Y3846520

Simulation of a Simple Dipole Problem and Attempted Parallelization of the Problem

Solving for the demagnetization factor of an arbitrary shape requires calculation of the interaction between all dipoles within the shape, a problem that can rapidly become very computationally demanding. Calculations were performed on systems ranging from one dipole up to 125000, for a serial program and parallel up to 8 threads. Parallelization was carried out with OpenMP across large nested do loops within which the bulk of calculations took place. This method revealed that, at least at this number of threads on a standard PC as opposed to a computer cluster, there was no speedup found from running in parallel, in fact it was found the program ran slower. However, there was noticeable speed up with an increasing number of threads used to solve the problem.

Introduction

The atoms in a simple cubic lattice can be considered as an array of static magnetic dipoles. This arises as a result of the individual spins and orbital angular momenta of the electrons which contribute to the total angular momentum. [1] A magnetic dipole (in the same way as a standard bar magnet) will of course have an associated magnetic field around it.

For an arbitrarily shaped magnetic object, in the specific case here an ellipsoid, there is a total magnetic field located within the object, resulting from the interactions of the magnetic dipoles within. The demagnetizing field, dependent upon the demagnetizing factor, is the H field generated within the object by the magnetization of it; it is so named due to its tendency to act on the magnetization and reduce the objects total magnetic moment. The demagnetization factor is itself dependent upon the shape of the object in question. [2]

For a large system of dipoles, the resulting magnetic field, and therefore the demagnetization factor, can clearly become very complex to calculate, requiring computational power. For example, for a relatively small system of 100 atoms there will be a requirement for on the order of 10^6 calculations to cover all interactions between all dipoles. Due to this, and the result of each interaction being independent of any other, the problem lends itself well to being parallelized. Therefore, the following report attempts to write a program to solve the demagnetization factor for an ellipsoid of magnetic dipoles, with the magnetic dipole moment in the x, y or z direction. Initially this is done in a serial manner, before implementing parallel architecture.

Theory

The field at a point around a dipole is given by:

$$\boldsymbol{B}_{dip}(\boldsymbol{r} > 0) = \frac{\mu\mu_0}{4\pi}\left[\frac{[3\hat{\boldsymbol{r}}(\hat{\boldsymbol{r}}\cdot\hat{\boldsymbol{m}})-\hat{\boldsymbol{m}}}{|\boldsymbol{r}|^3}\right] \quad (1)$$

In which the magnetic field at position $\boldsymbol{r}$ is $\boldsymbol{B}$, $\hat{\boldsymbol{r}}$ is the unit vector from the dipole to the point $\boldsymbol{r}$, $\hat{\boldsymbol{m}}$ is the unit vector of the magnetic moment of the dipole, μ_0 is the permeability of free space and μ is the magnetic dipole moment, here given by the Bohr magneton, μ_B:

$$\mu_B = \frac{e\hbar}{2m_e} \quad (2)$$

Where e is the charge of an electron and m_e electron rest mass. There is also an internal magnetic field for each dipole which is given by:

$$\boldsymbol{B}_{dip}(\boldsymbol{r} = 0) = \frac{\mu\mu_0}{4\pi l^3}\left[\frac{8\pi}{3}\hat{\boldsymbol{m}}\right] \quad (3)$$

In which l is the atomic spacing, here given as 3 Å. If all the dipoles are aligned along one direction (such that $\hat{\boldsymbol{m}}$ is equal for all dipoles) then the volumetric average magnetic field can be calculated by summing the interactions between all dipoles given by equation (1) and the self-interacting fields for each dipole given by equation (3) and divided by the total number of dipoles, this is done for the vectors of each dipole field, hence the average will also be a vector. This is then given by:

$$\bar{\boldsymbol{B}} = \mu_0 M(1 - \boldsymbol{D}) \quad (4)$$

Where M is the magnetisation given by the magnetic moment (Bohr magneton) over the atomic spacing cubed. $\boldsymbol{D}$ is the demagnetization factor, which specifically applies to the case of an ellipsoid as a relation between the magnetization and the demagnetizing field. Equation (4) can be rearranged to give the demagnetisation factor:

$$D_\alpha = 1 - \frac{B_\alpha}{\mu_0 M} \tag{5}$$

In which $\alpha = x, y, z$ is the cartesian components of the average magnetic and demagnetizing fields. Through this the demagnetising factor for an array of dipoles constrained by an ellipsoid (such that points outside the ellipsoid have a magnetic dipole moment of zero) can be computationally calculated, with the ellipsoid boundary defined as:

$$1 \geq \frac{(x-h)^2}{r_x^2} + \frac{(y-k)^2}{r_y^2} + \frac{(z-l)^2}{r_z^2} \tag{6}$$

Where x, y and z are the coordinates of the dipole, h, k and l are the origin coordinates of the ellipsoid and r_x, r_y and r_z are the ellipsoid radii along the corresponding axis.

Method

All results presented here were achieved using a PC with a quad-core, hyperthreaded Intel i7-7700HQ processor with clock speed of 2.80 MHz. For optimisation, the code was compiled using the flags -O3 and -march=native for optimum speed up and to automatically optimise the program to the native hardware. The -pg flag was also used for the profiling information, and -c for creation of module files.

To carry out the calculations, a 3D grid containing N^3 point dipoles was simulated by using a 1D array of possible coordinates, such that the array contained N values ranging from the lowest to the highest coordinate along a single axis. Hence for a system of $2 \times 2 \times 2$ dipoles the 1D array would contain 2 values at 3 Å and 6 Å. Then through a triple nested do-loop assigning the 1D arrays values to a pair of vectors corresponding to either the current dipole, or the external one with which the field is being calculated, the total magnetic field could be found by looping over all possible dipole locations. This was followed by adding the result from equation (3) multiplied by the total number of dipoles N^3.

Initially the code was written to loop over all N^3dipoles in the system before the ellipsoid was incorporated. This in itself lead to considerable speedup in execution, with a system of $N = 100$ reducing in run time from approximately seven hours to one. The ellipsoid was created with the x and y radii equal to 10 nm and the z radii equal to 20 nm.

The code was designed in such a way as to have the user decide important parameters upon starting a run, within the initialisation module (shown in appendix A). This section allows the user to unput the value for N, decide on if the atoms will have their dipole moments in the x, y or z directions and choose if the code runs serial or parallel. However, with small edits to the initialisation module and bash script (appendix E) the program can be automated to run multiple times, for instance for use on a computer cluster.

Due to the large number of loops involved in the program, it greatly leant itself to a parallelization strategy, this was done with OpenMP over the primary do loops. The main loops

were found by profiling the code during a serial run to find the most frequently called subroutines. This was done using gprof and is output to a text file for the user's convenience (appendix E).

Results

Consistently for both serial and parallel runs, the demagnetisation factor was calculated to be approximately $\frac{1}{3}$ in the direction of the magnetic dipole moment and 1.0 in other directions, such that:

$$D \approx \left(\frac{1}{3}, 1, 1\right) \quad \text{for} \quad \hat{\boldsymbol{m}} = (1, 0, 0)$$

$$D \approx \left(1, \frac{1}{3}, 1\right) \quad \text{for} \quad \hat{\boldsymbol{m}} = (0, 1, 0)$$

$$D \approx \left(1, 1, \frac{1}{3}\right) \quad \text{for} \quad \hat{\boldsymbol{m}} = (0, 0, 1)$$

Figure 1 shows the results of running the code at different values of N in series and in parallel with 8 threads. This shows that both series and parallel increasing in runtime with size of the system, but with the series calculations being consistently quicker than the parallel.

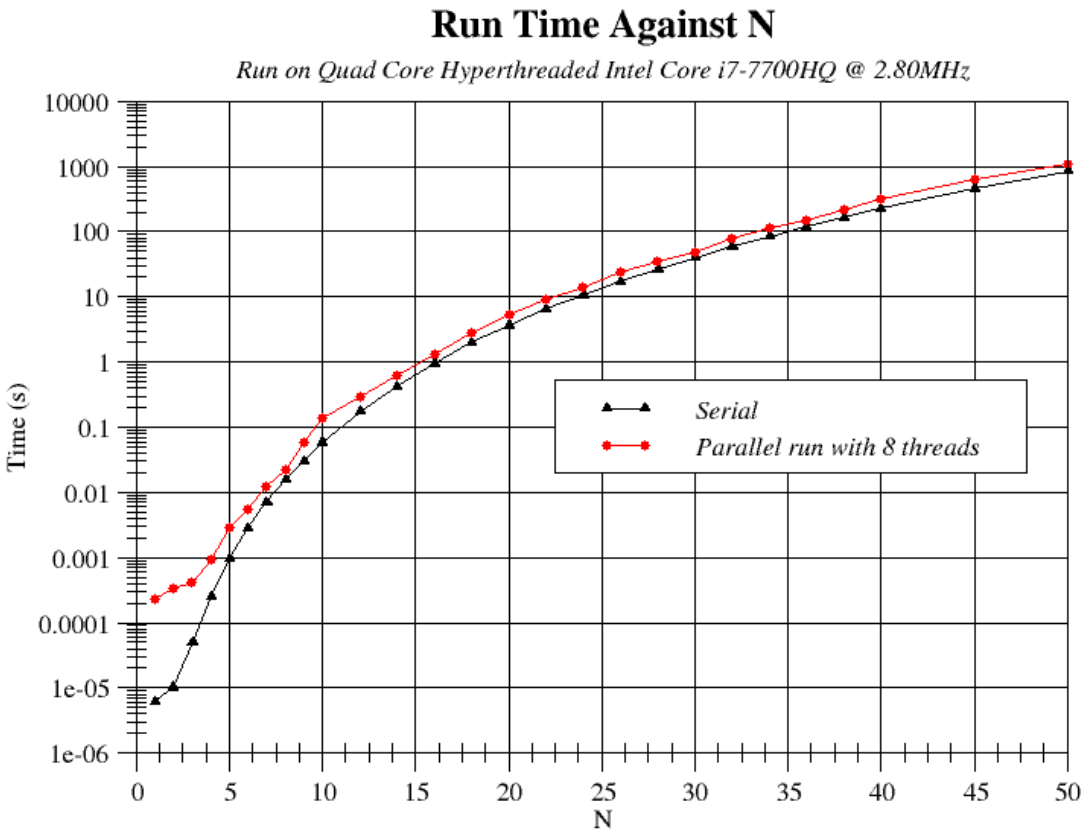

Figure 1: Plot of the run time for varying values of N, run in serial and in parallel with 8 threads. Calculations were run from N=1 to N=50, corresponding a total number of dipoles of 1 to 125000.

Figure 2 shows the results of running the code for different values of N and a varying number of threads available. This once again shows a clear increase in run time with N and also displays, once again, that the serial run was the fastest. Figure 2 also shows that there is a small, but noticeable, decrease in runtime with increasing number of threads from 2 upwards, with the percentage decrease in run time being consistent for different system sizes.

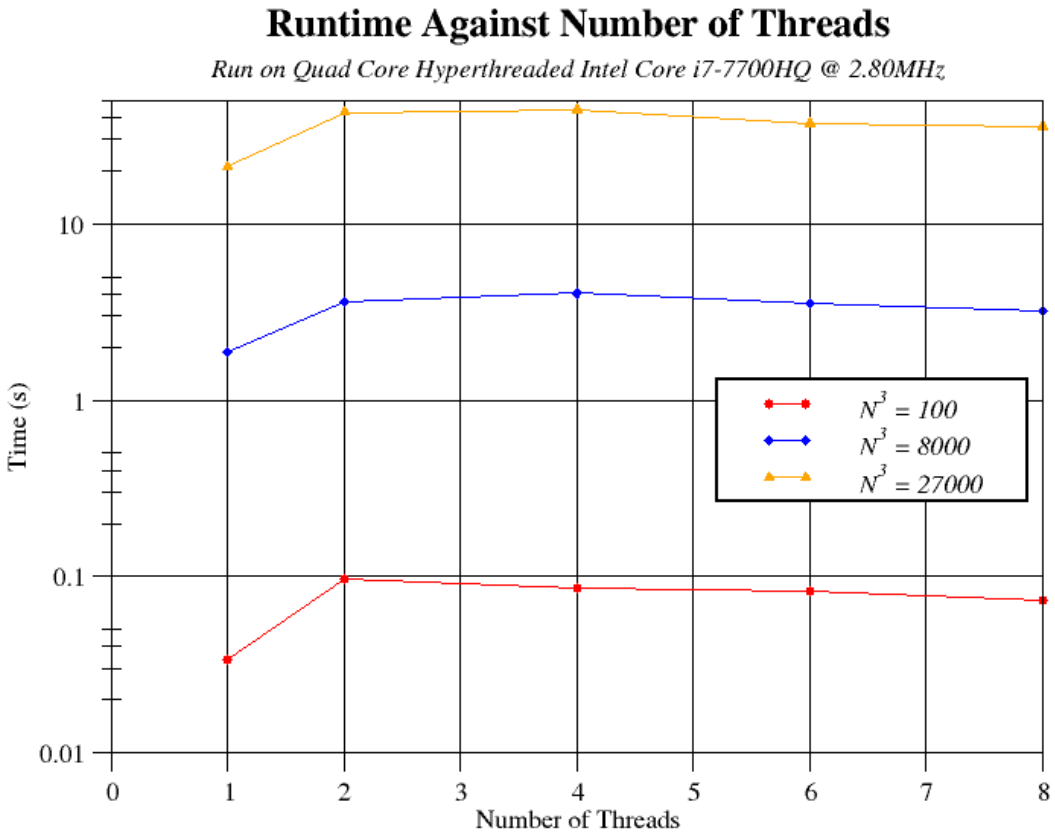

Figure 2: Plot of the run time against the number of threads, varying threads from 1 (serial) to 8. This was done for N = 10, 20, and 30 corresponding to system sizes of 100, 8000 and 27000.

Figure 3 shows a plot of the dipole positions within the ellipsoid for N = 50. It can be seen at the corners that the system of dipoles curve to follow the shape of the ellipsoid. These coordinates were written during the loop calculating the interactions for each dipole, such that if the condition which determines if the current dipole is within the ellipsoid is true the coordinates were written out to file. This allows confirmation that if a dipole is outside the ellipsoid, it is not being considered in calculations.

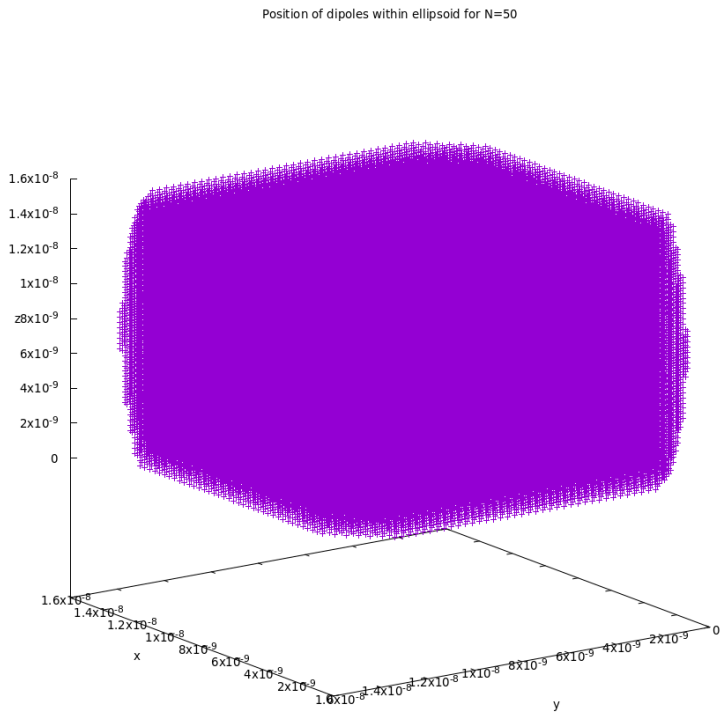

Figure 3: Plot of the dipole position for N=50, i.e., total number of dipoles 125000.

Discussion

When parallelizing, one of the main aims is of course improvement to the efficiency of a program such that it will run faster. The results displayed here suggest that the parallelization implemented in the code has not achieved that aim. However, the problem was only run up to 8 threads, if run to higher numbers of threads on a large cluster rather than home PC this may allow the parallelization to run faster than the serial, since the results in figure 2 show the speed of execution to reduce with higher threads towards the benchmark serial speed. A possible reason for the observed difference in speed may be the latency of each thread combining results at the end of the parallel region, which would also suggest the reduced difference in the speed in serial and parallel with a higher number of threads, as this latency becomes less impactful with the more rapid covering of the system via more threads.

Alternate methods, either in place of or alongside the use of OpenMP could be implemented into the code to further parallelize it, which would likely lead to more speed up and possibly cause it to run faster than when serial. The parallelization could be made hybrid along with MPI, or MPI implemented in place of OpenMP. Or the same could be said for implementing an accelerator, such as CUDA, alongside either one or both of these. But, of course, with this comes the challenge of effectively implementing each method alongside the others whilst considering what issues may arise, such as latency between CPU and GPU communication. Furthermore, there is the added difficulty in ensuring memory is allocated correctly, such that each process, thread, GPU etc can access what is needed when.

Conclusion

The results in figures 1 and 2 showed that, using OpenMP alone, there was no observable speed up in run time for the program. However, they also showed a noticeable increase in speed for higher numbers of threads, suggesting the possibility that at a higher thread number (such as on a cluster) there may be an improvement from the parallelization. The results shown in figure 3 show that the procedure of the code correctly distinguished dipoles that were within and outside the ellipsoid.

References

[1] – Tilley R.J.D. (2004), *Understanding Solids.* John Wiley and Sons.

[2] – Pugh B.K. *et al.* (2011), Demagnetizing Factors for Various Geometries Precisely Determined Using 3-D Electromagnetic Field Simulation, *IEEE Transactions on Magnetics.* doi: 10.1109/TMAG.2011.2157994

Appendices

Appendix A – Initialisation Module

```
module initialise
!------------------------------------------------------------------------------!
!Module to initialise problem by allowing user to select desired N, direction of--!
!------magnetic moments and whether to run in series or parallel.----------------!
!------------------------------------------------------------------------------!
  use physical_constants

  implicit none

  contains

  !initialisation subroutine
  recursive subroutine init(mag_vector, coords, N, s0Rp)

    real(kind=dp), dimension(3), intent(out) :: mag_vector

    real(kind=dp), dimension(:), allocatable, intent(out) :: coords

    integer, intent(out) :: N

    character :: mag_direction

    character, intent(out) :: s0Rp

    !If this read statement is uncommented, will be used to read in values for N,
    !mag_direction and s0Rp from bash script rather than them being entered at run
time.
    !For this must also comment out all other read statements in this module
    !read (*,*) N, mag_direction, s0Rp

    !0pen a file which can be 3D plotted (e.g. with gnuplot) to
    !display the ellipsoid of points (or cube should the dimensions be smaller than
the
    !ellipsoid radii.
    !open(unit=unit1, file="ellipsoid.txt", iostat=istat)
    !if(istat/=0) stop "Error opening ellipsoid.txt"

    !Section allows user to choose value for N
    print *, "------------------------------------------------------------------"

    print *, "Enter a value for N. The total number of dipoles will be", &
             " N cubed, such that they form a cube of side N dipoles."

    read(*,*) N

    !check to make sure N is > 0
    if(N<=0)then

      print *, "ERR0R: N may not be less than or equal to zero."

      print *, coords

      call init(mag_vector, coords, N, s0Rp)

    else

      print *, "N is set to:", N

    endif

    !Section allows user to choose direction of magnetic dipole moment
    print *, "------------------------------------------------------------------"

    print *, "Please choose the direction of the magnetic moment of each", &
             " dipole"

    print *, "Type 'x', 'y' or 'z' to choose the direction. 0r type 'r' to", &
             " restart."
```

```
read(*,*) mag_direction

if(mag_direction == "x")then

  mag_vector = [1.0_dp, 0.0_dp, 0.0_dp]

elseif(mag_direction == "y")then

  mag_vector = [0.0_dp, 1.0_dp, 0.0_dp]

elseif(mag_direction == "z")then

  mag_vector = [0.0_dp, 0.0_dp, 1.0_dp]

elseif(mag_direction == "r")then

  call init(mag_vector, coords, N, s0rp)

else

  print *, mag_direction, " is not a valid input."

  call init(mag_vector, coords, N, s0Rp)

endif

print *, "The unit vector for the magnetic moment of each dipole is"

print *, mag_vector

!check to see if coords has already been allocated
!This happens if N was given a not allowed value (<=0)
if(allocated(coords))then

  deallocate(coords)

endif

!allocate size of coordinate arrays
allocate(coords(1:N))

!set possible x, y and z coordinates
do i = 1, N

  coords(i) = real(i,dp) * l

enddo

!initialise ellipsoid
call ellipsoid(N, l, ellipCent)

print *, "-----------------------------------------------------------------"

print *, "Ellipsoid is centred on the coordinates x = y = z with x,y,z ="

print *, ellipCent

print *, "-----------------------------------------------------------------"

print *, "Select whether to calculate in serial('s') or parallel('p')., &
         " 0r type 'r' to restart."

read(*,*) s0Rp

if(s0Rp == "s")then

  print *, "Running in serial."

elseif(s0Rp == "p")then

  print *, "Running in parallel."

elseif(s0Rp == "r")then
```

```
      call init(mag_vector, coords, N, s0Rp)

    else

      print *, s0Rp, "is not a valid input."

      call init(mag_vector, coords, N, s0Rp)

    endif

  endsubroutine init

  !Subroutine to create the ellipsoid such that it is centered on the centre of the
cube
  !of dipoles produced.
  subroutine ellipsoid(N, l, center)

    real(kind=dp), intent(in) :: l

    real(kind=dp) :: halfway

    integer, intent(in) :: N

    !centre point of ellipsoid, centred on the centre of the cube of dipoles
    real(kind=dp), intent(out) :: center

    !calculate halfway point between lowest and highest coordinate points
    halfway = (N*l) - l

    !set central point of the ellipsoid
    center = (halfway / 2.0_dp) + l

  endsubroutine ellipsoid

endmodule initialise
```

Appendix B – Main Program

```
program serial

  use mag_field_calculations
  use vectors
  use initialise
  use physical_constants
  use omp_lib
  use omp_lib_kinds

  implicit none

  call init(mag_vector, coords, N, s0Rp)

  call main(coords, mag_vector)

  call finish(coords)

  contains

  !Main program
  subroutine main(coords, mag_vector)

    real(kind=dp), dimension(:), intent(inout) :: coords

    real(kind=dp), dimension(3), intent(inout) :: mag_vector

    !sum of magnetic field from all dipoles for calculating average
    real(kind=dp), dimension(3) :: magnet_total

    !average magnetic field
    real(kind=dp), dimension(3) :: magnet_ave

    !Demagnetisation factor
    real(kind=dp), dimension(3) :: D
```

```
  !timing
  real(kind=dp) :: start_time, end_time

  !Time at start of run
  start_time = omp_get_wtime()

  print *, "------------------------------------------------------------------"

  print *, "Demagnetisation calculations running"

  !initialise magnet_sum, magnet average and demagnetisation factor
  magnet_total = [0.0_dp, 0.0_dp, 0.0_dp]
  magnet_ave = [0.0_dp, 0.0_dp, 0.0_dp]

  !Calculate in serial
  if(s0Rp == "s")then

    magnet_total = serial_magnet_sum(coords, mag_vector) &
                 + (self_field(mag_vector) * real(N**3, dp))

  !Calculate in parallel
  elseif(s0Rp == "p")then

    magnet_total = parallel_magnet_sum(coords, mag_vector) &
                 + (self_field(mag_vector) * real(N**3, dp))

  endif

  !calculate the average magnetic field over all dipoles
  magnet_ave = magnet_total / N**3

  print *, "Average magnetic field is:", magnet_ave

  D = demag(magnet_ave)

  print *, "------------------------------------------------------------------"

  print *, "Demagnetisation factor is:", D

  end_time = omp_get_wtime()

  print *, "Time taken:", end_time - start_time, "s"

endsubroutine main

subroutine finish(coords)

  real(kind=dp), dimension(:), allocatable, intent(inout) :: coords

  close(unit=unit1, iostat=istat)
  if(istat/=0) stop "Error closing ellipsoid.txt"

  deallocate(coords)

endsubroutine finish

function serial_magnet_sum(coords, mag_vector)

  real(kind=dp), dimension(:) :: coords

  real(kind=dp), dimension(3) :: mag_vector

  real(kind=dp), dimension(3) :: current_dipole

  real(kind=dp), dimension(3) :: serial_magnet_sum

  real(kind=dp), dimension(3) :: current_ext_field

  real(kind=dp) :: z_component_sum

  do i = 1, N

    current_dipole(1) = coords(i)
```

```
      do j = 1, N

        current_dipole(2) = coords(j)

        do k = 1, N

          current_dipole(3) = coords(k)

          !check to see if the current dipole is within the ellipsoid,
          !otherwise the current dipole is ignored, i.e. m = 0.
          if(ellipsoidcheck(current_dipole, rx, ry, rz, ellipCent) &
                                <= 1.0_dp)then

            current_ext_field = magnetic_field(current_dipole, mag_vector)

            serial_magnet_sum = serial_magnet_sum + current_ext_field

          endif

        enddo

      enddo

    enddo

  endfunction serial_magnet_sum

  function parallel_magnet_sum(coords, mag_vector)

    real(kind=dp), dimension(:) :: coords

    real(kind=dp), dimension(3) :: mag_vector

    real(kind=dp), dimension(3) :: current_dipole

    real(kind=dp), dimension(3) :: parallel_magnet_sum

    integer :: nthreads, threadNum

    print *, "Check number of threads:"
    !Check number of threads
    !$omp parallel default(private)

      nthreads = omp_get_num_threads()

      threadNum = omp_get_thread_num()

      print *, "Thread", threadNum+1, "of", nthreads

    !$omp end parallel

    !$omp  parallel  do  shared(coords,mag_vector)  private(i,j,k,current_dipole)
reduction(+:parallel_magnet_sum)
    do i = 1, N

      current_dipole(1) = coords(i)

      do j = 1, N

        current_dipole(2) = coords(j)

        do k = 1, N

          current_dipole(3) = coords(k)

          !check to see if the current dipole is within the ellipsoid,
          !otherwise the current dipole is ignored, i.e. m = 0.
          if(ellipsoidcheck(current_dipole, rx, ry, rz, ellipCent) &
                                <= 1.0_dp)then
            parallel_magnet_sum = parallel_magnet_sum + &
                                  magnetic_field_parallel&
                                  (current_dipole,mag_vector)

          endif
```

```
        enddo

      enddo

    enddo
    !$omp end parallel do
  endfunction parallel_magnet_sum

endprogram serial
```

Appendix C – Module of Calculations for Magnetic Field

```
!-------------------------------------------------------------------------------!
!Module contains functions for calculating magnetic fields and for calculating-!
!the demagnetisation factor.-----------------------------------------------------!
!-------------------------------------------------------------------------------!
module mag_field_calculations

  use physical_constants

  use vectors

  use omp_lib

  use omp_lib_kinds

  implicit none

  contains

  !function to calculate the demagnetisation factor (in 3D) from a read in magnetic
field
  !vector B. This is the average magnetic field vector.
  function demag(B)

    real(kind=dp), dimension(3) :: demag

    real(kind=dp), dimension(3) :: B

    real(kind=dp) :: magnetisation

    !calculate magnetisation
    magnetisation = BohrMag / 1**3

    demag = 1.0_dp - (B / (PFS * magnetisation))

  endfunction demag

  !function to calculate the magnetic field at a dipole as a result of the same
  !dipole
  function self_field(mag_vector)

    real(kind=dp), dimension(3) :: self_field, mag_vector

    real(kind=dp), dimension(3) :: bracket

    real(kind=dp) :: coef

    coef = (PFS * BohrMag) / (4.0_dp * pi * 1**3)

    bracket = ((8.0_dp * pi) / 3.0_dp) * mag_vector

    self_field = coef * bracket

  endfunction self_field

  !function to calculate the magnetic field inbetween two dipoles, current
  !dipole and one that is at position r from current dipole
  function magnetic_field(pos, mag_vector)

    real(kind=dp), dimension(3) :: magnetic_field
```

```
  !pos is the coordinates of the current dipole, mag_vector is the
  !magnetic moment unit vector
  real(kind=dp), dimension(3) :: pos, mag_vector

  !vector to update for each other dipole
  real(kind=dp), dimension(3) :: dipoleExt

  !vector r, vector from current dipole to all others
  real(kind=dp), dimension(3) :: r

  !unit vector of r
  real(kind=dp), dimension(3) :: unit_r

  !magnitude of vector r
  real(kind=dp) :: r_mag

  real(kind=dp) :: coef

  real(kind=dp), dimension(3) :: numerator

  integer :: o, p, q

  !initialise magnetic field
  magnetic_field = 0.0_dp

  !Loop over all external dipoles.
  do o = 1, N

    dipoleExt(1) = coords(o)

    do p = 1, N

      dipoleExt(2) = coords(p)

      do q = 1, N

        dipoleExt(3) = coords(q)

        !check to see if current external dipole is within the ellipsoid
        if(ellipsoidcheck(dipoleExt, rx, ry, rz, ellipCent)<=1.0_dp)then

          !calculate vector r
          r = dipoleExt - pos

          !calculate magnitude of vector r
          r_mag = magnitude(r)

          !if magnitude of r is 0, i.e. r=[0,0,0], don't do calculations as
          !maths errors will occur
          if(r_mag /= 0.0_dp)then

            !calculate r unit vector
            unit_r = r / r_mag

            !calculate magnetic field between two dipoles
            coef = (PFS * BohrMag) / (4.0_dp * pi)

            numerator = (3.0_dp * dot(unit_r, mag_vector) * unit_r) &
                        - mag_vector

            magnetic_field = coef * (numerator / r_mag**3)

          endif

        endif

      enddo

    enddo

  enddo

endfunction magnetic_field
```

```
  !Function to calculate the magnetic field between 2 dipoles, at position pos and
dipoleExt
  !in parallel.
  function magnetic_field_parallel(pos, mag_vector)

  real(kind=dp), dimension(3) :: magnetic_field_parallel

  !pos is coordinates of the current dipole, mag_vector is the magnetic moment
  !of the unit vector
  real(kind=dp), dimension(3) :: pos, mag_vector

  !vector to update for each other dipole
  real(kind=dp), dimension(3) :: dipoleExt

  !vector, r, from current dipole to all others
  real(kind=dp), dimension(3) :: r

  !unit vector of r
  real(kind=dp), dimension(3) :: unit_r

  !magnitude of vector r
  real(kind=dp) :: r_mag

  real(kind=dp) :: coef

  real(kind=dp), dimension(3) :: numerator

  integer :: o, p, q

  !initialise magnetic field
  magnetic_field_parallel = 0.0_dp

  !$omp parallel do
  do o = 1, N

    dipoleExt(1) = real(o, dp) * l

    !$omp parallel do
    do p = 1, N

      dipoleExt(2) = real(p, dp) * l

      !$omp parallel do
      do q = 1, N

        dipoleExt(3) = real(q, dp) * l

        !check to see if current external dipole is within the ellipsoid
        if(ellipsoidcheck(dipoleExt, rx, ry, rz, ellipCent)<=1.0_dp)then

          !calculate vector r
          r = dipoleExt - pos

          !calculate magnitude of r
          r_mag = magnitude(r)

          !if magnitude r = 0 skip doing calculations
          if(r_mag /= 0.0_dp)then

            !calculate r unit vector
            unit_r = r / r_mag

            !calculate the magnetic field
            coef = (PFS * BohrMag) / (4.0_dp * pi)

            numerator = (3.0_dp * dot(unit_r, mag_vector) * unit_r) &
                        - mag_vector

            magnetic_field_parallel = coef * (numerator / r_mag**3)

          endif

        endif
```

```
      enddo
      !$omp end parallel do

    enddo
    !$omp end parallel do

  enddo
  !$omp end parallel do

  endfunction magnetic_field_parallel

  !Function checks if the input position vector lies within the boundary of the
ellipsoid.
  function ellipsoidcheck(pos, rx, ry, rz, origin)

    real(kind=dp), dimension(3) :: pos

    real(kind=dp) :: rx, ry, rz

    real(kind=dp) :: origin

    real(kind=dp) :: ellipsoidcheck

    ellipsoidcheck = ((pos(1) - origin)**2 / rx**2) + &
    ((pos(2) - origin)**2 / ry**2) + ((pos(3) - origin)**2 / rz**2)

  endfunction ellipsoidcheck

endmodule mag_field_calculations
```

Appendix D – Module of Vector Calculations

```
!-------------------------------------------------------------------------------!
!------Module contains functions for calculating the dot product---------------!
!------of two vectors, and the magnitude of a vector.--------------------------!
!-------------------------------------------------------------------------------!
module vectors

  use physical_constants

  implicit none

  contains

  !Function to calculate the dot product between two vectors, A and B.
  function dot(A,B)

    real(kind=dp) :: dot

    real(kind=dp), dimension(3) :: A, B

    dot = A(1)*B(1) + A(2)*B(2) + A(3)*B(3)

  endfunction dot

  !Function to calculate the magnitude of a vector A.
  function magnitude(A)

    real(kind=dp), dimension(3) :: A

    real(kind=dp) :: magnitude

    magnitude = sqrt(A(1)**2 + A(2)**2 + A(3)**2)

  endfunction magnitude

endmodule vectors
```

Appendix E – Bash Script to Compile and Run Code and Modules

```
#!/bin/bash
#shell script to compile and run program
```

```
#clear terminal screen
clear

#remove files from constants module
rm physical_constants.o physical_constants.mod

#remove files from main program
rm serial.o serial.exe

#remove files from vectors module
rm vectors.o vectors.mod

#remove files from magnetic field module
rm mag_field_calculations.o mag_field_calculations.mod

#remove files from initialisation module
rm initialise.o initialise.mod

#choose to remove any text files from previous runs, such as profile, or any text
files
#the user wishes to reproduce.
rm -i *.txt

#number of threads if running in parallel, can be changed by the user as desired.
export OMP_NUM_THREADS=8

#compile constants module
gfortran -pg -O3 -march=native -c physical_constants.f90
echo "Constants module compiled"

#compile initialise module
gfortran -pg -O3 -march=native -c initialise.f90
echo "Initialise module compiled"

#compile vectors module
gfortran -pg -O3 -march=native -c vectors.f90
echo "Vectors module compliled"

#compile magnetic field module
gfortran -pg -fopenmp -O3 -march=native -c mag_field_calculations.f90
echo "Magnetic field module compiled"

#compile main program
gfortran -pg -O3 -march=native -c -fopenmp mainprog.f90
echo "Main program compiled"

#Link main program with modules
gfortran -o mainprog.exe -pg -fopenmp -O3 -march=native physical_constants.o \
initialise.o vectors.o mag_field_calculations.o mainprog.o
echo "Main program and modules linked"

./mainprog.exe
#print profiling to file
gprof mainprog.exe > profile.txt

#From here down can be used to automate multiple runs of the program with different
values
#of N, mag (equaivalent to mag_direction in code) and s0Rp.
#All lines to be uncommented and can be edited by user as desired.
#Currently set to run the problem 6 times with different values of N, with the
#magnetic moment direction in the x-direction and all runs in serial.
#declare -i N
#mag=x
#s0Rp=s
#for N in 1 2 5 10 20 30
#do
#  echo "${N} ${mag} ${s0Rp}" | ./mainprog.exe >> out.txt
#done
```