Project 3
Numerical methods for Differential Equations

Daniel Holm, Erik Olsson

December 2019

The diffusion equation

The diffusion equation is formulated as follows:

$$\begin{aligned} u_t &= u_{xx} \\ u(t,0) &= u(t,1) = 0 \\ u(0,x) &= g(x) \end{aligned} \tag{1}$$

To approximate the second order derivative with regards to x, we will use the symmetric tridiagonal matrix $T_{\Delta x}$, which we used in previous projects. We are also going to investigate the stability of these methods with regards to different step sizes (Δt, Δx).

Solving the equation

We chose $g(x) = 0.5 - |x - 0.5|$ as our initial state. Then we use Euler's explicit method to calculate the function values, $u_{m+1} = u_m + \Delta t \cdot T_{\Delta x} u_m$. The computation for this was quite simple:

```
def eulerstep(Tdx, uold, dt):
    return uold + dt * Tdx.dot(uold)
```

We can now write a solver method `parEul(gvec, tf, N, M)`, where `gvec` is an array of initial values, `tf` is the final time and `N` and `M` are the amount of space and time points, respectively. The resulting function is returned in a matrix for plotting.

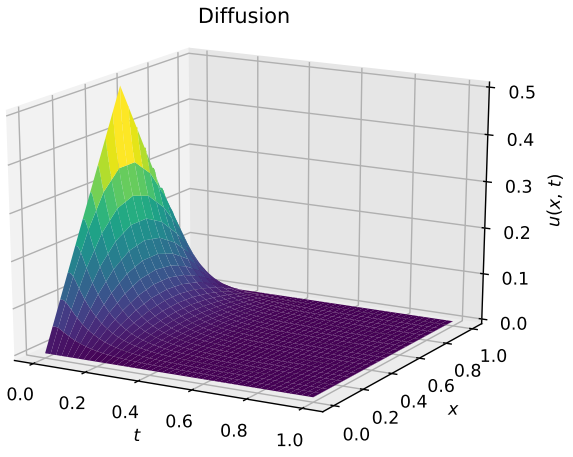

Figure 1: Simulation for one time unit with 19 space points and 800 time points, using an explicit method.

In Figure 1 we can see a plausible solution for (1), with the given $g(x)$.

Checking for stability

Because of the conditions of (1) it is impossible for the function value to ever, at any point in space, be greater than the maximum value of the initial state, since no material is created. Therefore we can use this as our stability check for the computed function values. If the returned matrix has any value greater than 0.5 we know for certain that the solution is invalid. In order to experimentally determine at what CFL number $\frac{\Delta t}{\Delta x^2}$ the solution is unstable we fixate the amount of space points (we chose 19) and only increment the amount of time points (starting from 800), each time checking for stability according to the above criteria. This way we can approximate the CFL condition.

In Figure 2 we can see that our condition for instability is fulfilled. The CFL number for this simulation is 0.5051. The CFL number in Figure 1 is 0.4999. So the CFL condition $\frac{\Delta t}{\Delta x^2} \lesssim \frac{1}{2}$ seems reasonable. In fact, further testing showed signs of instability even before our criteria above was met. The oscillations that start to happen in Figure 2 can be observed as early as with 794 time points (see Figure 3). The difference is that the oscillations are sufficiently small to pass through our test.

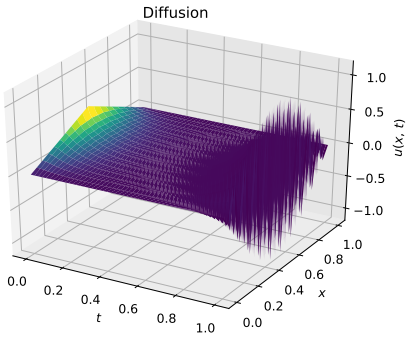

Figure 2: Simulation for one time unit with 19 space points and 792 time points, using an explicit method

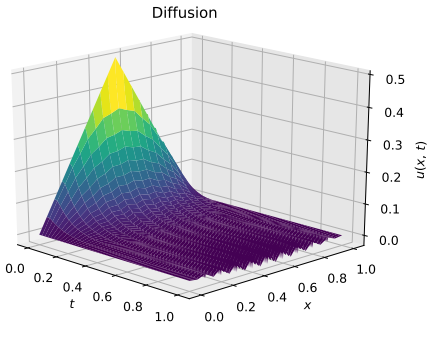

Figure 3: Simulation for one time unit with 19 space points and 794 time points, using an explicit method

This strengthens the claim that the condition is $CFL \leq \frac{1}{2}$.

The Crank-Nicolson method

We will use the trapezoidal rule to solve the equation. We now take implicit steps according to

$$u_{m+1} = u_m + \frac{\Delta t}{2}(T_{\Delta x} u_m + T_{\Delta x} u_{m+1}).$$

The advantage here is that we can take much larger time steps without sacrificing stability. The steps were taken using the following code:

```
def TRstep(Tdx, u, dt):
    I = identity(len(u))
    A = I - dt / 2 * Tdx
    b = (I + dt / 2 * Tdx).dot(u)
    return linalg.spsolve(A, b)
```

We can now modify our previous solver to use this step method instead of `eulerstep`.

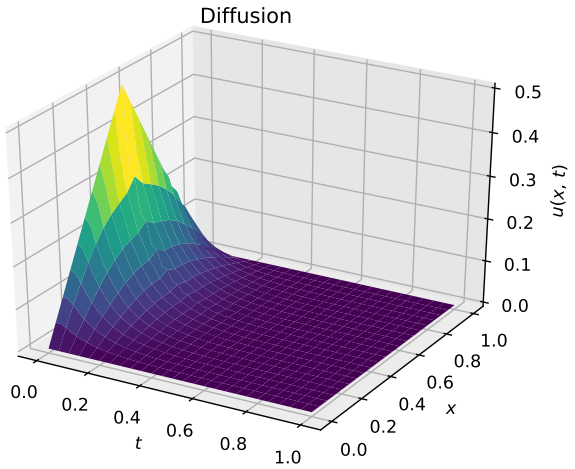

Figure 4: Simulation for one time unit with 19 space points and 50 time points, using an implicit method

When comparing Figures 1 and 4 we get very similar results, while computing a fraction of the points required for the explicit method. Note however the peaks at $x = 0.5$ early on in the simulation. Despite these slight inaccuracies the general "look" of the function is correct.

The advection equation

We will work with the linear advection equation

$$u_t + au_x = 0, \tag{2}$$

with periodic boundary conditions. Specifically, this means that $u(t, 0) = u(t, 1)$ for all times t. This also means that the initial condition $u(0, x) = g(x)$ must satisfy $g(0) = g(1)$ and $g'(0) = g'(1)$. In order to discretize this problem we will use an 2nd order method known as the Lax-Wendroff scheme. We will derive it using the Taylor series expansion to approximate a step in time, i.e.

$$u(t + \Delta t, x) \approx u(t, x) + \Delta t u_t + \frac{\Delta t^2}{2!} u_{tt}. \tag{3}$$

Using (2) we can replace some of the terms. Differentiating the equation in time gives $u_{tt} = -au_{xt}$ and in space gives $u_{tx} = -au_{xx}$. Together this gives $u_{tt} = a^2 u_{xx}$. Inserting this into the Taylor expansion we get

$$u(t + \Delta t, x) \approx u(t, x) - a\Delta t u_x + \frac{a^2 \Delta t^2}{2!} u_{xx}.$$

Now we discretize in space as well, replacing the space derivatives with symmetric finite differences. Finally, we have the Lax-Wendroff scheme,

$$u_n^{m+1} = \frac{a\mu}{2}(1 + a\mu)u_{n-1}^m + (1 - a^2\mu^2)u_n^m - \frac{a\mu}{2}(1 - a\mu)u_{n+1}^m,$$

where $\mu = \frac{\Delta t}{\Delta x}$. This means that the next value (in time) at a specific point in space is computed as a function of the current value and its two closest neighbors (in space). Note here that because of the periodic boundary conditions when we compute for the boundary values one of the "neighbors" is actually the other boundary value. In the code this was computed with a simple for-loop:

```
def LaxWenstep(u, amu):
    unew = np.zeros(len(u))

    for i in range(len(u) - 1):
        unew[i] = amu / 2 * (1 + amu) * u[i - 1] + (1 - amu ** 2) * u[i] -\
                  amu / 2 * (1 - amu) * u[i + 1]

    unew[-1] = amu / 2 * (1 + amu) * u[-2] + (1 - amu ** 2) * u[-1] - \
               amu / 2 * (1 - amu) * u[0]

    return unew
```

Next we implemented a solver `LaxWensolver(gvec, a, tf, N, M)` where `gvec` is an array of initial values, `a` is a from (2), `tf` is the final time and `N` and `M` is the amount of space and time points, respectively. The resulting function is returned in a matrix for plotting. The solver was tested for $g(x) = e^{-100(x-0.5)^2}$ and seeing how this pulse propagated for 5 units of time using various choices of a (both positive and negative).

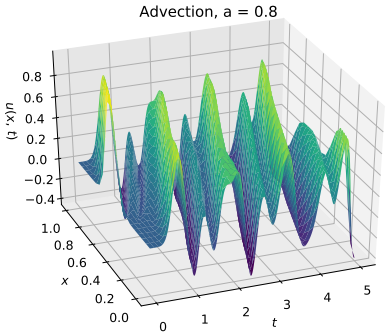

Figure 5: Simulation for five time units with 19 space points and 5000 time points. (CFL = 0.019)

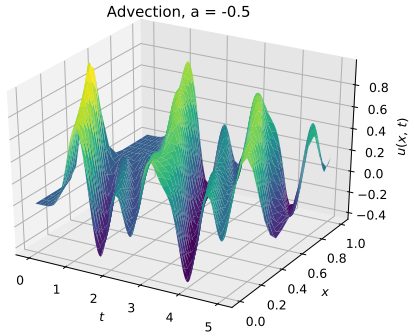

Figure 6: Simulation for five time units with 17 space points and 2500 time points. (CFL = 0.0034)

In Figures 5 and 6, we can see two examples of the advection with $a > 0$ and $a < 0$.

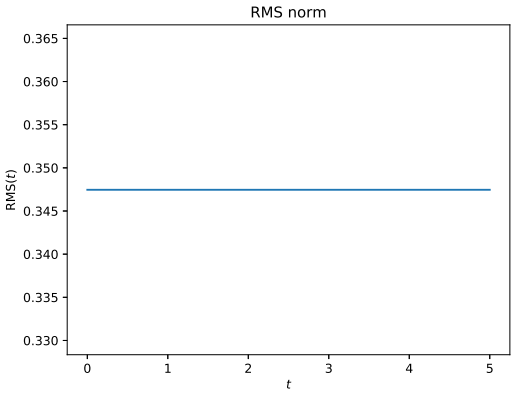

Figure 7: Norm of u, when $a\mu = 1$.

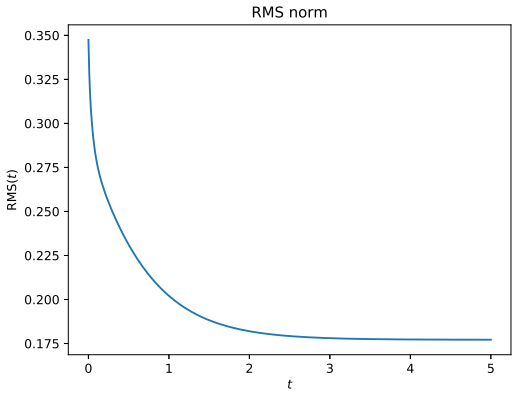

Figure 8: Norm of u, when $a\mu = 0.9$.

In Figures 7 and 8 above we see the RMS norm for two different choices of $a\mu$. Both of the simulations were done with 9 space points and 1000 time points. Because of the conditions of (2), the norm of an analytical solution should always be constant. From a physical point of view the equation describes transport of material and since we have periodicity it makes no sense for material to be created or eliminated. However, the RMS norm plots tell a different story. The conclusion we draw here is that the Lax-Wendroff method only conserves the norm of the solution for certain choices of $a\mu$. This means that even though the numerical solutions might look "nice and realistic" they should be taken with a grain of salt as the discretization can apply artificial dampening. This is supported in theory as at $a\mu$ the Lax-Wendroff scheme solves the equation exactly. Therefore it is preferable to run the scheme at the stability limit.

The convection-diffusion equation

The linear convection-diffusion equation is

$$u_t + a \cdot u_x = d \cdot u_{xx}, \tag{4}$$

where a is the convection velocity and d is is the diffusivity. We will work with the same kinds of periodic boundary conditions as before. An interesting ratio to consider in this problem is the Péclet number $Pe = |\frac{a}{d}|$ which describes the balance between the convective and diffusive terms. The solution is convection dominated at high Péclet numbers and diffusion dominates at low Pe. Another important quantity is the mesh Péclet number $Pe\Delta x$ which also puts a sort of restriction on the discretization. This number, however, is independent of time stepping and stability and is instead a measure of whether the discretization has similar properties to the differential operator. There is typically a condition associated with this number that is unknown to us. But in general we can say that for higher Péclet numbers (convection dominated), Δx has to be smaller.

Because of the diffusion, this problem is stiff and it is therefore preferable to use implicit time steps. We will use the trapezoidal method, which is second order in time, combined with a second

order in space discretization. By discretizing in space the equation becomes

$$\dot{u} = (d \cdot T_{\Delta x} - a \cdot S_{\Delta x})u,$$

where $T_{\Delta x}$ is the usual matrix representing second order symmetric finite differences with but now with ones in the outermost corners (because of the periodicity). $S_{\Delta x}$ is instead a matrix representing first order symmetric finite differences with either one or negative one in the outermost corners (because of periodicity). In the code this was done by generating the matrices and then taking the step using the trapezoidal rule (we actually reuse the previously written `TRstep` but now with a slightly more complex matrix):

```
def convdifstep(u, a, d, dt):
    Sdx, Tdx = generateST(len(u))
    return TRstep(d * Tdx - a * Sdx, u, dt)
```

Now we can implement a solver method `convdifsolver(gvec, a, d, N, M, tf=1)` and plot the results.

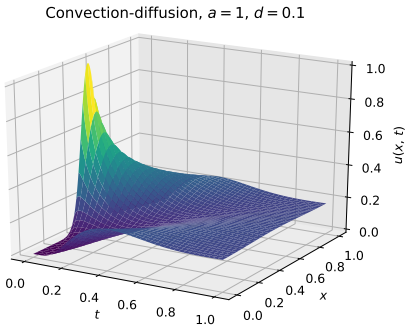

Figure 9: Simulation for one time unit with 200 space and time points. $Pe = 10$.

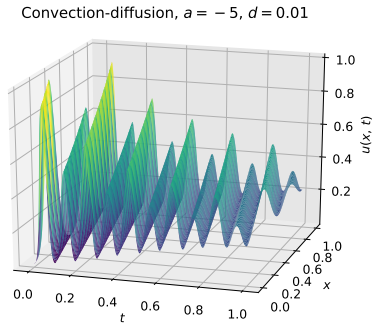

Figure 10: Simulation for one time unit with 500 space and time points. $Pe = 500$.

In Figure 9 we used the same pulse as in the previous task ($g(x) = e^{-100(x-0.5)^2}$). We see that the solution flattens out quite quickly due to diffusion. In Figure 10 we instead used a "double pulse" $g(x) = e^{-100(x-0.25)^2} + 0.7e^{-100(x-.75)^2}$ as our initial state as well as changing the constants. As expected the solution is convection dominated and does not flatten out as quickly since there is very little diffusion.

The viscous Burgers equation

The viscous Burgers equation is

$$u_t + u \cdot u_x = d \cdot u_{xx}. \tag{5}$$

It is very similar to (4), however in this case the convection velocity is the function itself. This means that the equation is non-linear, and the calculations are gonna get somewhat more complicated. We

are gonna divide the calculations into two steps. First, we are gonna implement a Lax-Wendroff scheme to solve the inviscid Burgers equation, $u_t = -uu_x$. We do this by differentiating this equation with regards to both t and x. Then we can re-write all the time-derivatives as expressions of u, u_x and u_{xx}.

$$u_{tt} = -u_t u_x - uu_{xt} = uu_x^2 - uu_{xt}$$
$$u_{tx} = -u_x u_x - uu_{xx}$$
$$u_{tt} = uu_x^2 - u(-u_x^2 - uu_{xx}) = 2uu_x^2 + u^2 u_{xx}$$

Inserting this into (3) gives us

$$u(t + \Delta t, x) \approx u(t, x) - \Delta t u u_x + \frac{\Delta t^2}{2!}(2uu_x^2 + u^2 u_{xx}).$$

Next we replace the space derivatives with symmetric finite differences. The resulting code is as follows:

```
def LW(u, dx, dt):
    unew = np.zeros(len(u))

    for i in range(len(u) - 1):
        unew[i] = u[i] - dt * u[i] * (u[i + 1] - u[i - 1]) / (2 * dx)
        unew[i] += dt ** 2 / 2 * (2 * u[i] * ((u[i + 1] - u[i - 1]) / (2 * dx)) ** 2)
        unew[i] += dt ** 2 / 2 * u[i] ** 2 *\
                   (u[i + 1] - 2 * u[i] + u[i - 1]) / (dx ** 2)

    unew[-1] = u[-1] - dt * u[-1] * (u[0] - u[-2]) / (2 * dx)
    unew[-1] += dt ** 2 / 2 * (2 * u[-1] * ((u[0] - u[-2]) / (2 * dx)) ** 2)
    unew[-1] += dt ** 2 / 2 * u[-1] ** 2 * (u[0] - 2 * u[-1] + u[-2]) / (dx ** 2)

    return unew
```

The next step is to add the diffusion term to the scheme. We are using the Trapezoidal Rule, so the expression will be:

$$u^{m+1} = LW(u^m) + d \cdot \frac{\Delta t}{2}(T_{\Delta x} u^m + T_{\Delta x} u^{m+1}) \tag{6}$$

$T_{\Delta x}$ is the regular numerical approximation matrix for the second derivative. It is also circulant, as the function is periodic. We have now obtained the final second order discretized form of the viscous Burgers equation, where the nonlinear convective part is handled by the explicit Lax-Wendroff method, and the diffusion part is handled by the Trapezoidal Rule. Computing u^{m+1} according to (6) was done using the following code:

```
def burgersstep(Tdx, u, d, dx, dt):
    I = identity(len(u))
    A = I - d * dt / 2 * Tdx
    b = LW(u, dx, dt) + d * dt / 2 * Tdx.dot(u)
    return linalg.spsolve(A, b)
```

Finally we write a solver method `burgerssolver(gvec, d, N, M, tf=1)` and plot the results.

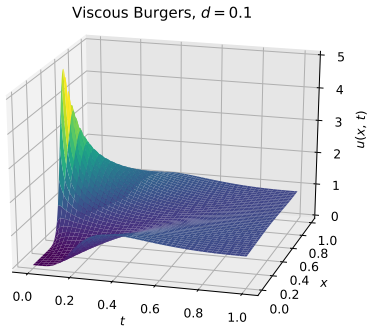

Figure 11: Simulation for one time unit with 200 space points and 1000 time points.

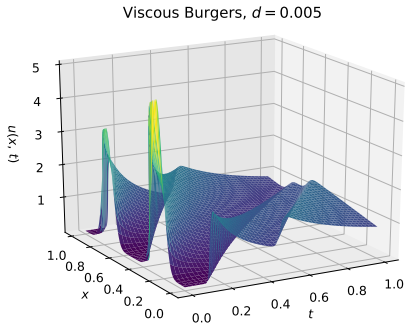

Figure 12: Simulation for one time unit with 300 space points and 2000 time points.

In Figure 11 we used the same pulse initial state as in previous tasks, this time scaled by 5, $g(x) = 5e^{-100(x-0.5)^2}$. With Figure 12 we tried to produce waves with very steep gradients. This was accomplished by having a double pulse similar to the one in the previous task, $g(x) = 5(e^{-200(x-0.25)^2} + 0.7e^{-200(x-0.75)^2})$, as our initial state and choosing a very small diffusion constant. Note the explosive growths at $x = 0$ that then move along the x-axis as time progresses.

Conclusions

In this project, we worked side by side, still writing our individual code, but there were more discussions and suggestions between us compared to the previous project. We chose to present Daniel's code, mostly because his code was a little more structured. We learned some general principles about numerically solving PDE:s, a lot about stability, the benefits (and drawbacks) of using implicit methods. It was sometimes difficult to confirm if a plot was legitimate, as our knowledge in physics is quite limited.

References

All the theory comes from the lecture notes and project instructions.

```
import numpy as np
from scipy.sparse import diags, linalg, identity


def generateT(N, L=1):
    dx = L / (N + 1)
    diagonals = [np.ones(N - 1), -2 * np.ones(N), np.ones(N - 1)]
    return 1 / (dx ** 2) * diags(diagonals, [-1, 0, 1], format='csr')


def generateST(N):
    dx = 1 / N
    diagonals = [np.ones(N - 1), -2 * np.ones(N), np.ones(N - 1)]
    Tdx = diags(diagonals, [-1, 0, 1], format='csr')
    Tdx[-1, 0] = Tdx[0, -1] = 1
    Tdx *= 1 / (dx ** 2)
    Sdx = diags((-np.ones(N - 1), np.ones(N - 1)), [-1, 1], format='csr')
    Sdx[-1, 0] = 1
    Sdx[0, -1] = -1
    Sdx *= 1 / (2 * dx)

    return Sdx, Tdx


def eulerstep(Tdx, uold, dt):
    return uold + dt * Tdx.dot(uold)


def parEul(gvec, tf, N, M):
    dt = tf / M
    dx = 1 / (N + 1)
    print(f"CFL = {dt / (dx ** 2)}")
    Tdx = generateT(N)
    U = np.zeros((N, M + 1))
    U[:, 0] = gvec

    for i in range(M):
        U[:, i + 1] = eulerstep(Tdx, U[:, i], dt)

    U = np.vstack((np.zeros(M + 1), U, np.zeros(M + 1)))

    return U


def testCFL(g, tf=1, start=800, N=19):
    i = start
```

```
    gvec = g(np.linspace(0, 1, N + 2)[1:-1])
    U = parEul(gvec, tf, N, i)
    while np.any(U > .5) == False:
        i -= 1
        U = parEul(gvec, tf, N, i)

    print('CFL Limit ^')


def TRstep(Tdx, uold, dt):
    I = identity(len(uold))
    A = I - dt / 2 * Tdx
    b = (I + dt / 2 * Tdx).dot(uold)
    return linalg.spsolve(A, b)


def parTR(gvec, tf, N, M):
    dt = tf / M
    dx = 1 / (N + 1)
    print(f"CFL = {dt / (dx ** 2)}")
    Tdx = generateT(N)
    U = np.zeros((N, M + 1))
    U[:, 0] = gvec

    for i in range(M):
        U[:, i + 1] = TRstep(Tdx, U[:, i], dt)

    U = np.vstack((np.zeros(M + 1), U, np.zeros(M + 1)))

    return U

def LaxWenstep(u, amu):
    unew = np.zeros(len(u))

    for i in range(len(u) - 1):
        unew[i] = amu / 2 * (1 + amu) * u[i - 1] + (1 - amu ** 2) * u[i] - amu / 2 * (1 - am

    unew[-1] = amu / 2 * (1 + amu) * u[-2] + (1 - amu ** 2) * u[-1] - amu / 2 * (1 - amu) *

    return unew


def LaxWensolver(gvec, a, tf, N, M):
    dt = tf / M
    dx = 1 / N
    print(f"CFL = {dt / dx}")
```

```
    amu = a * dt / dx
    print(f"amu = {amu}")
    U = np.zeros((N, M + 1))
    U[:, 0] = gvec

    for i in range(M):
        U[:, i + 1] = LaxWenstep(U[:, i], amu)

    U = np.vstack((U, U[0, :]))

    return U


def LaxWenRMS(U, N, M):
    l2 = np.zeros(M + 1)
    for i in range(M + 1):
        l2[i] = np.linalg.norm(U[:-1, i])
    return l2 * 1 / np.sqrt(N)


def convdifstep(u, a, d, dt):
    Sdx, Tdx = generateST(len(u))
    return TRstep(d * Tdx - a * Sdx, u, dt)


def convdifsolver(gvec, a, d, N, M, tf=1):
    dt = tf / M
    dx = 1 / N
    print(f"Pe = {abs(a / d)}")
    print(f"MeshPe = {abs(a / d) * dx}")
    U = np.zeros((N, M + 1))
    U[:, 0] = gvec

    for i in range(M):
        U[:, i + 1] = convdifstep(U[:, i], a, d, dt)

    U = np.vstack((U, U[0, :]))

    return U


def LW(u, dx, dt):
    unew = np.zeros(len(u))

    for i in range(len(u) - 1):
        unew[i] = u[i] - dt * u[i] * (u[i + 1] - u[i - 1]) / (2 * dx)
```

```
        unew[i] += dt ** 2 / 2 * (2 * u[i] * ((u[i + 1] - u[i - 1]) / (2 * dx)) ** 2)
        unew[i] += dt ** 2 / 2 * u[i] ** 2 * (u[i + 1] - 2 * u[i] + u[i - 1]) / (dx ** 2)

    unew[-1] = u[-1] - dt * u[-1] * (u[0] - u[-2]) / (2 * dx)
    unew[-1] += dt ** 2 / 2 * (2 * u[-1] * ((u[0] - u[-2]) / (2 * dx)) ** 2)
    unew[-1] += dt ** 2 / 2 * u[-1] ** 2 * (u[0] - 2 * u[-1] + u[-2]) / (dx ** 2)

    return unew


def burgersstep(Tdx, u, d, dx, dt):
    I = identity(len(u))
    A = I - d * dt / 2 * Tdx
    b = LW(u, dx, dt) + d * dt / 2 * Tdx.dot(u)
    return linalg.spsolve(A, b)


def burgerssolver(gvec, d, N, M, tf=1):
    Tdx = generateST(N)[1]
    dt = tf / M
    dx = 1 / N
    U = np.zeros((N, M + 1))
    U[:, 0] = gvec

    for i in range(M):
        U[:, i + 1] = burgersstep(Tdx, U[:, i], d, dx, dt)

    U = np.vstack((U, U[0, :]))

    return U
```