EIDGENÖSSISCHE TECHNISCHE HOCHSCHULE ZÜRICH

ETH zürich

High-order monotonic numerical diffusion

Ida Olsen
Andrea Leuthard
Manuel Brülisauer

31st August 2022

Abstract

In this project we implement the two-dimensional version of the diffusion equation in orders $n = 2,4,6$ and 8 together with two different filters of the over- and undershoots occuring in the higher order schemes. As filters we use the Zalesak type flux correction as well as a simple monotonic flux correction. For the simple flux limiter we conduct a performance analysis. Unfortunatelly, we were not able to implement the Zalesak flux limiter.

Contents

1 Introduction

If we want to integrate a differential equation with finite differences, we have to choose the order of the integration scheme. The higher the order, the more accurate the solution can be approximated, but with higher order, undesirable ripples occur - especially near steep gradients. These ripples can be removed by applying a filter, which ensures that no additional extrema is created.

In this project, we want to implement a high-order monotonic diffusion in two dimensions. Therefore, we extended the stencil2d program from the High Performance Computing course [1] with the orders $n = 2, 6$ and 8. As a filter we intended to implement the Zalesak-type flux correction, as well as an alternative flux-limiter. Those two filters are described in section 2 and they were implemented as presented by Xue [2].

2 Theory

Numerical diffusion is a technique that can be used to control near two grid interval noise. It is introduced by adding an additional term to the right hand side (RHS) of time dependent equations:

$$\frac{\partial \Phi}{\partial t} = S + (-1)^{n/2+1} \alpha_n \nabla^n \Phi \tag{1}$$

Here Φ is any prognostic variable, S represents other processes and the remaining term on the RHS represents the added diffusion term, where n denotes the order (here $n = 2, 4, 6, 8$) and α_n is called the diffusion coefficient. Higher order diffusion is more selective than lower order formulations [2], i.e. lower order diffusion damps significantly only waves that are longer than two grid intervals. Higher order schemes, however, have a downside: Not only do they produce under- and overshoots, but they become also increasingly unstable. In order for the code to be stable, we must choose the diffusion coefficient α_n or the time step Δt sufficiently small, for example [2]

$$\alpha_n = (\Delta t)^{-1} \, (\frac{\Delta x}{2})^n. \tag{2}$$

The second order diffusion term has a laplacian form and it resembles the heat equation, which describes the physical process of diffusion, where higher values are diffused towards lower values.

The diffusion equation of order n is shown in equation (3). By introducing the diffusive flux F, this equation can be written in two dimensions as represented in equation (4).

$$\frac{\partial \Phi}{\partial t} = (-1)^{\frac{n}{2}+1} \alpha_n \nabla^n \Phi \tag{3}$$

$$\frac{\partial \Phi}{\partial t} = -\nabla \begin{pmatrix} F_x \\ F_y \end{pmatrix} \tag{4}$$

This can be extended to higher orders, where the diffusive flux is given by

$$F = (-1)^{n/2} \, \alpha_n \, \frac{\partial^{n-1} \phi}{\partial^{n-1} x} \tag{5}$$

2.1 Simple flux correction

One of the simplest ways to treat over- and undershoots, created when using higher order schemes, is by using the fact that lower order schemes (up to second order) does not create any new extrema.

By comparing the sign of the higher order diffusive flux to that of the second order, it can be determined where over and undershoots are present, as these will be present exactly where the the fluxes have opposite sign. A simple way to correct these fluxes is by simply setting them to zero, whenever the higher order flux has a different sign to the lower order. This is shown in equation (6).

$$-F_{i+1/2,j+1/2} = -F_{i+1/2,j+1/2} \cdot \max\left[0, sign(-F_{i+1/2,j+1/2}\nabla\Phi)\right] \tag{6}$$

Where $-F_{i+1/2,j+1/2}$ is the higher order flux, $\nabla\Phi$ is the gradient and the sign operator works such that:

$$sign(x) = \begin{cases} -1, & \text{if } -F_{i+1/2,j+1/2}\nabla\Phi < 0 \\ 0, & \text{if } -F_{i+1/2,j+1/2}\nabla\Phi = 0 \\ 1, & \text{if } -F_{i+1/2,j+1/2}\nabla\Phi > 0 \end{cases}$$

This scheme ensures that whenever an non-downgradient flux is detected, it is set to zero, hence ensuring that no extrema not present in the lower order schemes are present in the higher order schemes.

2.2 Zalesak-type flux correction

When solving equation (4) by a finite difference method, we get

$$\Phi^{n+1} = \Phi^n - (A^x_{i+\frac{1}{2},j} - A^x_{i-\frac{1}{2},j}) - (A^y_{i,j+\frac{1}{2}} - A^y_{i,j-\frac{1}{2}}), \tag{7}$$

where $A^{x,y} = \frac{\Delta t}{\Delta x} F_{x,y}$. The idea of Zalesak [3] is to combine a lower order and a higher order diffusion scheme so that one gets the advantage of high accuracy together with the suppression of the over- and undershoots. Hence, the corrective flux $A^{HL} = A^H - A^L$ is introduced as the difference between a high-order diffusive flux A^H and a low-order diffusive flux A^L. Here, we choose the lower order flux to be of second order, because it is the highest order in the diffusion equation which is still monotonic. The corresponding time step is represented in equations (8).

$$\begin{aligned} \phi_i^{n+1} &= \phi_i^* - (C_{i+\frac{1}{2}} A^{HL,x}_{i+\frac{1}{2},j} - C_{i-\frac{1}{2},j} A^{HL,x}_{i-\frac{1}{2},j}) - (C_{i,j+\frac{1}{2}} A^{HL_y}_{i,j+\frac{1}{2}} - C_{i,j-\frac{1}{2}} A^{HL,y}_{i,j-\frac{1}{2}}) \\ \phi^*_{i,j} &= \phi_{i,j} - (A^{L,x}_{i+\frac{1}{2},j} - A^{L,x}_{i-\frac{1}{2},j}) - (A^{L,y}_{i,j+\frac{1}{2}} - A^{L,y}_{i,j-\frac{1}{2}}) \end{aligned} \tag{8}$$

The flux correction coefficient C is typically $0 \leq C \leq 1$ and it has to be computed for each time iteration and for each grid point. The algorithm for computing C can be found in [2]. For $C = 0$, the Zalesak filter reduces to the lower order filter, whereas $C = 1$ makes the Zalesak filter purely higher order.

3 Results

The test case used in this analysis is a 3D stencil, where diffusion is applied in the x and y directions creating a 2D diffusion field. The stencil is initialized to be symmetrical both along the x and y axis and with gridsizes $n_x = n_y = 128$, $n_z = 64$ and grid spacing $\Delta x = \Delta y = \Delta z = \Delta t = 1$, whereas throughout the project we do $t_n = 1024$ time steps.

We here present results obtained for the 2nd, 4th, 6th and 8th order diffusion schemes, along with a comparison of the output obtained for the higher order schemes with and without a flux limiter. We observed that formula 2 did not produce a stable code in every case. In particular, the 6th and 8th order were not stable. Consequently, we chose a smaller diffusion coefficient (powers of 1/2) until the code was stable. In this way we arrived at

$$\alpha_n = \frac{1}{32}, \frac{1}{32}, \frac{1}{256}, \frac{1}{2048} \text{ for } n = 2, 4, 6, 8, \tag{9}$$

whereas the 4th order diffusion coefficient is the one that we used already in the course [1].

In Figure 1 we plot the diffusion operator for different orders and figure 2 show the diffused fields before and after applying the simple filter described in section 2.1. In figure 3 a 2D representation of the stencil is shown before and after applying the filter.

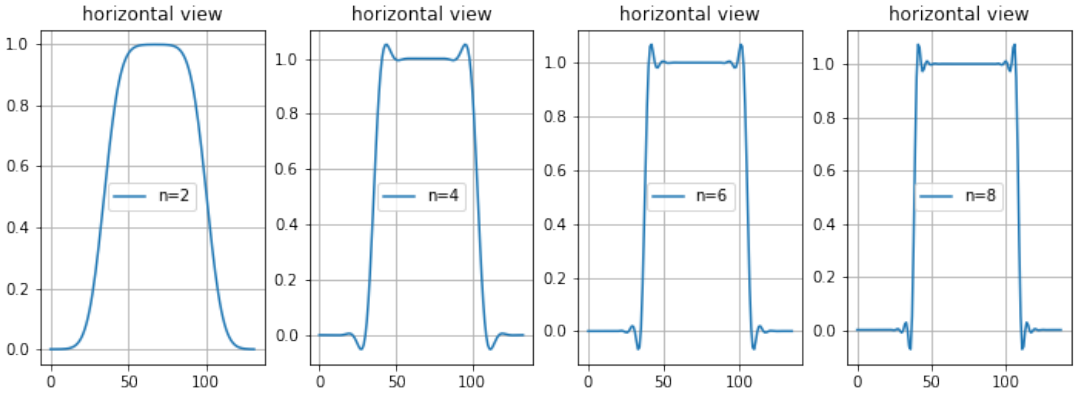

Figure 1 – Starting with the 4th order, the diffusion operator produces under- and overshoots, as anticipated.

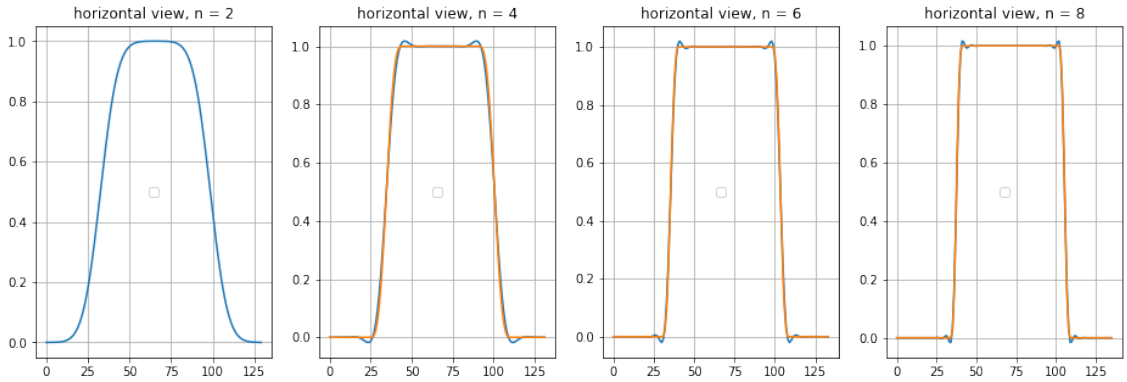

Figure 2 – We observe that the over and under shoots seen in figure 1 have successfully been removed after applying the filter

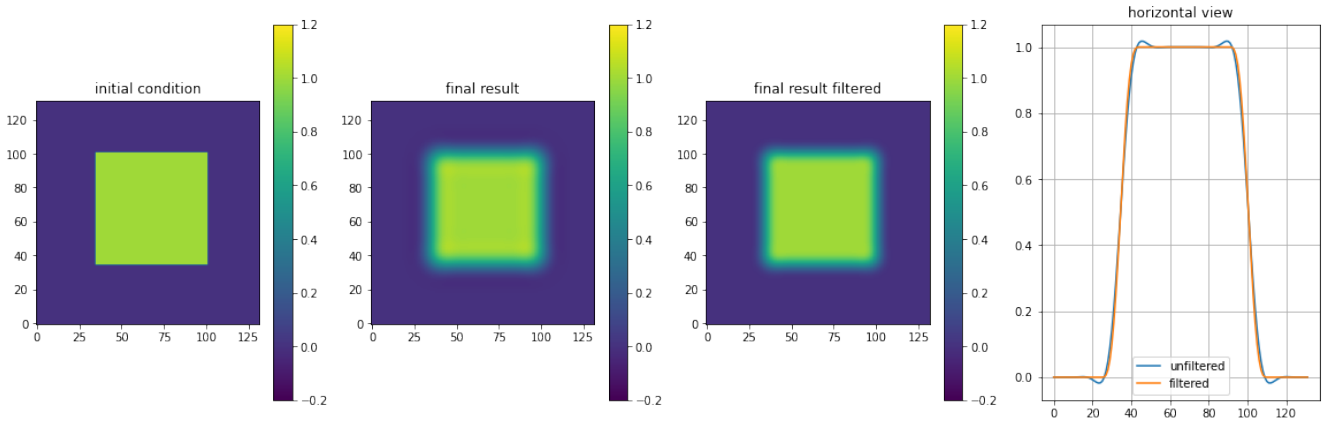

Figure 3 – 2D stencil shown at level z = 32 for the 4th order diffusion scheme

Figure 1 show that the diffusion schemes work as intended. We observe a less selective diffusion by the lower order schemes, with an output that is highly smoothed out compared to the initial. At the same time we observe under and over shoots for the three higher order schemes.

Figures 2 and 3 furthermore show that the simple flux limiter works as intended, as over and undershoots are removed from the filtered solutions, while keeping the shape of the respective diffusion operator.

3.1 Performance measures

Table 1 shows that with higher order diffusion schemes, a higher computation time also arises. This is expected as more computations take place when using a higher order scheme. Similarly we observe that the flux limiter comes with a computation time cost, which appears to be very similar to that of increasing the diffusion order with 2.

Contrary to this, it is observed that the computation time per gridpoint, remains somewhat stable when increasing the stencil size, as illustrated in Figure 4.

This indicates that the code is arranged in a way which optimizes the cache usage. If this was not the case it would be expected to see a significant increase in the runtime per gridpoint, as a result of increasing the working set size.

	4th order	6th order	8th order
Runtime no filter (s)	4.75	6.98	9.53
Runtime filter (s)	6.81	9.26	11.52

Table 1 – Total runtime (s) for diffusion operator with and without filter

The almost constant runtime per gridpoint for different working set sizes is possibly because the amount of memory accesses from main memory does not change significantly when increasing the working set size.

In this project we have used a code which is inlined and has k-blocking, as observed by the fact that only one k-loop is used for each iteration.

This signifies that the laplacian operators used for the difussion process are kept one or two dimensional, hence having sizes small enough to fit into cache memory for all gridsizes checked here.

We would expect to see a significant decrease in performance if we had managed to successfully implement the zalesak filter, as this is implemented with several loops including several k-loops. Going over a full cube is more than we can fit in the cache and hence, this would require more reads from main memory, which is the most expensive in terms of runtime.

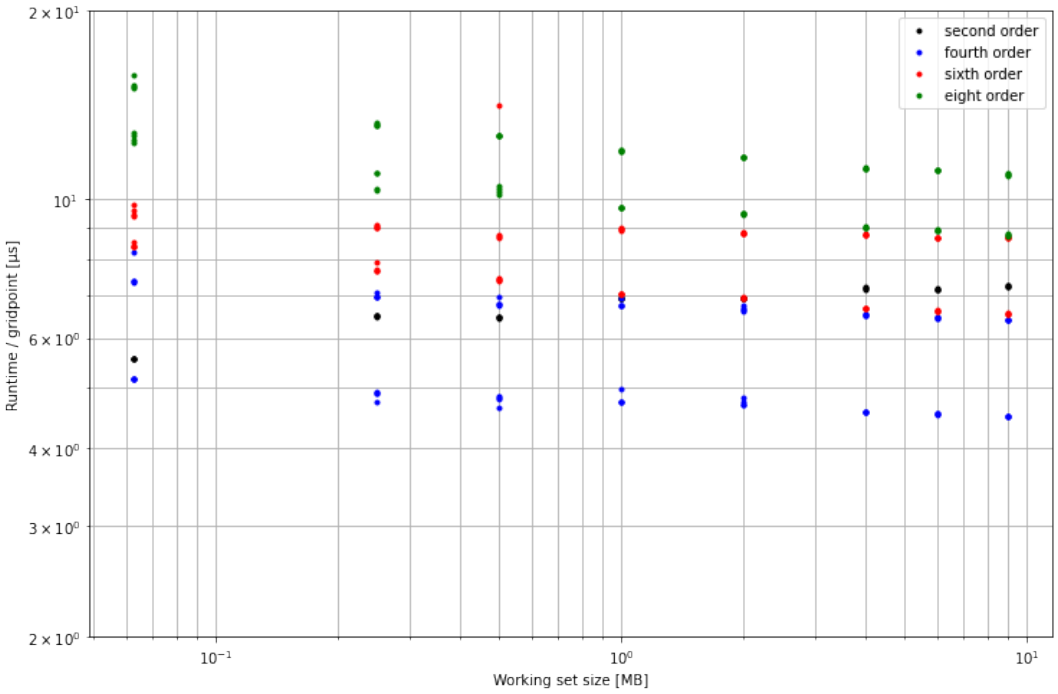

Figure 4 – Performance given by runtime/gridpoint for different orders. Upper lines indicate performance with filter and lower lines the performance without a filter. For each setup four values are calculated for the filtered and unfiltered solutions.

```
Table 2:  Profile by Group, Function, and Line

  Samp% |  Samp | Imb. |  Imb. | Group
        |       | Samp | Samp% |  Function=[MAX10]
        |       |      |       |   Source
        |       |      |       |    Line
        |       |      |       |     Thread=HIDE

 100.0% | 377.0 |   -- |    -- | Total
|-----------------------------------------------------------------------------
| 100.0% | 377.0 |   -- |    -- | USER
||----------------------------------------------------------------------------
|| 100.0% | 377.0 |   -- |    -- | main
|||---------------------------------------------------------------------------
3||  59.2% | 223.0 |   -- |    -- | /users/class170/project/stencil2d-pro.cpp
||||------------------------------------------------------------------------
4|||   4.2% |  16.0 |   -- |    -- | line.67
4|||   9.3% |  35.0 |   -- |    -- | line.69
4|||  28.6% | 108.0 |   -- |    -- | line.73
4|||  16.7% |  63.0 |   -- |    -- | line.79
||||========================================================================
3||  37.1% | 140.0 |   -- |    -- | /users/class170/project/utils.h
4||        |       |      |       |  line.13
3||   3.7% |  14.0 |   -- |    -- | include/g++/bits/stl_vector.h
4||        |       |      |       |  line.1043
|=============================================================================
```

Figure 5 – The CrayPat-lite Performance Statistics give us an idea on how the resources of a supercomputer are used to run our diffusion operator.

Performance statistics, provided by CrayPat-lite, prove to be invaluable if we want to understand how to optimize our code. In Figure (5) these statistics are listed for the $n = 2$ diffusion operator. Here, 59.2 % of the resources are used by the `stencil2d.cpp` code and 37.1 % by `utils.h`, adding almost to 100 %. Lines 67 to 69 are where the laplacian is calculated with the stencil code. Apparently however, lines 73 to 79 have a greater impact: This is where the actual timestep is performed and the new field value is written into the memory. Even more significantly are the 37.1 % which are spent to access the memory and return the field values from the memory unit onto the CPU. This leads us to think that the secret of improving performance lies in optimizing the access of memory. This can be achieved by parallelizing the stencil computation by dividing the grid into subsets so small that they can be stored in a higher cache level, making accessing the data much faster. In our case the total size of the field is (a `double` variable has a size of 8 `bytes` [4]).

$$8\,\texttt{bytes} \times 128 \times 128 \times 64 = 8388608 \approx 8\,\texttt{MB}, \tag{10}$$

whereas L3 Cache only spans 2.5 `MB` ([1]).

For higher orders, we create temporary arrays "`tmpField`" where we store the values of the laplacian. These temporary fields have a size of $128 \times 128 \times 1 \times 8\,\texttt{bytes} \approx 0.1\,\texttt{MB}$ and therefore are smaller than L2 cache.

4 Discussion, conclusions and further work

In this project we have successfully extended the stencil2d program from the High Performance Computing course with the orders n = 2, 6 and 8. Furthermore, we have implemented a simple flux limiter, which assures that over- and undershoots present in higher order schemes n = 4, 6 and 8 are treated, by setting the flux to zero, whenever it is not downgradient. We also attempted to implement the Zalesak flux limiter, but this was not managed. It is expected that the error lies in the computational implementation of equation (5), but this require further work to determine. As of now the ground work for implementing the filter has been made and it is a matter of understanding where the theory does not match the computations.

A crucial observation is that formula (2) for the diffusion coefficient fails to work for higher orders. We had to manually set the diffusion coefficient lower than proposed by the formula. In fact, the reason it doesn't work here, is, that the parameter depends on the grid spacing and the time step, but we work only with constant grid sizes and time steps $\Delta x = \Delta t = 1$. In a future project therefore, we could allow varying grid spacing as well as smaller time steps, which we could employ as a means to make the code more stable.

The simple flux limiter as well as the Zalesak flux limiter aim to filter out the unphysical behavior of the higher order schemes. Nevertheless, the approach is quite different. While the simple flux limiter seems at first sight quite brutal, it gives really good results in limiting. Unfortunatelly, we were not able to compare the results of those two filters since we did not manage to implement the Zalesak filter correctly. But accordingly to Xue [2], the simple flux limiter gives at least the same accuracy as the Zalesak filter. One big advantage of this simple flux limiter is that the computational costs are much smaller than with the implementation of the Zalesak type flux limiter. Apart from that, also the logistical effort is less: The Zalesak filter contains a lot of different arrays that need to be initialized, written to, accessed and stored, whereas in the simple filter only the array for the diffusive flux F needs to be created. In a future work, we should compare the runtime of those two filters - after the right implementation of the Zalesak filter.

References

[1] Oliver Fuhrer. High performance computing for weather and climate, 2022.

[2] Ming Xue. High-order monotonic numerical diffusion and smoothing. *Monthly Weather Review*, 128(8):2853 – 2864, 2000.

[3] Steven T Zalesak. Fully multidimensional flux-corrected transport algorithms for fluids. *Journal of Computational Physics*, 31(3):335–362, 1979.

[4] Microsoft c++ language reference. `https://docs.microsoft.com/en-us/cpp/cpp/data-type-ranges?view=msvc-170`. Accessed: 30/08/2022.