We would like to thank the reviewers for careful consideration of our work and for providing thoughtful feedback that has improved the quality of the manuscript in many ways, including the documentation, the detail and value of the examples, and the clarity of the discussion about automatic differentiation. We have numbered the comments from each reviewer below. Responses are in blue with quoted text from the new tracked-changes manuscript in *italics* with associated line numbers.

Reviewer 1

1. General: given the similarities and the shared foundation of SPEEDY, I think it would be good to cite the SpeedyWeather.jl project somewhere (I am not an author of this model incidentally).

   Thank you for pointing out this oversight. We do see SpeedyWeather as an inspiration for this work and should have included a formal citation.

   Updated text (lines 67–69): *"This work builds upon a growing movement to generate modular, differentiable Earth system components in modern programming languages with GPU acceleration (Klöwer et al., 2024; Häfner et al., 2018; Fang et al., 2024; Wagner et al., 2025)."*

2. Section 2.1, Dynamical Core: could you say a word about the limits of scalability of the JCM dycore? What determines the limit of T425? Is it the memory of the current generation of accelerators? Would it be feasible to do domain decomposition like traditional models, in order to run on distributed memory clusters, and therefore access higher resolutions? Or was it decided a priori that JCM will not target those resolutions for simplicity?

   Thanks for the question, we have clarified the existing technical obstacles and the fact that the current configuration is intended for global studies.

   Added text (lines 89–93): *"There are no technical obstacles to adding support for resolutions beyond T425, though other features may be of higher priority for high-resolution studies, such as support for more vertical levels or for a non-spectral dynamical core. (Spectral integration schemes must simulate the entire globe, making them inefficient for region-focused experiments, and the spectral transforms required at each timestep eventually bottleneck performance as they scale worse than linearly with the number of grid points.)"*

3. Line 112: typo - "parametrization" or "parameterization".

   Updated text (lines 118–119): *parameterization*

4. Section 3, Model Interface (API): nothing wrong with code listings, but it's neater to treat them like any other figure and make an explicit reference to them in the text, e.g. "Figure X gives an example usage of JCM".

Code listings have been converted to figures, and references to them have been added. In Section 3:

Added text (lines 171–172): *Example code for running the model is shown in Figure 1.*

In Section 6.3:

Added text (lines 337–339): *The code block in Figure 9 implements the described experiment, calculating the gradient of f(*longwave_rad.ftop*) with respect to the open-ocean surface albedo* Parameters.albsea *using JVP (*jax.jvp*).*

5. Section 5, Performance: I would recommend presenting all computational speed results in common units of e.g. "simulated years (or days) per day" (SYPD/SDPD). This gives a nice intuitive number which summarises what the user cares about.

Original Text: "Figure 5 shows the total time to run a default configuration of JCM at T85 (approx. 150km resolution at the Equator) using either a laptop CPU, laptop GPU (NVIDIA RTX 4050), or Google Cloud TPU. The behavior in the figure has been our typical experience: runtime on GPUs and TPUs is at least an order of magnitude faster than on CPUs (except for Apple Silicon chips), with parallelization (sharding the model state arrays so they can be distributed among multiple devices) giving a significant speedup to the TPU runs on Google Cloud. For this set of parameters, the laptop GPU was faster than the cloud TPUs, but initial testing indicates that the TPU (like any high-performance cloud platform) will scale better for higher resolutions than the laptop GPU with its limited VRAM. The platforms used here do not all have the same number of cores or clock speeds; the goal is to illustrate typical runtime on common devices.

As another point of comparison, JCM on the RTX 4050 laptop GPU was about 40% faster per year of model time than speedy.f90 on the same laptop's Intel i9 13th Gen CPU.

[...]

With regards to the model run configuration itself, the number of operations for a model run typically increases with the square of the spectral truncation, with an additional factor for reductions to the model timestep (sometimes needed for stability at a higher resolution). However, runtime is observed to scale better than the number of operations, presumably due to parallelization. Figure 6 shows how model runtime scales with the spatial resolution of the model grid, using as a benchmark a one year simulation with default 30 minute time step, run on a NVIDIA RTX 4050 laptop GPU. At T31 resolution, one year of model time takes only a minute and a half, and even T119 can be run for a year in less than 10 minutes."

Updated Text (lines 223–240): *"Figure 6 shows the simulation speed, in units of simulated years per day (SYPD), of running JCM at various resolutions and on various device types. The behavior in the figure has been our typical experience: runtime on GPUs and TPUs is significantly faster than on CPUs, with parallelization (sharding the model state arrays so they can be distributed among multiple devices) giving an additional significant speedup to TPU runs on Google Cloud (parallelization was not used for the results in Figure 6). The dashed line in the figure represents the theoretical quadratic scaling of number of operations (and memory requirement) with spectral truncation.*

[...]
It should be noted that in practice, the runtime for the higher resolutions may pick up another factor of 2 or 3 relative to lower resolutions due to reductions to the model timestep that are eventually needed for stability. For the results shown in Figure 6, the timestep was left as the default 30 minutes for all resolutions."

6. Figure 5: the presentation of the benchmark here is a little unusual. The graph shows elapsed time vs. simulation time. I would imagine that all of these lines are in fact straight when plotted on a linear scale, indicating that the wall time per step is constant, assuming each step is identical (e.g. no I/O on every nth step). Then, why bother with a graph at all? All that matters is the gradient - the speed of the simulation in simulated-things-per-actualthing, which can just be presented in a table. I would recommend that approach.

   By putting performance results in terms of simulated years per day, the new Figure 6 covers this information, so Figure 5 has been removed entirely.

7. Figure 6: could you mention in the caption what hardware you used?

   Updated Figure 6 caption: *"Model performance on different hardware for a default JCM run, 30 minute timestep. To generate this plot, runs of 1 simulated year were conducted on 3 types of Google Cloud virtual machines: a CPU VM (with 32x Intel Xeon Platinum 8581C), a GPU VM (with a 16GB Tesla P100 PCIe), and a TPU VM (with a Google Cloud TPU v5 lite). These devices were chosen as comparable based on having similar pricing (approx. $1.50 / hour) at time of writing."*

8. Line 241: I would be a bit more careful in wording here. ECMWF's IFS has hand-coded tangent-linear and adjoint for its entire atmospheric model, and has maintained them for 25 years now. Maintaining these parallel codes is of course a substantial burden, but far from "intractable".

   Thank you for the advice, we have updated the text accordingly.

   Updated Text (lines 250–252): *"Given the complexity of state-of-the-art Fortran models, this manual generation of gradients is expensive to implement and maintain. Thus, in most cases, gradient information is inaccessible or presents a substantial burden to model developers."*

9. Section 6.1, Automatic-differentiation (VJP and JVP): personally I would find this explanation much easier with a few equations. The verbal description left a few unanswered questions for me. For example, what is the mathematical difference between VJP and JVP? If I'm understanding the text correctly, one is simply the transpose of the other.

   This is a good question and we have updated the text to be more clear about the mathematical equivalence. While the TLM and adjoint operators are transposes of each other, the operations JVP and VJP are not. Importantly, VJP and JVP do not evaluate the whole Jacobian (it would be very expensive), they evaluate the relevant row or column.

Updated Text (lines 253–264): *"JAX provides JCM with its AD capabilities (Bradbury et al., 2018). In JAX, gradients can be computed in either forward (tangent linear) or reverse (adjoint) mode. Each mode has multiple use cases and can be applied to a wide variety of science problems. We use F(x) to denote the tangent linear model of a function F(x). The relationship of F(x) to the adjoint $F^T(x)$, where both are linearized about x, is*

$$\langle F(x)v, w \rangle = \langle v, F^T(x)w \rangle , \quad (1)$$

where v is the tangent vector, w is the co-tangent vector, and $\langle \cdot \rangle$ denotes the inner product. Forward-mode differentiation is applied through the Jacobian vector product (JVP) and is equivalent to F(x)v. This evaluates the multiplication of a user-specified tangent vector, v, with the tangent linear model, F(x). Reverse-mode differentiation evaluates a vector-Jacobian product (VJP), equivalent to $F^T(x)w$. This multiplies a user-specified co-tangent vector, w, with the AD-derived adjoint, $F^T(x)$, of the function F(x). Both VJP and JVP avoid computing the entire Jacobian of the function to minimize computational cost, but they can be used to reconstruct it if necessary. Example applications are discussed in more detail in the following sections."

Also, when you say such and such is "computationally efficient" under certain conditions, what is meant exactly?

Original Text: "VJP is computationally efficient when the output space of the function is much smaller than the input space… In contrast to VJP, JVP is computationally efficient when the output dimension is larger than or equal to the input dimension of a function."

Updated Text (lines 265–272): *"In large-scale geophysical models integrated over long time windows, JVP typically costs approximately 1–2 times the cost of a nonlinear forward model integration. In contrast, VJP often costs approximately 3–6 times the cost of the forward integration, depending on memory constraints, checkpointing strategy, and solver structure (Heimbach et al., 2005; Evensen et al., 2020). Reverse-mode differentiation is more computationally efficient than forward mode for gradient evaluation when the output dimension of the function of interest is much smaller than the input dimension (e.g., a scalar loss function). In this case, a single VJP evaluation produces the full gradient, whereas forward mode would require n separate JVP evaluations, where n is the input dimension. Conversely, forward-mode differentiation is more efficient when the number of inputs is much smaller than the number of outputs."*

10. Section 6.2, Calibration: could you be specific about what "statistics" you compute in this example?

    We updated the calibration example to be over two parameters (see Reviewer 2, question 6). We have also added the following sentence to clarify the statistics used in this problem

Updated Text (lines 285–288): "*We generate synthetic observed statistics by running the JCM in the default aquaplanet configuration, which has values of stratiform cloud albedo = 0.5 and relative humidity threshold = 0.3. The calculated statistics are the spatial and temporal average of each model output variable over a 5-day model run."*

11. Line 311-: again, this code listing would be better as a figure, though this one at least has a reference in the text. Why does the line number start at 13 by the way?

    The code listing has been converted to a figure, and code has been added to make it self-contained (the line number previously began at 13 because the code used variables from the prior code listing).

Reviewer 2

This paper describes the JAX Circulation Model (JCM), a differentiable atmospheric model written in Python using JAX. The paper is well written. I downloaded the code, and it was easy to follow the documentation and run the code on a laptop. I would be happy to accept the paper subject to some minor suggestions.

1. Line 152: Could you explain a bit more what the orographic corrections are?

   Original Text: "JCM does not yet implement orographic corrections in Version 1.0, but this is a subject of future work. The absence of these corrections is one of the primary differences between the JAX and Fortran implementations, in addition to the differences in horizontal diffusion."

   Updated Text (lines 160–164): *"Speedy.f90 includes corrections to the temperature and humidity fields that are not described in the SPEEDY documentation. These corrections appear to adjust near-surface temperature and humidity profiles to account for subgrid-scale topography. Version 1.1 of the JCM does not yet implement these corrections, but their inclusion is a subject of future work. The absence of these corrections is one of the ways in which the JAX implementation deviates from the Fortran 90 version."*

2. Line 186: Is the RMS difference using the monthly data, or the average of the three-year simulation?

   We are using the RMS difference of the 3-year simulations and have clarified with the following statement.

   Updated Text (line 184): *"The root-mean-square (RMS) difference of the 3-year simulations is shown in the bottom row."*

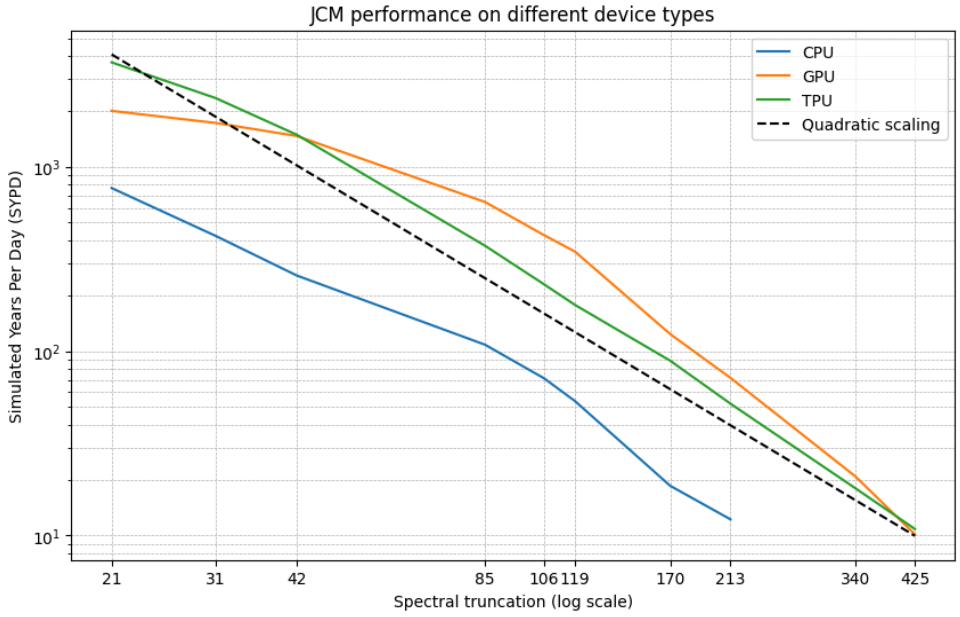
3. Lines 189 - 198 and Figures 2 and 3: The differences in zonal wind between JCM and speedy look pretty large to me. Could you explain more about how the differences in horizontal diffusion and orographic corrections result in these changes?

To the best of our understanding, differences in zonal wind are primarily due to the fact that we have a different dynamical core (Dinosaur), integration method (Runge-Kutta), and horizontal diffusion (explicit instead of implicit and different order). We have done some tuning to attempt to bring the models closer together, but we have stabilized at the current version, which is similar but not exactly the same. While we do want to match the Fortran (and we have verified that the parameterizations give the same output), we are not prioritizing reproducing SPEEDY exactly. Instead we are using SPEEDY as a starting point for an intermediate-complexity model. We have updated the text to clarify these points.

Updated Text (lines 190–198): *"The discrepancy between the zonal wind in speedy.f90 and JCM is likely due to differences in the implementation of the dynamical cores, time stepping, and horizontal diffusion. The JCM parameterizations have been tested against speedy.f90 to verify that they produce the same output. This does not include horizontal diffusion, which in speedy.f90 is implemented implicitly in coordination with the leap frog time stepping. Although the general formulation of the spectral dynamical cores is the same, they are not identical and the additional choices to use Runge-Kutta and explicit horizontal diffusion in the JCM lead to differing results. Speedy.f90 also implements additional linear drag on the mean vorticity and divergence in the stratosphere (this drag is undocumented and thus currently excluded from the JCM). JCM v1.1 has been tuned to reduce these differences where possible. While we aim to match the qualitative climate of the Fortran, we are not prioritizing reproducing speed.f90 exactly, and instead use it as a starting point for an intermediate-complexity model."*

4. Figures 5 and 6: I think it would be clearer to present performance metrics as SYPD (simulated years per day). It would also make it easier to compare with other models.

Figures 5 and 6 have been replaced with a single figure showing SYPD:



Original Text: "Figure 5 shows the total time to run a default configuration of JCM at T85 (approx. 150km resolution at the Equator) using either a laptop CPU, laptop GPU (NVIDIA RTX 4050), or Google Cloud TPU. The behavior in the figure has been our typical experience: runtime on GPUs and TPUs is at least an order of magnitude faster than on CPUs (except for Apple Silicon chips), with parallelization (sharding the model state arrays so they can be distributed among multiple devices) giving a significant speedup to the TPU runs on Google Cloud. For this set of parameters, the laptop GPU was faster than the cloud TPUs, but initial testing indicates that the TPU (like any high-performance cloud platform) will scale better for higher resolutions than the laptop GPU with its limited VRAM. The platforms used here do not all have the same number of cores or clock speeds; the goal is to illustrate typical runtime on common devices.

As another point of comparison, JCM on the RTX 4050 laptop GPU was about 40% faster per year of model time than speedy.f90 on the same laptop's Intel i9 13th Gen CPU.

[...]

With regards to the model run configuration itself, the number of operations for a model run typically increases with the square of the spectral truncation, with an additional factor for reductions to the model timestep (sometimes needed for stability at a higher resolution). However, runtime is observed to scale better than the number of operations, presumably due to parallelization. Figure 6 shows how model runtime scales with the spatial resolution of the model grid, using as a benchmark a one year simulation with default 30 minute time step, run on a NVIDIA RTX 4050 laptop GPU. At T31 resolution, one year of model time takes only a minute and a half, and even T119 can be run for a year in less than 10 minutes."

Updated Text (lines 223–240): *"Figure 6 shows the simulation speed, in units of simulated years per day (SYPD), of running JCM at various resolutions and on various device types. The behavior in the figure has been our typical experience: runtime on GPUs and TPUs is significantly faster than on CPUs, with parallelization (sharding the model state arrays so they can be distributed among multiple devices) giving an additional significant speedup to TPU runs on Google Cloud (parallelization was not used for the results in Figure 6). The dashed line in the figure represents the theoretical quadratic scaling of number of operations (and memory requirement) with spectral truncation.*

[...]

It should be noted that in practice, the runtime for the higher resolutions may pick up another factor of 2 or 3 relative to lower resolutions due to reductions to the model timestep that are eventually needed for stability. For the results shown in Figure 6, the timestep was left as the default 30 minutes for all resolutions."

5. Line 256: I don't think this is accurate. For example, ensemble methods that are gradient-free are commonly used in data assimilation in weather forecasting. They can be used for tuning parameters, too.

   Original Text: "Without gradient information, this process often involves hand-tuning by domain experts. Hand-tuning is limited in the degree to which it utilizes observational information and leaves much of the parameter space unexplored"

   Updated Text (lines 276–279): *"Without gradient information, this process involves ensemble methods and hand-tuning by domain experts. Hand-tuning is limited in the degree*

to which it utilizes observational information and leaves much of the parameter space unexplored. Ensemble methods can be effective, but we often lack comparisons of these approximations against gradient-based approaches. Such comparisons are a focus for future work with the JCM."

6. Section 6.2: I was hoping the authors could show a calibration of more parameters, especially as it is mentioned that "Calibration with gradient information allows for the assessment of model behaviour across more of the parameter space". I understand if it is beyond the scope of this paper, though.

   Although a truly complex parameter estimation example is outside the scope of this manuscript, we have updated the manuscript with a 2-parameter example. In this example we now optimize over both parameters at once and include a visualization of the 2-dimensional loss landscape.

   Updated Figure 7:

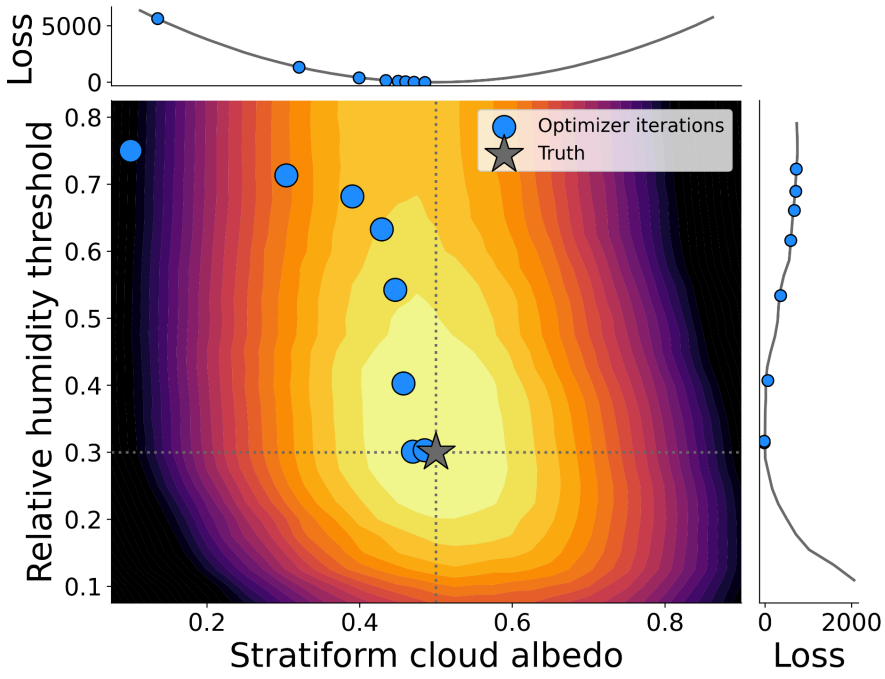


Figure 7 Caption: *Visualisation of the 2-D loss landscape for the described parameter estimation example. The optimization begins with values of 0.1 for stratiform cloud albedo and 0.75 for relative humidity threshold. The optimal value is successfully found near the true SPEEDY values of 0.5 and 0.3, respectively. Blue dots show the calculated loss at each iteration of gradient descent, and the grey star is the ``true'' value. Curves at the top and right sides of the contour plot show slices of the loss function obtained by varying the parameters independently.*

Updated Text (lines 282–291): *"A popular approach for ESM calibration is to estimate model parameters against climate statistics (Kennedy and O'Hagan, 2001; Schneider et al., 2017, 2024). The optimization problem is formulated to minimize a loss defined by the misfit between observed and modeled statistics. In order to illustrate how JCM can be calibrated using AD, we show an idealized example that simultaneously estimates parameter values for stratiform cloud albedo and relative humidity threshold. We generate synthetic observed statistics by running the JCM in the default aquaplanet configuration, which has values of stratiform cloud albedo = 0.5 and relative humidity threshold = 0.3. The calculated statistics are the spatial and temporal averages of each model output variable over a 5-day model run. We initialize the starting values of stratiform cloud albedo and relative humidity threshold to be 0.1 and 0.75, respectively, and then optimize over the parameter space to find values that most closely reproduce the synthetic statistics. The observations denoted by y in Equation 2 correspond to the simulated statistics. This example can be adapted or extended to use any climate statistics and any set of parameters."*

(cont. lines 300–304): *"…where x_k denotes the vector of parameter values at iteration k, $\boldsymbol{\alpha}$ denotes the step size, and the search direction is the negative gradient of the loss with respect to x_k . Figure 7 shows the calculated loss at each step of the optimization (blue dots) and the true parameter values (grey star). It can be seen that the optimizer succeeds at identifying the values that generated the simulated observations. We calculate the loss across the possible range for both parameters and include a visualization of the 2-dimensional loss landscape, where light colors signify lower cost."*

7. Line 269: Could you describe what is in y (the observation)?

   We have explained the "observations" and statistics in more detail (see response to question 6), and we have added the following statement to clarify further.

   Original Text: "As mentioned above, we are interested in optimizing over the parameter space to find the value that most closely reproduces "observed" statistics."

   Updated Text (line 290): *"The observations denoted by y in Equation (1) correspond to the simulated statistics."*

Juan Antonio Añel:

8. I would like to note that the term "open-source" used by the authors here is commercial terminology, which does not correspond to the academic nature of the scientific work. The more correct terminology to refer in the text to the model would be a free software or a FLOSS model. This argument is backed by the fact that the authors release it under the Apache v2.0 license, which is widely defined as a free software license by the Apache Foundation.

Original Text: JCM is an open-source, differentiable atmospheric model built in Python using the JAX numerical library.

Updated Text (abstract): *"JCM is a differentiable atmospheric model that is built in Python with the JAX numerical library and is released under a free and open-source software (FOSS) license."*

Maximilian Gelbrecht:

1. However, it is impossible to ignore that the authors of JCM seem to make no mention of other parallel efforts to develop modern, differentiable, and GPU-accelerated variants of SPEEDY, most notably the Julia package, SpeedyWeather.jl (Klöwer et al., 2024) which has recently been made differentiable using the Enzyme.jl automatic differentiation framework (Moses et al., 2025). Other similar efforts that might be mentioned include Oceananigans.jl (Wagner et al., 2025) for ocean modeling and the HOPE shallow water model in PyTorch (Zhou, Xue and Shen, 2025).

   Thank you for your comments. The citation of SpeedyWeather.jl, which is an important precursor to this work, is added to the introduction (and addressed in response to the comment by Reviewer 1 and by Milan Klöwer).

   We have also added a citation for (Wagner et al., 2025), which is relevant for the open-source and GPU-accelerated nature of our work. We have not added a citation for (Moses et al., 2025) as it is a concurrent preprint which was submitted very shortly before this manuscript. In addition, we have included citations for Veros and DifferLand (Python/JAX components) in the introduction (previously only mentioned in the conclusion).

   Updated Text (lines 67–69): *"This work builds upon a growing movement to generate modular, differentiable Earth system components in modern programming languages with GPU acceleration (Klöwer et al., 2024; Häfner et al., 2018; Fang et al., 2024; Wagner et al., 2025)."*

2. Another possible issue is visible in Figure 4, JCM seems to exhibit what could be spectral ringing artefacts in some of its parametrizations presented. Those are not present in the Fortran Speedy figures just below. It is hard to tell just from looking at these coarse figures, but it would be a good idea to investigate this further as they could point to problems with the parametrizations.

   (This is a duplicate of the response to Milan Klöwer, question 2). We agree this warrants investigation and have been looking into it. Based on our testing, we believe that it stems from two things: (1) ringing in low-resolution spectral models is a known phenomenon (there is some ringing seen in the results from speedy.f90 in Fig.5), and (2) additional ringing induced by the dynamical core. We do not believe it is an issue with the parametrizations.

We will be looking into the role of the dynamical core in this phenomenon more in future work. As of now we have run several "stress" tests of the current configuration (e.g., running for 100 years) in order to verify that the signal is not growing/leading to instability. In these tests we do not see signs that the ringing grows over time, nor do we see any induced instability. We are able to take the gradients in a variety of applications, including over these long runs. We have added a statement about the expected stability of the model to the text, and we do not believe that these artifacts lead to issues when actually using the model.

Updated Text (lines 215–219): *"Spectral ringing in low-resolution models is a common phenomenon (Durran, 1999). Artifacts of this type of ringing can be seen in Figure 5 in the results from both models. However, the ringing in JCM is stronger than in speedy.f90. We have investigated whether this difference affects model stability, and we have found no indication that it grows over time in these low-resolution configurations (T31-T85). This includes running the model for 100-year runs and taking gradients through those long integrations."*

3. Furthermore, since JCM and NeuralGCM share an identical dynamical core, it would be really interesting to see a comparison between the learned parametrizations of NeuralGCM and the SPEEDY parametrizations of JCM, either in the present or future work.

   While we think this could be an interesting future comparison, we feel that it is out of the scope of this manuscript, which is intended to document general purpose use of the JCM. The JCM should also be more intentionally calibrated for such a comparison.

4. A third point where we feel the authors could be clearer is in their discussion regarding the benefits of JAX as the underlying numerical framework. The authors note that "Python is the most commonly used programming language in the world, and its libraries are especially powerful for scientific analysis (e.g., Xarray and Dask) and ML (e.g., PyTorch and TensorFlow)...". However, we find this statement to be potentially misleading due to the fact that none of the aforementioned python libraries are directly compatible with the JAX domain-specific language. Due to its reliance on JAX, JCM would not be able to directly integrate existing python code based on Xarray/Dask, PyTorch, or TensorFlow; rather, this code would need to be partially or wholly ported to JAX, which may range of mildly inconvenient to enormously difficult depending on the complexity of the software. These tools could of course still be used for preprocessing and postprocessing of input and output data; however, this is already possible with models written in other programming languages. The authors should therefore clarify these limitations and briefly outline a potential strategy for the integration of JCM into the broader Python ecosystem.

   We respectfully suggest that the concern raised here overstates the degree of incompatibility between JAX and the broader Python scientific ecosystem. JAX is designed around a NumPy-like array API (i.e., it does serve as a drop-in numerical backend in many contexts), and therefore does not preclude the use of higher-level Python libraries such as Xarray or Dask.

The comments here state: “Due to its reliance on JAX, JCM would not be able to directly integrate existing python code based on Xarray/Dask, PyTorch, or TensorFlow;” which we view as inaccurate. In practice, these libraries operate on array abstractions that can be backed by either standard NumPy arrays or JAX arrays, and interoperability can be achieved through lightweight interface layers (e.g., Coordax). Adaptation of Xarray to JAX through Coordax is trivial. In our research, we directly interface JAX-based components with a PyTorch emulator using Coordax, in a nearly “drop-in” workflow. This specific example is not yet published (although it will become publicly available on our GitHub). We agree that, of course, JAX is not universally interchangeable/compatible with all libraries, but we do not claim that it is. Thus we believe these statements, which are intended to highlight the strength of the Python ecosystem as a whole rather than claim strict drop-in compatibility across all applications, are appropriate.

Finally, it is true that models written in non-Python languages do not prevent the use of Python tools for pre- and post-processing. We do not make any statements in our text about the usefulness of other languages (many of which can be differentiated and/or deployed on GPUs). Our point in this section is that JCM is written in a programming language that many scientists, engineers, and developers are already familiar with. Integrating JCM with another Python program or existing analysis pipeline also does not require any additional I/O, which can be necessary to move data between programs from different software ecosystems.

Milan Klöwer

The authors present JCM a JAX-based general circulation model whereby the authors contributions is to have translated existing parameterizations from Fortran SPEEDY into JAX so they can be used together with the existing dynamical core Dinosaur (which was developed for NeuralGCM). I very much agree with the overall narrative of the paper: Modern GCMs written in easily accessible and productive languages (such as Python) and for modern hardware (GPU, TPU) with modern features such as differentiability can lower the barrier for climate modelling and accelerate climate research. In that sense, the paper is timely, important and bridges existing model (Dinosaur and Fortran SPEEDY's parameterizations) for a useful and user-friendly model.

I do recommend this paper for publication after following points have been addressed

Major points:

1. Similar efforts by other authors: The introduction currently misses many modelling efforts pursued by other teams that achieve similar goals: On the Python side there is the ocean model Veros (only mentioned in the conclusions) which is also JAX-based and which has been tested on multiple GPUs too, and I believe they also have been working with JAX's AD capabilities too. Please cite them accordingly, otherwise the introduction reads as if JCM is the first JAX-based Earth-system model component that as GPU/TPU and differentiability in mind. On the Julia side there are also many efforts, primarily Oceananigans (e.g. https://arxiv.org/abs/2502.14148) and SpeedyWeather (https://doi.org/10.21105/joss.06323) with the latter being the closest to your efforts. AD for both are documented in https://doi.org/10.22541/essoar.176314951.18114616/v1. Otherwise the reader would get an

incomplete summary on ongoing modelling efforts and the literature published on them.

SpeedyWeather is an important and very relevant precursor to this work, and we have added a citation in the introduction, which was inadvertently omitted (also see response to Reviewer 1, question 1 and Maximilian Gelbrecht et al., question 1). We have also added a citation for (Wagner et al., 2025), which is a precursor for the open-source and GPU-accelerated nature of our work. Lastly, we have added additional mention of Veros and DifferLand in the introduction (previously only mentioned in the conclusion). We have not added a citation for (Moses et al., 2025) as it is a concurrent preprint which was submitted very shortly before this manuscript.

Updated text (lines 67–69): *"This work builds upon a growing movement to generate modular, differentiable Earth system components in modern programming languages with GPU acceleration (Klöwer et al., 2024; Häfner et al., 2018; Fang et al., 2024; Wagner et al., 2025)."*

2. Spectral ringing (Fig 2a, b, Fig. 4a-c). Evident from the figures is a strong spectral ringing present in JCM simulations. They do seem to be constant in time such that I would conclude they do not necessarily present a dynamic (e.g. CFL) stability issue in, e.g., strong winds. But they may originate as a consequence of orography. While you could use a super smooth orography this also may show deficiencies in your dynamical core. Some spectral ringing isn't strictly problematic as it's the equivalent of step-wise changes of smooth functions in grid-point models but given that this particularly projects onto precipitation your model clearly seems to react to "fake mountains" somehow. I suggest to include a discussion on this and test and report on the associated stability. A little bit of ringing is not a problem but a lot of ringing can easily make the model unusable through numerical instability.

   We agree this warrants investigation and have been looking into it. Based on our testing, we believe that it stems from two things: (1) ringing in low-resolution spectral models is a known phenomenon (there is some ringing seen in the results from speedy.f90 in Fig.5), and (2) additional ringing induced by the dynamical core. We will be looking into the role of the dynamical core in this phenomenon more in future work. As of now we have run several "stress" tests of the current configuration (e.g., running for 100 years) in order to verify that the signal is not growing/leading to instability. In these tests we do not see signs that the ringing grows over time, nor do we see any induced instability. We are able to take the gradients in a variety of applications, including over these long runs. We have added a statement about the expected stability of the model to the text, and we do not believe that these artifacts lead to issues when actually using the model.

   Updated Text (lines 215–219): *"Spectral ringing in low-resolution models is a common phenomenon (Durran, 1999). Artifacts of this type of ringing can be seen in Figure 5 in the results from both models. However, the ringing in JCM is stronger than in speedy.f90. We have investigated whether this difference affects model stability, and we have found no indication that it grows over time in these low-resolution configurations (T31-T85). This includes running the model for 100-year runs and taking gradients through those long integrations."*

Kudos to your efforts and many thanks for writing them up.

Minor points:

1. l30: Many/most operational NWP models use a hand-written adjoint? It's still tedious, error-prone and can massively slow down code development or require additional developers to maintain so your point still stands.

   The text has been updated to be more accurate.

   Updated Text (lines 29–30): *"The analytical gradients of these functions are often unknown or are challenging to derive by hand."*

2. l48: I guess you argue that the gained stability in hybrid models over MLWP is because NeuralGCM can be integrated for decades while e.g. GraphCast/GenCast can't? I don't disagree with this but it's still an apple to pears comparison as these models use vastly different ML architectures too. A fair comparison was done in https://doi.org/10.1029/2025MS004969

   We have included a mention of this study (which is a nice demonstration of the point) in addition to our mention of NeuralGCM.

   Updated Text (lines 47–49): *"This can significantly improve the stability and accuracy of the resulting hybrid models. These benefits are exemplified by NeuralGCM, a hybrid atmospheric model that replaces the full physics-suite with a neural network (Kochkov et al., 2024) and by the recent study of Gelbrecht et al. (2025)."*

3. l54: ICON won the Gordon Bell prize with a heterogeneous Fortran-CPU-GPU computing so not impossible, but given they required a whole team ended up with less than 50% of lines of code actually describing algorithms (most is OpenMP/MPI/CUDA boilerplate instructions for the compiler, wrapped around Fortran loops) and sold this positively as "separation of concerns" you have a very good argument for "extremely costly" and likely also unwieldy and killing a lot of user-friendliness.

   Thanks for this example. It is a good demonstration of what we were hoping to convey. We have removed the phrase "impossible" to be more accurate.

4. l81: geopotential is not a prognostic variable in the hydrostatic equations? Rather diagnosed from surface geopotential (gravity*orography) and the locally vertical virtual temperature profile?

   Yes, thank you for pointing this out. It is a diagnostic and not prognostic variable in SPEEDY. The description has been fixed.

   Updated Text (lines 83–84): *"The dycore's prognostic variables include horizontal wind (vorticity and divergence), temperature (anomaly from a reference temperature), and log surface pressure."*

5. l84: T? are the *spectral* resolutions (or truncation) which can be paired with different *grid* resolutions, presumably the full Gaussian grid with N latitudes yielding linear, quadratic or cubic truncation. Fortran SPEEDY uses T31 with a 96x48 full Gaussian grid

which is a quadratic truncation. Your sentences here could be clearer on this?

Clarifying text has been added.

Updated Text (lines 85–89): *"Dinosaur supports a range of horizontal grid resolutions; these are labeled by their spectral truncation, the maximum wavenumber of spherical harmonic they can resolve. Dinosaur's grids use a quadratic truncation, meaning there are 3 grid points per wavelength for the highest resolved wavenumber.*

Supported spectral truncations range from T21 (a 64x32 grid) to T425 (a 1280x640 grid), with JCM's default being T31 (96x48, roughly 4°×4° grid cell size at the equator)."

6. l95: Are these the same or different from Dinosaur or Fortran SPEEDY?

   Exponential filters are no longer applied, and the only filtering is that associated with horizontal diffusion. This has been updated in the text. The horizontal diffusion filtering is different from what is implemented in speedy.f90 and is described in more detail in the Model Validation section.

   Updated Text (lines 102–104): *"After computation of parameterized tendencies at each time step, horizontal diffusion of the prognostic variables is computed and the surface pressure field is corrected to maintain constant global mean surface pressure (mass conservation)."*

7. l100: The Fortran 90 version was written by Sam Hatfield, maybe give him credit?

   The Fortran 90 version by Sam Hatfield is cited on line 177 of the original manuscript.

8. l126: Fortran SPEEDY has no daily cycle, have you changed that?

   We have not changed this, and it is now mentioned in the manuscript.

   Updated Text (lines 119–120): *"Neither SPEEDY nor JCM v1.1 implements a daily cycle."*

9. l148: The time scales (or equivalently a normalization of the diffusion operator) should also depend on the horizontal resolution, is this currently done?

   Yes—this is already accounted for. The normalization uses the maximum Laplacian eigenvalue (i.e., the grid scale), so the specified timescale applies at the smallest resolved scale and automatically adjusts with horizontal resolution.

10. l152: These appear in the code but aren't documented in Fortran SPEEDY, I also don't know why there were introduced. SpeedyWeather left them out.

    Yes, we are in the same situation and have left these out for now. We are going to investigate whether it is worth reintroducing.

11. l172: Why is 'forcing' and option of model.run and not of the model constructor 'Model'?

The intent here is that a Model's geometry (resolution, orography, and time step) will be fixed at construction, but the forcing, much like initial conditions, is a natural thing to vary between different runs of the same Model. This could be in the context of studying the dependence of a model run on different surface boundary conditions, or in the context of e.g a set of consecutive 1-year runs forced by a multi-year time series (where a different chunk of the time series would be the forcing for each run). This is especially useful if we want to optimize over surface forcing conditions (like in variational DA), because it prevents recompilation of the model run function.

12. l182: Most of the forcing (in terms of energy) comes from solar radiation?

    Sorry for confusion, this was a mis-statement. The forcing is monthly climatology obtained from ERA-Interim and includes several variables which are now listed in the text.

    Updated Text (lines 178–180): *"The models are forced by a monthly climatology of sea-surface temperature, surface albedo, land surface temperature, soil moisture, sea ice concentration, and snow cover along with realistic orography."*

13. l184: I thought a 30min time step are default?

    30 min time step is the default in the JCM, and 40 min is the default in speedy.f90. We had to pick one of the defaults in order to make the test consistent between models, and we arbitrarily decided to use the speedy.f90 default.

14. l193: What is this wind drag? Only in the PBL? Why did you leave it out?

    There appears to be a linear wind drag applied to the zonal mean vorticity and divergence in the stratosphere (speedy.f90, time_stepping.f90:79-83). It is not documented in SPEEDY or speedy.f90 (similar to the orographic corrections), so we have initially left it out of our implementation. Similar to the orographic corrections, we are investigating the necessity of this term in the JCM if the priority is to bring it as close as possible to the Fortran implementation.

    Updated Text (lines 193–196): *"Although the general formulation of the spectral dynamical cores is the same, they are not identical and the additional choices to use Runge-Kutta and explicit horizontal diffusion in the JCM lead to differing results. Speedy.f90 also implements additional linear drag on the mean vorticity and divergence in the stratosphere (this drag is undocumented and thus currently excluded from the JCM)."*

15. l207: Is this because you apply a much stronger diffusion? implicit 4th order Laplacian vs explicit 2nd order (or even 1st for divergence)?

    We apply explicit Laplacian diffusion: 2nd-order (Laplacian) for divergence, and 4th-order (Laplacian squared) for vorticity, temperature, and humidity. Lower-order diffusion acts more strongly on large scales, while higher-order diffusion preferentially damps small scales. Although our explicit scheme is inherently less diffusive per timestep than the implicit scheme in SPEEDY, some fields in JCM appear more diffuse because our 2nd-order and 4th-order diffusion target larger scales than SPEEDY's effective 8th-order hyperdiffusion. Speedy.f90 implements leap-frog + implicit diffusion + RAW filter and JCM implements

Runge-Kutta + explicit diffusion. We've added a note in the text about the larger scales involved (the time-stepping is discussed in the section on Model Parameterization).

Updated Text (lines 212–213): *"precipitation due to convection is more concentrated in speedy.f90 and more diffuse in JCM, likely due to the larger spatial scales of the horizontal diffusion."*

16. l228: Why only sometimes? Otherwise you are violating CFL?

Updated Text (lines 238–239): *"It should be noted that in practice, the runtime for the higher resolutions may pick up another factor of 2 or 3 relative to lower resolutions due to reductions to the model timestep that are eventually needed for stability."*

17. l229: Do you mean undersaturation of GPU/TPU threads at lower resolution?

Yes, though this section has now been reworked to present the data differently (and new performance data has been computed). The difference between the observed scaling and theoretical quadratic scaling no longer seems apparent/notable so we're no longer making this claim.

18. l232: Give these timings in simulated year per day (SYPD) as it's the more common unit for performance?

Original Text: "Figure 5 shows the total time to run a default configuration of JCM at T85 (approx. 150km resolution at the Equator) using either a laptop CPU, laptop GPU (NVIDIA RTX 4050), or Google Cloud TPU. The behavior in the figure has been our typical experience: runtime on GPUs and TPUs is at least an order of magnitude faster than on CPUs (except for Apple Silicon chips), with parallelization (sharding the model state arrays so they can be distributed among multiple devices) giving a significant speedup to the TPU runs on Google Cloud. For this set of parameters, the laptop GPU was faster than the cloud TPUs, but initial testing indicates that the TPU (like any high-performance cloud platform) will scale better for higher resolutions than the laptop GPU with its limited VRAM. The platforms used here do not all have the same number of cores or clock speeds; the goal is to illustrate typical runtime on common devices.

As another point of comparison, JCM on the RTX 4050 laptop GPU was about 40% faster per year of model time than speedy.f90 on the same laptop's Intel i9 13th Gen CPU.

[...]

With regards to the model run configuration itself, the number of operations for a model run typically increases with the square of the spectral truncation, with an additional factor for reductions to the model timestep (sometimes needed for stability at a higher resolution). However, runtime is observed to scale better than the number of operations, presumably due to parallelization. Figure 6 shows how model runtime scales with the spatial resolution of the model grid, using as a benchmark a one year simulation with default 30 minute time step, run on a NVIDIA RTX 4050 laptop GPU. At T31 resolution, one year of model time takes only a minute and a half, and even T119 can be run for a year in less than 10 minutes."

Updated Text (lines 223–240): *"Figure 6 shows the simulation speed, in units of simulated years per day (SYPD), of running JCM at various resolutions and on various device types.*

The behavior in the figure has been our typical experience: runtime on GPUs and TPUs is significantly faster than on CPUs, with parallelization (sharding the model state arrays so they can be distributed among multiple devices) giving an additional significant speedup to TPU runs on Google Cloud (parallelization was not used for the results in Figure 6). The dashed line in the figure represents the theoretical quadratic scaling of number of operations (and memory requirement) with spectral truncation.

[...]

It should be noted that in practice, the runtime for the higher resolutions may pick up another factor of 2 or 3 relative to lower resolutions due to reductions to the model timestep that are eventually needed for stability. For the results shown in Figure 6, the timestep was left as the default 30 minutes for all resolutions."

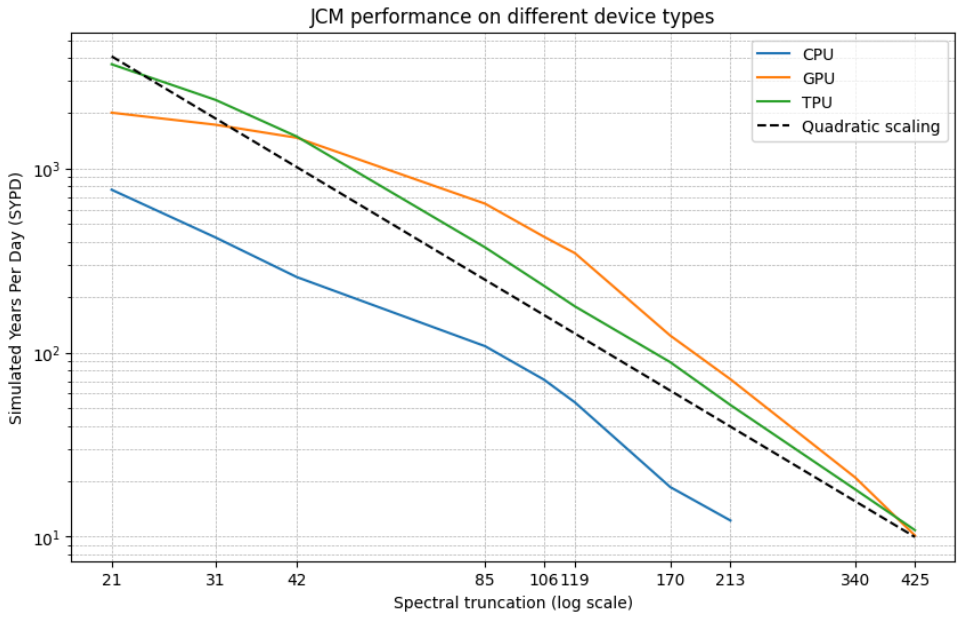
19. Fig. 5: If you put simulation period on the xaxis the this graph actual measures the overhead of initializating/compiling/precomputing (depends on what happens when you hit model.run) rather than the actual performance after that? Given you use a log yaxis that information should be in the vertical offset of the lines in the limit? I believe this is the actually interesting performance metric. The initial overhead may depend a lot on what's done at construction via 'Model' and what's done at initialization 'model.run'? Which is still useful information to be added! E.g. can a user run many performant 10-day simulations starting from varying initial conditions? Atm there seems to be quite an overhead for doing that!

    Good questions–the plot now shows SYPD, excluding compilation time, which hopefully avoids some of this confusion. We do have a way of avoiding re-compilation for varying initial conditions and forcings, and have made that explicit in the manuscript.

    Added text (lines 230–234): *"Before a run is executed, there is a small amount of compilation time (10-15 seconds for typical runs at default resolution) which varies little with resolution and run length. Advanced users conducting many short runs with varying initial conditions or forcings should use jcm.Model.run_from_state, a slight variant of jcm.Model.run which after the first run with a given resolution, timestep, save interval and total time requires no compilation time for any subsequent runs with the same values for those parameters. We encourage users to read the JCM documentation for details."*

20. Fig. 6: Y axis in SYPD? Add CPU, GPU, TPU performance to inform about where low/high resolutions are faster on CPU vs GPU/TPU?

    Done! Figure 6 has been replaced with a plot showing SYPD over a range of resolutions on a CPU, GPU, and TPU:



21. l241: See also comment on NWP adjoints above.

    This comment is addressed in response to Reviewer 1 (question 8), the updated text is pasted below.

    Updated Text (lines 250–252): *"Given the complexity of state-of-the-art Fortran models, this manual generation of gradients is expensive to implement and maintain. Thus, in most cases, gradient information is inaccessible or presents a substantial burden to model developers."*

22. l266: I agree with you that such a simple example is the first evidence that the method works but maybe note that for your example only the sign of the gradient has to be correct for the optimization to work?

    The simple example has been upgraded to a two-parameter example in response to Reviewer 2 (question 6). While it is still not realistically complex compared to cutting-edge parameter estimation challenges (so we have left our statement about simplicity), success of the two-parameter example depends on both the sign and magnitude of the gradients.

23. Fig 8/9: I'm missing a very brief discussion on the results of the sensitivity analysis. I agree that a proper analysis is beyond the scope of this paper but a) have you checked that the gradients are correct with finite differences? and b) it looks like there's some dynamically consistent response in the system that could be briefly explained, e.g. is the wind geostrophically adjusting to a higher geopotential due to higher SST? This would also provide more evidence on the correctness.

    We have tested gradients of the parameterizations using JAX built in verification methods, which compare the automatically derived values to finite differences. These tests are a part of our unit testing suite. We have noted this in the text.

Updated Text (lines 244–246): *"The JAX library provides verification functions to test model gradients against finite difference estimates, which are included in the unit testing suite for JCM."*

We have also added a brief explanation of the dynamical response seen in the sensitivities section.

Updated Text (lines 324–329): *"The localized warm SST anomaly centered near the equator drives a rapid, large-scale dynamical response in the upper-level winds, with both meridional and zonal wind perturbations propagating globally within 12–36 hours. The alternating meridional wind response straddling the equator resembles Rossby wave dispersion from an equatorial heat source (Matsuno, (1966), Gill (1980)). The TOA outgoing longwave radiation response remains local to the SST forcing throughout the 36-hour period, indicating that while the dynamical adjustment is fast and far-reaching, the thermodynamic/radiative response is still confined near the heating source on this timescale."*