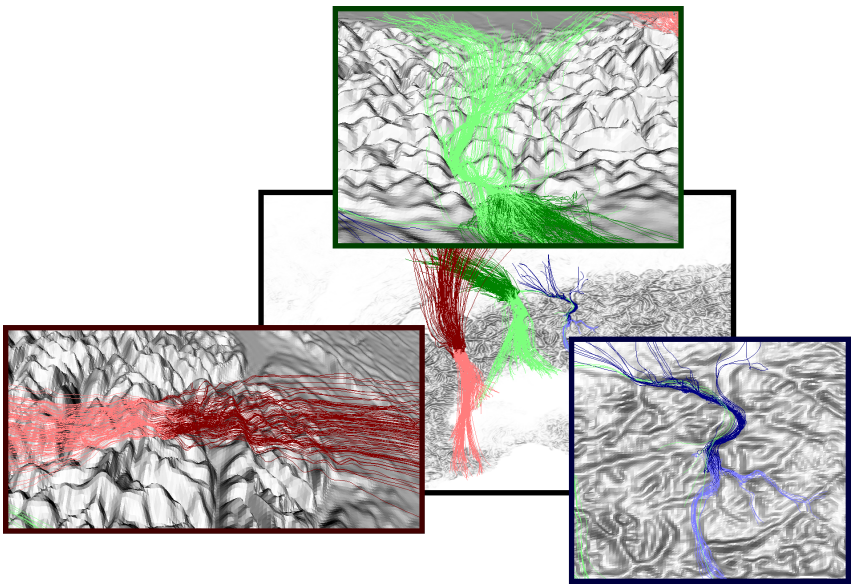
Visualization of Thermally-driven Flow over the Alpine Region



Linus Wigger

Semester Thesis
July 2020

Prof. Dr. Markus Gross, Dr. Tobias Günther

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

cgl
computer graphics laboratory

Abstract

The goal of this thesis was to create a tool for computing and displaying wind trajectories. We would then use it to visualize several test cases from meteorological data. Said data deals with Föhn, a phenomenon that is associated with warm and dry winds in the northern parts of the alps, also bad weather on the south side.

While the focus lay on visualization and analysis of results initially, it later shifted towards reproducing trajectories from a similar tool and improving on that. That worked in the sense that we can obtain trajectories that are practically the same. We can also do it faster and with fewer errors. The visualization part ended up as more of a debugging tool than a user-friendly application.

Zusammenfassung

Das Ziel dieser Arbeit war es, ein Tool zur Berechnung und Darstellung von Windtrajektorien zu erstellen. Es würde dann zum Visualisieren etlicher Testfälle aus meteorologischen Daten benutzt werden. Die Daten beschäftigen sich mit dem Föhn, einem Phänomen das mit warmen und trockenen Winden in den nördlichen Teilen der Alpen verbunden wird. Ausserdem schlechtes Wetter auf der Südseite.

Obwohl der Schwerpunkt ursprünglich auf dem Visualisieren und Analysieren von Resultaten lag, änderte er sich später und es wurde mehr darauf geachtet, dass die Trajektorien aus einem ähnlichen Programm reproduziert und verbessert werden können. Dies funktionierte in der Hinsicht, dass wir praktische identische Trajektorien produzieren können. Dabei können wir es auch schneller und mit weniger Fehlern. Der Visualisierungsteil wurde daher eher zu einem Debuggingwerkzeug als zu einer benutzerfreundlichen Anwendung.

Contents

List of Figures

List of Tables

1

Introduction

A common problem in meteorology is to find wind trajectories with certain properties, e.g. passing through a certain region, carrying particularly warm or humid air, etc. LAGRANTO ([SW15]) is an existing Fortran-based program for computing trajectories from wind velocity fields. LAGRANTO takes as its input a set of initial points (given by their positions and starting time), settings for the integration, and appropriate data files. Given those, it computes trajectories starting from the initial points and writes them down in an output file.

We aim to create a similar program in C++. While reconstructing LAGRANTO is our base goal, we would like to get better performance and/or results as well.

There are LAGRANTO variants for different types of input data. We work with the COSMO version because all our input data is in the COSMO model. [SDS+03] and [BSF+11]

For the visualization we use VTK, the Visualization Toolkit ([SML06]). It contains many features, most of them unused in our case. Mostly we just draw the trajectories as 3D curves and add the underlying geography for context.

2

Background

2.1 Data

We work on a set of NetCDF files containing assorted meteorological data in the COSMO model. Most of the files contain data at a certain point in time and have names like "lfff00000000.nc". The number in the filename corresponds to the time past the reference date using the format DDHHMMSS, so for example "lfff00015000.nc" would contain the data at one hour and fifty minutes. In addition, there is a file "lfff00000000c.nc" (note the c) which holds constant variables like the height of the surface. The reference time for our data is 2016, Nov, 21, $00:00$ and the files go from "lfff00000000.nc" (reference date) to "lfff03225000.nc" (3 days 22 hours 50 minutes later) with 10 minutes between files.

Most of the important variables are stored as three-dimensional arrays. The three dimensions are called $rlon$, $rlat$ and $level$. $rlon$ and $rlat$ are coordinates in a rotated geographical coordinate system. The $level$s correspond to the vertical position of a point, but it is not a simple linear transformation. Instead, the constants file holds the necessary information to convert $level$s to actual height. Further details can be found in section 2.3. Unless noted otherwise, the grid size for our data is always $1158 \times 774 \times 80$.

Table 2.1 gives an overview of the most interesting variables. The three variables UVW define the velocity field: U is the eastward (in the rotated system) component of the wind, V the northward component, and W the upward component. All three have the same units (m/s) and similar but not equal grids. The grids are staggered: All vertices in one grid are translated by half a cell size in one direction. Section 2.2 describes how the staggered grids are handled.

HHL maps the $level$ of a grid point to a physical height. Like W, it is staggered in the vertical direction and needs to be destaggered before it can be used with most other variables. HHL is important because the particle positions have a real height in meters as their third component and there needs to be a way to find grid coordinates from the particle position.

Name	Description	Dimensions	Time-invariant	Staggering	Unit
U	$rlon$ component of velocity	3	no	$rlon$	m/s
V	$rlat$ component of velocity	3	no	$rlat$	m/s
W	vertical component of velocity	3	no	$level$	m/s
HHL	$level$-to-height map	3	yes	$level$	m
$HSURF$	height of surface	2	yes	none	m
P	pressure	3	no	none	Pa
T	temperature	3	no	none	K
$RELHUM$	relative humidity	3	no	none	%

Table 2.1: *Important variables*

$HSURF$ contains the height of the surface for given ($rlon$, $rlat$) coordinates. It is mainly used to prevent particles from leaving the domain through the ground.

The pressure P, temperature T and relative humidity $RELHUM$ are not relevant for the tracing, but they work well as examples of the kind of data one may wish to track along the trajectories.

2.2 Destaggering

U, V, W and HHL are given in staggered grids, recognizable by using $srlon$, $srlat$ and $level1$ for certain axes. The staggered grid coordinates lie halfway between the unstaggered grid points. Destaggering is done by averaging the values at two vertices that are adjacent in the staggering direction, then storing the result at the grid position between those vertices. Figure 2.1 shows how staggered ($srlon$, $rlat$) and ($rlon$, $srlat$) grids are converted to ($rlon$, $rlat$). The image also shows that the destaggered version of the grid has one row/column less than the staggered original.

The dimensions of the UVW grid are effectively $1157 \times 773 \times 80$. Compared to the default size, this is one element less in $rlon$ and $rlat$. The number of $level$s remains at 80 because the staggered axis $level1$ has size 81.

2.3 Conversion between coordinate systems

The velocities U, V, W, as well as other variables like temperature, are defined on a regular grid with axes corresponding to ($rlon$, $rlat$, $level$).

$rlon$ and $rlat$ can be converted into lon and lat given the (global) coordinates of the rotated north pole (λ_{pole}, ϕ_{pole}). In our data, ϕ_{pole} is always $43°$ and λ_{pole} is $-170°$ Converting coordi-

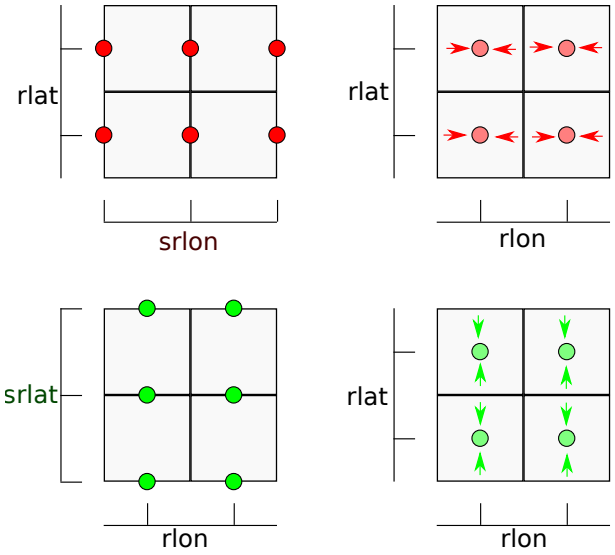


Figure 2.1: *Left: Points in staggered grids; Right: Destaggering by averaging two staggered points*

nates (λ_r, ϕ_r) in the rotated system to the global coordinates (λ_g, ϕ_g) is done as follows:

$$\phi_g = \sin^{-1}(\cos\phi_{pole} \cdot \cos\phi_r \cdot \cos\lambda_r + \sin\phi_{pole} \cdot \sin\phi_r); \tag{2.1}$$

$$c_1 = \sin\phi_{pole} \cdot \cos\lambda_r \cdot \cos\phi_r + \cos\phi_{pole} \cdot \sin\phi_r \tag{2.2}$$

$$c_2 = \sin\lambda_r \cdot \cos\phi_r \tag{2.3}$$

$$zarg1 = \sin\lambda_{pole} \cdot c_1 - \cos\lambda_{pole} \cdot c_2 \tag{2.4}$$

$$zarg2 = \cos\lambda_{pole} \cdot c_1 + \cos\lambda_{pole} \cdot c_2 \tag{2.5}$$

$$\lambda_g = atan2(zarg1, zarg2) \tag{2.6}$$

The vertical coordinates z are given in meters above sea level and need to be mapped to grid levels. To that purpose, we have the time-invariant scalar field HHL which maps (staggered) levels at specific grid points to their height.

The fact that the values are stored in a regular grid that corresponds to an irregular real shape means that one needs to be careful when interpolating values given at coordinates between grid points. Two possible methods are discussed in the following chapter in section 3.2.

3

Method

3.1 Numeric integration

Solving differential equations of all types is a topic for itself. In this section we limit ourselves to describing the methods we use to find the next point of a trajectory given the time-dependent velocity field $UVW(p,t)$, the starting position p_{t_0} and a timestep of size h.

LAGRANTO uses an iterative variant of Euler's method. The next point $p_{t_0}+h$ is computed using the average of the velocities at the original point p_{t_0} and the current guess for $p_{t_0}+h$. This method is related to the explicit trapezoidal rule: In the time-invariant case, stopping at q_2 is equivalent to using the explicit trapezoidal rule.

Iterative Euler

$$v_0 = UVW(p_{t_0}, t_0) \tag{3.1}$$

$$v_1 = UVW(p_{t_0}, t_0 + h) \tag{3.2}$$

$$q_1 = p_{t_0} + h\frac{v_0 + v1}{2} \tag{3.3}$$

$$v_2 = UVW(q_1, t_0 + h) \tag{3.4}$$

$$q_2 = p_{t_0} + h\frac{v_0 + v2}{2} \tag{3.5}$$

$$v_3 = UVW(q_2, t_0 + h) \tag{3.6}$$

$$p_{t_0+h} = p_{t_0} + h\frac{v_0 + v3}{2} \tag{3.7}$$

We preferred to use the classical Runge-Kutta integration scheme. It uses four samples of UVW per iteration like the iterative Euler method, but according to TODO cite that page the iterative

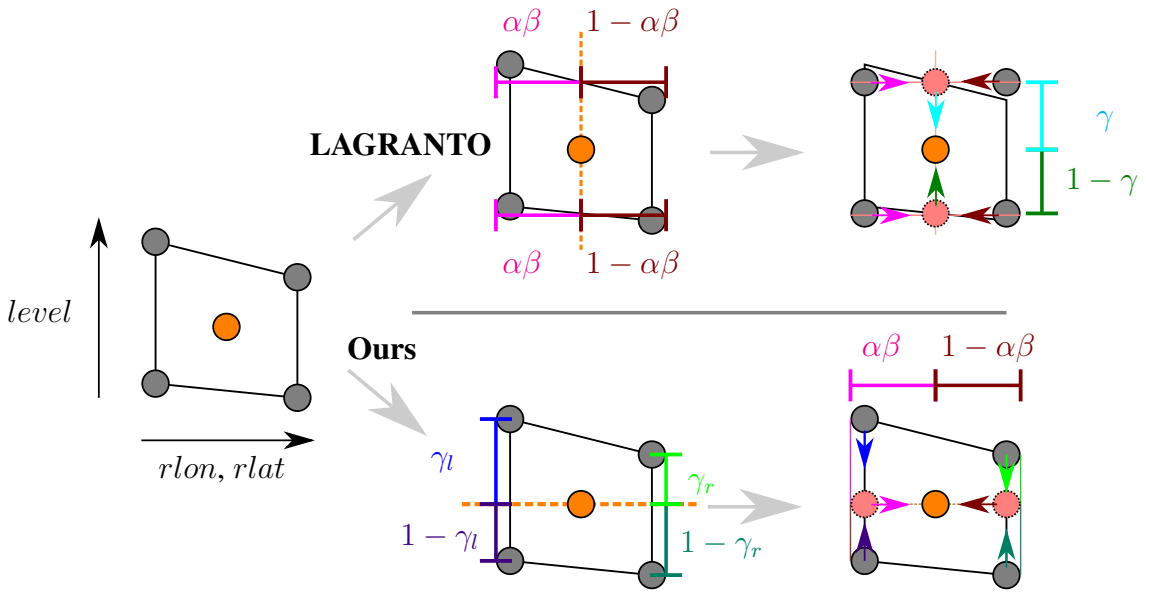


Figure 3.1: *Interpolation procedure for sampling at the orange point: Order and weights depend on the method used*

Euler method has an error $O(h^3)$ per step whereas the Runge-Kutta method has the lower error $O(h^5)$ (see for example [PTVF92]).

Classical Runge-Kutta

$$k_1 = UVW(p_{t_0}, t_0) \tag{3.8}$$

$$k_2 = UVW(p_{t_0} + k_1\frac{h}{2}, t_0 + \frac{h}{2}) \tag{3.9}$$

$$k_3 = UVW(p_{t_0} + k_2\frac{h}{2}, t_0 + \frac{h}{2}) \tag{3.10}$$

$$k_4 = UVW(p_{t_0} + k_3 h, t_0 + h) \tag{3.11}$$

$$p_{t_0+h} = p_{t_0} + (k_1 + 2k_2 + 2k_3 + k_4)\frac{h}{6} \tag{3.12}$$

3.2 Sampling

Sampling the velocity field UVW at a certain position (x,y,z) and time t is a common operation during particle tracing. Because UVW is defined in m/s on a ($rlon$, $rlat$, $level$) grid and the position is given in (°, °, m) some conversions are necessary.

3.2.1 Using local level heights

Mapping x and y to positions in the ($rlon$,$rlat$) grid is done using the fact that the grid is rectangular and regular: Assuming (λ_0, ϕ_0) are the ($rlon$,$rlat$) coordinates of the grid point (0,0) and the distance to the next vertex is Δ_λ (in $rlon$ direction) or Δ_ϕ (in $rlat$ direction), the

grid coordinates are obtained from the real coordinates (x,y) as ($\frac{x-\lambda_0}{\Delta_\lambda}$, $\frac{y-\phi_0}{\Delta_\phi}$). By rounding those grid coordinates up or down we get the coordinates of the nearest grid points.

The upper part of Figure 3.1 shows how LAGRANTO interpolates between levels: In a first step, two level heights for the upper and lower level are constructed (shown as split pink and dark red lines). This requires a binary search to locate two levels for the z-coordinate of the sampling point. LAGRANTO essentially performs trilinear interpolation in a box-shaped cell whose exact position and height depends depends on (x,y). Notice how in the third step the corner points have been moved slightly up or down: The differing real heights of the grid points only matter when determining the local level heights. For the final interpolation, all four corner points on one level are considered to be at the same height.

There are three interpolation weights (α, β, γ) for the axes ($rlon$,$rlat$,$level$). They are computed as ($\frac{x-x_0}{x_1-x_0}$,$\frac{y-y_0}{y_1-y_0}$,$\frac{z-z_0}{z_1-z_0}$), where x_0 and y_0 are the coordinates of the western and southern grid points and z_0 the (interpolated) height of the lower level. Accordingly, (x_1,y_1,z_1) is the position of the upper northeastern corner.

3.2.2 Using adjacent level heights

The grid coordinates of (x,y) and the bilinear interpolation weights α and β along the $rlon$ and $rlat$ axes are computed the same way as in the previous subsection.

The lower part of figure 3.1 shows how we compute the interpolated value at the sample point. On each of the four (two in the picture) columns, we compute interpolation weights $\gamma_i = \frac{z-z_{0i}}{z_{1i}-z_{0i}}$ after finding lower and upper heights z_{0i} and z_{1i} with a binary search on column i in HHL. The last step is bilinearly interpolating between those four values. As mentioned, the weights α and β for the horizontal interpolation are the same that LAGRANTO uses. For the vertical interpolation, LAGRANTO uses only one set of weights (γ and $1-\gamma$). Our version has different weights on each column (the pairs for γ_l and γ_r are visible in the picture), making the sampling process slightly more complicated and hopefully accurate.

3.3 Implementation

The tracing process starts by asking the user for initial points, start and end time, size of the timestep, and additional settings like which variables to track, what type of integrator to use, plus a few other options that matter for debugging and comparing to LAGRANTO (mostly concerning how UVW is sampled). After allocating space for the output data, the UVW fields are extracted from the first three appropriate files. As the simulation runs, the oldest field is regularly replaced by new UVW from the next file in line, minimizing the memory needed at runtime. At each step, all trajectories have to be advanced by h. Those that have left the domain are kept at their last positions while the others get positions for the next timestep based on the velocity at their current position.

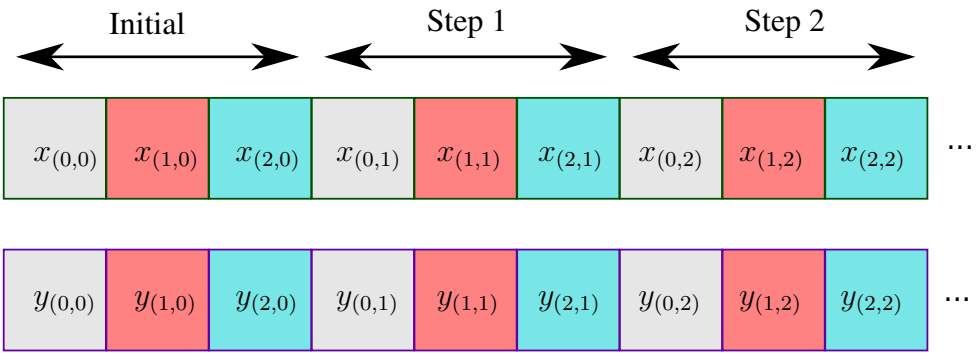


Figure 3.2: *Structure of the output data: $x_{(i,j)}$ is the value of variable x on trajectory i after the jth timestep*

3.3.1 Tracing output

The results from the particle tracing are written into a NetCDF file which contains an array for each variable. Time, coordinates in both (lon,lat) and ($rlon$,$rlat$), and height are always stored. Other variables like temperature or pressure need to be included in the initial input.

The number of elements per array is $N_{tra} \cdot (N_{steps} + 1)$, where the number of trajectories is N_{tra} and N_{steps} is the number of integration steps. The arrays are ordered according to timestep first and trajectory second. Figure 3.2 shows an example with 2 arrays (x and y), 3 trajectories (gray, red, blue), and 5 timesteps (plus the initial state), for a total of 18 elements per array.

This format matches the output file of LAGRANTO, allowing us to compare the results directly. It should be noted that while LAGRANTO uses ($rlon$,$rlat$) for the computations, the output only contains coordinates in (lon, lat) by default.

3.4 Analysis

3.4.1 Qualitative

We visualize the trajectories using VTK ([SML06]). The trajectories are loaded from an output file and drawn in 3D. The user can move the camera to get a better view. A surface obtained from $HSURF$ is also displayed to give a context beyond just the trajectory shape.

$rlon$ and $rlat$ or lon and lat (depending on the settings) correspond to the x and y axes of the renderer. For comparing the results to those from LAGRANTO, the global coordinates (lon,lat) are used because LAGRANTO includes only those in its output. The coordinates on the vertical axis z are rescaled by an arbitrary factor of $5 \cdot 10^{-5}$. While this rescaling does not lead to exact proportions (one unit in horizontal direction does not correspond to the same distance as one unit in vertical direction), it helps make the shapes recognizable. The rescaling is necessary because the x and y values are in degrees and the z values are in meters.

3.4.2 Quantitative

We compare different integrators by giving them the same input and measuring the average distance between their outputs over time. We split the distance into a horizontal and vertical component because the units are different and the total distance would be dominated by the much more chaotic vertical part otherwise.

We use the output from LAGRANTO as a reference and look how the difference to our method evolves over time.

Regarding the performance, we run simulations with the same integrator settings and initial points using both LAGRANTO and our method. We measure the time needed for several differently-sized sets of initial points.

4

Results

4.1 A look at trajectories

Figures 4.1 and 4.2 show examples of trajectories we computed and visualized. In the top left part, each trajectory has a constant color which depends on its index: Starting at blue for the first trajectory and ending at purple. Figure 4.2 includes another color scheme (black-orange) for the trajectories that were computed with a negative timestep. The upper right part of is colored according to temperature: Blue is cold, white is $0°$C, red is warm. The lower two sections visualize pressure (red at high values, bright at low ones) and relative humidity (yellow at 0% and blue at 100%).

Figure 4.1 shows a set of trajectories starting on the south side of the alps. The particles start at $13:00$ of the second day (22.Nov2016) at a latitude of $45.2°N$ and are traced for 5 hours as they move north. The initial points are spread across $7.5 - 10.5°E$ in lon-direction and $2500 - 3000m$ in height. The three variables temperature, pressure and humidity appear to be strongly correlated: The colors have similar patterns, starting cold/high/humid and becoming warmer/lower/drier after passing the mountains.

Figure 4.2 shows several trajectories passing over Chur (around $9.53°E\ 46.85°N$). The starting time for the simulation is midnight between the $22.$ and $23.$ November. The integration time is 3 hours in both directions (so from $21:00$ to $3:00$). Temperature and pressure behave as expected, becoming lower at higher altitudes. The relative humidity seems to have two wet and dry regions each.

Figure 4.3 shows a zoomed-in view where one can see how trajectories make wave shapes over the mountains. Less visible are the jumps that happen when the trajectory goes trough the ground. LAGRANTO has a similar behaviour and in both cases it can be disabled with an optional flag (in which case trajectories that leave the domain end there).

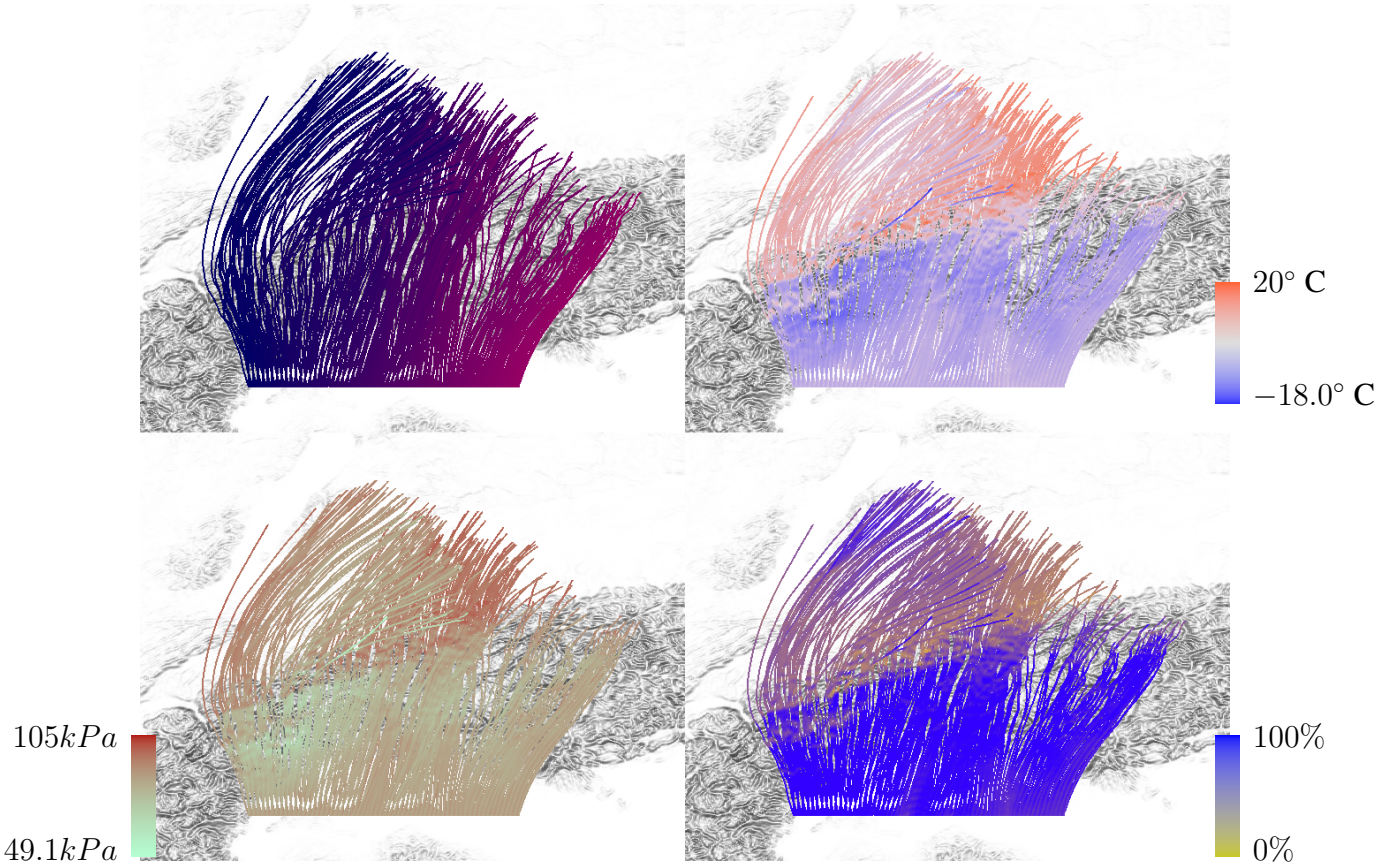


Figure 4.1: *Color according to: Trajectory index(top left), temperature(top right), pressure(bottom left), relative humidity(bottom right)*

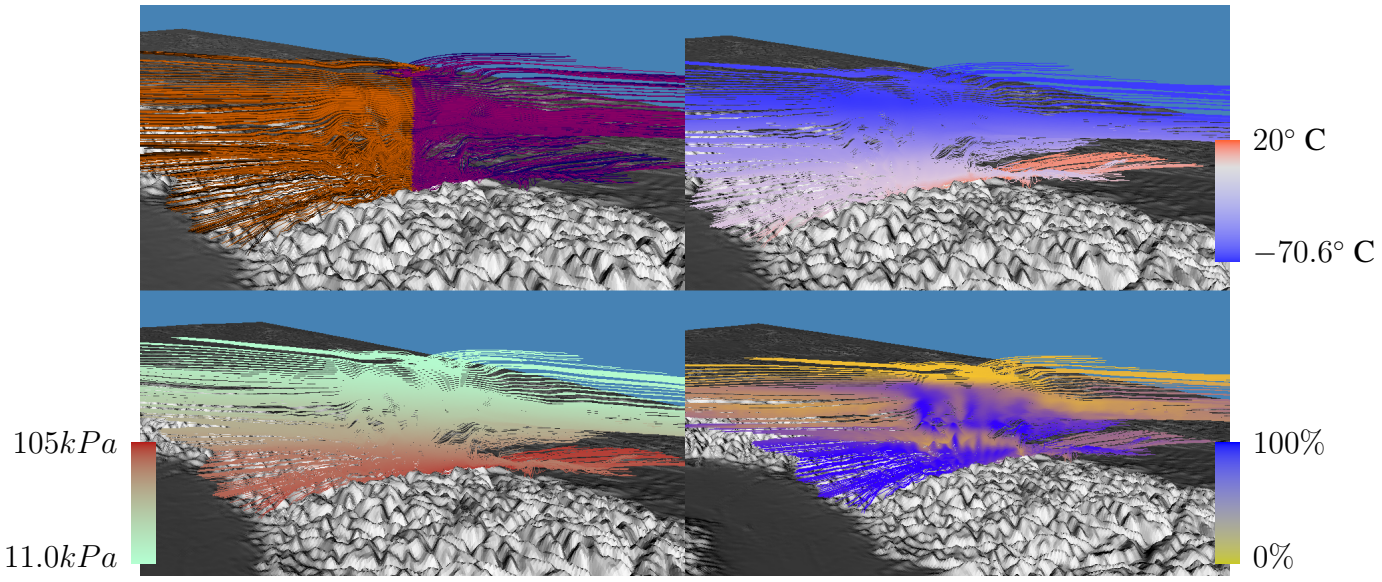


Figure 4.2: *Colors according to same criteria as in figure 4.1 above. The colors in the upper left case also depend on the trajectory set: Purple/blue forward, orange/black backward*

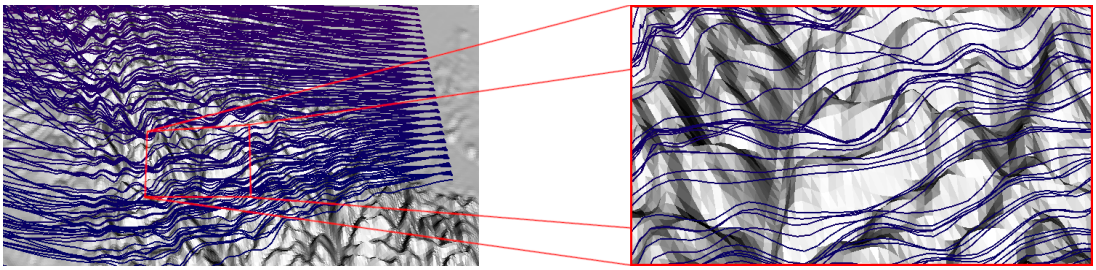


Figure 4.3: *A closer look of the top left part of figure 4.1*

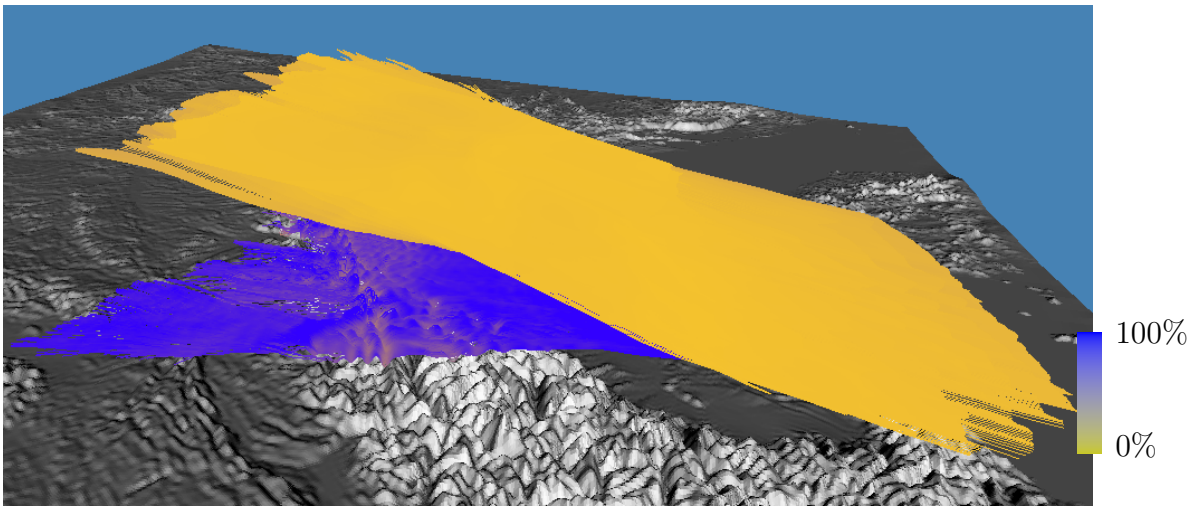


Figure 4.4: *Relative humidity is high below and low above*

Figure 4.4 shows that the air high above ($14 - 16km$) has low relative humidity, especially compared to the lower trajectories (which start around height $4km$). This is also forward and backward, third day, $3:00$ to $9:00$. Also visible: The wind speeds are higher above and so the upper trajectories cover more area than the lower ones.

4.2 Quality comparison

The plots in figure 4.5 plot the difference between trajectories computed by LAGRANTO and five variants of our tracing algorithm. All values are averaged over around 7000 trajectories (some of which are pictured in figuer 4.6). The five variants are:

- Copying LAGRANTO: Settings to perform almost exactly the same operations as LAGRANTO
- Sample W correctly: Examining the LAGRANTO code showed that the vertical velocity W was being sampled on a staggered grid even after it had been destaggered. This is an error.
- Level interpolation on 4 columns: Use the procedure described in section 3.2.2 .
- Runge-Kutta instead of Iterative Euler: Changes the ODE solver.
- All improvements: Combines the three variants above.

The test trajectories should not leave the domain because we handle particles outside in a different way than LAGRANTO. As seen in figure 4.6, all trajectories start and end at acceptable (lon,lat) coordinates. Collisions with the ground are avoided by initialising all particles at heights of at least $7km$. There is a very small number of trajectories that reach the surface when the time step is $h = 5min$. This is unfortunate because it distorts the plots, even though the effect is lessened by averaging with several thousand unproblematic trajectories.

What can be seen in the plots of figure 4.5 is that the black line (representing the LAGRANTO-like setting) stays at a very low value. There are small variations, but apart from the boundary cases we can reproduce the LAGRANTO results almost exactly.

The right side which measures the average vertical distance looks very chaotic and is not very useful for gathering information. The reason for this is unclear.

The choice of integrator does not matter that much for small timesteps, but in the case of $h = 5min$ the Runge-Kutta curve dominates the left plot. This is expected because the Euler integrator works worse with larger timesteps. Curiously, the green curve stays closer to the other methods, even though it also uses the Runge-Kutta integrator.

It appears that the choice of how to sample across $level$s makes more of a difference than correcting the sampling of W. The fact that both of those affect the z axis may be another factor in the irregular curves on the right side.

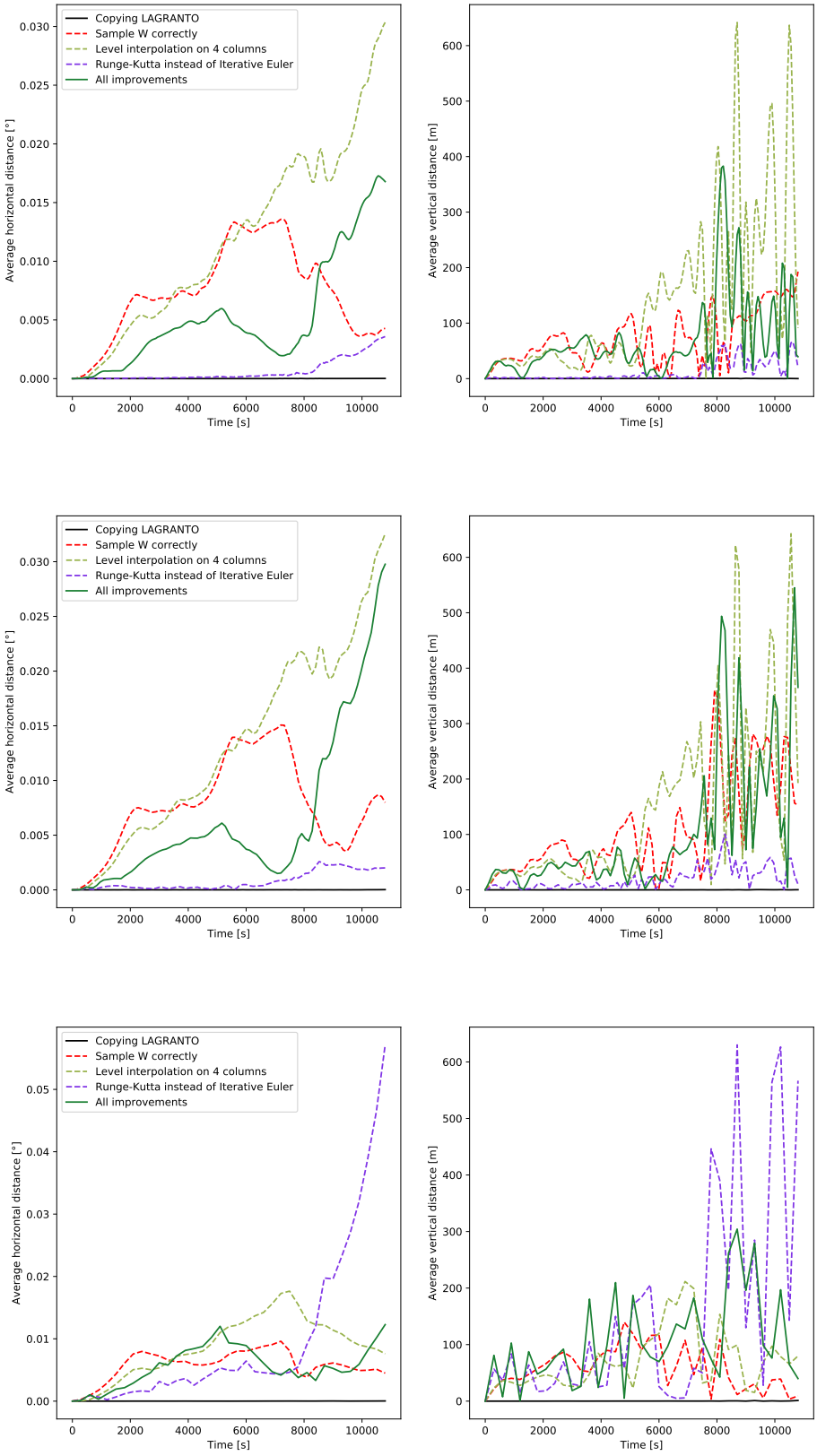


Figure 4.5: *Average distance between LAGRANTO trajectories and ours, timestep is* $h = 1min$ *on top,* $h = 2min$ *in the middle,* $h = 5min$ *on the bottom*

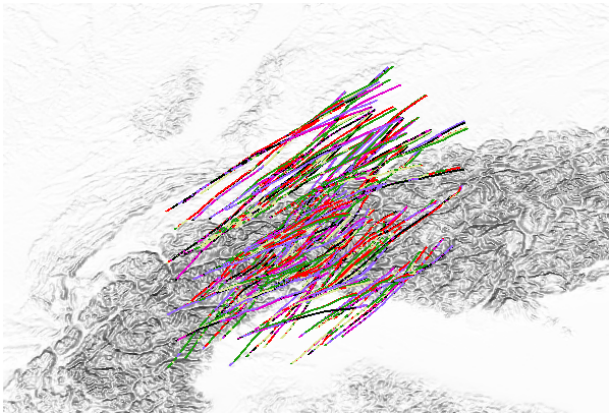


Figure 4.6: *Trajectories that feed into the plots of figure 4.5. use better pic? btw pink means lagranto*

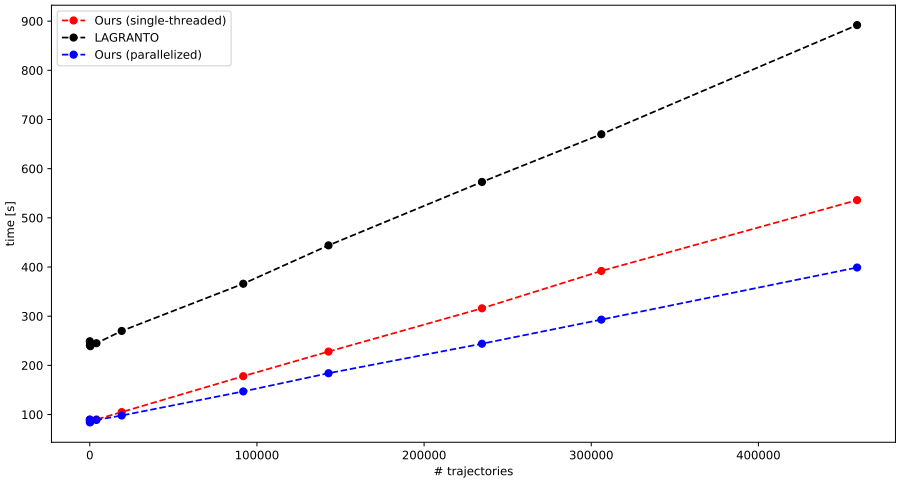


Figure 4.7: *Times for tracing a variable number of particles over* 60 *timesteps and* 7 *data files*

4.3 Performance

There are two main performance bottlenecks: Reading the data and doing computations on each particle and timestep. We added a simple OpenMP parallelization for iterating over all trajectories during the simulation phase and hope to at least match the speed of LAGRANTO with that.

Figure 4.7 plots the measured times for a test case of computing different numbers of trajectories. As expected, the time scales linearly in the number of trajectories. Our method is faster than LAGRANTO in all cases. In the early part, where the reading of the data makes up almost all of the time, our version does in roughly 1.5 minutes what LAGRANTO does in 4. Increasing the number of trajectories, it becomes evident that our version also has a better scaling, even when unparallelized.

It is possible that the rather slow speed of LAGRANTO is due to not using optimal compiler options for the chosen problem and the computer used.

All time measurements were taken on the same machine: It has an Intel® Core™ i5-3427U CPU with 4 cores running at 1.80 GHz with 7.7 GiB of memory.

5

Conclusion

The basic functionality of computing trajectories according to input data works well. The visualization and analysis parts ended up rather limited due to time constraints.

Comparing to LAGRANTO: Better performance and arguably better results have been achieved. Some features are missing but would be quite simple to add (initial points read from a file, write down only every nth iteration).

There are still some differences in our variant to reproduce LAGRANTO results: Points outside the domain are not handled the same way as in LAGRANTO. LAGRANTO uses fixed units vor certain variables (e.g. temperature in Celsius) whereas our code simply uses the values from the data files.

While the particle tracer works, it can still be extended. Because the region of interest for this project was relatively small and compact, the solver can not handle trajectories that reach the boundaries of the coordinate system (poles, date line). The borders of the local domain hold issues as well. Particles near the ground and outside the ($rlon$,$rlat$) domain are handled differently between LAGRANTO and our code and both methods lead to undesirable results sometimes. The parallelization of the solver is currently very simple and could most likely be improved in several ways. One feature that is missing when compared to LAGRANTO is reading input data that is not in the COSMO format.

The visualization with VTK is very limited. While it is possible to read trajectories and show how they pass over the landscape, most settings are hardcoded. Displaying different trajectories or changing the variables requires changing the code. Optimally this would have been done using a GUI. Currently all loaded trajectories are displayed at once and there is no way to, for example, select only those that pass through a certain region or those with a certain average temperature.

Although it is possible obtain trajectories and look at them, we did not really look into the non-technical results. The search for interesting events would become much easier if the visu-

alization tool was extended first. As is, we mostly found common features like wave-shaped trajectories.

Bibliography

[BSF+11] Michael Baldauf, Axel Seifert, Jochen Förstner, Detlev Majewski, Matthias Raschendorfer, and Thorsten Reinhardt. Operational Convective-Scale Numerical Weather Prediction with the COSMO Model: Description and Sensitivities. *Monthly Weather Review*, 139(12):3887–3905, December 2011.

[PTVF92] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C (2nd Ed.): The Art of Scientific Computing*. Cambridge University Press, USA, 1992.

[SDS+03] J. Steppeler, G. Doms, U. Schättler, H. W. Bitzer, A. Gassmann, U. Damrath, and G. Gregoric. Meso-gamma scale forecasts using the nonhydrostatic model lm. *Meteorology and Atmospheric Physics*, 82:75–96, 2003.

[SML06] Will Schroeder, Ken Martin, and Bill Lorensen. *The Visualization Toolkit (4th ed.)*. Kitware, 2006.

[SW15] M. Sprenger and Heini Wernli. The lagranto lagrangian analysis tool - version 2.0. *Geoscientific Model Development*, 8, 08 2015.