Technical University of Munich

Emulating SpeedyWeather.jl: Applications of SpeedyWeatherEmulator.jl

by Siegl Stefan

Project Report for the module *Dynamical Systems and Machine Learning in Julia*
held by Maximilian Gelbrecht and Alistair White
at TUM in the summer semester 2025

September 9, 2025

Contents

1 Introduction and Motivation

1.1 Introduction

Predictions in climate physics and meteorology are primarily based on solving systems of partial differential equations. In particular, the primitive equations, which include the Navier–Stokes equations, form the basis of general circulation models (GCMs). Solving such complex equations is computationally expensive and requires substantial resources. [1]

In contrast, matrix multiplications - the core operations of neural networks - are comparatively inexpensive. This makes it appealing to approximate the dynamics of the primitive equations with neural networks, thereby training a "weather emulator". The concept is straightforward: data from a dynamical system (here: the atmosphere) are first generated by solving the governing equations, and then used to train an emulator. While this initial data generation and training step is costly, subsequent forecasts with the trained emulator can be produced much more efficiently. [2]

1.2 Motivation

This project report demonstrates this idea in a minimal setup using the self-developed `SpeedyWeatherEmulator.jl` package. The package provides the infrastructure to generate data with `SpeedyWeather.jl` [3] and to define and train the emulator with `Flux.jl` [4]. Detailed documentation, including code applications referenced in this report, are available in the package documentation of `SpeedyWeatherEmulator.jl` [5] under *Examples*.

The aim of this work is to show that simple atmospheric models, in particular the barotropic T5 model with 27 complex spectral coefficients (the corresponding Gaussian grid is illustrated in Figure 1.1), can be approximated by an emulator with reasonable accuracy. This report presents the methodology and results of training such an emulator and evaluates its performance for both generic forecasts and a Rossby–Haurwitz test case.

The theoretical background on dynamical systems, machine learning, and Earth system modelling used in this report follows the lecture notes of Gelbrecht and White [6].

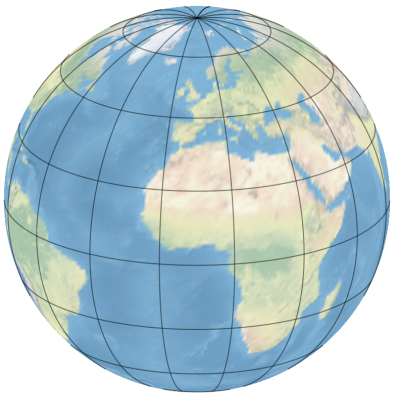


Figure 1.1: The Earth with a simple T5 Gaussian grid. Plot created using `GeoMakie.jl` [7]

2 Methods

2.1 Data Generation

A dataset of vorticity vectors was generated using the `SpeedyWeatherEmulator.jl` package to train the emulator, which generates simulation data using `SpeedyWeather.jl`. The following simulation parameters were chosen for this report:

- `trunc` = 5 (spectral truncation, corresponds to 27 complex spectral coefficients)
- `n_data` = 48 (data points per i.c., corresponds to 48 hours forecast for default `t_step`)
- `n_ic` = 1000 (number of independent initial conditions)

Otherwise, the default values of the package were used. In total, this setup produces 1000 independent trajectories, each with 48 vorticity vectors (after discarding spinup), where consecutive states are separated by one hour. Alternative parameter choices are possible, but the selected values offer a reasonable trade-off between runtime and information gain.

From these raw simulations, training pairs of the form $(x, y) = (\text{vor}(t), \text{vor}(t + \Delta t))$ were constructed. The dataset was split into 70% training, 15% validation, and 15% test data (package default), resulting in about 33,000 training samples and 7,000 each for validation and testing.

2.2 Emulator Architecture

The emulator is implemented as a multi-layer perceptron (MLP) using `Flux.jl` within the package `SpeedyWeatherEmulator.jl`. The emulator takes a vorticity vector of spectral coefficients as input and predicts the corresponding vector one hour ahead, in other words:

$$\text{em}(\text{vor}(t)) = \text{vor}_{\text{em}}(t + \Delta t) \approx \text{vor}(t + \Delta t). \tag{2.1}$$

The structure of the neural network is determined by three parameters: the input/output dimension (`io_dim`), the width of the hidden layers (`hidden_dim`, denoted W), and the number of hidden layers (`n_hidden`, denoted L). The input/output dimension depends on the size of the vorticity vector and therefore on the spectral truncation. For the present work, `io_dim` = 54 (corresponding to the T5) is used throughout.

All hidden layers use as default ReLU activation functions. In `SpeedyWeatherEmulator.jl`, before being passed to the network, the vorticity vectors are normalized using a Z-score transformation, as the spectral coefficients span different orders of magnitude.

Further, the default training setup of `SpeedyWeatherEmulator.jl` was used:

- Batch size: 32
- Epochs: 300
- Initial learning rate: 0.001, reduced by a factor of two every 30 epochs
- Optimizer: AdamW (from `Optimisers.jl`)
- Loss function: Mean Squared Error (MSE)

2.3 Hyperparameter Optimization

In this report, only the depth (L) and their width (W) were optimized, due to time and resource constraints. These parameters define the architecture of the network and have the strongest

influence on the complexity of the model, the accuracy of the emulator, and the training time.

Because training emulators is computationally expensive and only a limited number of configurations can be tested, a two-step optimization strategy was adopted. First, a coarse optimization was performed. Based on these results, a more targeted fine search was conducted (see Section 3.1).

The quality of each hyperparameter configuration was assessed using the mean relative error across different forecast horizons and the corresponding emulator training time.

3 Results

3.1 Hyperparameter Analysis

As described in Section 2.3, the number of hidden layers (L) and the dimension of the hidden layers (W) were optimized in two stages (coarse and fine optimization). Furthermore, the parameter count of each emulator network was taken as a direct proxy for training time (see Appendix A for details).

3.1.1 Coarse Optimization

For the coarse optimization, $L \in \{1, 2, 3\}$ and $W \in \{64, 128, 256, 512\}$ were tested. Figure 3.1 shows the validation error for the different configurations and horizons (forecast lengths):

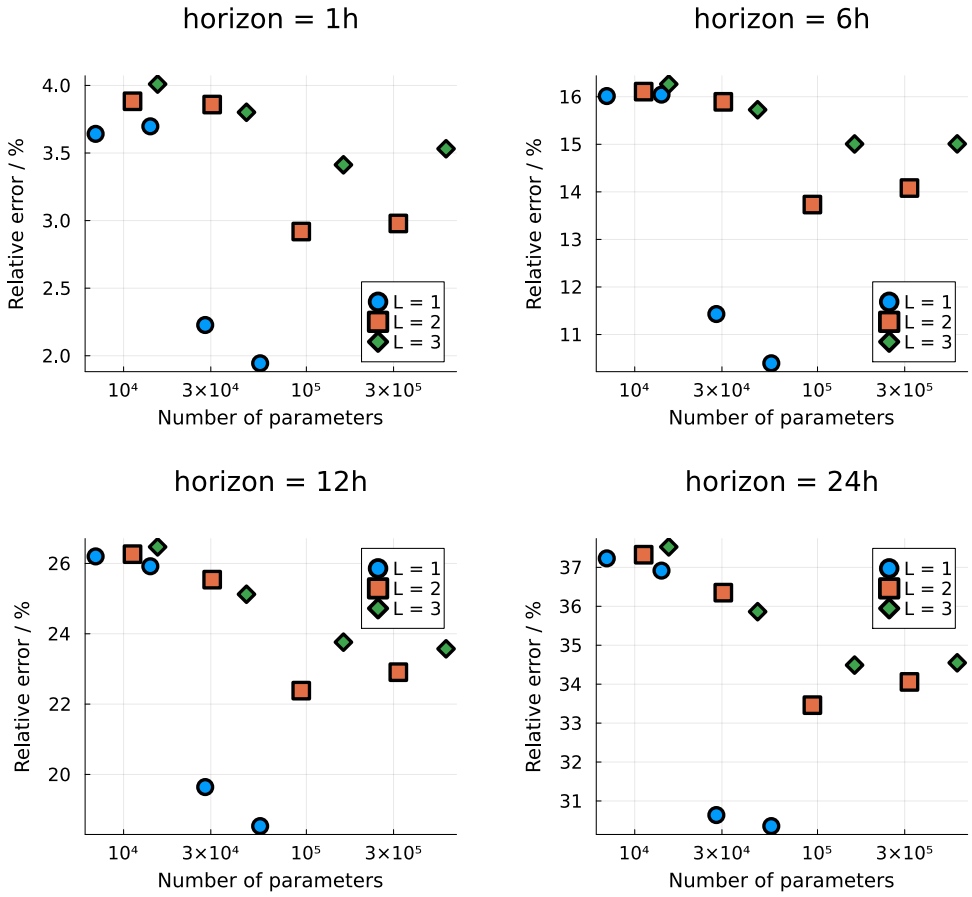


Figure 3.1: Validation error of the emulator for different network depths L and widths W during coarse hyperparameter optimization, shown for different forecast horizons. For fixed L, W increases from left to right.

It is evident that $L = 1$ achieves the best performance while also requiring the least training time. Furthermore, for $L = 1$, larger values of W consistently improve the emulator quality.

3.1.2 Fine Optimization

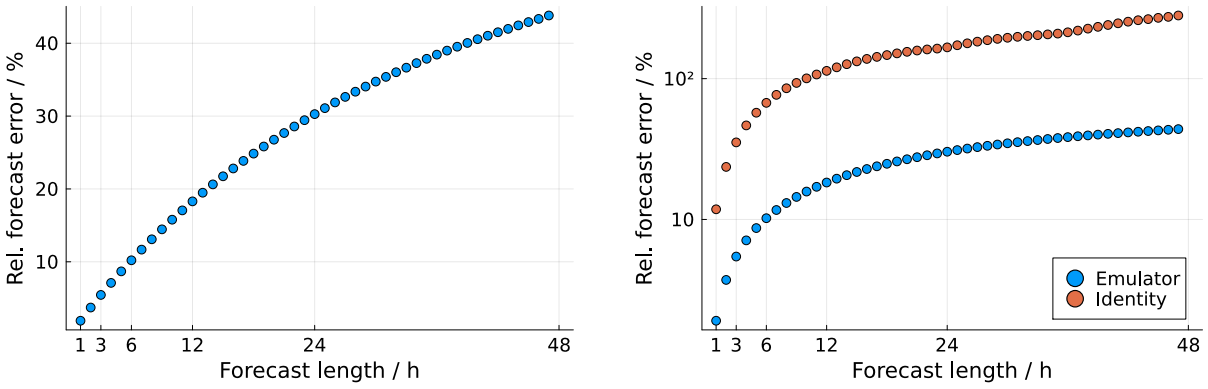
Based on these results, the fine optimization was restricted to L = 1 and W ∈ {256, 384, 512, 640, 768, 896, 1024, 1280, 1536, 2048}.

For the chosen values of W (except 256), the differences in emulator performance are marginal (below 1%). While certain configurations appear to perform better at specific forecast horizons, these variations may also be attributable to statistical noise. Overall, W = 640 represents a good compromise between emulator accuracy and training time. (See package documentation for fine optimization plot similar to Figure 3.1).

As a result, the final architecture selected for subsequent experiments was L = 1 and W = 640.

3.2 Long Forecast Quality

Using the optimal hyperparameters determined in Section 3.1, the performance of the emulator was further assessed for longer forecast horizons. Figure 3.2a shows the mean relative error as a function of forecast length. As expected, the error increases with longer horizons:



(a) Forecast error as a function of forecast length.

(b) Comparison with identity baseline (log scale).

Figure 3.2: Emulator performance at long forecast horizons. (a) Mean relative error over 48 hours. (b) Comparison to identity baseline.

The relative forecast error remains below 10% during the first six hours, rises to about 30% at 24 hours, and reaches around 45% after 48 hours. Figure 3.2b further compares the emulator error (same data as in Figure 3.2a) with the relative error of the unchanged (identity emulator) vorticity vectors in logarithmic scale. This demonstrates that the emulator clearly outperforms the identity baseline and captures meaningful dynamical evolution.

Appendix B contains an additional vorticity heatmap plot demonstrating the ability of the emulator to preserve structure. Appendix C shows the emulator applied to the Rossby–Haurwitz wave, illustrating that the emulator is not able to sufficiently emulate this case.

4 Discussion and Outlook

4.1 Discussion

The hyperparameter optimization in Section 3.1 showed that shallow networks (low L) with a relatively large number of neurons (high W) achieve the best emulator performance. In practice, a single hidden layer with sufficient width is already flexible enough to approximate the dynamics, while additional depth mainly increases training cost without clear benefits.

Furthermore, the results in Section 3.2 demonstrate that the forecast error increases monotonically with forecast length. A relative error of 10% is reached only after about six hours. Additional heatmap plots (Appendix B) confirm that the emulator preserves coherent structures, although these gradually fade over time.

The limitations are also evident. For longer forecasts, such as 48 hours, the relative error already exceeds 40%. This is expected, as atmospheric dynamics governed by the nonlinear primitive equations are chaotic and inevitably limit predictability. Moreover, the emulator was trained only on trajectories generated from random initial conditions, restricting it to one specific attractor. It therefore fails on artificial setups such as Rossby–Haurwitz waves (see Appendix C), underlining the limited generalization ability.

4.2 Outlook

A first natural extension is to move beyond the very coarse T5 grid. Higher spectral truncations would provide more realistic dynamics but also substantially increase the number of spectral coefficients, which grow quadratically with truncation. According to Equation A.1, this directly translates into a quadratic growth of the emulator's parameter count n, and thus into quadratic growth of training time. In addition, larger and potentially deeper networks would likely be required to capture the added complexity, which would further increase the parameter count and thus the training time.

Another direction of improvement is to stabilize the emulator against the observed fading of vorticity fields. Approaches that enforce physical invariants could help maintain the long-term coherence of the dynamics and prevent fading. [1]

Finally, the dataset itself could be extended. While forecasts were limited to 48 hours in this report, longer horizons would be possible once the emulator is further improved. Such experiments would provide a more complete picture of its practical usefulness. Another possible extension would be the use of a parametric estimator, which could facilitate systematic parameter tuning.

Appendix A

Training Time vs. Neural Network Parameters

The total number of trainable parameters of a neural network can be calculated using the formula

$$n = (d \cdot W + W) + (L - 1)(W \cdot W + W) + (W \cdot d + d), \tag{A.1}$$

where d is the input/output dimension, L is the depth and W is the width of the neural network. (This comes from counting weights and biases of each layer)

Figure A.1 shows training time plotted against the number of parameters of the different emulator networks:

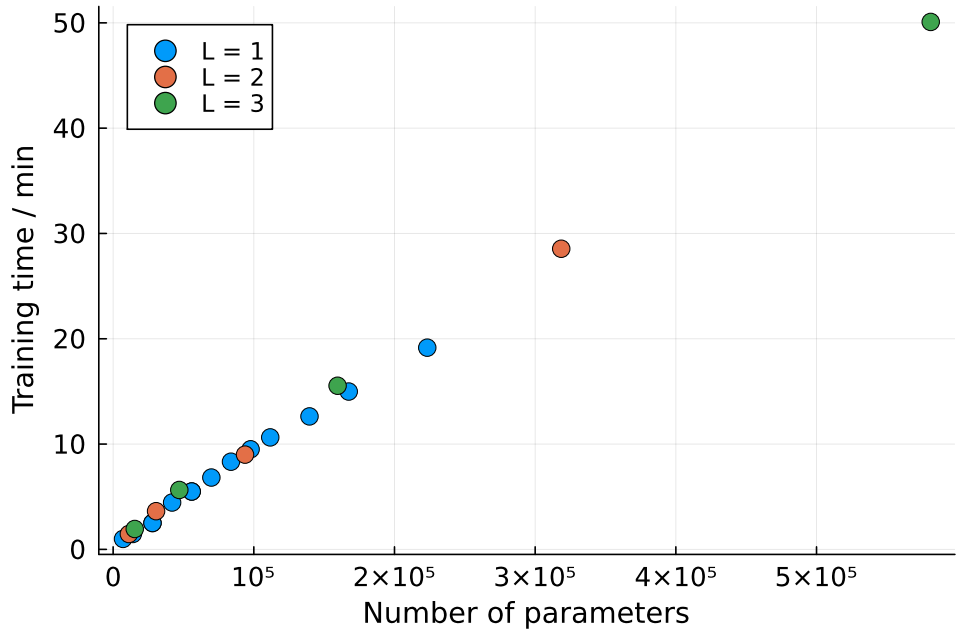


Figure A.1: Training time of the emulator as a function of the number of network parameters for depths $L \in \{1, 2, 3\}$. A clear linear correlation is observed, confirming that parameter count is a reliable proxy for training time.

The number of parameters was computed analytically using Equation A.1. A clear linear correlation for various L and W is observed, which demonstrates that parameter count can be used as a reliable proxy for training time.

Appendix B

Long Forecast Heatmaps

Figure B.1 shows vorticity heatmaps at different forecast horizons, complementing the quantitative error measures in Section 3.2 and illustrating that the emulator preserves the overall structure of the vorticity fields, although the patterns gradually fade with increasing forecast time.

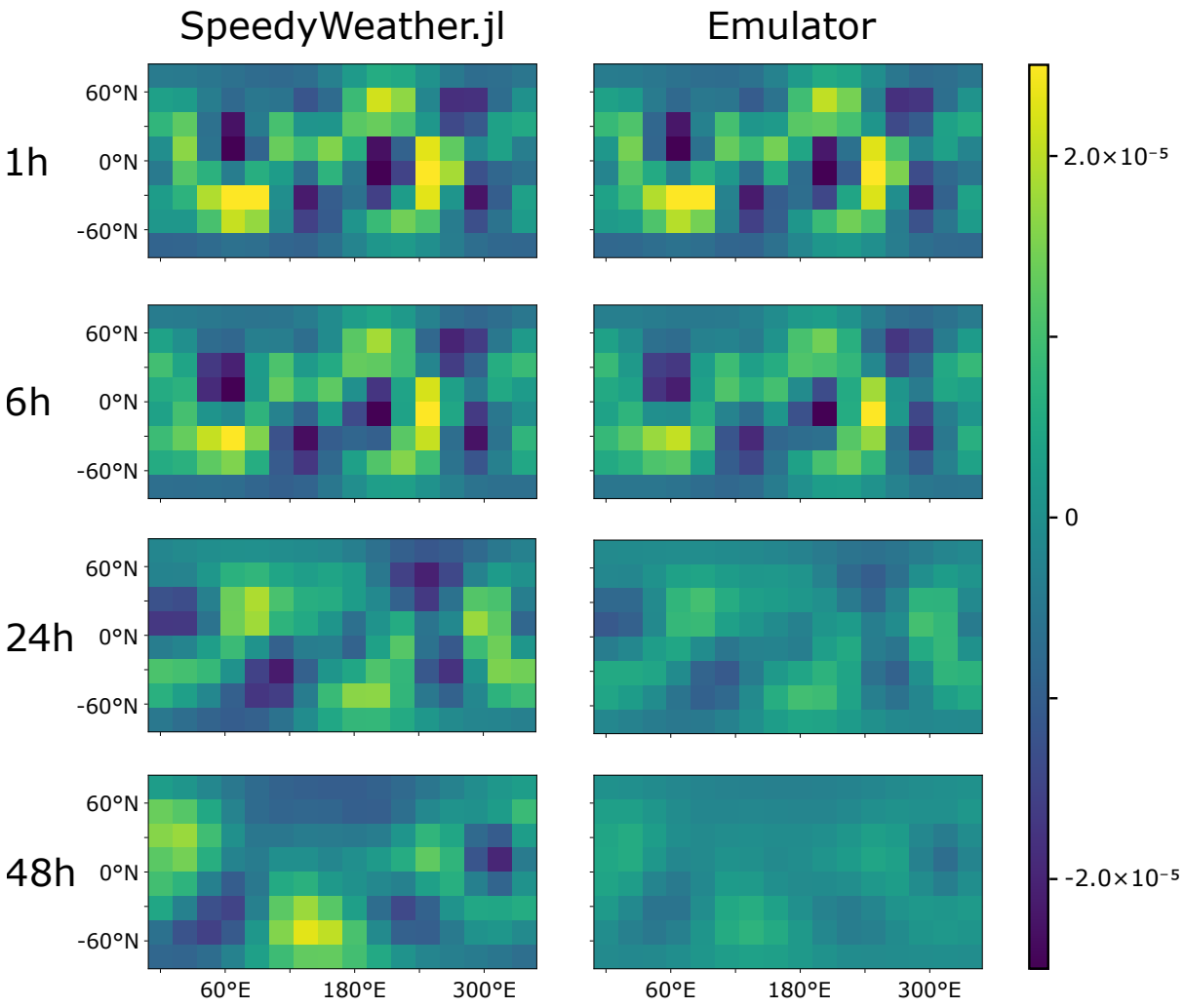


Figure B.1: Vorticity heatmaps at different forecast horizons (1h, 6h, 24h, 48h). The emulator preserves the overall structure of the vorticity fields compared to `SpeedyWeather.jl`, but the patterns gradually fade with increasing forecast time. Color scale corresponds to relative vorticity in 1/s. The individual heatmap plots were created with `CairoMakie` and combined using Inkscape.

Appendix C

Rossby–Haurwitz Wave Test

The Rossby–Haurwitz wave [8] provides a classical test case for atmospheric models and is defined by a special initial condition

$$\zeta(\lambda, \theta) = 2\omega \sin(\theta) - K \sin(\theta) \cos(\theta)^m \cdot (m^2 + 3m + 2) \cos(m\lambda), \tag{C.1}$$

with longitude λ and latitude θ and parameters $m = 4$ (zonal wavenumber), $\omega = 7.848 \cdot 10^{-6}$ s^{-1} and $K = 7.848 \cdot 10^{-6}$ s^{-1} (frequencies).

It propagates in a quasi-stable manner around the globe and therefore serves as a useful benchmark for emulator performance.

C.1 Forecast Length

Figure C.1 shows the forecast error as a function of forecast time, analogous to Figure 3.2b, and demonstrates that the relative error of the emulator for Rossby-Haurwitz wave data is very large compared to the usual random initial conditions testing data.

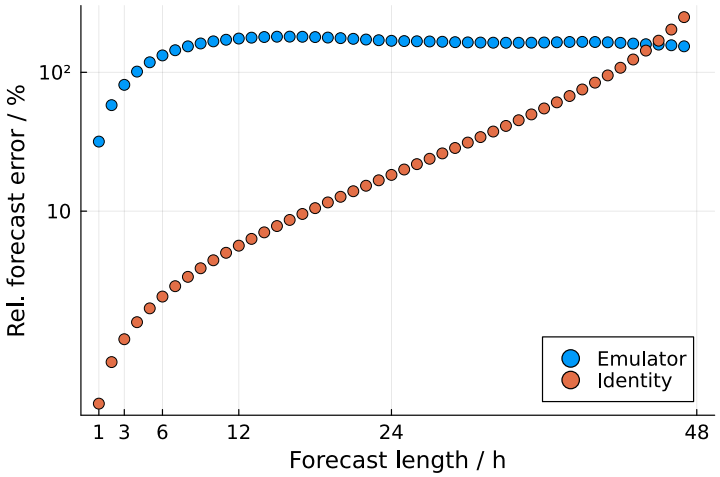


Figure C.1: Forecast error for the Rossby–Haurwitz wave as a function of forecast length. The relative error of the emulator is very large compared to the identity baseline, illustrating its failure to reproduce this artificial initial condition.

C.2 Heatmap Plots

Figure C.2 provides vorticity heatmaps at different forecast horizons, illustrating the inability of `SpeedyWeather.jl` to reproduce the Rossby-Haurwitz wave for low truncations. (High truncations, as `trunc` = 43, lead to stable solutions in `SpeedyWeather.jl`, as demonstrated in the package documentation). Further, the emulator captures the basic structure, but fades away quickly.

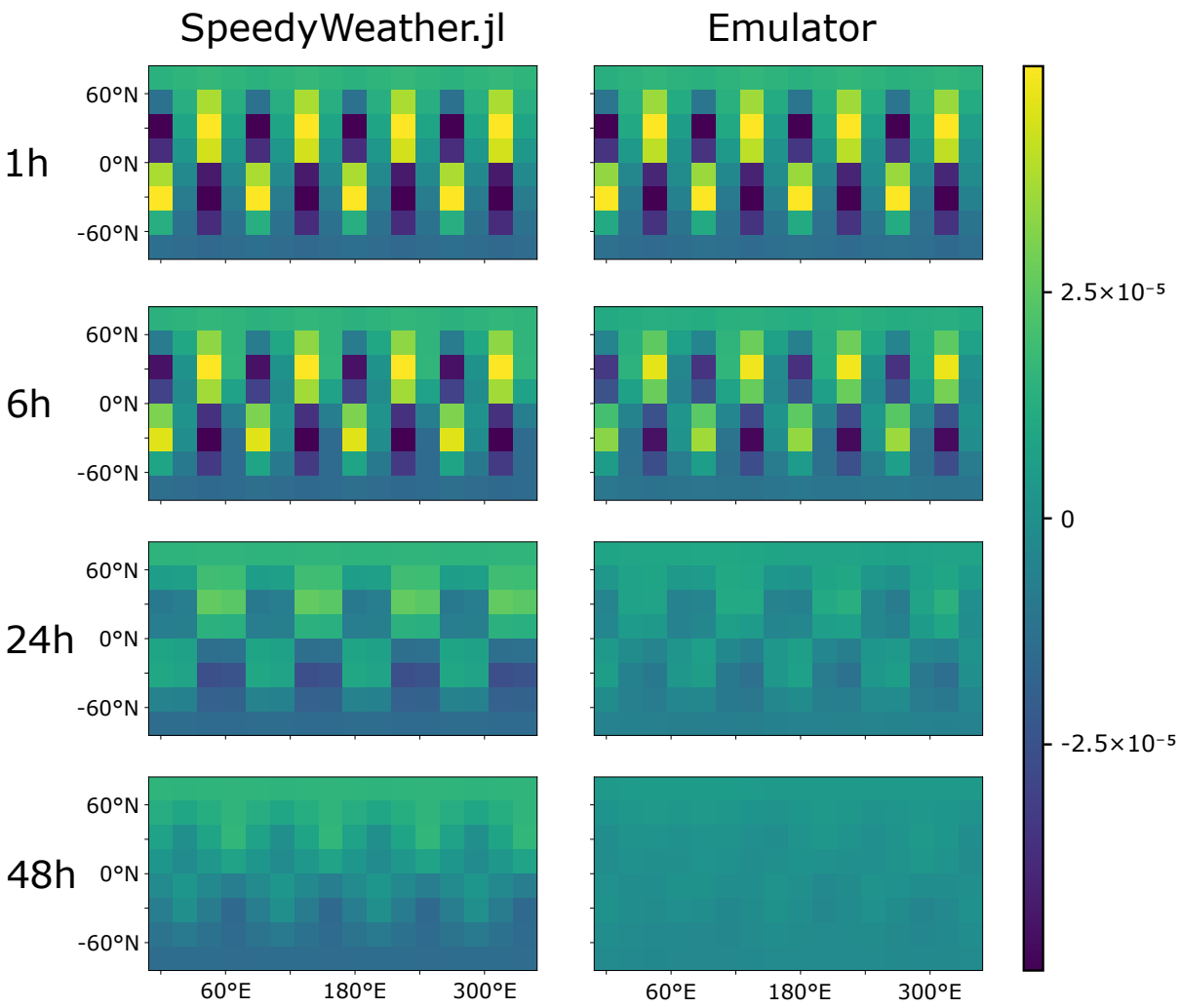


Figure C.2: Vorticity heatmaps for the Rossby–Haurwitz wave at different forecast horizons (1h, 6h, 24h, 48h). `SpeedyWeather.jl` cannot reproduce the stability of Rossby-Haurwitz waves for low truncations. The emulator captures the structure, but fades away quickly. Color scale corresponds to relative vorticity in 1/s. The individual heatmap plots were created with `CairoMakie` and combined using Inkscape.

Bibliography

[1] Sebastian Bathiany. Lecture Slides of "Earth System Modelling". Lecture notes, Technical University of Munich, 2025. Unpublished teaching material.

[2] Nikola B. Kovachki, Zongyi Li, Burigede Liu, Kamyar Azizzadenesheli, Kaushik Bhattacharya, Andrew M. Stuart, and Anima Anandkumar. Neural operator: Learning maps between function spaces. *CoRR*, abs/2108.08481, 2021. URL https://arxiv.org/abs/2108.08481.

[3] Milan Klöwer, Maximilian Gelbrecht, Daisuke Hotta, Justin Willmert, Simone Silvestri, Gregory L. Wagner, Alistair White, Sam Hatfield, Tom Kimpson, Navid C. Constantinou, and Chris Hill. SpeedyWeather.jl: Reinventing atmospheric general circulation models towards interactivity and extensibility. *Journal of Open Source Software*, 9(98):6323, 2024. doi: 10.21105/joss.06323. URL https://doi.org/10.21105/joss.06323.

[4] Mike Innes. Flux: Elegant machine learning with julia. *Journal of Open Source Software*, 3 (25):602, 2018. doi: 10.21105/joss.00602. URL https://doi.org/10.21105/joss.00602.

[5] Stefan Siegl. Speedyweatheremulator.jl: Emulator framework for speedyweather.jl. https://github.com/SieglStefan/SpeedyWeatherEmulator.jl, 2025. Julia package with documentation at https://SieglStefan.github.io/SpeedyWeatherEmulator.jl.

[6] Maximilian Gelbrecht and Alistair White. Lecture Notes of "Dynamical Systems and Machine Learning in Julia". Lecture notes, Technical University of Munich, 2025. Unpublished teaching material.

[7] Simon Danisch and Julius Krumbiegel. Makie.jl: Flexible high-performance data visualization for julia. *Journal of Open Source Software*, 6(65):3349, 2021. doi: 10.21105/joss.03349. URL https://doi.org/10.21105/joss.03349.

[8] David L. Williamson, John B. Drake, James J. Hack, Rüdiger Jakob, and Paul N. Swarztrauber. A standard test set for numerical approximations to the shallow water equations in spherical geometry. *Journal of Computational Physics*, 102(1):211–224, 1992. ISSN 0021-9991. doi: https://doi.org/10.1016/S0021-9991(05)80016-6. URL https://www.sciencedirect.com/science/article/pii/S0021999105800166.