RAY TRACING OF PHOTONS IN WATER USING CUDA

A Project Report

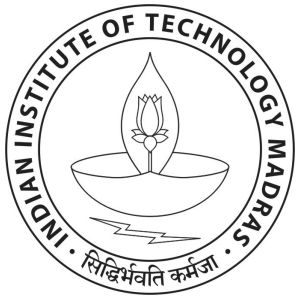
submitted by

SONU BADAIK

EE14B056

in partial fulfilment of the requirements
for the award of the degree of

BACHELOR OF TECHNOLOGY



DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS.

JUNE 2018

CERTIFICATE

This is to certify that the report titled **RAY TRACING OF PHOTONS IN WATER USING CUDA**, submitted by **Sonu Badaik**, to the Indian Institute of Technology, Madras, for the award of the degree of **Bachelor of Technology**, is a bona fide record of the research work done by him under our supervision. The contents of this report, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Harishankar Ramachandran
Project Guide
Dept. of Electrical Engineering
IIT-Madras, 600 036
Place: Chennai
Date: 15th June 2018

ACKNOWLEDGEMENTS

I would like to express my special thanks of gratitude to my Professor, Dr. Harishankar Ramachandran as well as our Head of Department, Dr. Devendra Jalihal who gave me the golden opportunity to do this wonderful project on the topic **Ray Tracing of Photons in Water Using CUDA**, which also helped me in doing a lot of research and I came to know about so many new things and I am really thankful to them.

Secondly I would also like to thank my friends who helped me a lot in finalizing this project within the limited time frame.

Sonu Badaik
EE14B056
Dept. of Electrical Engineering
IIT-Madras, 600 036
Place: Chennai
Date: 15th June 2018

TABLE OF CONTENTS

Contents

1 Introduction

1.1 Objective

In this project, we are projecting N (N=100000) number of rays into the sea from the sea level. The rays will get scattered inside the water in different directions. Inside the sea there is a submarine, on which the rays will hit. We have scattering data file with various angles and depths, which gives information about the rays. Our goal is to track these rays and calculate how many rays are hitting the submarine.

1.2 Serial Computing

Traditionally, software has been written for **serial** computation. In this method a problem is broken into series of instructions. Instructions are then executed sequentially one after another. These instructions are executed on a single processor. Only one instruction may execute at any moment in time. This takes more amount of time to execute a single program.

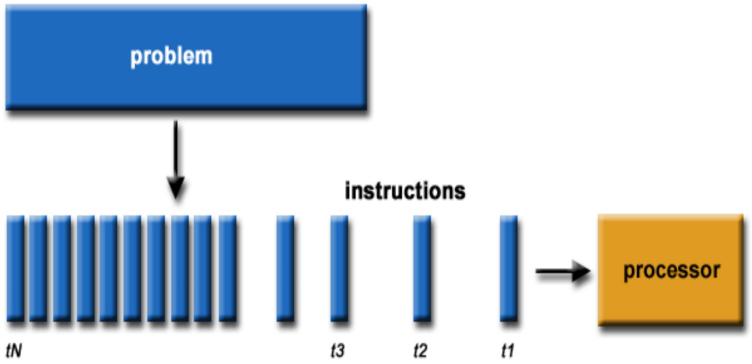


Figure 1: Program Execution in Serial Computing

1.3 Parallel Computing

Parallel Computing uses multiple processor to execute a single program. In this method a problem is broken into discrete parts that can be solved concurrently. Each part is further broken down to a series of instructions. Instructions from each part execute simultaneously on different processors, therefore Parallel Computing is faster than the Serial Computing.

The computational problem should be able to:

- Be broken apart into discrete pieces of work that can be solved simultaneously
- Execute multiple program instructions at any moment in time
- Be solved in less time with multiple compute resources than with a single compute resource

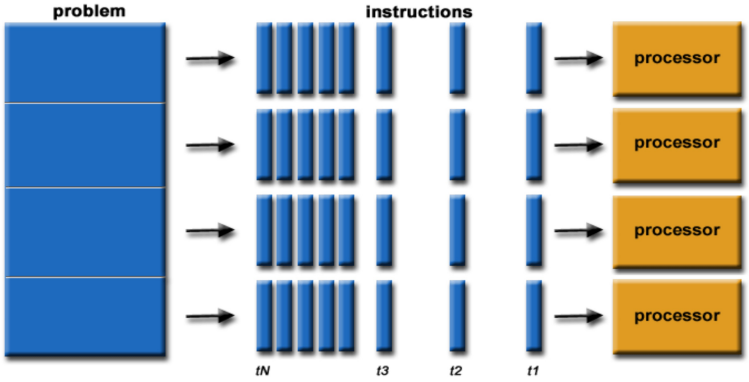


Figure 2: Program Execution in Parallel Computing

2 OpenMP

OpenMp (Open Multi-Processing) is a programming platform that helps in parallelizing code over a shared memory system (*e.g. a multi-core processor*). One can parallelize a set of operations over a multi-core processor, where the cores share memory between each other. This memory can be cache memory, RAM, hard disk memory etc. OpenMP supports C, C++ and Fortran on a wide variety of architectures.

OpenMP programs accomplish parallelism exclusively through the use of threads. A thread of execution is the smallest unit of processing that can be scheduled by an operating system. We can say it is a subroutine that can be scheduled to run autonomously. Each thread has its own program counter and executes one instruction at a time, similar to sequential program execution. Typically, the number of threads match the number of machine processors/cores. However, the actual use of threads is up to the application.

OpenMP also has some additional features such as parallel for loops which allow the programmer to specify whether the for loop iterations are independent of one another and hence, can be executed in parallel. It also allows for more sophisticated for loop optimization using reduction clauses which enable a for loop to operate on data in "chunks" and then combine these "chunks" at the end. This is particularly useful for loops which may, for instance, involve summing a set of data.

2.1 Loop Parallelization

Loop-Level Parallelism :

- Individual loops can be parallelized
- Each thread is assigned a unique range of the loop index
- Execution starts on a single serial thread

Requirements for Loop Parallelization :

- No dependencies between loop indices
- An element of an array is assigned to by at most one iteration
- No loop iteration reads array elements modified by any other dependency

2.2 Important Directives and Clauses

The followings are the list of important Directives and Clauses in Openmp.

Table 1: OpenMP Directives

Directive	Description
atomic	Specifies that a memory location that will be updated automically.
barrier	Synchronizes all threads in a team; all threads pause at the barrier, until all threads execute the barrier.
critical	Specifies that code is only executed on one thread at a time.
for	Causes the work done in a for loop inside a parallel region to be divided among threads.
master	Specifies that only the master thread should execute a section of the program.
ordered	Specifies that code under a parallelized for loop should be executed like a sequential loop.
parallel	Defines a parallel region, which is code that will be executed by multiple threads in parallel.
sections	Identifies code sections to be divided among all threads.
single	Lets you specify that a section of code should be executed on a single thread, not necessarily the master thread.
threadprivate	Specifies that a variable is private to a thread.

Table 2: OpenMP Clauses

Clause	Description
copyin	Allows threads to access the master thread's value, for a threadprivate variable.
default	Specifies the behavior of unscoped variables in a parallel region.
nowait	Overrides the barrier implicit in a directive.
num_threads	Sets the number of threads in a thread team.
ordered	Required on a parallel for statement if an ordered directive is to be used in the loop.
private	Specifies that each thread should have its own instance of a variable.
reduction	Specifies that one or more variables that are private to each thread are the subject of a reduction operation at the end of the parallel region.
schedule	Applies to the for directive.
shared	Specifies that one or more variables should be shared among all threads.

2.3 Matrix Multiplication: Serial and Parallel

Listing 1: Matrix Multiplication

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4  #include <omp.h>
5
6  int main()
7  {
8      int N = 2000, i, j, k;
9
10     /* allocating memory to arrays*/
11     int **mat1, **mat2, **res, **trp;
12     mat1 = (int **)malloc(N * sizeof(int*));
13     mat2 = (int **)malloc(N * sizeof(int*));
14     res = (int **)malloc(N * sizeof(int*));
15     trp = (int **)malloc(N * sizeof(int*));
16
17     for(i=0;i<N;i++){
18         mat1[i] = (int *)malloc(N*sizeof(int));
```

```
19          mat2[i] = (int *)malloc(N*sizeof(int));
20          res[i] = (int *)malloc(N*sizeof(int));
21          trp[i] = (int *)malloc(N*sizeof(int));
22      }
23
24      clock_t start, end;
25      double cpu_time_used;
26
27      /* initialization of arrays*/
28      for(i=0;i<N;i++){
29          for(j=0;j<N;j++){
30              mat1[i][j] = i + 1;
31              mat2[i][j] = i + 2;
32          }
33      }
34
35      /* matrix multiplication in serial*/
36      start = clock();
37      for(i=0;i<N;i++){
38          for(j=0;j<N;j++){
39              res[i][j] = 0;
40              for (k=0;k<N;k++){
41                  res[i][j] += mat1[i][k]*mat2[k][j];
42              }
43          }
44      }
45      end = clock();
46
47      cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
48      printf("Time taken to excecute in Serial: %lf\n",cpu_time_used);
49
50      /* matrix multiplication in parallel*/
51      start = clock();
52      #pragma omp parallel private(i,j,k) shared(mat1,mat2,res)
53      {
54          #pragma omp for collapse(2) schedule(dynamic,5000) nowait
55              for(i=0;i<N;i++){
56                  for(j=0;j<N;j++){
57                      res[i][j] = 0;
58                      for(k=0;k<N;k++){
59                          res[i][j] += mat1[i][k]*mat2[k][j];
60                      }
61                  }
62              }
63      }
64
65      end = clock();
```

```
66      cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
67      printf("Time taken to excecute in Parallel: %lf\n",cpu_time_used);
68
69      /* matrix multiplication in parallel using transpose matrix*/
70      start = clock();
71      #pragma omp parallel private(i,j,k) shared(mat1,mat2,res)
72      {
73          /* transpose of the second matrix */
74          #pragma omp for collapse(2) schedule(dynamic,1000)
75              for(i=0;i<N;i++){
76                  for(j=0;j<N;j++){
77                      trp[i][j] = mat2[j][i];
78                  }
79              }
80
81          #pragma omp for collapse(2) schedule(dynamic,5000) nowait
82              for(i=0;i<N;i++){
83                  for(j=0;j<N;j++){
84                      res[i][j] = 0;
85                      for(k=0;k<N;k++){
86                          res[i][j] += mat1[i][k]*trp[i][k];
87                      }
88                  }
89              }
90      }
91
92      end = clock();
93      cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
94      printf("Time taken to excecute in Parallel (Using Transpose): %lf\n",↩
            cpu_time_used);
95
96      free(mat1);
97      free(mat2);
98      free(res);
99      free(trp);
100
101     return 0;
102 }
```

In the first part of the code we are using traditional way of matrix multiplication. And we are calculating it's time taken. In the second part we are paralyzing the for loop which calculates matrix multiplication. And in the third part we are paralyzing the for which calculates matrix multiplication but first doing transpose of the second matrix.

Table 3: Time taken to execute the code in seconds

Dimension	Serial	Parallel Time	Parallel(Using Transpose)
100 X 100	0.015	0.016	0.000
200 X 200	0.062	0.031	0.031
400 X 400	0.437	0.283	0.233
600 X 600	1.544	1..062	0.787
800 X 800	4.721	2.737	1.868
1000 X 1000	14.221	6.557	3.647
1200 X 1200	61.559	32.288	18.959
1400 X 1400	106.014	55.081	30.560
1600 X 1600	171.507	86.906	45.171
1800 X 1800	251.937	138.095	73.799
2000 X 2000	361.892	204.729	94.842

3 CPU and GPU

The **CPU (central processing unit)** has often been called the brains of the PC. But increasingly, that brain is being enhanced by another part of the PC – the **GPU (graphics processing unit)**, which is its soul. CPUs and GPUs are pretty similar. They're both made from hundreds of millions of transistors and can process thousands of operations per second.

3.1 What is CPU?

The CPU is a collection of millions of transistors that can be manipulated to perform an awesome variety of calculations. A standard CPU has between one and four processing cores clocked anywhere from 1 to 4 GHz. A CPU is powerful because it can do everything. If a computer is capable of accomplishing a task, it's because the CPU can do it. Programmers achieve this through broad instruction sets and long feature lists shared by all CPUs.

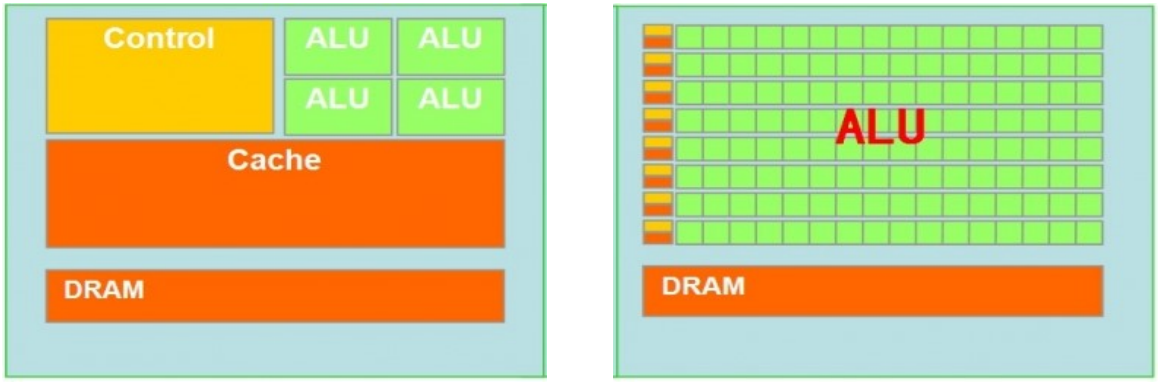
3.2 What is GPU?

A GPU is a specialized type of microprocessor. It's optimized to display graphics and do very specific computational tasks. It runs at a lower clock speed than a CPU but has many times the number of processing cores. We can almost think of a GPU as a specialized CPU that's been built for a very specific purpose. Video rendering is all about doing simple mathematical operations over and over again, and that's what a GPU is best at. A GPU will have thousands of processing cores running simultaneously. Each core, though slower than a CPU core, is tuned to be especially efficient at the basic mathematical operations required for video rendering.

3.3 Difference Between CPU and GPU

Architecturally, the CPU is composed of just few cores with lots of cache memory that can handle a few software threads at a time. In contrast, a GPU is composed of hundreds of cores that can handle thousands of threads simultaneously. The ability of a GPU with 100 plus cores to process thousands of threads can accelerate some software by 100 times over a CPU alone. What's more, the GPU achieves this acceleration while being more power and cost-efficient than a CPU.

A GPU can only do a fraction of the many operations a CPU does, but it does so with incredible speed. However, CPUs are more flexible than GPUs. CPUs have a larger instruction set, so they can perform a wider range of tasks. CPUs also run at higher maximum clock speeds and are capable of managing the input and output of all of a computer's components. For example, CPUs can organize and integrate with virtual memory, which is essential for running a modern operating system. That's just not something a GPU can accomplish.



(a) Memory Architecture in CPU (b) Memory Architecture in GPU

Figure 3: Memory Architecture in CPU and GPU

3.4 GPU for Parallel Programming

Due to presence of a very high number of cores in GPU compare to CPU, it is preferred more in the parallel computing field. GPU is very much faster than CPU, but while coming to handle the GPU cache memory, the programmer needs to take cake of it. Normally, copying variables from CPU to GPU takes more time than computation takes in GPU. Therefore, one needs to consider this fact and should use parallel computing in GPU effectively. GPUs are generally good for high memory computation. For example, matrix multiplication for lower dimension takes more time in GPU compare to CPU. However, for higher dimension GPU is much faster than CPU.

4 CUDA

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model developed by NVIDIA for general computing on graphical processing units (GPUs). With CUDA, developers are able to dramatically speed up computing applications by harnessing the power of GPUs. In GPU-accelerated applications, the sequential part of the workload runs on the CPU – which is optimized for single-threaded performance – while the compute intensive portion of the application runs on thousands of GPU cores in parallel. When using CUDA, developers program in popular languages such as C, C++, Fortran, Python and MATLAB and express parallelism through extensions in the form of a few basic keywords.

4.1 CUDA and C

C is a programming language, using which we write program to do certain task, and CUDA is a parallel programming platform which uses C syntax. C runs on the CPU while CUDA uses GPU to perform the task. We can assume GPU as a co-processor to accelerate CPU for parallel computing.

The GPU accelerates program running on the CPU by doing some of the compute-intensive and time consuming portions of the code. The rest of the application still runs on the CPU. From a user's perspective, the application runs faster because it's using the massively parallel processing power of the GPU to boost performance. This is known as "heterogeneous" or "hybrid" computing.

4.2 Programming flow on CUDA

We refer CPU and it's memory as *host* and GPU and it's memory as *device*. GPU uses a *kernel* to excecute code. A *kernel* is basically a *function* which runs on the device. A variable in CPU memory cannot be accessed directly in a GPU kernel and a variable in GPU memory cannot be accessed by CPU function. Therefore, we need to maintain copy of variables in both CPU and GPU.

CUDA adds the `__global__` qualifier to standard C. This mechanism alerts the compiler that a function (*kernel*) should be compiled to run on the device instead of the host. We add this qualifier before a kernel. This function or kernel is called from the CPU by using verb|kernel«<1,1»>|. Where, **kernel** is the name of the kernel and inside the three angular brackets suggests the number of threads and blocks to be launched. These are the basic steps to excecute a pragram in CUDA :

1. *Load data to into CPU memory.* This is the standard way of declaring variables or constants in C.

2. *Declare variables as pointers and allocate memory in GPU.* For allocating memory we use `cudaMalloc()`.
3. *Copy data from CPU to GPU memory.* To copy the data we use `cudaMemcpy()`.
4. *Call GPU kernel.*
5. *Copy results from GPU to CPU memory.* Again for this we use `cudaMemcpy()`.
6. *Use results on CPU.* After getting back results we can work on CPU as we do for standard C.

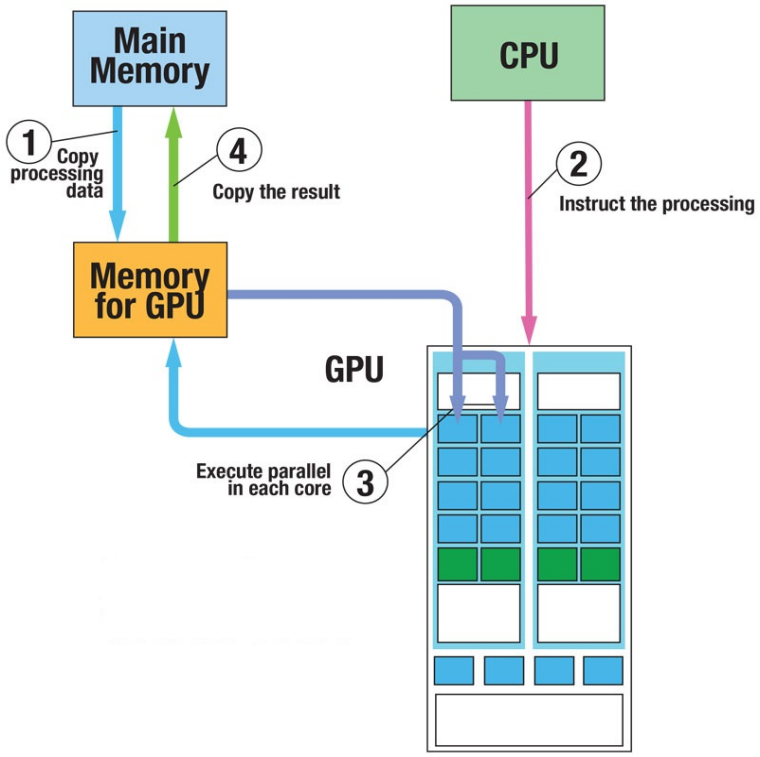


Figure 4: Programming flow on CUDA

4.2.1 Passing Parameters

We can pass parameters to a kernel as we would do with any C function. We need to allocate memory to do anything useful on a device, such as return value to the host. We allocate memory on the device by using `cudaMalloc()`. And we can free this memory by using `cudaFree()`. Now to pass the parameters to a kernel, the parameters should be device variables. Therefore we need to copy the data from CPU to GPU on GPU variables. After copying the data we can pass this variable to a kernel and use this as local variable to the kernel.

4.2.2 CUDA Functions Declarations

- `__device__ void kernel()`. This type of kernel which starts with `__device__` will be executed on device and can only be called from the device.

- `__global__ void kernel()`. This type of kernel which starts with `__host__` will be executed on device and can only be called from the host.
- `__host__ void kernel()`. This type of kernel which starts with `__device__` will be executed on host and can only be called from the host.

4.2.3 Threads, Blocks and Grids

- **Threads** - This is just an execution of a kernel with a given index. Each thread uses its index to access elements in array such that the collection of all threads cooperatively processes the entire data set.
- **Blocks** - This is a group of threads. There's not much we can say about the execution of threads within a block – they could execute parallely or serially and in no particular order. You can coordinate the threads, using the `syncthreads()` function that makes a thread stop at a certain point in the kernel until all the other threads in its block reach the same point.
- **Grid** - This is a group of blocks. There's no synchronization at all between the blocks.

We can call a kernel by `kernel<<<N,M>>>`. The first integer indicates the number of block to be launched. There can be maximum 256 or 512 blocks depending on the computer architecture. The second integer denotes the number of thread per block to be launched. Again, maximum thread can ne 512 or 1024.

4.3 Matrix Multiplication: Using CUDA

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4  #include<cuda.h>
5
6  __global__ void matmultiply(int *res, int *mat1, int *mat2, int N)
7  {
8      unsigned id = blockIdx.x*blockDim.x + threadIdx.x;
9
10     if (id >= N)
11         return;
12
13     for (int i = 0; i < N; i++)
14     {
15         int sum = 0;
16         for (int j = 0; j < N; j++)
17             sum += mat1[id*N+j] * mat2[j*N+id];
```

```
18
19          res[id*N+i] = sum;
20      }
21  }
22
23  int main()
24  {
25      clock_t start, end;
26      double gpu_time_used;
27
28      int N =800, i, j, k; //working till 863
29      int *mat1, *mat2, *res;
30
31      mat1 = (int *)malloc(N*N * sizeof(int));
32      mat2 = (int *)malloc(N*N * sizeof(int));
33      res = (int *)malloc(N*N * sizeof(int));
34
35      for (i = 0; i<N; i++) {
36          for (j = 0; j<N; j++) {
37              mat1[i*N+j] = i + 1;
38              mat2[i*N+j] = i + 2;
39          }
40      }
41
42      start = clock();
43      int numThreads = 256;  // number of threads
44
45      /* CUDA variables */
46      int *res_dev, *mat1_dev, *mat2_dev;
47      unsigned numbytes =N*N*sizeof(int);
48      res=(int*)malloc(numbytes);
49      cudaMalloc(&res_dev,numbytes);
50      cudaMalloc(&mat1_dev,numbytes);
51      cudaMalloc(&mat2_dev,numbytes);
52
53      /* copying variables from cpu to gpu */
54      cudaMemcpy(mat1_dev, mat1, numbytes,cudaMemcpyHostToDevice);
55      cudaMemcpy(mat2_dev, mat2, numbytes,cudaMemcpyHostToDevice);
56
57      /* calling kernel */
58      matmultiply<<<N/numThreads+1,numThreads>>>(res_dev, mat1_dev, mat2_dev↩
            ,N);
59
60      /* copying values from gpu to cpu */
61      cudaMemcpy(res,res_dev, numbytes,cudaMemcpyDeviceToHost);
62
63      end = clock();
```

```
64      gpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
65      printf("%f sec\n", gpu_time_used);
66
67      int *ans;
68      ans = (int *)malloc(N*N * sizeof(int));
69      for ( i = 0; i < N; i++)
70      {
71          for (j = 0; j < N; j++)
72          {
73              ans[i*N+j] = 0;
74              int temp = 0;
75              for (k = 0; k < N; k++)
76                  temp+= mat1[i*N+k] * mat2[k*N+j];
77
78              ans[i*N + j] = temp;
79          }
80      }
81      for (i = 0; i < N; i++)
82      {
83          for (j = 0; j < N; j++)
84          {
85              if (ans[i*N + j] != res[i*N + j])
86              {
87                  printf("error");
88                  break;
89              }
90          }
91
92      }
93
94      free(mat1);
95      free(mat2);
96      free(res);
97      free(ans);
98      cudaFree(mat1_dev);
99      cudaFree(mat2_dev);
100     cudaFree(res_dev);
101
102     return 0;
103 }
```

After allocating memory to matrices in CPU, we are allocating memories in GPU for GPU matrices. After that we are copying the data from CPU to GPU. Then we are calling the kernel for matrix multiplication. Inside the kernel we are generating thread id. This id will be for each row of the first matrix. We are using a simple for loop to do the multiplication, and store the

result. After computation, we are copying back the data from GPU to CPU.

Table 4: Time taken to execute the code in seconds

Dimension	Serial	Using OpenMP	Using CUDA
100 X 100	0.015	0.000	0.107
200 X 200	0.062	0.031	0.103
400 X 400	0.437	0.233	0.183
600 X 600	1.544	0.787	0.365
800 X 800	4.721	1.868	0.735
1000 X 1000	14.221	3.647	1.175
1200 X 1200	61.559	18.959	2.183
1400 X 1400	106.014	30.560	3.278
1600 X 1600	171.507	45.171	4.545
1800 X 1800	251.937	73.799	6.768
2000 X 2000	361.892	94.842	7.952

5 Working With CUDA

After completing conversion of Python code into C, we needed to parallelize the code to gain speed. I used CUDA 8.0 and Visual Studio 2015 to do this.

5.1 C99- C programming language standard

Microsoft Visual Compiler is not a C99 compiler. It still follows C90 standard and does not support many things which C99 offers. For example VS does not support variable length array. For this reason I had to define many variables as constant.

5.2 Generating ii values

```
for (int i3 = 0; i3<N; i3++) {
    if (status[i3] == 0) {
        ii[nii] = i3;
        nii++;
    }
}
```

This loop stores the indices of status, wherever it is 0 in ii. status 0 means the ray is still

active ans has not reached the submarine or bottom of the sea yet. As we can see the ii depends on the nii values which is very hard to parallelize in CUDA. But, somehow we need to get ii values in the device memory and copying from host to device is not an option as it consumes a lot of time. Therefore, what I was doing is running this loop sequentially in CUDA. Later on , I removed the ii values and in place of that I ran all the for loop till N and wherever status is 0 loop is doing the task, else not. This approach is good as it increased the speed up.

5.3 Generating Random Numbers

CUDA has it's inbuilt function **curand()** to generate random numbers. The same function can be utilised to generate random numbers with uniform distribution or Gaussian distribution.

```
1  /* this GPU kernel function is used to initialize the random states */
2  __global__ void init(unsigned int seed, curandState_t* states) {
3
4      /* we have to initialize the state */
5      curand_init(seed, blockIdx.x*blockDim.x + threadIdx.x, 0, &states[↩
           blockIdx.x*blockDim.x + threadIdx.x]);
6  }
7
8  /* this GPU kernel takes an array of states, and an array of ints, and ↩
       puts a random int into each */
9  __global__ void randoms(curandState_t* states, double* phi0_dev, int* ii ,↩
       int* nii, double* u_dev) {
10     /* curand works like rand - except that it takes a state as a ↩
           parameter */
11     unsigned id = blockIdx.x*blockDim.x + threadIdx.x;
12     if (id >= *nii)
13         return;
14     phi0_dev[ii[id]] = curand_uniform(&states[id]) * 2 * PI;
15     u_dev[id] = curand_uniform(&states[id]);
16 }
```

The first kernel initializes *states* which is required to generate different random numbers each time. In the second kernel we are using **curand_uniform()** to generate random numbers between 0 and 1 uniformly. The first kernel is need to get initialized only once. Then we can execute the second kernel to get different random numbers.

5.4 Device version of Spline and Splint Function

```
1  void spline(double *x, double *y, int a, double yp1, double ypn, double *↩
       y2) {
2      int i, k;
```

```
    double p, qn, sig, un, *u;
    u = (double*)malloc((a + 1) * sizeof(double));

    x--; y--; y2--;
    if (yp1 > 0.99e30)
        y2[1] = u[1] = 0.0;
    else {
        y2[1] = -0.5;
        u[1] = (3.0 / (x[2] - x[1]))*((y[2] - y[1]) / (x[2] - x[1]) - yp1);
    }
    for (i = 2; i <= a - 1; i++) {
        sig = (x[i] - x[i - 1]) / (x[i + 1] - x[i - 1]);
        p = sig*y2[i - 1] + 2.0;
        y2[i] = (sig - 1.0) / p;
        u[i] = (y[i + 1] - y[i]) / (x[i + 1] - x[i]) - (y[i] - y[i - 1]) /
            (x[i] - x[i - 1]);
        u[i] = (6.0*u[i] / (x[i + 1] - x[i - 1]) - sig*u[i - 1]) / p;
    }
    if (ypn > 0.99e30)
        qn = un = 0.0;
    else {
        qn = 0.5;
        un = (3.0 / (x[a] - x[a - 1]))*(ypn - (y[a] - y[a - 1]) / (x[a] -
            x[a - 1]));
    }
    y2[a] = (un - qn*u[a - 1]) / (qn*y2[a - 1] + 1.0);
    for (k = a - 1; k >= 1; k--)
        y2[k] = y2[k] * y2[k + 1] + u[k];

    free(u);
}
```

```
__device__ void spline_device(double *x, double *y, int a, double yp1,
    double ypn, double *y2) {
    int i, k;
    double p, qn, sig, un;
    double u[nj];

    x--; y--; y2--;
    if (yp1 > 0.99e30)
        y2[1] = u[1] = 0.0;
    else {
        y2[1] = -0.5;
        u[1] = (3.0 / (x[2] - x[1]))*((y[2] - y[1]) / (x[2] - x[1]) - yp1);
```

```
            ;
    }
    for (i = 2; i <= a - 1; i++) {
        sig = (x[i] - x[i - 1]) / (x[i + 1] - x[i - 1]);
        p = sig*y2[i - 1] + 2.0;
        y2[i] = (sig - 1.0) / p;
        u[i] = (y[i + 1] - y[i]) / (x[i + 1] - x[i]) - (y[i] - y[i - 1]) /
            (x[i] - x[i - 1]);
        u[i] = (6.0*u[i] / (x[i + 1] - x[i - 1]) - sig*u[i - 1]) / p;
    }
    if (ypn > 0.99e30)
        qn = un = 0.0;
    else {
        qn = 0.5;
        un = (3.0 / (x[a] - x[a - 1]))*(ypn - (y[a] - y[a - 1]) / (x[a] -
            x[a - 1]));
    }
    y2[a] = (un - qn*u[a - 1]) / (qn*y2[a - 1] + 1.0);
    for (k = a - 1; k >= 1; k--)
        y2[k] = y2[k] * y2[k + 1] + u[k];
}
```

As we can compare the above code and see that both are similar. Only the number of threads and implementation should be taken care of. Rest of the things are easy. In similar way I have written splint function in the device which I have discussed in appendix.

5.5 Generating tj, theta0 and other values

For generating tj, theta0 and other values, I have made separate kernels for each of them. Inside the kernels same work is happening as it was in the CPU to generate them. Writing kernels for these values was little easy as I have to only take care of the division of threads and there performance. I have discussed in detail about them in the appendix section.

5.6 Display Driver Issue

By default Windows assumes that we are using our GPU for graphics processing. In these types of applications the GPU would calculates things very quickly. The time it takes from the CPU's requesting to GPU to do something to when the GPU gets back with the result is extremely small. The graphics driver resets itself if it detects that the GPU is hanging. Unfortunately it detects this by timing the GPU, if it takes more than 2 seconds to do any one task, it assumes there is been a problem. This is excellent for detecting problems in graphics. But, it is bad for CUDA since it's natural for computations to take far longer than 2 seconds.

To prevent this from happening we need to change the appropriate keys in the Windows Registry. The values which we need to change are for the following key:
HKEY_LOCAL_MACHINE/SYSTEM/CurrentControlSet/Control/Graphic
We need to add two Dword values to this subkey-
1) TdrLevel: Timeout Detection and Recovery Level
0: Off, don't detect timeout
3: Timed; This id default, it will use the value of TdrDelay
2) TdrDelay: Timeout Detection and Recovery Delay Number of Seconds; 2 default

5.7 Performance Analyzation in GPU

Table 5: Time taken for different number of loops in CUDA and to copy the variables

No. of loops	Total Time	Host to Device	Device to Host
100	3.648s	0.030s	0.006s
200	7.280s	0.029s	0.006s
500	18.152s	0.029s	0.007s
1000	36.231s	0.030s	0.007s
2000	72.299s	0.030s	0.008s
5000	218.231s	0.031s	0.008s
10002	450.096s	0.029s	0.009s

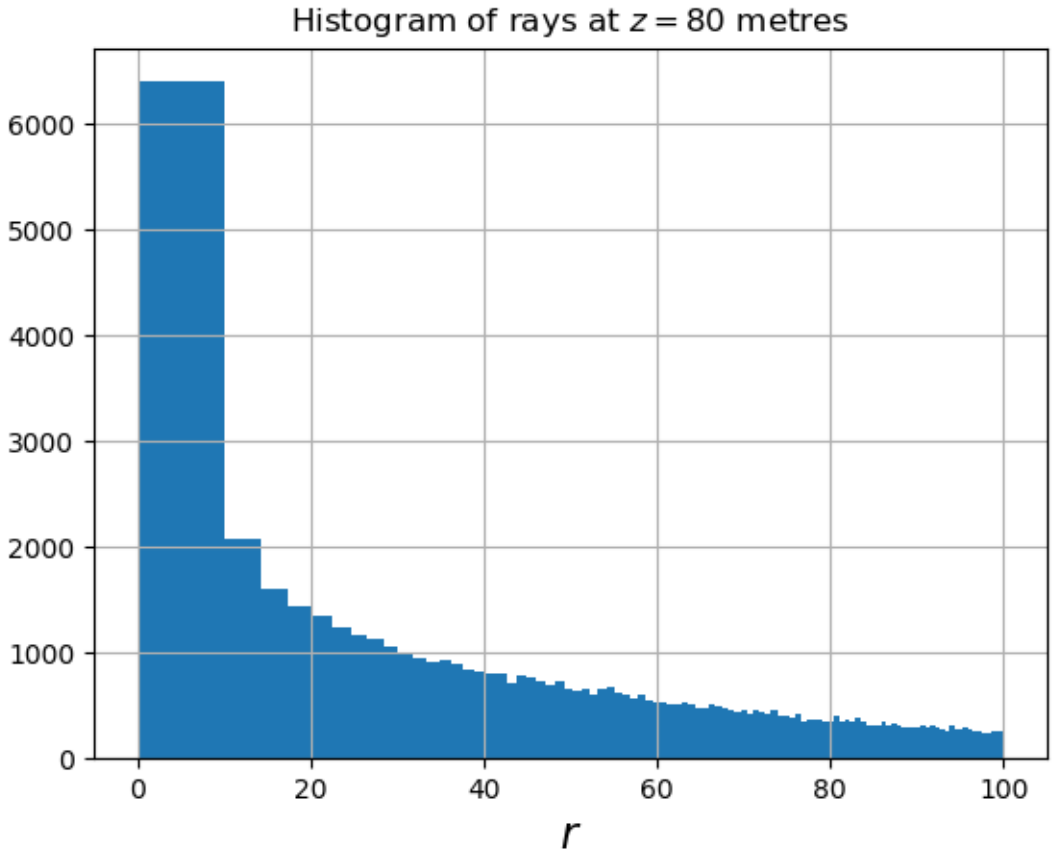
Table 6: Comparison between Python, C and CUDA

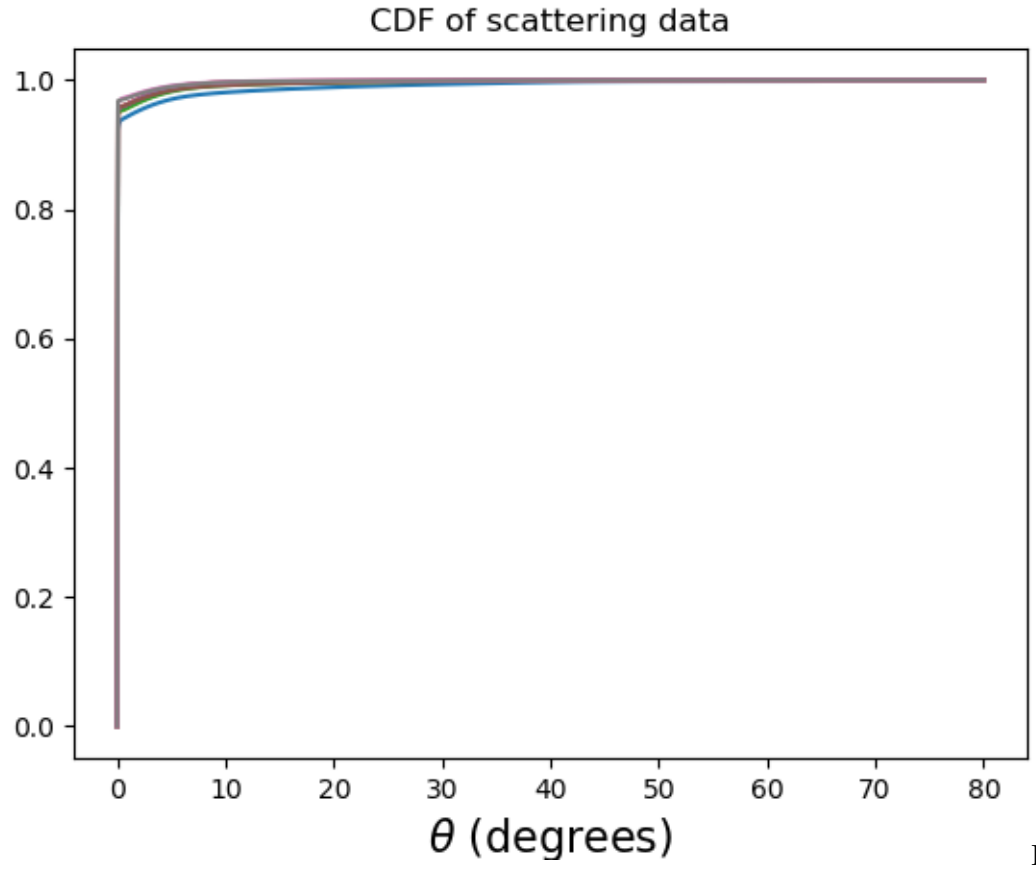
No. of loops	Python	C	CUDA
1	4s	0.4s	-
100	432s	45.63s	3.648s
10002	-	-	450.096s

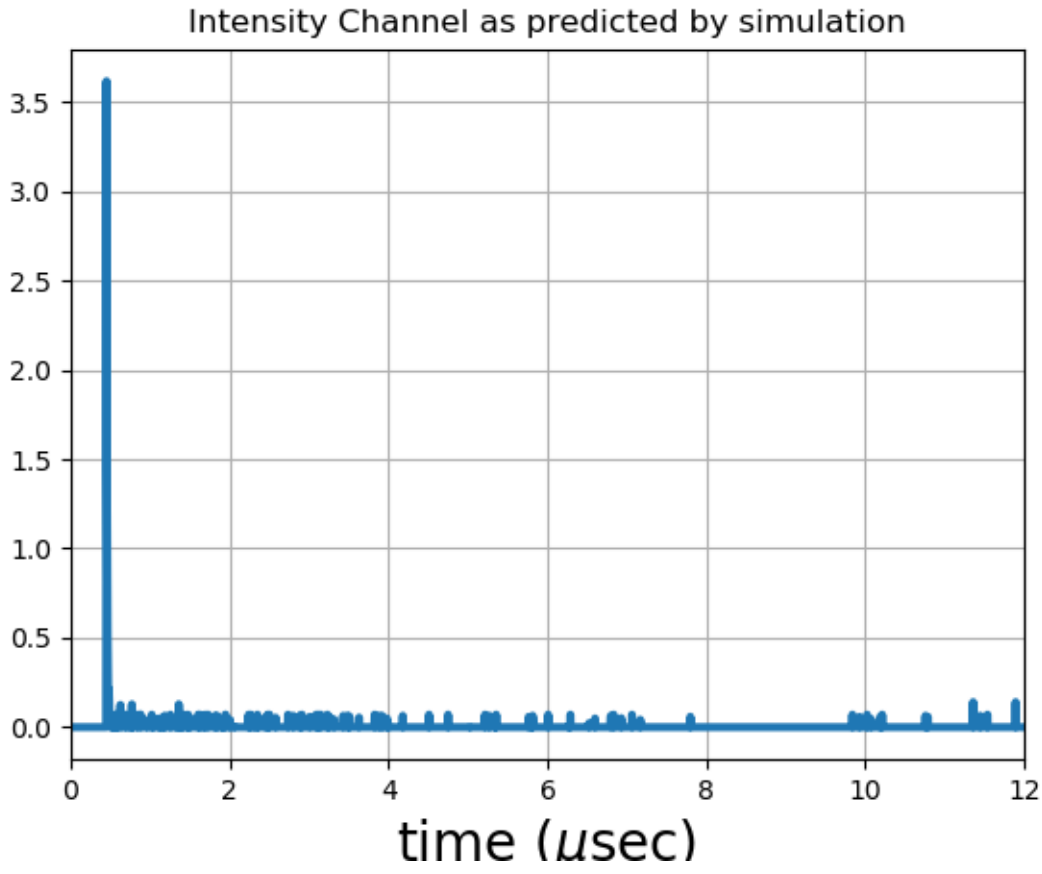
We can not say about individual loop in CUDA as they take different time because CPU and GPU work simultaneously. For 10002 Python will take more than 5 hours that is why I have not run the code in it. In C too it will take more than 1 hour.

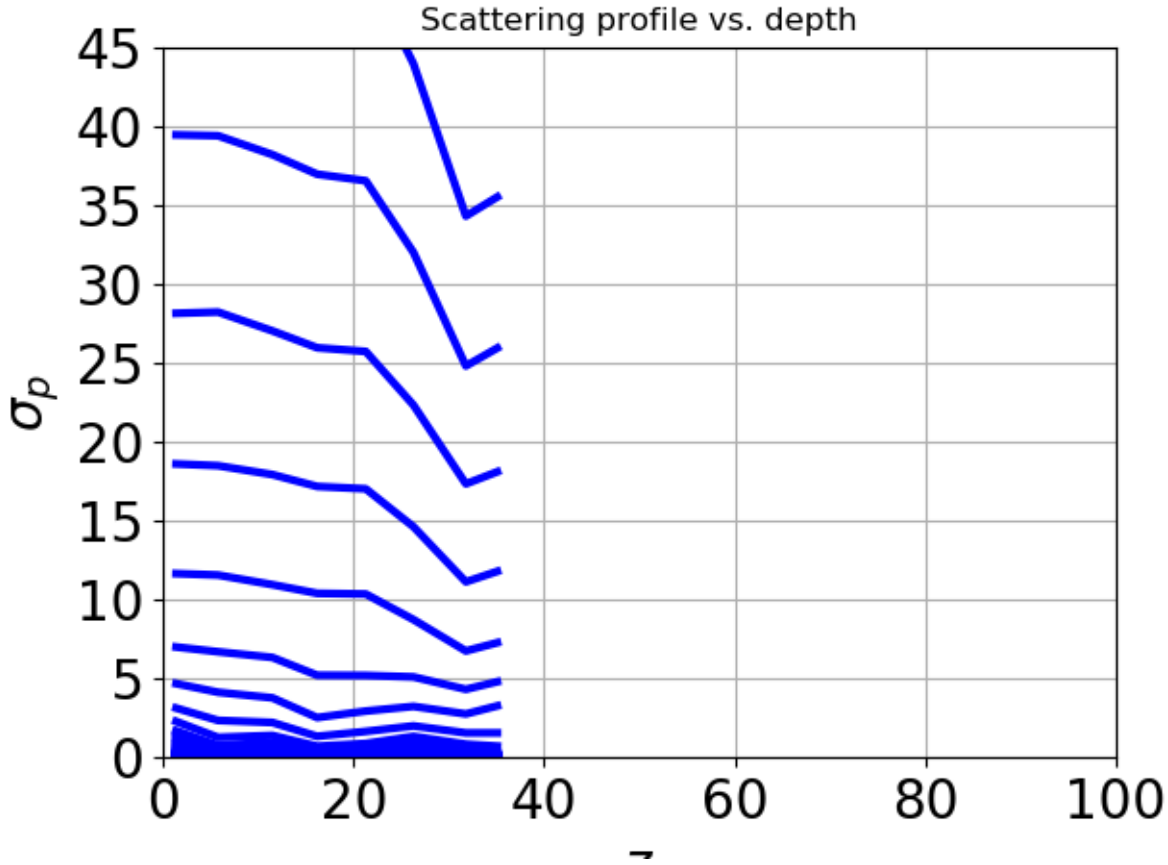
6 Plots

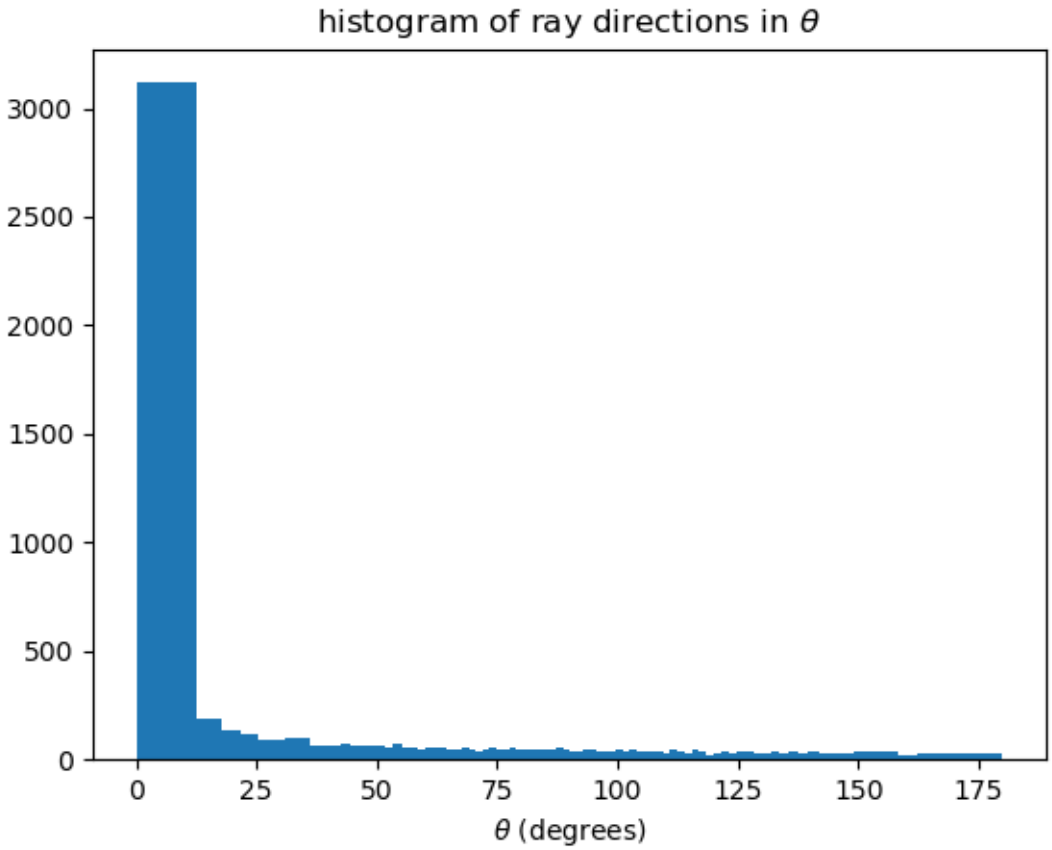
6.1 Plots Generated in C

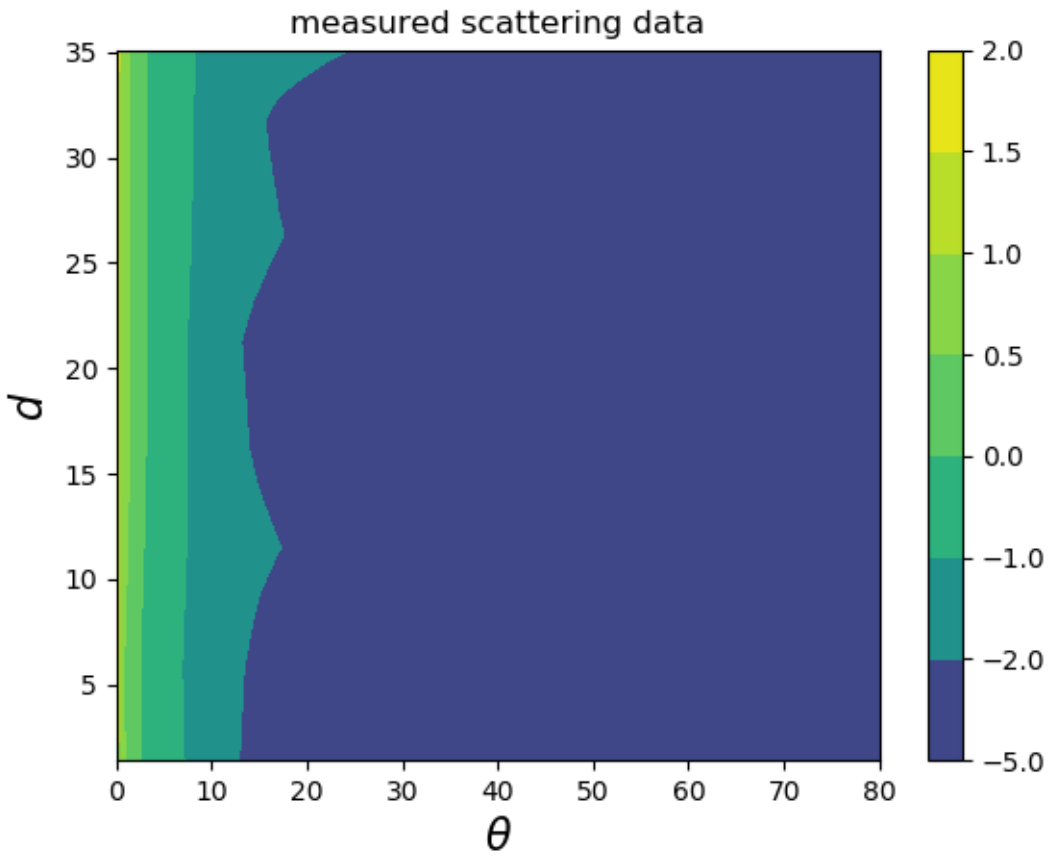


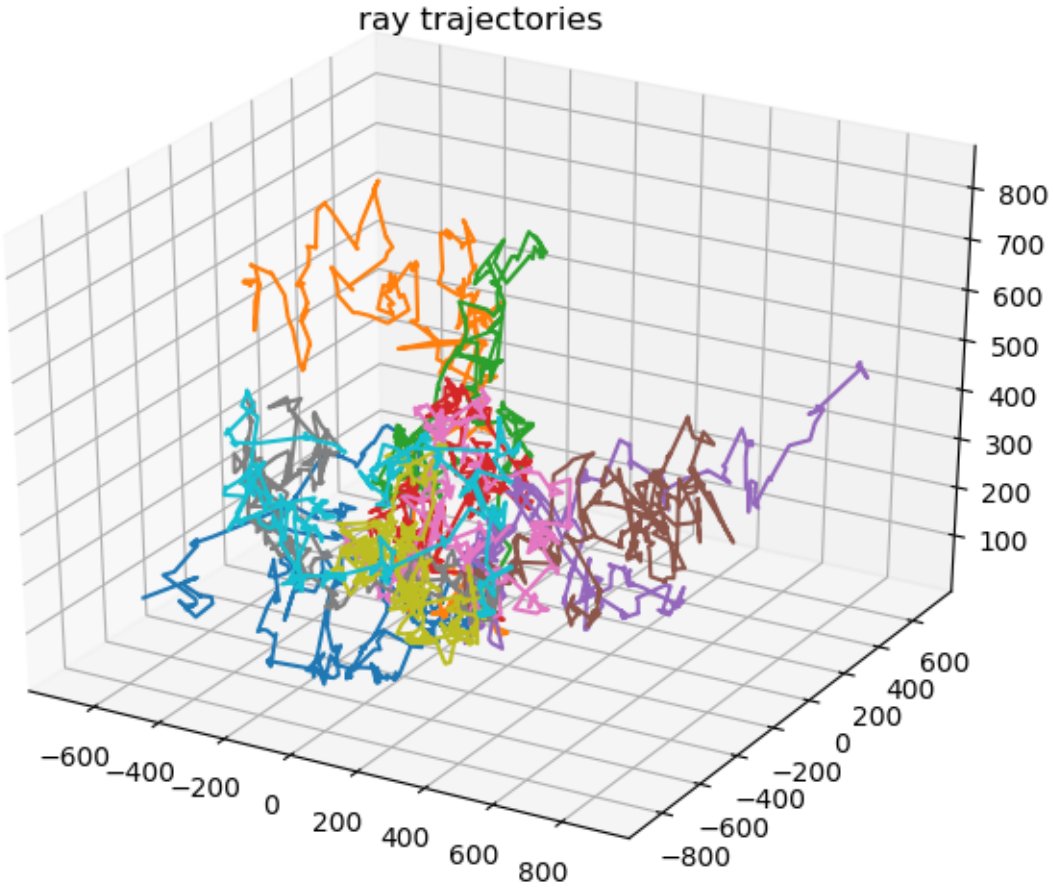


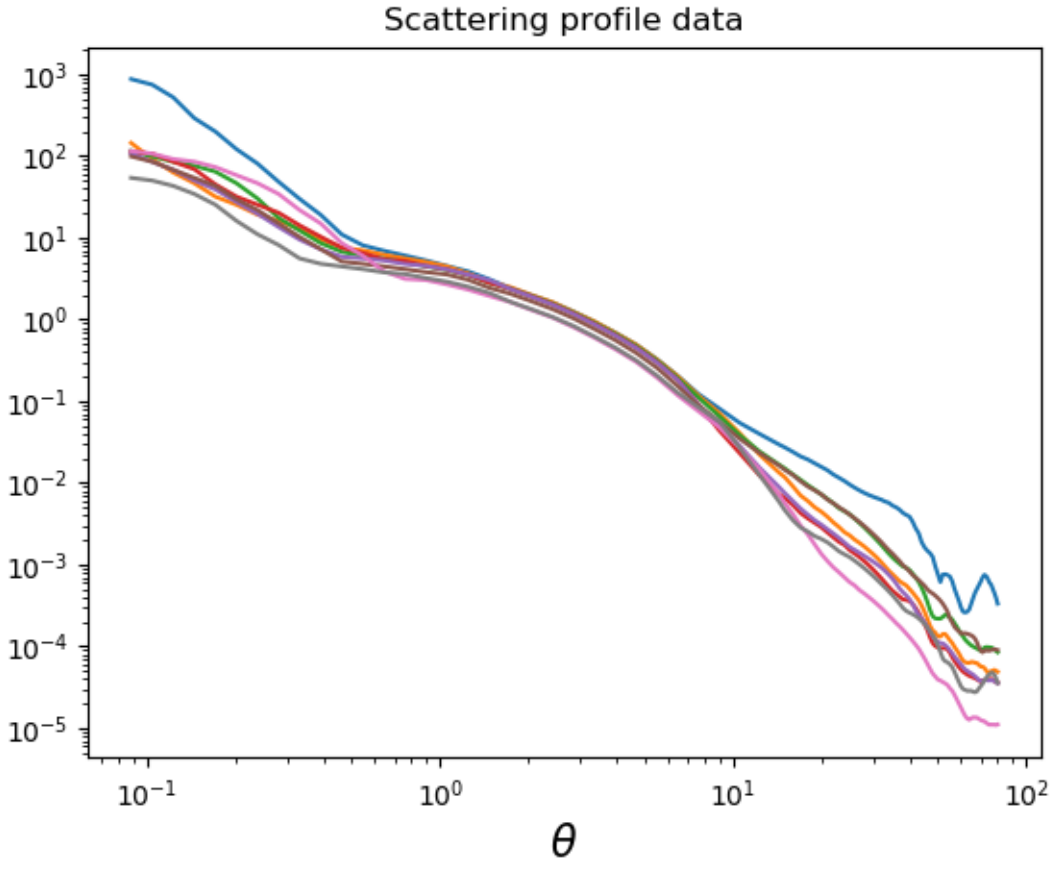

Intensity Channel as predicted by simulation
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
2
4
6
8
10
12
time (μsec)


Scattering profile vs. depth
45
40
35
30
25
20
15
10
5
0
σ_p
0
20
40
60
80
100


histogram of ray directions in θ
3000
2500
2000
1500
1000
500
0
0
25
50
75
100
125
150
175
θ (degrees)


measured scattering data
35
30
25
20
15
10
5
d
0
10
20
30
40
50
60
70
80
θ
2.0
1.5
1.0
0.5
0.0
−1.0
−2.0
−5.0


ray trajectories
800
700
600
500
400
300
200
100
600
400
200
0
−200
−400
−600
−800
−600
−400
−200
0
200
400
600
800

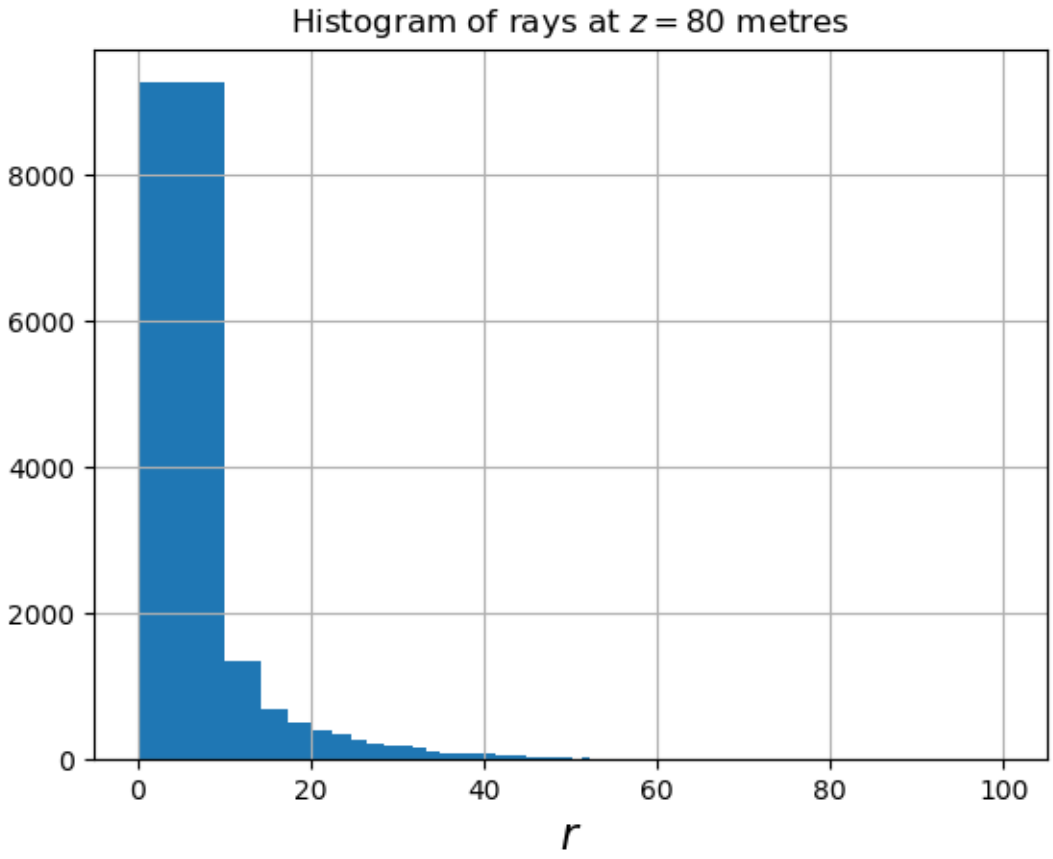

Scattering profile data
10^{3}
10^{2}
10^{1}
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-1}
10^{0}
10^{1}
10^{2}
θ

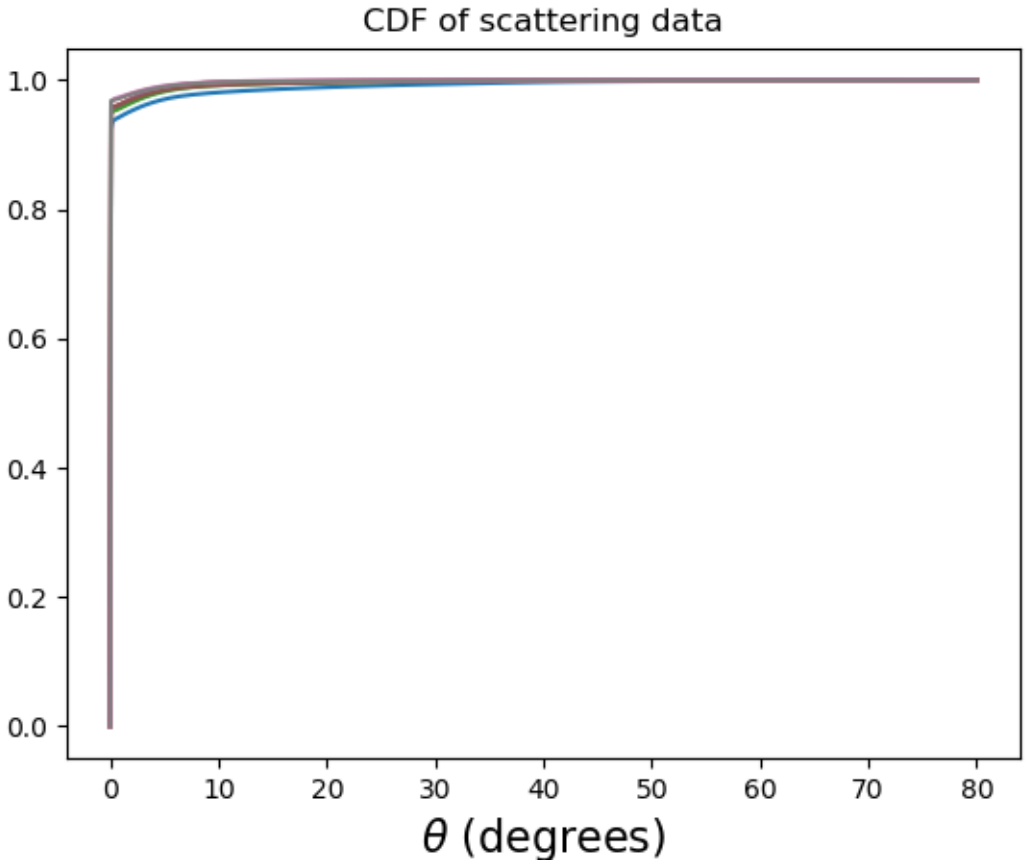
For the above graphs it gives constant extrapolation for splint() function. That is, if x-values are less than x1 then all the y-values will be y1. And if x-values are more than xn then all tn y-values will be tn. Following are the parameters used :
Number time steps for simulation, nt = 10002
Number of rays to track, N=100000

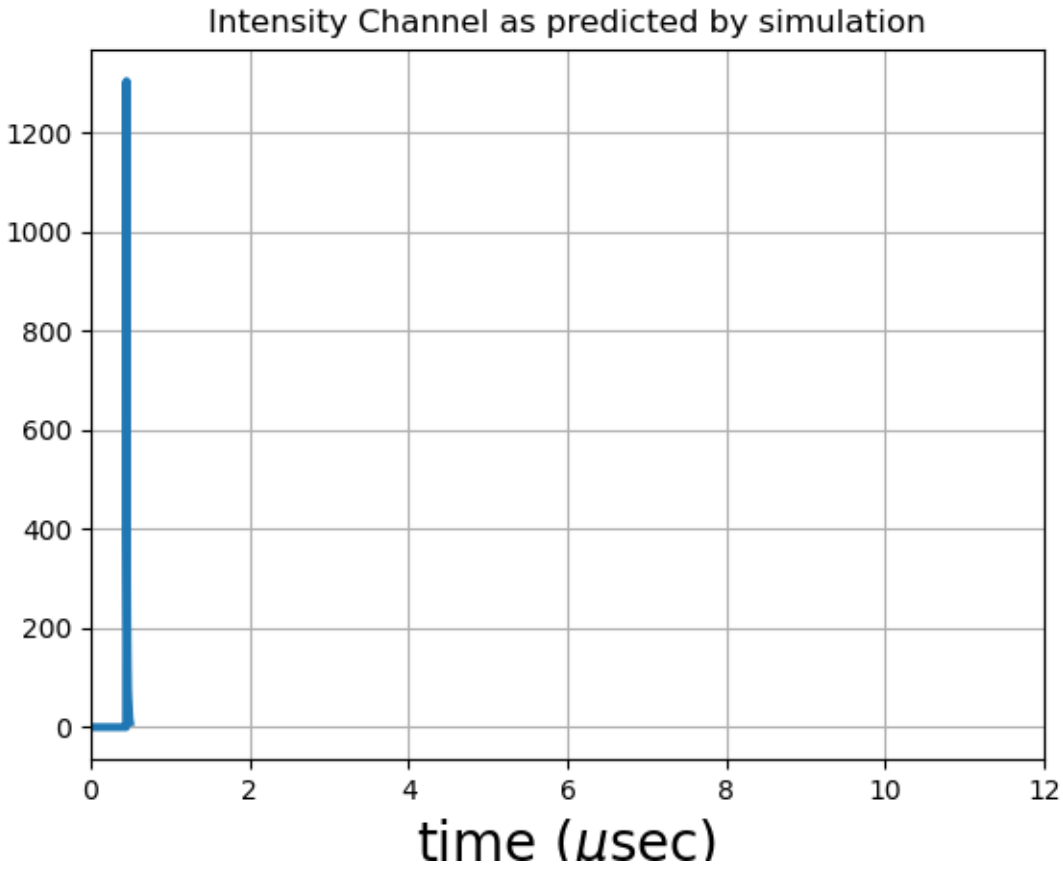
6.1.1 Output

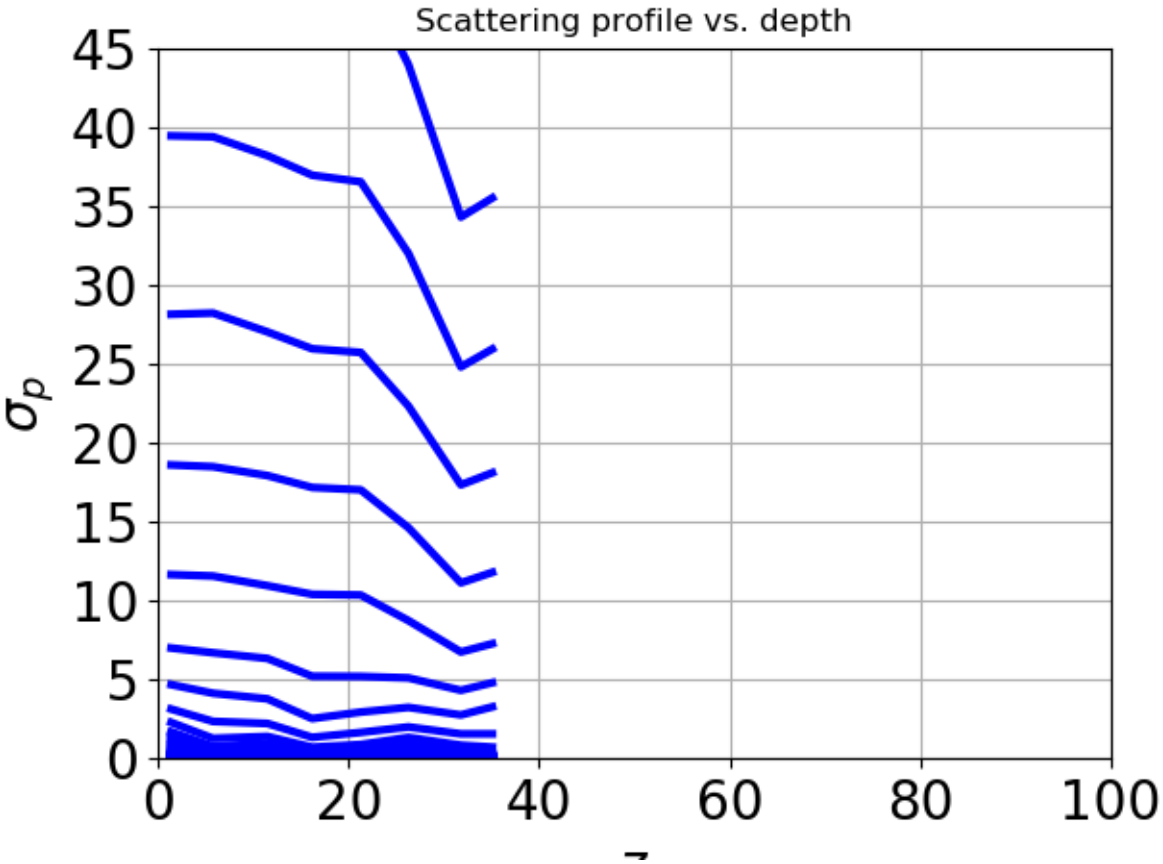
- 7244 rays reached out of 100000 to the submarine.
- Average time to reach the submarine for one ray is 857.15 time steps.
- Stdeviation of time to reach the submarine is 1601.58 time steps

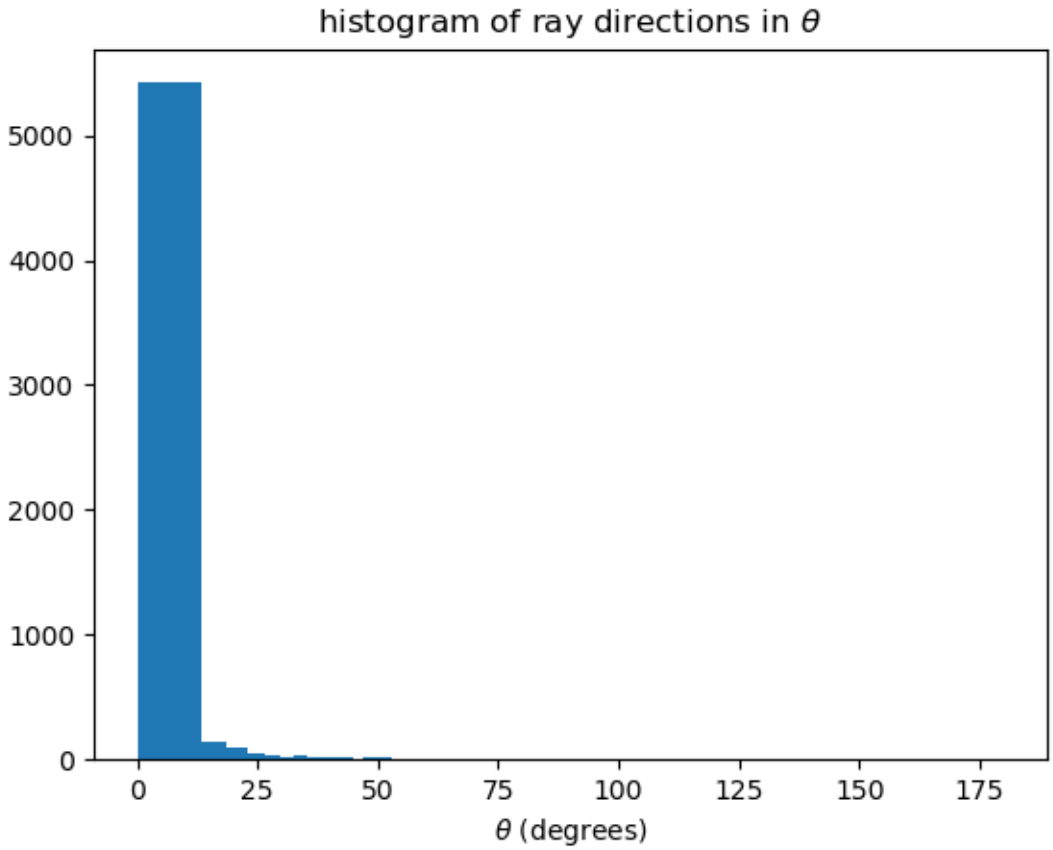
6.2 Plots Generated in Python

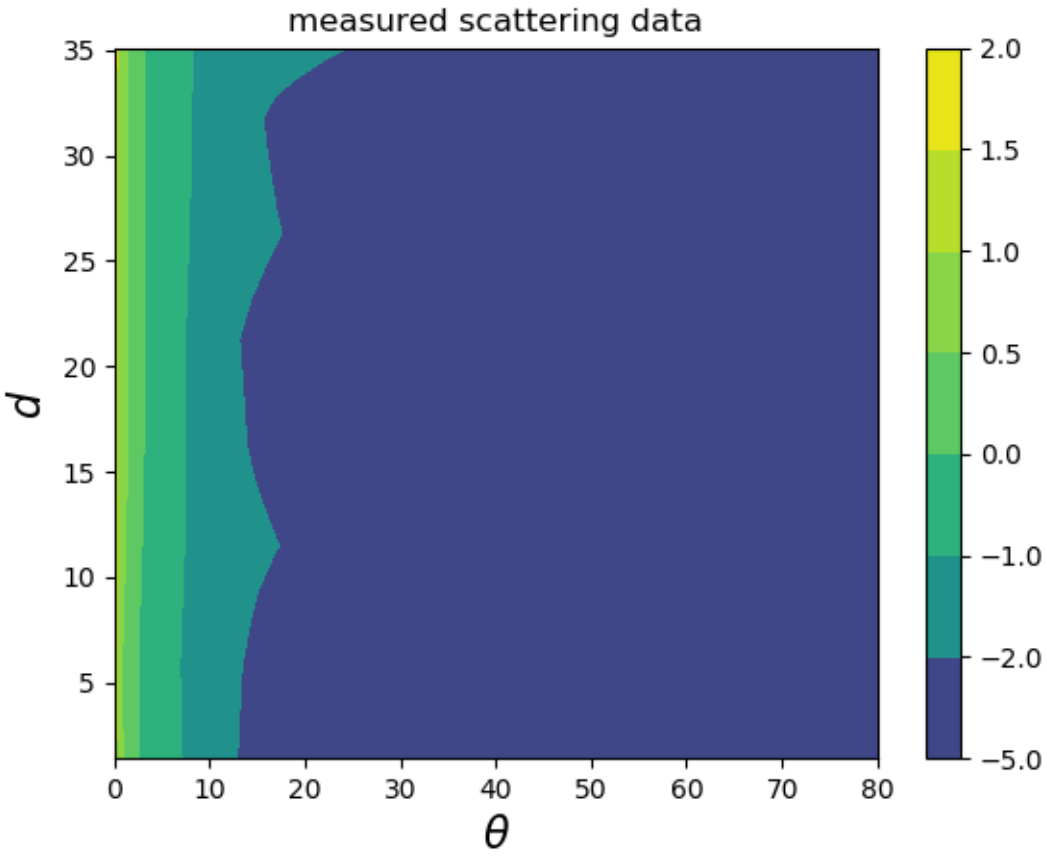


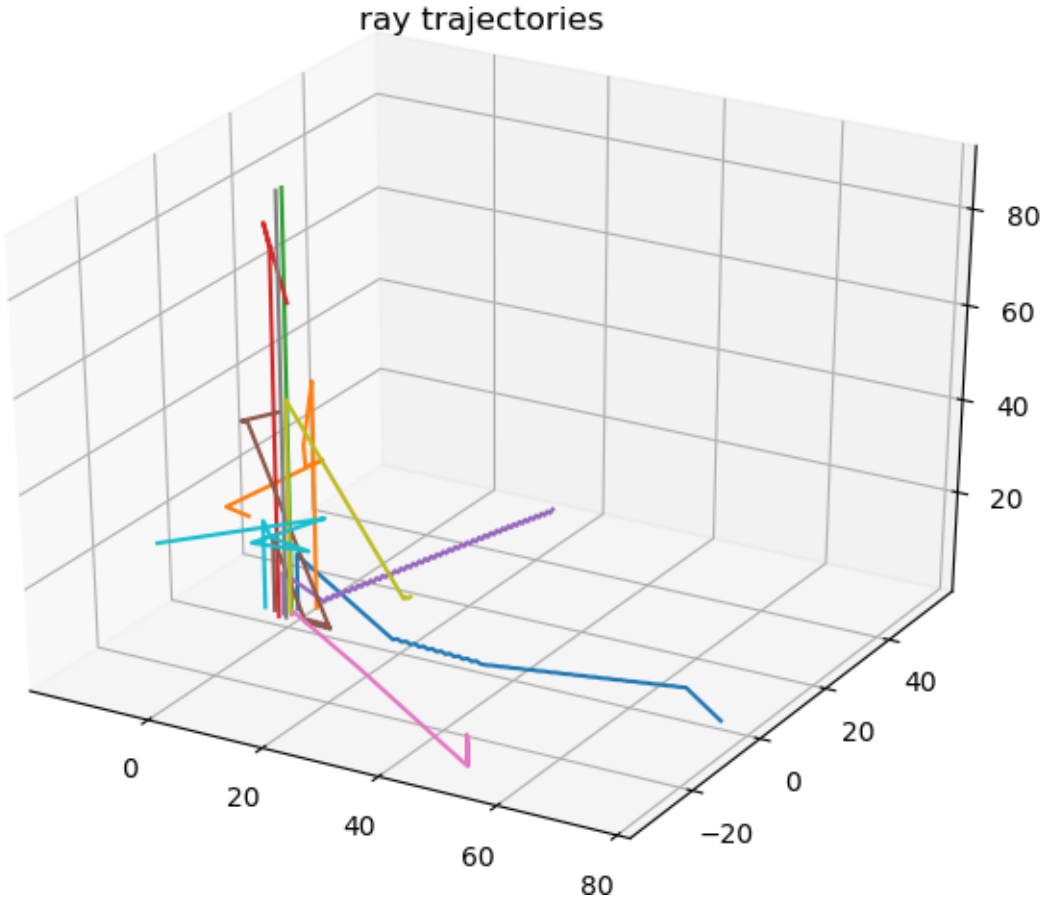


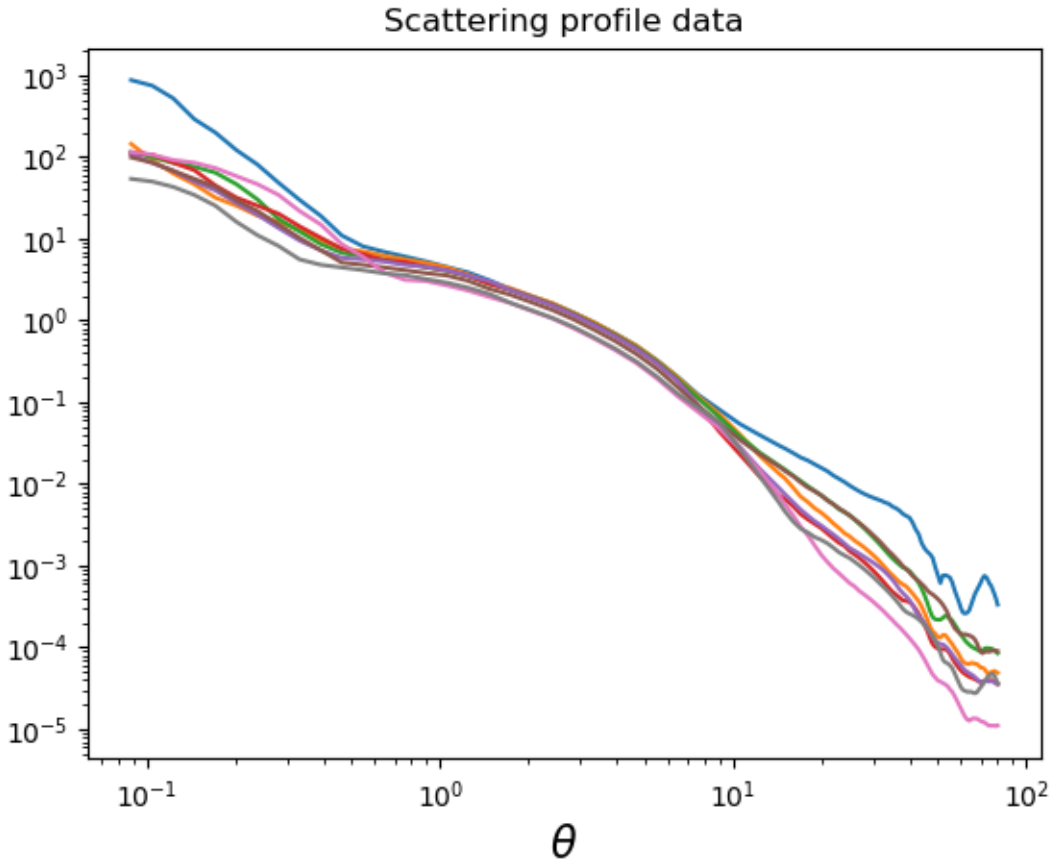

Intensity Channel as predicted by simulation
1200
1000
800
600
400
200
0
0
2
4
6
8
10
12
time (μsec)


Scattering profile vs. depth
45
40
35
30
25
20
15
10
5
0
σ_p
0
20
40
60
80
100


histogram of ray directions in θ
5000
4000
3000
2000
1000
0
0
25
50
75
100
125
150
175
θ (degrees)


measured scattering data
35
30
25
20
15
10
5
d
0
10
20
30
40
50
60
70
80
θ
2.0
1.5
1.0
0.5
0.0
−1.0
−2.0
−5.0


ray trajectories
80
60
40
20
0
20
40
60
80
40
20
0
−20

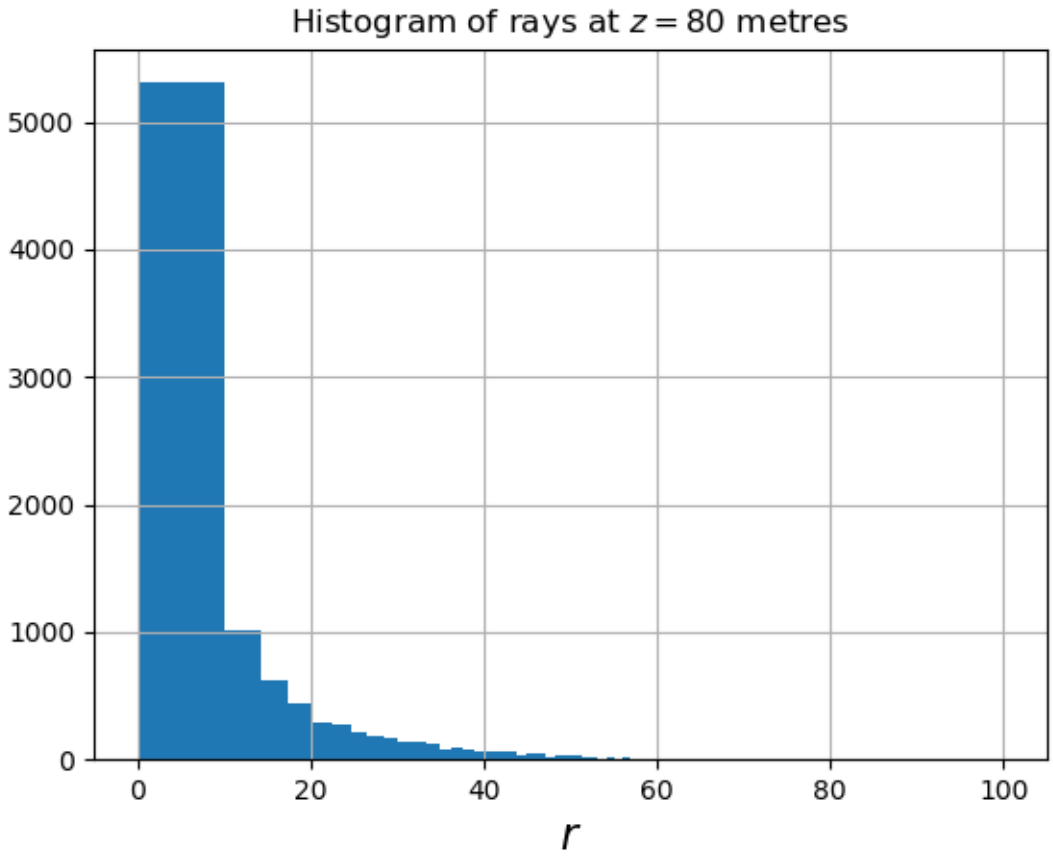

Scattering profile data
10^{3}
10^{2}
10^{1}
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-1}
10^{0}
10^{1}
10^{2}
θ

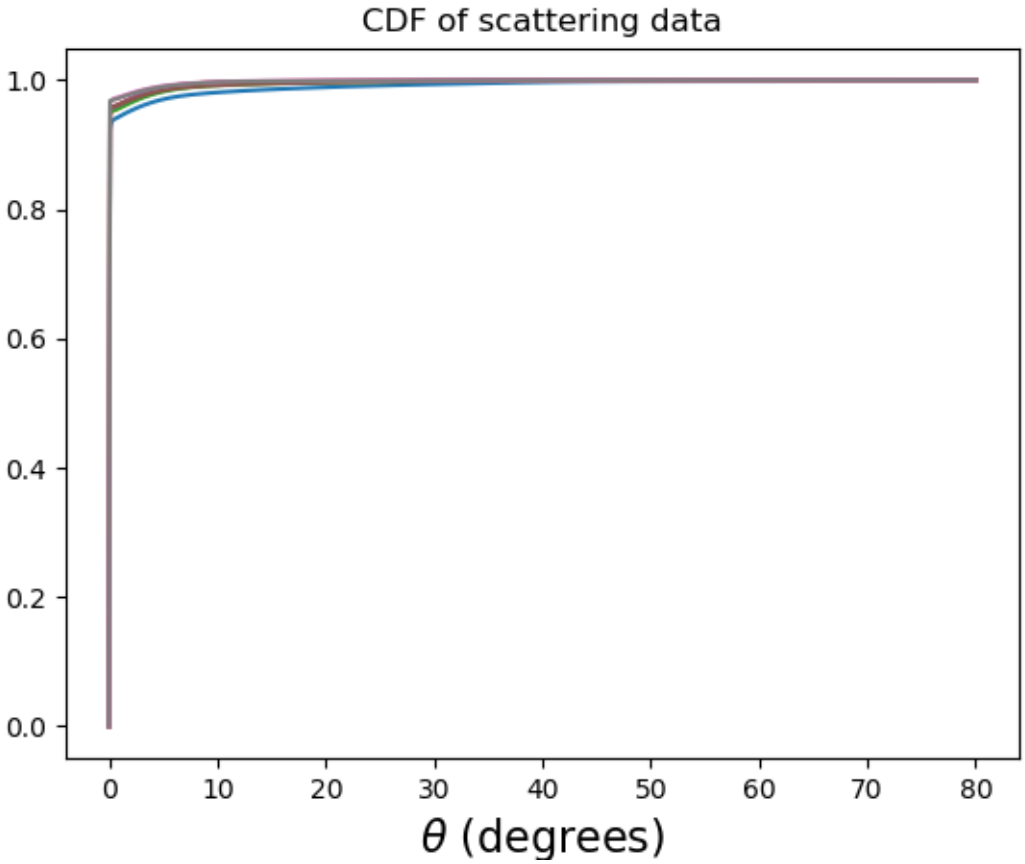
Following are the parameters used :
Number time steps for simulation, nt = 100
Number of rays to track, N=100000

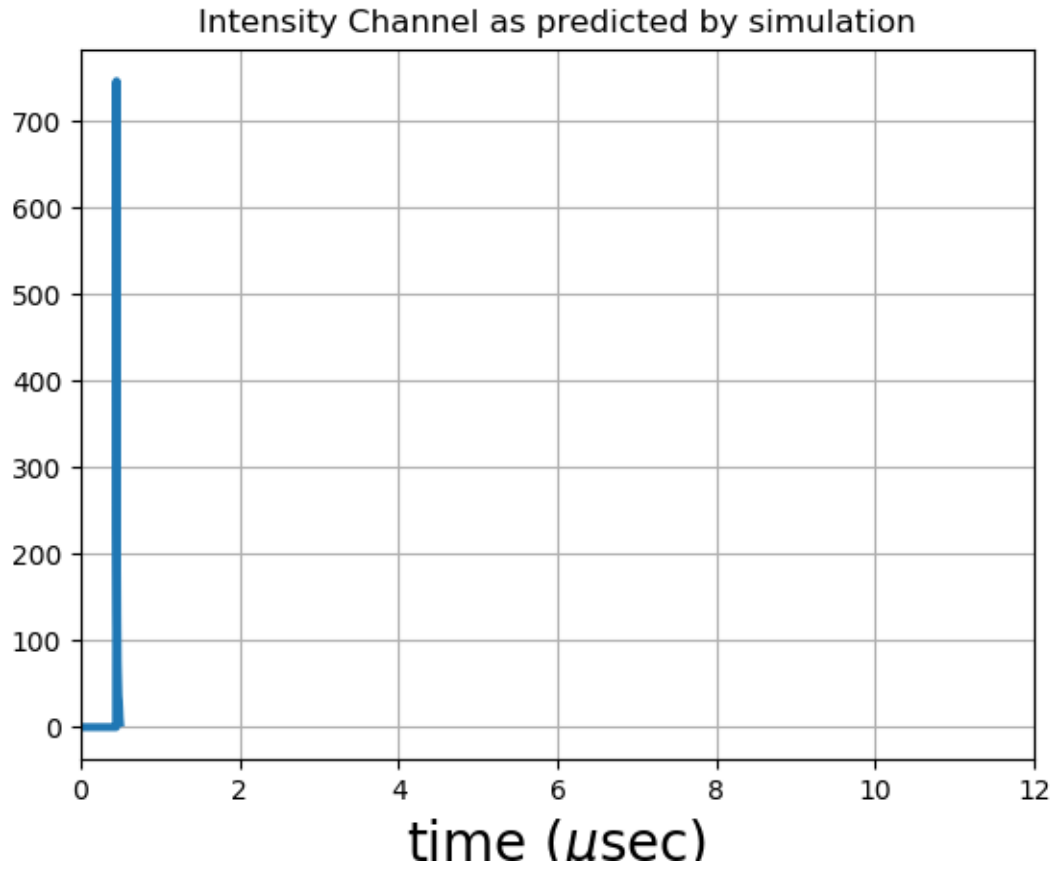
6.2.1 Output

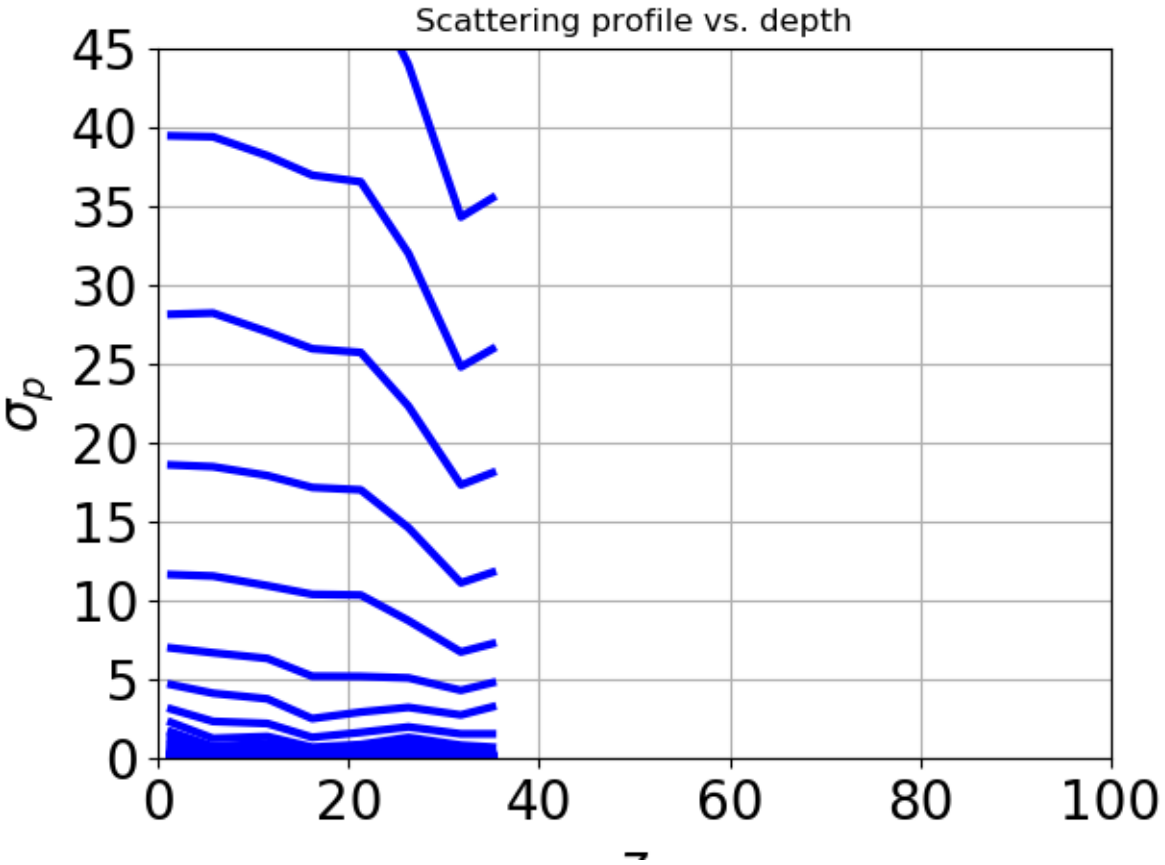
- 5993 rays reached out of 100000 to the submarine.
- Average time to reach the submarine for one ray is 91.07 time steps.
- Stdeviation of time to reach the submarine is 1.75 time steps

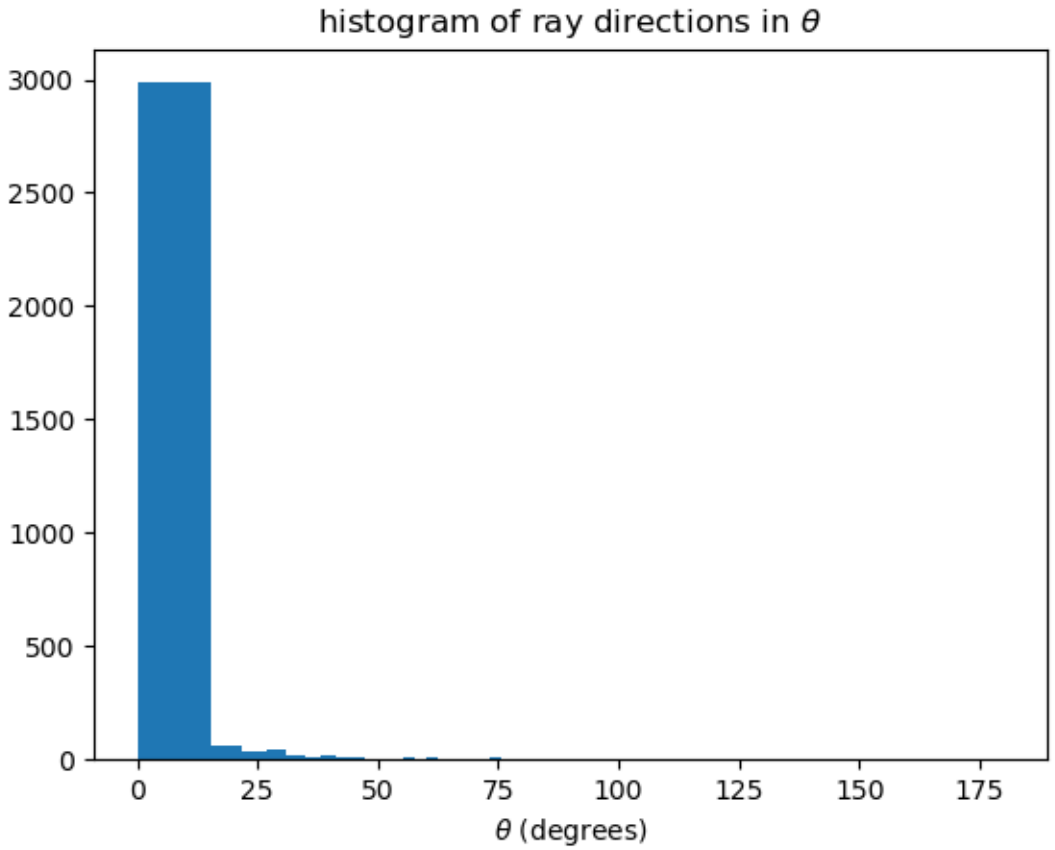
6.3 Plots Generated in CUDA

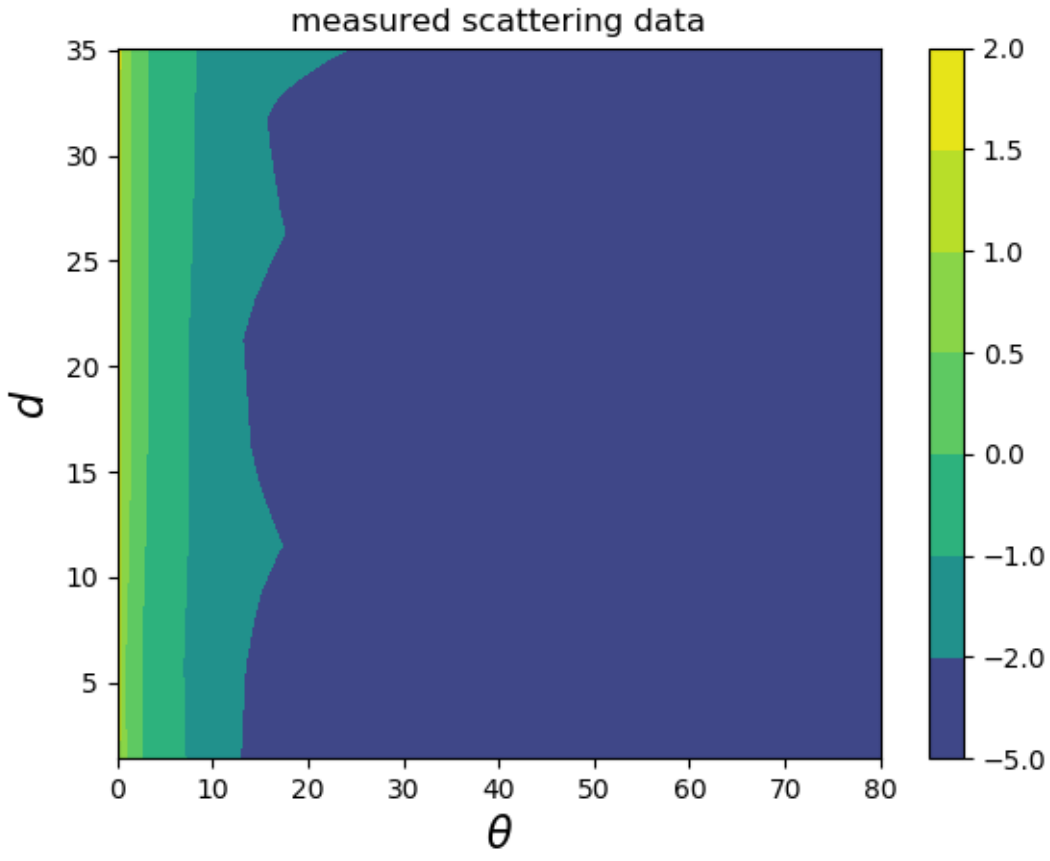


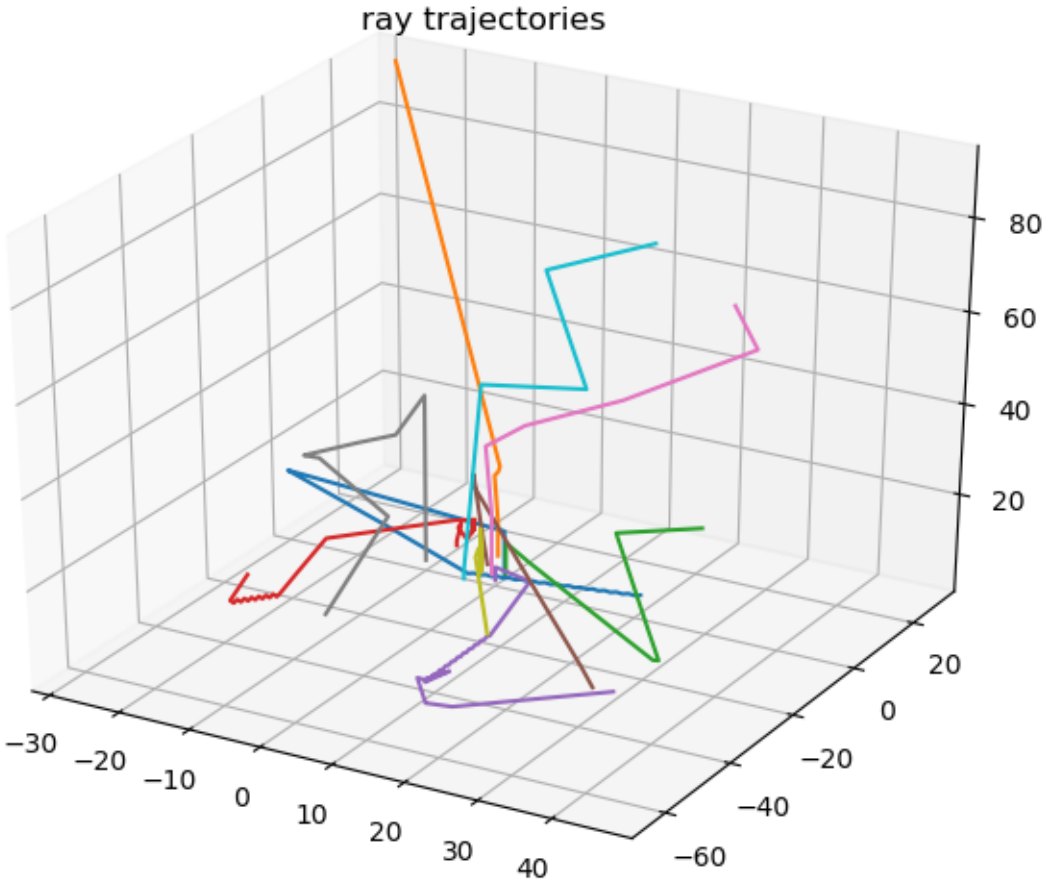


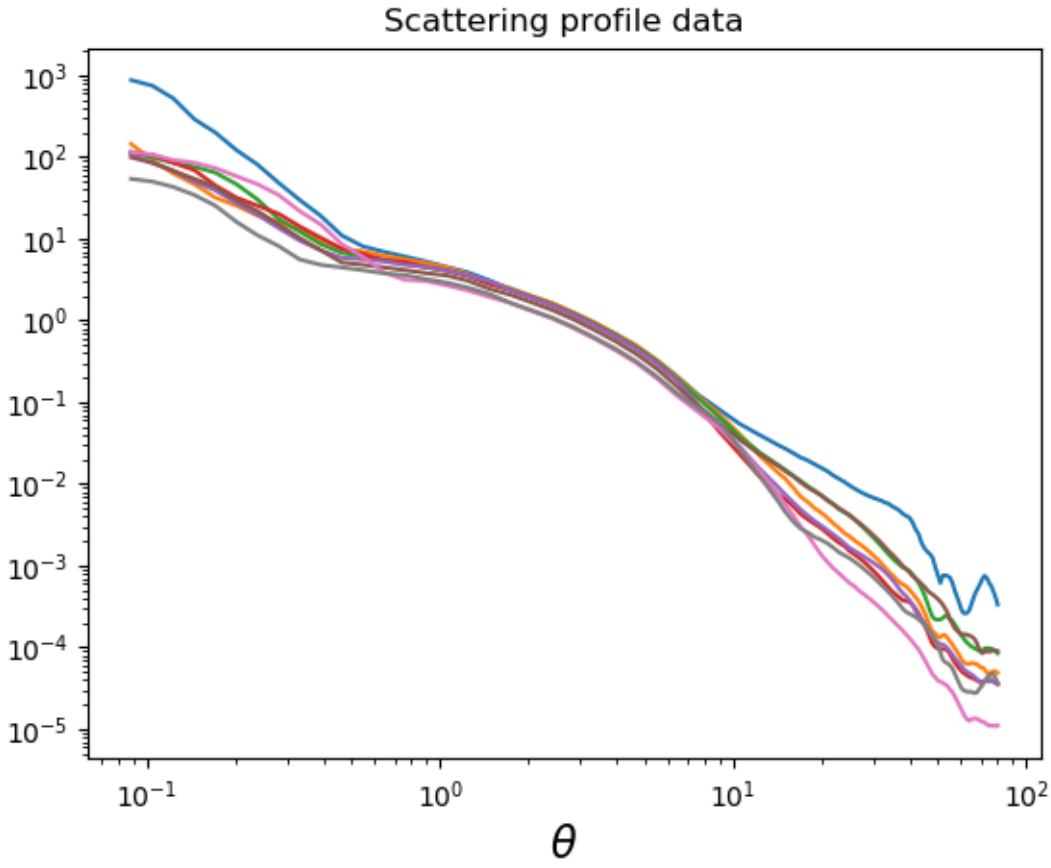

Intensity Channel as predicted by simulation
700
600
500
400
300
200
100
0
0
2
4
6
8
10
12
time (μsec)


Scattering profile vs. depth
45
40
35
30
25
20
15
10
5
0
σ_p
0
20
40
60
80
100


histogram of ray directions in θ
3000
2500
2000
1500
1000
500
0
0
25
50
75
100
125
150
175
θ (degrees)


measured scattering data
35
30
25
20
15
10
5
d
0
10
20
30
40
50
60
70
80
θ
2.0
1.5
1.0
0.5
0.0
−1.0
−2.0
−5.0


ray trajectories
80
60
40
20
−30
−20
−10
0
10
20
30
40
20
0
−20
−40
−60


Scattering profile data
10^{3}
10^{2}
10^{1}
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-1}
10^{0}
10^{1}
10^{2}
θ

Following are the parameters used :
Number time steps for simulation, nt = 100
Number of rays to track, N=100000

6.3.1 Output

- 3255 rays reached out of 100000 to the submarine.
- Average time to reach the submarine for one ray is 91.22 time steps.
- Stdeviation of time to reach the submarine is 1.89 time steps

7 References:

[1] Dr. Yogish Sabharwa. (2017, June 8). *Introduction to Parallel Programming in OpenMp* [Online]. Available: http://nptel.ac.in/courses/106102163/

[2] Dr. Subodh Kumar.(2013, November 14). *Parallel Computing* [Online]. Available: http://nptel.ac.in/courses/106102114/

[3] Jason Sanders and Edward Kandrot. *CUDA BY EXAMPLE*

[4] University of Mary Washington. *CUDA Random Numbers* [Online]. Available: http://cs.umw.edu/ finlayson/class/fall16/cpsc425/notes/cuda-random.html

[5] Dartmouth College. *What is OpenMP?* [Online]. Available: https://www.dartmouth.edu/ rc/classes/intro_openmp/

[6] KEVIN KREWELL. (2009, Dec 16). *What's the Difference Between a CPU and a GPU?* [Online]. Available: https://blogs.nvidia.com/blog/2009/12/16/whats-the-difference-between-a-cpu-and-a-gpu/

[7] Rupesh Nasre. *CS6023 GPU Programming Course Materials* [Online]. Available: http://www.cse.iitm.ac.in/ rupesh/teaching/gpu/aug17/

[8] What's a Creel?. *Cuda Tutorial 3: Display Driver has Stopped Working and has Recovered* [Online]. Available: https://www.youtube.com/watch?v=8NtHDkUoN98

Appendix

A Ray Tracing Code

```
1  #include <stdio.h>
2  #include <cuda.h>
3  #include <stdlib.h>
4  #include <string.h>
5  #include <math.h>
6  #include <time.h>
7
8  /* We need these includes for CUDA's random number stuff */
9  #include <curand.h>
10 #include <curand_kernel.h>
11
12 /* We need to to define these parameters as constants
13 because Visual Studio does not support C99 compiler */
14 #define PI 3.14159265358979323846
15 #define n_att 12
16 #define Nc 99
17 #define ni 99
```

```
#define nj 9
#define nt 10002
#define n 98

/* this GPU kernel function is used to initialize the random states */
__global__ void init(unsigned int seed, curandState_t* states) {

    /* we have to initialize the state */
    curand_init(seed, blockIdx.x*blockDim.x + threadIdx.x, 0, &states[
        blockIdx.x*blockDim.x + threadIdx.x]);
}

/* this GPU kernel takes an array of states, and an array of ints, and
    puts a random int into each */
__global__ void randoms(curandState_t* states, double* phi0_dev, int*
    status, double* u_dev, int N) {
    /* curand works like rand - except that it takes a state as a
        parameter */
    unsigned id = blockIdx.x*blockDim.x + threadIdx.x;
    if (id >= N)
        return;
    if (status[id] == 0) {
        phi0_dev[id] = curand_uniform(&states[id]) * 2 * PI;
        u_dev[id] = curand_uniform(&states[id]);
    }
}
```

I have declared required header files. We need to define parameters like Nc, ni as constants because Visual Studio does not support C99 compiler. Therefore if an array is defined based on the value of some integer variable that variable must be declared as constant. Also we can pass a 2-D array to a function by using an integer variable as it's size. To avoid these problems in Visual Studio, we need to define these parameters as constants. After defining them as constants, we can use them to define an array.

`__global__void init()` is a CUDA kernel to intialise the states, which we require to generate different random numbers each time we run the code.

`__global__ void randoms()` takes *states* as one of the argument and generate random numbers then stores them in the array which has been passed to the kernel.

```
__global__ void calc_traj (double *traj, double *pos, int ntraj, int i,
    int N) {
    unsigned id = blockIdx.x*blockDim.x + threadIdx.x;
    int a = id/ntraj;
    int b = id%ntraj;
```

```
    if (id>= 3*ntraj) {
        return;
    }

    traj[a*ntraj*nt + b*nt + i] = pos[a*N + b];
}
```

This device function is to generatetraj values. In traj, we store the trajectory of each ray by storing it's current position.

```

__device__ double splint_device(double *xa, double *ya, double *y2a, int c↩
    , double x) {
    int klo, khi, k;
    double h, b, a, y;

    if (x<xa[0])
        return ya[0];
    else if (x>xa[c - 1])
        return ya[c - 1];

    xa--; ya--; y2a--;
    /*if (c == Nc - 1) {
        for (int i = 0; i < 10; i++)
        printf(" from xa= %lf ya= %lf y2a= %lf \n", xa[i], ya[i], y2a[i]);
    }*/
    klo = 1;
    khi = c;
    while (khi - klo > 1) {
        k = (khi + klo) >> 1;
        if (xa[k] > x)
            khi = k;
        else
            klo = k;
    }
    h = xa[khi] - xa[klo];
    /*if (h == 0.0) {
        puts("Bad xa input to routine splint");
        exit(1);
    }*/
    a = (xa[khi] - x) / h;
    b = (x - xa[klo]) / h;
    y = 0.0 + a*ya[klo] + b*ya[khi] + ((a*a*a - a)*y2a[klo]
        + (b*b*b - b)*y2a[khi])*(h*h) / 6.0;
    return y;
```

```
}

__global__ void calc_pos(double *theta0, double *flx, double *intensity, double *x_att, double *y_att, double *y2_att, double *pos, int *status, double *phi0, int N) {
    unsigned id = blockIdx.x*blockDim.x + threadIdx.x;

    if (id >= N)
        return;

    if (status[id] == 0) {
        double sx = cos(phi0[id])*sin(theta0[id]);
        double sy = sin(phi0[id])*sin(theta0[id]);
        double sz = cos(theta0[id]);
        double phi = atan2(flx[N + id], flx[id]);
        double theta1 = atan2(sqrt(pow(flx[id], 2) + pow(flx[N + id], 2)), flx[2 * N + id]);
        double cosphi = cos(phi); double sinphi = sin(phi);
        double costheta = cos(theta1); double sintheta = sin(theta1);

        flx[id] = (cosphi*costheta*sx) + (-sinphi*sy) + cosphi*sintheta*sz;
        flx[N + id] = (sinphi*costheta*sx) + (cosphi*sy) + sinphi*sintheta*sz;
        flx[2 * N + id] = (-sintheta*sx) + costheta*sz;
        int fl1 = (int)(flx[id] * 1000000);
        int fl2 = (int)(flx[N + id] * 1000000);
        int fl3 = (int)(flx[2*N + id] * 1000000);
        flx[id] = fl1*1.0 / 1000000;
        flx[N + id] = fl2*1.0 / 1000000;
        flx[2 * N + id] = fl3*1.0 / 1000000;

        intensity[id] = intensity[id] * exp(-splint_device(x_att, y_att, y2_att, n_att, pos[2 * N + id]));

        pos[id] = pos[id] + flx[id];
        pos[N + id] = pos[N + id] + flx[N + id];
        pos[N * 2 + id] = pos[N * 2 + id] + flx[2 * N + id];
        int po1 = (int)(pos[id] * 1000000);
        int po2 = (int)(pos[N + id] * 1000000);
        int po3 = (int)(pos[2 * N + id] * 1000000);
        pos[id] = po1*1.0 / 1000000;
        pos[N + id] = po2*1.0 / 1000000;
        pos[2 * N + id] = po3*1.0 / 1000000;
        if (pos[2 * N + id] < 0.0) {
            pos[2 * N + id] = pos[2 * N + id]*(-1);
        }
```

```
76      }
77  }
```

splint_device is device kernel which generate interpolation or extrapolation. In the next kernel we are updating values of intensity, flux and position of rays. Since all are double values, I am storing them till 6th decimal place.

```
1   __global__ void  calc_status(double *pos, int *status, int z0, int N, int ↩
        s, int i) {
2       unsigned id = blockIdx.x*blockDim.x + threadIdx.x;
3       if (id >= N) {
4           return;
5       }
6
7       if (status[id] == 0) {
8           double t2;
9           t2 = sqrt(pos[id] * pos[id] + pos[N + id] * pos[N + id] + (pos[2 *↩
                N + id] - z0)*(pos[2 * N + id] - z0));
10
11          if (t2 <= s) {
12          status[id] = i;
13          }
14      }
15  }
16
17  __global__ void calc_beam2(double *beam2, double *pos, int z2, int N) {
18      unsigned id = blockIdx.x*blockDim.x + threadIdx.x;
19      if (id >= N)
20          return;
21      if ((pos[2 * N + id] > z2) && (beam2[id * 2 + 0]<0)) {
22          beam2[id * 2 + 0] = sqrt(pos[id] * pos[id] + pos[N + id] * pos[N +↩
                id]);
23          beam2[id * 2 + 1] = atan2(pos[id], pos[N + id]);
24      }
25  }
```

In the first kernel we are updating status values. If the ray has reached the submarine then it's value is getting updated by the number of time steps it has taken to reach the submarine. In the second kernel we are updating values of beam2.

```
1   __global__ void calc_tj(double *xa, double *ya, double *y2a, int c, double↩
        *u, double *tj, int *status, int N) {
2       int id = blockIdx.x*blockDim.x + threadIdx.x;
3       int j2 = id % (nj - 1);
```

```
    int temp = id / (nj - 1);

    if (id >= (nj - 1)*(N))
        return;
    if (status[temp] == 0) {
        if (u[temp] >= xa[j2*ni + 1]) {
            tj[j2 + temp * (nj - 1)] = splint_device((double *)xa + j2 * Nc + 1, (double *)ya, (double *)y2a + j2 * n, Nc - 1, u[temp]);
        }
        else {
            tj[j2 + temp * (nj - 1)] = 0;
        }
    }
}
```

Here we are generating values of tj.

```
__device__ void spline_device(double *x, double *y, int a, double yp1, double ypn, double *y2) {
    int i, k;
    double p, qn, sig, un;
    double u[nj];

    x--; y--; y2--;
    if (yp1 > 0.99e30)
        y2[1] = u[1] = 0.0;
    else {
        y2[1] = -0.5;
        u[1] = (3.0 / (x[2] - x[1]))*((y[2] - y[1]) / (x[2] - x[1]) - yp1);
    }
    for (i = 2; i <= a - 1; i++) {
        sig = (x[i] - x[i - 1]) / (x[i + 1] - x[i - 1]);
        p = sig*y2[i - 1] + 2.0;
        y2[i] = (sig - 1.0) / p;
        u[i] = (y[i + 1] - y[i]) / (x[i + 1] - x[i]) - (y[i] - y[i - 1]) / (x[i] - x[i - 1]);
        u[i] = (6.0*u[i] / (x[i + 1] - x[i - 1]) - sig*u[i - 1]) / p;
    }
    if (ypn > 0.99e30)
        qn = un = 0.0;
    else {
        qn = 0.5;
        un = (3.0 / (x[a] - x[a - 1]))*(ypn - (y[a] - y[a - 1]) / (x[a] - x[a - 1]));
```

```
    }
    y2[a] = (un - qn*u[a - 1]) / (qn*y2[a - 1] + 1.0);
    for (k = a - 1; k >= 1; k--)
        y2[k] = y2[k] * y2[k + 1] + u[k];
}
```

This is device kernel to generate spline. Spline generates double derivatives value of function.

```
__global__  void calc_theta0(double *yp1, double *ypn, double* tj, int *status, double *sdepth, int N, double *pos, double *theta0/*, double *sdep2, double *ydep*/)
{
    unsigned id = blockIdx.x*blockDim.x + threadIdx.x;

    if (id >= N)
        return;
    if (status[id] == 0) {
        double yp12 = (tj[id * (nj - 1) + 1] - tj[id * (nj - 1) + 0]) / (sdepth[1] - sdepth[0]);
        double ypn2 = (tj[id * (nj - 1) + nj - 2] - tj[id * (nj - 1) + nj - 2]) / (sdepth[nj - 2] - sdepth[nj - 3]);

        double ydep[nj - 1];

        for (int j3 = 0; j3<nj - 1; j3++) {
            ydep[j3] = tj[id * (nj - 1) + j3];
        }
        double sdep2[nj - 1];
        spline_device(sdepth, ydep, nj - 1, yp12, ypn2, sdep2);
        theta0[id] = splint_device(sdepth, ydep, sdep2, nj - 1, pos[2*N + id]);
    }
}
```

Here we are generating theta0 values on the device. This kernel call both the device kernel for spline and splint.

```
/* generates random variable between 0 and 1
with normal distribution*/
double randn(double mu, double sigma) {
    double U1, U2, W, mult;
    static double X1, X2;
    static int call = 0;
```

```
7
8       if (call == 1) {
9           call = !call;
10          return (mu + sigma * (double)X2);
11      }
12
13      do {
14          U1 = -1 + ((double)rand() / RAND_MAX) * 2;
15          U2 = -1 + ((double)rand() / RAND_MAX) * 2;
16          W = (double)pow(U1, 2) + (double)pow(U2, 2);
17      } while (W >= 1 || W == 0);
18
19      mult = sqrt((-2 * log(W)) / W);
20      X1 = U1 * mult;
21      X2 = U2 * mult;
22      call = !call;
23
24      return (mu + sigma * (double)X1);
25  }
```

randn() takes mu (mean) and sigma as arguments and generates random numbers between 0 and 1 of normal distribution (Gaussian Distribution).

```
1   /* this function takes x-values, y-values, slope between first and last
2   two points and generates second derivative of the function ( of x and y ↩
        values).
3   Double derivative is required to generate interpolation*/
4   void spline(double *x, double *y, int a, double yp1, double ypn, double *↩
        y2) {
5       int i, k;
6       double p, qn, sig, un, *u;
7       u = (double*)malloc((a + 1) * sizeof(double));
8
9       x--; y--; y2--;
10      if (yp1 > 0.99e30)
11          y2[1] = u[1] = 0.0;
12      else {
13          y2[1] = -0.5;
14          u[1] = (3.0 / (x[2] - x[1]))*((y[2] - y[1]) / (x[2] - x[1]) - yp1)↩
                ;
15      }
16      for (i = 2; i <= a - 1; i++) {
17          sig = (x[i] - x[i - 1]) / (x[i + 1] - x[i - 1]);
18          p = sig*y2[i - 1] + 2.0;
19          y2[i] = (sig - 1.0) / p;
```

```
        u[i] = (y[i + 1] - y[i]) / (x[i + 1] - x[i]) - (y[i] - y[i - 1]) /↩
            (x[i] - x[i - 1]);
        u[i] = (6.0*u[i] / (x[i + 1] - x[i - 1]) - sig*u[i - 1]) / p;
    }
    if (ypn > 0.99e30)
        qn = un = 0.0;
    else {
        qn = 0.5;
        un = (3.0 / (x[a] - x[a - 1]))*(ypn - (y[a] - y[a - 1]) / (x[a] - ↩
            x[a - 1]));
    }
    y2[a] = (un - qn*u[a - 1]) / (qn*y2[a - 1] + 1.0);
    for (k = a - 1; k >= 1; k--)
        y2[k] = y2[k] * y2[k + 1] + u[k];

    free(u);
}
```

spline() is function which generates second derivatives of a function given it's x and y values. It also requires slope at the first coordinate and the last coordinate that is *yp1* and *ypn*. As we don't have that parameters, we can either assume it to be 0 or the slope between the nearest coordinate.

```
/* this function takes x-values, y-values and double derivatives
as arguments and gives y point upon giving x point (interpolation)*/
double splint(double *xa, double *ya, double *y2a, int c, double x) {
    int klo, khi, k;
    double h, b, a, y;
    if(x<xa[0])
        return ya[0];
    else if (x>xa[c-1])
        return ya[c-1];

    xa--; ya--; y2a--;
    klo = 1;
    khi = c;
    while (khi - klo > 1) {
        k = (khi + klo) >> 1;
        if (xa[k] > x) khi = k;
        else klo = k;
    }
    h = xa[khi] - xa[klo];
    if (h == 0.0) {
        puts("Bad xa input to routine splint");
        exit(1);
```

```
    }
    a = (xa[khi] - x) / h;
    b = (x - xa[klo]) / h;
    y = 0.0 + a*ya[klo] + b*ya[khi] + ((a*a*a - a)*y2a[klo]
        + (b*b*b - b)*y2a[khi])*(h*h) / 6.0;
    return y;
}
```

This is **splint()** function. It does interpolation by taking x-values, y-values and second derivatives of a function. Second derivatives is generated by the **spline()** function. It takes any x-value as input and gives back the corresponding y-value by interpolating or extrapolating. For extrapolation it simply treats y-values as constants. That is for x less than x1 all y values will be equal to the y1. And for x greater than xn all y-values will be equal to the yn.

```
/* for reading files*/
int readfile(char* name, double *mat, int max_r, int r, int c) {
    char buffer[1024];
    char *record, *line;
    int i = 0, j = 0;
    FILE *fstream = fopen(name, "r");
    if (fstream == NULL) {
        printf("\n file opening failed readfiel()\n");
        return 0;
    }
    /*else{
    printf("\n file opened ");
    } */

    while ((line = fgets(buffer, sizeof(buffer), fstream)) != NULL)
    {
        record = strtok(line, ",");
        if (record[0] == '#') {
            continue;
        }
        while (record != NULL)
        {
            //here you can put the record into the array as per your ↩
               requirement.
            *(mat + j) = atof(record);  // converts the value in the file
            j++;                        // to a floating number
            record = strtok(NULL, ",");
            if ((j%c) == 0) break;
        }
        ++i;
        if (max_r != 0 && i >= max_r) break;
```

```
    }
    return i;
}
```

readfile() reads the file which has been passed to it. The file contains values which are in strings format. It converts those strings into float numbers. And it stores all the values in the matrix *mat*.

```
/* for integration*/
double intgrl(double *theta, double *sangle, double Sca_T[nj][ni], double sc[8][98], int j, double start, double end) {
    double r2deg = 180.0 / PI;
    int N = 1000; //@@Increase it for accuracy.
    double i, inc = (end - start) / N, sum = 0;
    /* taking y-values from splint()*/
    double a = splint(sangle, (double *)Sca_T + ni*j + 1, (double *)sc + (j - 1)*(ni - 1), n, start*r2deg) * 2 * PI*sin(start);
    double b;
    start += inc;

    for (i = start; i < end; i += inc) {
        b = splint(sangle, (double *)Sca_T + ni*j + 1, (double *)sc + (j - 1)*(ni - 1), n, i*r2deg) * 2 * PI*sin(i);
        sum += 0.5 * inc * (a + b);
        a = b;
    }
    return sum;
}
```

intgrl() integrates between **sangle** and Sca_T. It calls **splint()** and gets the y-values by passing x-values. Then it integrates by calculating area between two x-values and two y-values (integration by summation). It assumes the area between those points as trapezium.

```
int main(int argc, char const *argv[])
{
    clock_t start, end, p_start, p_end, p_st;
    double cpu_time_used;

    /*rays have decayed to exp(-attn*nt) of its initial value.*/
    int z0 = 100;  /* centre of submarine*/
    int s = 10;   /* size of submarine (cube)*/
    int ntraj = 10;  /* number of trajectories to track for plotting*/
    int r0 = 5;   /*size of the input beam region*/
    int wavelength = 513; /*nanometers*/
```

```
12      srand(time(0)); //# randomize the random number generator
13
14      /*file containing attenuation information*/
15      char fattenuation[] = "C:/Users/Sonu Badaik/Downloads/btp/↩
            absorption_coefficient_april2017_T3D3.csv";
16      double attnConst = 0.001;
17      double attn; //Introduced by me. Removing the function
18
19      int i, j;
20      //int i, j, n_att = 12, n;
21      double yp1, ypn;
22      double Att[351][13];
23      double *mat1;   //pointer to the array Att
24      mat1 = &Att[0][0];
25      double y2_att[n_att], y_att[n_att], x_att[n_att];
26      int row = readfile(fattenuation, mat1, 0, 351, 13);
27      for (i = 0; i<row; i++) {
28          if (Att[i][0] == wavelength) {
29              for (j = 0; j<n_att; j++) {
30                  x_att[j] = Att[0][j + 1];
31                  y_att[j] = Att[i][j + 1];
32              }
33              yp1 = (Att[i][2] - Att[i][1]) / (Att[0][2] - Att[0][1]);
34              ypn = (Att[i][n_att] - Att[i][n_att - 1]) / (Att[0][n_att] - ↩
                    Att[0][n_att - 1]);
35              spline(x_att, y_att, n_att, yp1, ypn, y2_att);
36              break;
37          }
38      }
```

This is the \textbf{main()} function. Size of the submarine has been defined on which rays should fall. We need to track the rays of wavelength 513 nanometers. The code is reading the file named \verb|absorption_coefficient_april2017_T3D3.csv| where absorption coefficient for each ray of different wavelengths are stored. \textit{Att} will store those values. If wavelength is equal to 513 nanometers, it will store that x-values and y-values. After that it will calculate slope at the first point and last point (\textit{yp1} and \textit{ypn} respectively). After that it will call \textbf{spline()} function to calculate second derivative. The second derivatives will be stored in \verb|y2_att|.

```
\begin{lstlisting}
/* fscattering contains the file name for scattering. If blank,
    model is used.
```

```
6       #fscatname="VSF-april2017-T3D3"*/
7       char fscatname[] = "C:/Users/Sonu Badaik/Downloads/btp/VSF-sept2017-↩
            T2D3";
8       //# fscatname = "VSF-sept2017-T2D1"
9       char fscattering[100];
10      strcpy(fscattering, fscatname);
11      strcat(fscattering, ".csv");
12
13      /*int ni, nj;
14      int Nc = 99;
15      ni = Nc;
16      nj = 9;*/
17
18      double Sca[ni][nj], Sca_T[nj][ni];  //Sca_T is transpose of Sca
19      double *mat2;   //pointer to Sca
20      mat2 = &Sca[0][0];
21      double sdepth[nj - 1], sangle[ni - 1], stheta[ni - 1], dtheta[ni - 1];
22      double r2deg = 180.0 / PI, angle[ni], theta[ni];
23      double sc[nj - 1][ni - 1];
24
25      if (strcmp(fscattering, "") != 0) {
26          char fsflag[] = "scatdata";
27          //Scb=loadtxt(fscattering,delimiter=",")
28          //@@Fill it. Missing from 66 - 128
29
30          row = readfile(fscattering, mat2, Nc, 99, 9);
31
32          angle[0] = 0.0; theta[0] = 0.0;
33          dtheta[0] = 0; dtheta[ni - 2] = 0;
34          for (i = 0; i<nj - 1; i++) {
35              sdepth[i] = Sca[0][i + 1];
36          }
37          for (i = 0; i<ni - 1; i++) {
38              sangle[i] = Sca[i + 1][0];
39              stheta[i] = Sca[i + 1][0] * PI / 180.0;
40              angle[i + 1] = Sca[i + 1][0];
41              theta[i + 1] = Sca[i + 1][0] * PI / 180.0;
42              if (i >= 2) {
43                  dtheta[i - 1] = (stheta[i] - stheta[i - 2])*0.5;
44              }
45          }
46
47          for (i = 0; i<ni; i++) {  //transpose of Sca
48              for (j = 0; j<nj; j++) {
49                  Sca_T[j][i] = Sca[i][j];
50              }
51          }
```

```
52
53          //n = 98;
54          for (j = 1; j<nj; j++) {//@@Check this function
55              yp1 = (Sca[2][j] - Sca[1][j]) / (sangle[1] - sangle[0]);
56              ypn = (Sca[Nc - 1][j] - Sca[Nc - 2][j]) / (sangle[Nc - 2] - ↩
                    sangle[Nc - 3]);
57              //spline(sangle,*(Sca_T+Nc*j+1),n,yp1,ypn,*(sc+(j-1)*(Nc-1)))↩
                    ;//@@Check this line
58              spline(sangle, (double *)Sca_T + (Nc*j) + 1, n, yp1, ypn, (↩
                    double *)sc + (j - 1)*(Nc - 1));//@@Checked
59          }
60
61      }
```

In this part we are reading the file `VSF-sept2017-T2D3.csv`, which contains VSF values for different angles. Values are getting stored in the array Sca. We are storing the angle values in *angle* and so on. Then we are calling the **spline()** to make spline curves between *Sca* and *angle* which stores all the second order derivatives in sc.

```
1       double P[nj][ni];
2       for (int temp1 = 0; temp1<nj; temp1++) {
3           for (int temp2 = 0; temp2<ni; temp2++) {
4               P[temp1][temp2] = 0.0;
5           }
6       }
7       //# Pnorm=zeros(P.shape)
8
9       for (j = 0; j<nj - 1; j++) {
10          for (i = 0; i<ni - 1; i++) {
11              P[j][i + 1] = intgrl(theta, sangle, Sca_T, sc, j + 1, stheta↩
                    [0], stheta[i]);
12          }
13          for (i = 1; i<ni; i++) {
14              P[j][i] = P[j][i] + 1 - P[j][ni - 1];
15          }
16      }
17
18      FILE *fp1;
19      fp1 = fopen("P.csv", "w");
20      if (fp1 == NULL) {
21          printf("Error!");
22          exit(1);
23      }
24
25      for (int i2 = 0; i2<nj; i2++) {
```

```
26          for (int j2 = 0; j2<ni; j2++) {
27              fprintf(fp1, "%lf", P[i2][j2]);
28              if (j2 != (ni - 1)) fprintf(fp1, ",");
29          }
30          fprintf(fp1, "\n");
31      }
32      fclose(fp1);
33
34      double Pinv[nj - 1][n];
35      double tt;
36      for (j = 0; j<nj - 1; j++) {
37          yp1 = (angle[2] - angle[1]) / (P[j][2] - P[j][1]);
38          ypn = (angle[Nc - 1] - angle[Nc - 2]) / (P[j][Nc - 1] - P[j][Nc - ↩
                2]);
39          spline((double *)P + Nc*j + 1, (double *)angle + 1, n, yp1, ypn, (↩
                double *)Pinv + j*n);
40      }
```

In this listing, P[][][] stores the CDF between *sangle* and *Sca* by calling **intgrl()** function which then calls the splint function to get the integrands. Then it writes those values in a file which is then used for plotting. Next we fit a spline curve between P and *angle* and store the second order derivatives in Pinv.

```
1
2   //# Allocate arrays for storing the results
3       int N = 100000;// # number of rays in all
4
5       double *pos[3];//=zeros((3,N)) # tracks position of rays.D
6       for (int i = 0; i<3; i++) {
7           pos[i] = (double*)malloc(N * sizeof(double));
8       }
9
10      double *direction[3];// = zeros((3,N)); # remembers dirction of ray.D
11      for (int i = 0; i<3; i++) {
12          direction[i] = (double*)malloc(N * sizeof(double));
13      }
14
15
16      double *intensity = (double*)malloc(N * sizeof(double));// = ones(N)↩
            ;//  # intensity of ray initially one.D
17
18      int *tsrc = (int*)malloc(N * sizeof(int));// = zeros(N,dtype='int');//↩
                # time at which ray was born.D
19
20      double *flx[3];//=zeros((3,N));//   # direction from which the ray hit↩
```

```
        the submarine.D
21      for (int i = 0; i<3; i++) {
22          flx[i] = (double*)malloc(N * sizeof(double));
23      }
24
25      int *status = (int*)malloc(N * sizeof(int));//=-1*ones(N,dtype='int')
            ;//  # if zero active, if positive, reached sub at that time.D
26
27      double channel[nt];//=zeros(nt);//  # will hold the channel.D
28                         //double traj[3][ntraj][nt];//=zeros((3,ntraj,nt))
                               ;// # holds the trajectories of a selected
                               number of rays.D
29                         /*# We store the positions of rays when they first
30                         # cross z=z1 and z=z2. The arrays are initialized
31                         # to an impossible number.*/
32
33      double *traj = (double*)malloc(3 * ntraj*nt * sizeof(double));
34
35      double **beam1;// = -1*ones((N,2)); /*# holds the (r,theta) positions
            of rays when they first cross z1.D*/
36      beam1 = (double**)malloc(N * sizeof(double*));
37      for (int i = 0; i<N; i++) {
38          beam1[i] = (double*)malloc(2 * sizeof(double));
39      }
40      int z1 = 0;
41
42      double *beam2 = (double*)malloc(2 * N * sizeof(double));
43      int z2 = z0 - 2 * s; //Both are ints defined earlier
44      int rmax2 = 10000;
45      int nbins2 = 100;
46      //# values per time step .
47      int a2 = (nt * 100) + 1000;
48      double **theta0vals;// = zeros((nt*100+1000,2)) /*#to hold 100 random
            theta0 */
49                          //# values per time step .
50      theta0vals = (double**)malloc(sizeof(double*)*a2);
51      for (int i = 0; i<a2; i++) {
52          theta0vals[i] = (double*)malloc(2 * sizeof(double));
53      }
54
55      //# Main simulation loop
56      int k = 0;//  # keeps track of where we can insert new rays
57
58      double *phi0 = (double*)malloc(sizeof(double)*N);//=zeros(status.shape
            );D
59      double *theta0 = (double*)malloc(sizeof(double)*N);//=zeros(status.
            shape);D
```

```
60
61      for (int temp1 = 0; temp1<3; temp1++) {
62          for (int temp2 = 0; temp2<ntraj; temp2++) {
63              for (int temp3 = 0; temp3<nt; temp3++) {
64                  traj[temp1 * ntraj * nt + temp2*nt + temp3] = 0.0;
65              }
66          }
67      }
68
69      for (int temp1 = 0; temp1<N; temp1++) {
70          phi0[temp1] = 0;
71          theta0[temp1] = 0;
72          status[temp1] = 0;
73          tsrc[temp1] = 0;
74          intensity[temp1] = 1;
75
76          for (int temp2 = 0; temp2<2; temp2++) {
77              beam1[temp1][temp2] = -1.0;
78              beam2[temp1 * 2 + temp2] = -1.0;
79          }
80
81          for (int temp2 = 0; temp2<3; temp2++) {
82              direction[temp2][temp1] = 0;
83          }
84      }
```

In this part we have declared the required variables. We are tracking N arrays. *pos* stores the current positions of all the rays. Similarly *traj* stores the trajectory of all the rays. *status* stores whether the ray has hit the submarine or not.

```
1   double phi1, r;
2       FILE *fpt, *fp;
3       //fpt = fopen("pos_c.csv", "w");
4       //fp = fopen("phi0_c.csv", "w");
5       /*if(fpt == NULL){
6       printf("Error!");
7       exit(1);
8       }*/
9
10      for (int i2 = 0; i2<N; i2++) {
11          phi1 = (1.0*rand() / RAND_MAX) * 2 * PI;
12          r = randn(0, 1)*r0;
13          pos[0][i2] = r*cos(phi1);
14          pos[1][i2] = r*sin(phi1);
15          pos[2][i2] = 0;
```

```
16
17          flx[0][i2] = 0; flx[1][i2] = 0; flx[2][i2] = 1;
18      }
19      //fclose(fpt);
20
21      int l = 0, iii, jjj, kk, cnt;
22
23      double *tj = (double*)malloc(N*(nj - 1) * sizeof(double));
24
25      int *iit;
26      iit = (int *)malloc(N * sizeof(int));
27      int *ii;
28      ii = (int *)malloc(N * sizeof(int));
29
30      double sx, sy, sz, phi, theta1, cosphi, sinphi, sintheta, costheta;
31
32      double *u;
33      u = (double*)malloc(N * sizeof(double));
34
35      p_start = clock();
36
37      /* CUDA variables for random numbers */
38
39      curandState_t* states;
40      cudaMalloc((void**)&states, N * sizeof(curandState_t));  //check if it↩
            is N or nii
41
42      double *phi0_dev, *u_dev;
43      int *ii_dev, *nii_dev;
44
45      cudaMalloc(&phi0_dev, N * sizeof(double));
46      cudaMemcpy(phi0_dev, phi0, N * sizeof(double), cudaMemcpyHostToDevice)↩
            ;
47      cudaMalloc(&u_dev, N * sizeof(double));
48      cudaMalloc(&ii_dev, N * sizeof(int));
49      cudaMemcpy(ii_dev, ii, N * sizeof(double), cudaMemcpyHostToDevice);
50      cudaMalloc(&nii_dev, sizeof(int));
51
52
53      /* CUDA variables for tj part */
54      double *tj_dev, *P_dev, *angle_dev, *Pinv_dev;
55
56      cudaMalloc(&tj_dev, N *(nj - 1) * sizeof(double));
57      cudaMalloc(&P_dev, nj*ni * sizeof(double));
58      cudaMalloc(&angle_dev, ni * sizeof(double));
59      cudaMalloc(&Pinv_dev, (nj - 1) *n * sizeof(double));
60      cudaMemcpy(P_dev, P, nj*ni * sizeof(double), cudaMemcpyHostToDevice);
```

```
61      cudaMemcpy(angle_dev, angle, ni * sizeof(double), ↩
            cudaMemcpyHostToDevice);
62      cudaMemcpy(Pinv_dev, Pinv, (nj - 1)*n * sizeof(double), ↩
            cudaMemcpyHostToDevice);
63
64      /* CUDA variables for theta0 */
65
66      double *yp1_dev, *ypn_dev, *sdepth_dev, *theta0_dev, *pos_dev, *↩
            sdep2_dev, *ydep_dev;
67
68      cudaMalloc(&yp1_dev, sizeof(double));
69      cudaMalloc(&ypn_dev, sizeof(double));
70      cudaMalloc(&sdepth_dev, (nj - 1) * sizeof(double));
71      cudaMalloc(&theta0_dev, N * sizeof(double));
72      cudaMalloc(&pos_dev, 3 * N * sizeof(double));
73      cudaMalloc(&sdep2_dev, (nj - 1) * sizeof(double));
74      cudaMalloc(&ydep_dev, (nj - 1) * sizeof(double));
75      cudaMemcpy(sdepth_dev, sdepth, (nj - 1) * sizeof(double), ↩
            cudaMemcpyHostToDevice);
76
77      cudaMemcpy(pos_dev, pos[0], N * sizeof(double), cudaMemcpyHostToDevice↩
            );
78      cudaMemcpy(pos_dev + N, pos[1], N * sizeof(double), ↩
            cudaMemcpyHostToDevice);
79      cudaMemcpy(pos_dev + 2 * N, pos[2], N * sizeof(double), ↩
            cudaMemcpyHostToDevice);
80
81      /* CUDA variable for flx and pos */
82
83      double *flx_dev, *intensity_dev, *y2_att_dev, *y_att_dev, *x_att_dev;
84      cudaMalloc(&flx_dev, N * 3 * sizeof(double));
85      cudaMalloc(&intensity_dev, N * sizeof(double));
86      cudaMalloc(&y2_att_dev, n_att * sizeof(double));
87      cudaMalloc(&y_att_dev, n_att * sizeof(double));
88      cudaMalloc(&x_att_dev, n_att * sizeof(double));
89      cudaMemcpy(flx_dev, flx[0], N * sizeof(double), cudaMemcpyHostToDevice↩
            );
90      cudaMemcpy(flx_dev + N, flx[1], N * sizeof(double), ↩
            cudaMemcpyHostToDevice);
91      cudaMemcpy(flx_dev + N * 2, flx[2], N * sizeof(double), ↩
            cudaMemcpyHostToDevice);
92      cudaMemcpy(intensity_dev, intensity, N * sizeof(double), ↩
            cudaMemcpyHostToDevice);
93      cudaMemcpy(y2_att_dev, y2_att, n_att * sizeof(double), ↩
            cudaMemcpyHostToDevice);
94      cudaMemcpy(y_att_dev, y_att, n_att * sizeof(double), ↩
            cudaMemcpyHostToDevice);
```

```
95      cudaMemcpy(x_att_dev, x_att, n_att * sizeof(double), ↩
            cudaMemcpyHostToDevice);
96
97      /* CUDA variable for status */
98
99      int *status_dev;
100     cudaMalloc(&status_dev, N * sizeof(int));
101     cudaMemcpy(status_dev, status, N * sizeof(int), cudaMemcpyHostToDevice↩
            );
102
103     /* CUDA variables for traj */
104     double *traj_dev;
105     cudaMalloc(&traj_dev, 3 * ntraj*nt * sizeof(double));
106     cudaMemcpy(traj_dev, traj, 3 * ntraj*nt * sizeof(double), ↩
            cudaMemcpyHostToDevice);
107
108     /* CUDA variables for beam2*/
109     double *beam2_dev;
110     cudaMalloc(&beam2_dev, N * 2 * sizeof(double));
111     cudaMemcpy(beam2_dev, beam2, N * 2 * sizeof(double), ↩
            cudaMemcpyHostToDevice);
```

In this section too we have declared various required variables. All the variables which has been declared using malloc() needs to be declare in this section or before this section. Because in the next section main for loop starts. If we use malloc() inside the main for loop then it will reduce the speed. Hence, it is better to avoid using malloc() inside a for loop.

```
1   p_start = clock();
2
3       init << <N / 512 + 1, 512 >> >(time(0), states); cudaDeviceSynchronize↩
            ();
4
5       p_end = clock();
6       cpu_time_used = ((double)(p_end - p_start)) / CLOCKS_PER_SEC;
7       printf("Initialization of States >(%f sec)\n", cpu_time_used);
8
9       start = clock();//Storing clock value
10
11      for (k = 0; k < nt; k += 100) {
12          if (k + 100 <= nt) {
13              kk = k + 100;
14          }
15          else {
16              kk = nt;
17          }
```

```
18
19          for (i = k; i<kk; i++) {
20              p_st = clock();
21
22              //generate the random move.
23              //Bias the rays to scatter within p radians of orig direction.
24              if (fscattering != "") { //use CSV data
25                  p_start = clock();
26
27                  /*Random number Generator in CPU*/
28                  /*for (iii = 0; iii<nii; iii++) {
29                      phi0[ii[iii]] = (1.0*rand() / RAND_MAX) * 2 * PI;
30                      u[iii] = (1.0*rand() / RAND_MAX);
31                  }*/
32
33
34                  /* random number generator in CUDA */
35                  p_start = clock();
36
37                  randoms << <N / 512 + 1, 512 >> > (states, phi0_dev, ↩
                        status_dev, u_dev, N);
38                  //cudaDeviceSynchronize();
39
40                  p_end = clock();
41                  cpu_time_used = ((double)(p_end - p_start)) / ↩
                        CLOCKS_PER_SEC;
42                  printf("random numbers > %d (%f sec)\n", k, cpu_time_used)↩
                        ;
```

Before the main for loop we are calling device kernel init«<»> which initializes states to produce different random numbers each time we run the program. After that the main for loop starts. Here it is generating random numbers and storing in *phi0* and *u*. Random number generator for the CPU part has been commented and GPU part has been used.

```
1   p_start = clock();
2
3                   /* generating tj in CPU */
4                   /*for (int j2 = 0; j2<nj - 1; j2++) {
5                       for (int temp = 0; temp<N; temp++) {
6                           if (status1[temp] == 0) {
7                               if (u[temp] >= P[j2][1]) {
8                                   tj1[temp][j2] = splint((double *)P + j2 * ↩
                                        Nc + 1, (double *)angle + 1, (double ↩
                                        *)Pinv + j2 * n, Nc - 1, u[temp]);
9                               }
```

```
10                          else tj1[temp][j2] = 0;
11                      }
12                  }
13              }*/
14
15
16              /* for generating tj in GPU */
17              calc_tj << <(nj - 1)*N / 512 + 1, 512 >> > (P_dev, ↩
                    angle_dev + 1, Pinv_dev, Nc - 1, u_dev, tj_dev, ↩
                    status_dev, N);
18              //cudaDeviceSynchronize();
19
20              p_end = clock();
21              cpu_time_used = ((double)(p_end - p_start)) / ↩
                    CLOCKS_PER_SEC;
22              printf("for tj > %d (%f sec)\n", k, cpu_time_used);
23
24              p_start = clock();
25
26              /* generating theta0 in cpu */
27              /*for (int i2 = 0; i2<N; i2++) {
28                  if (status[i2] == 0) {
29                      yp1 = (tj[i2*(nj-1)+1] - tj[i2*(nj - 1)+0]) / (↩
                            sdepth[1] - sdepth[0]);
30                      ypn = (tj[i2*(nj-1)+nj - 2] - tj[i2*(nj - 1)+nj - ↩
                            2]) / (sdepth[nj - 2] - sdepth[nj - 3]);
31
32                      for (int j3 = 0; j3<nj - 1; j3++) {
33                          ydep[j3] = tj[i2*(nj - 1) + j3];
34                      }
35
36                      spline(sdepth, ydep, nj - 1, yp1, ypn, sdep2);
37
38                      theta01[i2] = splint(sdepth, ydep, sdep2, nj - 1, ↩
                            pos[2][i2]);
39                  }
40              }*/
41
42              /* generating theta0 in gpu */
43
44              calc_theta0 << <N / 512 + 1, 512 >> >(yp1_dev, ypn_dev, ↩
                    tj_dev, status_dev, sdepth_dev, N, pos_dev, theta0_dev↩
                    );
45              //cudaDeviceSynchronize();
46
47              p_end = clock();
48              cpu_time_used = ((double)(p_end - p_start)) / ↩
```

```
                    CLOCKS_PER_SEC;
49                  printf("theta0 > %d (%f sec)\n", k, cpu_time_used);
50              }
```

In this section we are generating tj[][]. It is calling both *spline()* and *splint()*. Generating tj[][] in the CPU has been commented and tj[][] has been generated from GPU. The spline curve (Pinv) generated earlier is used to find the values of tj. The if condition ensures that the spline curve is being interpolated. Based on this tj values, we again draw spline curves along rows of tj and stores the splint values in theta0[].

```
1   if (i % 10 == 0) {
2                   int l1;
3                   int count = 0;
4                   for (int iii = 0; iii<N; iii++) {
5                       if (theta0[iii]>0.08) {
6                           iit[count] = iii;
7                           count++;
8                       }
9                   }
10                  if (count>1000) l1 = 1000;
11                  else l1 = count;
12
13                  for (count = l; count<l + l1; count++) {
14                      theta0vals[count][1] = theta0[iit[count - l]];
15                      theta0vals[count][0] = pos[2][iit[count - l]];
16                  }
17                  l += l1;
18              }
```

After every 10th iteration we will update the value of theta0vals[] whenever theta0[] will be greater than 0.08, iit[] will store it's indices. And depending on the value of l1, we update theta0vals. But as we progress we do find that this section is not used anywhere and can be removed.

```
1               p_start = clock();
2
3               /* updating flx, pos and intensity in cpu */
4                   /*for (iii = 0; iii<N; iii++) {
5                       if (status1[iii] == 0) {
6                           sx = cos(phi0[iii])*sin(theta01[iii]);
7                           sy = sin(phi0[iii])*sin(theta01[iii]);
8                           sz = cos(theta01[iii]);
9                           phi = atan2(flx1[1][iii], flx1[0][iii]);
10                          theta1 = atan2(sqrt(pow(flx1[0][iii], 2) + pow(↩
```

```
                        flx1[1][iii], 2)), flx1[2][iii]);
11                      cosphi = cos(phi); sinphi = sin(phi);
12                      costheta = cos(theta1); sintheta = sin(theta1);
13
14                      flx1[0][iii] = (cosphi*costheta*sx) + (-sinphi*sy)↩
                            + cosphi*sintheta*sz;
15                      flx1[1][iii] = (sinphi*costheta*sx) + (cosphi*sy) ↩
                           + sinphi*sintheta*sz;
16                      flx1[2][iii] = (-sintheta*sx) + costheta*sz;
17
18                      int fl1 = (int)(flx1[0][iii] * 1000000);
19                      int fl2 = (int)(flx1[1][iii] * 1000000);
20                      int fl3 = (int)(flx1[2][iii] * 1000000);
21                      flx1[0][iii] = fl1*1.0/ 1000000;
22                      flx1[1][iii] = fl2*1.0 / 1000000;
23                      flx1[2][iii] = fl3*1.0 / 1000000;
24
25                      intensity1[iii] = intensity1[iii] * exp(-splint(↩
                           x_att, y_att, y2_att, n_att, pos1[2][iii]));
26                      pos1[0][iii] = pos1[0][iii] + flx1[0][iii];
27                      pos1[1][iii] = pos1[1][iii] + flx1[1][iii];
28                      pos1[2][iii] = pos1[2][iii] + flx1[2][iii];
29                      int po1 = (int)(pos1[0][iii] * 1000000);
30                      int po2 = (int)(pos1[1][iii] * 1000000);
31                      int po3 = (int)(pos1[2][iii] * 1000000);
32                      pos1[0][iii] = po1*1.0 / 1000000;
33                      pos1[1][iii] = po2*1.0 / 1000000;
34                      pos1[2][iii] = po3*1.0 / 1000000;
35                      if (pos1[2][iii] < 0.0) {
36                          pos1[2][iii] = pos1[2][iii]*(-1);
37                      }
38                  }
39              }*/
40
41              /* updating flx, pos and intensity in gpu */
42
43              calc_pos << <N / 512 + 1, 512 >> >(theta0_dev, flx_dev, ↩
                    intensity_dev, x_att_dev, y_att_dev, y2_att_dev, pos_dev, ↩
                    status_dev, phi0_dev, N);
44              //cudaDeviceSynchronize();
45
46              p_end = clock();
47              cpu_time_used = ((double)(p_end - p_start)) / CLOCKS_PER_SEC;
48              printf("flx, pos > %d (%f sec)\n", k, cpu_time_used);
49
50              p_start = clock();
51
```

```
52              /* generating status in cpu */
53              /*for (int i2 = 0; i2<N; i2++) {
54                  if (status1[i2] == 0) {
55                      double t2;
56                      t2 = sqrt(pos1[0][i2] * pos1[0][i2] + pos1[1][i2] * ↩
                            pos1[1][i2] + (pos1[2][i2] - z0)*(pos1[2][i2] - z0↩
                            ));
57
58                      if (t2 <= s) {
59                      status1[i2] = i;
60                  }
61              }*/
62
63              /* generating status in gpu */
64
65              calc_status << <N / 512 + 1, 512 >> > (pos_dev, status_dev, z0↩
                    , N, s, i);
66              //cudaDeviceSynchronize();
67
68              p_end = clock();
69              cpu_time_used = ((double)(p_end - p_start)) / CLOCKS_PER_SEC;
70              printf("status > %d (%f sec)\n", k, cpu_time_used);
71
72              p_start = clock();
73
74              //save trajectories
75              /* saving values of pos in traj in cpu */
76              /*for (int tt = 0; tt<3; tt++) {
77                  for (int tp = 0; tp<ntraj; tp++) {
78                  traj1[tt*ntraj*nt+tp*nt+i] = pos1[tt][tp];
79                  }
80              }*/
81
82              /* saving values of pos in traj in gpu */
83
84              calc_traj << <3 * ntraj / 1024 + 1, 1024 >> > (traj_dev, ↩
                    pos_dev, ntraj, i, N);
85              //cudaDeviceSynchronize();
86
87              p_end = clock();
88              cpu_time_used = ((double)(p_end - p_start)) / CLOCKS_PER_SEC;
89              printf("traj > %d (%f sec)\n", k, cpu_time_used);
```

In this section, we are finding the next position of the rays. First we are calculating values of *sx*,*sy* and *sz* by using phi0[] and theta0[]. After that we are calculating flx[][] values for all

the rays for 3 direction i.e. x, y and z. After getting values of flx[][][] we are updating the new position of the rays. And we are storing the new positions in traj[][][]. Now if a ray has reached the submarine i.e. if it's height($t2$) is less than s then we will update it's status[] value to i. status[] values indicate that the ray has reached the submarine in that number of steps.

```
/*# not a good algorithm: assumes rays with (0,0) have not yet reached corresponding depth.
            # save rays crossing z1*/
            /*for (int tt = 0; tt<N; tt++) {
                if ((pos[2][tt] > z1) && (beam1[tt][0]<0)) {
                    beam1[tt][0] = sqrt(pos[0][tt] * pos[0][tt] + pos[1][tt] * pos[1][tt]);
                    beam1[tt][1] = atan2(pos[0][tt], pos[1][tt]);
                }
            }*/

            p_start = clock();

            //# save rays crossing z2
            /* generating beam2 in cpu */
            /*for (int tt = 0; tt<N; tt++) {
                if ((pos1[2][tt] > z2) && (beam21[tt*2]<0)) {
                beam21[tt*2] = sqrt(pos1[0][tt] * pos1[0][tt] + pos1[1][tt] * pos1[1][tt]);
                beam21[tt*2 + 1] = atan2(pos1[0][tt], pos1[1][tt]);
                }
            }*/

            /* generating beam2 in gpu */

            calc_beam2 << <N / 512 + 1, 512 >> >(beam2_dev, pos_dev, z2, N);
            //cudaDeviceSynchronize();

            p_end = clock();
            cpu_time_used = ((double)(p_end - p_start)) / CLOCKS_PER_SEC;
            printf("beam2 > %d (%f sec)\n", k, cpu_time_used);

            end = clock();
            cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
            double cpu_time_used1 = ((double)(end - p_st)) / CLOCKS_PER_SEC;
            printf("\n\nLoop - %d (%lf sec) loop time (%lf sec)\n", i, cpu_time_used, cpu_time_used1);

        }
```

```
36      }
37
38      p_start = clock();
39
40      cudaMemcpy(flx[0], flx_dev, N * sizeof(double), cudaMemcpyDeviceToHost↩
            );
41      cudaMemcpy(flx[1], flx_dev + N, N * sizeof(double), ↩
            cudaMemcpyDeviceToHost);
42      cudaMemcpy(flx[2], flx_dev + 2 * N, N * sizeof(double), ↩
            cudaMemcpyDeviceToHost);
43      cudaMemcpy(intensity, intensity_dev, N * sizeof(double), ↩
            cudaMemcpyDeviceToHost);
44
45      cudaMemcpy(traj, traj_dev, 3 * ntraj*nt * sizeof(double), ↩
            cudaMemcpyDeviceToHost);
46
47      cudaMemcpy(status, status_dev, N * sizeof(int), cudaMemcpyDeviceToHost↩
            );
48      cudaMemcpy(beam2, beam2_dev, N * 2 * sizeof(double), ↩
            cudaMemcpyDeviceToHost);
49
50      p_end = clock();
51      cpu_time_used = ((double)(p_end - p_start)) / CLOCKS_PER_SEC;
52      printf("Copying variables DeviceToHost >(%f sec)\n", cpu_time_used);
```

This is the final part of the main for loop. In this section we are updating the values of beam1[] and beam2[] according to the new position of the rays. Hare also we can calculate how much time each iteration of the loop takes. At the end of the loop we are copying back all the values from device to host.

```
1   double pvals[nj - 1][Nc - 1];
2
3       for (i = 0; i<nj - 1; i++) {
4           for (j = 0; j<ni - 1; j++) {
5               pvals[i][j] = Sca_T[i + 1][j + 1] * 180.0 / PI;
6           }
7       }
8
9       printf("Storing Variables -> status,theta0vals,flx,intensity,tsrc,↩
            channel,traj,beam2,pvals\n");
10      printf("Constant Variables -> rmax2,ntraj,nbins2,nt,sdepth,sangle\n");
11
12      FILE *fptr;
13
14      fptr = fopen("st.csv", "w");
```

```
    if (fptr == NULL) {
        printf("Error!");
        exit(1);
    }
    fprintf(fptr, "#Status\n");
    for (i = 0; i<N; i++) {
        fprintf(fptr, "%d", status[i]);
        if (i != (N - 1)) fprintf(fptr, ",");
    }
    fclose(fptr);

    fptr = fopen("pvals.csv", "w");
    for (i = 0; i<nj - 1; i++) {
        for (j = 0; j<Nc - 1; j++) {
            fprintf(fptr, "%lf", pvals[i][j]);
            if (j != Nc - 2) fprintf(fptr, ",");
        }
        fprintf(fptr, "\n");
    }
    fclose(fptr);

    fptr = fopen("th.csv", "w");
    fprintf(fptr, "#theta0vals - Transpose\n");
    for (i = 0; i<2; i++) {
        for (j = 0; j<a2; j++) {
            fprintf(fptr, "%lf", theta0vals[j][i]);
            if (j != a2 - 1) fprintf(fptr, ",");
        }
        fprintf(fptr, "\n");
    }
    fclose(fptr);

    fptr = fopen("flx.csv", "w");
    fprintf(fptr, "#flx\n");
    for (i = 0; i<3; i++) {
        for (j = 0; j<N; j++) {
            fprintf(fptr, "%lf", flx[i][j]);
            if (j != N - 1) fprintf(fptr, ",");
        }
        fprintf(fptr, "\n");
    }
    fclose(fptr);
```

In this section we are storing the values of `Sca_T` in pvals[] to plot graph between *sangle* and pvals[]. After that we storing all the variables in excel files. First we are storing values of

status[]. Then pvals[], theta0vals[] and flx[][].

```
fptr = fopen("in.csv", "w");
    fprintf(fptr, "#intensity\n");
    for (i = 0; i<N; i++) {
        fprintf(fptr, "%lf", intensity[i]);
        if (i != N - 1) fprintf(fptr, ",");
    }
    fclose(fptr);

    fptr = fopen("tsrc.csv", "w");
    fprintf(fptr, "#tsrc\n");
    for (i = 0; i<N; i++) {
        fprintf(fptr, "%d", tsrc[i]);
        if (i != N - 1) fprintf(fptr, ",");
    }
    fclose(fptr);

    /*fptr = fopen("ch.csv", "w");
    fprintf(fptr, "#channel\n");
    for(i=0;i<nt;i++){
    fprintf(fptr, "%lf",channel[i]);
    if(i!=(nt-1)) fprintf(fptr, ",");
    }
    fclose(fptr);*/

    fptr = fopen("beam.csv", "w");
    fprintf(fptr, "#beam2 - Transpose\n");
    for (i = 0; i<2; i++) {
        for (j = 0; j<N; j++) {
            fprintf(fptr, "%lf", beam2[j * 2 + i]);
            if (j != N - 1) fprintf(fptr, ",");
        }
        fprintf(fptr, "\n");
    }
    fclose(fptr);

    fptr = fopen("traj.csv", "w");
    fprintf(fptr, "#traj\n");
    for (i = 0; i<3; i++) {
        for (j = 0; j<10; j++) {
            for (k = 0; k<nt; k++) {
                fprintf(fptr, "%lf", traj[i * ntraj * nt + j*nt + k]);
                if (k != nt - 1) fprintf(fptr, ",");
            }
            fprintf(fptr, "\n");
        }
```

```
46          fprintf(fptr, "\n\n");
47      }
48      fclose(fptr);
```

This section continues to store the variables in excel files. We are storing intensity[] values. Then tsrc[], beam2[][] and traj[][].

```
1   /* freeing pointers */
2
3       free(tsrc);
4       free(iit);
5       for (int i = 0; i<a2; i++) {
6           free(theta0vals[i]);
7       }
8       free(theta0vals);
9
10      /* random numbers */
11      free(phi0);
12      free(u);
13      free(ii);
14      cudaFree(phi0_dev);
15      cudaFree(u_dev);
16      cudaFree(ii_dev);
17      cudaFree(nii_dev);
18
19      /* tj */
20      free(tj);
21      cudaFree(tj_dev);
22      cudaFree(P_dev);
23      cudaFree(angle_dev);
24      cudaFree(Pinv_dev);
25
26      /* theta0 */
27      free(theta0);
28      for (int i = 0; i < 3; i++) {
29          free(pos[i]);
30          free(flx[i]);
31          free(direction[i]);
32      }
33      cudaFree(yp1_dev);
34      cudaFree(ypn_dev);
35      cudaFree(sdepth_dev);
36      cudaFree(theta0_dev);
37      cudaFree(pos_dev);
38      cudaFree(sdep2_dev);
```

```
39      cudaFree(ydep_dev);
40
41      /* flx,intensity*/
42      free(intensity);
43      cudaFree(flx_dev);
44      cudaFree(intensity_dev);
45      cudaFree(y2_att_dev);
46      cudaFree(y_att_dev);
47      cudaFree(x_att_dev);
48
49      /* status */
50      free(status);
51      cudaFree(status_dev);
52
53      /* traj */
54      free(traj);
55      cudaFree(traj_dev);
56
57      /* beam2 */
58      for (int i = 0; i<N; i++) {
59          free(beam1[i]);
60      }
61      cudaFree(beam2);
62      return 0;
63  }
```

In this section, we freeing all the variables which were defined in the heap section.

B Python Code for Plotting Graphs

```
from pylab import *
import mpl_toolkits.mplot3d.axes3d as p3
import sys
#from scipy import weave # not used now
from scipy.integrate import quad
import time
from scipy.interpolate import UnivariateSpline

N = 100000
nt=10002  # number of time steps to simulate. By this time, active
z0=100  # centre of submarine
s=10    # size of submarine (cube)
ntraj=10 # number of trajectories to track for plotting
seed() # randomize the random number generator
r0=5   # size of the input beam region
wavelength=513 # nanometers
# file containing attenuation information
fattenuation="C:/Users/Sonu Badaik/Downloads/btp/absorption_coefficient_april2017_T3D3.csv"
# fattenuation=""
attnConst=0.001
# if file is an empty string, attenuation is a fixed number, given by attnConst

fscatname="C:/Users/Sonu Badaik/Downloads/btp/VSF-sept2017-T2D3"
# fscatname="VSF-sept2017-T2D1"
fscattering=fscatname+".csv"

Nc=99
fsflag="scatdata"
Scb=loadtxt(fscattering,delimiter=",")
ni,nj=Scb.shape

nj=9
#####################################################
Sca=Scb[:Nc,:nj] # truncate to nj columns
ni=Nc
del Scb
# Sca has scattering data. Column 0 contains the angles and
# row zero contains the depth values. The rest are the
# scattering data for that depth, angle combination.
sdepth=Sca[0,1:]
sangle=Sca[1:,0]       # in degrees
stheta=sangle*pi/180.0 # in radians
```

```
43 dtheta=zeros(stheta.shape)
44 dtheta[1:-1]=(stheta[2:]-stheta[:-2])*0.5
45 r2deg=180.0/pi
46 angle=zeros(ni)
47 angle[1:]=sangle # data does not have theta=0. angle has this.
48 theta=angle*pi/180.0
49
50 z1=0
51 z2=z0-2*s
52 rmax2=10000
53 nbins2=100
54
55 P = loadtxt("P.csv",delimiter=",")
56 status = loadtxt("st.csv",delimiter=",")
57 flx = loadtxt("flx.csv",delimiter=",")
58 theta0vals = loadtxt("th.csv",delimiter=",")
59 beam2 = loadtxt("beam.csv",delimiter=",")
60 temp = loadtxt("traj.csv",delimiter=",")
61 #channel = loadtxt("ch.csv",delimiter=",")
62 pvals = loadtxt("pvals.csv",delimiter=",")
63 channel=zeros(nt)
64 tsrc = loadtxt("tsrc.csv",delimiter=",")
65 intensity = loadtxt("in.csv",delimiter=",")
66 beam2 = beam2.T
67 theta0vals = theta0vals.T
68
69 traj=zeros((3,ntraj,nt))
70 traj[0,:,:] = temp[:10,:]
71 traj[1,:,:] = temp[10:20,:]
72 traj[2,:,:] = temp[20:,:]
73
74
75 fname = "test"
76
77 # post processing
78 print ("\n\n%d rays (out of %d) reached the submarine" % (len(where(status↩
      >0)[0]),N))
79 '''z0max=int(theta0vals[0,:].max())
80 for zz in range(z0max):
81         iii=where( abs(theta0vals[0,:]-zz)<=0.5 )[0]
82         print (len(iii))
83         theta0vals[1,iii[1000:]]=-1
84 '''
85
86 z=zeros((ni-1,nj-1))
87
88 figure(7)
```

```
for j in range(1,nj):
    loglog(sangle,Sca[1:,j])
    z[:,j-1]=Sca[1:,j]

title("Scattering profile data")
xlabel(r"$\theta$",size=16)
name = fname+"scat-profile.png"
savefig(name)


figure(8)
w=log10(abs(z.T))
w[w<-5]=-5
contourf(sangle,sdepth,w[-1::-1],[-5,-2,-1,0,0.5,1,1.5,2])
colorbar()
xlabel(r'$\theta$',size=16)
ylabel(r'$d$',size=16)
title(r"measured scattering data")
name = fname+"-scat-contour.png"
savefig(name)



figure(10)
for j in range(nj-1):
    plot(angle,P[j,:])

title("CDF of scattering data")
xlabel(r"$\theta$ (degrees)",size=16)
print ("Done quad block")
name = fname+"-CDF.png"
savefig(name)


figure(6)
plot(sdepth,pvals,'b',lw=3)
xlabel(r"$z$",size=20)
ylabel(r"$\sigma_p$",size=20)
xlim([0,z0]);xticks(size=20)
ylim([0,45]),yticks(size=20)
title("Scattering profile vs. depth")
grid(True)
name = fname+"-pvsz.png"
savefig(name)


figure(2)
```

```
136 ll=where(status>0)[0] # find rays that hit submarine
137 sphi=arctan2(flx[1,ll],flx[0,ll])
138 r2=flx[0,ll]**2+flx[1,ll]**2
139 subtheta=arccos(flx[2,ll]/sqrt(r2+flx[2,ll]**2))
140 nbins1=sqrt(len(ll))
141 factor=180/pi
142 hist(subtheta*factor,arccos(linspace(1,-1,nbins1))*factor)
143 title(r"histogram of ray directions in $\theta$")
144 xlabel(r"$\theta$ (degrees)")
145 name = fname+"-raydir.png"
146 savefig(name)
147
148 # figure(6)
149 nbins=sqrt(len(ll))/10
150 x=linspace(-pi,pi,nbins)
151 y=arccos(linspace(1,-1,nbins))
152 # hist2d(sphi,subtheta,[x,y])
153 # title(r"histogram of ray directions. Note $\theta=0$ is bottom surface")
154 # xlabel(r"$\phi$")
155 # ylabel(r"$\theta$ (non-uniform bins)")
156 ll=where(status>0)[0]
157 tof=(status[ll]-tsrc[ll])
158 #figure(3)
159 #hist(tof*5e-3,100,log=True)
160 #title("Histogram of ray time of flight")
161 #xlabel(r"time ($\mu$sec)")
162 #savefig(fname+"-tof.png")
163 print ("Average time to reach submarine=%.2f time steps" % (mean(tof)))
164 print ("Stdeviation of time to reach submarine=%.2f time steps" % (std(tof↩
    )))
165 for i in range(nt):
166     mm=where(tof==i)[0]
167     channel[i]=sum(intensity[mm])
168
169 figure(4)
170 plot(arange(nt)*5e-3,channel,lw=3)
171 xlim([0,12])
172 title("Intensity Channel as predicted by simulation")
173 xlabel(r"time ($\mu$sec)",size=20)
174 grid(True)
175 name = fname+"-channel.png"
176 savefig(name)
177
178 # We finally plot a few actual ray trajectories.
179 fig5=figure(5)
180 bx=p3.Axes3D(fig5)
181 for i in range(ntraj):
```

```
182     bx.plot3D(traj[0,i,:],traj[1,i,:],traj[2,i,:])
183 title("ray trajectories")
184 grid(True)
185 name = fname+"-traj.png"
186 savefig(name)
187 
188 # Histogram at z2. Bins are equal area bins.
189 fig7=figure(14)
190 ii=find(beam2[:,0]>=0) # find all rays that crossed z2
191 locs=sqrt(linspace(0,rmax2,nbins2))
192 hist(beam2[ii,0],locs)
193 grid(True)
194 title("Histogram of rays at $z=%d$ metres" % z2)
195 xlabel(r"$r$",size=16)
196 name = fname+"-beam.png"
197 savefig(name)
198 show()
```