Coursework cover sheet - MA40177 Coursework 1

General instructions

Set Fri, 10 Mar 2023

Due Tue, 28 Mar 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 1

– –

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. `assignment1_skl27`. This directory should contain fully working implementations of the code from Questions 1,2 and 4. 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 (e.g. `assign1_smith`) into a single file using the command

```
tar czvf assignment1_skl27.tgz assignment1_skl27
```

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

1 Nonlinear Thermal Conduction – (Quasi-)Newton Methods

This assignment is about practical aspects of iterative methods for nonlinear systems of the form

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

with a given function $\mathbf{F} : \mathbb{R}^n \to \mathbb{R}^n$, using Newton's method and quasi-Newton type methods.

1.1 Background

As a motivation, consider nonlinear thermal combustion in a self–heating medium. 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].

In nonlinear heat conduction, we are concerned with finding solutions of the (dimensionless) nonlinear thermal conduction equation

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

Mathematically, both $\mathcal{F}$ and g are *functionals* which map a given function $u : \Omega \to \mathbb{R}$ to a different function, subject to parameters λ, β. The nonlinear functional g is defined as

$$g(u)[\lambda,\beta](\boldsymbol{x}) \;:=\; \lambda\, \exp\left(\frac{u(\boldsymbol{x})}{1+\beta u(\boldsymbol{x})}\right) \;+\; 100\sin(\pi x_1)\sin(\pi x_2). \tag{3}$$

It contains (as the first term) the Arrhenius reaction rate which depends on the two parameters λ and β and (as the second term) the source/sink term. We will always choose $\lambda = 0.19$ and $\beta = 0.12$. For simplicity we assume that $u = 0$ on the boundary of Ω. Note that a simpler, linear version of this is discussed in the *Tutorial 4 – Poisson's Equation [PE].*

The relationship between the dimensionless variables in Eqs. (2) and (3) and the physical quantities are given by Adler [1]. We note that u is a dimensionless temperature excess, λ the Frank–Kamenetskii parameter, and β is the dimensionless activation energy. For simplicity, we will not write down the dependency of $\mathcal{F}$ and g on λ and β in the following.

Applying finite difference discretisation on a uniform grid with a step size $h = 1/m$ in each direction, as described in [PE], we obtain a discrete version of Eq. (2) at each point in the domain:

$$-\left(u_{i-1,j} - 2u_{i,j} + u_{i+1,j}\right) - \left(u_{i,j-1} - 2u_{i,j} + u_{i,j+1}\right) \;-\; h^2 g(u_{i,j}) = 0,\;\; i,j = 1,\ldots m-1,$$

where $u_{i,j}$ is an approximation to $u(\boldsymbol{x}_{i,j})$, $\boldsymbol{x}_{i,j} = (ih, jh)$ are the interior nodes of the mesh and $g(u_{i,j})$ is the discrete equivalent of $g(u)[\lambda,\beta](\boldsymbol{x}_{i,j})$. Following [PE] and ordering the indices (i,j) in lexicographical order we can write this in the form

$$\mathbf{F}(\mathbf{U}) \;=\; \mathbf{0}\,, \quad \text{where} \quad \mathbf{F}(\mathbf{U}) \;:=\; A\mathbf{U} - \mathbf{G}(\mathbf{U})\,, \tag{4}$$

with A, the discretisation of the Laplacian, as in [PE]. Equation (4) is a nonlinear system of the form (1) for the vector $\mathbf{U} = (U_1,\ldots,U_n) = (u_{1,1}, u_{2,1}, \ldots, u_{m-1,m-1}) \in \mathbb{R}^n$ with $n = (m-1)^2$. There is one equation in Eq. (4) corresponding to each interior node $\boldsymbol{x}_{i,j}$, $i,j = 1,\ldots m-1$. The vector $\mathbf{G}(\mathbf{U})$ consists of the entries $g(u_{i,j})$.

For the remainder, we introduce the following notation. For any (column) vector $\mathbf{U} \in \mathbb{R}^n$, $\mathbf{U}^T$ denotes its transpose (which is a row vector) and $\|\mathbf{U}\|$ denotes its 2-norm, that is $\|\mathbf{U}\| = \left(U_1^2 + \ldots + U_n^2\right)^{1/2} = \sqrt{\mathbf{U}^T\mathbf{U}}$. For $k = 1,\ldots,n$, $F_k(\mathbf{U})$ denotes the kth component of $\mathbf{F}(\mathbf{U})$, i.e. $\mathbf{F}(\mathbf{U}) = (F_1(\mathbf{U}),\ldots F_n(\mathbf{U}))^T$. Also, $\mathbf{F}'$ denotes the **Jacobian** matrix of $\mathbf{F}$, i.e.

$$\Big(\mathbf{F}'(\mathbf{U})\Big)_{k,\ell} = \frac{\partial F_k(\mathbf{U})}{\partial U_\ell} \qquad \text{for } \mathbf{U} \in \mathbb{R}^n.$$

Newton's method for solving Eq. (1) iteratively is now defined as follows.

Algorithm 1 Newton method

1: Choose an *initial guess* $\mathbf{U}_0 \in \mathbb{R}^n$ and a *tolerance* $\tau > 0$.
2: Set $\boldsymbol{r}_0 = \mathbf{F}(\mathbf{U}_0)$.
3: **for** $k = 0,\ldots,K_{\max}-1$ **do**
4: **If** ($\|\boldsymbol{r}_k\| \le \tau$) **exit**
5: Solve $\mathbf{F}'(\mathbf{U}_k)\mathbf{s}_k = -\boldsymbol{r}_k$ for the *Newton step* $\mathbf{s}_k$.
6: Solution update: $\mathbf{U}_{k+1} = \mathbf{U}_k + \mathbf{s}_k$.
7: Residual update: $\boldsymbol{r}_{k+1} = \mathbf{F}(\mathbf{U}_{k+1})$.
8: **end for**

Some properties of Newton's method are given in the following theorem, which you can assume:

Theorem. *Suppose each component of the function* $\mathbf{F}$ *has two continuous derivatives on* $\mathbb{R}^n$ *and suppose Eq.* (1) *has a solution* $\mathbf{U}^*$*. Suppose also that* $\mathbf{F}'(\mathbf{U}^*)$ *is nonsingular. Then there exists a neighbourhood of* $\mathbf{U}^*$ *such that the iterates* $\mathbf{U}_k$ *produced by Newton's method are well-defined and converge to* $\mathbf{U}^*$*, from any initial guess in this neighbourhood. Moreover*

$$\|\mathbf{U}^* - \mathbf{U}_{k+1}\| \;\leq\; C\|\mathbf{U}^* - \mathbf{U}_k\|^2$$

for some constant C *independent of* k*, i.e. the rate of convergence is quadratic.*

For more details on the theory of Newton's method and on the iterative solution of systems of nonlinear equations in general see Kelley [3].

1.2 Improvements: Quasi-Newton method

When the dimension of the Jacobian matrix is large, solving the linear equation in Line 5 of Alg. 1 might be numerically expensive. The idea of Quasi-Newton methods is to replace the true Jacobian $\mathbf{F}'(\mathbf{U}_k)$ in Line 5 by some approximation, an $n \times n$ matrix B_k which is easier to assemble and invert. Here we consider the following specific Quasi-Newton method:

Algorithm 2 Quasi-Newton method (general description)

1: Choose an *initial guess* $\mathbf{U}_0 \in \mathbb{R}^n$, an initial approximate Jacobian B_0 and a *tolerance* $\tau > 0$.
2: Set $\boldsymbol{r}_0 = \mathbf{F}(\mathbf{U}_0)$.
3: **for** $k = 0, \ldots, K_{\max} - 1$ **do**
4: **If** ($\|\boldsymbol{r}_k\| \leq \tau$) **exit**
5: Solve $B_k \mathbf{s}_k = -\boldsymbol{r}_k$ for the *Quasi-Newton step* $\mathbf{s}_k$.
6: Solution update: $\mathbf{U}_{k+1} = \mathbf{U}_k + \mathbf{s}_k$.
7: Residual update: $\boldsymbol{r}_{k+1} = \mathbf{F}(\mathbf{U}_{k+1})$.
8: Approximate Jacobian update: $B_{k+1} = B_k + \mathbf{p}_k \mathbf{q}_k^T$,
9: where $\mathbf{p}_k = \boldsymbol{r}_{k+1}/\|\mathbf{s}_k\|$ and $\mathbf{q}_k = \mathbf{s}_k/\|\mathbf{s}_k\|$.
10: **end for**

In fact, in Alg. 2 the matrix B_k does not even need to be stored explicitly. Using Eq. (6) from Q3(d) (which you will need to prove), we can conclude (see Q4(a)) that only a system of equations with B_0 needs to be solved explicitly. So one can *precompute* some information that can make this solution efficient (such as LU factors), and/or choose a simple form of B_0. Possible (but not the only) choices for B_0 are the identity matrix, $\mathbf{F}'(\mathbf{U}_0)$ or the diagonal of $\mathbf{F}'(\mathbf{U}_0)$. On the other hand, we need to store all the previously computed Quasi-Newton steps $\mathbf{s}_0, \ldots, \mathbf{s}_k$, so the maximum number of iterations $K_{\max}$ should be limited to reduce memory requirements.

2 The Assignment

Copy over all files from the directory `$MA40177_DIR/assignment1`, which form a skeleton for your assignment:

- `laplace.f90` creates a $n \times n$ finite difference Laplace matrix A [PE], where $n = (m-1)^2$.
- `func.f90` computes the residual vector $\mathbf{F}(\mathbf{U})$, given a vector $\mathbf{U} \in \mathbb{R}^n$.
- `jacobian.f90` computes the Jacobian $\mathbf{F}'(\mathbf{U})$, given a vector $\mathbf{U} \in \mathbb{R}^n$.
- `newton.f90` a skeleton of the program for Newton's method (Alg. 1).

Q1: Newton's method

Q1(a) **Write the main program** which solves Eq. (1) by Newton's method (Algorithm 1) in the file `newton.f90`. This program should allow the user to specify any m and tolerance τ and should make calls to subroutines or functions which return values of $\mathbf{F}(\mathbf{U})$ and $\mathbf{F}'(\mathbf{U})$, for any input $\mathbf{U}$. The initial guess for the solution should be $\mathbf{U} = \mathbf{0}$. Use suitable calls to `BLAS` and to `LAPACK` and make your program as efficient as you can wherever possible by employing the most compact storage format for the Laplace matrix A and the Jacobian $\mathbf{F}'(\mathbf{U})$.
Hint: you are **expected** to modify `laplace.f90`, `func.f90` and `jacobian.f90` accordingly. Full points will be awarded for using the most efficient format. However, partial points will be given for any correct and consistent implementation, using e.g. the dense $n \times n$ matrix format, so you may decide to implement it first. You should also comment your program and make sure it exits gracefully, if Newton's method fails to converge. **Write a** `Makefile` that can be used to compile and link an executable `newton` which solves the problem given by Eq. (4).

[11 points]

Q1(b) Test your code on simplified problems where you know the behaviour of the method and/or the exact solution, and can check your code against this theory. You may consider the following tests:

- Take only the linear part of Eq. (4) (i.e. $\lambda = 0$).
- Compare different formats for A and $\mathbf{F}'$ and the corresponding `BLAS`/`LAPACK` routines.
- Consider a small 2 dimensional problem $\mathbf{F}(\mathbf{U}) = \mathbf{F}(U_1, U_2) = \mathbf{0}$, where

$$\mathbf{F}(\mathbf{U}) = \begin{pmatrix} U_1 + U_1^2 + U_2^2 - 3 \\ U_2 + 2U_1U_2 - 3 \end{pmatrix}, \tag{5}$$

  with the exact solution $\mathbf{U} = (1, 1)$.

Write a short **report** (about 1 page) on how you tested your code and how you made it efficient. You do not need to submit your test codes, but the report should consider at least two of these tests, stating *what exactly* you checked with each test, what you expected to observe and what you actually observed. Similarly, in the efficiency report you may consider only the most time consuming parts of the code, but you need to explain *why* these parts are most relevant for the performance and why your particular way of coding makes them efficient. Identify any other issues that may arise for the user and suggest how you would deal with them in the design of your program.

[4 points]

To ensure that the algorithm design is correct, it was required to see if **newton** produces an accurate solution $\mathbf{U}$. This was done by considering a small 2-dimensional problem, as described in equation (5) , in which the exact solution $\mathbf{U}_*$ is known. The program **newton** was adapted by integrating two new testing subroutines to calculate $\mathbf{F}(U_1, U_2)$ and $\mathbf{F}'(U_1, U_2) = \begin{pmatrix} 1+2U_1 & 2U_2 \\ 2U_2 & 1+2U_1 \end{pmatrix}$. The solution to $\mathbf{F}'(\mathbf{U}_k)\mathbf{s}_k = -\mathbf{r}_k$ was solved using the LAPACK function DGESV. Then, setting $\tau = 1.0E-7$ ($\mathbf{U}$ computed in 7 iterations), the relative error in the Frobenius norm was computed

$$\frac{||\mathbf{U} - \mathbf{U}_*||_2}{||\mathbf{U}_*||_2} = 1.343E - 11$$

which is very small as expected.

Next, proceed use **newton** to solve a similar equation to (4). Then, testing the performance of various drivers, error of central difference approximation, and accuracy of computed solution $\mathbf{U}$

$$-\frac{\partial^2 u}{\partial y_1^2} - \frac{\partial^2 u}{\partial y_2^2} - f = 0 \quad \text{with} \quad f(\boldsymbol{y}) = (3y_1 + y_1^2)e^{y_1} y_2(1-y_2) + 2y_1(1-y_1)e^{y_1}.$$

It can be proved that the exact solution is $u(\boldsymbol{y}) = y_1(1-y_1)e^{y_1}y_2(1-y_2)$. The most consuming part of newtons method is solving $\mathbf{F}'(\mathbf{U}_k)\mathbf{s}_k = -\mathbf{r}_k$ for $\mathbf{s}_k$. In order for the best possible performance, LAPACK routines were used. Note that for the dense and banded drives, DGEMV and DSBMV (Level 2 BLAS) were used when calculating $\mathbf{F}(\mathbf{U}_k)$ respectively (ie $A\mathbf{U} - \mathbf{G}'(\mathbf{U})$). The average time for one run of the iteration loop containing drivers DGESV (dense) and DPBSV (banded) was measured. Note that the Jacobian must be SPD. This is because if it was not SPD, then the Cholesky decomposition wouldn't have worked (the 'info' variable in the subroutine DPBSV would have been non-zero).

m	DGESV (s)	DPBSV (s)	error at $\mathbf{u}(1/2, 1/2)$	Relative error
8	1.68E-4	1.45E-4	4.12E-4	3.97E-3
16	7.54E-4	1.85E-4	1.03E-4	9.94E-4
32	9.38E-3	4.90E-4	2.58E-5	2.49E-4
64	0.48E+0	2.82E-3	6.45E-6	6.22E-5
128	killed	1.25E-2	1.61E-6	1.55E-5

Notice that for the dense driver, the time increases dramatically with m. This rate is approximately consistent with the cost of the LU decomposition $\mathcal{O}(n^3) = \mathcal{O}(m^6)$. The banded driver (using the Banded Cholesky decomposition) is significantly faster, with approximately $\mathcal{O}(n^2) = \mathcal{O}(m^4)$. Also, notice that as $m \to \infty$, the error approaches 0 ($\mathcal{O}(h^2)$ convergence), and the relative error becomes significantly more accurate (the solution $\mathbf{U}$ becomes accurate to more decimal places). Proceed to test the algorithm when solving equation (4).

m	DGESV (s)	DPBSV (s)
8	1.39E-4	8.23E-5
16	7.78E-4	1.48E-4
32	1.60E-2	4.96E-4
64	2.84E+0	5.18E-3
128	killed	2.30E-2

Conclude that, as expected, the most efficient implementation was the banded SPD implementation for computing the solution to $\mathbf{F}'(\mathbf{U}_k)\mathbf{s}_k = -\mathbf{r}_k$ (implementation using DSBMV and DPBSV). Note that using a banded symmetric implementation is also significantly more memory efficient. The dimensions of A and $\mathbf{F}'(\mathbf{U}_k)$ reduce to $m \times (m-1)^2$ from $(m-1)^2 \times (m-1)^2$ (taken advantage of in DSBMV when computing $A\mathbf{U}$ since unnecessary 0 multiplications from dense implementation ignored).

Common user issues may arise from imputing a large m when there is limited memory available; choosing a initial guess $\mathbf{U}_0$ which is not in the neighbourhood of $\mathbf{U}^*$; and the algorithm prematurely terminating even when $\mathbf{U}_k$ is not close to $\mathbf{U}_*$. It is therefore recommended that the user makes sure that the system has enough memory before running, an option to not save r_k was also included; the $\mathbf{U}_0$ is automatically chosen to be inside the neighbourhood; allowing the user to pick a suitable tolerance, with a predetermined suitable maximum iteration stopping condition.

Q2: numerical convergence study

Use your Newton program to solve Eq. (4) with $\lambda = 0.19$ and $\beta = 0.12$. Choose a large enough value for m and a small enough tolerance τ to find the solution u at $\boldsymbol{x} = (\frac{1}{2}, \frac{1}{2})$, correct to three digits after the decimal point. **Write down** the corresponding values in the following table, using the exponential format for τ and at least four decimal digits for $u\left(\frac{1}{2}, \frac{1}{2}\right)$:

m	τ	$u\left(\frac{1}{2}, \frac{1}{2}\right)$
256	$1.0E-1$	5.2624

Produce a table of values $\|\boldsymbol{r}_k\|$ and $\|\boldsymbol{r}_{k+1}\|/\|\boldsymbol{r}_k\|^2$ with $m = 32$ and $\tau = 10^{-7}$ for all the Newton iterates which you computed and record it here (use the exponential number format):

k	$\|\boldsymbol{r}_k\|$	$\|\boldsymbol{r}_{k+1}\|/\|\boldsymbol{r}_k\|^2$
0	$0.16049236E+04$	$0.14720370E-04$
1	$0.37916432E+02$	$0.25545629E-04$
2	$0.36725823E-01$	$0.24600366E-04$
3	$0.33180631E-07$	

[3 points]

Q3: theoretical derivation of the Quasi-Newton method

(a) Show that $B_{k+1}(\mathbf{U}_{k+1} - \mathbf{U}_k) = \mathbf{F}(\mathbf{U}_{k+1}) - \mathbf{F}(\mathbf{U}_k)$, where B_k is recursively defined in Alg. 2. Explain why this may make B_{k+1} a good approximation to $\mathbf{F}'(\mathbf{U}_k)$.

[3 points]

(b) Prove that if B is any non-singular $n \times n$ matrix and $\mathbf{p}, \mathbf{q} \in \mathbb{R}^n$ are such that $\mathbf{q}^T B^{-1}\mathbf{p} \neq -1$ then $B + \mathbf{p}\mathbf{q}^T$ is non-singular and

$$(B + \mathbf{p}\mathbf{q}^T)^{-1} = \left(I - \frac{(B^{-1}\mathbf{p})\mathbf{q}^T}{1 + \mathbf{q}^T B^{-1}\mathbf{p}}\right) B^{-1}.$$

[2 points]

(c) Deduce that, under suitable conditions (which you should state), given $B_k, \mathbf{p}_k$ and $\mathbf{q}_k$ as defined in Alg. 2, it holds that

$$B_k^{-1} = P_k B_0^{-1} = \prod_{j=0}^{k-1}(I - \mathbf{w}_j \mathbf{q}_j^T) B_0^{-1}, \quad \text{where} \quad \mathbf{w}_j := \frac{B_j^{-1}\mathbf{p}_j}{(1 + \mathbf{q}_j^T B_j^{-1}\mathbf{p}_j)}$$

and where the notation

$$P_k = \prod_{j=0}^{k-1}(I - \mathbf{w}_j \mathbf{q}_j^T)$$

means the product $P_k = (I - \mathbf{w}_{k-1}\mathbf{q}_{k-1}^T)(I - \mathbf{w}_{k-2}\mathbf{q}_{k-2}^T)\ldots(I - \mathbf{w}_0\mathbf{q}_0^T)$ when $k \geq 1$ and $P_0 = I$, the identity matrix.

[3 points]

(d) Given $\mathbf{s}_k$ defined in Alg. 2, show that $\mathbf{w}_k = -\mathbf{s}_{k+1}/\|\mathbf{s}_k\|$, for each $k = 0, 1, \ldots$, and hence obtain the formulae:

$$B_k^{-1} = \prod_{j=0}^{k-1} \left(I + \frac{\mathbf{s}_{j+1}\mathbf{s}_j^T}{\|\mathbf{s}_j\|^2} \right) B_0^{-1}$$

and

$$\mathbf{s}_{k+1} = -\frac{B_k^{-1}\boldsymbol{r}_{k+1}}{1 + \mathbf{s}_k^T B_k^{-1}\boldsymbol{r}_{k+1}/\|\mathbf{s}_k\|^2} \,. \tag{6}$$

You may assume that the denominator in Eq. (6) does not vanish.

[3 points]

Insert your proofs for Q3 here

$\mathbf{U}_{k+1} - \mathbf{U}_k = \mathbf{s}_k$. Hence, multiplying by B_{k+1} and substituting in the Jacobian update gives

$$B_{k+1}(\mathbf{U}_{k+1} - \mathbf{U}_k) = B_{k+1}\mathbf{s}_k = (B_k + \mathbf{p}_k\mathbf{q}_k^\top)\mathbf{s}_k = B_k\mathbf{s}_k + \mathbf{p}_k\mathbf{q}_k^\top\mathbf{s}_k$$

Since $\mathbf{p}_k = \frac{\mathbf{r}_{k+1}}{||\mathbf{s}_k||}$, $\mathbf{q}_k = \frac{\mathbf{s}_k}{||\mathbf{s}_k||}$, $B_k\mathbf{s}_k = -\mathbf{r}_k$, $\mathbf{r}_k = \mathbf{F}(\mathbf{U}_k)$ and $\mathbf{s}_k^\top\mathbf{s}_k = ||\mathbf{s}_k||^2$,

$$B_k\mathbf{s}_k + \mathbf{p}_k\mathbf{q}_k^\top\mathbf{s}_k = -\mathbf{r}_k + \frac{\mathbf{r}_{k+1}\mathbf{s}_k^\top\mathbf{s}_k}{||\mathbf{s}_k||^2} = \mathbf{F}(\mathbf{U}_{k+1}) - \mathbf{F}(\mathbf{U}_k)$$

B_{k+1} may be a good approximation of $\mathbf{F}'(\mathbf{U}_k)$ since by the mean value theorem for vector valued functions,

$$|B_{k+1}| = \frac{|\mathbf{F}(\mathbf{U}_{k+1}) - \mathbf{F}(\mathbf{U}_k)|}{|\mathbf{U}_{k+1} - \mathbf{U}_k|} \approx |\mathbf{F}'(\mathbf{U}_k)|$$

As $\mathbf{U}_{k-1} - \mathbf{U}_k \to 0$, B_{k+1} provides a better approximation to the gradient of the tangent line at $\mathbf{F}'(\mathbf{U}_k)$, making it a good approximation.

Since B is non-singular, if $\mathbf{p}$ or $\mathbf{q}$ equal $\mathbf{0}$, then $(B + \mathbf{p}\mathbf{q}^\top)$ is certainly non-singular. Proceed to prove for $\mathbf{p}, \mathbf{q} \neq \mathbf{0}$. Suppose $(B + \mathbf{p}\mathbf{q}^\top)$ singular and $\mathbf{q}^\top B^{-1}\mathbf{p} \neq -1$. $(B + \mathbf{p}\mathbf{q}^\top)$ singular $\iff \exists \mathbf{x} \neq \mathbf{0}$ s.t. $(B + \mathbf{p}\mathbf{q}^\top)\mathbf{x} = 0 \iff B\mathbf{x} = -\mathbf{p}\mathbf{q}^\top\mathbf{x} \iff \mathbf{x} = -B^{-1}\mathbf{p}\mathbf{q}^\top\mathbf{x} \iff \mathbf{q}^\top\mathbf{x} = -(\mathbf{q}^\top B^{-1}\mathbf{p})(\mathbf{q}^\top\mathbf{x}) \iff (1 + \mathbf{q}^\top B^{-1}\mathbf{p})(\mathbf{q}^\top\mathbf{x}) = 0$ Notice that $\mathbf{q}^\top\mathbf{x} \neq 0 \Rightarrow (1 + \mathbf{q}^\top B^{-1}\mathbf{p}) = 0$ which contradicts $\mathbf{q}^\top B^{-1}\mathbf{p} \neq -1$. Hence, $(B + \mathbf{p}\mathbf{q}^\top)$ must be non-singular.

$$\begin{aligned}
(B + \mathbf{p}\mathbf{q}^\top)^{-1}(B + \mathbf{p}\mathbf{q}^\top) &= \left(B^{-1} - \frac{(B^{-1}\mathbf{p})\mathbf{q}^\top B^{-1}}{1 + \mathbf{q}^\top B^{-1}\mathbf{p}} \right)(B + \mathbf{p}\mathbf{q}^\top) \\
&= I + B^{-1}\mathbf{p}\mathbf{q}^\top - \left(\frac{B^{-1}\mathbf{p}\mathbf{q}^\top + B^{-1}\mathbf{p}(\mathbf{q}^\top B^{-1}\mathbf{p})\mathbf{q}^\top}{1 + \mathbf{q}^\top B^{-1}\mathbf{p}} \right) \\
&= I + B^{-1}\mathbf{p}\mathbf{q}^\top - B^{-1}\mathbf{p}\mathbf{q}^\top \left(\frac{1 + \mathbf{q}^\top B^{-1}\mathbf{p}}{1 + \mathbf{q}^\top B^{-1}\mathbf{p}} \right) = I
\end{aligned}$$

Which proves part (b) since $(B + \mathbf{p}\mathbf{q}^\top)$ is a square matrix.

Insert your proofs for Q3 here

Suitable conditions: B_0 is non-singular and $\mathbf{q}_0^\top B_0^{-1}\mathbf{p}_0 \neq -1$.
Statement: $B_k^{-1} = P_k B_0^{-1}\ \forall k \geq 0,\ k \in \mathbb{N}_0$.
Proceed by induction. When $k = 0$, $P_0 = I$ so $B_0^{-1} = P_0 B_0^{-1} = B_0^{-1}$. For $k = 1$,

$$B_1^{-1} = (B_0 + \mathbf{p}_0\mathbf{q}_0^\top)^{-1} = \left(I - \frac{B_0^{-1}\mathbf{p}_0\mathbf{q}_0^\top}{1 + \mathbf{q}_0^\top B_0^{-1}\mathbf{p}_0}\right) B_0^{-1} = (I - \mathbf{w}_0\mathbf{q}_0^\top)B_0^{-1} = P_1 B_0^{-1}$$

Assume the **statement** holds for some $k \geq 1$, then show it holds for $k + 1$.

$$B_{k+1}^{-1} = (B_k + \mathbf{p}_k\mathbf{q}_k^\top)^{-1} = \left(I - \frac{B_k^{-1}\mathbf{p}_k\mathbf{q}_k^\top}{1 + \mathbf{q}_k^\top B_k^{-1}\mathbf{p}_k}\right) B_k^{-1} = (I - \mathbf{w}_k\mathbf{q}_k^\top)B_k^{-1}$$

Where the induction hypothesis is used,

$$\begin{aligned} B_k^{-1} = P_k B_0^{-1} \Rightarrow B_{k+1}^{-1} &= (I - \mathbf{w}_k\mathbf{q}_k^\top)(I - \mathbf{w}_{k-1}\mathbf{q}_{k-1}^\top)\ldots(I - \mathbf{w}_0\mathbf{q}_0^\top)B_0^{-1} \\ &= \prod_{j=0}^{k}(I - \mathbf{w}_j\mathbf{q}_j^\top)B_0^{-1} = P_{k+1}B_0^{-1} \end{aligned}$$

Hence the statement of the question is proven by mathematical induction.

Proceed to show that $\mathbf{w}_k = -\frac{\mathbf{s}_{k+1}}{||\mathbf{s}_k||}$. Since $B_{k+1}\mathbf{s}_{k+1} = -\mathbf{r}_{k+1} \Rightarrow \mathbf{s}_{k+1} = -B_{k+1}^{-1}\mathbf{r}_{k+1}$. Substituting $\mathbf{s}_{k+1} = -B_{k+1}^{-1}\mathbf{r}_{k+1}$ and the Jacobian update into the expression for $\mathbf{w}_k$, then using part (b) gives,

$$\begin{aligned} \mathbf{w}_k &= \frac{B_{k+1}^{-1}\mathbf{r}_{k+1}}{||\mathbf{s}_k||} = (B_k + \mathbf{p}_k\mathbf{q}_k^\top)^{-1}\mathbf{p}_k = \left(I - \frac{B_k^{-1}\mathbf{p}_k\mathbf{q}_k^\top}{1 + \mathbf{q}_k^\top B_k^{-1}\mathbf{p}_k}\right) B_k^{-1}\mathbf{p}_k \\ &= \left(B_k^{-1}\mathbf{p}_k - \frac{B_k^{-1}\mathbf{p}_k\mathbf{q}_k^\top B_k^{-1}\mathbf{p}_k}{1 + \mathbf{q}_k^\top B_k^{-1}\mathbf{p}_k}\right) = \left(\frac{B_k^{-1}\mathbf{p}_k(1 + \mathbf{q}^\top B_k^{-1}\mathbf{p}_k) - B_k^{-1}\mathbf{p}_k\mathbf{q}_k^\top B_k^{-1}\mathbf{p}_k}{1 + \mathbf{q}_k^\top B_k^{-1}\mathbf{p}_k}\right) \\ &= \left(\frac{B_k^{-1}\mathbf{p}_k}{1 + \mathbf{q}_k^\top B_k^{-1}\mathbf{p}_k}\right) := \mathbf{w}_k \quad \text{as in part (c)} \end{aligned}$$

All arguments can be reversed. As in algorithm 2, $\mathbf{q}_j^\top = \frac{\mathbf{s}_j^\top}{||\mathbf{s}_j||}$. then substitute this and $\mathbf{w}_j$ into P_k as defined in part (c)

$$P_k = \prod_{j=0}^{k-1}(I - \mathbf{w}_j\mathbf{q}_j^\top) = \prod_{j=0}^{k-1}\left(I + \frac{\mathbf{s}_{j+1}\mathbf{s}_j^\top}{||\mathbf{s}_j||^2}\right) \Rightarrow B_k^{-1} = P_k B_0^{-1} = \prod_{j=0}^{k-1}\left(I + \frac{\mathbf{s}_{j+1}\mathbf{s}_j^\top}{||\mathbf{s}_j||^2}\right) B_0^{-1}$$

Then proceed to prove formula for $\mathbf{s}_{k+1}$:

$$\mathbf{w}_k = \left(\frac{B_k^{-1}\mathbf{p}_k}{1 + \mathbf{q}_k^\top B_k^{-1}\mathbf{p}_k}\right) = \left(\frac{B_k^{-1}\mathbf{r}_{k+1}/||\mathbf{s}_k||}{1 + \mathbf{s}_k^\top B_k^{-1}\mathbf{r}_{k+1}/||\mathbf{s}_k||^2}\right) = \frac{1}{||\mathbf{s}_k||}\left(\frac{B_k^{-1}\mathbf{r}_{k+1}}{1 + \mathbf{s}_k^\top B_k^{-1}\mathbf{r}_{k+1}/||\mathbf{s}_k||^2}\right)$$

$$-\frac{\mathbf{s}_{k+1}}{||\mathbf{s}_k||} \Rightarrow s_{k+1} = -\frac{B_k^{-1}\mathbf{r}_{k+1}}{1 + \mathbf{s}_k^\top B_k^{-1}\mathbf{r}_{k+1}/||\mathbf{s}_k||^2}$$

as required.

Q4: implementation of the Quasi-Newton method

Q4(a) Using the formulae from Q3(d), **derive** an efficient practical pseudocode of Alg. 2, where the matrices B_k are never stored explicitly, and the system $B_k \mathbf{s}_k = -\boldsymbol{r}_k$ is not solved explicitly for $k \geq 1$. Instead, in each iteration you should be able to produce the next Quasi-Newton step $\mathbf{s}_{k+1}$ using only vector operations and the matrix B_0 (or its LU decomposition).

[4 points]

Algorithm 1 Efficient Practical Quasi-Newton Method

Choose an initial guess $\mathbf{U}_0 \in \mathbb{R}^n$, an initial approximate Jacobian B_0 and a tolerance $\tau > 0$.
Set $\mathbf{r}_0 = \mathbf{F}(\mathbf{U}_0)$
if $(||\mathbf{r}_0|| \leq \tau)$ **exit**
Compute LU factors of B_0 and store in B_0
Solve $B_0 \mathbf{s}_0 = -\mathbf{r}_0$ for the Quasi-Newton step $\mathbf{s}_0$, using back-substitution.
for $k = 0, \ldots, K_{max} - 1$ **do**
    Solution update: $\mathbf{U}_{k+1} = \mathbf{U}_k + \mathbf{s}_k$
    Residual update: $\mathbf{r}_{k+1} = \mathbf{F}(\mathbf{U}_{k+1})$
    if $(||\mathbf{r}_{k+1}|| \leq \tau)$ **exit**
    if $(k = K_{max} - 1)$ **exit**
    Solve $B_0 \hat{\mathbf{s}} = -\mathbf{r}_{k+1}$ for $\hat{\mathbf{s}}$ using back-substitution, with $\hat{\mathbf{s}}$ overwriting $\mathbf{r}_{k+1}$
    if $k > 0$ **then**
        for $j = 0, \ldots, k - 1$ **do**
            $\hat{\mathbf{s}} = \hat{\mathbf{s}} + \mathbf{s}_{j+1} \mathbf{s}_j^\top \hat{\mathbf{s}} / ||\mathbf{s}_j||_2^2$
        end for
    end if
    Quasi-Newton step update: $\mathbf{s}_{k+1} = \hat{\mathbf{s}} / (1 - \mathbf{s}_k^\top \hat{\mathbf{s}} / ||\mathbf{s}_k||_2^2)$
end for

Q4(b) **Implement** the practical version of Alg. 2 in the main program file `quasi_newton.f90` and use it to solve Eq. (4), again with $\lambda = 0.19$ and $\beta = 0.12$. At the kth iteration, your method should only require storage of the approximate solution $\mathbf{U}_{k+1}$ and of the residual $\boldsymbol{r}_{k+1}$, together with the LU factors of B_0 (in an appropriate format) and all the previously computed Quasi-Newton steps $\mathbf{s}_0, \ldots, \mathbf{s}_k$. Limit the user to no more than $K_{\max} = 100$ steps. You should choose $\mathbf{F}'(\mathbf{U}_0)$ as the initial matrix B_0. Make use of `BLAS` and `LAPACK` as much as possible to ensure efficiency. Test your program thoroughly, possibly using the test problems from Q1. **Extend the** `Makefile` such that it can compile also the Quasi-Newton program into an executable `quasi_newton`.

[11 points]

Q4(c) For $m = 32$ and $\tau = 10^{-7}$ **produce a table** of values for $\|\boldsymbol{r}_k\|$, $\|\boldsymbol{r}_{k+1}\|/\|\boldsymbol{r}_k\|^2$ and $\|\boldsymbol{r}_{k+1}\|/\|\boldsymbol{r}_k\|$ for all the Quasi-Newton iterates which you computed (use exponential number format). Compare the results to those in Q2 and explain your observations.

[3 points]

k	$\|\boldsymbol{r}_k\|$	$\|\boldsymbol{r}_{k+1}\|/\|\boldsymbol{r}_k\|^2$	$\|\boldsymbol{r}_{k+1}\|/\|\boldsymbol{r}_k\|$
0	$0.16049236E+04$	$0.14720370E-04$	$0.23625070E-01$
1	$0.37916432E+02$	$0.89132661E-03$	$0.33795925E-01$
2	$0.12814209E+01$	$0.14010992E-02$	$0.17953978E-02$
3	$0.23006603E-02$	$0.21245017E+01$	$0.48877569E-02$
4	$0.11245068E-04$	$0.54199340E+03$	$0.60947529E-02$
5	$0.68535913E-07$		

For the Newton iterates, the column $\frac{||r_{k+1}||}{||r_k||^2}$ suggests that the sequence of computed $(\mathbf{U}_k)_{k\in\mathbb{N}_0}$ converges to $\mathbf{U}_*$ quadratically as $k \to \infty$. This is because

$$\lim_{k\to\infty} \frac{||r_{k+1}||}{||r_k||^\alpha} \in (0,1) \quad \text{where } \alpha = 2$$

However, for the Quasi Newton iterates, the column $\frac{||r_{k+1}||}{||r_k||}$ suggests that the sequence of computed $(\mathbf{U}_k)_{k\in\mathbb{N}_0}$ converges to $\mathbf{U}_*$ super-linearly as $k \to \infty$. This is because

$$\lim_{k\to\infty} \frac{||r_{k+1}||}{||r_k||} = 0$$

Notice that super linear convergence implies that only a few iterates are needed to reach an accurate solution for $\mathbf{U}_*$

Q4(d) **Write** a short **report** on how you made the program efficient. Compare **CPU times** of your Newton and Quasi-Newton codes for two different values of m and explain the results. To get reliable timing, you will have to make sure that the runtime is no less than about 0.1 seconds since the timer resolution is 0.01 seconds. Make sure that you compile your code with optimisation.

[3 points]

Insert your report for Q4(d) here

Since B_0 is SPD (Banded Cholesky decomposition works and A is symmetric) and banded (A is banded and $\mathbf{F}'(\mathbf{U}_0) = A - \text{diag}(\mathbf{G}'(\mathbf{U}_0))$), **quasi_newton** was made efficient by exploiting the structure of B_0 when solving $B_0\mathbf{s}_0 = -\mathbf{r}_0$ and $B_0\hat{\mathbf{s}} = -\mathbf{r}_{k+1}$. The only $(B_k)_{k\in\mathbb{N}_0}$ needed in **quasi_newton** was B_0. For efficiency, the Banded Cholesky decomposition was computed for B_0 only in the first step, and stored in $m \times (m-1)^2$ banded format, overwriting B_0. This was done with a single call to LAPACK subroutine DPBTRF. Notice that using a $(m-1)^2 \times (m-1)^2$ storage format to store computed LU factors, and using a dense LAPACK driver (DGETRS) to solve $B_0\mathbf{s}_0 = -\mathbf{r}_0$ and $B_0\hat{\mathbf{s}} = -\mathbf{r}_{k+1}$, would be significantly less efficient. This is illustrated in the table below. The average time to complete a run of the iterative loop when using DGETRS and DPBTRS (banded driver) was computed as m grows.

m	DGETRS (s)	DPBTRS (s)
8	1.945E-5	2.25E-5
16	5.80E-5	4.78E-5
32	8.24E-4	1.32E-4
64	2.00E-2	5.60E-4
128	killed	4.14E-3

When computing line 12 on pseudocode (Q4a), the BLAS subroutine DAXPY was used (in conjunction with DDOT and DNRM2), to replace Fortran intrinsic functions (dot_product and norm2), for efficiency. Memory efficiency was achieved, with storage requirements of $Kmax+2$ vectors ($\hat{\mathbf{s}}$ and $\mathbf{r}$ share the same memory location) and two $m \times (m-1)^2$ matrices (neither of which being overwritten).

Comparison of CPU times for completion of Newton and Quasi-Newton methods (tolerance $= 1.0E-7$)

m	Newton (s)	Quasi-Newton (s)
200	0.42	0.23
400	4.15	1.92

It was expected that the Efficient Practical Quasi-Newton method would perform significantly better than Newtons method for large scale problems. This is because, in Newtons method, $\mathbf{F}(\mathbf{U}_k)\mathbf{s}_k = -\mathbf{r}_k$ is solved for each k. This is very computationally expensive; it is required to deform $\mathbf{F}(\mathbf{U}_k)$ using the Banded Cholesky decomposition (DPBTRF) and then apply back substitution (DPBTRS). Calling DPBSV does both these tasks at most $kmax-1$ many times. However, the Efficient Practical Quasi-Newton method only requires one call to DPBTRF and then at most $kmax$ calls to DPBTRS. Also, since super-linear convergence implies few iterates are needed to reach an accurate solution $\mathbf{U}$, the most computationally costly routine in the iterative loop (DPBTRS) will not be called too often. Calculating the next $\mathbf{s}_{k+1}$ after routine DPBTRS has a complexity of $\mathcal{O}(nk)$, which certainly should help efficiency. Writing and rewriting large quantities of data is costly. In Newtons method, it is required to write data to an $m \times (m-1)^2$ matrix at most $kmax+1$ times (initialising A and writing/rewriting $\mathbf{F}'(\mathbf{U}_k)$). The Efficient Practical Quasi-Newton method minimises this; it is only required to write data to an $m \times (m-1)^2$ matrix at most twice, in conjunction with only writing/rewriting vector updates of size $(m-1)^2$. Note that the Efficient Practical Quasi-Newton method is aided here since $\mathbf{F}'(\mathbf{U}_0)$ happens to be a good initial approximation to $\mathbf{F}'(\mathbf{U}_*)$; if this was not the case, then expect to see less of an advantage for using the Efficient Practical Quasi-Newton method.

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 α, $\beta \in \mathbb{R}$.
- 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`)