Parallel Computing with CUDA

Lab guide

This lab guide assumes that you are working under Linux (e.g. Ubuntu), and that you have a CUDA-enabled NVIDIA GPU. Also, the required software (namely the CUDA Toolkit) should have been correctly installed and configured.

Checking the hardware

1. Open a terminal and execute the command `lspci` to list all PCI devices attached to your machine.

2. Locate the NVIDIA devices on your system. For convenience, you may execute the command: `lspci | grep NVIDIA`

3. Check the architecture version (compute capability) of your GPU on this webpage: https://developer.nvidia.com/cuda-gpus

4. Take note of the compute capability of your GPU. You will need it later when compiling code for your device.

Example: squaring numbers

5. Create a new source file (square.cu) and open it in a text editor (gedit).

6. Start by including some header files that we will be needing in our program:

```
#include <cuda.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
```

7. After that, write the following code:

```
void square(double* A, double* B, int N)
{
      for(int i=0; i<N; i++)
      {
            B[i] = A[i]*A[i];
      }
}
```

The code above defines a host function that computes the square of every element in array A. The result is stored in array B. The size of both arrays is N.
Make sure you understand this code before proceeding.

8. Now we will create a kernel function to do the same thing on the GPU. Add the following code:

```
__global__ void kernel_square(double* d_A, double* d_B, int N)
{
      int i = blockIdx.x * blockDim.x + threadIdx.x;
      if (i < N)
      {
            d_B[i] = d_A[i]*d_A[i];
      }
}
```

The pointers d_A and d_B refer to arrays in device memory. The code above computes the square of every element in array d_A. The result is stored in array d_B. The size of both arrays is N.
Make sure you understand this code before proceeding.

9. After the functions above, begin the main() function:

```
int main(int argc, char **argv)
{
      if (argc < 2)
      {
            printf("Missing argument.\n");
            return 0;
      }

      int N = atoi(argv[1]);
```

The program expects a command-line argument (N, the size of the arrays). If the argument is not provided, it prints a message and returns immediately. Otherwise, it converts the argument to an integer N.
Make sure you understand this code before proceeding.

10. Add the following code to the main() function:

```
      double* A = (double*)malloc(N*sizeof(double));
      double* B = (double*)malloc(N*sizeof(double));
```

This code allocates (host) memory for two arrays (A and B) of length N.
Make sure you understand this code before proceeding.

11. After that, add the following code:

```
for(int i=0; i<N; i++)
{
      A[i] = sin(2.*M_PI*i/N);
}
```

This code initializes the array A with N points sampled from one cycle of a sinusoid.
M_PI is defined in math.h.
Make sure you understand this code before proceeding.

12. Call the host function square() to calculate the square of each element in array A, and store the result in array B.

```
square(A, B, N);
```

This calculation is taking place on the CPU. In the next steps, we will do it on the GPU.

13. Add the following code to declare two (device) arrays:

```
double* d_A;
double* d_B;
```

14. Now add the following instructions to allocate (device) memory for those arrays:

```
cudaMalloc(&d_A, N*sizeof(double));
cudaMalloc(&d_B, N*sizeof(double));
```

15. Transfer the contents of array A (in host memory) to array d_A (in device memory):

```
cudaMemcpy(d_A, A, N*sizeof(double), cudaMemcpyHostToDevice);
```

When calling this function, the destination array (in this case, d_A) is always specified before the source array (in this case, A).

16. We are almost ready to call our kernel function kernel_square(). However, before we do that, we need to know how many threads and blocks will be necessary.

As an example, we will use 128 threads per block.[1] In this case, we will need about N/128 blocks (to be rounded up).

We start by defining the number of threads per block, and then we calculate the required number of blocks:

[1] The ideal number of threads per block should be set according to the GPU architecture. It should be equal to the number of cores per streaming multiprocessor, or a multiple of that value, up to a maximum of 1024.

```
    dim3 threads(128);
    dim3 blocks((int)ceil((double)N/(double)threads.x));
```

The ceil() function rounds up a given floating point number.

17. Now we are ready to call our kernel function:

```
    kernel_square<<<blocks, threads>>>(d_A, d_B, N);
```

18. The kernel executes and the results are stored in array d_B in device memory. To transfer the results back to host memory, use the following instruction:

```
    cudaMemcpy(B, d_B, N*sizeof(double), cudaMemcpyDeviceToHost);
```

Note that this will overwrite any previous contents of B.

19. Now that we have the results in host memory, we can free the arrays in device memory:

```
    cudaFree(d_A);
    cudaFree(d_B);
```

20. We can also free the arrays in host memory:

```
    free(A);
    free(B);
```

21. Finally, we reset the GPU device and end the main() function:

```
    cudaDeviceReset();
    return 0;
}
```

Compiling and running the program

22. In a terminal, execute the following command to compile the program:

```
nvcc -arch=sm_20 square.cu -o square
```

Note: in the parameter "-arch=sm_20", replace "20" with the two digits that correspond to the compute capability of your GPU.

23. Try to run the program with the command:

```
./square
```

The program responds with:

```
Missing argument.
```

24. Now try to run the program with an argument such as:

```
./square 30
```

The program runs but produces no output.

Printing the results

25. Add the following function to the program:

```
void print_array(double* A, int N)
{
      for(int i=0; i<N; i++)
      {
            printf("%.3f ", A[i]);
      }
      printf("\n");
}
```

This function prints a host array of length N.
Make sure you understand this code before proceeding.

26. After calling the square() function, use print_array() to print the first 10 elements of B:

```
   print_array(B, 10);
```

27. After transferring the results from the device to the host with cudaMemcpy(B, d_B, ...), use print_array() to print the first 10 elements of B:

```
   print_array(B, 10);
```

28. Compile and run the program again:

```
nvcc -arch=sm_20 square.cu -o square
./square 30
```

The program should now produce the following output:

```
0.000 0.043 0.165 0.345 0.552 0.750 0.905 0.989 0.989 0.905
0.000 0.043 0.165 0.345 0.552 0.750 0.905 0.989 0.989 0.905
```

The CPU and GPU produce the same results (at least up to the precision begin shown).

Timing the host function and the kernel function

29. To measure the time it takes to execute the host function and the kernel function, we will use a helper function. Add the following function to your code:

```
inline double seconds()
{
      struct timeval tv;
      gettimeofday(&tv, NULL);
      return (double)tv.tv_sec + (double)tv.tv_usec*1.e-6;
}
```

This function returns the current time up to microsecond resolution.
The gettimeofday() function fills in a structure (timeval) with the number of seconds and microseconds since 1970-01-01 00:00:00. For more information, you can run in a terminal: `man gettimeofday`
You are not required to fully understand this code, just remember that you can call the seconds() function to get the current, wall-clock time (in seconds).

30. When calling the square() function, enclose that call with the following code:

```
      double t0 = seconds();
      square(A, B, N);
      double t1 = seconds();
      printf("CPU time: %.6f\n", t1-t0);
```

With these two calls to the seconds() function, we can determine the time elapsed while executing host function (t1-t0).

31. When calling the kernel_square() function, enclose that call with the following code:

```
      double t2 = seconds();
      kernel_square<<<blocks, threads>>>(d_A, d_B, N);
      cudaDeviceSynchronize();
      double t3 = seconds();
      printf("GPU time: %.6f\n", t3-t2);
```

Here, before the second call to the seconds() function, we call cudaDeviceSynchronize() to wait for the GPU to finish its work. Only then we determine the time elapsed (t3-t2).

32. Compile and run the program. The program should produce the following output:

```
CPU time: 0.000000
0.000 0.043 0.165 0.345 0.552 0.750 0.905 0.989 0.989 0.905
GPU time: 0.000019
0.000 0.043 0.165 0.345 0.552 0.750 0.905 0.989 0.989 0.905
```

Apparently, the GPU takes more time than the CPU to do the same thing.

33. Now run the program with increasing N and see what happens:

```
./square 300
CPU time: 0.000003
0.000 0.000 0.002 0.004 0.007 0.011 0.016 0.021 0.028 0.035
GPU time: 0.000019
0.000 0.000 0.002 0.004 0.007 0.011 0.016 0.021 0.028 0.035

./square 3000
CPU time: 0.000054
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
GPU time: 0.000022
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000

./square 30000
CPU time: 0.000543
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
GPU time: 0.000031
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000

./square 300000
CPU time: 0.004049
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
GPU time: 0.000079
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
```

In the CPU, the time it takes to run the host function is increasing linearly with N. However, in the GPU, the time it takes to run the kernel function increases only slightly, because most of the work is being done in parallel.

Congratulations! You have finished the first example. Now let's do an exercise.

Exercise: checking that $\sin^2\theta + \cos^2\theta = 1$

In this exercise we will use five arrays (A, B, C, D, E) as illustrated in the following table:

A[i] = sin(2.*M_PI*i/N);	B[i] = A[i]*A[i];	C[i] = cos(2.*M_PI*i/N);	D[i] = C[i]*C[i];	E[i] = B[i]+D[i];
0.000	0.000	1.000	1.000	1.000
0.208	0.043	0.978	0.957	1.000
0.407	0.165	0.914	0.835	1.000
0.588	0.345	0.809	0.655	1.000
0.743	0.552	0.669	0.448	1.000
0.866	0.750	0.500	0.250	1.000
0.951	0.905	0.309	0.095	1.000
0.995	0.989	0.105	0.011	1.000
0.995	0.989	-0.105	0.011	1.000
0.951	0.905	-0.309	0.095	1.000
0.866	0.750	-0.500	0.250	1.000
...	...	...	...	...

34. Make a copy of the previous program (square.cu) and save it as a new file (sqsum.cu). We will be working on this new file.

35. Write a host function sum() to sum two (host) arrays A and B, and store the results in array C. All arrays have length N.

36. Write a kernel function kernel_sum() to sum two (device) arrays d_A and d_B, and store the results in array d_C. All arrays have length N.

37. In the main() function of the program, allocate the (host) arrays C, D, E.

38. Before we forget, free those (host) arrays at the end of the program.

39. Initialize the (host) array C according to the formula in the table above.

40. Call the square() function to square C and store the results in D.

41. Call the sum() function to sum B and D, and store the results in E.

42. Print the first 10 elements of array E.

43. Declare three device arrays d_C, d_D, d_E.

44. Allocate those arrays in device memory.

45. Before we forget, free those (device) arrays at the end of the program.

46. Transfer the contents of the (host) array C to the (device) array d_C.

47. Call kernel_square() to square d_C and store the results in d_D.

48. Call kernel_sum() to sum d_B and d_D, and store the results in d_E.

49. Transfer the results in the (device) arrays d_D, d_E to the (host) arrays D, E.

50. Print the first 10 elements of array E.

51. Compile and run the program.

52. Check that the results are all 1.0 as expected.

53. Try increasing N until you see that the GPU is significantly faster than the CPU.

Congratulations! You have just finished the first exercise.
Now let's move on to a second example.

Example: FFT of a sine wave

54. Create a new source file (sinewave.cu) and open it in a text editor.

55. Start by including some header files that we will be needing in our program:

```
#include <cuda.h>
#include <cufft.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
```

Note that we have included here a new header file (cufft.h).

56. Start the main() function as we did previously:

```
int main(int argc, char **argv)
{
	if (argc < 2)
	{
		printf("Missing argument.\n");
		return 0;
	}

	int N = atoi(argv[1]);
```

57. Initialize two host arrays of type cufftComplex and length N.

```
	cufftComplex* A = (cufftComplex*)malloc(N*sizeof(cufftComplex));
	cufftComplex* B = (cufftComplex*)malloc(N*sizeof(cufftComplex));
```

58. Initialize the contents of array A with one cycle of a sine wave:

```
	for(int i=0; i<N; i++)
	{
		A[i].x = sin(2*M_PI*i/N);
		A[i].y = 0.0;
	}
```

The signal is stored in the real part of A. The imaginary part is zero.

59. Declare two device arrays of type cufftComplex.

```
	cufftComplex* d_A;
	cufftComplex* d_B;
```

60. Allocate memory for these two device arrays:

```
cudaMalloc(&d_A, N*sizeof(cufftComplex));
cudaMalloc(&d_B, N*sizeof(cufftComplex));
```

61. Copy the contents of the host array A to the device array d_A:

```
cudaMemcpy(d_A, A, N*sizeof(cufftComplex), cudaMemcpyHostToDevice);
```

62. Create a plan for executing the FFT:

```
cufftHandle plan;
cufftPlan1d(&plan, N, CUFFT_C2C, 1);
```

63. Execute the FFT:

```
cufftExecC2C(plan, d_A, d_B, CUFFT_FORWARD);
```

The FFT will be stored in the device array d_B.

64. Copy the results from device memory (d_B) to host memory (B):

```
cudaMemcpy(B, d_B, N*sizeof(cufftComplex), cudaMemcpyDeviceToHost);
```

65. Print the original signal together with its FFT:

```
for(int i=0; i<N; i++)
{
      printf("%10.3f %10.3f %10.3f %10.3f\n",
             A[i].x, A[i].y, B[i].x, B[i].y);
}
```

66. Free the resources associated with the plan:

```
cufftDestroy(plan);
```

67. Free the device arrays:

```
cudaFree(d_A);
cudaFree(d_B);
```

68. Free the host arrays:

```
free(A);
```

```
    free(B);
```

69. Reset the device and end the main() function:

```
    cudaDeviceReset();
    return 0;
}
```

70. Compile the program with the -lcufft option:

```
nvcc -arch=sm_20 sinewave.cu -lcufft -o sinewave
```

The option "-lcufft" tells the compiler to link the program with the cuFFT library.

71. Execute the program:

```
./sinewave 30
    0.000      0.000      0.000      0.000
    0.208      0.000      0.000    -15.000
    0.407      0.000     -0.000      0.000
    0.588      0.000      0.000      0.000
    0.743      0.000      0.000     -0.000
    0.866      0.000     -0.000      0.000
    0.951      0.000     -0.000      0.000
      ...        ...        ...        ...
```

Congratulations! You have finished the second example.
Now let's do an exercise.

Exercise: checking that $FFT(x+y) = FFT(x) + FFT(y)$

In this exercise, we will be working with six arrays (A, B, C, D, E, F) as follows:

- A contains a signal: A[i].x = sin(2.*M_PI*i/N);
- B contains the FFT of A
- C contains a signal: C[i].x = sin(6.*M_PI*i/N);
- D contains the FFT of C
- E contains the sum of A and C
- F contains the FFT of E

72. Make a copy of the previous program (sinewave.cu) and save it as a new file (sumfft.cu). We will be working on this new file.

73. Write a kernel function kernel_sum_complex() to sum two device arrays d_A and d_B of type cufftComplex, and store the results in array d_C. All arrays have length N. You can adapt the kernel function from the previous exercise. Remember that here we are summing complex numbers, with separate real and imaginary parts.

74. In the main() function of the program, allocate the host arrays C, D, E, F of type cufftComplex and length N.

75. Before we forget, free those host arrays at the end of the program.

76. Initialize the host array C according to the formula given above (a sine wave with three times the frequency of A).

77. Declare the device arrays d_C, d_D, d_E, d_F.

78. Allocate those arrays in device memory.

79. Before we forget, free those device arrays at the end of the program.

80. Transfer the contents of the host array C to the device array d_C.

81. Execute the FFT of d_C and store the results in d_D. You can reuse the existing plan.

82. After executing the FFT, call the kernel function kernel_sum_complex() to sum d_A and d_C, and store the results in d_E. You will need to define the number of threads and blocks before calling the kernel (see the previous exercise).

83. After calling the kernel, execute the FFT of d_E and store the results in d_F. You can reuse the existing plan.

84. Transfer the results in the device arrays d_D, d_E, d_F to the host arrays D, E, F.

85. Print the contents of all host arrays that contain FFTs (B, D, F).

86. Compile and run the program.

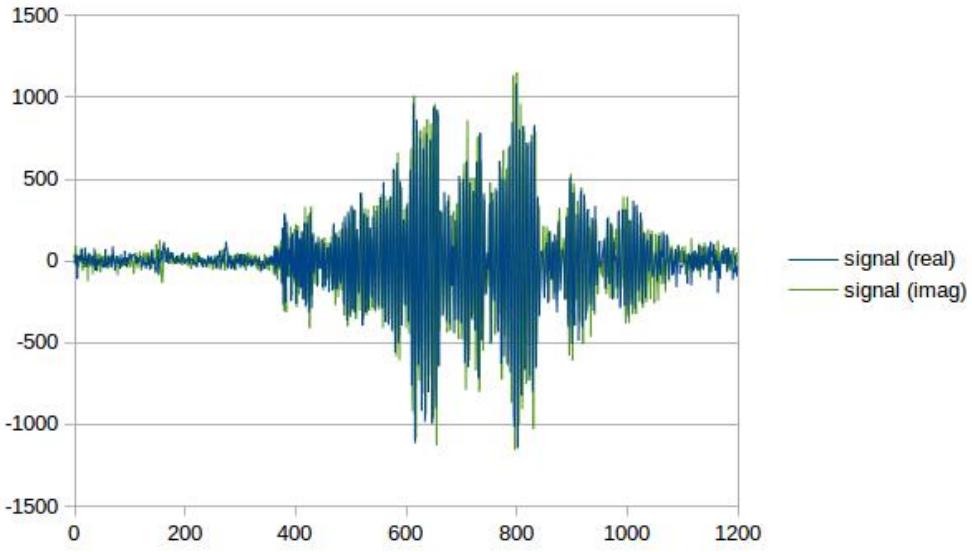
87. Check that F (the FFT of A+C) is equal to B plus D (the FFTs of A and C, respectively).

Congratulations! You have finished the second exercise.
Now let's do a small project.

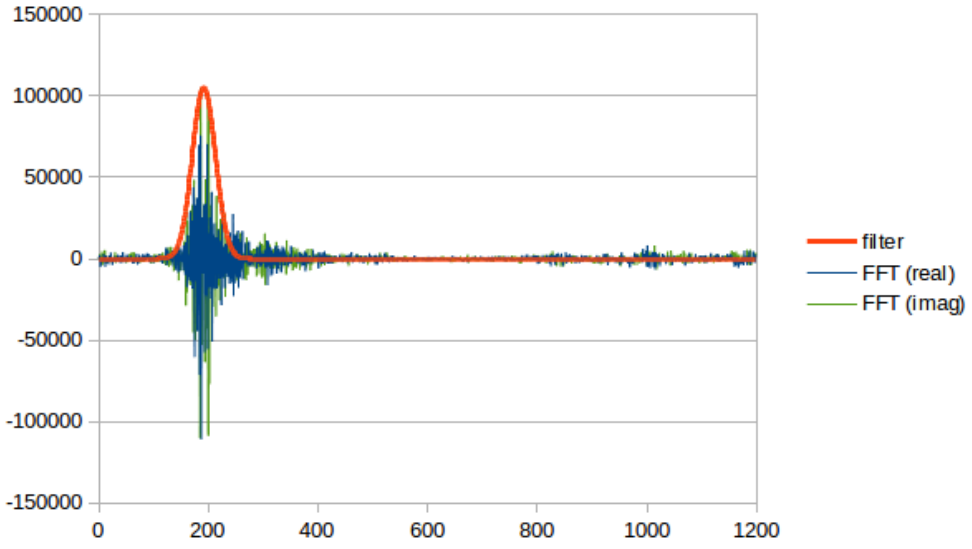
Project

The file "iq_data.csv" contains actual data from a reflectometry system installed at JET. The file contains the I/Q (complex) signal coming from the reflectometer. The first column is the real part, and the second column is the imaginary part. The file has 1200 readings.

The following figure shows a plot of the real and imaginary parts of the signal.

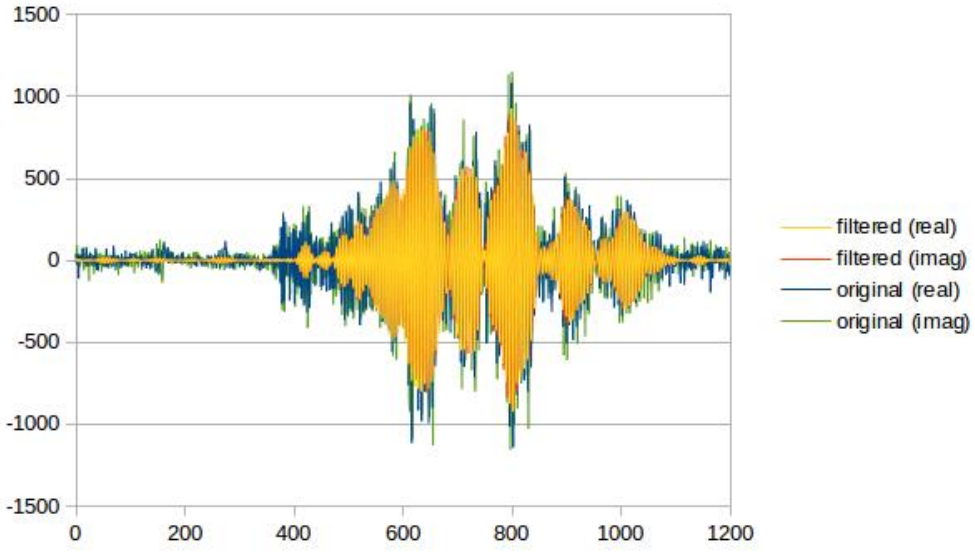


If we compute the FFT of this signal, we find that its main frequencies are located around the point 200 in the spectrum.



The idea is to apply a Gaussian filter around that point to keep only the main frequencies and throw away the rest.

The signal is then reconstructed by applying the inverse FFT to the filtered spectrum. The following figure shows how the reconstructed signal will look like.



In this project, you are asked to develop a program (project.cu) to do the following:

- The program should receive, as command-line arguments, the parameters μ and σ for the Gaussian filter.[2]

- The size of all arrays to be used in the program is N=1200.

- To read the "iq_data.csv" file, you can use the following function:

```
void read_iq_data(cufftComplex* A, int N)
{
    FILE* fin = fopen("iq_data.csv", "r");
    for (int i=0; i<N; i++)
    {
        char line[256];
        fgets(line, sizeof(line), fin);
        A[i].x = atof(strtok(line, " "));
        A[i].y = atof(strtok(NULL, " "));
    }
    fclose(fin);
}
```

  Note that the host array A must have been allocated before calling this function.

- Transfer the signal from host memory to device memory.

[2] The expression for the Gaussian filter is: $f(x) = e^{-\left(\frac{x-\mu}{\sigma}\right)^2}$.

- Use the cuFFT library to compute the FFT of the signal on the GPU.
- Write a kernel to multiply the FFT by a Gaussian filter with the parameters μ and σ.
- Execute the inverse FFT to get the filtered signal. This can be done with:

```
cufftExecC2C(plan, d_input, d_output, CUFFT_INVERSE);
```

- After computing the inverse FFT, it is necessary to normalize the results by applying the normalization factor 1/N. Write a kernel to do this.
- Transfer all results from device memory to host memory.
- Print the original signal, its FFT, the FFT after applying the filter, and the filtered signal. The output should be similar to this:

```
./project 192 30
    36.876     21.273      0.087     -0.216      0.000     -0.000     -0.356    -10.770
   -24.124     58.273   1454.714   1191.457      0.000      0.000      8.416     -6.023
   -15.124     92.273   2186.451   2706.694      0.000      0.000      9.254      3.469
    41.876     37.273  -4976.688   -805.571     -0.000     -0.000      2.178      9.127
    15.876    -12.727  -4099.534    520.496     -0.000      0.000     -5.993      6.525
  -110.124     23.273   -424.752  -2036.704     -0.000     -0.000     -8.227     -1.246
     3.876     33.273  -1161.589   4625.549     -0.000      0.000     -3.308     -7.038
    28.876    -34.727   -708.667  -3632.159     -0.000     -0.000      3.765     -6.179
    13.876    -35.727  -1310.314   3195.170     -0.000      0.000      6.710     -0.190
     9.876     22.273   3005.672  -2609.039      0.000     -0.000      3.471      5.161
    36.876      0.273   1020.236  -1228.084      0.000     -0.000     -2.408      5.246
    -2.124     28.273   -559.376   5300.548     -0.000      0.000     -5.365      0.538
     1.876     76.273  -4929.751   1608.794     -0.000      0.000     -2.927     -4.177
    73.876     29.273   2097.769  -2206.879      0.000     -0.000      2.204     -4.399
    10.876    -48.727   -405.648  -1986.045     -0.000     -0.000      4.865     -0.101
    12.876    -61.727  -2305.566  -2804.830     -0.000     -0.000      2.392      4.320
   -17.124     20.273    -54.221  -1483.808     -0.000     -0.000     -2.769      4.311
    -7.124    -40.727   -726.707  -1469.385     -0.000     -0.000     -5.385     -0.333
    25.876     21.273  -3338.578   1311.522     -0.000      0.000     -2.564     -5.109
    79.876     46.273   2058.529  -1296.933      0.000     -0.000      3.223     -5.119
       ...        ...        ...        ...        ...        ...        ...        ...
```

That's it! Good luck!