MA40177: Scientific Computing
Assignment 1

Handed out: Friday, 11 Mar
Due in: Friday, 25 Mar (*by 17.00 at Assignment 1 Submission Point on Moodle*)

This assignment is worth 50% of the total assessment for MA40177 and should take an average student about 20 hours to complete (provided they have done all the problem sheet questions and tutorial exercises). 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.

---

1 Turing instability in reaction-diffusion systems (a.k.a. stripes & spots)

This assignment is about practical aspects of solving nonlinear differential equations

$$\begin{aligned} \frac{d\mathbf{y}(t)}{dt} &= \mathbf{A}\mathbf{y}(t) + \mathbf{N}(\mathbf{y}(t)), \qquad \text{on } t \in [0,T] \\ \mathbf{y}(0) &= \mathbf{y}_0 \end{aligned} \tag{1}$$

using operator splitting time schemes, where $\mathbf{A} \in \mathbb{R}^{2n\times 2n}$ is a given stiff square matrix, $\mathbf{N}(\mathbf{y}) : \mathbb{R}^{2n} \to \mathbb{R}^{2n}$ is a non-stiff but nonlinear function, $\mathbf{y}_0 \in \mathbb{R}^{2n}$ is a given initial state at time $t = 0$, and $n \in \mathbb{N}$ is a discretisation size in space to be defined in Section 1.1 below.

Equations of the form (1) arise for example in the numerical solution of reaction-diffusion equations, that is, systems of Partial Differential Equations (PDEs) that combine linear diffusion of substances with nonlinear chemical reactions between them. Such systems may demonstrate a surprising variety of complex patterns. Those can be observed both in nature (spots and stripes on butterflies, fish or zebras) and in lab (diffusive auto-catalysis in open gel reactors). The precise pattern may change significantly even after a small variation of coefficients in (1). This kind of instability was called the Turing instability, after Alan Turing's pioneering study of morphogenesis [1].

A particular model we consider here was formulated originally by Gray and Scott, and analysed extensively by Pearson [2]. It consists of two reactions

$$\begin{aligned} U + 2V &\to 3V \\ V &\to \emptyset \end{aligned}$$

between two substances U and V. The substance U is being fed to the reactor at a constant rate $F > 0$ and morphed into the substance V by the first reaction, catalysed by V itself (hence the name auto-catalysis). In contrast, the substance V is being removed from the reactor at a constant "kill" rate $k > 0$. Moreover, both U and V can diffuse through the reactor at rates $D_u > 0$ and $D_v > 0$, respectively. In the first assignment, we start with a simple one-dimensional circular reactor, such that $u(x,t) \geq 0$, the concentration of U, and $v(x,t) \geq 0$, the concentration of V, become periodic functions of the spatial position $x \in [0,1]$ in the reactor. Putting these assumptions together, we arrive at the Gray-Scott system of PDEs defining the concentrations:

$$\begin{aligned}
\frac{\partial u(x,t)}{\partial t} &= -uv^2 + F(1-u) + D_u \frac{\partial^2 u}{\partial x^2} && \text{on } x \in [0,1], \quad t \in [0,T] && (2)\\
\frac{\partial v(x,t)}{\partial t} &= uv^2 - (F+k)v + D_v \frac{\partial^2 v}{\partial x^2} && && (3)\\
u(1,t) = u(0,t), &\qquad v(1,t) = v(0,t) && \text{(Boundary conditions)} && (4)\\
u(x,0) = u_0(x), &\qquad v(x,0) = v_0(x). && \text{(Initial conditions)} && (5)
\end{aligned}$$

In our assignment, the diffusion coefficients are fixed to $D_u = 2 \cdot 10^{-5}$ and $D_v = 10^{-5}$.

1.1 Discretisation in space

To arrive from (2)–(5) to (1) we need to discretise the spatial variable x, and, correspondingly, the functions u and v. We divide the interval $[0,1]$ into $n \in \mathbb{N}$ subintervals of length $h = 1/n$. Then we can define discrete fields

$$u_i = u(ih,t), \qquad v_i = v(ih,t), \tag{6}$$

where $i = 1,\dots,n$. The derivatives of u, v at the midpoints $x = (i-1/2)h$ are approximated by the central finite difference as

$$\left.\frac{\partial u}{\partial x}\right|_{x=(i-1/2)h} \to \frac{u_{i+1}-u_i}{h}, \qquad \left.\frac{\partial v}{\partial x}\right|_{x=(i-1/2)h} \to \frac{v_{i+1}-v_i}{h},$$

with the periodic boundary conditions (4) implying that

$$u_{n+1} = u_1, \quad u_0 = u_n, \qquad v_{n+1} = v_1, \quad v_0 = v_n. \tag{7}$$

Applying the central difference scheme again to get the second derivatives, we obtain an approximation at the whole points,

$$\left.\frac{\partial^2 u}{\partial x^2}\right|_{x=ih} \to \frac{u_{i-1}-2u_i+u_{i+1}}{h^2}, \qquad \left.\frac{\partial^2 v}{\partial x^2}\right|_{x=ih} \to \frac{v_{i-1}-2v_i+v_{i+1}}{h^2} \tag{8}$$

subject to discrete boundary conditions (7).

Collecting discrete values u_i, v_i into vectors

$$\mathbf{u}(t) = \begin{bmatrix} u_1 & u_2 & \cdots & u_n \end{bmatrix}^\top, \qquad \mathbf{v}(t) = \begin{bmatrix} v_1 & v_2 & \cdots & v_n \end{bmatrix}^\top,$$

we can write the second central differences (8) as matrix-vector products $\mathbf{\Delta u}$, $\mathbf{\Delta v}$ with the matrix

$$\mathbf{\Delta} = \frac{1}{h^2}\begin{bmatrix} -2 & 1 & \cdots & 0 & 1 \\ 1 & -2 & 1 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 1 & -2 & 1 \\ 1 & 0 & \cdots & 1 & -2 \end{bmatrix} \in \mathbb{R}^{n\times n}.$$

In turn, nonlinear reaction terms in (2),(3) can be written via vector-functions

$$\mathbf{N}_u(\mathbf{u},\mathbf{v}) = -\mathbf{u}\odot\mathbf{v}\odot\mathbf{v} + \begin{bmatrix} F & \cdots & F \end{bmatrix}^\top - F\mathbf{u}, \qquad \mathbf{N}_v(\mathbf{u},\mathbf{v}) = \mathbf{u}\odot\mathbf{v}\odot\mathbf{v} - (F+k)\mathbf{v},$$

where "$\odot$" denotes multiplication of vectors elementwise.

1.2 Discretisation in time and Strang splitting

Finally, introducing block vectors

$$\mathbf{y}(t) = \begin{bmatrix}\mathbf{u}(t)\\ \mathbf{v}(t)\end{bmatrix}, \quad \mathbf{y}_0 = \begin{bmatrix}\mathbf{u}(0)\\ \mathbf{v}(0)\end{bmatrix}, \qquad \mathbf{N}(\mathbf{y}) = \begin{bmatrix}\mathbf{N}_u(\mathbf{u},\mathbf{v})\\ \mathbf{N}_v(\mathbf{u},\mathbf{v})\end{bmatrix}, \tag{9}$$

and a block matrix

$$\mathbf{A} = \begin{bmatrix}D_u\mathbf{\Delta} & \\ & D_v\mathbf{\Delta}\end{bmatrix}, \tag{10}$$

we can write the spatially-discrete equations in the differential equation form (1). However, $\mathbf{Ay}$ and $\mathbf{N}(\mathbf{y})$ desire different time integration schemes:

- the first term (linear and stiff) can be tackled most efficiently with an implicit scheme (such as Crank-Nicolson) that is stable for a wide range of time steps, whereas
- the second term warrants an explicit scheme (such as Runge-Kutta) to avoid solving nonlinear equations, whereas the stability of bounded reaction coefficients is not an issue.

Those can be merged together by using so-called *splitting* schemes. The motivation stems from the solution of a linear equation $d\mathbf{y}/dt = (\mathbf{A}+\mathbf{B})\mathbf{y}$ (where $\mathbf{B}$ is another matrix) in the form of the matrix exponential $\mathbf{y} = \exp(t(\mathbf{A}+\mathbf{B}))\mathbf{y}_0$. In contrast to real numbers, for matrices $\exp(t\mathbf{A}+t\mathbf{B}) \neq \exp(t\mathbf{A})\exp(t\mathbf{B})$ in general, but for *small* t the product of exponentials approximates the exponential of sum with an error proportional to t. A more accurate approximation is provided by the *Strang* splitting [3] $\exp(t\mathbf{A}/2)\exp(t\mathbf{B})\exp(t\mathbf{A}/2)$. In turn, the product of each of those matrix exponentials by a vector can be approximated by a more traditional time scheme (Crank-Nicolson or Runge-Kutta). Lifting this idea to nonlinear terms gives us the Strang splitting for (1), outlined in Algorithm 1.

Algorithm 1 Strang splitting method

```
1: Input: initial vector $\mathbf{y}_0 = \mathbf{y}(0)$, time step size $\tau > 0$, final time $T > 0$.
2: for $\ell = 0, \dots, \lceil T/\tau \rceil - 1$ do
3:     Let $t_\ell = \ell\tau$.
4:     Solve $d\mathbf{z}/dt = \mathbf{Az}$ for $t \in [0, \tau/2]$ starting from $\mathbf{z}_0 = \mathbf{z}(0) = \mathbf{y}_\ell$.          ▷ cf.(11)
5:     Solve $d\mathbf{w}/dt = \mathbf{N}(\mathbf{w})$ for $t \in [0, \tau]$ starting from $\mathbf{w}_0 = \mathbf{w}(0) = \mathbf{z}(\tau/2)$.   ▷ cf.(12)
6:     Solve $d\mathbf{q}/dt = \mathbf{Aq}$ for $t \in [0, \tau/2]$ starting from $\mathbf{q}_0 = \mathbf{q}(0) = \mathbf{w}(\tau)$.          ▷ cf.(11)
7:     Let $\mathbf{y}_{\ell+1} = \mathbf{q}(\tau/2) \approx \mathbf{y}(t_\ell + \tau)$.
8: end for
```

To approximate the solution in Lines 4 and 6 we use the Crank-Nicolson scheme, which can be implemented by solving linear systems

$$\left(\mathbf{I} - \frac{\tau}{4}\mathbf{A}\right)\mathbf{z}_1 = \left(\mathbf{I} + \frac{\tau}{4}\mathbf{A}\right)\mathbf{z}_0, \qquad \left(\mathbf{I} - \frac{\tau}{4}\mathbf{A}\right)\mathbf{q}_1 = \left(\mathbf{I} + \frac{\tau}{4}\mathbf{A}\right)\mathbf{q}_0 \tag{11}$$

with respect to the vectors $\mathbf{z}_1 \approx \mathbf{z}(\tau/2)$ and $\mathbf{q}_1 \approx \mathbf{q}(\tau/2)$, where $\mathbf{I}$ is the identity matrix. For Line 5 we use the order-2 Runge-Kutta method, also known as the Heun method:

$$\mathbf{w}_{1/2} = \mathbf{w}_0 + \tau\mathbf{N}(\mathbf{w}_0), \qquad \mathbf{w}_1 = \mathbf{w}_0 + \tau\frac{\mathbf{N}(\mathbf{w}_0) + \mathbf{N}(\mathbf{w}_{1/2})}{2}, \tag{12}$$

where $\mathbf{w}_1 \approx \mathbf{w}(\tau)$. Both (11) and (12) have second order of consistency, that is,

$$\|\mathbf{z}(\tau/2) - \mathbf{z}_1\|_2 = \mathcal{O}(\tau^3), \quad \|\mathbf{q}(\tau/2) - \mathbf{q}_1\|_2 = \mathcal{O}(\tau^3), \quad \|\mathbf{w}(\tau) - \mathbf{w}_1\|_2 = \mathcal{O}(\tau^3).$$

Here $\|\mathbf{y}\|_2$ is the usual 2-norm defined for any vector $\mathbf{y} = [y_1, \dots, y_{2n}]^\top$ as $\|\mathbf{y}\|_2 = \sqrt{\sum_{i=1}^{2n} y_i^2}$. It can be shown (see Q6 in Part III of the assignment) that the entire Strang splitting approximation $\mathbf{y}_\ell \approx \mathbf{y}(\ell\tau)$ has the second order of consistency as well.

1.3 Improvements

Since $\mathbf{A}$ is a block matrix, we can keep the original vectors $\mathbf{u}$ and $\mathbf{v}$ throughout the computations in Algorithm 1. We can also store only the current time step. Only a few extra vectors are then actually needed in intermediate steps.

The Crank-Nicolson scheme (11) depends on four matrices, which we can denote

$$\mathbf{M}_{lu} = \mathbf{I} - D_u \frac{\tau}{4}\boldsymbol{\Delta}, \quad \mathbf{M}_{lv} = \mathbf{I} - D_v \frac{\tau}{4}\boldsymbol{\Delta}, \qquad \mathbf{M}_{ru} = \mathbf{I} + D_u \frac{\tau}{4}\boldsymbol{\Delta}, \quad \mathbf{M}_{rv} = \mathbf{I} + D_v \frac{\tau}{4}\boldsymbol{\Delta}. \tag{13}$$

For brevity, we can define the *Courant numbers*

$$\tau_u = \frac{D_u \tau}{4h^2} \quad \text{and} \quad \tau_v = \frac{D_v \tau}{4h^2}.$$

It can be shown (see Q7 in Part III) that $\mathbf{M}_{l*}$ and $\mathbf{M}_{r*}$ (where $*$ stands for u or v) can be written as a sum of a banded matrix and a rank-1 matrix,

$$\mathbf{M}_{l*} = \mathbf{B}_{l*} - \tau_* \mathbf{c}\mathbf{c}^\top, \quad \mathbf{M}_{r*} = \mathbf{B}_{r*} + \tau_* \mathbf{c}\mathbf{c}^\top, \quad \text{where} \quad \mathbf{c} = \begin{bmatrix} 1 & 0 & \cdots & 0 & 1 \end{bmatrix}^\top, \tag{14}$$

and

$$\mathbf{B}_{l*} = \begin{bmatrix} 1+3\tau_* & -\tau_* & & & \\ -\tau_* & 1+2\tau_* & -\tau_* & & \\ & \ddots & \ddots & \ddots & \\ & & -\tau_* & 1+2\tau_* & -\tau_* \\ & & & -\tau_* & 1+3\tau_* \end{bmatrix}, \qquad \mathbf{B}_{r*} = \begin{bmatrix} 1-3\tau_* & \tau_* & & & \\ \tau_* & 1-2\tau_* & \tau_* & & \\ & \ddots & \ddots & \ddots & \\ & & \tau_* & 1-2\tau_* & \tau_* \\ & & & \tau_* & 1-3\tau_* \end{bmatrix}$$

are matrices with bandwidth 1. Both multiplication by a vector and solution of a linear system can be computed much faster if the matrix is stored in an appropriate banded form. To solve linear systems with $\mathbf{M}_{l*}$ we can show that these matrices can be written as

$$\mathbf{M}_{l*}^{-1} = (\mathbf{I} + \mathbf{h}_* \mathbf{c}^\top)\mathbf{B}_{l*}^{-1}, \tag{15}$$

where

$$\mathbf{h}_* = \tau_*(1-\rho_*)^{-1} \cdot \hat{\mathbf{h}}_*, \qquad \mathbf{B}_{l*}\hat{\mathbf{h}}_* = \mathbf{c}, \qquad \rho_* = \tau_* \mathbf{c}^\top \hat{\mathbf{h}}_*. \tag{16}$$

Hence, to solve a system $\mathbf{M}_{lu}\mathbf{u} = \mathbf{r}_u$ (and similarly for $\mathbf{v}$), we can proceed as follows: in a setup phase (outside the timestepping loop),

1. calculate $\hat{\mathbf{h}}_u$, $\hat{\mathbf{h}}_v$ by solving the banded systems $\mathbf{B}_{lu}\hat{\mathbf{h}}_u = \mathbf{c}$, $\mathbf{B}_{lv}\hat{\mathbf{h}}_v = \mathbf{c}$,

2. calculate ρ_u, ρ_v and $\mathbf{h}_u, \mathbf{h}_v$ as shown in (16).

Then, given the precomputed $\mathbf{B}_{l*}$ and $\mathbf{h}_*$, the equations $\mathbf{M}_{lu}\mathbf{u} = \mathbf{r}_u$, $\mathbf{M}_{lv}\mathbf{v} = \mathbf{r}_v$ can be solved at every time step as follows:

1. calculate $\hat{\mathbf{u}}$ by solving $\mathbf{B}_{lu}\hat{\mathbf{u}} = \mathbf{r}_u$ and $\hat{\mathbf{v}}$ by solving $\mathbf{B}_{lv}\hat{\mathbf{v}} = \mathbf{r}_v$,

2. calculate $\mathbf{u} = \hat{\mathbf{u}} + \mathbf{h}_u(\mathbf{c}^\top \hat{\mathbf{u}})$ and $\mathbf{v} = \hat{\mathbf{v}} + \mathbf{h}_v(\mathbf{c}^\top \hat{\mathbf{v}})$.

Note that only solves with the banded, symmetric positive definite matrices $\mathbf{B}_{l*}$ are needed.

2 The Assignment

Part I: Basic Implementation

Copy over the files from the directory `$MA40177_DIR/assignment1`. They are designed to solve numerically the model problem defined above. In particular, the matrices $\mathbf{M}_{lu}, \mathbf{M}_{lv}, \mathbf{M}_{ru}, \mathbf{M}_{rv}$ defined in (13) are created in the subroutine in `create_matrices.f90`. However, the subroutines in `crank_nicolson.f90`, `heun.f90` and `timestepping.f90` needed for Algorithm 1 are currently empty. It will be your task to write them.

Q1: Crank-Nicolson step

Implement a subroutine `crank_nicolson(n,Mlu,Mlv,Mru,Mrv,u,v)` in the file `crank_nicolson.f90` which gets passed the grid size n, the matrices $\mathbf{M}_{lu}, \mathbf{M}_{lv}, \mathbf{M}_{ru}, \mathbf{M}_{rv}$, and the vectors $\mathbf{u}$ and $\mathbf{v}$, in this order. On input to the subroutine, the vectors $\mathbf{u}$ and $\mathbf{v}$ should contain the initial conditions of the corresponding components (that is, top and bottom parts of $\mathbf{z}_0$ or $\mathbf{q}_0$ in Algorithm 1). On completion of the subroutine, the vectors $\mathbf{u}$ and $\mathbf{v}$ should contain the parts of $\mathbf{z}_1$ or $\mathbf{q}_1$, approximating the exact solution at time $\tau/2$. That is, this subroutine must compute the matrix-vector products $\mathbf{r}_u = \mathbf{M}_{ru}\mathbf{u}$ and $\mathbf{r}_v = \mathbf{M}_{rv}\mathbf{v}$, and solve the linear systems $\mathbf{M}_{lu}\mathbf{u} = \mathbf{r}_u$, $\mathbf{M}_{lv}\mathbf{v} = \mathbf{r}_v$.

Write a subroutine `test_crank(n,Mlu,Mlv,Mru,Mrv)` in the file `test_crank.f90` which gets passed the problem size n and the matrices $\mathbf{M}_{lu}, \mathbf{M}_{lv}, \mathbf{M}_{ru}, \mathbf{M}_{rv}$, and tests that the solution computed by `crank_nicolson` is accurate. There are several possible tests you can design. The most straightforward one is to create vectors $\mathbf{u}_{test}$, $\mathbf{v}_{test}$ filled with random values (you can use the Fortran subroutine `random_number(u_test)`), and reverse the Crank-Nicolson scheme, computing $\mathbf{u}_{in} = \mathbf{M}_{ru}^{-1}(\mathbf{M}_{lu}\mathbf{u}_{test})$ and $\mathbf{v}_{in} = \mathbf{M}_{rv}^{-1}(\mathbf{M}_{lv}\mathbf{v}_{test})$. Then, we call `crank_nicolson` with $\mathbf{u}_{in}$ and $\mathbf{v}_{in}$ as inputs, returning some output vectors $\mathbf{u}, \mathbf{v}$. Finally, we calculate and print the errors $\|\mathbf{u} - \mathbf{u}_{test}\|_2$, $\|\mathbf{v} - \mathbf{v}_{test}\|_2$. There is a more easy (but less general) test that is based on the properties of the matrix $\mathbf{\Delta}$. You can have only one test implemented in the final submission, but you should comment about assumptions and outcomes of this test in your code.

Run the test(s) by **calling** the `test_crank` subroutine once **in the main program** `grayscott.f90` before a call to `timestepping`. Use suitable optimised BLAS/LAPACK library calls wherever this is possible. Only use as many temporary vectors and matrices as necessary and avoid redundant calculations.

[7 points]

Q2: Heun step

Implement a subroutine `heun(n,tau,F,k,u,v)` in the file `heun.f90` which gets passed the problem size n, time step $\tau > 0$, feed rate $F > 0$, kill rate $k > 0$ and the solution vectors $\mathbf{u}, \mathbf{v}$. On input to the subroutine, the vectors $\mathbf{u}$ and $\mathbf{v}$ should contain the top and bottom parts of the initial condition $\mathbf{w}_0$ in Algorithm 1. On completion of the subroutine, the vectors $\mathbf{u}$ and $\mathbf{v}$ should contain the parts of $\mathbf{w}_1$ as defined in (12), approximating the exact solution at time τ.

[4 points]

Q3: Strang time stepping method and main program

Implement the time stepping method from Algorithm 1. For this, write a subroutine `timestepping(n,Mlu,Mlv,Mru,Mrv,tau,T,F,k,u,v)` in the file `timestepping.f90` with the following arguments:

- `n`: an integer problem size n
- `Mlu`,`Mlv`,`Mru`,`Mrv`: real-valued matrices (13) of dimension $n \times n$
- `tau`: a real step size $\tau > 0$
- `T`: a real final time $T > 0$
- `F`: a real feed rate $F > 0$
- `k`: a real kill rate $k > 0$
- `u`,`v`: real-valued vectors of length n of the discretised concentrations.

The subroutine should be passed the initial conditions for the vectors $\mathbf{u}$ and $\mathbf{v}$ and return the solutions $\mathbf{u}_\ell \approx \mathbf{u}(T)$ and $\mathbf{v}_\ell \approx \mathbf{v}(T)$ after $\ell = T/\tau$ time steps. For simplicity, you can assume that $T = \ell\tau$ with an integer number of steps ℓ. Use as few variables as possible and make use of suitable library calls to implement the linear algebra operations. Use the subroutines `crank_nicolson` and `heun` developed previously. To help you with the implementation, the following files are already provided, but need to be adapted accordingly:

- main program in `grayscott.f90` (contains **no** call to `timestepping` at the moment)
- subroutine for calculating $\mathbf{M}_{lu}, \mathbf{M}_{lv}, \mathbf{M}_{ru}, \mathbf{M}_{rv}$ in `create_matrices.f90`
- subroutine for calculating initial conditions $\mathbf{u}_0, \mathbf{v}_0$ in `initial.f90`
- subroutine for saving the vector values to a text file in `save_fields.f90` (see Part IV)

Extend the main program so that it reads the variables n, τ, T, F, k from a text file with the following contents (an example input file is given in `input.dat`):

```
n
tau
T
F
k
```

Write a Makefile to compile the code. **Write a short report** (0.5-1 page) describing how you implemented and tested the algorithm, and how you made it efficient (explain *why* this particular choice improves efficiency, or *why* this particular test is helpful). Points will be awarded for correct, efficient and well structured code with suitable comments.

[11 points]

To start with I wrote crank_nicholson.f90 and test_crank.f90 subroutines. To do this I started off by copying the method from the statement of Q1 and used the most general forms of the solvers and matrix-vector multipliers. (dgesv and dgemv resp.). I wrote test_crank in unison with this and was able to test the results of the linear algebra of the LAPACK and BLAS functions. This allowed me to test each part of the algorithm sequentially to be able to correctly identify errors on the way throughout writing the code. This became particularly useful after noticing that most solvers edit the input matrices. After writing both crank_nicholson and test_crank (and the initial part of the grayscott programm to be able to run both crank_nicholson and test_crank), I started to optimise the actual subroutines themselves. M_{ru} and M_{rv} are both symmetric and therefore we can optimise dgemv to use dsymv for symmetric matrices. Similarly M_{lu} and M_{lv} are symmetric positive definite (SPD) and so we can optimise dgesv to dposv with is specifically for SPD matrices. Similarly in test_crank.f90 we can optimise the BlAS functions to dsymv and dsysv (as dpomv doesn't exit). After these edits, the outputs remained the same for a wide range of test input values and hence tested correctly. For Q2, the heun step, I closely followed the given instructions at (12) and created an appropriate algorithm. Originally I added new variables for each new variable of the algorithm. This was to be able to read the program clearer until it was working properly. I didn't optimise this until I had written the timestepping algorithm, to be able to test the subroutine properly. This was just writing algorithm 1 in code and was a two calls of crank_nicholson and one call of heun repeated $\lceil T/\tau \rceil$ times. This was then added to the main program grayscott and testing was able to begin. Having done a little research on the internet about these PDE's I had a rough idea of what I was looking for. There were a few errors throughout the debugging process that needed to be resolved. Most of these were faulty inputs into BLAS and LAPACK functions. This was able to be solved by using print functions around these functions are looking at the inputs and outputs. This was particularly key around parts of the code where the original matrices, M_{**}, could be edited. Looking at specific entries or columns of each matrix could give away if it were edited in one or more of the subroutines or functions and so I was able to look at these parts individually. I also ended up using visualise.py to be help this debugging process. As said, I had some idea of what I was looking for and was able to look at different plots of the end results to see whether what I was looking at was in the right direction or not and potentially where I should be looking for errors.

After all this I was able to remove all the bugs and get repeatable results. Using the steady states given in Q9 I was able to visualise the results of the algorithm and these confirmed my results. After this I was able to optimise the heun subroutine with BLAS and LAPACK subroutines. This included calls to the dcopy subroutine and calls to the daxpy subroutine which optimised the addition and scaling of variables throughout the subroutine. There was also the chance to remove some excess variables in the code to be able to optimise memory usage. This is why u_{copy} and v_{copy} are used multiple times throughout the code (and the same for N_u and N_v).

Part II: Improved Implementation

Improve the performance of the time stepping by using the banded matrix representations in Eq (14)–(16). For this, implement a new version of the code in a separate directory `PartII`. When you are finished, this directory should contain a fully working code which can be compiled with its own Makefile you wrote there.

Q4: Banded Crank-Nicolson solve

Implement a new subroutine `create_banded(n,tau,Blu,Blv,Bru,Brv,hu,hv)` in the file `create_banded.f90`. This subroutine should return (suitable representations of) the tridiagonal matrices $\mathbf{B}_{lu}, \mathbf{B}_{lv}, \mathbf{B}_{ru}, \mathbf{B}_{rv}$, as well as the vectors $\mathbf{h}_u, \mathbf{h}_v$ Based on this, **write** another subroutine `banded_crank(n,tau,Blu,Blv,Bru,Brv,hu,hv,u,v)` in the file `banded_crank.f90` which implements the Crank-Nicolson steps (11), but using the algorithm described below Eq. (16). **Implement** a new subroutine `test_banded(n,tau,Blu,Blv,Bru,Brv,hu,hv)` in the file `test_banded.f90` which tests the subroutine `banded_crank()`. You might compare to the inversion of the dense representations of $\mathbf{M}_{l*}$ or use a synthetic test vector as previously. Again, exploit the structure of the matrices/vectors to make the code efficient. **Write a short report** on how you implemented and tested your code.

[7 points]

Insert answer for Q6 here

To start this part, I implemented a new subroutine, create-banded. This was tested by printing the completed matrices for multiple sizes of the input n and checking if they were correct. The B_{**}'s are calculated using a do loop with a conditional to check if the entry is either the first of the last to be able to insert different values into these. The calculations of the h_*'s are quite different. Again here I started of with the most basic BLAS and LAPACK subroutines to be able to get the code to work first. I started with dgesv to perform the solve functions. Later I swapped to dptsv to be able to save a little memory for the auxiliary vectors and this is the most optimised subroutine for SPD tridiagonal matrices. To test this I used the equations in (16) to "reverse" the method to check that the starting value was the same. This was written into the same program and then compiled as part of the main program before testing. Again test-banded was implemented, optimised and tested the same way as test_crank. First it was written so it would work and then optimised and tested further using the methods already spoken about. As in test-crank, I used the create_matrices subroutine to add the M_{**} matrices into the subroutine. This was the compromise between memory and cpu time. This was used more memory but was quicker overall as it could use more optimised BLAS and LAPACK subroutines and functions.

I repeated the same procedure as in test_crank to calculate u_{in} and v_{in} from u_{test} and v_{test} and then called banded_crank and calculated the norm of the difference between u_{in} u_{test} and v_{in} v_{test} resp. and these results were then printed. As a late addition here I added a calls to daxpy to calculate the difference between the vectors before calculating the norms of the results of these calls. This was a small optimisation that I later added to test_crank too. To test both of these, I was looking for a result the same of that as in test_crank. Inputting certain vectors rather than purely random ones allowed me to be certain that these were producing the same results. The errors were also similar.

Q5: Improved time stepping method

In the file `timestepping_improved.f90` **implement** an improved version of the subroutine from Part I called `timestepping_improved(n,Blu,Blv,Bru,Brv,hu,hv,tau,T,F,k,u,v)`. Instead of the dense matrices $\mathbf{M}_{lu}, \mathbf{M}_{lv}, \mathbf{M}_{ru}, \mathbf{M}_{rv}$, this subroutine should be passed the banded matrices $\mathbf{B}_{lu}, \mathbf{B}_{lv}, \mathbf{B}_{ru}, \mathbf{B}_{rv}$, and the correction vectors $\mathbf{h}_u, \mathbf{h}_v$. The solution vectors $\mathbf{u}, \mathbf{v}$ should be $\mathbf{u}_0, \mathbf{v}_0$ on input, and $\mathbf{u}_{T/\tau}, \mathbf{v}_{T/\tau}$ on return.
You will compare the performance of the basic version and the improved version in Part IV. **Modify** the main program `grayscott.f90` to call the new subroutines using banded matrices. **Write a short report** on how you modified your code to use the banded matrices and how you checked it for correctness.

[5 points]

For the timestepping_improved subroutine itself, I just copied the original subroutine over and changed the inputs to the B_{**}'s and the calls of crank_nicholson to banded_crank. The call to heun was left the same as was the number of calls to each of these subroutines in the do loop. After creating this subroutine I added it to the main program grayscott and started testing. This started with really small matrices (done by changing n in the input.dat file) and inserting the same custom initial conditions as with the grayscott program for the normal crank. These led to the same results and hence the code was correct. Again here visualise.py was used to view the solution data on a plot as it was much easier to compare the data quickly than just reading the solution.dat file.

Part III: Theory

Q6: Consistency of the Strang splitting

Assuming that the initial condition $\mathbf{y}_\ell = \mathbf{y}(t_\ell)$ is exact, prove that one iteration of Algorithm 1 has the second order of consistency, that is,

$$\|\mathbf{y}_{\ell+1} - \mathbf{y}(t_\ell + \tau)\|_2 = \mathcal{O}(\tau^3).$$

You may assume without proof that the 2-norms of $\mathbf{y}(t)$ and $\mathbf{N}(\mathbf{y}(t))$ on $t \in [t_\ell, t_\ell + \tau]$, and the induced operator norms of the Jacobian and the Hessian of $\mathbf{N}(\mathbf{y})$ are all bounded.
Hint: Use Taylor series and the fact that Crank-Nicolson and Heun methods are 2nd order too.

[4 points]

First we start at line 4 of Alg.1 and taylor expand around the solution z (with the initial condition of $z_0 = y_\ell$):

$$z(\tau/2) = y_\ell + \frac{\tau}{2}Ay_\ell + \frac{\tau^2}{8}A^2y_\ell + \mathcal{O}(\tau^3) \qquad (1)$$

Next we linearise the system in line 4 of Alg.1 to give us:

$$\frac{d\underline{w}}{dt} = N(\underline{w}) \Leftrightarrow \frac{d\underline{w}}{dt} = DN(w_o)\underline{w} \qquad (2)$$

This holds for small τ and as the induced operator norm of $DN(w_0)$ is bounded. We then can consider the Taylor expansion of the solution of w (Alg.1 line 5). Next is to substitute (1) as the initial condition for (2) (with $B = DN(w_0)$ here for simplicity):

$$\begin{aligned} w(\tau) &= w_0 + \tau Bw_0 + \frac{\tau^2}{2}B^2w_0 + \mathcal{O}(\tau^3) \\ &= y_\ell + \frac{\tau}{2}Ay_\ell + \frac{\tau^2}{8}A^2y_\ell + \tau B(y_\ell + \frac{\tau}{2}Ay_\ell) + \frac{\tau^2}{2}B^2y_\ell + \mathcal{O}(\tau^3) \\ &= y_\ell + \tau y_\ell(\frac{1}{2}A + B) + \tau^2 y_\ell(\frac{1}{8}A^2 + \frac{1}{2}BA + \frac{1}{2}B^2) + \mathcal{O}(\tau^3) \qquad (3) \end{aligned}$$

Again if we look at the Taylor expansion of the solution q at $\tau/2$ of line 6 Alg.1 and substitute $w(\tau)$ as the initial condition into this, we get:

$$q(\frac{\tau}{2}) = y_\ell + \tau y_\ell(A + B) + \frac{\tau^2}{2}y_\ell(A^2 + AB + BA + B^2) + \mathcal{O}(\tau^3)$$

Finally let's consider the Taylor expansion of $y(t_\ell + \tau)$:

$$\begin{aligned} y(t_\ell + \tau) &= y(t_\ell) + \tau y^{'}(t_\ell) + \frac{\tau^2}{2}y^{''}(t_\ell) + \mathcal{O}(\tau^3) \\ &= y_\ell + \tau y_\ell(A + B) + \frac{\tau^2}{2}y_\ell(A^2 + AB + BA + B^2) + \mathcal{O}(\tau^3) \end{aligned}$$

As we can see this looks very similar to our statement in (3). The τ^2 coefficient is also bounded as we have that the 2-norm of $N(y(t))$ as well as the operator norms of the Jacobian and Hessian of $N(y(t))$ are bounded. Hence we can consider the 2-norm:

$$\begin{aligned} \|\mathbf{y}_{\ell+1} - \mathbf{y}(t_\ell + \tau)\|_2 &= \|y_\ell + \tau y_\ell(A + B) + \frac{\tau^2}{2}y_\ell(A^2 + AB + BA + B^2) \\ &\quad - (y_\ell + \tau y_\ell(A + B) + \frac{\tau^2}{2}y_\ell(A^2 + AB + BA + B^2)) + \mathcal{O}(\tau^3)\|_2 \\ &= \|0\|_2 + \mathcal{O}(\tau^3) \\ &= \mathcal{O}(\tau^3) \end{aligned}$$

Hence we get that one iteration of the Strang Splitting method is of second order consistency.

Q7: Low rank updates

Prove the formula (14). Then, use Eq. (16) to show that $\mathbf{M}_{l*}\mathbf{h}_* = \tau_*\mathbf{c}$. Finally, prove that $\mathbf{M}_{l*}(\mathbf{I} + \mathbf{h}_*\mathbf{c}^\top) = \mathbf{B}_{l*}$, and formula (15). You can assume that $\rho_* \neq 1$.

[3 points]

$$
\begin{aligned}
M_{l*} &= \mathbb{I} - \frac{D_*\tau}{4}\Delta \\
&= \mathbb{I} - \frac{D_*\tau}{4*h^2}\begin{bmatrix} -2 & 1 & \dots & 1 \\ 1 & -2 & 1 & \dots \\ \vdots & \ddots & \ddots & \ddots \\ \dots & 1 & -2 & 1 \\ 1 & \dots & 1 & -2 \end{bmatrix} = \mathbb{I} + \tau_* \begin{bmatrix} 2 & -1 & \dots & -1 \\ -1 & 2 & -1 & \dots \\ \vdots & \ddots & \ddots & \ddots \\ \dots & -1 & 2 & -1 \\ -1 & \dots & -1 & 2 \end{bmatrix} \\
&= B_{l*} - \tau_* \begin{bmatrix} 1 & \dots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \dots & 1 \end{bmatrix} \\
&= B_{l*} - \tau_* cc^T
\end{aligned}
$$

$$
\begin{aligned}
M_{r*} &= \mathbb{I} + \frac{D_*\tau}{4}\Delta \\
&= \mathbb{I} + \frac{D_*\tau}{4*h^2}\begin{bmatrix} -2 & 1 & \dots & 1 \\ 1 & -2 & 1 & \dots \\ \vdots & \ddots & \ddots & \ddots \\ \dots & 1 & -2 & 1 \\ 1 & \dots & 1 & -2 \end{bmatrix} = \mathbb{I} - \tau_* \begin{bmatrix} 2 & -1 & \dots & -1 \\ -1 & 2 & -1 & \dots \\ \vdots & \ddots & \ddots & \ddots \\ \dots & -1 & 2 & -1 \\ -1 & \dots & -1 & 2 \end{bmatrix} \\
&= B_{r*} + \tau_* \begin{bmatrix} 1 & \dots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \dots & 1 \end{bmatrix} \\
&= B_{r*} + \tau_* cc^T
\end{aligned}
$$

$$
\begin{aligned}
M_{l*}h_* = B_{l*}h_* - \tau_* cc^T h_* \quad &= \frac{\tau_*}{1-\rho_*}B_{l*}\hat{h}_* - \tau_* cc^T h_* = \frac{\tau_*}{1-\rho_*}c - \frac{\tau_*}{1-\rho_*}\tau_* cc^T \hat{h}_* \\
= \frac{\tau_*}{1-\rho_*}(1-\rho_*)c \quad &= \tau_* c \\
M_{l*}(\mathbb{I} + h_* c^T) = M_{l*}\mathbb{I} + M_{l*}h_* c^T \quad &= M_{l*} + M_{l*}h_* c^T = M_{l*} + \tau_* cc^T \quad = B_{l*}
\end{aligned}
$$

$$
\begin{aligned}
M_{l*}(\mathbb{I} + h_* c^T) &= B_{l*} \\
\Leftrightarrow M_{l*}^{-1} M_{l*}(\mathbb{I} + h_* c^T) &= M_{l*}^{-1} B_{l*} \\
\Leftrightarrow (\mathbb{I} + h_* c^T) B_{l*}^{-1} &= M_{l*}^{-1} B_{l*} B_{l*}^{-1} \\
\Leftrightarrow (\mathbb{I} + h_* c^T) B_{l*}^{-1} &= M_{l*}^{-1}
\end{aligned}
$$

Part IV: Numerical experiments

In the experiments below, fix $n = 128$, $\tau = 0.4$, $T = 4000$, $F = 0.031$ and $k = 0.057$ unless instructed otherwise. Either Part I or Part II code can be used unless instructed otherwise.

Q8: Empirical error analysis

Run your code for $\tau = 0.1, 0.2, 0.4, 0.8, 1.6$ and 3.2, and for each τ print out the element $u_{n/2} =: u_{n/2}^{(\tau)}$ of the solution vector. This element approximates the exact solution $u(0.5, T)$. Compute the error estimates $u_{n/2}^{(\tau)} - u_{n/2}^{(2\tau)}$ and the experimental order of convergence (EOC)

$$\mathrm{EOC}_\tau = \log_2 \left(\frac{u_{n/2}^{(2\tau)} - u_{n/2}^{(4\tau)}}{u_{n/2}^{(\tau)} - u_{n/2}^{(2\tau)}} \right).$$

Populate the following table with the values you obtained. Determine the number of significant digits you need.

τ	$u_{n/2}^{(\tau)}$	$u_{n/2}^{(\tau)} - u_{n/2}^{(2\tau)}$	EOC_τ
0.1	0.790147365178	-0.000002147408	1.992493791218819
0.2	0.790149512586	-0.000008545057	1.971820143906065
0.4	0.790158057643	-0.000033519071	1.913152501997742
0.8	0.790191576714	-0.000126243278	2.035279209345958
1.6	0.790317819992	-0.000517473786	—
3.2	0.790835293778	—	—

As we double τ we see that we quadruple the difference between successive values of $u_{n/2}$. This then leads to an EOC_τ of about 2 for each τ. This shows that the convergence rate is of order τ^2 using the method that we use here.

[3 points]

Q9: Metastable patterns of the concentration

Produce plots of the solution $\mathbf{u}, \mathbf{v}$ for three different values of the kill rate $k = 0.055, 0.057$ and 0.061. You can use the subroutine `save_fields(n,u,v,'solution.dat')` (in `save_fields.f90`) to save the vectors $\mathbf{u}, \mathbf{v}$ to the file `solution.dat`. If you then call

```
python visualise.py
```

from the command line, the data in `solution.dat` will be plotted to the file `solution.pdf`, which you can view and save after copying it to your H: drive.
Note that the Gray-Scott equations (2),(3) have homogeneous equilibria $u = 1/2$, $v = \sqrt{F}$ for $k = \sqrt{F/4} - F$, and $u = 1$, $v = 0$ for any k, F. Comparing the three suggested values of k to the phase curve $k = \sqrt{F/4} - F$ and to the solutions on the plots, discuss stability of those equilibria.

[2 points]

For $k = 0.055$ we have what appears to be quite random behaviour. This therefore appears to show unstable behaviour of both equilibria. For $k = 0.057$, we get that k is very close to the phase curve and therefore displays stable behaviour in the sense that it is a center. This would be represented as an ellipse on a phase plane between u and v over time around the point $(u, v) = (0.5, \sqrt{F})$. If k were exactly equal to $\sqrt{F/4} - F$, then u and v would converge to the respective equilibrium. When $k = 0.061$ we appear to cross over into the thresh hold for stable behaviour, where u and v converge to 1 and 0 resp. This is represented on the graph. Strangely we appear to have a hump in the graph for v but this is on a scale of e-157 so quite negligible. I suspect this is just calculation errors over time.

Q10: Performance measurements

Run your Part I code for a range of problem sizes n and plot the CPU time **of one iteration** of Alg. 1 on a log-log plot. The total time in seconds is measured with the `cpu_time()` method which has an accuracy of 0.01s, so you need to choose n or T large enough such that the total time is about half a second at least. Remember to compile the code in "optimised" mode by setting the appropriate compiler flags. Repeat the same numerical experiment for the optimised code written in Part II (you might have to choose larger values of n or T) and plot the results in the same log-log plot. Compare the measured slopes with what is predicted by the theory, i.e. the leading terms in the cost functions.

[4 points]

The plot is added to the bottom of this submission document. The blue line represents the log-log of matrix size against CPU-time for crank_nicholson and the green line is that for banded_crank. We can see in the top left that the equations for these lines have gradient approx. 2 and 1 resp. This implies that the CPU-time is order n^2 and n resp. This is what we expect from our programs as in timestepping we use storage that is of order n^2 for the matrices and for timestepping_improved we have storage $\mathcal{O}(n)$. This overall, with the improved algorithms that can be used for these banded matrices, reduces the order of the CPU-time to order n rather than n^2.

IMPORTANT: 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. `assign1_smith` or `assignment1_skl27`. This directory should contain two subdirectories, `PartI` and `PartII` which should each contain a fully working implementation of the code implemented in Part I and Part II respectively.
Zip up the directory (e.g. `assignment1_skl27`) into a single file using the command

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

and then upload the tgz-file that you obtain **and** scans or compiled documents of reports, figures and theoretical derivations on Moodle at Assignment 1 Submission Point.

General Remarks

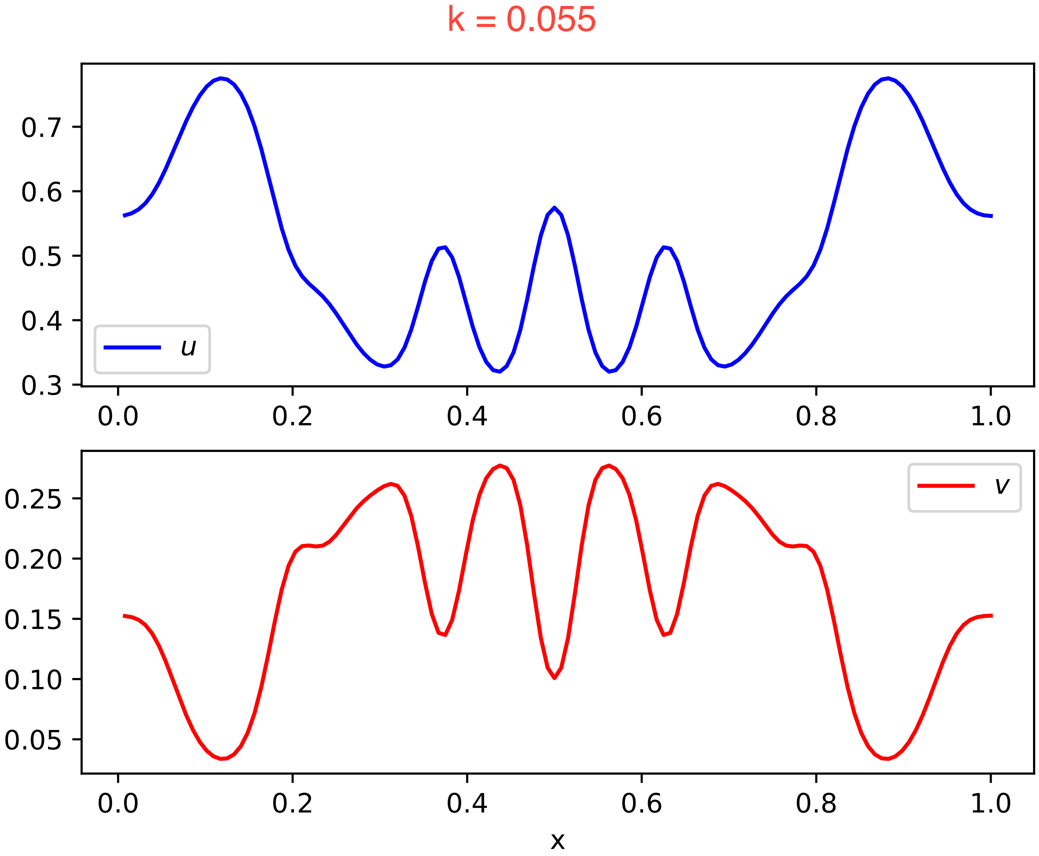
- Write routines and programs to test parts of your code.
- Support your observations with numerical results. You can also use graphs.
- Typical trends for convergence and cost with respect to some variable n are αn^β and $\alpha\beta^n$ for some constants α, $\beta \in \mathbb{R}$.
- Use as many subroutines 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
- Together with your Fortran code and Makefile, provide a `README` file explaining how to compile and run your 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.

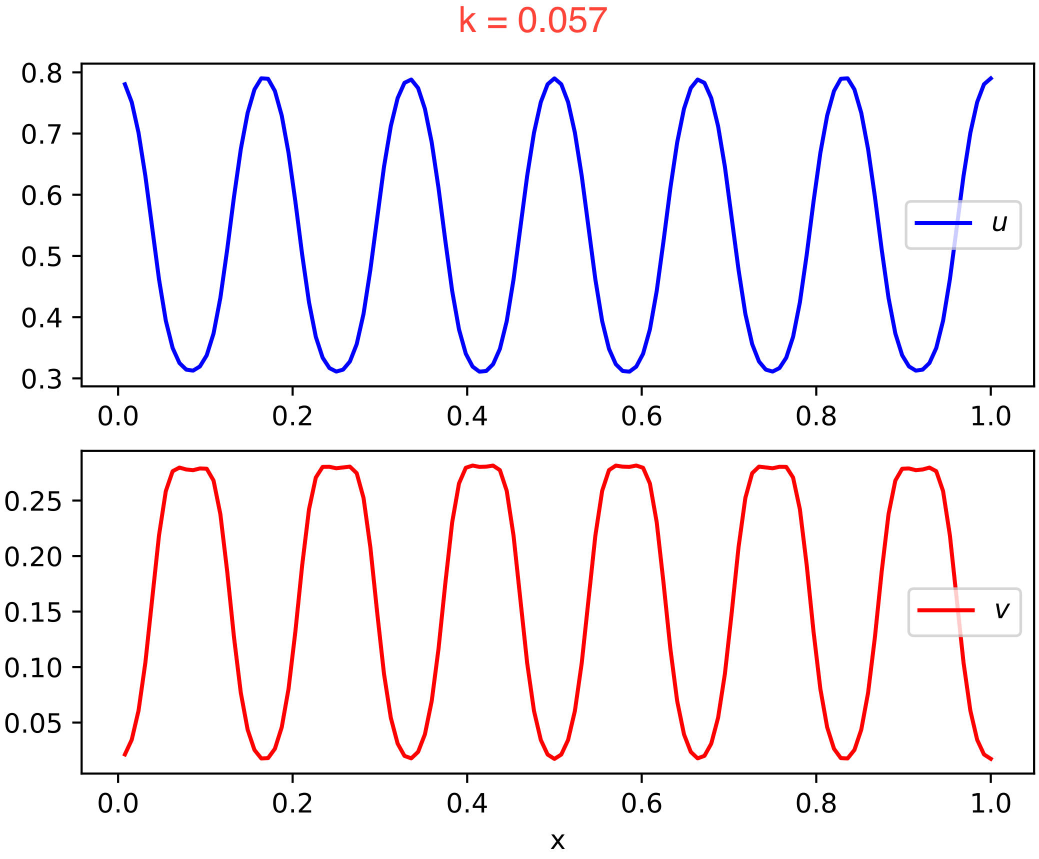
References

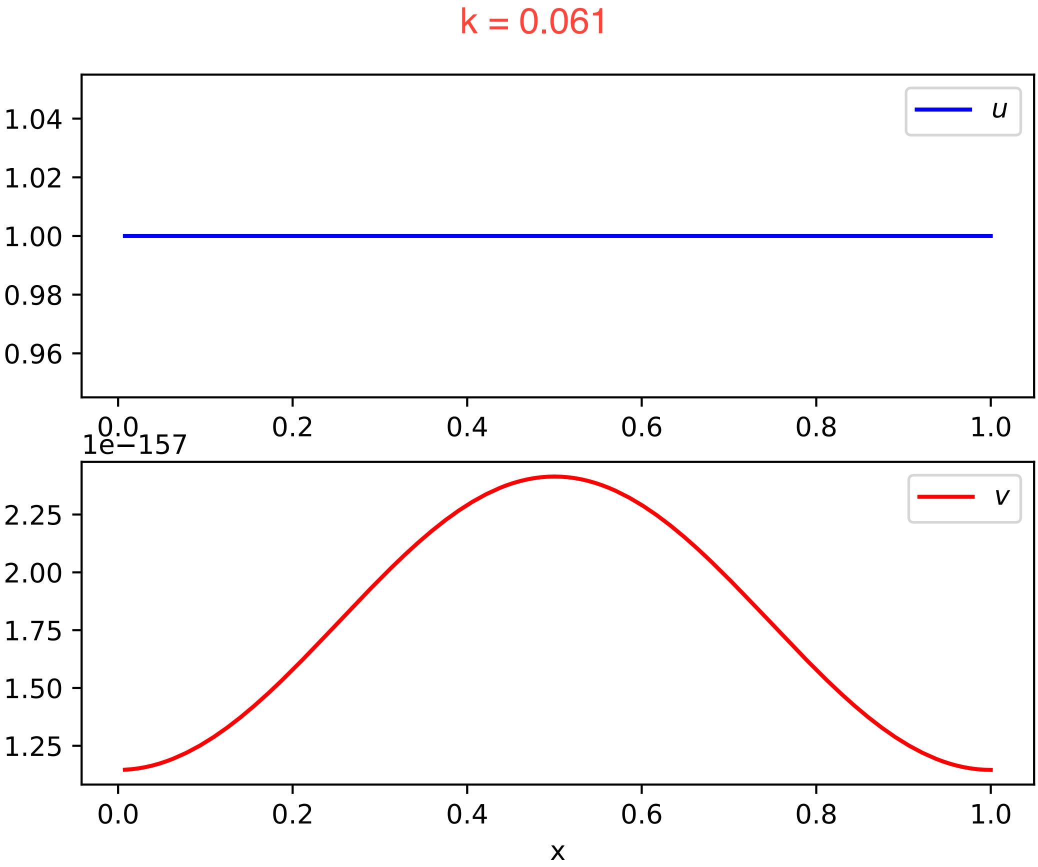
[1] A. M. Turing, The chemical basis of morphogenesis. *Phil. Trans. R. Soc. Lond. B,* **237**: 37–72 (1952). `https://doi.org/10.1098/rstb.1952.0012`

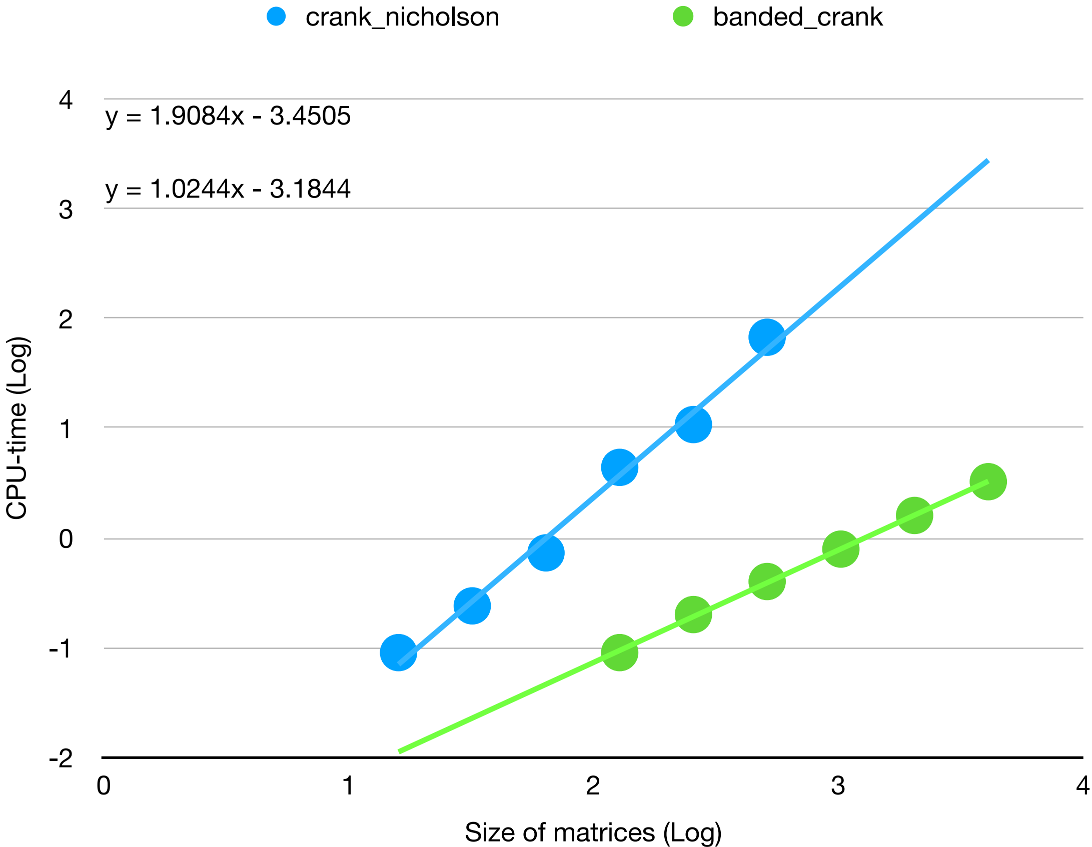
[2] J. E. Pearson, Complex Patterns in a Simple System. *Science*, **261 (5118)**: 189-192 (1993). `https://doi.org/10.1126/science.261.5118.189`

[3] S. MacNamara and G. Strang, Operator Splitting. *In: R. Glowinski, S. Osher, W. Yin (eds) Splitting Methods in Communication, Imaging, Science, and Engineering. Scientific Computation. Springer, Cham.* (2016). `https://doi.org/10.1007/978-3-319-41589-5_3`


k = 0.055
0.3
0.4
0.5
0.6
0.7
u
0.0
0.2
0.4
0.6
0.8
1.0
0.05
0.10
0.15
0.20
0.25
v
0.0
0.2
0.4
0.6
0.8
1.0
x


k = 0.057
0.8
0.7
0.6
0.5
0.4
0.3
0.0
0.2
0.4
0.6
0.8
1.0
u
0.25
0.20
0.15
0.10
0.05
0.0
0.2
0.4
0.6
0.8
1.0
v
x


k = 0.061
1.04
1.02
1.00
0.98
0.96
u
0.0
0.2
0.4
0.6
0.8
1.0
1e−157
2.25
2.00
1.75
1.50
1.25
v
0.0
0.2
0.4
0.6
0.8
1.0
x


● crank_nicholson
● banded_crank
y = 1.9084x - 3.4505
y = 1.0244x - 3.1844
4
3
2
1
0
-1
-2
CPU-time (Log)
0
1
2
3
4
Size of matrices (Log)