Table of Contents

[INTRODUCTION 2](#_Toc449385695)

[The Functions 3](#_Toc449385696)

[Sfactors Function 3](#_Toc449385697)

[FastSN 3](#_Toc449385698)

[FastTN 3](#_Toc449385699)

[FastUN 4](#_Toc449385700)

[Others 4](#_Toc449385701)

[Speed and Efficiency – Question 4 6](#_Toc449385702)

[DirectSN vs FastSN 6](#_Toc449385703)

[Time Taken 6](#_Toc449385704)

[Memory Comparison 8](#_Toc449385705)

[Number of Operations 9](#_Toc449385706)

[Theoretical Maximum Speed 14](#_Toc449385707)

[Overheads 16](#_Toc449385708)

[Comparison of Speed 16](#_Toc449385709)

[Poisson 2D Equation 18](#_Toc449385710)

[Comparison between FastSN and exercise 2 19](#_Toc449385711)

[Storage Requirements 19](#_Toc449385712)

[Gauss() 19](#_Toc449385713)

[BGauss() 19](#_Toc449385714)

[FastSN() 19](#_Toc449385715)

[Timing Comparison 20](#_Toc449385716)

[Summary 21](#_Toc449385717)

[Contour Plot 21](#_Toc449385718)

[Appendix 22](#_Toc449385719)

[Tables 22](#_Toc449385720)

[Comparing Number of Iterations between DirectSN and FastSN 22](#_Toc449385721)

[Representative Code 23](#_Toc449385722)

[SFactors, FastSN, FastTN, FastUN 23](#_Toc449385723)

[Matrix Functions – Needed for Main functions after this 27](#_Toc449385724)

[Q2 – Demonstrating FastSN works 32](#_Toc449385725)

[Q4 – Timing DirectSN vs FastSN 33](#_Toc449385726)

[Q5 – Timing Poisson2D using FastSN 34](#_Toc449385727)

[Maple Code for finding number of operations 37](#_Toc449385728)

[Maple Code for finding coefficients 38](#_Toc449385729)

All work done in this project is my own unless stated otherwise

INTRODUCTION

This project deals with computing linear systems of equations, .

In the first section I implement functions that's solve the equations of the above form for given matrices . Namely, . N can be of the form , and

In the second section I analyse the speed and efficiency of my written code through calculations of the theoretical maximum speed and the achieved speed.

In the final section, I solve a certain case of Poisson’s equation for the 2D case using my newly created functions and compare it to the results produced in the previous exercise.

Finally, I’d like to mention that the mastery component of this exercise has been incorporated into the various sections before and thus does not have its own dedicated section. I have however highlighted mastery material wherever relevant.

The Functions

In this section I briefly talk about the functions implemented as described by the exercise sheet[[1]](#footnote-1).

## Sfactors Function

Prototype:

**double** **\***SFactors**(int** N**);**

This function takes an integer, N, as input and returns a vector of size with values depending on if 5, 3, or, 2 are a factor of N.

/mastery

If 5 is a factor,

mastery/

If 3 is a factor,

If 2 is a factor,

## FastSN

Prototype:

**int** FastSN**(double** **\***x**,** **double** **\***y**,** **double** **\***w**,** **double** **\***S**,int** N**,** **int** skip**);**

The function checks if N==2, N==3, or N==5 and explicitly calculates the values of for for these cases by minimizing the number of floating point operations through grouping and creating temporary variables to store values of operations needed more than once. If N>2 but Nmod2==0 then FastSN calls FastSN(N/2) and FastTN(N/2). It also performs an additional N-2 operations to calculate the values to be send to each of these functions.

## FastTN

Prototype:

**int** FastTN**(double** **\***x**,** **double** **\***y**,** **double** **\***w**,** **double** **\***S**,int** N**,** **int** skip**);**

Similar to FastSN, FastTN checks if N==1, N==2, N==3, or N==5 and explicitly calculates the values of for for these cases by minimizing the number of floating point operations. If N>2 but Nmod2==0 then FastTN calls FastTN(N/2) and FastUN(N/2) and performs an additional N operations to calculate the result.

## FastUN

Prototype:

**int** FastUN**(double** **\***x**,** **double** **\***y**,** **double** **\***w**,** **double** **\***S**,int** N**,** **int** skip**);**

Similar to FastSN and FastTN, FastUN checks if N==1, N==2, N==3, or N==5 and explicitly calculates the values of for for these cases by minimizing the number of floating point operations. If N>2 but Nmod2==0 then FastUN calls FastTN(N/2) twice and performs an additional 4\*N-2 operations to prepare the values and calculate the result.

## Others

Functions for directly computing the matrices, Sn, Tn, and Un are in the matrix\_functions.c file at the end.

There is also a main function titled “Q2 – demonstrating FastSN works” that does exactly as the title says. For an inputted value of N it first computes the using direct matrix multiplication such that for as we know that . It then uses FastSN to demonstrate that we get back the desired .For invalid inputs of N, it can be seen that FastSN returns -1.

A couple of examples of the output have been produced below:

/mastery

Name: Bhageria, Yadu

CID: 00733164

Course Code: M3SC

Email Address: yrb13@ic.ac.uk

Time: 12:23:46

Date: Apr 24 2016

Enter N: 5

Computed y: direct matrix multiplication by Sn\*2/N

1 | 3.07768

2 | -1.37638

3 | 0.726543

4 | -0.32492

Returned Value: 0

1 | 1

2 | 2

3 | 3

4 | 4

mastery/

Enter N: 16

Computed y: direct matrix multiplication by Sn\*2/N

1 | 10.1532

2 | -5.02734

3 | 3.29656

4 | -2.41421

5 | 1.87087

6 | -1.49661

7 | 1.2185

8 | -1

9 | 0.820679

10 | -0.668179

11 | 0.534511

12 | -0.414214

13 | 0.303347

14 | -0.198912

15 | 0.0984914

Returned Value: 0

1 | 1

2 | 2

3 | 3

4 | 4

5 | 5

6 | 6

7 | 7

8 | 8

9 | 9

10 | 10

11 | 11

12 | 12

13 | 13

14 | 14

15 | 15

Speed and Efficiency – Question 4

I start by comparing the speed of using FastSN to a direct matrix multiplication function that I refer to as the DirectSN method.

# DirectSN vs FastSN

## Time Taken

Below are the tables for the time taken by FastSN compared to the time taken by the Direct Matrix Multiply for the cases of , and .

Table 1: Time Taken - DirectSN vs FastSN for 2 times powers of 2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Power | N | DirectSN Time | FastSN Time | DvS Ratio |
| 2 | 4 | 2.00E-06 | 0.00E+00 | inf |
| 3 | 8 | 1.00E-06 | 2.00E-06 | 0.5 |
| 4 | 16 | 2.00E-06 | 2.00E-06 | 1.0 |
| 5 | 32 | 1.00E-06 | 1.00E-06 | 1.0 |
| 6 | 64 | 8.00E-06 | 3.00E-06 | 2.7 |
| 7 | 128 | 3.00E-05 | 4.00E-06 | 7.5 |
| 8 | 256 | 8.50E-05 | 6.00E-06 | 14.2 |
| 9 | 512 | 2.59E-04 | 6.00E-06 | 43.2 |
| 10 | 1024 | 9.63E-04 | 1.50E-05 | 64.2 |
| 11 | 2048 | 4.50E-03 | 5.50E-05 | 81.8 |
| 12 | 4096 | 1.66E-02 | 1.38E-04 | 120.6 |
| 13 | 8192 | 7.04E-02 | 3.38E-04 | 208.3 |
| 14 | 16384 | 2.88E-01 | 1.30E-03 | 221.4 |
| 15 | 32768 | 5.91E+00 | 2.90E-03 | 2042.4 |
| 16 | 65536 |  | 6.23E-03 |  |
| 17 | 131072 |  | 1.41E-02 |  |
| 18 | 262144 |  | 4.05E-02 |  |
| 19 | 524288 |  | 1.02E-01 |  |
| 20 | 1048576 |  | 3.23E-01 |  |

Table 2: : Time Taken - DirectSN vs FastSN for 3 times powers of 2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Power | N | DirectSN Time | FastSN Time | DvS Ratio |
| 1 | 6 | 2.00E-06 | 2.00E-06 | 1.0 |
| 2 | 12 | 2.00E-06 | 2.00E-06 | 1.0 |
| 3 | 24 | 3.00E-06 | 2.00E-06 | 1.5 |
| 4 | 48 | 5.00E-06 | 2.00E-06 | 2.5 |
| 5 | 96 | 1.70E-05 | 3.00E-06 | 5.7 |
| 6 | 192 | 6.30E-05 | 4.00E-06 | 15.8 |
| 7 | 384 | 2.35E-04 | 8.00E-06 | 29.4 |
| 8 | 768 | 1.52E-03 | 1.50E-05 | 101.2 |
| 9 | 1536 | 3.63E-03 | 3.50E-05 | 103.7 |
| 10 | 3072 | 1.47E-02 | 1.23E-04 | 119.3 |
| 11 | 6144 | 5.62E-02 | 3.38E-04 | 166.3 |
| 12 | 12288 | 2.23E-01 | 8.86E-04 | 251.9 |
| 13 | 24576 | 8.93E-01 | 2.62E-03 | 340.5 |
| 14 | 49152 | 4.39E+01 | 1.35E-02 | 3253.8 |
| 15 | 98304 |  | 1.44E-02 |  |
| 16 | 196608 |  | 4.08E-02 |  |
| 17 | 393216 |  | 1.35E-01 |  |
| 18 | 786432 |  | 3.19E-01 |  |
| 19 | 1572864 |  | 8.55E-01 |  |
| 20 | 3145728 |  | 1.99E+00 |  |

/mastery

Table 3: : Time Taken - DirectSN vs FastSN for 5 times powers of 2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Power | N | DirectSN Time | FastSN Time | DvS Ratio |
| 0 | 5 | 0.00E+00 | 0.00E+00 | inf |
| 1 | 10 | 0.00E+00 | 1.00E-06 | 0 |
| 2 | 20 | 1.00E-06 | 1.00E-06 | 1 |
| 3 | 40 | 2.00E-06 | 1.00E-06 | 2 |
| 4 | 80 | 9.00E-06 | 1.00E-06 | 9 |
| 5 | 160 | 3.50E-05 | 3.00E-06 | 11.7 |
| 6 | 320 | 1.79E-04 | 5.00E-06 | 35.8 |
| 7 | 640 | 6.21E-04 | 1.20E-05 | 51.8 |
| 8 | 1280 | 2.51E-03 | 2.20E-05 | 114.1 |
| 9 | 2560 | 9.09E-03 | 8.90E-05 | 102.2 |
| 10 | 5120 | 3.62E-02 | 2.42E-04 | 149.7 |
| 11 | 10240 | 1.56E-01 | 6.30E-04 | 246.9 |
| 12 | 20480 | 6.38E-01 | 1.90E-03 | 336.1 |
| 13 | 40960 | 2.01E+01 | 5.88E-03 | 3418.1 |
| 14 | 81920 | 1.25E+02 | 3.30E-02 | 3782.5 |
| 15 | 163840 | 0.00E+00 | 2.50E-02 | 0 |
| 16 | 327680 | 0.00E+00 | 7.74E-02 | 0 |
| 17 | 655360 | 0.00E+00 | 2.51E-01 | 0 |
| 18 | 1310720 | 0.00E+00 | 6.84E-01 | 0 |
| 19 | 2621440 | 0.00E+00 | 1.53E+00 | 0 |
| 20 | 5242880 | 0.00E+00 | 3.65E+00 | 0 |

/mastery

For small values of N, that is when N<10, the timing program is not accurate enough to distinguish between DirectSN and FastSN.

**FastSN is twice as fast as DirectSN for .**

**FastSN is 10x faster than DirectSN for .**

FastSN is faster than DirectSN because it requires far fewer floating point operations. Tables comparing the number of operations (+/- and \* grouped separately) needed for DirectSN with the ones from FastSN next to it can be found in the appendix.

## Memory Comparison

The memory requirements for the two methods are very different and thus can also lead to performance increases in using FastSn for large N.

Storing the matrix grows on an order of and becomes too large to store on the RAM after – approximately – when . The size of the matrix is

which has a space requirement of

Table 4: Size of Sn matrix for N [[[2]](#footnote-2)]

|  |  |
| --- | --- |
| N | Size of Sn Matrix (gb) |
| 8192 | 0.499877937 |
| 16384 | 1.999755867 |
| 32768 | 7.999511726 |
| 65536 | 31.99902344 |
| 131072 | 127.9980469 |

In comparison the memory requirement for FastSN is the space required for SFactors (size N/2) and the matrix w needed for workings (size N-1)

This is of the order and clearly grows much more slowly and thus we can apply it to much larger vectors . The table below gives some numbers for the same range as the one for DirectSN and shows how it requires significantly less space.

Table 5: Size of memory needed for solving FastSN for powers of 2

|  |  |
| --- | --- |
| **N** | **Size of Storage needed (gb)** |
| 8192 | 9.15453E-05 |
| 16384 | 0.000183098 |
| 32768 | 0.000366203 |
| 65536 | 0.000732414 |
| 131072 | 0.001464836 |

Now before examining the theoretical maximum speed, I discuss the number of floating point operations FastSN takes.

# Number of Operations

Number of operations needed to do when we try to minimize operations would be 0 as

Number of operations needed to do when we try to minimize operations would be 3 as

Here we save one floating point operation by storing the value of in a temporary variable and using it for both and . Also multiplication to 1 is not necessary.

Similarly, the values for calculating the minimized number of operations for have been calculated and put in the table below (refer to the code in the appendix if needed):

Table 6: Minimized number of +- and \* operations needed

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| N | FastSN +- | FastSN \* | FastSN Total | FastTN +- | FastTN \* | FastTN Total | FastUN +- | FastUN \* | FastUN Total |
| 2 | 0 | 0 | 0 | 2 | 1 | 3 | 2 | 4 | 6 |
| 3 | 2 | 2 | 4 | 4 | 2 | 6 | 6 | 6 | 12 |
| 5 | 8 | 8 | 16 | 12 | 8 | 20 | 20 | 18 | 38 |

As expressed in the first section, the total number of operations FastSN, FastTn, and FastUN take depends recursively on themselves for N/2 and then N/4 and so on and so forth until N==2, or N==3 or N==5. Look back for formulae.

The formulae for number of additions and subtractions is given below for all three functions:

And the formulae for number of multiplications is:

The tables – for 2, 3, and 5 - in the following pages give the number of +/- and \* needed for each value of N. The values that have been fitted for the given equation

were found in Maple and MATLAB and both results agree to over 8 decimal places. See the code in the appendix

Table 7: Number of Operations Required case 2 times powers of 2

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Power | N | W\_S2N +- | W\_S2N \* | W\_S2N | W\_T2N +- | W\_T2N \* | W\_T2N | W\_U2N +- | W\_U2N \* | W\_U2N |
| 1 | 2 | 0 | 0 | 0 | 2 | 1 | 3 | 2 | 4 | 6 |
| 2 | 4 | 4 | 1 | 5 | 8 | 5 | 13 | 10 | 10 | 20 |
| 3 | 8 | 18 | 6 | 24 | 26 | 15 | 41 | 30 | 26 | 56 |
| 4 | 16 | 58 | 21 | 79 | 72 | 41 | 113 | 82 | 62 | 144 |
| 5 | 32 | 160 | 62 | 222 | 186 | 103 | 289 | 206 | 146 | 352 |
| 6 | 64 | 408 | 165 | 573 | 456 | 249 | 705 | 498 | 334 | 832 |
| 7 | 128 | 990 | 414 | 1404 | 1082 | 583 | 1665 | 1166 | 754 | 1920 |
| 8 | 256 | 2326 | 997 | 3323 | 2504 | 1337 | 3841 | 2674 | 1678 | 4352 |
| 9 | 512 | 5340 | 2334 | 7674 | 5690 | 3015 | 8705 | 6030 | 3698 | 9728 |
| 10 | 1024 | 12052 | 5349 | 17401 | 12744 | 6713 | 19457 | 13426 | 8078 | 21504 |
| 11 | 2048 | 26842 | 12062 | 38904 | 28218 | 14791 | 43009 | 29582 | 17522 | 47104 |
| 12 | 4096 | 59154 | 26853 | 86007 | 61896 | 32313 | 94209 | 64626 | 37774 | 102400 |
| 13 | 8192 | 129240 | 59166 | 188406 | 134714 | 70087 | 204801 | 140174 | 81010 | 221184 |
| 14 | 16384 | 280336 | 129253 | 409589 | 291272 | 151097 | 442369 | 302194 | 172942 | 475136 |
| 15 | 32768 | 604374 | 280350 | 884724 | 626234 | 324039 | 950273 | 648078 | 367730 | 1015808 |
| 16 | 65536 | 1296142 | 604389 | 1900531 | 1339848 | 691769 | 2031617 | 1383538 | 779150 | 2162688 |
| 17 | 131072 | 2767060 | 1296158 | 4063218 | 2854458 | 1470919 | 4325377 | 2941838 | 1645682 | 4587520 |
| 18 | 262144 | 5883660 | 2767077 | 8650737 | 6058440 | 3116601 | 9175041 | 6233202 | 3466126 | 9699328 |
| 19 | 524288 | 12466386 | 5883678 | 18350064 | 12815930 | 6582727 | 19398657 | 13165454 | 7281778 | 20447232 |
| 20 | 1048576 | 26330890 | 12466405 | 38797295 | 27029960 | 13864505 | 40894465 | 27729010 | 15262606 | 42991616 |

Fitting W\_S2N +-, W\_S2N \*, W\_S2N to gives

|  |  |  |  |
| --- | --- | --- | --- |
| A2 +/- | 1.33336449 | B2 +/- | -1.55619325 |
| A2 \* | 0.66666341 | B2 \* | -1.44437825 |
| A2 | 2.00002790 | B2 | -3.00057149 |

Table 8: Number of Operations Required case 3 times powers of 2

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Power | N | W\_S3N +- | W\_S3N \* | W\_S3N | W\_T3N +- | W\_T3N \* | W\_T3N | W\_U3N +- | W\_U3N \* | W\_U3N |
| 1 | 3 | 2 | 2 | 4 | 4 | 2 | 6 | 6 | 6 | 12 |
| 2 | 6 | 10 | 4 | 14 | 16 | 8 | 24 | 18 | 16 | 34 |
| 3 | 12 | 36 | 12 | 48 | 46 | 24 | 70 | 54 | 40 | 94 |
| 4 | 24 | 104 | 36 | 140 | 124 | 64 | 188 | 138 | 96 | 234 |
| 5 | 48 | 274 | 100 | 374 | 310 | 160 | 470 | 342 | 224 | 566 |
| 6 | 96 | 678 | 260 | 938 | 748 | 384 | 1132 | 810 | 512 | 1322 |
| 7 | 192 | 1616 | 644 | 2260 | 1750 | 896 | 2646 | 1878 | 1152 | 3030 |
| 8 | 384 | 3748 | 1540 | 5288 | 4012 | 2048 | 6060 | 4266 | 2560 | 6826 |
| 9 | 768 | 8526 | 3588 | 12114 | 9046 | 4608 | 13654 | 9558 | 5632 | 15190 |
| 10 | 1536 | 19106 | 8196 | 27302 | 20140 | 10240 | 30380 | 21162 | 12288 | 33450 |
| 11 | 3072 | 42316 | 18436 | 60752 | 44374 | 22528 | 66902 | 46422 | 26624 | 73046 |
| 12 | 6144 | 92832 | 40964 | 133796 | 96940 | 49152 | 146092 | 101034 | 57344 | 158378 |
| 13 | 12288 | 202058 | 90116 | 292174 | 210262 | 106496 | 316758 | 218454 | 122880 | 341334 |
| 14 | 24576 | 436894 | 196612 | 633506 | 453292 | 229376 | 682668 | 469674 | 262144 | 731818 |
| 15 | 49152 | 939336 | 425988 | 1365324 | 972118 | 491520 | 1463638 | 1004886 | 557056 | 1561942 |
| 16 | 98304 | 2009756 | 917508 | 2927264 | 2075308 | 1048576 | 3123884 | 2140842 | 1179648 | 3320490 |
| 17 | 196608 | 4281670 | 1966084 | 6247754 | 4412758 | 2228224 | 6640982 | 4543830 | 2490368 | 7034198 |
| 18 | 393216 | 9087642 | 4194308 | 13281950 | 9349804 | 4718592 | 14068396 | 9611946 | 5242880 | 14854826 |
| 19 | 786432 | 19223876 | 8912900 | 28136776 | 19748182 | 9961472 | 29709654 | 20272470 | 11010048 | 31282518 |
| 20 | 1572864 | 40544920 | 18874372 | 59419292 | 41593516 | 20971520 | 62565036 | 42642090 | 23068672 | 65710762 |

Fitting W\_S3N +-, W\_S3N \*, W\_S3N to gives

|  |  |  |  |
| --- | --- | --- | --- |
| A3 +/- | 1.33335400 | B3 +/- | -1.66927409 |
| A3 \* | 0.66666341 | B3 \* | -1.72318864 |
| A3 | 2.00002790 | B3 | -3.39246273 |

Table 9: Number of Operations Required case 5 times powers of 2 (Mastery Material)

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Power | N | W\_S5N +- | W\_S5N \* | W\_S5N | W\_T5N +- | W\_T5N \* | W\_T5N | W\_U5N +- | W\_U5N \* | W\_U5N |
| 1 | 5 | 8 | 8 | 16 | 12 | 8 | 20 | 20 | 18 | 38 |
| 2 | 10 | 28 | 16 | 44 | 42 | 26 | 68 | 42 | 36 | 78 |
| 3 | 20 | 88 | 42 | 130 | 104 | 62 | 166 | 122 | 92 | 214 |
| 4 | 40 | 230 | 104 | 334 | 266 | 154 | 420 | 286 | 204 | 490 |
| 5 | 80 | 574 | 258 | 832 | 632 | 358 | 990 | 690 | 468 | 1158 |
| 6 | 160 | 1364 | 616 | 1980 | 1482 | 826 | 2308 | 1582 | 1036 | 2618 |
| 7 | 320 | 3164 | 1442 | 4606 | 3384 | 1862 | 5246 | 3602 | 2292 | 5894 |
| 8 | 640 | 7186 | 3304 | 10490 | 7626 | 4154 | 11780 | 8046 | 5004 | 13050 |
| 9 | 1280 | 16090 | 7458 | 23548 | 16952 | 9158 | 26110 | 17810 | 10868 | 28678 |
| 10 | 2560 | 35600 | 16616 | 52216 | 37322 | 20026 | 57348 | 39022 | 23436 | 62458 |
| 11 | 5120 | 78040 | 36642 | 114682 | 81464 | 43462 | 124926 | 84882 | 50292 | 135174 |
| 12 | 10240 | 169742 | 80104 | 249846 | 176586 | 93754 | 270340 | 183406 | 107404 | 290810 |
| 13 | 20480 | 366806 | 173858 | 540664 | 380472 | 201158 | 581630 | 394130 | 228468 | 622598 |
| 14 | 40960 | 788236 | 375016 | 1163252 | 815562 | 429626 | 1245188 | 842862 | 484236 | 1327098 |
| 15 | 81920 | 1685716 | 804642 | 2490358 | 1740344 | 913862 | 2654206 | 1794962 | 1023092 | 2818054 |
| 16 | 163840 | 3589898 | 1718504 | 5308402 | 3699146 | 1936954 | 5636100 | 3808366 | 2155404 | 5963770 |
| 17 | 327680 | 7616722 | 3655458 | 11272180 | 7835192 | 4092358 | 11927550 | 8053650 | 4529268 | 12582918 |
| 18 | 655360 | 16107272 | 7747816 | 23855088 | 16544202 | 8621626 | 25165828 | 16981102 | 9495436 | 26476538 |
| 19 | 1310720 | 33962192 | 16369442 | 50331634 | 34836024 | 18117062 | 52953086 | 35709842 | 19864692 | 55574534 |
| 20 | 2621440 | 71419654 | 34486504 | 105906158 | 73167306 | 37981754 | 111149060 | 74914926 | 41477004 | 116391930 |

Fitting W\_S3N +-, W\_S3N \*, W\_S3N to gives

|  |  |  |  |
| --- | --- | --- | --- |
| A5 +/- | 1.33334808 | B5 +/- | -1.18511451 |
| A5 \* | 0.66666222 | B5 \* | -1.05896701 |
| A5 | 2.00001030 | B5 | -2.24408152 |

# Theoretical Maximum Speed

The fastest my computer can compute the problem will be if there were no overheads and it simply had to compute the floating point operations. So the fastest – or minimum – time will be

I measure theoretical maximum speed, in megaflops, as

This is equal to 2300 megaflops for my computer with a 2.3Ghz CPU. Also the real speed of my functions will be

Below are tables table include the time taken, real speed (in megaflops), and ratio between real speed and theoretical maximum speed for DirectSN and FastSN.

Table 10: Comapring Speeds of DirectSN and FastSN for 2 times powers of 2

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| N | DirectSN Time | FastSN Time | DvS Ratio | DirectOps | FastOps | DirectSN Speed | FastSN Speed | DirectSN Ratio | FastSN Ratio |
| 4 | 2.00E-06 | 0.00E+00 | inf | 15 | 5 | 7.50 | #DIV/0! | 0.00 | #DIV/0! |
| 8 | 1.00E-06 | 2.00E-06 | 0.5 | 91 | 24 | 91.00 | 12.00 | 0.04 | 0.01 |
| 16 | 2.00E-06 | 2.00E-06 | 1.0 | 435 | 79 | 217.50 | 39.50 | 0.09 | 0.02 |
| 32 | 1.00E-06 | 1.00E-06 | 1.0 | 1891 | 222 | 1891.00 | 222.00 | 0.82 | 0.10 |
| 64 | 8.00E-06 | 3.00E-06 | 2.7 | 7875 | 573 | 984.38 | 191.00 | 0.43 | 0.08 |
| 128 | 3.00E-05 | 4.00E-06 | 7.5 | 32131 | 1404 | 1071.03 | 351.00 | 0.47 | 0.15 |
| 256 | 8.50E-05 | 6.00E-06 | 14.2 | 129795 | 3323 | 1527.00 | 553.83 | 0.66 | 0.24 |
| 512 | 2.59E-04 | 6.00E-06 | 43.2 | 521731 | 7674 | 2014.41 | 1279.00 | 0.88 | 0.56 |
| 1024 | 9.63E-04 | 1.50E-05 | 64.2 | 2092035 | 17401 | 2172.41 | 1160.07 | 0.94 | 0.50 |
| 2048 | 4.50E-03 | 5.50E-05 | 81.8 | 8378371 | 38904 | 1863.10 | 707.35 | 0.81 | 0.31 |
| 4096 | 1.66E-02 | 1.38E-04 | 120.6 | 33533955 | 86007 | 2014.90 | 623.24 | 0.88 | 0.27 |
| 8192 | 7.04E-02 | 3.38E-04 | 208.3 | 134176771 | 188406 | 1905.60 | 557.41 | 0.83 | 0.24 |
| 16384 | 2.88E-01 | 1.30E-03 | 221.4 | 536788995 | 409589 | 1866.18 | 315.31 | 0.81 | 0.14 |
| 32768 | 5.91E+00 | 2.90E-03 | 2042.4 | 2147319811 | 884724 | 363.05 | 305.50 | 0.16 | 0.13 |
| 65536 |  | 6.23E-03 |  | 8589606915 | 1900531 |  | 305.16 |  | 0.13 |
| 131072 |  | 1.41E-02 |  | 34359083011 | 4063218 |  | 288.07 |  | 0.13 |
| 262144 |  | 4.05E-02 |  | 1.37438E+11 | 8650737 |  | 213.56 |  | 0.09 |
| 524288 |  | 1.02E-01 |  | 5.49753E+11 | 18350064 |  | 180.72 |  | 0.08 |
| 1048576 |  | 3.23E-01 |  | 2.19902E+12 | 38797295 |  | 120.22 |  | 0.05 |

Table 11: Comapring Speeds of DirectSN and FastSN for 3 times powers of 2

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| N | DirectSN Time | FastSN Time | DvS Ratio | DirectOps | FastOps | DirectSN Speed | FastSN Speed | DirectSN Ratio | FastSN Ratio |
| 6 | 2.00E-06 | 2.00E-06 | 1.0 | 45 | 1.40E+01 | 22.50 | 7.00 | 0.01 | 0.00 |
| 12 | 2.00E-06 | 2.00E-06 | 1.0 | 231 | 4.80E+01 | 115.50 | 24.00 | 0.05 | 0.01 |
| 24 | 3.00E-06 | 2.00E-06 | 1.5 | 1035 | 1.40E+02 | 345.00 | 70.00 | 0.15 | 0.03 |
| 48 | 5.00E-06 | 2.00E-06 | 2.5 | 4371 | 3.74E+02 | 874.20 | 187.00 | 0.38 | 0.08 |
| 96 | 1.70E-05 | 3.00E-06 | 5.7 | 17955 | 9.38E+02 | 1056.18 | 312.67 | 0.46 | 0.14 |
| 192 | 6.30E-05 | 4.00E-06 | 15.8 | 72771 | 2.26E+03 | 1155.10 | 565.00 | 0.50 | 0.25 |
| 384 | 2.35E-04 | 8.00E-06 | 29.4 | 292995 | 5.29E+03 | 1246.79 | 661.00 | 0.54 | 0.29 |
| 768 | 1.52E-03 | 1.50E-05 | 101.2 | 1175811 | 1.21E+04 | 774.58 | 807.60 | 0.34 | 0.35 |
| 1536 | 3.63E-03 | 3.50E-05 | 103.7 | 4710915 | 2.73E+04 | 1297.42 | 780.06 | 0.56 | 0.34 |
| 3072 | 1.47E-02 | 1.23E-04 | 119.3 | 18859011 | 6.08E+04 | 1285.64 | 493.92 | 0.56 | 0.21 |
| 6144 | 5.62E-02 | 3.38E-04 | 166.3 | 75466755 | 1.34E+05 | 1342.28 | 395.85 | 0.58 | 0.17 |
| 12288 | 2.23E-01 | 8.86E-04 | 251.9 | 301928451 | 2.92E+05 | 1353.03 | 329.77 | 0.59 | 0.14 |
| 24576 | 8.93E-01 | 2.62E-03 | 340.5 | 1207836675 | 6.34E+05 | 1352.17 | 241.52 | 0.59 | 0.11 |
| 49152 | 4.39E+01 | 1.35E-02 | 3253.8 | 4831592451 | 1.37E+06 | 110.06 | 101.20 | 0.05 | 0.04 |
| 98304 |  | 1.44E-02 |  | 19326861315 | 2.93E+06 |  | 203.71 |  | 0.09 |
| 196608 |  | 4.08E-02 |  | 77308428291 | 6.25E+06 |  | 153.11 |  | 0.07 |
| 393216 |  | 1.35E-01 |  | 3.09236E+11 | 1.33E+07 |  | 98.47 |  | 0.04 |
| 786432 |  | 3.19E-01 |  | 1.23695E+12 | 2.81E+07 |  | 88.31 |  | 0.04 |
| 1572864 |  | 8.55E-01 |  | 4.94779E+12 | 5.94E+07 |  | 69.47 |  | 0.03 |

/mastery

Table 12: Comapring Speeds of DirectSN and FastSN for 5 times powers of 2

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| N | DirectSN Time | FastSN Time | DvS Ratio | DirectOps | FastOps | DirectSN Speed | FastSN Speed | DirectSN Ratio | FastSN Ratio |
| 5 | 0.00E+00 | 0.00E+00 | inf | 28 | 16 | #DIV/0! | #DIV/0! | #DIV/0! | #DIV/0! |
| 10 | 0.00E+00 | 1.00E-06 | 0 | 153 | 44 | #DIV/0! | 44.00 | #DIV/0! | 0.02 |
| 20 | 1.00E-06 | 1.00E-06 | 1 | 703 | 130 | 703.00 | 130.00 | 0.31 | 0.06 |
| 40 | 2.00E-06 | 1.00E-06 | 2 | 3003 | 334 | 1501.50 | 334.00 | 0.65 | 0.15 |
| 80 | 9.00E-06 | 1.00E-06 | 9 | 12403 | 832 | 1378.11 | 832.00 | 0.60 | 0.36 |
| 160 | 3.50E-05 | 3.00E-06 | 11.7 | 50403 | 1980 | 1440.09 | 660.00 | 0.63 | 0.29 |
| 320 | 1.79E-04 | 5.00E-06 | 35.8 | 203203 | 4606 | 1135.21 | 921.20 | 0.49 | 0.40 |
| 640 | 6.21E-04 | 1.20E-05 | 51.8 | 816003 | 10490 | 1314.01 | 874.17 | 0.57 | 0.38 |
| 1280 | 2.51E-03 | 2.20E-05 | 114.1 | 3270403 | 23548 | 1302.95 | 1070.36 | 0.57 | 0.47 |
| 2560 | 9.09E-03 | 8.90E-05 | 102.2 | 13094403 | 52216 | 1440.05 | 586.70 | 0.63 | 0.26 |
| 5120 | 3.62E-02 | 2.42E-04 | 149.7 | 52403203 | 114682 | 1446.12 | 473.89 | 0.63 | 0.21 |
| 10240 | 1.56E-01 | 6.30E-04 | 246.9 | 209664003 | 249846 | 1348.15 | 396.58 | 0.59 | 0.17 |
| 20480 | 6.38E-01 | 1.90E-03 | 336.1 | 838758403 | 540664 | 1314.77 | 284.86 | 0.57 | 0.12 |
| 40960 | 2.01E+01 | 5.88E-03 | 3418.1 | 3355238403 | 1163252 | 167.05 | 197.97 | 0.07 | 0.09 |
| 81920 | 1.25E+02 | 3.30E-02 | 3782.5 | 13421363203 | 2490358 |  | 75.36 |  | 0.03 |
| 163840 | 0.00E+00 | 2.50E-02 | 0 | 53686272003 | 5308402 |  | 212.33 |  | 0.09 |
| 327680 | 0.00E+00 | 7.74E-02 | 0 | 2.14747E+11 | 11272180 |  | 145.67 |  | 0.06 |
| 655360 | 0.00E+00 | 2.51E-01 | 0 | 8.5899E+11 | 23855088 |  | 94.88 |  | 0.04 |
| 1310720 | 0.00E+00 | 6.84E-01 | 0 | 3.43597E+12 | 50331634 |  | 73.60 |  | 0.03 |

mastery/

### Overheads

From the tables it is clearly evident that the FastSN function has a large percentage of overheads. This is evident in its comparatively lower real speed to theoretical speed ratio when compared to the DirectSN method. But because of FastSN’s significantly lower number of operations even though the overheads for FastSN grow as N grows, the total time taken is much less than that for the DirectSN method.

For DirectSN, the overheads are significantly lower than for FastSN as DirectSN does not call any functions recursively. For extremely large N, the speed drops off because of overheads in accessing memory that is far apart (see N>32768). For very small N, the timing method is not accurate enough to give a reasonable value for speed.

For FastSN, the overheads reduce to an approximate minimum of 50% at and then start to increase again to almost account for 97% of the time taken. This is because for small N our timing method is not accurate enough and after N>1000 the number of recursive calls to FastSN, FastTN, and FastUN increase exponentially whereas the floating point operations do not.

### Comparison of Speed against Exercise 2

Find the actual speed of my code in the tables above and also in the tables below which have the speed for Gauss and BGauss in 1D. This is because in the 1D case we have a (N-1) by (N-1) matrix acting upon a vector of size N-1 as is the case with DirectSN and FastSN and so a direct comparison can be made.

It can be seen that the DirectSN method has the highest speed as it does not manipulate the data whatsoever. It does instead have significantly more floating point operations than the other methods and thus is actually slower.

Gaussian Elimination had the lowest speed due to the way it was implemented with loops within loops within loops and various checks to minimize redundant floating point operations when the matrix contained zero values.

FastSN starts out with a lot more speed than BGauss (1D) but for it starts to lag behind as the speed of BGauss continues to increase the speed of FastSN continues to fall post N>1024.

It is worth mentioning here that the speeds of the Gauss and BGauss methods here depend heavily on the choice of problem being solved and thus are not equivalent to FastSN in the same way DirectSN is. Furthermore, Gauss and BGauss allocate memory within the functions whereas DirectSN and FastSN do not. More comparison FastSN and Gauss and BGauss is made in the next section where I use FastSN to solve Poisson’s Equation in 2D.

Table 13: Speed Comparison between this Exercise and the previous one (in MegaFlops)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | DirectSN | FastSN | Gauss 1D | Bgauss 1D |
| 8 | 91.00 | 12.00 | 0.02 | 54.00 |
| 16 | 217.50 | 39.50 | 0.54 | 63.00 |
| 32 | 1891.00 | 222.00 | 1.28 | 90.00 |
| 64 | 984.38 | 191.00 | 2.36 | 79.71 |
| 128 | 1071.03 | 351.00 | 2.36 | 189.00 |
| 256 | 1527.00 | 553.83 | 10.71 | 190.50 |
| 512 | 2014.41 | 1279.00 | 19.55 | 208.64 |
| 1024 | 2172.41 | 1160.07 | 41.11 | 187.71 |
| 2048 | 1863.10 | 707.35 | 82.90 | 191.81 |
| 4096 | 2014.90 | 623.24 | 157.25 | 15.16 |
| 8192 | 1905.60 | 557.41 | 297.47 | 257.73 |
| 16384 | 1866.18 | 315.31 | 317.05 | 287.40 |
| 32768 | 363.05 | 305.50 | 285.21 | 326.93 |
| 65536 |  | 305.16 | 145.78 | 351.28 |
| 131072 |  | 288.07 |  | 387.65 |
| 262144 |  | 213.56 |  | 350.93 |
| 524288 |  | 180.72 |  | 264.51 |
| 1048576 |  | 120.22 |  | 386.93 |

Poisson 2D Equation

In this section I solve Poisson’s 2D Equations for a given case (the same as Exercise 2) using the FastSN function developed. The Exercise sheet demonstrates how to compute the result for a given vector – that represents a 2D grid – and I do not elaborate on this here. See the code in the appendix for more information.

The required table has been produced below.

Note FD Maxval is the max value found by using the finite difference approximation and A Maxval is the max value found using the analytic formula.

Table 14: Results for Poisson 2D using FastSN (results in red are for the mastery section)

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Pow of 2** | **N** | **FD Maxval** | **x pos** | **y pos** | **A Maxval** | **x pos** | **y pos** | **CPU Time** | **Wall Time** |
| 5\*2^ 0 | 5 | 1.604242 | 0.400000 | 0.400000 | 1.320501 | 0.400000 | 0.400000 | 0.001075 | 0.000501 |
| 3\*2^ 1 | 6 | 1.722959 | 0.333333 | 0.333333 | 1.533368 | 0.333333 | 0.333333 | 0.000568 | 0.000311 |
| 2\*2^ 2 | 8 | 1.670186 | 0.375000 | 0.375000 | 1.570453 | 0.375000 | 0.375000 | 0.000622 | 0.000299 |
| 5\*2^ 1 | 10 | 1.732380 | 0.400000 | 0.400000 | 1.678310 | 0.400000 | 0.400000 | 0.000553 | 0.000244 |
| 3\*2^ 2 | 12 | 1.645431 | 0.416667 | 0.416667 | 1.604003 | 0.416667 | 0.416667 | 0.000641 | 0.000304 |
| 2\*2^ 3 | 16 | 1.701806 | 0.375000 | 0.375000 | 1.682483 | 0.375000 | 0.375000 | 0.000824 | 0.000406 |
| 5\*2^ 2 | 20 | 1.696736 | 0.400000 | 0.400000 | 1.683578 | 0.400000 | 0.400000 | 0.000694 | 0.000311 |
| 3\*2^ 3 | 24 | 1.708222 | 0.375000 | 0.375000 | 1.698920 | 0.375000 | 0.375000 | 0.000632 | 0.000318 |
| 2\*2^ 4 | 32 | 1.710522 | 0.375000 | 0.375000 | 1.705460 | 0.375000 | 0.375000 | 0.000652 | 0.000260 |
| 5\*2^ 3 | 40 | 1.711597 | 0.375000 | 0.375000 | 1.708308 | 0.375000 | 0.375000 | 0.000721 | 0.000319 |
| 3\*2^ 4 | 48 | 1.712184 | 0.375000 | 0.375000 | 1.709921 | 0.375000 | 0.375000 | 0.000748 | 0.000290 |
| 2\*2^ 5 | 64 | 1.714084 | 0.390625 | 0.390625 | 1.712793 | 0.390625 | 0.390625 | 0.000949 | 0.000339 |
| 5\*2^ 4 | 80 | 1.716023 | 0.387500 | 0.387500 | 1.715200 | 0.387500 | 0.387500 | 0.000874 | 0.000325 |
| 3\*2^ 5 | 96 | 1.716748 | 0.385417 | 0.385417 | 1.716177 | 0.385417 | 0.385417 | 0.001182 | 0.000361 |
| 2\*2^ 6 | 128 | 1.717014 | 0.382812 | 0.382812 | 1.716693 | 0.382812 | 0.382812 | 0.001856 | 0.000541 |
| 5\*2^ 5 | 160 | 1.716831 | 0.381250 | 0.381250 | 1.716626 | 0.381250 | 0.381250 | 0.005181 | 0.001164 |
| 3\*2^ 6 | 192 | 1.716998 | 0.385417 | 0.385417 | 1.716856 | 0.385417 | 0.385417 | 0.007826 | 0.001689 |
| 2\*2^ 7 | 256 | 1.717155 | 0.382812 | 0.382812 | 1.717075 | 0.382812 | 0.382812 | 0.014309 | 0.002976 |
| 5\*2^ 6 | 320 | 1.717180 | 0.384375 | 0.384375 | 1.717129 | 0.384375 | 0.384375 | 0.019123 | 0.006608 |
| 3\*2^ 7 | 384 | 1.717181 | 0.382812 | 0.382812 | 1.717146 | 0.382812 | 0.382812 | 0.030332 | 0.005767 |
| 2\*2^ 8 | 512 | 1.717190 | 0.382812 | 0.382812 | 1.717170 | 0.382812 | 0.382812 | 0.073792 | 0.013635 |
| 5\*2^ 7 | 640 | 1.717203 | 0.384375 | 0.384375 | 1.717190 | 0.384375 | 0.384375 | 0.122907 | 0.017088 |
| 3\*2^ 8 | 768 | 1.717221 | 0.384115 | 0.384115 | 1.717212 | 0.384115 | 0.384115 | 0.170650 | 0.029563 |
| 2\*2^ 9 | 1024 | 1.717233 | 0.383789 | 0.383789 | 1.717228 | 0.383789 | 0.383789 | 0.329476 | 0.064349 |
| 5\*2^ 8 | 1280 | 1.717235 | 0.383594 | 0.383594 | 1.717231 | 0.383594 | 0.383594 | 0.543779 | 0.097314 |
| 3\*2^ 9 | 1536 | 1.717234 | 0.383464 | 0.383464 | 1.717231 | 0.383464 | 0.383464 | 0.809890 | 0.142808 |
| 2\*2^10 | 2048 | 1.717235 | 0.383789 | 0.383789 | 1.717234 | 0.383789 | 0.383789 | 2.568084 | 0.398695 |
| 5\*2^ 9 | 2560 | 1.717236 | 0.383594 | 0.383594 | 1.717235 | 0.383594 | 0.383594 | 3.679709 | 0.570100 |
| 3\*2^10 | 3072 | 1.717235 | 0.383789 | 0.383789 | 1.717235 | 0.383789 | 0.383789 | 6.750945 | 0.956419 |
| 2\*2^11 | 4096 | 1.717236 | 0.383545 | 0.383545 | 1.717236 | 0.383545 | 0.383545 | 16.447932 | 2.173878 |
| 5\*2^10 | 5120 | 1.717236 | 0.383594 | 0.383594 | 1.717236 | 0.383594 | 0.383594 | 23.236140 | 3.113227 |
| 3\*2^11 | 6144 | 1.717237 | 0.383626 | 0.383626 | 1.717236 | 0.383626 | 0.383626 | 40.935915 | 5.365346 |
| 2\*2^12 | 8192 | 1.717237 | 0.383667 | 0.383667 | 1.717237 | 0.383667 | 0.383667 | 87.824416 | 11.609539 |
| 5\*2^11 | 10240 | 1.717237 | 0.383691 | 0.383691 | 1.717236 | 0.383691 | 0.383691 | 142.746318 | 19.978539 |
| 3\*2^12 | 12288 | 1.717237 | 0.383626 | 0.383626 | 1.717237 | 0.383626 | 0.383626 | 223.163206 | 31.381505 |
| 2\*2^13 | 16384 | 1.717237 | 0.383667 | 0.383667 | 1.717237 | 0.383667 | 0.383667 | 461.699637 | 64.533413 |

I was able to go up to while keeping my computation time under 5 minutes by utilizing openmp and potentially could go even higher within this time constraint but was limited by memory allocation issues. The value of N obviously depends on the specifications of the computer running the program.

For , my vector would be of size (N-1)^2 containing doubles which means in terms of memory it would require of memory and thus becomes a limiting factor.

Time increases as a function of N by roughly where .

## Comparison between FastSN and exercise 2

Below I look at the storage and time requirements for the FastSN, Gauss and BGauss functions in solving Poisson’s Equation in 2D.

I would like to mention that theoretically FastSN also requires less floating point operations than Gauss and BGauss functions with full matrices. But in the case of the values only being on the diagonal for the Gauss and Bgauss matrices this is not necessarily true.

## Storage Requirements

There are limitations due to memory allocation.

### Gauss()

For Guass() I have that where the A matrix that I use for Gaussian elimination is of size

This grows to the order of in terms of memory required.

### BGauss()

For BGauss I have that

This grows to the order of in terms of memory required.

### FastSN()

For FastSN I perform 4 times N-1 fast sine transforms along with one direct computation on to calculate . Other than the memory required for and , this method only needs an additional vector of size N/2 for Sfactors and of size N-1 for - and also N for a vector containing values of - that is needed in FastSN for as storage space.

So memory other than and that is needed is:

This grows only to the order of 3N/2 – 5N/2 if I store the values of in a vector of size N - and thus memory needed to store is an issue long before the extra memory needed to use FastSN becomes an issue.

Below is a table that compares the memory requirements – other than storing and - for Gauss, Bgauss and FastSN. It can clearly be seen that FastSN is vastly superior in terms of storage requirements and allows us to compute Poisson Equation in the 2D case for vastly larger N than possible using the Gauss and BGauss methods.

Table 15: Memory requirements of the various methods to solve Poisson's Equation in 2D

|  |  |  |  |
| --- | --- | --- | --- |
| N | Gauss (gb) | Bgauss (gb) | FastSN (gb) |
| 4 | 6.03E-07 | 4.69E-07 | 3.73E-08 |
| 8 | 1.79E-05 | 5.48E-06 | 8.20E-08 |
| 16 | 3.77E-04 | 5.20E-05 | 1.71E-07 |
| 32 | 6.88E-03 | 4.51E-04 | 3.50E-07 |
| 64 | 1.17E-01 | 3.76E-03 | 7.08E-07 |
| 128 | 1.94E+00 | 3.06E-02 | 1.42E-06 |
| 256 | 3.15E+01 | 2.48E-01 | 2.85E-06 |
| 512 | 5.08E+02 | 1.99E+00 | 5.71E-06 |
| 1024 | 8.16E+03 | 1.60E+01 | 1.14E-05 |
| 2048 | 1.31E+05 | 1.28E+02 | 2.29E-05 |
| 4096 | 2.10E+06 | 1.02E+03 | 4.58E-05 |
| 8192 | 3.35E+07 | 8.19E+03 | 9.15E-05 |
| 16384 | 5.37E+08 | 6.55E+04 | 1.83E-04 |
| 32768 | 8.59E+09 | 5.24E+05 | 3.66E-04 |
| 65536 | 1.37E+11 | 4.19E+06 | 7.32E-04 |
| 131072 | 2.20E+12 | 3.36E+07 | 1.46E-03 |
| 262144 | 3.52E+13 | 2.68E+08 | 2.93E-03 |
| 524288 | 5.63E+14 | 2.15E+09 | 5.86E-03 |
| 1048576 | 9.01E+15 | 1.72E+10 | 1.17E-02 |

## Timing Comparison

Here I compare the time taken for the various values of N to solve Poisson’s Equation in 2D. Below is a table for comparison.

Table 16: Time requirements of the various methods to solve Poisson's Equation in 2D

|  |  |  |  |
| --- | --- | --- | --- |
| N | Gauss | Bgauss | FastSN |
| 8 | 0.002583 | 0.000008 | 0.000299 |
| 16 | 0.012182 | 0.000075 | 0.000406 |
| 32 | 0.048872 | 0.001058 | 0.000260 |
| 64 | 0.358834 | 0.011202 | 0.000339 |
| 128 | 11.360894 | 0.150360 | 0.000541 |
| 256 | 466.246308 | 3.639611 | 0.002976 |
| 512 |  | 20.807215 | 0.013635 |
| 640 |  | 41.278272 | 0.017088 |
| 768 |  | 75.250111 | 0.029563 |
| 1024 |  |  | 0.064349 |
| 2048 |  |  | 0.398695 |
| 4096 |  |  | 2.173878 |
| 8192 |  |  | 11.609539 |
| 16384 |  |  | 64.533413 |

Apart from the cases of N=8 and N=16, FastSN is significantly faster than the Gauss and BGauss methods implemented in exercise 2, where BGauss itself is also significantly faster than the Gauss method.

All the methods here use OMP to parallelize the code. The difference between FastSN and Gauss and BGauss functions is that for FastSN, the process of doing fast sine transforms on the N-1 rows and N-1 columns has been parallelized and not FastSN itself. For Gauss and BGauss the functions themselves have been parallelized.

It is worth noting here that FastSN can be parallelized by declaring a global variable that only parallelizes the first and second - perhaps even the third depending on number of processors - branch of the recursive calls in FastSN to FastTn and FastUN but this did not lead to a speed increase in comparison to just parallelizing the N-1 fast sine transforms that are performed 4 times.

## Summary

FastSN is faster and requires less memory than the Gauss and BGauss methods implemented in exercise 2.

## Contour Plot

Below is a contour plot of produced using FastSN.

Appendix

# Tables

## Comparing Number of Iterations between DirectSN and FastSN

Table 17: For 2 times powers of 2

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Power | N | DirectSN +- | DirectSN \* | DirectSN Total | FastSN +- | FastSN \* | FastSN Total | DvS Ratio |
| 1 | 2 | 0 | 1 | 1 | 0 | 0 | 0 |  |
| 2 | 4 | 6 | 9 | 15 | 4 | 1 | 5 | 3.00 |
| 3 | 8 | 42 | 49 | 91 | 18 | 6 | 24 | 3.79 |
| 4 | 16 | 210 | 225 | 435 | 58 | 21 | 79 | 5.51 |
| 5 | 32 | 930 | 961 | 1891 | 160 | 62 | 222 | 8.52 |
| 6 | 64 | 3906 | 3969 | 7875 | 408 | 165 | 573 | 13.74 |
| 7 | 128 | 16002 | 16129 | 32131 | 990 | 414 | 1404 | 22.89 |
| 8 | 256 | 64770 | 65025 | 129795 | 2326 | 997 | 3323 | 39.06 |
| 9 | 512 | 260610 | 261121 | 521731 | 5340 | 2334 | 7674 | 67.99 |
| 10 | 1024 | 1045506 | 1046529 | 2092035 | 12052 | 5349 | 17401 | 120.22 |
| 11 | 2048 | 4188162 | 4190209 | 8378371 | 26842 | 12062 | 38904 | 215.36 |
| 12 | 4096 | 16764930 | 16769025 | 33533955 | 59154 | 26853 | 86007 | 389.90 |
| 13 | 8192 | 67084290 | 67092481 | 134176771 | 129240 | 59166 | 188406 | 712.17 |
| 14 | 16384 | 268386306 | 268402689 | 536788995 | 280336 | 129253 | 409589 | 1310.56 |
| 15 | 32768 | 1073643522 | 1073676289 | 2147319811 | 604374 | 280350 | 884724 | 2427.11 |
| 16 | 65536 | 4294770690 | 4294836225 | 8589606915 | 1296142 | 604389 | 1900531 | 4519.58 |
| 17 | 131072 | 17179475970 | 17179607041 | 34359083011 | 2767060 | 1296158 | 4063218 | 8456.13 |
| 18 | 262144 | 68718690306 | 68718952449 | 1.37438E+11 | 5883660 | 2767077 | 8650737 | 15887.39 |
| 19 | 524288 | 2.74876E+11 | 2.74877E+11 | 5.49753E+11 | 12466386 | 5883678 | 18350064 | 29959.20 |
| 20 | 1048576 | 1.09951E+12 | 1.09951E+12 | 2.19902E+12 | 26330890 | 12466405 | 38797295 | 56679.67 |

Table 18: For 3 times powers of 2

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Power** | **N** | DirectSN +- | DirectSN \* | DirectSN Total | FastSN +- | FastSN \* | FastSN Total | DvS Ratio |
| 1 | 3 | 2 | 4 | 6 | 2 | 2 | 4 | 1.50 |
| 2 | 6 | 20 | 25 | 45 | 10 | 4 | 14 | 3.21 |
| 3 | 12 | 110 | 121 | 231 | 36 | 12 | 48 | 4.81 |
| 4 | 24 | 506 | 529 | 1035 | 104 | 36 | 140 | 7.39 |
| 5 | 48 | 2162 | 2209 | 4371 | 274 | 100 | 374 | 11.69 |
| 6 | 96 | 8930 | 9025 | 17955 | 678 | 260 | 938 | 19.14 |
| 7 | 192 | 36290 | 36481 | 72771 | 1616 | 644 | 2260 | 32.20 |
| 8 | 384 | 146306 | 146689 | 292995 | 3748 | 1540 | 5288 | 55.41 |
| 9 | 768 | 587522 | 588289 | 1175811 | 8526 | 3588 | 12114 | 97.06 |
| 10 | 1536 | 2354690 | 2356225 | 4710915 | 19106 | 8196 | 27302 | 172.55 |
| 11 | 3072 | 9427970 | 9431041 | 18859011 | 42316 | 18436 | 60752 | 310.43 |
| 12 | 6144 | 37730306 | 37736449 | 75466755 | 92832 | 40964 | 133796 | 564.04 |
| 13 | 12288 | 150958082 | 150970369 | 301928451 | 202058 | 90116 | 292174 | 1033.39 |
| 14 | 24576 | 603906050 | 603930625 | 1207836675 | 436894 | 196612 | 633506 | 1906.59 |
| 15 | 49152 | 2415771650 | 2415820801 | 4831592451 | 939336 | 425988 | 1365324 | 3538.79 |
| 16 | 98304 | 9663381506 | 9663479809 | 19326861315 | 2009756 | 917508 | 2927264 | 6602.36 |
| 17 | 196608 | 38654115842 | 38654312449 | 77308428291 | 4281670 | 1966084 | 6247754 | 12373.80 |
| 18 | 393216 | 1.54618E+11 | 1.54618E+11 | 3.09236E+11 | 9087642 | 4194308 | 13281950 | 23282.40 |
| 19 | 786432 | 6.18473E+11 | 6.18474E+11 | 1.23695E+12 | 19223876 | 8912900 | 28136776 | 43961.92 |
| 20 | 1572864 | 2.4739E+12 | 2.4739E+12 | 4.94779E+12 | 40544920 | 18874372 | 59419292 | 83269.16 |

Table 19: For 5 times powers of 2

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Power | N | DirectSN +- | DirectSN \* | DirectSN Total | FastSN +- | FastSN \* | FastSN Total | DvS Ratio |
| 1 | 5 | 12 | 16 | 28 | 8 | 8 | 16 | 1.75 |
| 2 | 10 | 72 | 81 | 153 | 28 | 16 | 44 | 3.48 |
| 3 | 20 | 342 | 361 | 703 | 88 | 42 | 130 | 5.41 |
| 4 | 40 | 1482 | 1521 | 3003 | 230 | 104 | 334 | 8.99 |
| 5 | 80 | 6162 | 6241 | 12403 | 574 | 258 | 832 | 14.91 |
| 6 | 160 | 25122 | 25281 | 50403 | 1364 | 616 | 1980 | 25.46 |
| 7 | 320 | 101442 | 101761 | 203203 | 3164 | 1442 | 4606 | 44.12 |
| 8 | 640 | 407682 | 408321 | 816003 | 7186 | 3304 | 10490 | 77.79 |
| 9 | 1280 | 1634562 | 1635841 | 3270403 | 16090 | 7458 | 23548 | 138.88 |
| 10 | 2560 | 6545922 | 6548481 | 13094403 | 35600 | 16616 | 52216 | 250.77 |
| 11 | 5120 | 26199042 | 26204161 | 52403203 | 78040 | 36642 | 114682 | 456.94 |
| 12 | 10240 | 104826882 | 104837121 | 209664003 | 169742 | 80104 | 249846 | 839.17 |
| 13 | 20480 | 419368962 | 419389441 | 838758403 | 366806 | 173858 | 540664 | 1551.35 |
| 14 | 40960 | 1677598722 | 1677639681 | 3355238403 | 788236 | 375016 | 1163252 | 2884.36 |
| 15 | 81920 | 6710640642 | 6710722561 | 13421363203 | 1685716 | 804642 | 2490358 | 5389.33 |
| 16 | 163840 | 26843054082 | 26843217921 | 53686272003 | 3589898 | 1718504 | 5308402 | 10113.45 |
| 17 | 327680 | 1.07373E+11 | 1.07374E+11 | 2.14747E+11 | 7616722 | 3655458 | 11272180 | 19051.04 |
| 18 | 655360 | 4.29495E+11 | 4.29495E+11 | 8.5899E+11 | 16107272 | 7747816 | 23855088 | 36008.68 |
| 19 | 1310720 | 1.71798E+12 | 1.71798E+12 | 3.43597E+12 | 33962192 | 16369442 | 50331634 | 68266.56 |
| 20 | 2621440 | 6.87194E+12 | 6.87194E+12 | 1.37439E+13 | 71419654 | 34486504 | 105906158 | 129774.16 |

It can clearly be seen here that FastSN takes far fewer floating point operations compared to a direct matrix multiply and the ratio of number of operations keeps growing for larger N.

# Representative Code

Below is all the code for the exercise

## SFactors, FastSN, FastTN, FastUN

*/\* Yadu Bhageria, 00733164, M3SC \*/*

*#include <stdio.h>*

*#include <stdlib.h>*

*#include <math.h>*

*/\* incase Pi is not defined \*/*

*#ifndef M\_PI*

*# define M\_PI acos(-1.)*

*#endif*

*/\* -Functions-implemented-in-current-file------------------------------------- \*/*

**double** **\***SFactors**(int);**

**int** FastSN**(double** **\*,** **double** **\*,** **double** **\*,** **double** **\*,int,** **int);**

**int** FastTN**(double** **\*,** **double** **\*,** **double** **\*,** **double** **\*,int,** **int);**

**int** FastUN**(double** **\*,** **double** **\*,** **double** **\*,** **double** **\*,int,** **int);**

*/\* --------------------------------------------------------------------------- \*/*

**double** **\***SFactors**(int** N**){**

*/\* Yadu Bhageria, 00733164, M3SC \*/*

**double** numerator **=** **1.,** denominator**;**

**double** **\***S**;**

**if** **(**N**%5==0){**

denominator **=** **10.;**

S **=** **(double** **\*)** malloc**((**N**/2+1)\*sizeof(double));**

S**[1]** **=** sin**(**M\_PI**\*1./5.);**

S**[2]** **=** sin**(**M\_PI**\*2./5.);**

**for** **(int** i**=1;** i**<=(-2+**N**/2);** i**++){**

**if** **(**numerator**/**denominator **>=** **1./2.){**

numerator **=** **1.;**

denominator **\*=** **2.;**

**}**

S**[**i**+2]** **=** sin**(**M\_PI**\***numerator**/**denominator**);**

numerator **+=** **2.;**

**}**

**}** **else** **if** **(**N**%3==0** **||** N**%2==0){**

denominator **=** **(**N**%3==0)** **?** **3.** **:** **2.;**

S **=** **(double** **\*)** malloc**((**N**/2+1)\*sizeof(double));**

**for** **(int** i**=1;** i**<=**N**/2;** i**++){**

**if** **(**numerator**/**denominator **>=** **1./2.){**

numerator **=** **1.;**

denominator **\*=** **2.;**

**}**

S**[**i**]** **=** sin**(**M\_PI**\***numerator**/**denominator**);**

numerator **+=** **2.;**

**}**

**}** **else** **{** *//When N is not divisible by 2, 3, or 5*

S **=** **(double** **\*)** malloc**(2\*sizeof(double));**

S**[1]** **=** **0.;**

**}**

S**[0]** **=** **1.;** *//Given*

**return** S**;**

**}**

*/\* --------------------------------------------------------------------------- \*/*

**int** FastSN**(double** **\***x**,** **double** **\***y**,** **double** **\***w**,** **double** **\***S**,int** N**,** **int** skip**){**

*/\* Yadu Bhageria, 00733164, M3SC \*/*

*#ifdef DEBUG*

printf**(**"in Sn: N=%3d, skip=%3d \n"**,**N**,**skip**);**

*#endif*

**int** returnval **=** **0;**

**if** **(**N**<=1){** *//Check N is sensible*

returnval **=** **-1;**

**}** **else** **if** **(**N**==2){**

x**[**skip**]** **=** y**[**skip**];**

**}** **else** **if** **(**N**==3){**

x**[** skip**]** **=** S**[1]** **\*** **(**y**[**skip**]** **+** y**[2\***skip**]);**

x**[2\***skip**]** **=** S**[1]** **\*** **(**y**[**skip**]** **-** y**[2\***skip**]);**

**}** **else** **if** **(**N**==5){**

**double** t1**,** t2**;**

t1 **=** S**[1]\***y**[** skip**]** **+** S**[2]\***y**[3\***skip**];**

t2 **=** S**[2]\***y**[2\***skip**]** **+** S**[1]\***y**[4\***skip**];**

x**[** skip**]** **=** t1 **+** t2**;**

x**[4\***skip**]** **=** t1 **-** t2**;**

t1 **=** S**[2]\***y**[** skip**]** **-** S**[1]\***y**[3\***skip**];**

t2 **=** S**[1]\***y**[2\***skip**]** **-** S**[2]\***y**[4\***skip**];**

x**[2\***skip**]** **=** t1 **+** t2**;**

x**[3\***skip**]** **=** t1 **-** t2**;**

**}** **else** **if** **(**N **%** **2** **==** **0){**

**int** e1**,** e2**;**

w**[(**N**-1)\***skip**]** **=** y**[**skip**\***N**/2];**

**for** **(int** i**=1;** i**<**N**/2;** i**++){**

w**[**skip**\*(2\*(**i**-1)+1)]** **=** y**[**skip**\***i**]** **+** y**[**skip**\*(**N**-**i**)];**

w**[**skip**\*2\***i**]** **=** y**[**skip**\***i**]** **-** y**[**skip**\*(**N**-**i**)];**

**}**

e1 **=** FastSN**(**x**,**w**,**y**,**S**,**N**/2,2\***skip**);**

**if** **(**e1**!=0)** **return** e1**;**

e2 **=** FastTN**(**x**-**skip**,**w**-**skip**,**y**-**skip**,**S**,**N**/2,2\***skip**);**

**if** **(**e1**!=0)** **return** e2**;**

**}** **else{**

returnval **=** **-1;**

**}**

**return** returnval**;**

**}**

*/\* --------------------------------------------------------------------------- \*/*

**int** FastTN**(double** **\***x**,** **double** **\***y**,** **double** **\***w**,** **double** **\***S**,int** N**,** **int** skip**){**

*/\* Yadu Bhageria, 00733164, M3SC \*/*

*#ifdef DEBUG*

printf**(**"in Tn: N=%3d, skip=%3d \n"**,**N**,**skip**);**

*#endif*

**int** returnval **=** **0;**

**if** **(**N**<=0){** *//Check N is sensible*

returnval **=** **-1;**

**}** **else** **if** **(**N**==1){**

x**[**skip**]** **=** y**[**skip**];**

**}** **else** **if** **(**N**==2){**

**double** t1 **=** S**[1]\***y**[**skip**];**

x**[**skip**]** **=** t1 **+** y**[2\***skip**];**

x**[2\***skip**]** **=** t1 **-** y**[2\***skip**];**

**}** **else** **if** **(**N**==3){**

**double** t1 **=** S**[2]\***y**[**skip**]** **+** y**[3\***skip**];**

**double** t2 **=** S**[1]\***y**[2\***skip**];**

x**[**skip**]** **=** t1 **+** t2**;**

x**[2\***skip**]** **=** y**[**skip**]** **-** y**[3\***skip**];**

x**[3\***skip**]** **=** t1 **-** t2**;**

**}** **else** **if** **(**N**==5){**

**double** t1**,** t2**;**

t1 **=** S**[3]\***y**[** skip**]** **+** S**[4]\***y**[3\***skip**]** **+** y**[5\***skip**];**

t2 **=** S**[1]\***y**[2\***skip**]** **+** S**[2]\***y**[4\***skip**];**

x**[** skip**]** **=** t1 **+** t2**;**

x**[5\***skip**]** **=** t1 **-** t2**;**

t1 **=** S**[4]\***y**[** skip**]** **+** S**[3]\***y**[3\***skip**]** **-** y**[5\***skip**];**

t2 **=** S**[2]\***y**[2\***skip**]** **-** S**[1]\***y**[4\***skip**];**

x**[2\***skip**]** **=** t1 **+** t2**;**

x**[4\***skip**]** **=** t1 **-** t2**;**

x**[3\***skip**]** **=** y**[**skip**]** **-** y**[3\***skip**]** **+** y**[5\***skip**];**

**}** **else** **if** **(**N **%** **2** **==** **0){**

**int** e1 **=** FastTN**(**w**,**y**,**x**,**S**,**N**/2,2\***skip**);**

**if** **(**e1**!=0)** **return** e1**;**

**int** e2 **=** FastUN**(**w**-**skip**,**y**-**skip**,**x**-**skip**,**S**,**N**/2,2\***skip**);**

**if** **(**e2**!=0)** **return** e2**;**

**for** **(int** i**=1;** i**<=**N**/2;** i**++){**

x**[** i**\***skip**]** **=** w**[(2\***i**-1)\***skip**]** **+** w**[2\***i**\***skip**];**

x**[(**N**+1-**i**)\***skip**]** **=** w**[(2\***i**-1)\***skip**]** **-** w**[2\***i**\***skip**];**

**}**

**}** **else{**

returnval **=** **-1;**

**}**

**return** returnval**;**

**}**

*/\* --------------------------------------------------------------------------- \*/*

**int** FastUN**(double** **\***x**,** **double** **\***y**,** **double** **\***w**,** **double** **\***S**,int** N**,** **int** skip**){**

*/\* Yadu Bhageria, 00733164, M3SC \*/*

*#ifdef DEBUG*

printf**(**"in Un: N=%3d, skip=%3d \n"**,**N**,**skip**);**

*#endif*

**int** returnval **=** **0;**

**if** **(**N**<=0){** *//Check N is sensible*

returnval **=** **-1;**

**}** **else** **if** **(**N**==1){**

x**[**skip**]** **=** S**[1]\***y**[**skip**];**

**}** **else** **if** **(**N**==2){**

x**[** skip**]** **=** S**[2]\***y**[**skip**]** **+** S**[3]\***y**[2\***skip**];**

x**[2\***skip**]** **=** S**[3]\***y**[**skip**]** **-** S**[2]\***y**[2\***skip**];**

**}** **else** **if** **(**N**==3){**

**double** t1 **=** S**[4]\***y**[2\***skip**];**

x**[** skip**]** **=** S**[3]\***y**[**skip**]** **+** S**[5]\***y**[3\***skip**]** **+** t1**;**

x**[2\***skip**]** **=** S**[4]\*(**y**[**skip**]** **+** y**[2\***skip**]** **-** y**[3\***skip**]);**

x**[3\***skip**]** **=** S**[5]\***y**[**skip**]** **+** S**[3]\***y**[3\***skip**]** **-** t1**;**

**}** **else** **if** **(**N**==5){**

**double** t1 **=** S**[7]\***y**[3\***skip**];**

x**[**skip**]** **=** S**[5]\***y**[**skip**]** **+** S**[6]\***y**[2\***skip**]** **+** t1 **+** S**[8]\***y**[4\***skip**]** **+** S**[9]\***y**[5\***skip**];**

x**[2\***skip**]** **=** S**[6]\***y**[**skip**]** **+** S**[9]\***y**[2\***skip**]** **+** t1 **-** S**[5]\***y**[4\***skip**]** **-** S**[8]\***y**[5\***skip**];**

x**[3\***skip**]** **=** S**[7]\*(**y**[**skip**]** **+** y**[2\***skip**]** **-** y**[3\***skip**]** **-** y**[4\***skip**]** **+** y**[5\***skip**]);**

x**[4\***skip**]** **=** S**[8]\***y**[**skip**]** **-** S**[5]\***y**[2\***skip**]** **-** t1 **+** S**[9]\***y**[4\***skip**]** **-** S**[6]\***y**[5\***skip**];**

x**[5\***skip**]** **=** S**[9]\***y**[**skip**]** **-** S**[8]\***y**[2\***skip**]** **+** t1 **-** S**[6]\***y**[4\***skip**]** **+** S**[5]\***y**[5\***skip**];**

**}** **else** **if** **(**N **%** **2** **==** **0){**

**int** i**;**

**for** **(**i**=1;** i**<**N**/2;** i**++){**

w**[**skip**\*2\***i**]** **=** y**[(**N**+1-2\***i**)\***skip**]** **-** y**[(**N**-2\***i**)\***skip**];**

w**[**skip**\*(2\*(**i**-1)+1)]** **=** y**[(2\***i**)\***skip**]** **+** y**[(2\***i**+1)\***skip**];**

**}**

w**[** N**\***skip**]** **=** y**[**skip**];**

w**[(**N**-1)\***skip**]** **=** y**[**N**\***skip**];**

**int** e1 **=** FastTN**(**y**-**skip**,**w**,**x**-**skip**,**S**,**N**/2,2\***skip**);**

**if** **(**e1**!=0)** **return** e1**;**

**int** e2 **=** FastTN**(**y**,**w**-**skip**,**x**,**S**,**N**/2,2\***skip**);**

**if** **(**e2**!=0)** **return** e2**;**

**for** **(**i**=1;** i**<=**N**/2;** i**++){**

**if** **(**i**%2==0){**

x**[** i**\***skip**]** **=** **-**S**[**N**+**i**-1]\***y**[(2\***i**-1)\***skip**]** **+** S**[2\***N**-**i**]\***y**[(2\***i**)\***skip**];**

x**[(**N**+1-**i**)\***skip**]** **=** **-**S**[2\***N**-**i**]\***y**[(2\***i**-1)\***skip**]** **-** S**[**N**+**i**-1]\***y**[(2\***i**)\***skip**];**

**}** **else{**

x**[** i**\***skip**]** **=** S**[**N**+**i**-1]\***y**[(2\***i**-1)\***skip**]** **+** S**[2\***N**-**i**]\***y**[(2\***i**)\***skip**];**

x**[(**N**+1-**i**)\***skip**]** **=** S**[2\***N**-**i**]\***y**[(2\***i**-1)\***skip**]** **-** S**[**N**+**i**-1]\***y**[(2\***i**)\***skip**];**

**}**

**}**

**}** **else{**

returnval **=** **-1;**

**}**

**return** returnval**;**

**}**

## Matrix Functions – Needed for Main functions after this

Note that these functions are not needed for SFactors, FastSN, FastTN, and FastUN. They are however needed for my main functions

*#include <stdio.h>*

*#include <stdlib.h>*

*#include <math.h>*

*#include <stdbool.h>*

*#include <float.h>*

*/\*incase Pi is not defined \*/*

*#ifndef M\_PI*

*# define M\_PI acos(-1.)*

*#endif*

*/\* --------------------------------------------------------------------------- \*/*

**void** print\_statements**(){**

*/\* Bhageria, Yadu, 00733164, M3SC \*/*

printf**(** " Name: Bhageria, Yadu"**);**

printf**(**"\n CID: 00733164"**);**

printf**(**"\n Course Code: M3SC"**);**

printf**(**"\nEmail Address: yrb13@ic.ac.uk"**);**

printf**(**"\n Time: %s "**,**\_\_TIME\_\_**);**

printf**(**"\n Date: %s "**,**\_\_DATE\_\_**);**

printf**(**"\n \n"**);**

**}**

*/\* --------------------------------------------------------------------------- \*/*

**void** print\_matrix**(double** **\*\***A**,** **int** N**,** **int** M**){**

*/\* Yadu Bhageria, 00733164, M3SC \*/*

*/\**

*Extremely useful for debugging*

*\*/*

**for(int** i**=1;** i**<**N**+1;** i**++){**

**for(int** j**=1;** j**<**M**+1;** j**++){**

printf**(**"%6.3g "**,** A**[**i**][**j**]);**

**}**

printf**(**"\n"**);**

**}**

printf**(**"\n"**);**

**}**

*/\* --------------------------------------------------------------------------- \*/*

**void** print\_vector**(double** **\***x**,** **int** N**){**

*/\* Yadu Bhageria, 00733164, M3SC \*/*

**for** **(int** i**=1;** i**<**N**+1;** i**++){**

printf**(**"%3d | %12.6g\n"**,**i**,** x**[**i**]);**

**}**

printf**(**"\n"**);**

**}**

*/\* --------------------------------------------------------------------------- \*/*

**void** print\_2dvector**(double** **\***x**,** **int** N**,** **int** M**){**

*/\* Yadu Bhageria, 00733164, M3SC \*/*

**for** **(int** j**=1;** j**<**M**+1;** j**++){**

**for** **(int** i**=1;** i**<**N**+1;** i**++){**

printf**(**"%6.3g, "**,** x**[(**N**)\*(**i**-1)+**j**]);**

**}**

printf**(**"\n"**);**

**}**

printf**(**"\n"**);**

**}**

*/\* --------------------------------------------------------------------------- \*/*

**double** **\*\***allocate\_zero\_matrix**(int** N**,** **int** M**){**

*/\* Yadu Bhageria, 00733164, M3SC \*/*

*//The safe approach; better sizes over 2^27 in size*

**double** **\*\*** A**;**

A**=(double** **\*\*)** malloc**((**N**+1)\*sizeof(double** **\*));**

**for(int** i**=1;** i**<**N**+1;** i**++){**

A**[**i**]=(double** **\*)** calloc**((**M**+1),sizeof(double));**

**}**

**return** A**;**

*/\**

*//The cautious approach:*

*double \*\*A; int i;*

*A = (double \*\*)malloc((N+1)\*sizeof(double \*));*

*A[0] = (double \*)calloc((N\*M+1),sizeof(double));*

*A[1] = A[0];*

*for (i=2; i<=N; i++) A[i] = A[i-1]+M;*

*return A;*

*\*/*

**}**

*/\* --------------------------------------------------------------------------- \*/*

**double** **\***allocate\_zero\_vector**(int** N**){**

*/\* Yadu Bhageria, 00733164, M3SC \*/*

**double** **\***X**;**

X **=** **(double** **\*)** calloc**((**N**+1),sizeof(double));**

**return** X**;**

**}**

*/\* --------------------------------------------------------------------------- \*/*

**int** maxvalpos\_vec**(double** **\*** X**,** **int** N**){**

*/\* Yadu Bhageria, 00733164, M3SC \*/*

**double** maxvalue**=**X**[1];**

**int** xpos **=** **1;**

**for(int** i**=2;** i**<**N**+1;** i**++){**

**if** **(**X**[**i**]** **>** maxvalue**){**

maxvalue **=** X**[**i**];**

xpos **=** i**;**

**}**

**}**

**return** xpos**;**

**}**

*/\* --------------------------------------------------------------------------- \*/*

**void** free\_matrix**(double** **\*\***A**,** **int** N**){**

*/\* Yadu Bhageria, 00733164, M3SC \*/*

*//for the safe approach*

**for** **(int** i**=**N**;** i**>0;** i**--)** free**(**A**[**i**]);**

*/\**

*//for the cautious approach*

*free(A[0]); free(A);*

*\*/*

**}**

*/\* --------------------------------------------------------------------------- \*/*

**void** multiply\_vec**(double** **\***X**,** **int** N**,** **double** factor**){**

**int** i**;**

**for** **(**i**=1;** i**<**N**+1;** i**++){**

X**[**i**]** **\*=** factor**;**

**}**

**}**

*/\* --------------------------------------------------------------------------- \*/*

**double** **\***make\_Yvec2D**(int** N**,** **bool** smooth**,** **double** delta**){**

**int** i**,**j**;**

**double** **\***F **=** allocate\_zero\_vector**(** **(**N**-1)\*(**N**-1)** **);**

**for** **(**j**=1;** j**<**N**;** j**++){**

**if** **((double)**j**/**N **==** **0.5** **||** **(double)**j**/**N **==** **0.25){**

**for** **(**i**=1;** i**<**N**;** i**++){**

**if** **((double)**i**/**N **==** **0.5** **||** **(double)**i**/**N **==** **0.25){**

**if** **(**smooth **==** true**){**

F**[(**N**-1)\*(**j**-1)+**i**]** **=** **-25.0\***delta**;**

**}** **else{**

F**[(**N**-1)\*(**j**-1)+**i**]** **=** **-100.0\***delta**;**

**}**

**}** **else** **if** **((double)**i**/**N **<=** **0.5** **&&** **(double)**i**/**N **>=** **0.25){**

**if** **(**smooth **==** true**){**

F**[(**N**-1)\*(**j**-1)+**i**]** **=** **-50.0\***delta**;**

**}** **else{**

F**[(**N**-1)\*(**j**-1)+**i**]** **=** **-100.0\***delta**;**

**}**

**}**

**}**

**}** **else** **if** **((double)**j**/**N **<** **0.5** **&&** **(double)**j**/**N **>** **0.25){**

**for** **(**i**=1;** i**<**N**;** i**++){**

**if** **((double)**i**/**N **==** **0.5** **||** **(double)**i**/**N **==** **0.25){**

**if** **(**smooth **==** true**){**

F**[(**N**-1)\*(**j**-1)+**i**]** **=** **-50.0\***delta**;**

**}** **else** **{**

F**[(**N**-1)\*(**j**-1)+**i**]** **=** **-100.0\***delta**;**

**}**

**}** **else** **if** **((double)**i**/**N **<=** **0.5** **&&** **(double)**i**/**N **>=** **0.25){**

F**[(**N**-1)\*(**j**-1)+**i**]** **=** **-100.0\***delta**;**

**}**

**}**

**}**

**}**

**return** F**;**

**}**

*/\* --------------------------------------------------------------------------- \*/*

**void** contour\_print**(double** **\***x**,** **int** N**){**

*/\* Yadu Bhageria, 00733164, M3SC \*/*

**int** i**,**j**;**

**int** nby32 **=** N**/32;**

**for** **(**i**=0;** i**<=32;** i**++){**

printf**(**" 0.0,"**);**

**}**

printf**(**"\n"**);**

**for** **(**i**=1;** i**<32;** i**++){**

printf**(**" 0.0,"**);**

**for** **(**j**=1;** j**<32;** j**++){**

printf**(**"%8.5f,"**,** x**[(**N**-1)\*(**i**\***nby32**-1)+**j**\***nby32**]);**

**}**

printf**(**" 0.0,\n"**);**

**}**

printf**(**"\n"**);**

**for** **(**i**=0;** i**<=32;** i**++){**

printf**(**" 0.0,"**);**

**}**

**}**

*/\* --------------------------------------------------------------------------- \*/*

**double** **\***allocate\_vector**(int** N**){**

*/\* Yadu Bhageria, 00733164, M3SC \*/*

**double** **\***X**;**

X **=** **(double** **\*)** malloc**((**N**+1)\*sizeof(double));**

**return** X**;**

**}**

*/\* --------------------------------------------------------------------------- \*/*

**double** **\*\***allocate\_matrix**(int** N**,** **int** M**){**

*/\* Yadu Bhageria, 00733164, M3SC \*/*

*//The safe approach; better sizes over 2^27 in size*

**double** **\*\*** A**;**

A**=(double** **\*\*)** malloc**((**N**+1)\*sizeof(double** **\*));**

**for(int** i**=1;** i**<**N**+1;** i**++){**

A**[**i**]=(double** **\*)** malloc**((**M**+1)\*sizeof(double));**

**}**

**return** A**;**

*/\**

*//The cautious approach:*

*double \*\*A; int i;*

*A = (double \*\*)malloc((N+1)\*sizeof(double \*));*

*A[0] = (double \*)malloc((N\*M+1)\*sizeof(double));*

*A[1] = A[0];*

*for (i=2; i<=N; i++) A[i] = A[i-1]+M;*

*return A;*

*\*/*

**}**

*/\* --------------------------------------------------------------------------- \*/*

**double** **\*\***Sn\_matrix**(int** N**){**

*/\* Yadu Bhageria, 00733164, M3SC \*/*

**int** i**,**j**;**

**double** **\*\***A **=** allocate\_matrix**(**N**-1,**N**-1);**

**for** **(**i**=1;** i**<**N**;** i**++){**

**for** **(**j**=1;** j**<**N**;** j**++){**

A**[**i**][**j**]** **=** sin**(**M\_PI**\***i**\***j**/**N**);**

**}**

**}**

**return** A**;**

**}**

*/\* --------------------------------------------------------------------------- \*/*

**double** **\*\***Tn\_matrix**(int** N**){**

*/\* Yadu Bhageria, 00733164, M3SC \*/*

**int** i**,**j**;**

**double** **\*\***A **=** allocate\_matrix**(**N**,**N**);**

**for** **(**i**=1;** i**<=**N**;** i**++){**

**for** **(**j**=1;** j**<=**N**;** j**++){**

A**[**i**][**j**]** **=** sin**((2.\***i**-1.)\***j**\***M\_PI**/(2.\***N**));**

**}**

**}**

**return** A**;**

**}**

*/\* --------------------------------------------------------------------------- \*/*

**double** **\*\***Tnt\_matrix**(int** N**){**

*/\* Yadu Bhageria, 00733164, M3SC \*/*

**int** i**,**j**;**

**double** **\*\***A **=** allocate\_matrix**(**N**,**N**);**

**for** **(**i**=1;** i**<=**N**;** i**++){**

**for** **(**j**=1;** j**<=**N**;** j**++){**

A**[**j**][**i**]** **=** sin**((2.\***i**-1.)\***j**\***M\_PI**/(2.\***N**));**

**}**

**}**

**return** A**;**

**}**

*/\* --------------------------------------------------------------------------- \*/*

**double** **\*\***Un\_matrix**(int** N**){**

*/\* Yadu Bhageria, 00733164, M3SC \*/*

**int** i**,**j**;**

**double** **\*\***A **=** allocate\_matrix**(**N**,**N**);**

**for** **(**i**=1;** i**<=**N**;** i**++){**

**for** **(**j**=1;** j**<=**N**;** j**++){**

A**[**i**][**j**]** **=** sin**((2.\***i**-1.)\*(2.\***j**-1.)\***M\_PI**/(4.\***N**));**

**}**

**}**

**return** A**;**

**}**

*/\* --------------------------------------------------------------------------- \*/*

**double** **\*\*** multiply\_square\_matrices**(double** **\*\***A**,** **double** **\*\***B**,** **int** N**){**

*/\* Yadu Bhageria, 00733164, M3SC \*/*

**double** **\*\***X **=** allocate\_zero\_matrix**(**N**,**N**);**

**for** **(int** i**=1;** i**<=**N**;** i**++){**

**for** **(int** j**=1;** j**<=**N**;** j**++){**

**for** **(int** k**=1;** k**<=**N**;** k**++){**

X**[**i**][**j**]** **+=** A**[**i**][**k**]\***B**[**k**][**j**];**

**}**

*/\* For neater printing \*/*

**if** **(**fabs**(**X**[**i**][**j**])** **<** **100.\***DBL\_EPSILON**){**

X**[**i**][**j**]** **=** **0.;**

**}**

**}**

**}**

**return** X**;**

**}**

*/\* --------------------------------------------------------------------------- \*/*

**void** multiply\_matrix\_vector**(double** **\*\***A**,** **double** **\***B**,** **double** **\***X**,** **int** N**){**

*/\* Yadu Bhageria, 00733164, M3SC \*/*

**for** **(int** i**=1;** i**<=**N**;** i**++){**

**for** **(int** k**=1;** k**<=**N**;** k**++){**

X**[**i**]** **+=** A**[**i**][**k**]\***B**[**k**];**

**}**

*/\* For neater printing \*/*

*/\*if (fabs(X[i]) < 100.\*DBL\_EPSILON){*

*X[i] = 0.;*

*}\*/*

**}**

**}**

*/\* --------------------------------------------------------------------------- \*/*

**double** **\*** add\_vectors**(double** **\***X**,** **double** **\***Y**,** **double** factor**,** **int** N**){**

*/\* Yadu Bhageria, 00733164, M3SC \*/*

**double** **\***Z **=** allocate\_vector**(**N**);**

**for** **(int** i**=1;** i**<=**N**;** i**++){**

Z**[**i**]** **=** X**[**i**]** **+** factor**\***Y**[**i**];**

**if** **(**fabs**(**Z**[**i**])<1e-10){**

Z**[**i**]** **=** **0;**

**}**

**}**

**return** Z**;**

**}**

*/\* --------------------------------------------------------------------------- \*/*

## Q2 – Demonstrating FastSN works

*#include <stdio.h>*

*#include <stdlib.h>*

*#include <math.h>*

*/\* -Functions-needed-from-other-files----------------------------------------- \*/*

**void** print\_statements**();**

**void** print\_vector**(double** **\*,** **int);**

**double** **\***SFactors**(int);**

**double** **\***allocate\_zero\_vector**(int);**

**double** **\***SFactors**(int);**

**double** **\*\***Sn\_matrix**(int);**

**void** multiply\_vec**(double** **\*,** **int,** **double);**

**void** multiply\_matrix\_vector**(double** **\*\*,** **double** **\*,** **double** **\*,** **int);**

**void** print\_vector**(double** **\*,** **int);**

**void** free\_matrix**(double** **\*\*,** **int);**

**int** FastSN**(double** **\*,** **double** **\*,** **double** **\*,** **double** **\*,int,** **int);**

*/\* --------------------------------------------------------------------------- \*/*

**int** main**(void){**

*/\* Yadu Bhageria, 00733164, M3SC \*/*

**int** N**;**

print\_statements**();**

*//Choose N*

printf**(**"Enter N: "**);**

scanf**(**"%d"**,** **&**N**);**

printf**(**"\n"**);**

*//Initialize memory and x\_i = i vector*

**double** **\***y **=** allocate\_zero\_vector**(**N**-1);**

**double** **\***xp **=** allocate\_zero\_vector**(**N**-1);**

**for** **(int** i**=1;** i**<**N**;** i**++){**

xp**[**i**]** **=** **(double)**i**;**

**}**

*//Find y vector by doing a direct matrix multiply with Sn*

**double** **\*\***Sn **=** Sn\_matrix**(**N**);**

multiply\_matrix\_vector**(**Sn**,**xp**,**y**,**N**-1);**

multiply\_vec**(**y**,**N**-1,2./**N**);**

free\_matrix**(**Sn**,**N**-1);**

*//Use FastSN on y and print out the answer x*

*//to demonstrate it is of the form x\_i = i*

**double** **\***S **=** SFactors**(4.\***N**);**

**double** **\***x **=** allocate\_zero\_vector**(**N**-1);**

**double** **\***w **=** allocate\_zero\_vector**(**N**-1);**

**int** returnval **=** FastSN**(**x**,**y**,**w**,**S**,**N**,1);**

printf**(**"Returned Value: %d\n"**,** returnval**);**

print\_vector**(**x**,**N**-1);**

*#ifdef DEBUG*

printf**(**"Computed y: direct matrix multiplication by Sn\*2/N\n"**);**

print\_vector**(**y**,**N**-1);**

printf**(**"\nS[0]=%g, S[1]=%g, S[2]=%g, S[3]=%g, S[4]=%g, S[5]=%g\n"**,** S**[0],**S**[1],**S**[2],**S**[3],**S**[4],**S**[5]);**

*#endif*

**}**

*/\* --------------------------------------------------------------------------- \*/*

## Q4 – Timing DirectSN vs FastSN

*#include <stdio.h>*

*#include <stdlib.h>*

*#include <math.h>*

*#include <time.h>*

*/\* -Functions-needed-from-other-files----------------------------------------- \*/*

**void** print\_statements**();**

**void** print\_vector**(double** **\*,** **int);**

**double** **\***SFactors**(int);**

**double** **\***allocate\_zero\_vector**(int);**

**double** **\***SFactors**(int);**

**double** **\*\***Sn\_matrix**(int);**

**void** multiply\_vec**(double** **\*,** **int,** **double);**

**double** **\*** multiply\_matrix\_vector**(double** **\*\*,** **double** **\*,** **double** **\*,** **int);**

**void** print\_vector**(double** **\*,** **int);**

**void** free\_matrix**(double** **\*\*,** **int);**

**int** FastSN**(double** **\*,** **double** **\*,** **double** **\*,** **double** **\*,int,** **int);**

*/\* --------------------------------------------------------------------------- \*/*

**int** main**(void){**

*/\* Yadu Bhageria, 00733164, M3SC \*/*

**int** N1**;**

**double** timet1**=0.,** timet2**;**

**clock\_t** start**,** end**;**

print\_statements**();**

printf**(**"Choose N (2, 3 or 5): "**);**

scanf**(**"%d"**,** **&**N1**);**

printf**(**"\n"**);**

*//Makes sure the input is valid*

**if** **(**N1**==2** **||** N1**==3** **||** N1**==5){**

printf**(**" i | N | DirectSN | FastSN | Ratio |\n"**);**

printf**(**"---|------------|------------|------------|---------|\n"**);**

**for** **(int** i**=0;** i**<=20;** i**++){**

*//Initialize memory and variables needed*

**int** N **=** **(int)** N1**\***pow**(2,**i**);**

**double** **\***x **=** allocate\_zero\_vector**(**N**);**

**double** **\***xp **=** allocate\_zero\_vector**(**N**);**

**for** **(int** i**=1;** i**<=**N**;** i**++){**

xp**[**i**]** **=** **(double)**i**;**

**}**

*//Direct Matrix Multiply (upto a specified power)*

**if** **(**i**<15){**

*//Initialize Memory needed*

**double** **\*\***Sn **=** Sn\_matrix**(**N**);**

*//Timing*

start **=** clock**();**

multiply\_matrix\_vector**(**Sn**,**xp**,**x**,**N**-1);**

end **=** clock**();**

free\_matrix**(**Sn**,**N**-1);**

timet1 **=** **((double)**end**-**start**)/**CLOCKS\_PER\_SEC**;**

**}** **else** **{**timet1 **=** **0.;}**

*//Initialize Memory needed for FastSN*

**double** **\***S **=** SFactors**(**N**);**

**double** **\***w **=** allocate\_zero\_vector**(**N**);**

*//Timing FastSN*

start **=** clock**();**

FastSN**(**x**,**xp**,**w**,**S**,**N**,1);**

end **=** clock**();**

timet2 **=** **((double)**end**-**start**)/**CLOCKS\_PER\_SEC**;**

*//print data*

printf**(**"%3d|%12d| %10.4e | %10.4e | %7.1f |\n"**,** i**+1,** N**,** timet1**,** timet2**,** timet1**/**timet2**);**

*//Free up used memory*

free**(**x**);**free**(**xp**);**free**(**w**);**free**(**S**);**

**}**

**}** **else** **{**

printf**(**"Please enter a valid value for N\n"**);**

**}**

**}**

*/\* --------------------------------------------------------------------------- \*/*

## Q5 – Timing Poisson2D using FastSN

*#include <stdio.h>*

*#include <stdlib.h>*

*#include <math.h>*

*#include <time.h>*

*#include <sys/time.h>*

*#include <stdbool.h>*

*#include <omp.h>*

*/\*incase Pi is not defined \*/*

*#ifndef M\_PI*

*# define M\_PI acos(-1.)*

*#endif*

*/\* --Data-structure-for-moving-wanted-data-between-functions------------------ \*/*

**typedef** **struct** poisson2d\_data **{**

**double** maxval**;**

**double** xpos**;**

**double** ypos**;**

**double** cpu\_time**;**

**double** wall\_time**;**

**}** poisson2d\_data**;**

*/\* -Functions-needed-from-other-files----------------------------------------- \*/*

**void** print\_statements**();**

**double** **\***allocate\_zero\_vector**(int);**

**double** **\***SFactors**(int);**

**double** **\***make\_Yvec2D**(int,** **bool,** **double);**

**int** FastSN**(double** **\*,** **double** **\*,** **double** **\*,** **double** **\*,int,** **int);**

**int** maxvalpos\_vec**(double** **\*,** **int);**

**void** contour\_print**(double** **\*,** **int);**

*/\* -Functions-implemented-in-current-file------------------------------------- \*/*

**double** **\***cos\_vec**(int);**

poisson2d\_data sn\_poisson2d**(int,** **bool);**

*/\* --------------------------------------------------------------------------- \*/*

**int** main**(void){**

*/\* Yadu Bhageria, 00733164, M3SC \*/*

**int** N**,** i**,** j**;**

poisson2d\_data Phi\_FD**,** Phi\_a**;**

print\_statements**();**

printf**(**"Pow of 2| N | FD Maxval | x pos | y pos | A Maxval | x pos | y pos | CPU Time | Wall Time |\n"**);**

printf**(**"--------|------------|------------|------------|------------|------------|------------|------------|------------|------------|\n"**);**

**int** N1**[]** **=** **{5,6,8};**

**for** **(**i**=0;** i**<=15;** i**++){**

**for** **(**j**=0;** j**<3;** j**++){**

*//Solve Poisson 2D for the exact and non exact cases*

N **=** **(int)** N1**[**j**]\***pow**(2,**i**);**

Phi\_a **=** sn\_poisson2d**(**N**,** true**);**

Phi\_FD **=** sn\_poisson2d**(**N**,** false**);**

*//Print out acquired data*

printf**(**" %1d\*2^%2d |%11d |"**,**N1**[**j**]/(int)**pow**(2,**j**),** i**+**j**,** N**);**

printf**(**" %10.6f | %10.6f | %10.6f |"**,** Phi\_FD**.**maxval**,** Phi\_FD**.**xpos**,** Phi\_FD**.**ypos**);**

printf**(**" %10.6f | %10.6f | %10.6f | %10.6f | %10.6f |\n"**,** Phi\_a**.**maxval**,** Phi\_a**.**xpos**,** Phi\_a**.**ypos**,** Phi\_a**.**cpu\_time**,** Phi\_a**.**wall\_time**);**

**}**

**}**

**}**

*/\* --------------------------------------------------------------------------- \*/*

**double** **\***cos\_vec**(int** N**){**

*/\* Yadu Bhageria, 00733164, M3SC \*/*

**double** **\***C **=** allocate\_zero\_vector**(**N**);**

**for** **(int** i**=0;** i**<=**N**;** i**++){**

C**[**i**]** **=** cos**(**i**\***M\_PI**/**N**);**

**}**

**return** C**;**

**}**

*/\* --------------------------------------------------------------------------- \*/*

poisson2d\_data sn\_poisson2d**(int** N**,** **bool** analytic**){**

*/\* Yadu Bhageria, 00733164, M3SC \*/*

**int** i**,** j**,** rv**,** returnval **=** **0;**

**clock\_t** start**,** end**;**

**struct** timeval walltime\_s**,** walltime\_e**;**

poisson2d\_data data**;**

*//Initialize Variables and Memory for Data*

**double** delta **=** **1./**N**;**

**int** grid\_size **=** **(**N**-1)\*(**N**-1);**

**double** **\***S **=** SFactors**(**N**);**

**double** **\***x **=** allocate\_zero\_vector**(**grid\_size**);**

**double** **\***w **=** allocate\_zero\_vector**(**grid\_size**);**

**double** **\***y **=** make\_Yvec2D**(**N**,**true**,-1.);**

**double** **\***C **=** cos\_vec**(**N**);**

*//Start Timing*

start **=** clock**();**

gettimeofday**(&**walltime\_s**,**NULL**);**

*//Transform along rows of the grid*

*//This is only done from i = N/4 to i = N/2 as all other rows are zero and*

*// so the transform would be zero too.*

*#pragma omp parallel for*

**for** **(**i**=**N**-1-**N**/2;** i**<**N**-**N**/4;** i**++){**

rv **=** FastSN**(**x**-**i**,** y**-**i**,** w**-**i**,** S**,** N**,** N**-1);**

*#ifdef DEBUG*

returnval **=** **(**rv**==-1** **||** returnval**==-1)** **?** **-1** **:** **0;**

*#endif*

**}**

*//Transform along columns of the grid*

*#pragma omp parallel for*

**for** **(**j**=0;** j**<**N**-1;** j**++){**

**int** shift **=** j**\*(**N**-1);**

rv **=** FastSN**(**y**+**shift**,** x**+**shift**,** w**+**shift**,** S**,** N**,** **1);**

*#ifdef DEBUG*

returnval **=** **(**rv**==-1** **||** returnval**==-1)** **?** **-1** **:** **0;**

*#endif*

**}**

*//Compute Phi based whether Analytic or FD*

**if** **(**analytic**){**

**double** constant **=** **4.** **/** **(**N**\***N **\*** M\_PI**\***M\_PI**);**

*#pragma omp parallel for*

**for** **(**i**=1;** i**<**N**;** i**++){**

**for** **(**j**=1;** j**<**N**;** j**++){**

x**[(**N**-1)\*(**i**-1)+**j**]** **=** constant **\*** y**[(**N**-1)\*(**i**-1)+**j**]** **/** **(**i**\***i **+** j**\***j**);**

**}**

**}**

**}** **else** **{**

**double** constant **=** **4.** **/** **(**N**\***N**);**

*#pragma omp parallel for*

**for** **(**i**=1;** i**<**N**;** i**++){**

**for** **(**j**=1;** j**<**N**;** j**++){**

x**[(**N**-1)\*(**i**-1)+**j**]** **=** delta**\***delta **\*** constant **\*** y**[(**N**-1)\*(**i**-1)+**j**]** **/** **(4.** **-** **2.\***C**[**i**]** **-** **2.\***C**[**j**]);**

**}**

**}**

**}**

*//Transform back along columns of the grid*

*#pragma omp parallel for*

**for** **(**j**=0;** j**<**N**-1;** j**++){**

**int** shift **=** j**\*(**N**-1);**

rv **=** FastSN**(**y**+**shift**,** x**+**shift**,** w**+**shift**,** S**,** N**,** **1);**

*#ifdef DEBUG*

returnval **=** **(**rv**==-1** **||** returnval**==-1)** **?** **-1** **:** **0;**

*#endif*

**}**

*//Transform back along rows of the grid*

*//This is only done from i = N/4 to i = N/2 as the first transform was*

*// only done from i=N/4 to i=N/2. Also all values outside of this be zero*

*#pragma omp parallel for*

**for** **(**i**=0;** i**<**N**-1;** i**++){**

rv **=** FastSN**(**x**-**i**,** y**-**i**,** w**-**i**,** S**,** N**,** N**-1);**

*#ifdef DEBUG*

returnval **=** **(**rv**==-1** **||** returnval**==-1)** **?** **-1** **:** **0;**

*#endif*

**}**

*//Finish timing*

end **=** clock**();**

gettimeofday**(&**walltime\_e**,**NULL**);**

*//Compute the maxval, its position and time taken*

**int** maxvalpos **=** maxvalpos\_vec**(**x**,**grid\_size**);**

**int** maxval\_ypos **=** maxvalpos **%** **(**N**-1);**

**int** maxval\_xpos **=** **1** **+** **(**maxvalpos**-**maxval\_ypos**)/(**N**-1);**

data**.**maxval **=** x**[**maxvalpos**];**

data**.**ypos **=** **(double)** maxval\_ypos **/** N**;**

data**.**xpos **=** **(double)** maxval\_xpos **/** N**;**

data**.**cpu\_time **=** **((double)**end**-**start**)/**CLOCKS\_PER\_SEC**;**

data**.**wall\_time **=** **(double)(**walltime\_e**.**tv\_sec **-** walltime\_s**.**tv\_sec **+** **(**walltime\_e**.**tv\_usec **-** walltime\_s**.**tv\_usec**)/1000000.0);**

*/\**

*// To print out the contour plot*

*if (N>=64) contour\_print(x,N);*

*\*/*

*//Free Memory*

free**(**x**);**free**(**y**);**free**(**w**);**free**(**S**);**

*#ifdef DEBUG*

printf**(**"DEBUG REPORT| RV = %d\n"**,** returnval**);**

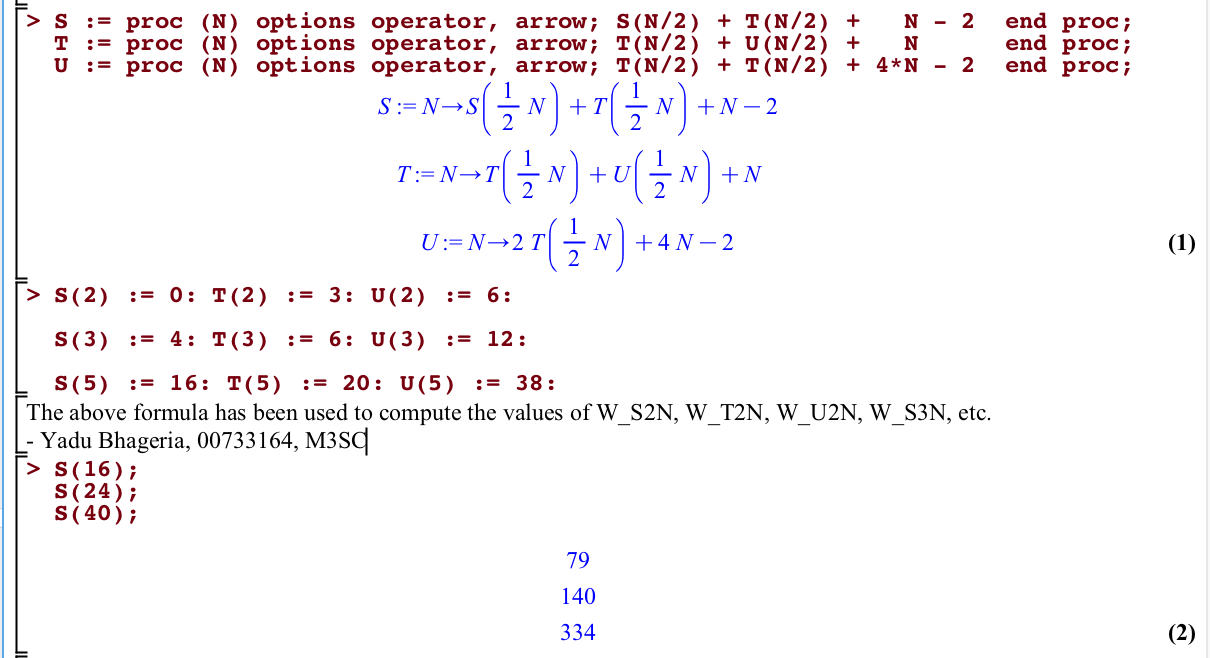
*#endif*

**return** data**;**

**}**

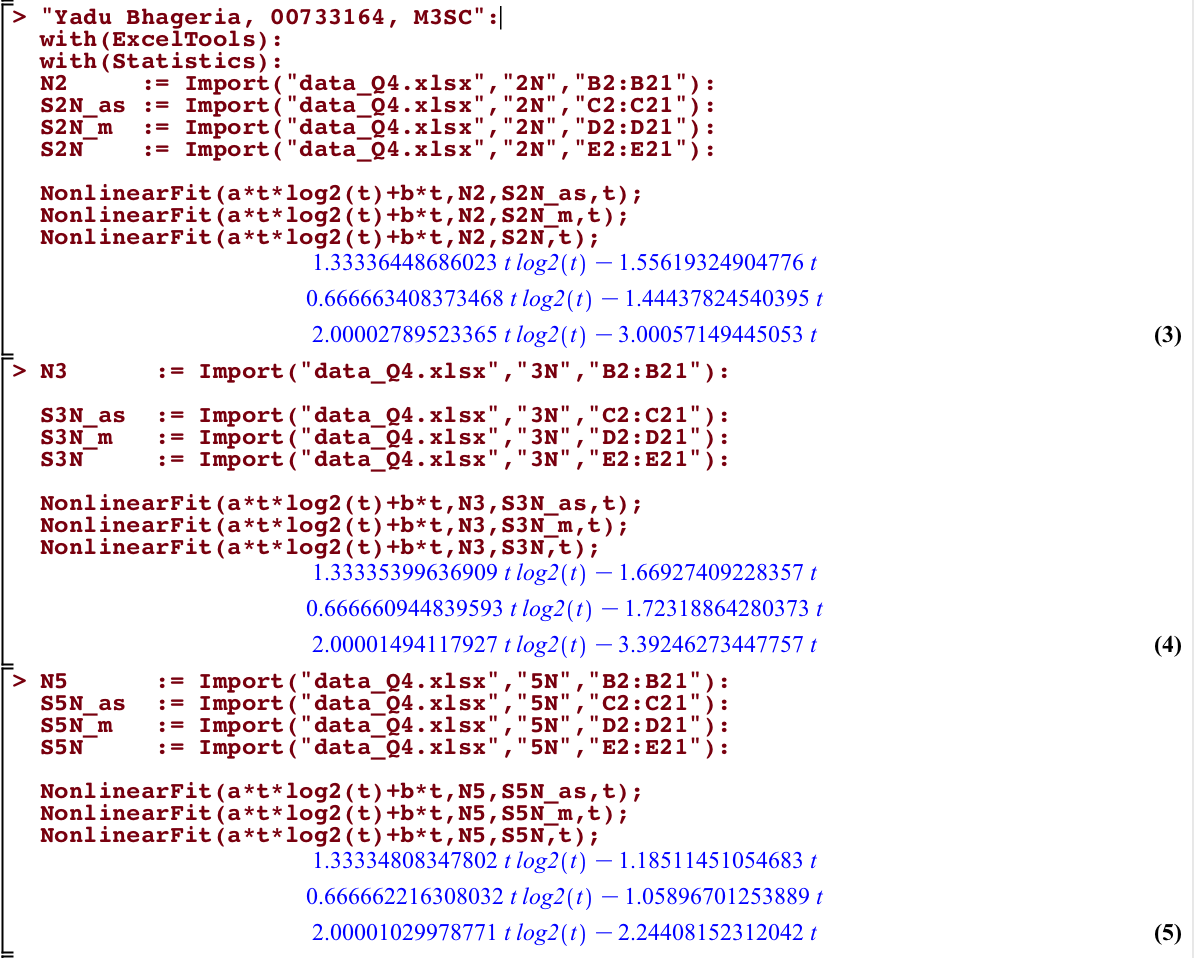
*/\* --------------------------------------------------------------------------- \*/*

## Maple Code for finding number of operations



## Maple Code for finding coefficients

Assumes there is an file called “data\_q4.xlsx” with the data in the appropriate sheets and cells.



##### MATLAB Code for finding coefficients

This gives the same result as the Maple code to 8 decimal places. Below is the code for the case. Different Data is read in for the other two cases.

%Yadu Bhageria, 00733164, M3SC

format long;

y = @(B,x) B(1).\*x.\*log2(x) + B(2).\*x; % B(1) = a, B(2) = b

%Read in Data about N

N = xlsread('data\_Q4.xlsx','2N','B2:B21');

%Read in Data about +/- operations

S2N\_as = xlsread('data\_Q4.xlsx','2N','C2:C21');

%Initial Guess

beta0 = [2,-1.8];

%Finding the coefficients

beta\_as = nlinfit(N,S2N\_as,y,beta0)

%Read in Data about \* operations

S2N\_m = xlsread('data\_Q4.xlsx','2N','D2:D21');

%Initial Guess

beta0 = [2,-1.8];

%Finding the coefficients

beta\_m = nlinfit(N,S2N\_m,y,beta0)

%Read in Data about total operations

S2N\_t = xlsread('data\_Q4.xlsx','2N','E2:E21');

%Initial Guess

beta0 = [2,-1.8];

%Finding the coefficients

beta\_t = nlinfit(N,S2N\_t,y,beta0)

1. Dr. Dan Moore, http://wwwf.imperial.ac.uk/~drmii/M3SC\_2016/index.htm [↑](#footnote-ref-1)
2. Yadu Bhageria, Exercise 2, Section: Poisson’s Equation in 1D, Table 1. I have changed powers of 2 to corresponding values of N for clarity. [↑](#footnote-ref-2)