MATH96012 M3C Final Project

Oscar Pring 01198242

December 13, 2019

Question 1.1

So the goal here is to compute the matrix multiplication $Ay = z$ and $A^T y = z$, where $A, A^T \in R^{m \times m}$, and both $y, z \in R^m$, $m = (n+2)^2$.
First we consider the subroutine mvec. We know that the kth element of y corresponds to $w_{i,j}$, so we construct the kth row of A by considering the discretised governing equation or the boundary conditions at (r_i, θ_j).

We begin by looking at the rows of A that are determined by the governing equation. Of these, take the kth row $k \in \{(n+2)+1, ..., (n+1)*(n+2)\}$. Here, this row has 5 non-zero elements which I describe using Python index notation:

$$A[k, k-(n+2)] \in f_m \tag{1}$$

$$A[k, k-1] \in f_2 \tag{2}$$

$$A[k, k] = -1 \tag{3}$$

$$A[k, k+1] \in f_2 \tag{4}$$

$$A[k, k+(n+2)] \in f_p \tag{5}$$

,
where f_m, f_2, f_p are some of the input vectors of coefficients into *mvec*.

We undertake a similar approach for considering the rows of A that are determined by the boundary conditions, although here the relevant rows only contain two non-zero elements. My initial approach was to directly build the matrix A, and then use Fortran's *matmul* function to compute the matrix vector product. However, as A is mostly zeros, this would require a lot of redundant, expensive computation. Instead, I decided to directly sum only the non-zero elements that make up each entry of the matrix-vector product, which is a much more efficient way to do the computation.

For the subroutine, *mtvec*, I took a different approach. Like with *mvec*, my first idea was to build the matrix A directly, transpose it, and then use *matmul* to compute the matrix-vector product. Again this would have been a very computationally expensive way to do it. So, my approach was as follows.

First, we observe that the columns of A^T are the rows of A. We then compute the product using the fact that

$$z = A^T y = \sum_{i=1}^{m} y_i [A^T]_i \tag{6}$$

where $[A^T]_i$ is the ith column of A^T, which is also the ith row of A. This insight allows for a much easier computation of the matrix vector product, as we can build the rows of A easily from considering the governing equation and the boundary conditions and then take cumulative sums, iterating through the rows of A.

Question 1.4

We first verify visually that the Fortran and Python routines using the jacobi method, as well as *sgisolve*, produce the same outputs when provided with the same input. This is indeed the case, shown in figures 1 through 7.

In figure 8, we establish that the time taken for the subroutine *mvec* to run against n increases slightly as n increases. The subroutine is very efficient and fast. In figure 9 we compare the runtimes of *mvec* and *mtvec* against n. Whilst *mtvec* is still fast, it is exponentially slower than *mvec*. This is to be as expected as my algorithm for *mtvec* involved summing large 1-dimensional arrays containing mostly zeros, a fairly computationally expensive exercise.

In figure 10, we compare the runtimes of Fortran jacobi, Python jacobi and *sgisolve* with both 1 and 2 threads. First of all, as expected, *sgisolve* with 2 threads is slightly quicker than the function with only 1 thread, but not by much as only a small part of *sgisolve* could be parallelised. For $5 \leq n \leq 24$,

all of the Fortran routines are slightly faster than the Python jacobi implementation, which can be attributed to compiler optimisations. But for $n \geq 24$, as the problem size and memory requirements grow larger, the Fortran compiler struggles to find effective optimisations for *sgisolve*, so the vectorised Python jacobi implementation becomes increasingly faster.

However, both *sgisolve* routines are exponentially slower than the two jacobi routines for $n \geq 24$, which I believe comes down to the repeated matrix-vector multiplication using *mtvec* in the main iterative loop of *sgisolve*.

In figure 11, we make the same comparisons as in figure 9, but with a larger range of n values. The purpose of this was to demonstrate fully that the *sgisolve* routines are exponentially slower than the jacobi routines, as can be clearly seen.

Figures

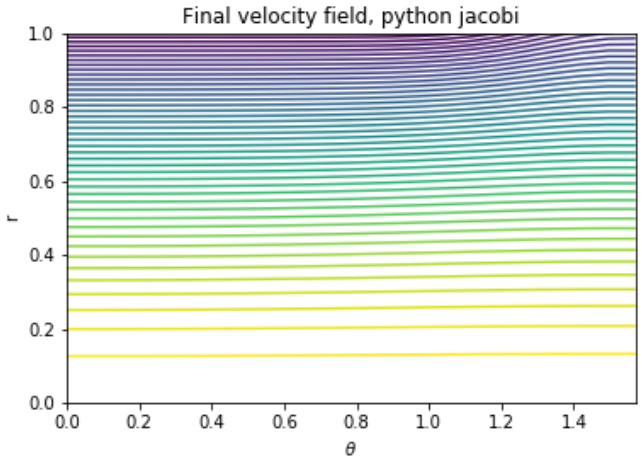

Figure 1:

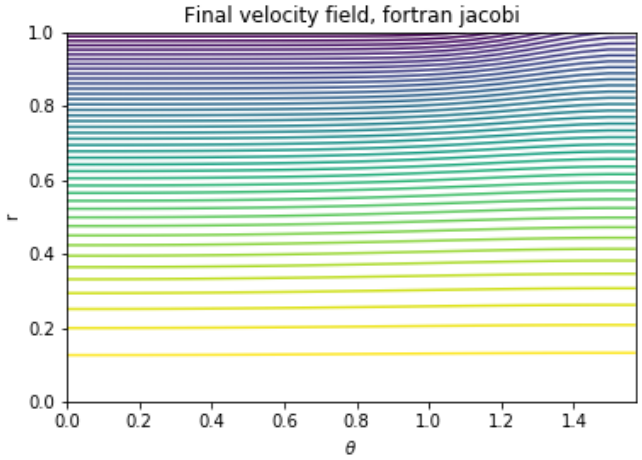

Figure 2:

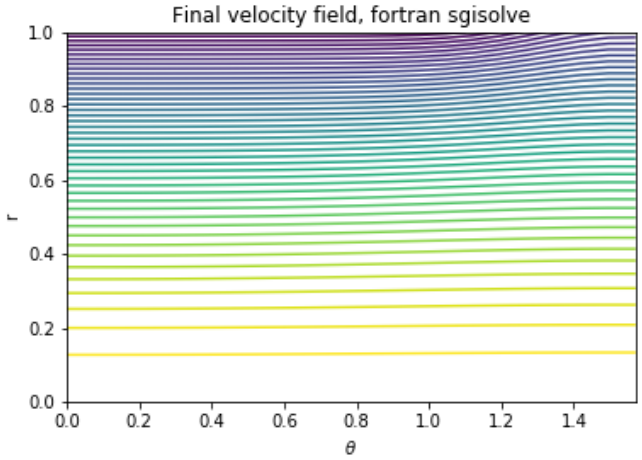

Figure 3:

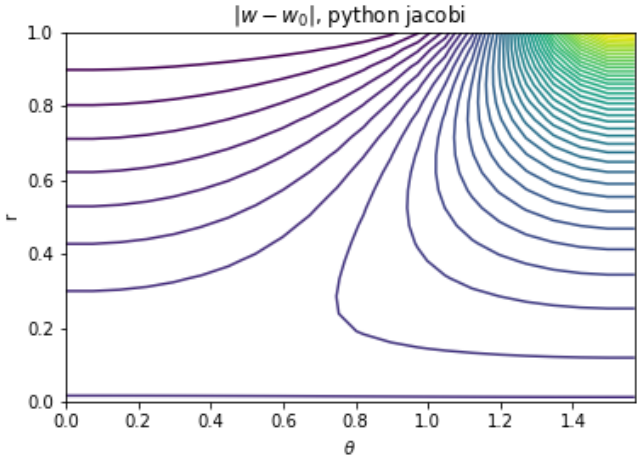

Figure 4:

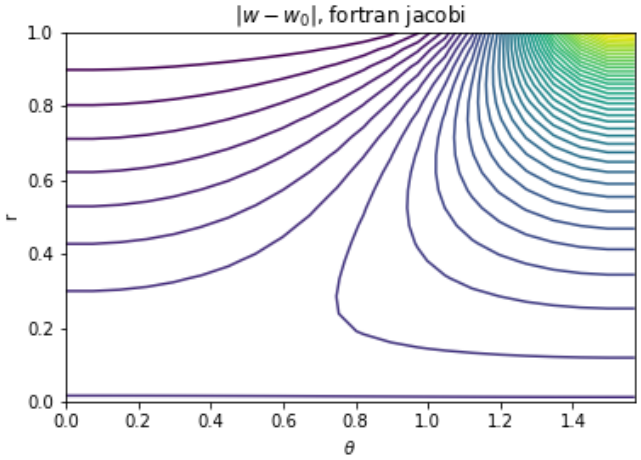

Figure 5: Figure for question 1

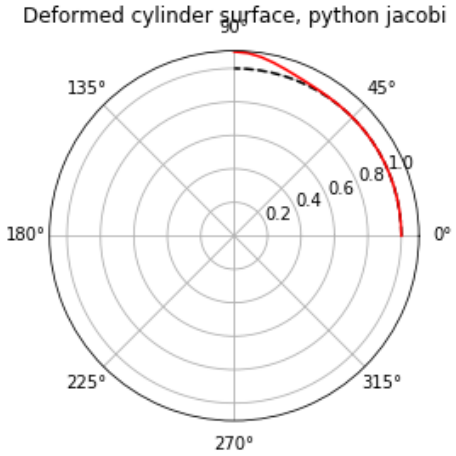

Figure 6:

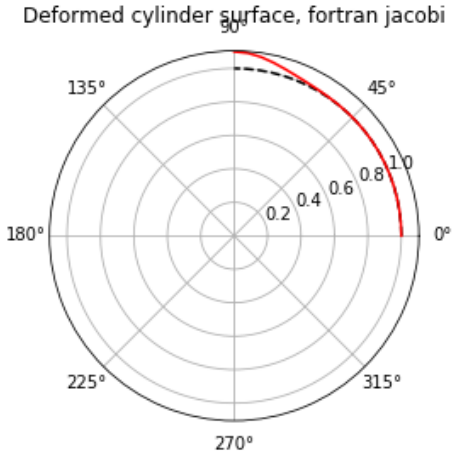

Figure 7:

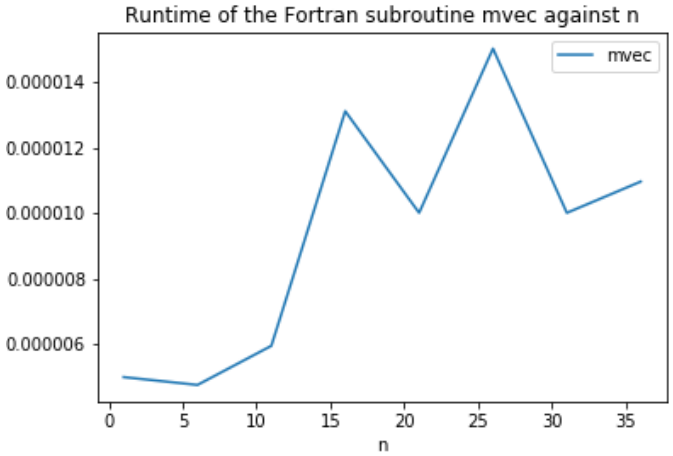

Figure 8:

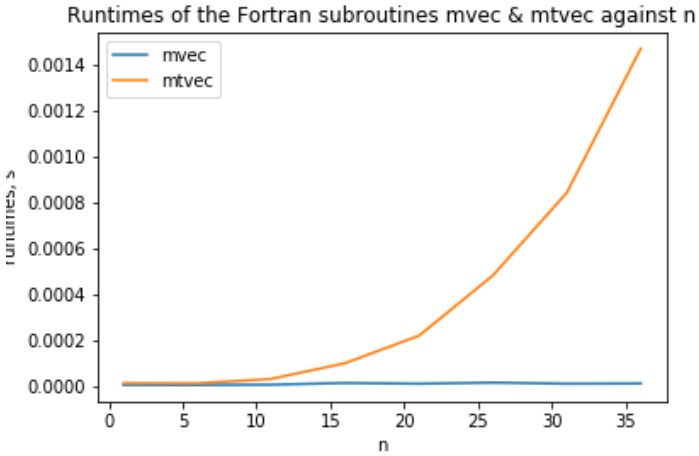

Figure 9:

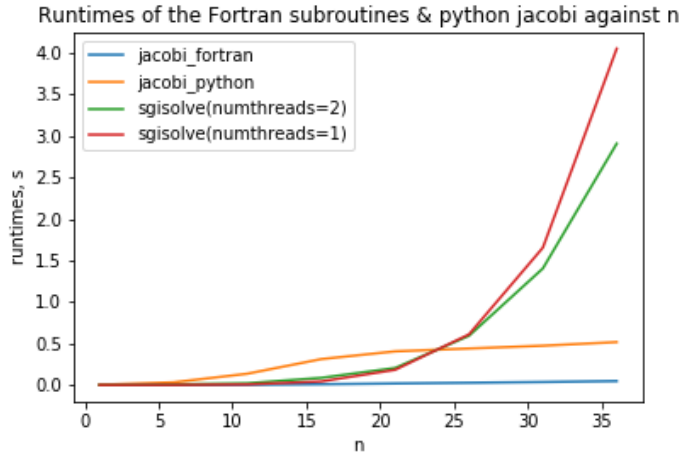

Figure 10:

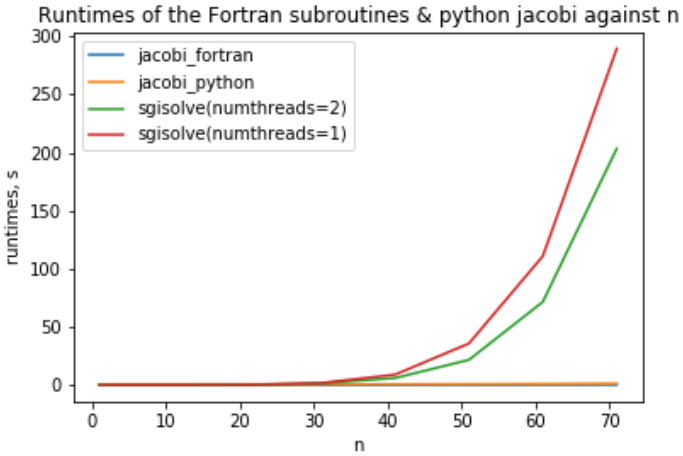

Figure 11:

Question 1.5

So we need to distribute m elements in the vector x to numprocs processes. This is accomplished using the subroutine MPE_DECOMP1D. "Boundary" ele-

ments, near the process partitions, will need to recieve neighbouring information to compute mvec and mtvec. Here, the data exchanged at each boundary is at most the n+2 elements adjacient to the process partition flowing in both directions. This data is sent and recieved using MPI_ISEND and MPI_IRECV, followed MPI_BARRIER to act as a safety feature. This data would then need to be gathered onto the root process before the main iterative loop can begin, using MPI_GATHERV.
The main iterative loop however, of *sgisolve*, would not be possible to do in parallel, as each iteration is dependent on the result of the previous iteration.

The main advantage to implementing *sgisolve* using MPI is that it would greatly speed up the matrix-vector product computations, *mvec* and *mtvec* at the start of *sgisolve*, particularly *mtvec* as it exponentially slower than mvec. However, the main disadvantages of using MPI is that it is really difficult to implement correctly and to see worthwhile results m would have to be very large.

Question 2.3

Here I will breakdown my approach to the problem.

First we generate the decomposition using MPE_DECOMP1D. We then allocate the subdomain variables, where each process gets nlocal oscillators. Each process may need to recieve at most $2 \times max(ai)$ phases, i.e $1 \times max(ai)$ from each side.

I then set up the send/recieve protocol to do this, pairing corresponding sending and recieving processes, and then sending the relevant data using MPI_ISEND, MPI_IRECV and then MPI_BARRIER to act as a safety feature.

Once this is done, the main iterative loop begins. The subroutine RHS takes in an $Nlocal + 2 \times max(ai)$ length array, computes the calculation as described in the project description, and returns the result, which is of length $Nlocal$. ylocal is then updated.

We then iterate through the main loop, calculating the relevant quantities on each process, sending and recieving as necessary, keep track of the alignment parameter R(t). Once the iterative loop is complete, we then use MPI_GATHERV to collect the ylocals into the variable y on the process with $myid = 0$.

I have been unable to get my attempt at implementing this task to work. I have submitted my attempt, in the file p42.f90, which compiles but does not execute. I have been unable to identify what the errors are and how to resolve them.

Question 2.4

There are various changes that would need to be made to adapt the model to work on a square grid.

First, instead of using MPE_DECOMP1D to generate the decomposition, I would use one of MPIs tools for creating and managing complex topologies. On a square grid we would be dealing with 2 dimensions, so $ndims = 2$. Using MPI_DIMS_CREATE(numprocs,ndims) to get the dimensions of the decompositions = $dims$. Then use MPI_CART_CREATE to create the optimal grid, and the use MPI_CART_SHIFT to provide the id of neighbouring processes and to set up the send/recv sequences for exchanging boundary data.
