Homework 2

Mats Thijssen

September 27, 2017

1 Introduction

In this exercise I estimate integrals using the techniques of Romberg Integration and Gaussian Quadrature. I show that these method, while taking a while to converge to the actual result, becoming very accurate without a high cost of computing power.
I also apply a Forward in Time, Centered in Space (FTSC) scheme to solve a 1D scalar advection equation. I show that the straight-forward application creates undamped exponential behaviour in part of the solution. This can be fixed by applying Lax-Friedrich method which damps the exponential behaviour but preserves the solution, unless the function is very narrowly peaked. The various parts of this report sort of turned into a big mash, but it felt the most natural to do it like this. Apologies in advance if it seems unstructured!

2 Theory/Discussion/Results

2.1 Integration

2.1.1 Romberg Integration

Romberg Integrations works as a sort of iterative version of the trapezoidal rule. The leading order of error in the trapezoidal rule is:

$$ch_i^2 = \tfrac{1}{3}(I_i - I_{i-1})$$

for some constant c. By plugging into the true value of the integral, $I = I_i + ch_i^2 + O(h_i^4)$, we get:

$$I = I_i + \tfrac{1}{3}(I_i - I_{i-1}) + O(h_i^4)$$

which is now accurate to third order. For simplicity, we define:

$$R_{i,1} = I_i \qquad R_{i,2} = I_i + \tfrac{1}{3}(I_i - I_{i-1}) = R_{i,1} + \tfrac{1}{3}(I_i - I_{i-1})$$

Which gives us:

$$I = R_{i,2} + c_2 h_i^4 + O(h_i^6)$$

and

$$I = R_{i-1,2} + c_2 h_{i-1}^4 + O(h_{i-1}^6) = I = R_{i-1,2} + c_2 h_i^4 + O(h_i^6)$$

which can be equated and rearranged to give:

$$c_2 h_i^4 = \tfrac{1}{15}(R_{i,2} - R_{i-1,2}) + O(h_i^6)$$

And now we are accurate to fifth order! This method can be continued and generalized to give the estimate:

$$R_{i,m} = R_{i,m-1} + \tfrac{1}{4^m - 1}(R_{i,m-1} - R_{i-1,m-1})$$

which is accurate to order h^{2m+1} with error of order h^{2m+1}. In practice, computationally, we calculate $R_{i,0}$ by:

$$R(i,0) = \tfrac{1}{2}R(i-1,0) + h_i \sum_{k=1}^{2^{i-1}} f(a + (2k-1)h_i) \qquad R(0,0) = h_1(f(a) + f(b))$$

where the data is stored in vectors/arrays, so I've switched to parentheses-notation for a more visual representation. $R(i,m)$ is then calculated as the method above, iteratively. This method was applied to the following integrals:

$$f1 = \int_0^{4\pi} x^2 cos(x)dx = 8\pi \qquad f2 = \int_0^{100\pi} x^2 cos(x)dx = 200\pi \qquad f3 = \int_0^1 x^{1/2}dx = \frac{2}{3}$$

A plot of the log of the error(L1) vs the sampling pts is given above

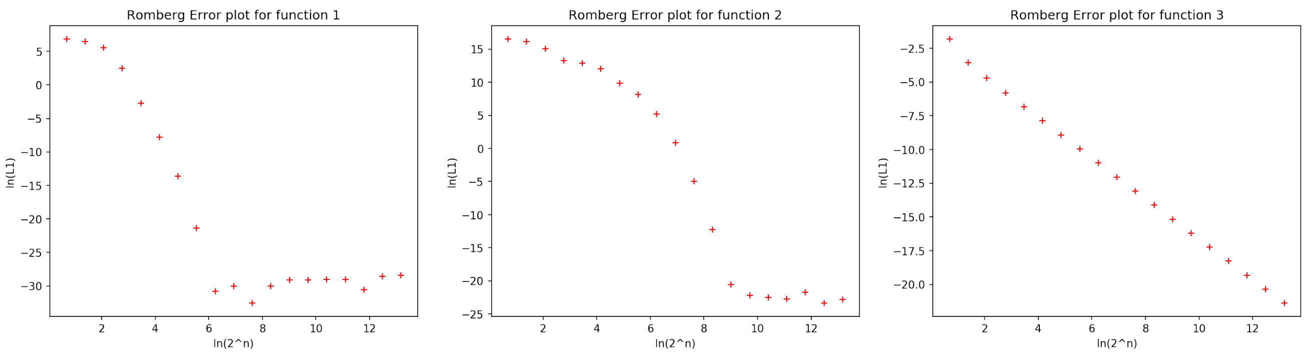


Figure 1: Error in Romberg Integration

Because it is a method of making a series expansion, it will take a few iterations for it to converge well, as can be seen in the plots. For small values of N, it thus does not give a great result for more complicated functions, while for f3, the result is accurate within $\pm$ 1 for even the first estimate. Around N=9, all three integrals have converged to very low error and are getting dominated by rounding error. If assuming a power law $(L_1\ N^{-p})$, the slope (-p) of the straight line fit, as calculated by numpy.polyfit, is -5.059,-1.951, -1.522, for function 1,2 and 3 respectively (for the parts before rounding error dominates). As can be seen, it doesn't seem to follow a power law at all for the first 2 plots, which makes sense given the oscillatory nature of the functions integrated.

2.1.2 Gaussian Quadrature

Gaussian quadrature is a method for calculating integrals with non-uniform sample points. To do this, we need to get the appropriate weights so we can use the following equation:

$$\int_a^b f(x)dx = \sum_{k=1}^{N} w_k f(x_k)$$

To do this, we fit a polynomial through the values $f(x_k)$ as such:

$$\phi_k(x) = \prod_{m=1...N,m\neq k} \frac{x - x_k}{x_k - x_m}$$

for evaluation at the points x_m we thus have $\phi_k(x_m) = \delta_{km}$. We now consider the quantity:

$$\Phi(x) = \sum_{k=1}^{N} f(x_k)\phi_k(x)$$

which when integrated is an approximation of our integral! The weights are thus simply:

$$w_k = \int_a^b \phi_k(x)dx$$

Once we have calculated these weights (done for us by a provided module), we can integrate any function using them. Applying this method to the same functions as mentioned above, gives the following error plots:
As with the Romberg Integration, the first two plots don't converge especially quickly, because

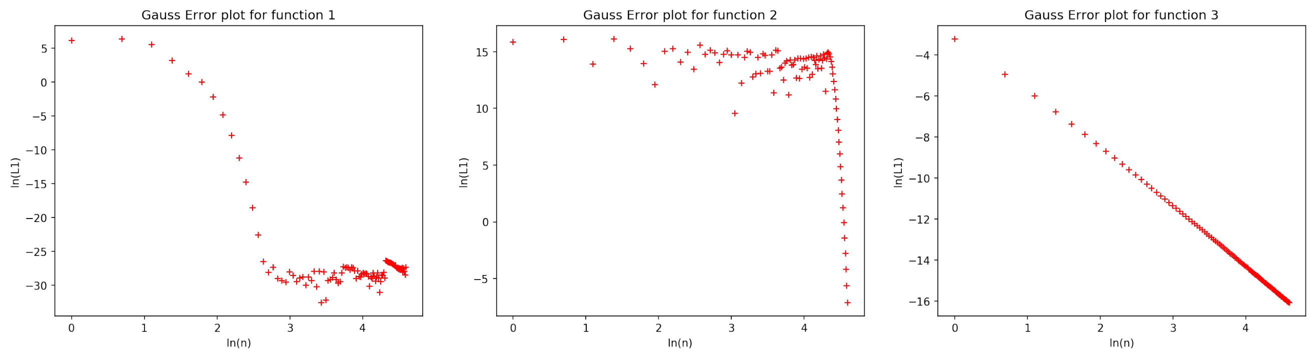


Figure 2: Error in Gaussian Quadrature

the order of the fitted polynomials isn't high enough to capture the oscillatory behaviour of the cosine function. Once we get a good enough polynomial, the approximations quickly converge, especially for f2. However, it takes almost N=100 to get there. For the square root from 0 to 1, it is easier to get a good fit with a low-order polynomial and thus we converge quickly. We see that rounding error starts dominating around 12 for f1, 120 for f2 (not shown in this plot), and never for the square root. The "converge-powers" in a power model would be -3.311,-0.165, and -0.121, respectively. The first two obviously don't follow power laws, because of the oscillatory nature of the integrand, making it take a few iterations to get a good fit.

2.2 Lax-Friedrichs Method

To solve 1D scalar advection equations, the most basic and naive approach is to use a finite-difference scheme, where the easiest way to accomplish this is through a FTCS scheme (Forward in Time, Centered in Space). We wish to approximate solutions to equations of the form:

$$\frac{\partial u}{\partial x} = v\frac{\partial u}{\partial t}$$

for some scalar u and constant v. We denote $u(t^n, x) = u_j^n$ for u at some x and t. Approximating the derivatives as follows:

$$\partial_t u = \frac{u_j^{n+1} - u_j^n}{\Delta t} + O(\Delta t^2) \qquad \partial_x u = \frac{u_{j+1}^n - u_{j-1}^n}{\Delta x} + O(\Delta x^2)$$

we end up with:

$$u_j^{n+1} = u_j^n - v\Delta t\left(\frac{u_{j+1}^n - u_{j-1}^n}{2\Delta x}\right)$$

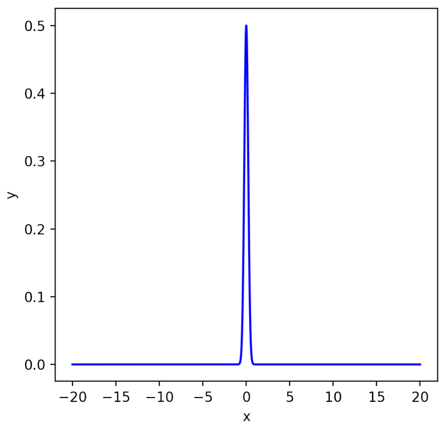
which determines the forward-in-time evolving of u. We apply this scheme to a Gaussian wave of form $u = \frac{1}{2}e^{-x^2/0.1}$. The Gaussian is made narrow to more easily see the unstable behaviour of this scheme. Some plots of the wave at different times are included in the end of this document. They show the Gaussian at times 0,1,2,4,8 and 16. $\Delta t = 0.001$. The dashed, red line denotes the correct behaviour by a propagating Gaussian. As is very clear by the plot, this does not behave like we expect, and we see a growing unstable behaviour. A fix for this unstability is the Lax-Friedrich method, which implements the following change:

$$u_j^n \rightarrow \frac{1}{2}u_{j+1}^n - u_{j-1}^n$$

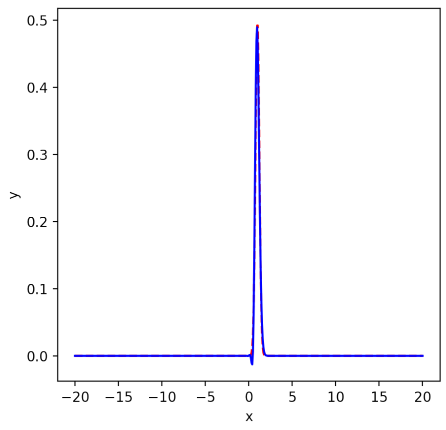
That is, taking the average of the two neighbouring points instead of using the point itself when moving forward in time. Now, the FTCS scheme is stable if the Courant condition is satisfied:

$$\frac{|v|\Delta t}{\Delta x} \leq 1$$

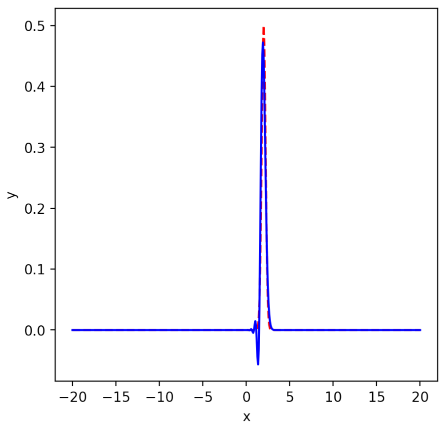
I now tested this for a Gaussian wave, with $\Delta t = 0.01, \Delta x = 0.01$ and $v = 1$, satisfying the condition. I also change the width of the Gaussian, as if it is too narrow, the averaging method will severely dampen the wave itself. The Gaussian is now $u = e^{-x^2/500}$. Several plots for the same time scales are included for this as well, and we can clearly see that no unstabilities arise, however there is slight damping of the wave itself, as is expected when we average over the nearest points: The peak will always get pulled down, however how quickly it happens depends on the number of sample-points.



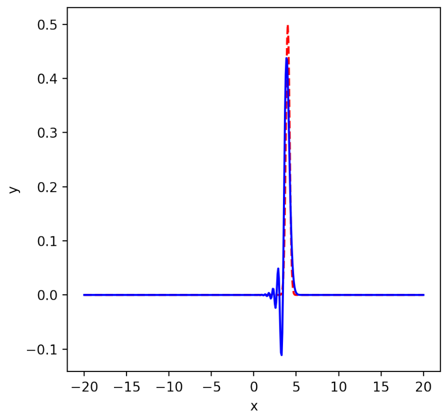
(a) Gaussian at t=0



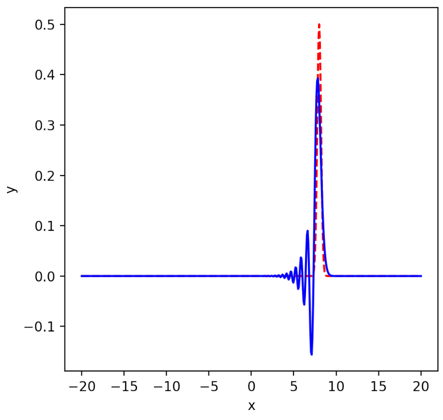
(b) Unstable scheme, t=1



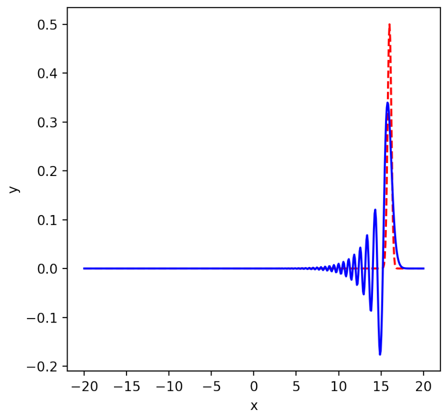
(c) Unstable scheme, t=2



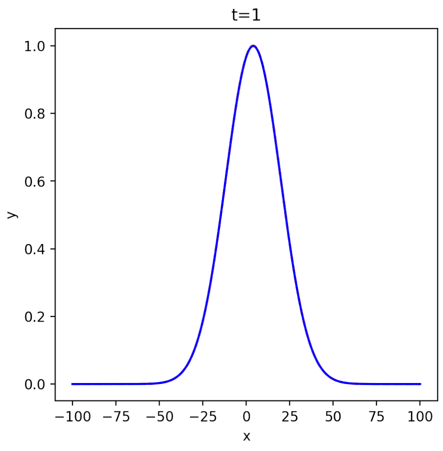
(d) Unstable scheme, t=4



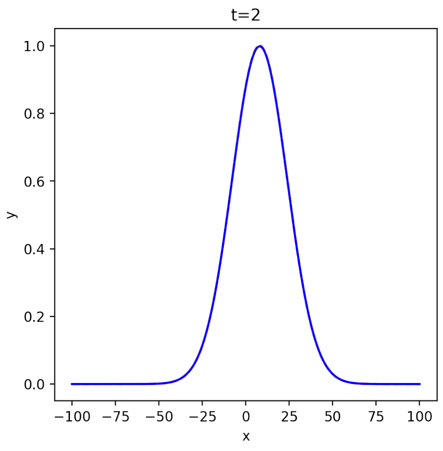
(e) Unstable scheme, t=8



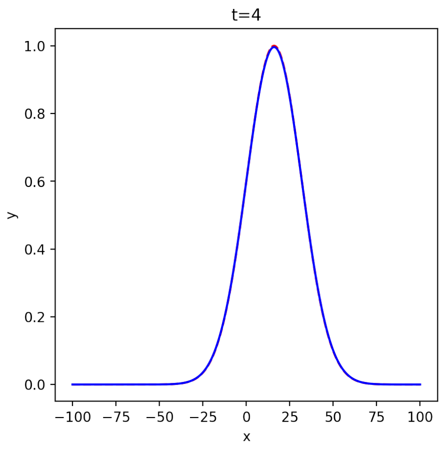
(f) Unstable scheme, t=16



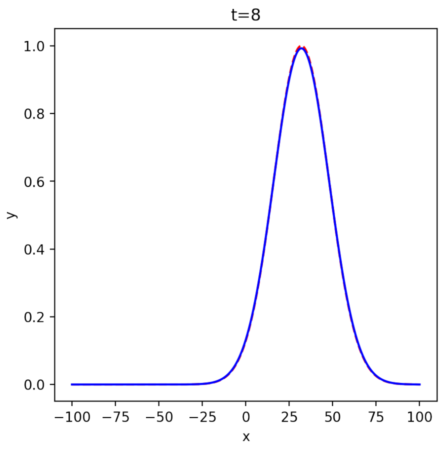
(a) Stable scheme, t=1



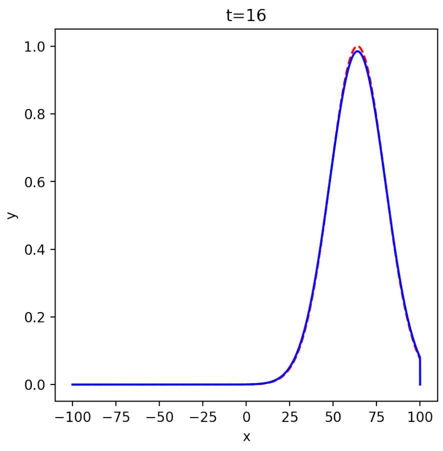
(b) Stable scheme, t=2



(c) Fixed Scheme, t=4



(d) Stable scheme, t=8



(e) Stable scheme, t=16