Coursework cover sheet - MA40177 Coursework 2

General instructions

Set Tue, 18 Apr 2023

Due Thu, 4 May 2023, 12noon

Estimated time required This assignment should take an average student about 25 hours to complete (provided they have done all the problem sheet questions and tutorial exercises)

Submission Submit you solution in the marked box (not the pigeonhole) on 4West Level 1 and on `Moodle`. See instructions below for further details.

Value This assignment is worth 50% of the total assessment for MA40177

Support and advice You can ask questions on the coursework on the online Padlet board and in the lectures, where they will be answered when all students are present. In the interest of fairness, it will not be possible to answer individual question from students, for example by email.

Feedback You will receive feedback within a maximum of three semester weeks following the submission deadline. The feedback will consist of your marked work and an overall feedback document commenting on the assessment across the cohort.

Late submission of coursework If there are valid circumstances preventing you from meeting the deadline, your Director of Studies may grant you an extension to the specified submission date, if it is requested before the deadline. Forms to request an extension are available on SAMIS.

- If you submit a piece of work after the submission date, and no extension has been granted, the maximum mark possible will be the pass mark.
- If you submit work more than five working days after the submission date, you will normally receive a mark of 0 (zero), unless you have been granted an extension.

Academic integrity statement Academic misconduct is defined by the University as "the use of unfair means in any examination or assessment procedure". This includes (but is not limited to) cheating, collusion, plagiarism, fabrication, or falsification. The University's Quality Assurance Code of Practice, QA53 Examination and Assessment Offences, sets out the consequences of committing an offence and the penalties that might be applied.

MA40177: Scientific Computing
Assignment 2

– –

Marks given in square brackets below indicate the marks available for each part. Please provide answers in the spaces below. You may word process your work if you wish, but please still insert it into the marked spaces. No marks will be lost if it is not word processed, provided it is legible. All programs should be written in `FORTRAN95`. All real arithmetic should be done in double precision (`kind = 8`). When writing/modifying subroutines, use the exact order of parameters specified in the questions and use the given filenames.

You should not discuss the details of your work with anyone else. The work which you hand in must be your own. You should be prepared to explain anything which you write to an examiner if asked to do so. In particular, if it is discovered that all or part of your code has been copied, both parties involved risk a severe penalty and might lose all their marks on the assignment.

IMPORTANT Submission Requirements: Please hand in your written solution to the assignment including **hardcopies (i.e. printouts)** of all the files you write or modify to the box (not the pigeonhole) provided on Level 1 in 4West. In addition, when you have finished the assignment, please put all the relevant files (i.e. those you copied over and those you wrote) into a directory with a distinct name that identifies you as the author, e.g. `assignment2_skl27`. This directory should contain a fully working implementation of your code. Together with your Fortran code and Makefiles, provide a `README` file explaining how to compile and run your programs. Remove any temporary and generated file (e.g. executables, .o-files and files ending in ∼). Zip up the directory into a single file using the command

```
tar czvf assignment2_skl27.tgz assignment2_skl27
```

and then upload the tgz-file that you obtain on the course moodle page at the appropriate submission point.

Nonlinear Thermal Conduction

This assignment is about practical aspects of solving sparse systems of nonlinear equations

$$\mathbf{F}(\boldsymbol{U}) \;=\; \mathbf{0} \tag{1}$$

using the inexact Newton method. In particular we will use parallel Newton–CG and look at a case study on nonlinear thermal combustion in a self–heating medium in a partially insulated square domain. This is the same model as the one considered in the first assignment. This problem arises when investigating critical parameter values for underground repositories of self–heating waste which are partially covered by buildings. Beyond certain critical parameter values the steady state solutions may become very large or even unbounded leading to an explosion in the medium. For more information see Greenway & Spence [2] and Adler [1].

Model formulation

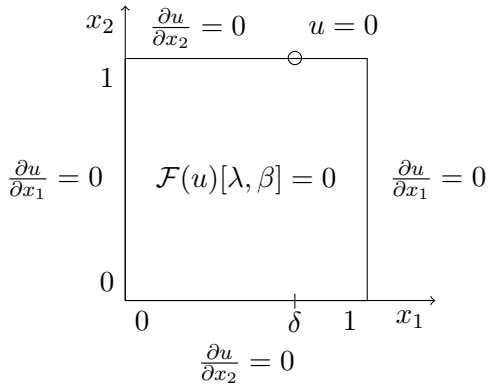
We are concerned with finding solutions of the (dimensionless) nonlinear thermal conduction equation

$$\mathcal{F}(u)[\lambda,\beta] \;:=\; -\,\frac{\partial^2 u}{\partial x_1^2} \;-\; \frac{\partial^2 u}{\partial x_2^2} \;-\; g(u)[\lambda,\beta] \;=\; 0 \qquad \text{in } \Omega := [0,1]\times[0,1], \tag{2}$$

subject to the following mixed boundary conditions on $\partial\Omega$:

$$\text{Dirichlet} \left\{ \; u \;=\; 0 \quad \text{if} \;\; x_1 > \delta \;\; \text{and} \;\; x_2 = 1 \right.$$

$$\text{Neumann} \left\{ \begin{array}{lcll} \dfrac{\partial u}{\partial x_1} & = & 0 & \text{if} \;\; x_1 = 0 \;\; \text{or} \;\; x_1 = 1 \\ \dfrac{\partial u}{\partial x_2} & = & 0 & \text{otherwise} \end{array} \right.$$



for $0 \le \delta < 1$. Note that these (more realistic) boundary conditions differ from those from Assignment 1. The nonlinear functional

$$g(u)[\lambda, \beta] \;:=\; \lambda \, \exp\left(\frac{u}{1+\beta u}\right) \tag{3}$$

is the Arrhenius reaction rate which depends on the two parameters λ and β.

The relationship between the dimensionless variables in Eqs. (2), (3) and the physical quantities is given by Adler [1]. We note that u is a dimensionless temperature excess and λ the Frank–Kamenetskii parameter; β is the dimensionless activation energy and δ the dimensionless half–width of an insulating strip.

Finite Difference Discretisation

As for the linear Poisson equation, suppose that Ω is subdivided into small equal squares of size $h \times h$ with $h := 1/m$. The interior and boundary nodes of the mesh are given by $\boldsymbol{x}_{i,j} := (ih, jh)$, with $i, j = 0, \ldots, m$. Let $m_\delta := [\delta m] \in \mathbb{N}$ be the integer part of δm (i.e. largest integer $m_\delta \le \delta m$). Then $m_\delta h \le \delta$ and $(m_\delta + 1)h > \delta$, and so all the points $\boldsymbol{x}_{i,m}$ with $i > m_\delta$ lie on the Dirichlet boundary.

The finite difference approach to the above problem is now to seek approximations $u_{i,j}$ to $u(\boldsymbol{x}_{i,j})$ such that

$$-\frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{h^2} \;-\; \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{h^2} \;-\; g(u_{i,j}) \;=\; 0.$$

We can discretise the Neumann boundary condition $\frac{\partial u}{\partial x_2} = 0$ at the nodes $\boldsymbol{x}_{i,0}$, $i = 0, \ldots, m$, by using the finite difference approximation

$$\frac{u_{i,0} - u_{i,-1}}{h} \;=\; 0 \; .$$

Therefore the second derivative at $\boldsymbol{x}_{i,0}$ becomes

$$-\frac{-u_{i,0} + u_{i,1}}{h^2} \qquad \text{instead of} \qquad -\frac{u_{i,-1} - 2u_{i,0} + u_{i,1}}{h^2}.$$

We can proceed in a similar way at the other Neumann boundary nodes $\{\boldsymbol{x}_{i,m} : i = 0, \ldots, m_\delta\}$, $\{\boldsymbol{x}_{0,j} : j = 0, \ldots, m\}$ and $\{\boldsymbol{x}_{m,j} : j = 0, \ldots, m-1\}$, e.g.

$$-\frac{u_{m-1,j} - 2u_{m,j} + u_{m+1,j}}{h^2} \qquad \text{is replaced by} \qquad -\frac{u_{m-1,j} - u_{m,j}}{h^2} \; .$$

On the other hand, zero coefficients $u_{i,m}$ for $i = m_\delta + 1, \ldots, m$, corresponding to the Dirichlet boundary nodes, are not stored at all.

We order the indices (i, j) lexicographically again, i.e. the unknowns $u_{i,j}$ are stored in a vector $\boldsymbol{U}$ in the order $(0,0), (1,0), \ldots, (m,0), (0,1), \ldots, (m,1), \ldots, (0,m), \ldots, (m_\delta, m)$. Then the above equations form a system of

$$n \;:=\; m \cdot (m+1) \;+\; m_\delta \;+\; 1$$

nonlinear equations in n unknowns (for simplicity we will not write down the dependency of $\mathbf{G}$ on λ and β from now on):

$$\mathbf{F}(\mathbf{U}) \;:=\; A_\delta\, \mathbf{U} \;-\; \mathbf{G}(\mathbf{U}) \;=\; 0 \tag{4}$$

where $\boldsymbol{U} := (u_{0,0}, u_{1,0}, \ldots, u_{m_\delta,m})^T \in \mathbb{R}^n$, $\mathbf{G}(\mathbf{U}) := (g(u_{0,0}), \ldots, g(u_{m_\delta,m}))^T \in \mathbb{R}^n$, and A_δ is the $n \times n$ matrix

$$A_\delta \;:=\; \frac{1}{h^2} \begin{pmatrix} B & -I & & & & \\ -I & C & -I & & & \\ & -I & C & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & & & \ddots & C & -I_\delta^T \\ & & & & -I_\delta & B_\delta \end{pmatrix}$$

with $(m+1) \times (m+1)$ matrices

$$B \;:=\; \begin{pmatrix} 2 & -1 & & & & \\ -1 & 3 & -1 & & & \\ & -1 & 3 & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & & & \ddots & 3 & -1 \\ & & & & -1 & 2 \end{pmatrix}, \quad C \;:=\; \begin{pmatrix} 3 & -1 & & & & \\ -1 & 4 & -1 & & & \\ & -1 & 4 & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & & & \ddots & 4 & -1 \\ & & & & -1 & 3 \end{pmatrix}.$$

The $(m_\delta + 1) \times m$ matrix I_δ is given by $[I\ 0]$, where I is the identity matrix, and the $(m_\delta + 1) \times (m_\delta + 1)$ matrix B_δ consists of the first $m_\delta + 1$ rows and columns of B.

Now the Jacobian matrix $\mathbf{F}'(\mathbf{U})$ can be written similarly to that in the first assignment,

$$\mathbf{F}'(\mathbf{U}) = A_\delta - \mathbf{G}'(\mathbf{U}), \tag{5}$$

where $\mathbf{G}'(\mathbf{U})$ is a diagonal matrix with the values $g'(u_{i,j})$ on the diagonal.

A routine which assembles the matrix A_δ for $\delta = 0.5$ will be provided. However, you will need to construct the Jacobian as defined in Eq. (5).

Inexact Newton Method

To solve the system of nonlinear equations (4) we will use the *Inexact Newton Method*:

Algorithm 1 Inexact Newton

1: Choose an *initial guess* $\boldsymbol{U}_0 \in \mathbb{R}^n$, a *nonlinear tolerance* $\tau > 0$ and *linear tolerances* $\varepsilon_k > 0$.
2: Compute $\mathbf{r}_0 = \mathbf{F}(\boldsymbol{U}_0)$.
3: **for** $k = 0, 1, 2, \ldots, k_{\max} - 1$ **do**
4: **If** $(\|\mathbf{r}_k\| \leq \tau)$ **exit**
5: Approximately solve $\mathbf{F}'(\boldsymbol{U}_k)\mathbf{s}_k = -\mathbf{r}_k$ for the Newton step $\mathbf{s}_k$
6: up to $\left\|\boldsymbol{r}_\ell^{\text{lin}}\right\| < \varepsilon_k \left\|\boldsymbol{r}_0^{\text{lin}}\right\|$, as defined in Eq. (6).
7: Solution update: $\mathbf{U}_{k+1} = \mathbf{U}_k + \mathbf{s}_k$
8: Residual update: $\mathbf{r}_{k+1} = \mathbf{F}(\mathbf{U}_{k+1})$.
9: **end for**

To (approximately) solve the linear systems in Line 5 for each k, we will use the Conjugate Gradient (CG) method with initial guess $\mathbf{0}$ and stopping criterion

$$\|\boldsymbol{r}_\ell^{\text{lin}}\| \;<\; \varepsilon_k \|\boldsymbol{r}_0^{\text{lin}}\|, \tag{6}$$

where $\boldsymbol{r}_\ell^{\text{lin}}$ denotes the linear residual after ℓ CG iterations. For an arbitrary linear system

$$A\boldsymbol{x} = \boldsymbol{b}, \tag{7}$$

the linear residual is defined as $\boldsymbol{r}_\ell^{\text{lin}} := \boldsymbol{b} - A\boldsymbol{x}_\ell$, where $\boldsymbol{x}_\ell$ is an approximate solution after ℓ iterations. The **Conjugate Gradient Algorithm** for solving system in Eq. (7) with a symmetric positive definite matrix A can be written down as follows:

Algorithm 2 Conjugate gradient (CG) method

1: Set $\boldsymbol{x}_0 = \mathbf{0}$, $\boldsymbol{r}_0^{\text{lin}} = \boldsymbol{b}$, $\boldsymbol{q}_0 = \boldsymbol{b}$.
2: **for** $\ell = 0, 1, 2, \ldots, \ell_{\max} - 1$ **do**
3: $\alpha_\ell = ((\boldsymbol{r}_\ell^{\text{lin}})^T \boldsymbol{r}_\ell^{\text{lin}})/(\boldsymbol{q}_\ell^T A \boldsymbol{q}_\ell)$ ▷ search parameter
4: $\boldsymbol{x}_{\ell+1} = \boldsymbol{x}_\ell + \alpha_\ell \boldsymbol{q}_\ell$ ▷ update solution
5: $\boldsymbol{r}_{\ell+1}^{\text{lin}} = \boldsymbol{r}_\ell^{\text{lin}} - \alpha_\ell A \boldsymbol{q}_\ell$ ▷ update residual
6: **If** $(\|\boldsymbol{r}_{\ell+1}^{\text{lin}}\| < \varepsilon_k \|\boldsymbol{r}_0^{\text{lin}}\|)$ **exit**
7: $\beta_{\ell+1} = ((\boldsymbol{r}_{\ell+1}^{\text{lin}})^T \boldsymbol{r}_{\ell+1}^{\text{lin}})/((\boldsymbol{r}_\ell^{\text{lin}})^T \boldsymbol{r}_\ell^{\text{lin}})$
8: $\boldsymbol{q}_{\ell+1} = \boldsymbol{r}_\ell^{\text{lin}} + \beta_{\ell+1} \boldsymbol{q}_\ell$ ▷ update search direction
9: **end for**

Note that the dot product of any two vectors $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{R}^n$ is defined as $\boldsymbol{a}^T\boldsymbol{b} := \sum_{j=1}^n a_j b_j$ and hence the quotients α_ℓ and $\beta_{\ell+1}$ are real numbers.

A routine which implements Alg. 2 will be provided. However, you will need to write a function to compute dot products of distributed vectors.

Two reasonable choices for the linear tolerances ε_k in the stopping criterion in Eq. (6) are:

1. fixed tolerance, e.g. $\varepsilon_k = 10^{-5}$
2. variable tolerance: $\varepsilon_k = \min\left(\bar{\varepsilon}, \gamma \cdot \dfrac{\|\mathbf{F}(\mathbf{U}_k)\|^2}{\|\mathbf{F}(\mathbf{U}_{k-1})\|^2}\right)$ with $\varepsilon_0 = \bar{\varepsilon} = 0.1$ and $\gamma = 0.9$.

In this assignment we will only investigate the second option. For more details on the inexact Newton method and on the iterative solution of systems of nonlinear equations in general see Kelley [3].

The Assignment

Copy over the files from the directory `$MA40177_DIR/assignment2`, which are designed to solve numerically the model problem defined above (since I'm in the US at the moment, there might be a small delay with making these files available, but they will be there by Wed 19th April at the latest).

`main.f90` - main program

`header.f90` - module containing the (parallel) sparse data structures (as in lectures)

`laplace.f90` - contains the routine `Laplace()` which sets up the matrix $A_{0.5}$ in compressed row format

`save_solution.f90` - contains a routine that writes the solution vector $\boldsymbol{U}$ to a file for post-processing in `Python` (works only in sequential mode, i.e. for one processor)

`maximum.f90` - contains a function `Maximum()` which calculates the maximum of a distributed vector

`newton.f90` - a skeleton for the `Newton()` routine

`func.f90` - a skeleton for the `Func()` routine

`jacobian.f90` - a skeleton for the `Jacobian()` routine

`cg.f90` - contains the routine `CG()` which implements the CG method for the iterative solution of sparse linear systems in compressed row storage format. For a description of the arguments that `CG()` takes and of their `intent` see the comments at the beginning of `cg.f90`

`matmult.f90` - contains the routine `Mat_Mult()` which implements the *sequential* sparse matrix vector product

`vecdot.f90` - a skeleton for the `Vec_Dot()` function.

`Makefile` - Makefile to compile and link all the necessary files on the cluster

`visualise.py` - Python program to visualise u as a 3D surface plot

Note that the subroutines in `func.f90`, `jacobian.f90` and `newton.f90` which implement the nonlinear function, its Jacobian and the inexact Newton method for solving Eq. (1), as well as the function in `vec_dot.f90` which implements the dot product, are empty. It will be your task to implement them. You will also need to parallelise the sequential code in `matmult.f90`. You might also have to use other subroutines from the lectures/tutorials and need to update the Makefile accordingly.

Implementation

Q1: Dot product of parallel vectors

Algorithm 2 contains a lot of dot products. Write a function

```
Vec_Dot(a,b) result (d)
```

in the file `vecdot.f90` which implements a parallel version of the dot product of two vectors. It should take as input two distributed vectors **a** and **b** (each process should access only a local part of each vector, encapsulated in an appropriate derived type, cf. `header.f90`) and return $d = \sum_{i=1}^{n} a_i b_i$, the dot product. The value d has to be returned on each processor.

Try to make your function as efficient as possible. Test that it returns correct results in simple cases that you can calculate yourself.

[4 points]

Q2: Nonlinear residual function and Jacobian matrix

Write subroutines `Func()` and `Jacobian()` in the files `func.f90` and `jacobian.f90`, respectively, which compute the function in Eq. (4) and its Jacobian matrix in Eq. (5). Note that the Jacobian must be constructed in the parallel Compressed Row Storage format, the same format as the one used in `Laplace()`. *Hint:* `Laplace()` stores each diagonal element of A_δ as the first entry in the respective compressed row.

[5 points]

Q3: Parallelisation of the matrix-vector product

The subroutine `Mat_Mult()` in the file `matmult.f90` contains the sequential code for multiplication of a CRS matrix by a vector. Introduce necessary changes to make this code parallel and efficient.

[3 points]

Q4: Parallel inexact Newton method

Write the subroutine `Newton()` in the file `newton.f90` which implements the inexact Newton method (Alg. 1), using the CG Algorithm (Alg. 2) for solving the linearised systems in Step 5 of Alg. 1. The subroutine should take the following arguments:

- `A`: the real matrix A_δ in the parallel CRS format
- `u`: the real solution vector $\mathbf{U}$ in the parallel vector storage format
- `lambda`: a real Frank–Kamenetskii parameter
- `beta`: a real activation energy parameter
- `tau`: a real stopping tolerance for Alg. 1
- `kmax`: an integer maximum number of iterations of Alg. 1
- `its`: an integer actual number of iterations of Alg. 1 conducted
- `ierr`: an integer output parameter which should be 0 if the Newton method has converged, and a nonzero value if it has terminated prematurely.

For example, make sure that the Newton method terminates safely in the case of divergence or if the solution becomes negative, and returns a nonzero `ierr` in this case. Try to make your routine as efficient as possible. For simplicity, you can always allow $\ell_{\max} = 9999$ iterations for the CG algorithm. Make sure first that your subroutine works on one processor (partial credit will be given for a working sequential code).

[12 points]

Q5: Main program and report

Q5(a) Make the main program in `main.f90` more user–friendly by allowing the user to specify m and λ in a file `input.dat`. Write a `jobscript.slm` file for running your code through the scheduler on the cluster. Verify that your program works on one processor first: the Newton method should converge for the default parameters $\lambda = \beta = 0.25$, and the solution should be nonnegative. You can use the Fortran subroutine in `save_solution.f90` and the Python script `visualise.py` to plot the solution to a PDF file. Bear in mind that `save_solution.f90` is a sequential code.

[3 points]

Q5(b) Write a 0.5-1 page report describing how you implemented, modified and tested the codes and why your particular implementation is efficient. What further improvements can you suggest? Justify your reasoning by theoretical complexity estimates or numerical evidence.

[4 points]

To test for a working sequential/parallel algorithm, it was required to see if **newton** produces an accurate solution $\mathbf{U}$. This was done by considering a small 2-dimensional problem in which the exact solution $\mathbf{U}_* = (1, 1)$ is known. The program **newton** was adapted by integrating two new testing subroutines to calculate

$$\mathbf{F}(U_1, U_2) = \begin{pmatrix} U_1 + U_1^2 + U_2^2 - 3 \\ U_2 + 2U_1U_2 - 3 \end{pmatrix} \quad \text{and} \quad \mathbf{F}'(U_1, U_2) = \begin{pmatrix} 1 + 2U_1 & 2U_2 \\ 2U_2 & 1 + 2U_1 \end{pmatrix}$$

In fact, **newton** converged in one iteration with $||\mathbf{U}_1 - \mathbf{U}_*||_2 = 0$.

Next, it was required to test the efficiency of **newton** sequentially. This was done by computing the average time taken per iteration of the iterative loop in the Inexact Newton algorithm. In order for the best possible performance, LAPACK and BLAS functions/routines where used to replace FORTRAN intrinsic functions/procedures. For memory efficiency, DCOPY() was used when copying/initiating vectors. DSCAL(), DAXPY() and DDOT() were used for efficient scaling, multiplication and addition of vector elements.

m	Time per newton iteration	u_{max}
2	1.58E-005	0.717
4	6.25E-005	0.381
8	2.81E-004	0.287
16	2.18E-003	0.251
32	1.57E-002	0.236
64	0.119	0.229
128	0.811	0.226

Time increases by a factor of 4 when doubling $m \in \{2, 4\}$. This is approximately consistent with the estimated cost per iteration of the CG method $\mathcal{O}(m^2)$ (which is the most computationally costly routine). Then, time grows more rapidly, as a factor of 8, when doubling $m \in \{8, 16, 32\}$ ($\approx \mathcal{O}(m^3)$). This is unexpected and suggests potential inefficiencies in the programmed Inexact Newton algorithm (suggests areas for improvement in the algorithm for large scale problems).

Then, it was required to test the parallel implementation for different numbers of processors P, and verify that the result does not change significantly for different P (checking that implantation works when n does not divide P and is accurate). In **Mat_Mult()**, **MPI_ALLGATHERV()** was used. This is not very efficient, particularly for large scale problems. This is because all distributed non zero vector elements from each row are sent to each process, and not necessarily used by each processor. A modified **sparsegather()** would have been significantly more efficient, since only the elements required by each neighbouring process would be sent and received.

Number of Processors	Time per newt. iter. (s) m=32	Time per newt. iter. (s) m=128	u_{max} for m=32	u_{max} for m=128
2	8.14E-003	0.437	0.236	0.226
4	4.64E-003	0.253	0.236	0.226
8	2.88E-003	0.134	0.236	0.226
16	1.99E-003	8.41E-002	0.236	0.226
32	1.92E-003	5.81E-002	0.236	0.226

It is verified that the result does not change significantly for P, and that the implementation is working when 'n' divides the number of processes. Notice that despite the inefficiencies of the **MPI_ALLGATHERV()** implementation, there is certainly significant speed up for the larger $m = 128$ as the number of processors increases. This is observed to a lesser extent for the smaller $m = 32$.

Nonlinear case study

Q6: Solution

Fix the parameters $\delta = 0.5$, $\tau = 10^{-5}$, $k_{\max} = 20$, $\lambda = 0.25$ and $\beta = 0.25$. Find approximations to the maximum temperature $u_{max} := \max_{\mathbf{x}\in\bar{\Omega}} u(\mathbf{x})$ in Ω using the inexact Newton method for (2) for increasing values of $m = 32, 64, 128, \ldots$ (you can use one processor in this question). Stop increasing m when the third decimal place of the approximation of u_{max} does not change anymore. Record your values of m and u_{max} (with at least 4 digits after the decimal point) in the following table.

m	u_{max}
256	0.2249

Produce a plot of the solution $\boldsymbol{U}$ by running your program for $m = 64$ and invoking `python visualise.py` in the command line on the cluster. Print `solution.pdf` after copying it to your Windows directory and attach the plot on a separate sheet.

[3 points]

Q7: Performance Model

Q7(a) Estimate the number of FLOPs and the number of memory references in one iteration of the CG Algorithm 2 as functions of m, assuming $\delta = 0.5$. Using the values of t_{flop} and t_{mem} from the *Handout on Processor and Network timings*, estimate the time t_{seq} of one iteration of CG on one process as a function of m.

Q7(b) Modify `cg.f90` such that the subroutine returns the number of iterations conducted, and `newton.f90` such that it prints the total number of all CG iterations. Measure the total time and deduce the time of one CG iteration. Compare the theoretically predicted time with the measured sequential time for $m = 128$ and $m = 512$.

Q7(c) Now estimate in a similar way the communication time for each processor in one CG iteration, again as a function of m and the number of processes P. Use the values of t_{lat} and t_{word} from the handout. Plot the theoretically predicted speedup for the parallel implementation of Algorithm 2 and the parallel efficiency $E(n, P)$ (where $n = m^2$) for both $m = 128$ and $m = 512$, and the numbers of processes $P = 1, 2, 4, 8, 16, 32$.

[4 points]

Insert your answer for Q7 here and attach plots on separate sheets

Note: $n = m*(m+1)+m_\delta+1 \approx m^2$ so use this when referencing 'function of m'

7a) Assume that $\bar{n}_{nz} \approx 5$ is much less than n. Number of FLOPS and memory references per iteration of CG,

- **mat_mult()** ($A\mathbf{q}_l$): From PS6, the number of FLOPS is $2\bar{n}_{nz} \cdot n$, and the number of memory references is $2n + 3\bar{n}_{nz}/2 + (n+1)/2 \approx 10n$
- Updating $\mathbf{x}_{l+1}$, $\mathbf{r}_{l+1}^{lin}$ and $\mathbf{q}_{l+1}$ (constant $\times$ vector + constant): $3 \times n$ multiplications and $3 \times n$ additions. This implies $6n$ many FLOPS. There are also $3n$ memory references required.
- Computing/updating scalar values α_l and β_{l+1}: There are two **Vec_Dot()** calls $\Rightarrow 2 \times n$ many multiplications and additions. Therefore $4n$ FLOPS and $2n$ memory references in total

Note that there are also three division operations. Therefore, the total number of FLOPS is $20n + 3 \approx 20n = 20m^2 = \mathcal{O}(m^2)$; and the total number of memory references is $15n \approx 15m^2 = \mathcal{O}(m^2)$. $t_{\text{mem}} = 2.74 \cdot 10^{-10}s$ and $t_{\text{FLOP}} = 1.77 \cdot 10^{-11}s$. Hence,

$$t_{\text{seq}}(n) \approx (20t_{\text{FLOP}} + 15t_{\text{mem}})n = 4.46 \cdot 10^{-9}n = \mathcal{O}(m^2)$$

7b)

m	Theoretically predicted time (P=1)	Measured time (P=1)
128	7.39E-005	2.68E-003
512	1.17E-003	4.10E-002

The predicted times are approximately 36 and 35 times smaller than the measured times for 'm' at 128 and 512 respectively.

7c) Proceed to compute $t_{\text{par}}(n,P)$ by computing t_{comm} for **MPI_ALLGATHERV()** since it was used in Q3 (butterfly topology). Using that $t_{\text{lat}} = 1.34 \cdot 10^{-6}s$ and $t_{\text{word}} = 2.70 \cdot 10^{-9}s$,

$$t_{\text{comm}}(n,P) \approx 1.34{\cdot}10^{-6}{\cdot}\log P + 2.70{\cdot}10^{-9}{\cdot}\left(\frac{P-1}{P}n\right) \Rightarrow t_{\text{par}}(n,P) = 4.46{\cdot}10^{-9}{\cdot}\frac{n}{P} + t_{\text{comm}}(n,P)$$

The predicted speedup for the parallel implementation is $S(n,P) = t_{\text{seq}}(n)/(t_{\text{par}}(n,P))$. Therefore, the parallel efficiency (where $n = m^2$) is $E(n,P) = t_{\text{seq}}(n)/(P \cdot t_{\text{par}}(n,P))$

Q8: Influence of the nonlinearity parameters λ and β

Let $m = 100$, $\tau = 10^{-5}$, $k_{\max} = 20$, $\beta = 0.25$ and $\delta = 0.5$ as above. Starting with $\lambda = 0$ and using a sufficiently small step size $\Delta\lambda$, calculate an approximation to u_{max} for a sequence of values of $\lambda \in [0, 1.5]$. **Use the solution U at the previous value of λ as the initial guess** for computing the solution at the next value of λ. Note that if $\Delta\lambda$ is not small enough, the method may diverge for larger λ. Produce a graph of u_{max} as a function of λ and hand in a hardcopy of the plot.

Now choose $\beta = 0.1$ and calculate approximations to u_{max} starting again with $\lambda = 0$ and then increasing λ. The Newton method should fail to converge at a certain critical value λ_c. Calculate λ_c and explain why Newton fails to converge.

[4 points]

Insert your comments for Q8 here

For $\beta = 0.1$, starting from $\lambda = 0$ and taking small steps of size $\Delta\lambda = 0.01$, at $\lambda_c = 0.66$ divergence is first observed. Notice that the computed λ_c induces a bifurcation; a steady state for the system is reached for $\lambda < \lambda_c$, but no steady state is obtained for $\lambda > \lambda_c$. Divergence is therefore observed at λ_c because the Jacobian of F becomes singular (this would imply that the 's' computed by CG would not be a decent direction, hence divergence).

For the following parallel scaling experiments, always allocate processors on a single node.

Q9: Strong scaling

Set $m = 512$ and calculate the solution on $P = 1, 2, 4, 8, 16, 32$ processors. Verify that the result does not change significantly for different P. Measure the time spent in one iteration of CG for different numbers of processors (hint: measure the total time and divide by the number of iterations). Plot both the time and the parallel efficiency $E(n, P)$, where $n = m^2$. Repeat the experiment for two different values of m (e.g. $m = 128$ and $m = 512$). Compare the measured times to the theoretical estimates in Q7. Explain your observations.

Attach scaling plots on a separate sheet and write your explanations below

[4 points]

Q10: Weak scaling

Starting from $m = 128$, successively double the number of points m in each direction and at the same time increase the number of processors by a factor of four (i.e. keep the problem size on a single processor nearly the same). Perform this experiment for $P = 1, 4, 16$ processes and measure the time spent in one iteration of the CG algorithm. Then repeat the same experiment for $P = 2, 8, 32$, starting with $m = 181 \approx \sqrt{2} \cdot 128$ on $P = 2$ processors. Plot the measured time per CG iteration as a function of the number of processors P and plot the scaled efficiency $E_s(n, P)$. Explain your observations.

Attach scaling plots on a separate sheet

[4 points]

Write here a short report, summarising your results in Q9 and Q10

m	u_max for p=1	u_max for p=2	u_max for p=4	u_max for p=8	u_max for p=16	u_max for p=32
512.0	0.224422	0.224422	0.224422	0.224422	0.224422	0.224422

It is verified that the result does not change significantly for different P.

Number of Processors	Pred. time (s) m=32	Meas. time (s) m=32	Pred. time (s) m=128	Meas. time (s) m=128	Pred. time (s) m=512	Meas. time (s) m=512
1.0	0.000005	0.000199	0.000073	0.002677	0.001169	0.040995
2.0	0.000005	0.000103	0.000060	0.001443	0.000939	0.020879
4.0	0.000005	0.000058	0.000053	0.000835	0.000825	0.010794
8.0	0.000006	0.000036	0.000051	0.000444	0.000768	0.005686
16.0	0.000007	0.000025	0.000050	0.000278	0.000740	0.003179
32.0	0.000007	0.000024	0.000050	0.000192	0.000727	0.002062

Notice that the predicted times are significantly smaller than the measured times. It was expected that the predicted time would be a lower bound, since when computing the estimates, perfect cashing was assumed (this assumption is particularly important in **mat_mult()**). The times, combined with the plots indicates that the implementation has very poor parallel scaling. This was certainly expected as the implementation was based on **MPI_ALLGATHERV()** (as discussed before).

10) Observe that the scaled efficiency plot indicates poor weak scaling. This was expected since as the number of processors increases, increasingly large amounts of data is required to be exchanged between processors. As 'm' increases, **MPI_ALLGATHERV()** requires all processors to receive distributed vector components, even if they are not required for computation. This results in increasing overhead communications, which is observed in the 'measured time per CG iter. as a function of P' plot, a significant increase in time per iteration is observed as the number of processors grows with the larger problem.

General Remarks

- Write routines and programs to test parts of your code.
- Support your observations with numerical results. You can also use graphs.
- Typical trends for convergence and cost with respect to some variable n are αn^{β} and $\alpha\beta^{n}$ for some constants $\alpha, \beta \in \mathbb{R}$.
- Use as many subroutines and functions as you deem necessary.
- Choose helpful names for variables, subroutines, functions, etc.
- Provide comments in your code where appropriate.
- Make sure your code is properly indented. Your text editor may be able to help you with this.
- "Programs must be written for people to read, and only incidentally for machines to execute.", Abelson & Sussman, Structure and Interpretation of Computer Programs
- Try to make sure that it is easy to repeat the experiments you used to verify your code and to generate the results you use in your report.

References

[1] Adler J., Thermal–explosion theory for a slab with partial insulation, *Combustion and Flame* **50**, 1983, pp. 1–7. (Library: `PER66`)

[2] Greenway P. and Spence A., Numerical calculation of critical points for a slab with partial insulation, *Combustion and Flame* **62**, 1985, pp. 141–156. (Library: `PER66`)

[3] Kelley CT., *Iterative Methods for Linear and Nonlinear Equations*, SIAM, Philadelphia, 1995. (Library: `512.978KEL`)