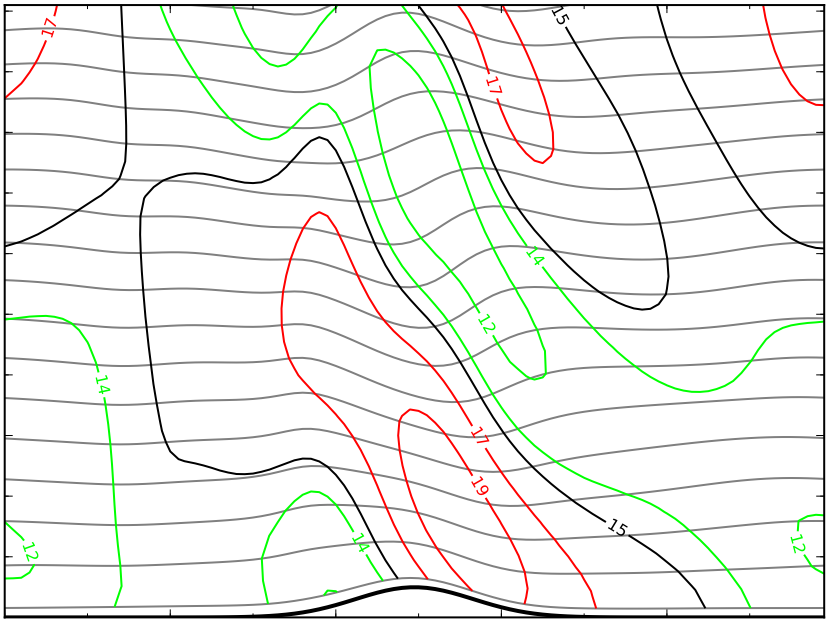
Term project

Isen: A two-dimensional isentropic vertical coordiante model

Fabian Thüring

thfabian@student.ethz.ch

June, 2016



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

1 Introduction

This report will present **Isen**, a two dimensional high-performance isentropic model derived from the existing Matlab implementation of the course Numerical Modelling of Weather and Climate. The newly developed model[1], written in C++11, achieves a speedup factor of 49x with respect to the Matlab code while retaining the ease of use due to a powerful Python frontend. The code is carefully optimized for modern shared-memory multiprocessors using different programming models such as OpenMP. On the example of several experiments, ranging from high-performance computing to numerical modelling and atmospheric physics, the viability of the new model will be demonstrated.

1.1 Isentropic Models

Isen is an isentropic model, thus the vertical coordinate is given by the potential temperature,

$$\theta = T \left(\frac{p_{ref}}{p} \right)^{R/cp},$$

with $p_{ref} = 1000$ hPa being an arbitrary chosen reference pressure, $R = 287$ J/(K kg) the gas constant for dry air and $c_p = 1004$ J/(K kg) the specific heat of dry air at constant pressure. The vertical velocity can then be defined as

$$\dot{\theta} = \frac{D\theta}{Dt}.$$

Consequently, we have vanishing vertical velocities under adiabatic flow regimes ($D\theta/Dt = 0$). Isentropic models are therefore very popular to study idealized adiabatic flow problems as the system reduces to a stack of two dimensional θ-layers.

In this project we will study a two-dimensional, adiabatic flow over a mountain ridge (as depicted on the title page). We will further simplify the system by neglecting effects due to Earth's rotation and frictional forces.

The next section will describe the physical model in greater detail. Section three will present an overview of the implementation highlights such as the seamless interaction of C++ and Python or the verification framework and conclude with a performance analysis of the model. In the last section we will discuss some numerical and atmospheric physics experiments performed with **Isen** by exploiting the gained compute-power.

2 Model Description

In this section the foundation of the model, as well as the theoretical background, are discussed.

2.1 Governing Equations

Given the full Navier-Stokes equation and the first law of thermodynamics, we can apply our simplifying assumptions to derive the six governing equations of our model:

[1] code available at `https://github.com/thfabian/Isen`

Horizontal momentum equation in x-direction

$$\frac{Du}{Dt} = \frac{\partial u}{\partial t} + u\left(\frac{\partial u}{\partial x}\right)_\theta = -\left(\frac{\partial M}{\partial x}\right)_\theta \tag{1}$$

with the Montgomery potential given by $M = \phi + c_pT = gz + c_pT$.

Two-dimensional equation of continuity

$$\frac{\partial \sigma}{\partial t} + \left(\frac{\partial \sigma u}{\partial x}\right)_\theta = 0 \tag{2}$$

where the isentropic density is given by $\sigma = -\frac{1}{g}\frac{\partial p}{\partial \theta}$.

Hydrostatic relation

$$\pi = \frac{\partial M}{\partial \theta} \text{ and } \pi = c_p\left(\frac{p}{p_{ref}}\right)^{\frac{R}{c_p}} \tag{3}$$

where π is the Exner function which can be viewed as a non-dimensionalized pressure.

Passive advection of moisture scalars

$$\frac{\partial q_v}{\partial t} + u\frac{\partial q_v}{\partial x} = 0 \tag{4}$$

$$\frac{\partial q_c}{\partial t} + u\frac{\partial q_c}{\partial x} = 0 \tag{5}$$

$$\frac{\partial q_r}{\partial t} + u\frac{\partial q_r}{\partial x} = 0 \tag{6}$$

with q_v, q_c and q_r corresponding to the mixing ratio of water vapor, cloud water and rain droplets. These equations describe the passive advection of the moisture scalars without any microphysical interactions among them. We will correct this later by using a microphysical parametrization.

2.2 Discretized System

In order to solve the previously derived equations on a computer, we will discretize them on a regular two dimensional staggered grid of size $N_i \times N_k$. Hence, a grid point (x_i, z_k) is defined as

$$(x_i, z_k) = ((i - \frac{N_i}{2} + 1) \cdot \Delta x, k \cdot \Delta z) \quad i \in \{0, \ldots, N_i - 1\}\ k \in \{0, \ldots, N_k - 1\} \tag{7}$$

with Δx and Δz being the spatial resolution in the horizontal and vertical, respectively. The continuous system of equations (1) - (6) will be transformed to a discrete system defined at the grid points. This requires to approximate the derivatives in a discrete manner using central finite differences. Hence, for a function $\phi(x_i) = \phi_i$ this results in

$$\frac{\partial \phi(x_i)}{\partial x} \approx \frac{\phi(x_i + \Delta x) - \phi(x_i - \Delta x)}{2\Delta x} = \frac{\phi_{i+1} - \phi_{i-1}}{2\Delta x} \tag{8}$$

with a discretization error of order $\mathcal{O}(\Delta x^2)$. As already mentioned, we use a staggered grid in the horizontal as well as in the vertical to avoid the infamous odd-even decoupling.

This means, the isentropic density σ and Montgomery potential M are placed at the grid points while horizontal velocity u is located between two grid points. In the vertical we stagger the pressure p and the Exner function π and again store the Montgomery potential M at full grid points.

2.3 Initial Conditions

By default, the horizontal velocity u is initialized with a constant

$$u(x, z, t = 0) \equiv u_0. \tag{9}$$

The topography *topo* is described as

$$topo(x_i) = a \cdot e^{-(x_i/b)^2} \tag{10}$$

where a is the maximum height of the mountain and b its full width at half maximum. The initial condition for the Exner function π, pressure p, Montgomery potential M as well as the isentropic density σ are determined from the Brunt-Väisälä frequency N, corresponding to the vertical stratification,

$$N^2 = \frac{g}{\theta}\frac{d\theta}{dz} \tag{11}$$

which is initialized to a constant $N(x, z, t = 0) \equiv N_0$.

2.4 Boundary Conditions

The model supports two different kinds of lateral boundary conditions: periodic and relaxed. In the first case we will enforce periodicity by copying the boundary fields while in the latter the prognostic fields are relaxed towards their initial states boundary values. This means for a prognostic field $\phi(x, z)$

$$\phi(x, z) = (1 - \chi(x))\phi(x, z) + \chi(x)\phi_b(z) \tag{12}$$

where ϕ_b is the value at the boundary and $\chi(x)$ is a weighting function which determines if an arriving wave is reflected at the boundary or not.

For the upper and lower boundaries we will use constant isentropic planes θ_s and θ_t. The pressure at the upper boundary $p(\theta_t) = p_0(\theta_t)$ will remain at the initial pressure distribution.

2.5 Microphysical Parametrizations

To be able to forecast precipitation we have to add sources and sinks to the moisture scalar equations (4)-(6). In this project we will use the Kessler microphysical scheme [4]. The Kessler parametrization tries to capture the warm-rain processes which occur as soon as the atmosphere achieves saturation and water vapor starts to condensate, initiating exchange processes between water vapor, cloud droplets and rain droplets. The moisture

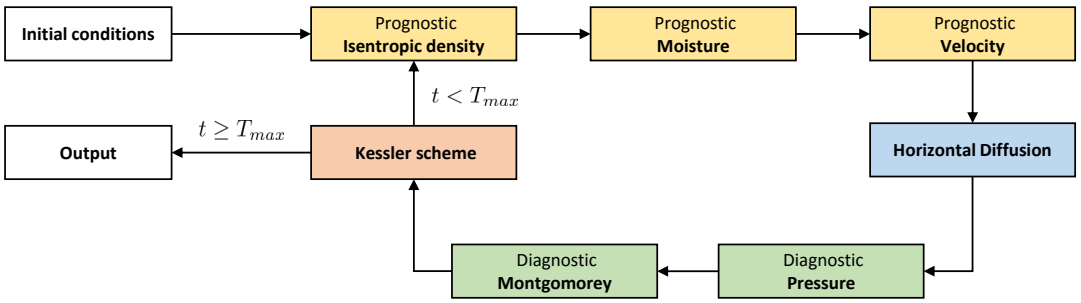


Figure 1: Program flow of the model containing prognostic, diagnostic, microphysical and diffusion steps.

scalar equations are modified to

$$\frac{\partial q_v}{\partial t} + u\frac{\partial q_v}{\partial x} = -G + EP \tag{13}$$

$$\frac{\partial q_c}{\partial t} + u\frac{\partial q_c}{\partial x} = G - CC - AC \tag{14}$$

$$\frac{\partial q_r}{\partial t} + u\frac{\partial q_r}{\partial x} = CC + AC - EP \tag{15}$$

with G being the condensation rate of water vapor and the evaporation rate of cloud water. CC corresponds to the conversion rate of cloud water q_c and precipitation particles q_r due to collision and coalescence. AC represents the accretion process and finally EP includes the sedimentation of rain.

3 Implementation Details

To gain a deeper understanding of the model, and improve my programming skills, I decided to rewrite the entire model in C++ . The new model pursues vastly different design choices by embracing object oriented and modular programming. Relying on specialized libraries such as the Eigen vector library [2] it was possible to retain the Matlab like notation and readability while decreasing the runtime by orders of magnitudes. Furthermore, **Isen** can be seen as a demonstration on how lower level programming languages (C++) can be combined with higher level ones (Python) to enhance the prototyping experience.

3.1 Design of Isen

To solve the system of equations ((1)-(3),(13)-(15)) a series of diagnostic, prognostic, microphysical and diffusion steps are executed as shown in Figure 1. For a dry model the Kessler parametrization and prognostic step of the moisture variables can be skipped.

While the code base grew reasonably large (~10'000 lines of code), most of the actual simulation code is confined in the `Solver` class (`lib/IsenCore/Solver.cpp`), which implements the prognostic, diagnostic and diffusion steps in a straightforward way. The `Solver` class was designed in a way which allows to easily extend and possibly override the individual methods with more powerful implementations using programming models such OpenMP [6] or SIMD vectorization.

To initiate the simulation, some input variables such as the number of grid points n_x, the height of the topography or the initial velocity distribution u_0 have to be specified. While the model has defaults for all these values, most users want to manually specify these quantities. Therefore, Isen has an elaborate `Parser` which is able to directly read the Matlab or Python `namelist` files and thus making comparisons of the implementations easy. Like it's Matlab counterpart, Isen is capable of writing the results of the simulation to an output file while supporting various formats such as plain ASCII text files, formatted XML files or native-binary archives.

Isen relies on a few external libraries such as the Boost C++ libraries [5] to abstract operating system dependencies and perform utility tasks. Furthermore, the Eigen vector library [2] is used to ease to burden of dealing with arrays. The code of Isen was written cross platform compatible, meaning Isen runs on Windows, Linux and Mac OSX. Instructions on how to compile the code on these plaforms can be found on the project's GitHub page (`https://github.com/thfabian/Isen`).

3.2 Python Interface

As C++ is quite complex and, more importantly, a compiled language, it is often not the first choice for fast prototyping and experimenting with a model. However, with for example the help of Boost.Python [1] it is possible to expose C++ interfaces to Python and thus combining the flexibility and scriptability of Python with the raw power of compiled and carefully optimized C++ code. For **Isen** this exact route was taken by exposing the `Solver`, `NameList` and `Output` interfaces to Python. Isen can therefore either be used direclty from the command-line with the binary driver `isen.exe` or called via the `IsenPython` module from Python as shown in the following code snippet (Listing 1).

```
from IsenPython import Solver, NameList, Output
from IsenPython.Visualizer import Visualizer

# Setup Namelist
namelist = NameList()
namelist.nx = 100

# Setup & run Solver
solver = Solver()
solver.init(namelist)

solver.run()

# Plot horizontal velocity distribution
output = solver.getOutput()
visualizer = Visualizer(output)
visualizer.plot('horizontal_velocity', 6)
```

Listing 1: Example program of the model in Python.

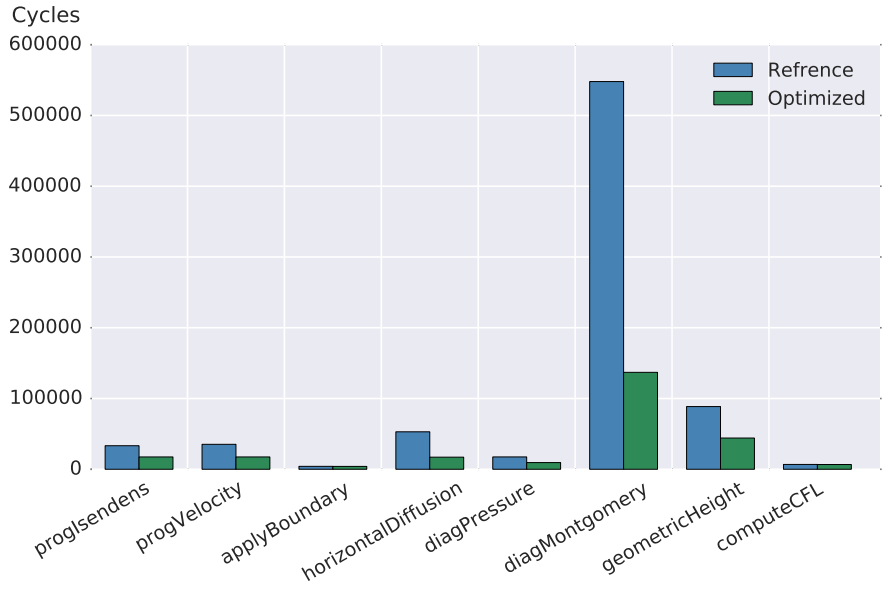


Figure 2: Average runtime in cycles of one timestep of the dry model for the reference implemenation (`Solver`) and the optimized CPU implementation (`SolverCpu`)

3.3 Verification Framework

While the Matlab and Python implementations completely lack any kind of automated validation, **Isen** was built with a strong verification framework. This resulted in over 300 unittests assuring the correctness of the Parser, I/O operations as well as the Solver implementation itself. For example, to test the Solver implementation all fields of the new model are compared against serialized fields from the Matlab implementation. This task is is done after initializing the fields and again after performing some timesteps to guarantee equality within an accuracy of atleast 10^{-10}.

To maintain correctness of the program across all operating systems, it is useful to use some sort of Continuous Integration (CI). CI services, such as Travis CI [7] for Linux and Mac OSX, will fetch the code repository after each commit to GitHub and build/test the project on a dedicated server. The **Isen** codebase is thus thoroughly tested on all major platforms.

3.4 Performance Analysis

While development in Matlab or Python is super fast, it often comes at a price of suboptimal code. At some point one might want to explore the computational boundaries of the model by for example drastically increasing the spatial resolution. Generally, these kind of tasks are infeasible with an implementation purely written in an high-level programming language. However, porting the code to a low-level programming language in a straightforward manner by no means guarantees a major performance boost. Often one is forced to do tedious micro-optimizations to gain an increase in performance which justifies the effort of porting the code. Writing high-performance code is therefore a vital aspect of numerical modelling.

Optimization strategies The `Solver` class implements the algorithm presented in Figure 1 in the same fashion as it's Matlab counterpart. The code itself looks astonishingly similar to the Matlab code (e.g see `Solver::progVelocity`) and coding it was rather straightforward. The `Solver` methods already achieve a speedup factor of about 15-20x compared to the Matlab version which is partly because unnecessary memory allocations were removed but mainly because C++ can be mapped more efficiently to machine code.

However, we most likely can do better. For instance, the `Solver` code still only uses one core and we can gain some speedup (up to the number of cores) by parallelizing it with OpenMP. Further, inspecting the assembly code revealed the compiler's inability to use single instruction multiple data (SIMD) instructions, commonly referred to as auto-vectorization. Modern SIMD vector instructions can process 4 double precision floating point numbers at the same time and allow a theoretical, though often unreachable, speedup of 4x. These considerations were brought to code in a new Solver implementation: `SolverCpu`. `SolverCpu` differs from the `Solver` by using C function kernels to more reliably trigger certain compiler optimizations (see `lib/IsenCore/SolverCpu.cpp`).

Figure 2 depicts a break-down of the individual methods of a time step of the dry model and shows the improvements from the `Solver` to the optimized `SolverCpu` implementation. It is particularly interesting that 70% of the time is spent in diagnostically calculating the exner function π and the Montgomery potential M. This is of course due to the costly power function in the exner function (3).

Performance Comparisons Figure 3 shows the speedup of the final version of **Isen** compared to the Matlab code. The vertical resolution was fixed at 60 grid points while the number of horizontal grid points were increased. Note that the drop in performance for large problem sizes is due to cache effects. This could be fixed by using a blocked implementation for stencils which have extents in the vertical, like both diagnostic steps.

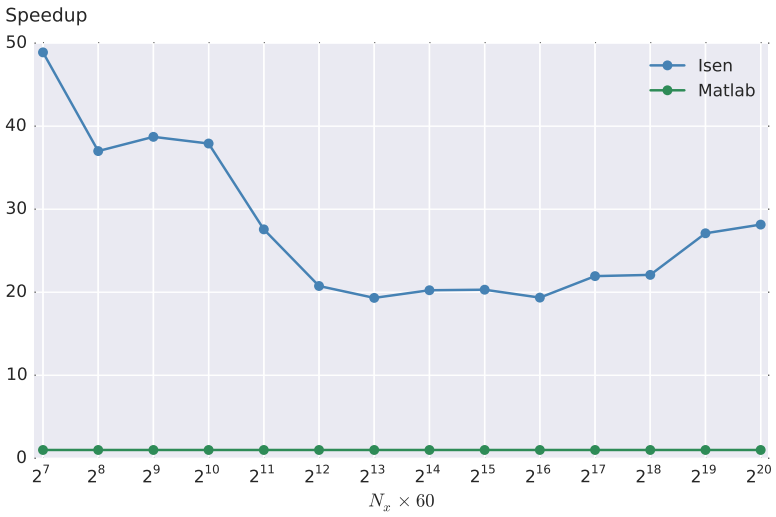


Figure 3: Speedup of **Isen** compared to the Matlab implementation averaged over 100 timesteps of the dry model. **Isen** was compiled using the Intel compiler (16.0.1) and the benchmark was performed on an Intel Core i7-4771 running Linux.

4 Numerical Experiments

In this last section we will use **Isen** to gain some insight on how the model resolution effects the physics by running simulations with high resolution.

4.1 Effect of Horizontal Resolution on Steady State

The experiment will compare the steady state of the dry and later moist model under different horizontal resolutions $\Delta x = \frac{x_L}{n_x}$ with x_L being the horizontal extent, which will be kept at 500 km, and n_x being the number of grid points in the horizontal.

Steady State In our simple model a *steady state* is represented by a stationary gravity wave above the mountain ridge. Considering the fact that we start with a uniform wind profile, the flow will need some time to reach this state. Figure 4 shows the velocity profile after 5, 20 and 60 hours of simulation. By observation we conclude that after 60h the steady state seems to be reached.

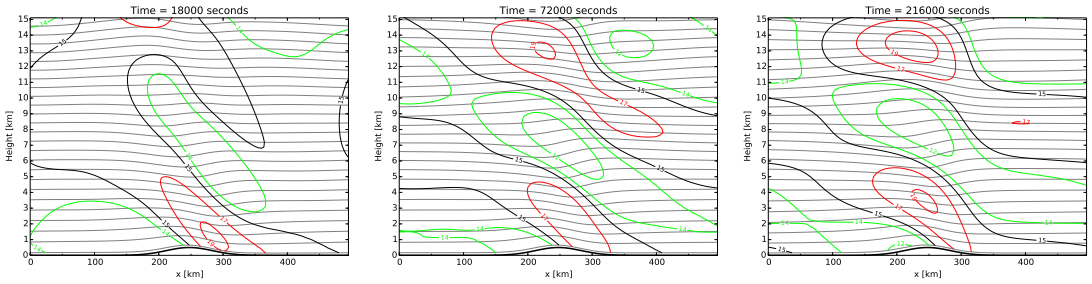


Figure 4: Velocity profile after 5, 20 and 60 hours of simulation.

Horizontal Resolution We will run the experiment with 100, 500, 1000 and 5000 horizontal grid points n_x corresponding to spatial resolutions of 5 km, 1km, 500 m and 100 m. By increasing the spatial resolution, we have to make sure to *not* violate the Courant–Friedrichs–Lewy (CFL) condition. The CFL condition is a necessary condition for convergence of the finite difference approximations and given as

$$\left| \frac{u_{max} \Delta t}{\Delta x} \right| = C_{max} \leq 1. \tag{16}$$

The following pairs of Δx and Δt were used in the experiment

Δx	5 km	1 km	500 m	100 m
Δt	1 s	0.2 s	0.1 s	0.02 s

Note that this yields the 100 m simulation 2500 times more expensive than the 5 km one.

Setup All four simulations use 60 isentropic planes ($n_z = 60$) and a mountain with a height of 500 m and a half width of 50 km. The Brunt-Väisälä frequency was set to 0.01, the surface potential temperature to 300 and the velocity profile was uniformly initialized to $u_0 = 15$ m/s.

Results The simulations were run on one node of the Euler cluster of ETH Zürich which is equipped with 12-core Intel Xeon E5-2697v2 processors. Nevertheless, the 100

m simulation took 9 hours to complete which would roughly correspond to 20 days running the same simulation in Matlab. The following velocity profiles in Figure 5, 6, 7 and 8 show the steady state (60h) of the different simulations.

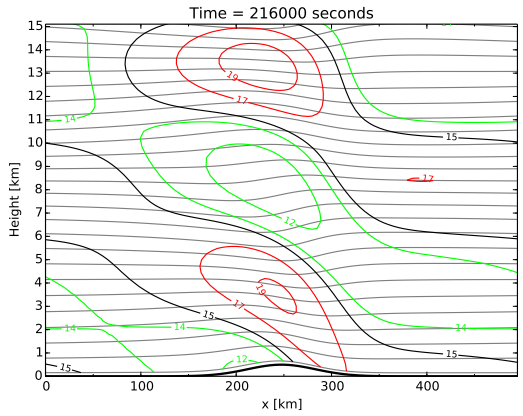


Figure 5: 60h simulation ($\Delta x = 5$ km)

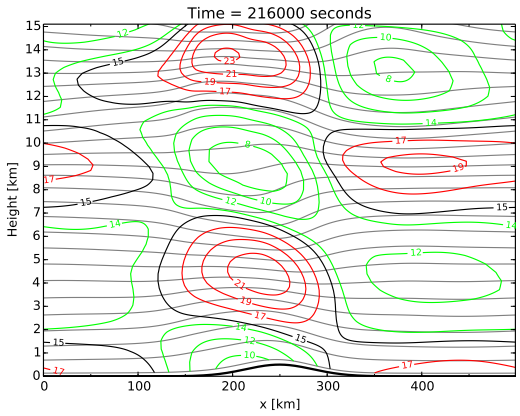


Figure 6: 60h simulation ($\Delta x = 1$ km)

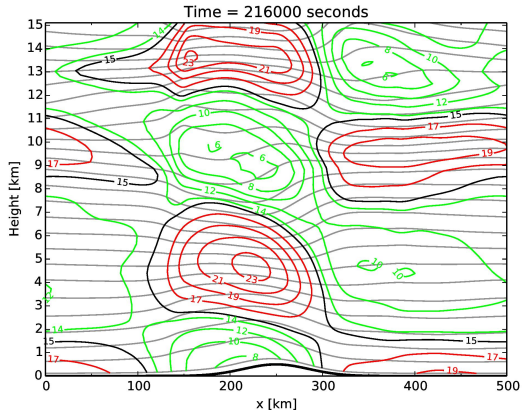


Figure 7: 60h simulation ($\Delta x = 500$ m)

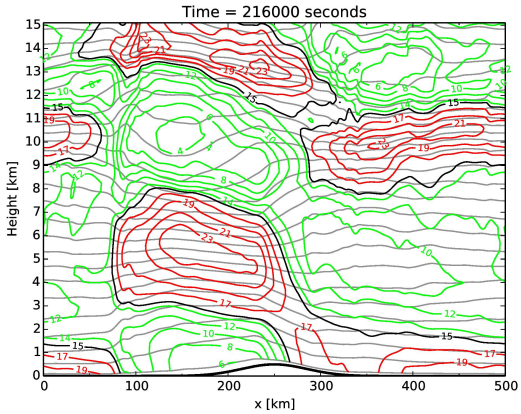


Figure 8: 60h simulation ($\Delta x = 100$ m)

All four simulations capture the gravity wave above the mountain ridge with a wave length of approximately 10 km. The flow regimes of the 1 km, 500 m and 100 m simulations are further able to reflect additional gravity waves east and west of the mountain, which are completely absent in the 5km simulation. Another interesting observation is the different maximum wind speeds above the mountain, the smaller scale simulations suggest 23 m/s where the large scale ones tend to be lower with 21 and 19 m/s, respectively.

Next, we will investigate how those slightly distinctive flow regimes effect the overall precipitation. We will compare the total precipitation of the 5 km, 1 km and the 500 m simulation. To do so, we have to turn on microphysical interactions via the Kessler scheme. While **Isen** implements the Kessler parametrization, it is not as well optimized as the rest of the `Solver` methods. Nevertheless, it is fully parallelized to run efficiently on Euler.

Figure 9 shows the accumulated precipitation of the 5 km, 1km and 500 m simulation. To compare the simulations, we need to down-scale the accumulated precipitation of the 1 km and 500 m simulations to be of the same order as the 5 km simulation. This

is necessary because the simulations have different number of time steps. The 5 km simulation forecasts roughly 50 % more precipitation than the smaller-scale simulations. All three simulations show two local precipitation maxima with the global maxima on the west side of the mountain. The 500 m and 1 km forecasts are much closer to each other and it is likely that their results are closer to what would be observed in nature. If this would indeed be the case, than it would be a legitimate question if the 500 m simulation is worth the increased computation time (4 times longer) or if the 1 km forecast already suffices.

Summary This experiment showed how important it is to investigate the computational boundaries of a model. Small-scale simulations might resolve certain phenomena which are not captured by large-scale simulations and thus the forecasts may lead to vastly different results.

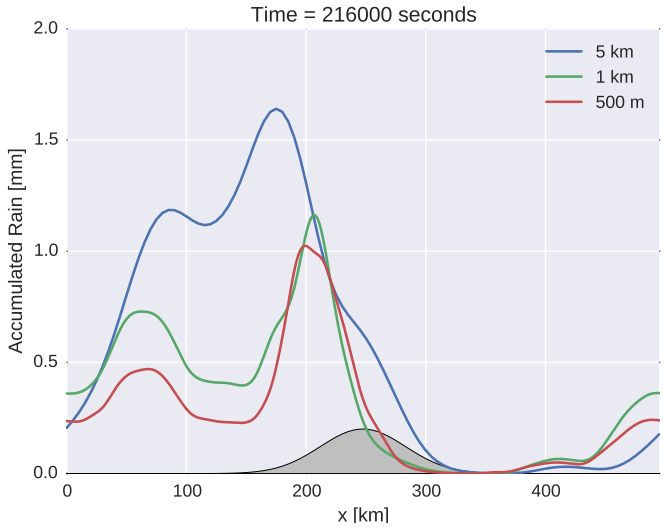


Figure 9: Accumulated precipitation of the 5 km, 1 km and 500 m simulation after 60h. The accumulated precipitation of the 1 km and 500 m simulations are relative to the 5 km simulation.

5 Conclusion

With a simple model such as the isentropic one, consisting of just a handful of prognostic and diagnostic equations paired with a parametrization, we can produce some interesting flow regimes and even forecast precipitation.

In this project we were were able to successfully redesign and implement the model in C++ and optimize it for high-performance usage. Experiments performed with the new model gave some insight on the effect of different spatial resolutions. The short detour in performance modelling showed how tedious it can be to micro optimize the code for peak performance. Supporting an even greater deal of architectures, such as graphics processing units (GPUs), would further increase the code complexity. This shows the importance of domain-specific languages (DSLs) which separate the architecture dependent implementation from the user-code. DSLs are currently under active development and are already employed in a few next generation numerical weather forecast and climte models (see [3] for STELLA in COSMO).

References

[1] D. Abrahams and R. W. Grosse-Kunstleve. *Building Hybrid Systems with Boost.Python*. C/C++ Users Journal, July 2003.

[2] G. Guennebaud, B. Jacob, et al. Eigen v3. `http://eigen.tuxfamily.org`, 2010.

[3] T. Gysi, C. Osuna, O. Fuhrer, M. Bianco, and T. C. Schulthess. *STELLA: A Domain-specific Tool for Structured Grid Methods in Weather and Climate Models*. SC '15 Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2015.

[4] E. Kessler. *On the distribution and continuity of water substance in atmosferic circulations*. Meteorol. Mon. Amer. Meteorol. Soc., 10, 84, 1969.

[5] B. Schling. *The Boost C++ Libraries*. XML Press, 2011.

[6] The OpenMP ARB. The OpenMP API Specification for Parallel Programming. `http://www.openmp.org`. Accessed: 2016-06-19.

[7] Travis CI community. Travis CI. `https://travis-ci.org/`. Accessed: 2016-06-19.