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 visualizing wind trajectories "over the alpine region". LAGRANTO is used as a template for computing trajectories and VTK for drawing the results. While the initial focus was on the visualization, it shifted from making the basics work to comparing results with LAGRANTO over the duration of three months.

Zusammenfassung

Deutsche Version kommt erst, wenn die englische Version fertig ist.

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 tool for computing trajectories from wind velocity fields. We aim to create a similar program in a more modern programming language - C++ as opposed to Fortran. While reconstructing LAGRANTO is our base goal, we would like to get better performance and/or results as well. We succeeded in reproducing the results from LAGRANTO. Along the way, we discovered a few questionable algorithmic choices in the LAGRANTO code, some of which were later reimplemented for comparison purposes.

2

Background

2.1 Data

We work on a set of NetCDF files containing assorted meteorological data. 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 time 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.

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 northward component. All three have the same units (m/s) and similar but not equal grids. The grids are all staggered: All vertices in one grid are translated by half a cell 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.

$HSURF$ contains the height of the surface for given ($rlon$, $rlat$) coordinates. It is mainly used

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*

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-

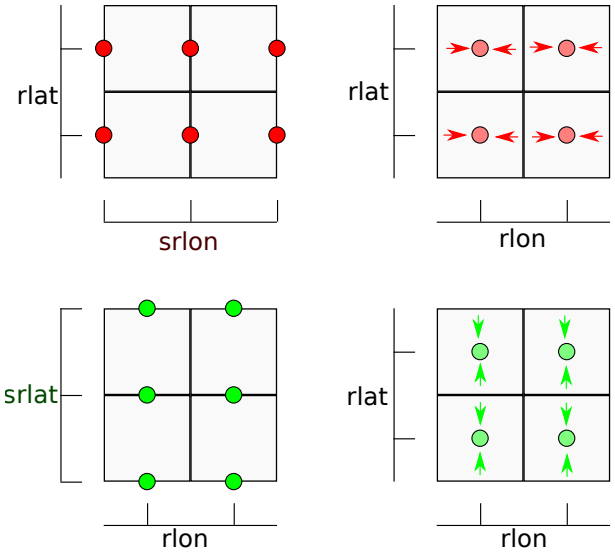

Figure 2.1: *Left: Points in staggered grids; Right: Destaggering by averaging two staggered points*

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.

3

Method

3.1 Numeric integration

Solving differential equations of all types is a topic for itself, so here we have just the methods used to find the next point of a trajectory given the 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.

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 Euler method has an error $O(h^3)$ per step (order 2) whereas the Runge-Kutta method has the lower error $O(h^5)$ (order 4).

Iterative Euler

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

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

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

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

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

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

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

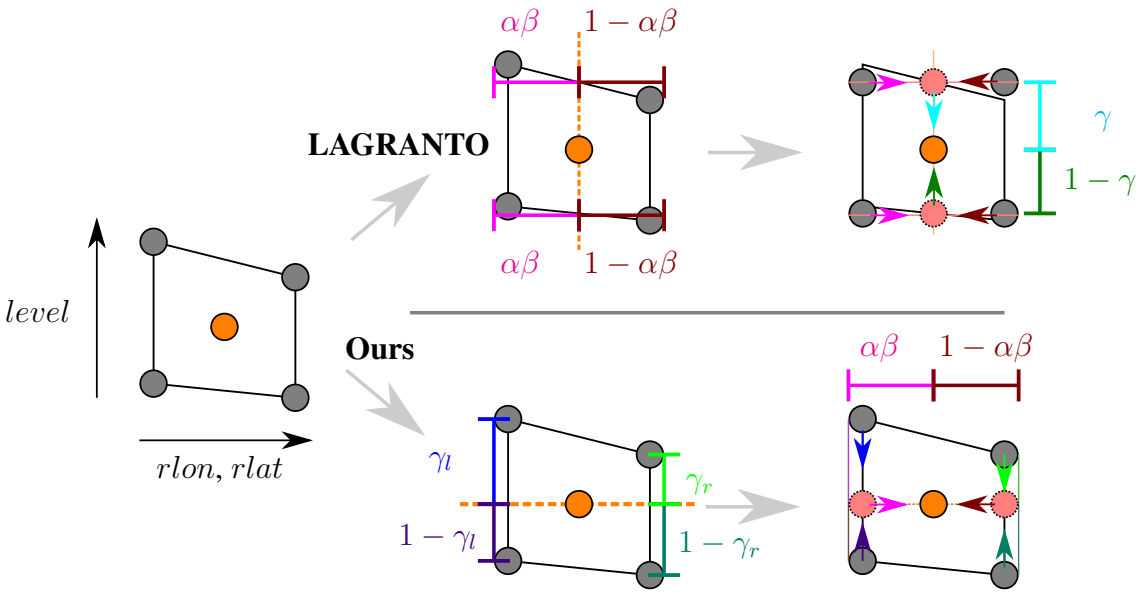

Figure 3.1: *Interpolation procedure for sampling at the orange point: Order and weights depend on the method used*

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: First 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 ($rlon$,$rlat$,$level$). All of them 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. (x_1,y_1,z_1) is the position of the upper northeastern corner.

3.2.2 Using adjacent level heights

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

The lower part of figure 3.1 shows how we compute the interpolated value at the orange 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. The weights for the horizontal interpolation (magenta and dark red in the picture) 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.

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.

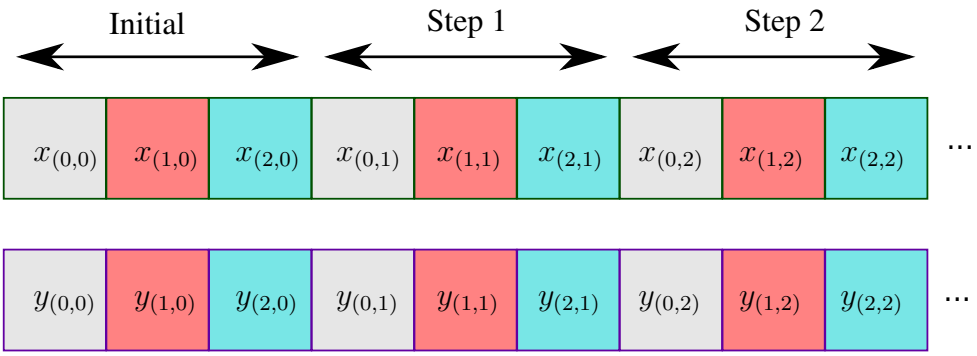

Figure 3.2: *Structure of the output data: $x_{(i,j)}$ is the value of variable x on trajectory i after the jth timestep*

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, 3 trajectories (gray, red, blue), and 5 timesteps, for a total of 18 elements per array.

This format matches the output file of LAGRANTO, allowing us to compare the results directly.

3.4 Analysis

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 HHL 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 a 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.

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.

4

Results

4.1 Pictures and stuff

4.2 Comparison

The plots in figure 4.3 plot the difference between trajectories computed by LAGRANTO and five variants of our tracing algorithm. All values are averaged over around 7000 trajectories. The five variants are:

- Copying LAGRANTO: Settings to perform almost 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: Use a different ODE solver
- All improvements: Combines the three variants above

TODO visualize trajectories as well

All trajectories start and end within the boundaries of the domain. Frequent collisions with the ground would distort the results significantly and have been avoided by the choice of starting positions and by not tracing particles for too long.

What can be seen is that the black line is stays at a very low value. There are small variations, but we can reproduce the LAGRANTO results almost exactly.

The right side which measures the average vertical distance looks very chaotic and is not very

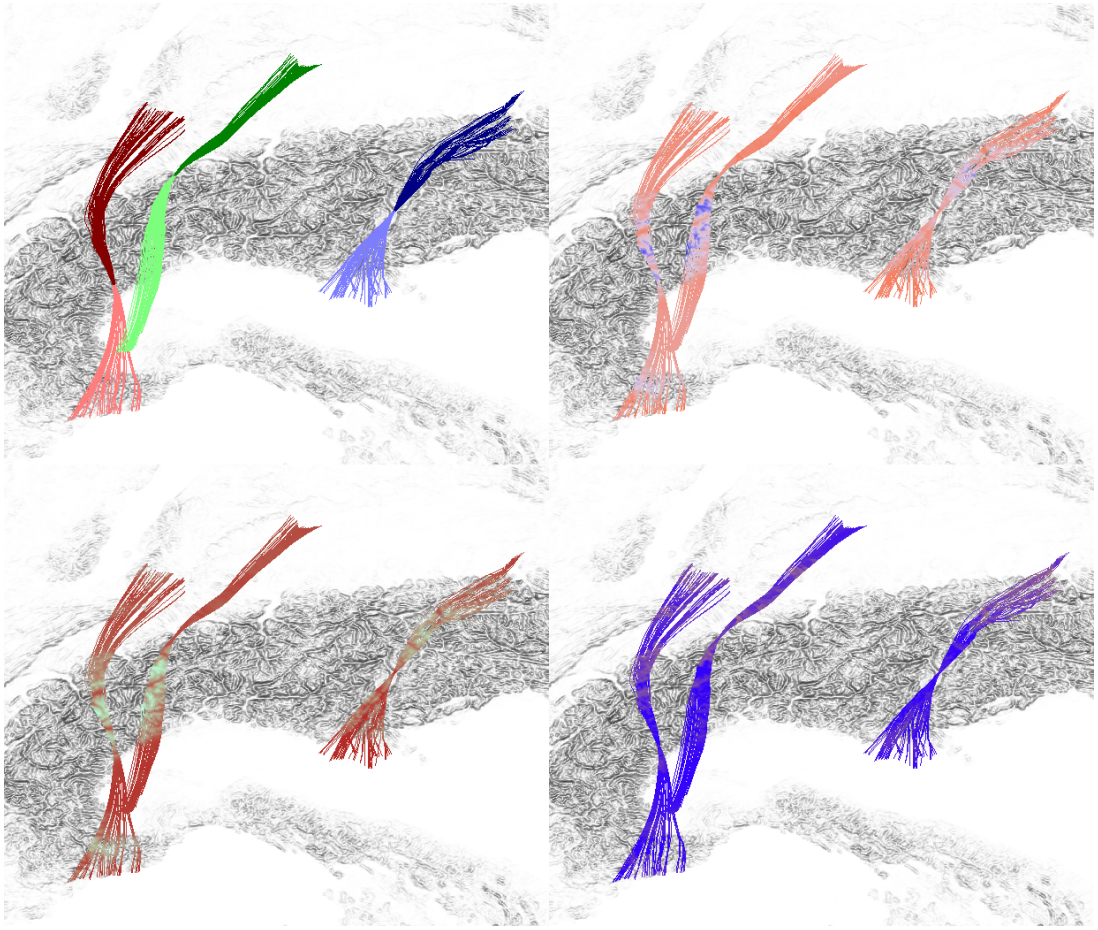

Figure 4.1: *Top left: Color by trajectory set; Top right: Color by temperature; Bottom left: Color by Pressure; Bottom right: Color by humidity*

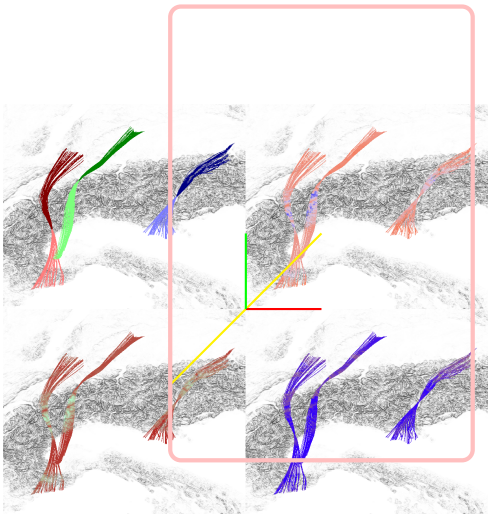

Figure 4.2: *drawing stuff with tikz*

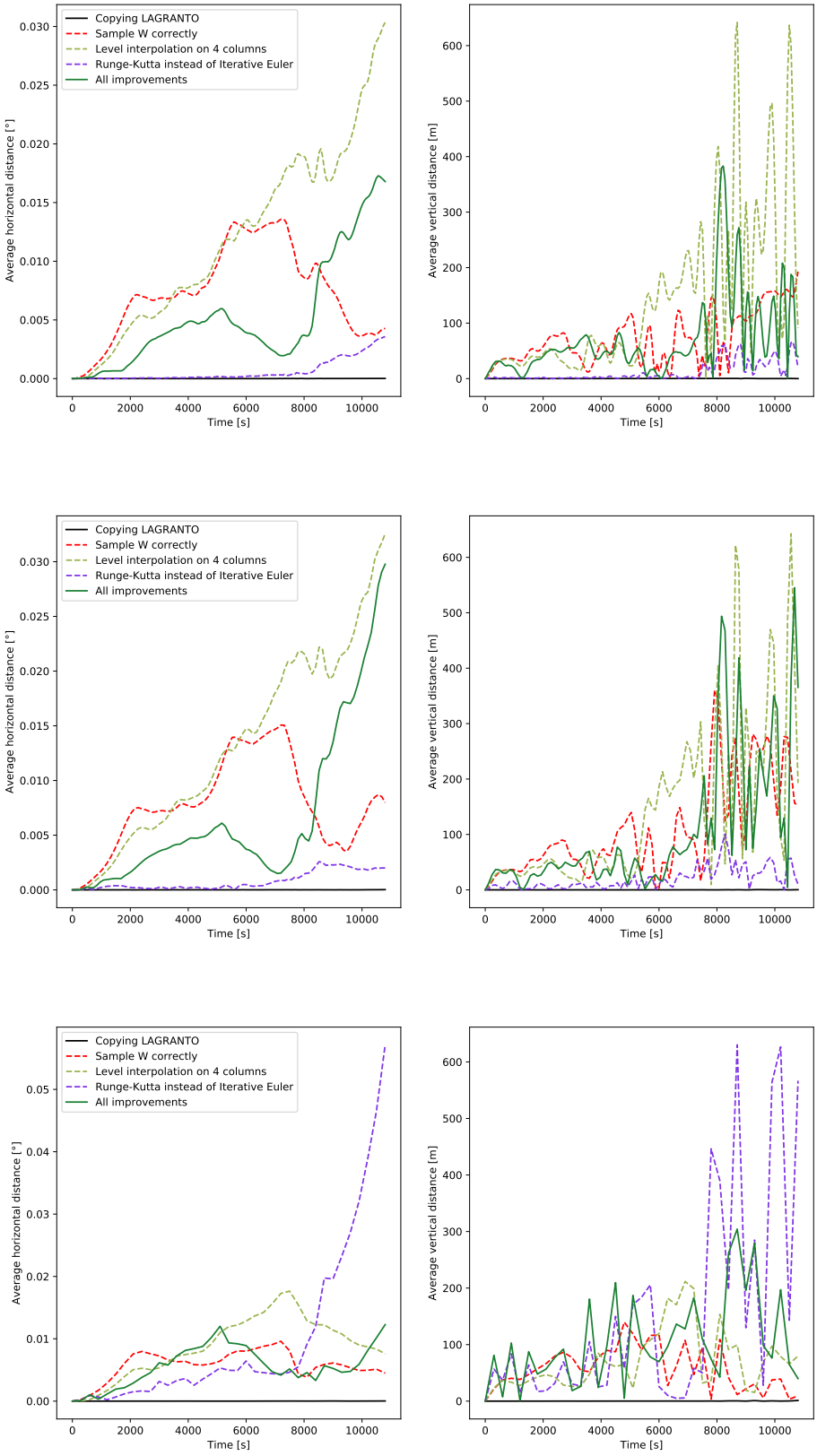

Figure 4.3: *Average distance between LAGRANTO trajectories and ours, timestep is* $h = 1min$ *on top,* $h = 2min$ *in the middle,* $h = 5min$ *on the bottom*

useful for gathering information. The reason for that is unclear, possibly related to how trajectories make wave shapes over mountains.

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. When viewing the actual trajectories, they look smoother for Runge-Kutta in that case.

It appears that 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.

4.3 old stuff starts here

Comparing results: I can reproduce LAGRANTO pretty well ... and which improvements do more or less? Figure **??** shows the average distance between a bunch of points in my version and lagranto. left is horizontal, right is vertical. i really cant tell much from that example. i mean, an almost constant, almost zero difference is possible, but how much the various improvements change the result is hard to say... Sampling on four pillars seems more significant than using the Runge-Kutta integrator or correcting W, but maybe that is because of the timestep

also, its possible to diverge from lagranto even with the "right" settings, but whatever

4.4 Performance

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

In a first test, there were only four trajectories, meaning most of the work is reading UVW. For 1 hour of simulation data, this means 7 data files need to be used. Reading UVW represents the main workload in this case. The measured times are $4m12s$ when using LAGRANTO $1m37s$ in our case.

The second test uses the same data and similarly computes trajectories over 1 hour using a timestep of 1 minute and reading a new UVW every 10 minutes. By tracing over 400000 particles, the time shifts toward the tracing process. The times obtained were $19m8s$ for LAGRANTO and $4m35s$ using our code.

This seems to imply that our C++ implementation is significantly faster than LAGRANTO. yaay

5

Conclusion

everything is great

all questions have been answered

nobody has to work on this topic ever again

...

that was a joke

5.1 more serious

Overall, it's fine. It is possible to compute trajectories and view them without bugs. Performance-wise, our code runs significantly faster than LAGRANTO.

There are several possible extensions:

- Improving the input interface so it is possible to read most parameters from files instead of entering them manually each time

- Parallelizing more: Currently the loop for propagating all trajectories is parallelized with OpenMP in a very simple manner. That is all.

- Better solver: Who knows if I really did it right?

- Different outputs: Maybe you want something other than just points?

- The viewer has only very basic functionality. Most settings are hardcoded and there is no GUI to speak of.

Bibliography

[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.