                Num Pairs = %ld\n",cut_count);
    printf("                  Total E = %14.10f\n",total_e);
    printf("    Time to read coord file = %14.4f Seconds\n",
        ((double )(time1-time0))/(double )CLOCKS_PER_SEC);
    printf("       Time to calculate E = %14.4f Seconds\n",
        ((double )(time2-time1))/(double )CLOCKS_PER_SEC);
    printf("      Total Execution Time = %14.4f Seconds\n",
        ((double )(time2-time0))/(double )CLOCKS_PER_SEC);

    /* Step 7 - Deallocate the arrays - we should strictly check the
    *       return values here but for the purposes of this tutorial we can
    *             ignore this. */
    free(q);
    double_2D_array_free(coords);

    fclose(fptr);

    TARGET_E = total_e;
    TARGET_CUT = cut_count;
}

double **alloc_2D_double(int nrows, int ncolumns)
{
    /* Allocates a 2d_double_array consisting of a series of pointers
       pointing to each row that are then allocated to be ncolumns
       long each. */

    /* Try's to keep contents contiguous - thus reallocation is
       difficult! */

    /* Returns the pointer **array. Returns NULL on error */
    int i;

    double **array = (double **)malloc(nrows*sizeof(double *));
    if (array==NULL)
        return NULL;
    array[0] = (double *)malloc(nrows*ncolumns*sizeof(double));
    if (array[0]==NULL)
        return NULL;

    for (i = 1; i < nrows; ++i)
        array[i] = array[0] + i * ncolumns;

    return array;

}

void double_2D_array_free(double **array)
{
    /* Frees the memory previously allocated by alloc_2D_double */
    free(array[0]);
    free(array);
}
```

Script to Run Tests: `perf.sh`

Included here is the script to run all of the tests I did on one of the UVA CS SLURM nodes. I test the code at all optimization levels (O1, O2, and O3) for $N = 500, 1000, 2500, 5000, 10000,$ and 20000 each with $C = 0.5$ and $C = 0.1$. I also tested the code at all optimization levels (O1, O2, O3) for $C = 0.1, 0.2, 0.3, 0.4,$ and 0.5

for $N = 10000$.

```
#!/bin/bash
#
#SBATCH --job-name="CS 4444 HW 01" Name of the job which appears in squeue
#
#SBATCH --mail-type=ALL What notifications are sent by email
#SBATCH --mail-user=sgs7cr@virginia.edu
#
# Set up your user environment!!
#SBATCH --get-user-env
#
#SBATCH --error="perf.err"                Where to write stderr
#SBATCH --output="perf.out"             Where to write stdout

export NODE="nibbler4"
export SEED=1
export CUT=0.5
export SIZE=20000
IN=input_${SIZE}_${SEED}.txt

hostname
lscpu
echo ""
echo "----------- None ------------"
echo " SIZE = $SIZE CUT = $CUT "
./generate_input $SIZE $SEED > $IN
gcc -c opt.c
gcc -c kdtree.c
gcc opt.o kdtree.o -o opt -lm
./opt $IN $CUT
echo ""
echo "----------- O1  ------------"
echo " SIZE = $SIZE CUT = $CUT "
gcc -c opt.c -O1
gcc -c kdtree.c -O1
gcc opt.o kdtree.o -o optO1 -lm
./optO1 $IN $CUT
echo ""
echo "----------- O2  ------------"
echo " SIZE = $SIZE CUT = $CUT "
gcc -c opt.c -O2
gcc -c kdtree.c -O2
gcc opt.o kdtree.o -o optO2 -lm
./optO2 $IN $CUT
echo ""
echo "----------- O3  ------------"
echo " SIZE = $SIZE CUT = $CUT "
gcc -c opt.c -O3
gcc -c kdtree.c -O3
gcc opt.o kdtree.o -o optO3 -lm
./optO3 $IN $CUT

export SIZE=10000
IN=input_${SIZE}_${SEED}.txt
echo ""
hostname
lscpu
echo ""
```

```
echo "----------- None ------------"
echo " SIZE = $SIZE CUT = $CUT "
./generate_input $SIZE $SEED > $IN
gcc -c opt.c
gcc -c kdtree.c
gcc opt.o kdtree.o -o opt -lm
./opt $IN $CUT
echo ""
echo "----------- O1  ------------"
echo " SIZE = $SIZE CUT = $CUT "
gcc -c opt.c -O1
gcc -c kdtree.c -O1
gcc opt.o kdtree.o -o optO1 -lm
./optO1 $IN $CUT
echo ""
echo "----------- O2  ------------"
echo " SIZE = $SIZE CUT = $CUT "
gcc -c opt.c -O2
gcc -c kdtree.c -O2
gcc opt.o kdtree.o -o optO2 -lm
./optO2 $IN $CUT
echo ""
echo "----------- O3  ------------"
echo " SIZE = $SIZE CUT = $CUT "
gcc -c opt.c -O3
gcc -c kdtree.c -O3
gcc opt.o kdtree.o -o optO3 -lm
./optO3 $IN $CUT

export SIZE=5000
IN=input_${SIZE}_${SEED}.txt

echo ""
hostname
lscpu
echo ""
echo "----------- None ------------"
echo " SIZE = $SIZE CUT = $CUT "
./generate_input $SIZE $SEED > $IN
gcc -c opt.c
gcc -c kdtree.c
gcc opt.o kdtree.o -o opt -lm
./opt $IN $CUT
echo ""
echo "----------- O1  ------------"
echo " SIZE = $SIZE CUT = $CUT "
gcc -c opt.c -O1
gcc -c kdtree.c -O1
gcc opt.o kdtree.o -o optO1 -lm
./optO1 $IN $CUT
echo ""
echo "----------- O2  ------------"
echo " SIZE = $SIZE CUT = $CUT "
gcc -c opt.c -O2
gcc -c kdtree.c -O2
gcc opt.o kdtree.o -o optO2 -lm
./optO2 $IN $CUT
echo ""
echo "----------- O3  ------------"
```

```
echo " SIZE = $SIZE CUT = $CUT "
gcc -c opt.c -O3
gcc -c kdtree.c -O3
gcc opt.o kdtree.o -o opt03 -lm
./opt03 $IN $CUT

export SIZE=2500
IN=input_${SIZE}_${SEED}.txt

echo ""
hostname
lscpu
echo ""
echo "----------- None ------------"
echo " SIZE = $SIZE CUT = $CUT "
./generate_input $SIZE $SEED > $IN
gcc -c opt.c
gcc -c kdtree.c
gcc opt.o kdtree.o -o opt -lm
./opt $IN $CUT
echo ""
echo "----------- O1  ------------"
echo " SIZE = $SIZE CUT = $CUT "
gcc -c opt.c -O1
gcc -c kdtree.c -O1
gcc opt.o kdtree.o -o opt01 -lm
./opt01 $IN $CUT
echo ""
echo "----------- O2  ------------"
echo " SIZE = $SIZE CUT = $CUT "
gcc -c opt.c -O2
gcc -c kdtree.c -O2
gcc opt.o kdtree.o -o opt02 -lm
./opt02 $IN $CUT
echo ""
echo "----------- O3  ------------"
echo " SIZE = $SIZE CUT = $CUT "
gcc -c opt.c -O3
gcc -c kdtree.c -O3
gcc opt.o kdtree.o -o opt03 -lm
./opt03 $IN $CUT

export SIZE=1000
IN=input_${SIZE}_${SEED}.txt

echo ""
hostname
lscpu
echo ""
echo "----------- None ------------"
echo " SIZE = $SIZE CUT = $CUT "
./generate_input $SIZE $SEED > $IN
gcc -c opt.c
gcc -c kdtree.c
gcc opt.o kdtree.o -o opt -lm
./opt $IN $CUT
echo ""
echo "----------- O1  ------------"
echo " SIZE = $SIZE CUT = $CUT "
```

```
gcc -c opt.c -O1
gcc -c kdtree.c -O1
gcc opt.o kdtree.o -o opt01 -lm
./opt01 $IN $CUT
echo ""
echo "----------- O2  ------------"
echo " SIZE = $SIZE CUT = $CUT "
gcc -c opt.c -O2
gcc -c kdtree.c -O2
gcc opt.o kdtree.o -o opt02 -lm
./opt02 $IN $CUT
echo ""
echo "----------- O3  ------------"
echo " SIZE = $SIZE CUT = $CUT "
gcc -c opt.c -O3
gcc -c kdtree.c -O3
gcc opt.o kdtree.o -o opt03 -lm
./opt03 $IN $CUT

export SIZE=500
IN=input_${SIZE}_${SEED}.txt

echo ""
hostname
lscpu
echo ""
echo "----------- None ------------"
echo " SIZE = $SIZE CUT = $CUT "
./generate_input $SIZE $SEED > $IN
gcc -c opt.c
gcc -c kdtree.c
gcc opt.o kdtree.o -o opt -lm
./opt $IN $CUT
echo ""
echo "----------- O1  ------------"
echo " SIZE = $SIZE CUT = $CUT "
gcc -c opt.c -O1
gcc -c kdtree.c -O1
gcc opt.o kdtree.o -o opt01 -lm
./opt01 $IN $CUT
echo ""
echo "----------- O2  ------------"
echo " SIZE = $SIZE CUT = $CUT "
gcc -c opt.c -O2
gcc -c kdtree.c -O2
gcc opt.o kdtree.o -o opt02 -lm
./opt02 $IN $CUT
echo ""
echo "----------- O3  ------------"
echo " SIZE = $SIZE CUT = $CUT "
gcc -c opt.c -O3
gcc -c kdtree.c -O3
gcc opt.o kdtree.o -o opt03 -lm
./opt03 $IN $CUT

export CUT=0.4
export SIZE=10000
IN=input_${SIZE}_${SEED}.txt
```

```
echo ""
hostname
lscpu
echo ""
echo "----------- None ------------"
echo " SIZE = $SIZE CUT = $CUT "
./generate_input $SIZE $SEED > $IN
gcc -c opt.c
gcc -c kdtree.c
gcc opt.o kdtree.o -o opt -lm
./opt $IN $CUT
echo ""
echo "----------- O1  ------------"
echo " SIZE = $SIZE CUT = $CUT "
gcc -c opt.c -O1
gcc -c kdtree.c -O1
gcc opt.o kdtree.o -o optO1 -lm
./optO1 $IN $CUT
echo ""
echo "----------- O2  ------------"
echo " SIZE = $SIZE CUT = $CUT "
gcc -c opt.c -O2
gcc -c kdtree.c -O2
gcc opt.o kdtree.o -o optO2 -lm
./optO2 $IN $CUT
echo ""
echo "----------- O3  ------------"
echo " SIZE = $SIZE CUT = $CUT "
gcc -c opt.c -O3
gcc -c kdtree.c -O3
gcc opt.o kdtree.o -o optO3 -lm
./optO3 $IN $CUT

export CUT=0.3
IN=input_${SIZE}_${SEED}.txt

echo ""
hostname
lscpu
echo ""
echo "----------- None ------------"
echo " SIZE = $SIZE CUT = $CUT "
./generate_input $SIZE $SEED > $IN
gcc -c opt.c
gcc -c kdtree.c
gcc opt.o kdtree.o -o opt -lm
./opt $IN $CUT
echo ""
echo "----------- O1  ------------"
echo " SIZE = $SIZE CUT = $CUT "
gcc -c opt.c -O1
gcc -c kdtree.c -O1
gcc opt.o kdtree.o -o optO1 -lm
./optO1 $IN $CUT
echo ""
echo "----------- O2  ------------"
echo " SIZE = $SIZE CUT = $CUT "
gcc -c opt.c -O2
gcc -c kdtree.c -O2
```

```
gcc opt.o kdtree.o -o opt02 -lm
./opt02 $IN $CUT
echo ""
echo "----------- O3  ------------"
echo " SIZE = $SIZE CUT = $CUT "
gcc -c opt.c -O3
gcc -c kdtree.c -O3
gcc opt.o kdtree.o -o opt03 -lm
./opt03 $IN $CUT

export CUT=0.2
IN=input_${SIZE}_${SEED}.txt

echo ""
hostname
lscpu
echo ""
echo "----------- None ------------"
echo " SIZE = $SIZE CUT = $CUT "
./generate_input $SIZE $SEED > $IN
gcc -c opt.c
gcc -c kdtree.c
gcc opt.o kdtree.o -o opt -lm
./opt $IN $CUT
echo ""
echo "----------- O1  ------------"
echo " SIZE = $SIZE CUT = $CUT "
gcc -c opt.c -O1
gcc -c kdtree.c -O1
gcc opt.o kdtree.o -o opt01 -lm
./opt01 $IN $CUT
echo ""
echo "----------- O2  ------------"
echo " SIZE = $SIZE CUT = $CUT "
gcc -c opt.c -O2
gcc -c kdtree.c -O2
gcc opt.o kdtree.o -o opt02 -lm
./opt02 $IN $CUT
echo ""
echo "----------- O3  ------------"
echo " SIZE = $SIZE CUT = $CUT "
gcc -c opt.c -O3
gcc -c kdtree.c -O3
gcc opt.o kdtree.o -o opt03 -lm
./opt03 $IN $CUT

export CUT=0.1
IN=input_${SIZE}_${SEED}.txt

echo ""
hostname
lscpu
echo ""
echo "----------- None ------------"
echo " SIZE = $SIZE CUT = $CUT "
./generate_input $SIZE $SEED > $IN
gcc -c opt.c
gcc -c kdtree.c
gcc opt.o kdtree.o -o opt -lm
```

```
./opt $IN $CUT
echo ""
echo "----------- O1  ------------"
echo " SIZE = $SIZE CUT = $CUT "
gcc -c opt.c -O1
gcc -c kdtree.c -O1
gcc opt.o kdtree.o -o opt01 -lm
./opt01 $IN $CUT
echo ""
echo "----------- O2  ------------"
echo " SIZE = $SIZE CUT = $CUT "
gcc -c opt.c -O2
gcc -c kdtree.c -O2
gcc opt.o kdtree.o -o opt02 -lm
./opt02 $IN $CUT
echo ""
echo "----------- O3  ------------"
echo " SIZE = $SIZE CUT = $CUT "
gcc -c opt.c -O3
gcc -c kdtree.c -O3
gcc opt.o kdtree.o -o opt03 -lm
./opt03 $IN $CUT

export SIZE=20000
export CUT=0.1
IN=input_${SIZE}_${SEED}.txt

echo ""
hostname
lscpu
echo ""
echo "----------- None ------------"
echo " SIZE = $SIZE CUT = $CUT "
./generate_input $SIZE $SEED > $IN
gcc -c opt.c
gcc -c kdtree.c
gcc opt.o kdtree.o -o opt -lm
./opt $IN $CUT
echo ""
echo "----------- O1  ------------"
echo " SIZE = $SIZE CUT = $CUT "
gcc -c opt.c -O1
gcc -c kdtree.c -O1
gcc opt.o kdtree.o -o opt01 -lm
./opt01 $IN $CUT
echo ""
echo "----------- O2  ------------"
echo " SIZE = $SIZE CUT = $CUT "
gcc -c opt.c -O2
gcc -c kdtree.c -O2
gcc opt.o kdtree.o -o opt02 -lm
./opt02 $IN $CUT
echo ""
echo "----------- O3  ------------"
echo " SIZE = $SIZE CUT = $CUT "
gcc -c opt.c -O3
gcc -c kdtree.c -O3
gcc opt.o kdtree.o -o opt03 -lm
./opt03 $IN $CUT
```

```
export SIZE=5000
export CUT=0.1
IN=input_${SIZE}_${SEED}.txt

echo ""
hostname
lscpu
echo ""
echo "----------- None ------------"
echo " SIZE = $SIZE CUT = $CUT "
./generate_input $SIZE $SEED > $IN
gcc -c opt.c
gcc -c kdtree.c
gcc opt.o kdtree.o -o opt -lm
./opt $IN $CUT
echo ""
echo "----------- O1  ------------"
echo " SIZE = $SIZE CUT = $CUT "
gcc -c opt.c -O1
gcc -c kdtree.c -O1
gcc opt.o kdtree.o -o optO1 -lm
./optO1 $IN $CUT
echo ""
echo "----------- O2  ------------"
echo " SIZE = $SIZE CUT = $CUT "
gcc -c opt.c -O2
gcc -c kdtree.c -O2
gcc opt.o kdtree.o -o optO2 -lm
./optO2 $IN $CUT
echo ""
echo "----------- O3  ------------"
echo " SIZE = $SIZE CUT = $CUT "
gcc -c opt.c -O3
gcc -c kdtree.c -O3
gcc opt.o kdtree.o -o optO3 -lm
./optO3 $IN $CUT

export SIZE=2500
export CUT=0.1
IN=input_${SIZE}_${SEED}.txt

echo ""
hostname
lscpu
echo ""
echo "----------- None ------------"
echo " SIZE = $SIZE CUT = $CUT "
./generate_input $SIZE $SEED > $IN
gcc -c opt.c
gcc -c kdtree.c
gcc opt.o kdtree.o -o opt -lm
./opt $IN $CUT
echo ""
echo "----------- O1  ------------"
echo " SIZE = $SIZE CUT = $CUT "
gcc -c opt.c -O1
gcc -c kdtree.c -O1
gcc opt.o kdtree.o -o optO1 -lm
```

```
./opt01 $IN $CUT
echo ""
echo "----------- O2  ------------"
echo " SIZE = $SIZE CUT = $CUT "
gcc -c opt.c -O2
gcc -c kdtree.c -O2
gcc opt.o kdtree.o -o opt02 -lm
./opt02 $IN $CUT
echo ""
echo "----------- O3  ------------"
echo " SIZE = $SIZE CUT = $CUT "
gcc -c opt.c -O3
gcc -c kdtree.c -O3
gcc opt.o kdtree.o -o opt03 -lm
./opt03 $IN $CUT

export SIZE=1000
export CUT=0.1
IN=input_${SIZE}_${SEED}.txt

echo ""
hostname
lscpu
echo ""
echo "----------- None ------------"
echo " SIZE = $SIZE CUT = $CUT "
./generate_input $SIZE $SEED > $IN
gcc -c opt.c
gcc -c kdtree.c
gcc opt.o kdtree.o -o opt -lm
./opt $IN $CUT
echo ""
echo "----------- O1  ------------"
echo " SIZE = $SIZE CUT = $CUT "
gcc -c opt.c -O1
gcc -c kdtree.c -O1
gcc opt.o kdtree.o -o opt01 -lm
./opt01 $IN $CUT
echo ""
echo "----------- O2  ------------"
echo " SIZE = $SIZE CUT = $CUT "
gcc -c opt.c -O2
gcc -c kdtree.c -O2
gcc opt.o kdtree.o -o opt02 -lm
./opt02 $IN $CUT
echo ""
echo "----------- O3  ------------"
echo " SIZE = $SIZE CUT = $CUT "
gcc -c opt.c -O3
gcc -c kdtree.c -O3
gcc opt.o kdtree.o -o opt03 -lm
./opt03 $IN $CUT

export SIZE=500
export CUT=0.1
IN=input_${SIZE}_${SEED}.txt

echo ""
hostname
```

```
lscpu
echo ""
echo "----------- None ------------"
echo " SIZE = $SIZE CUT = $CUT "
./generate_input $SIZE $SEED > $IN
gcc -c opt.c
gcc -c kdtree.c
gcc opt.o kdtree.o -o opt -lm
./opt $IN $CUT
echo ""
echo "----------- O1  ------------"
echo " SIZE = $SIZE CUT = $CUT "
gcc -c opt.c -O1
gcc -c kdtree.c -O1
gcc opt.o kdtree.o -o optO1 -lm
./optO1 $IN $CUT
echo ""
echo "----------- O2  ------------"
echo " SIZE = $SIZE CUT = $CUT "
gcc -c opt.c -O2
gcc -c kdtree.c -O2
gcc opt.o kdtree.o -o optO2 -lm
./optO2 $IN $CUT
echo ""
echo "----------- O3  ------------"
echo " SIZE = $SIZE CUT = $CUT "
gcc -c opt.c -O3
gcc -c kdtree.c -O3
gcc opt.o kdtree.o -o optO3 -lm
./optO3 $IN $CUT
```