JPL-SIP Report

Introducing Covariance Constraints to Atmospheric Control Adjustments Generated Via Ocean Climate Simulation

Shaunticlair Ruiz
Mentored by Ian Fenty, Ou Wang at Jet Propulsion Laboratory

August 19th, 2024

Contents

1 Abstract

Climate scientists need to make the best estimates of the atmosphere possible. One way to evaluate an atmospheric estimate, is to use that estimate in a physics simulation of the ocean, and compute the accuracy of the resulting ocean simulation. Based on how the ocean simulation differs from ocean observations, we can use gradients computed from simulation equations to determine the best way to adjust the atmospheric estimate, to improve ocean simulation accuracy. This method has a weakness, however: just because an atmosphere creates a more accurate ocean simulation, doesn't mean it's a more accurate atmosphere. Some inaccurate atmospheres may create a more realistic ocean simulation. Our solution is to modify our adjustment, and pressure it to have realistic statistical properties. In particular, we constrain the adjustment covariance to be more similar to the realistic, expected covariance. This eliminates some adjustments that, despite improving simulation accuracy, are statistically unrealistic. We test this method by creating an ocean simulation, modeling observations of that simulation, and using gradient descent to optimize the estimated atmosphere based on those observations, comparing performance with and without covariance constraints. Our approach measurably improves accuracy of atmospheric estimates. This technique could be implemented in large-scale climate models.

2 Background

2.1 MITgcm: Ocean Climate Simulation

As climate change worsens, and becomes more difficult to predict, it's ever more important to have effective climate models. This is exactly what our project pursues: an improvement to our existing climate modeling technology.

Here, we iterate on a powerful tool for simulating ocean climate: MITgcm (MIT General Circulation Model). This model divides the ocean state into discrete grid cells on a 2D map, and simulates that ocean state over discrete timesteps, starting from some initial state. This simulation runs according to approximated physical laws, and physical parameters. The output is a prediction for how the ocean state evolves in time, starting from those initial conditions.

- The parameters in the MITgcm model can represent any facet of the ocean that could affect our state: heat diffusion rate between grid cells, ocean current flows, atmospheric conditions, etc.

- For the purposes of this paper, we'll focus on one subset of parameters: atmospheric conditions.

This, however, is only the "forward-time" process of MITgcm. Next, we consider the "reverse-time" process of MITgcm.

2.2 MITgcm: Atmospheric Adjustments

We've already discussed one application of this model: predicting possible future ocean climates, given different possible atmospheric conditions. However, we can also use MITgcm to improve our estimates of real-world atmospheric conditions. How?

We'll assume our simulation physics are accurate. If so, an accurate atmosphere should generate an accurate ocean simulation.

However, the atmosphere we provide to MITgcm is only an estimation, based on observations. This estimate is limited by the quality and quantity of those observations. Uncertainty in those observations creates an imperfect atmospheric estimate.

MITgcm receives this atmospheric estimate, and simulates the ocean over time, starting from some initial state.

Inaccuracies in our atmospheric estimates cause inaccuracies in our ocean simulation. We can measure this by comparing that ocean simulation to ocean observations.

This is actually something we can use to our advantage: we use our observations as a proxy for the "true" state of the ocean, and we can measure how far off our simulation is.

- We would expect a more accurate atmospheric estimate to create a more accurate ocean state prediction. So we try the converse: if we find an atmospheric estimate that better predicts real-life ocean state, it might be closer to real-life atmospheric conditions.

We assume, however, that our atmospheric estimate is already close to correct. So, rather than creating a new atmosphere from scratch, we add an **adjustment** to our atmospheric estimate.

2.3 MITgcm: Gradient and the Adjoint Method

How do we select our adjustment? We want to adjust our atmosphere, in order to improve the accuracy of our ocean simulation.

- Our atmosphere will be represented as a vector f.
- The "inaccuracy" of our ocean simulation will be represented by the loss function J.

Based on these conventions, we want to determine the way of modifying f that will maximally decrease J (decrease inaccuracy). The most useful piece of information for adjusting f is the gradient, dJ/df.

- This gradient tells us the direction that *locally* increases J by the maximal amount.

Once we have this gradient, we can use it to modify atmosphere f, and improve our ocean simulation accuracy.

- This adjusted atmosphere f_{new} more accurately predicts ocean conditions: thus, we expect it to be more accurate.

In the most simple approach, computing dJ/df would involve a very large multivariable chain rule. However, in a massive model like MITgcm, this is prohibitively expensive.

Instead, MITgcm uses a more efficient method for computing gradients, known as the **adjoint method**. The details are outlined in 10.7

2.4 Gradient Descent

In this paper, we'll put aside the more sophisticated optimization methods that MITgcm uses. Instead, we'll use gradient descent to optimize our atmosphere f.

This means we iteratively apply repeated updates to our atmospheric estimate. After each estimate, we re-compute dJ/df, and take another update.

In the simplest case, using a step size $\eta \in \mathbb{R}$, our update rule takes the form:

$$f_{new} = f_{old} - \eta \left(\frac{dJ}{df} \right)_{f=f_{old}} \tag{1}$$

3 Conventions

3.1 Important Variables

Here, we introduce some conventions useful for the rest of this paper:

The **control vector** f, also known as the **atmospheric control**.

- We're using our the atmosphere to determine, or "control", the simulated ocean state.
- Thus, we call this our **control** vector.

The **control adjustment** a.

- Our optimization will aim to improve f, "adjusting" it with some other vector.
- We call this vector our **control adjustment**:

The **update** u.

- We'll actually be adjusting f multiple times, using gradient descent.
- Each of these is an **update** to our control.

3.2 Iteration

This notation is still insufficient: we need a distinct update u for every iteration of gradient descent.

We'll address this by using i to indicate the particular iteration of gradient descent we are discussing:

- u_i is the gradient descent update that occurs during iteration i.

This convention can be extended to our other variables, f and a:

- f_i is our current control estimate at the beginning of iteration i, before we add u_i.

$$f_{i+1} = f_i + u_i \tag{2}$$

- a_i is our total control adjustment at the beginning of iteration i: it's the sum of every gradient descent step we've taken so far.

$$a_i = \sum_{j=0}^{i-1} u_i \tag{3}$$

We can represent our current control as the sum of our "first-guess" control f_0, plus our total control adjustment, a_i.

$$f_i = f_0 + a_i \tag{4}$$

3.3 Gradient Descent

Below, we'll consider several "modified" version of gradient descent. All of them involve modifying the update rule, u_i.

But for the simplest version of gradient descent, we can define our update rule:

$$u_i = -\eta \left(\frac{dJ}{df} \right)_{f=f_i} \tag{5}$$

3.4 Loss Function

Above, we mentioned a loss function J, being our measure of "inaccuracy" in our ocean simulated.

How do we measure this? We'll take a common approach: we measure the **squared difference** between the ocean state we want to match, and the ocean state we simulated. We'll repeat this at each timestep.

- The "ocean state we want to match" is the **observed** ocean. We'll represent our observations at time t as $Hy(t)$ (the justification for this notation is provided in 5.4.1).
- The ocean state we simulate for time t will be represented as $x(t)$.

$x(t)$ and $Hy(t)$ are vectors of the same length. So, in order to get the squared difference, we'll need to multiply them as:

$$J_t(x) = \Big(x(t) - Hy(t)\Big)^\top \Big(x(t) - Hy(t)\Big) \tag{6}$$

Finally, we add up these "misfits" J_t over all timesteps. Our final timestep will be designated as τ.

$$J(x) = \sum_{t=0}^{\tau} \Big(x(t) - Hy(t)\Big)^\top \Big(x(t) - Hy(t)\Big) \tag{7}$$

3.5 Gradient dJ/df

This loss function J is what we wish to optimize over.

Our next important property is the gradient dJ/df: this will be a crucial property in every one of our gradient descent methods defined below. We'll need to condense our notation for later equations.

Each timestep will have its own gradient, based on the current control estimate, f_i. This gradient is also known as a "sensitivity". Thus, we notate it as:

$$s_i = \left(\frac{dJ}{df} \right)_{f=f_i} \tag{8}$$

Thus, our gradient descent update rule can be simplified as:

$$u_i = -\eta s_i \tag{9}$$

4 Introduction to our Work

4.1 Our Problem

This technique is useful, but there's a major concern: we would expect the real-life atmosphere to better predict the ocean state, but the converse isn't always true. Just because a chosen atmosphere would induce a realistic ocean state, doesn't mean it's a realistic atmosphere.

- This is especially a problem in the case of MITgcm: we have an enormous number of parameters (roughly 100,000), and not nearly enough observations to fully constraint that model.
- There may be many ways to fit our atmosphere to the ocean observations.

We need more information we can use to constrain the problem.

4.2 Our Solution: Covariance Constraints

Our approach is to use the **statistical structure** of the atmosphere to constrain the problem. In other words, we already know some information about what we expect the atmosphere looks like: we should encourage it to match that pattern.

What kind of structure are we referring to? We'll start by considering an example. We expect nearby regions of the atmosphere to "vary together": if we learn that one region is hot, we expect very nearby regions to be hot, too.

We can represent this with **covariance**: we can use simple assumptions, or past data, to estimate the expected covariance between different regions of the atmosphere.

Based on this, we can state our solution more clearly: when we're optimizing f, we constrain our adjustment to have a covariance close to the expected covariance.

- We'll designate this with a covariance matrix C.

4.2.1 Benefits of Covariance Constraints

We hope to get two major benefits from this approach:

- By constraining our optimization to be more realistic, the resulting solution should be more realistic.
- We can *spatially propogate information*: if we learn information about one cell, and another cell is closely correlated, then we can apply this information to that other cell, too.
  - As a result, we can learn more from each observation.

4.3 Measuring Covariance Similarity

We need one more tool: we want our improvement to have a covariance similar to C, but this requires a *metric* for determining how similar our covariance is to C.

We implement this using the **mahalanobis distance**. For a vector z, our mahalanobis distance is

$$z^\top C^{-1} z \tag{10}$$

This function will become larger if z has a covariance more different from C.

However, this function has a secondary effect: it penalizes the magnitude of z. Even if z has the correct correlation structure, increasing the magnitude will increase the mahalanobis distance.

- This effect may act as regularization, so for the time being, we leave it be.

Where does our covariance matrix C come from, and what does it look like? This problem is addressed in 10.5.

4.4 Various Covariance Constraint Methods

Here, we'll list out each of the methods we attempted, to enforce covariance constraints.

4.4.1 Modifying the loss function J

In optimization, typically, the simplest way to introduce a constraint is to include it in the loss function.

- So, we'll add the mahalanobis distance to our loss function, creating a new function, J'.
- We want to penalize the covariance dissimilarity of our overall adjustment, a_i.
- We'll include a scaling factor α to determine how strongly we want to prioritize our covariance constraint.

$$J' = J + \alpha \cdot a_i^\top C^{-1} a_i \tag{11}$$

With this new loss function, we can take the gradient, and run gradient descent.

$$u_i = -\eta \left(\frac{dJ'}{df} \right)_{f=f_i, a=a_i} \tag{12}$$

$$u_i = -\eta \Big(s_i + \alpha \cdot 2C^{-1} a_i \Big) \tag{13}$$

4.4.2 Dan's Method

Dan Amrhein [1] proposed a different approach for the optimization: thus, we informally refer to this as "Dan's Method".

In this method, we first compute s_i. Then, we want to choose our update u_i, according to two requirements:

- We want our update rule to improve our ocean misfit J. We'll select some desired decrease in J, δ.

We can approximate the change in J as $s_i^\top u_i$. So, we'll use the constraint:

$$s_i^\top u_i = \delta \tag{14}$$

- We want our update u_i to have a similar covariance to C. So, we want to penalize the mahalanobis distance,

$$u_i^\top C^{-1} u_i \tag{15}$$

We combine these constraints into the Lagrangian $\mathcal{L}$:

$$\mathcal{L}(u_i) = \lambda(s_i^\top u_i - \delta) + u_i^\top C^{-1} u_i \tag{16}$$

Now, we can derive u_i from this lagrangian, using the constraints $d\mathcal{L}/du_i = d\mathcal{L}/d\lambda = 0$ (derived in 10.2):

$$u_i = \delta\left(\frac{Cs_i}{s_i^\top C s_i}\right) \tag{17}$$

How do we choose δ? For now, we'll keep things simple: if we swap from simple gradient descent to Dan's method, we want ΔJ to be the same in both cases. In other words, we don't want Dan's method to move any more slowly than simple gradient descent.

$$\delta = s_i^\top(-\eta s) \tag{18}$$

4.4.3 Dan's Method Modified

During our work, we realized a potential weakness of this method.

Our original goal was to create an adjustment a_i with covariance C. However, in the above approach, we encourage each adjustment u_i to have covariance C.

At first glance, this shouldn't be too much of a problem: if we're adding many u_i terms with covariance C, then we'll keep the same correlations between different indices.

- However, we know that if we sum independent variables, the covariances should add together.
- In our case, we don't know that each u_i is independent, but the same sort of "gradual accumulation" of covariance could occur, as we add more and more u_i terms together.

The solution is to, instead of encouraging u_i to have covariance C, we encourage the *total adjustment*, $a_i + u_i$, to have covariance C.

We can modify the Mahalanobis distance of Dan's method to accommodate this. Our new lagrangian takes the form

$$\mathcal{L}(u_i) = \lambda(s_i^\top u_i - \delta) + (u_i + a_i)^\top C^{-1}(u_i + a_i) \tag{19}$$

Once again, we derive u_i using the constraints $d\mathcal{L}/du_i = d\mathcal{L}/d\lambda = 0$ (derived in 10.3):

$$u_i = -a_i + (\delta + s_i^\top a_i)\left(\frac{Cs_i}{s_i^\top C s_i}\right) \tag{20}$$

5 Methods

Now, we get into the details of our implementation and testing of the results.

5.1 Discretizing our state

Our spatial discretization of the ocean breaks it up into a 2D rectangular grid of cells, each having its own temperature. Our code allows for an arbitrary number of rows and columns, but for our experiments, we primarily used a 32 x 32 map.

Our state encodes the "temperature" of the ocean on each of these grid cells. To create our state vector, we stack each column of our 2D map, to create a single column vector x.

Our temporal discretization creates a single state vector state at each evenly-spaced timestep, starting at timestep $t = 0$ and terminating on some arbitrary timestep $t = \tau$.

The state vector at time t is henceforth indicated by $x(t)$.

5.2 Advection-Diffusion-Forcing Model

5.2.1 Continuous Differential Equation

The techniques described above are intended to improve the MITgcm optimization process. However, this model is very time-intensive. Using supercomputers, these models require hours to simulate, and potentially days in order to compute adjoints.

So, instead, to evaluate the effectiveness of these techniques, we chose to use a simplified version of this model. We decided to implement 3 forms of heat transfer: diffusion, advection, and forcing.

$$\frac{\partial x}{\partial t} = \overbrace{K\nabla^2 x}^{\text{Diffusion}} - \overbrace{\nu\nabla x}^{\text{Advection}} + \overbrace{F(f-x)}^{\text{Forcing}} \tag{21}$$

- F is a scalar coefficient, uniform across our entire map.

5.2.2 Discretization of Differential Equation

This equation was discretized in space and time to first-order, as outlined in 10.4.

- ∇x and $\nabla^2 x$ approximately differentiate x "spatially": in our discrete model, this means with respect to the 2D grid cell indices.

Our temporal discretization allows us to approximate $\partial x/\partial t$. With this method, we can "simulate forward" our ocean state, using Euler's method:

$$x(t+1) \approx x(t) + \left(\frac{\partial x}{\partial t}\right)\Delta t \tag{22}$$

Using a first-order approximation has an additional benefit: it allows us to treat the updated state $x(t+1)$ as an **affine** function of the current state, x. Thus, we introduce

$$x(t+1) \approx M \tag{23}$$

5.3 World Parameter Generation

After implementing our advection-diffusion-forcing model, we need a "world" to simulate according to these physics: an initial state, an atmosphere, and additional parameters that impact our physics.

5.3.1 Initial State

Our initial state can be generated randomly, with each grid cell drawn from from a normal distribution.

5.3.2 Atmosphere

A few initial comments:

- In our simplified model, the atmosphere only has a temperature at each grid cell. We don't model wind currents, pushing air (or surface water) between cells.
- Moreover, our atmosphere is *time-invariant*: the atmosphere temperature is the same for all time.
  - Our code can be smoothly adjusted to accommodate for time-variant atmosphere, however.

Now, we can proceed.

We need to generate two version of the atmosphere:

- The **true** atmosphere: this is used to generate the "real" ocean state, which we observe.
- The **first-guess** atmosphere: this is the we want to optimize, to bring it closer to the true atmosphere.

We want a few things from these atmospheres:

- The first-guess atmosphere should be similar to the true atmosphere, but have some inaccurate component.
- Both atmospheres should have similar covariances: we will assume that the source of this first-guess also has access to information about covariance, and will include it in their atmospheric estimate.

Our approach to solve this is to separate these atmospheres into three components:

- f_0: this is the "known component" of the true atmosphere. This represents the component that our first-guess has accurately predicted: we include it in both atmospheres.
- f_1: This is the "unknown component" of the true atmosphere. We don't include it in the first-guess.
- f_2: This is the "error component" of the first-guess atmosphere. It's not part of the true atmosphere: it's the part we get wrong.

To keep the desired covariance structure for the true and first-guess atmospheres, we assume that all three of these terms have covariance C.

- For our simplified case, we compute C using a gaussian dropoff, as described in 10.5.

$$f_0, f_1, f_2 \sim \mathcal{N}(0, C) \tag{24}$$

What percent of the atmosphere does our first-guess atmosphere get right? We represent this with the parameter $\gamma \in (0, 1)$.

With all of these variables, we can finally create our first guess atmosphere f_{guess}, and our true atmosphere f_{true}.

$$f_{true} = \gamma f_0 + (1-\gamma) f_1 \tag{25}$$

$$f_{guess} = \gamma f_0 + (1-\gamma) f_2 \tag{26}$$

5.3.3 Current Flow

Each grid cell in our ocean should have a current flow v, in order to make use of advection.

- But we need to be careful: we don't want to accidentally end up with a cell where water is flowing in more than it's flowing out.
- Not only is this unrealistic, it also violates the assumptions of our advection-diffusion-forcing differential equation.

Our solution is to create "circulating" current fields. If they flow in loops, whatever water enters a grid cell will come out as it continues along the loop.

We start by creating several "gaussian bumps" on our map: we add together several gaussian functions, randomly placed at different positions on our map.

Then, we take the gradient of this function.

- These gradients will all point towards the top of the nearest hill.

We then rotate these gradients 90 degrees: we replace each vector $[v_x, v_y]$ with $[v_y, -v_x]$.

The result are vectors which flow along the contours of our original plot. These contours typically form "looping" patterns.

5.3.4 Other parameters

In order to make our model more flexible, each of our parameters can be different for each grid cell. For example, two different grid cells might have different x-axis and y-axis side lengths, diffusivities along each edge, areas, etc.

This allows us to model several different types of behaviors: land continents, non-flat planets (since each cell on a globe would need to have a different size and shape), etc.

5.3.5 Cyclic Variables

We also include two additional variables, allowing for our world to be cyclic along the x-axis or the y-axis.

- Allowing east-west cyclic behavior allows us to better model an earth-like planet.

5.4 Observations

Once we have all of our variables, we can simulate the "true" ocean state, and observe it. These observations will be used to optimize f.

At each timestep, we randomly select n grid cells to observe: we choose a different random selection at each timestep.

For every single observation, we add some random gaussian noise to the true state, modelling real-world uncertainty in our observation instruments.

5.4.1 Hy(t) notation

When we were defining our loss function J, why did we represent our observed data as $Hy(t)$?

- We treat $y(t)$ as the full, "true" ocean state, before we observe it.
- H is our **observation operator**: it represents our process of selecting random grid cells, and observing them with noise added.

Thus, $Hy(t)$ is the result of observing our true state.

5.4.2 Unobserved grid cells

How do we deal with unobserved cells? We simply assign them as a NaN value, and ignore them when computing J.

- $Hy(t)$ is the same length as $y(t)$: any cell we don't observe, is left as a NaN value.

Suppose $y_i(t)$ is the i^{th} element of the true state at time t. Meanwhile, $w_i(t)$ is the normally distributed noise (standard deviation σ) we add to it.

$$w_i \sim \mathcal{N}(0, \sigma) \tag{27}$$

$$Hy_i(t) = \begin{cases} y_i(t) + w_i(t) & \text{if observed} \\ \text{NaN} & \text{otherwise} \end{cases} \tag{28}$$

5.5 Computing the gradient

Computing the gradient is, as mentioned previously, handled through the adjoint method (10.7).

5.6 Implementing our gradient descent variants

Each gradient descent variant (code shown in 10.8) uses the same template, only modifying the code that computes u_i between each variation.

5.7 Measuring performance: success metrics

We use three distinct ways of measuring the performance of our gradient descent variants.

5.7.1 Ocean Misfit

This is the loss we're optimizing over, J:

$$\ell_{ocean} = J(x) = \sum_{t=0}^{\tau} \Big(x(t) - Hy(t)\Big)^{\top} \Big(x(t) - Hy(t)\Big) \tag{29}$$

The larger J is, the more inaccurate our ocean simulation is. Our gradient descent procedure is explicitly designed to minimize this.

5.7.2 Atmoshere Misfit

Similar to the "ocean misfit", this is a squared difference:

$$\ell_{atm} = \Big(f_i - f_{true}\Big)^{\top} \Big(f_i - f_{true}\Big) \tag{30}$$

This gives us a notion for how different our current estimated atmosphere is from the true atmosphere.

We don't have access to f_{true} during training. Thus, this can be seen as a measure of overfitting: if our ocean misfit is low, but the atmosphere misfit is high, then our procedure isn't actually improving our atmospheric estimate.

- Which is important, since improving our atmospheric estimate is our original goal.

5.7.3 Mahalanobis Distance

Our alternative approaches were all intended to encourage a_i to have covariance similar to C.

Given that we measured our covariance similarity using the mahalanobis distance,

$$\ell_C = a_i^\top C^{-1} a_i \tag{31}$$

It is worthwhile for us to investigate how successful each of our four approaches were for this task.

6 Results

6.1 Simple Gradient Descent vs. Modifying J

First, we show off plots of:

- Our simple gradient descent method (3.3)

$$u_i = -\eta s_i \tag{32}$$

- constraining the covariance by modifying J (4.4.1)

$$u_i = -\eta\Big(s_i + \alpha \cdot 2C^{-1}a_i\Big) \tag{33}$$

Some mild hyperparameter tuning has been applied to α, to get the best results.

6.1.1 Plots

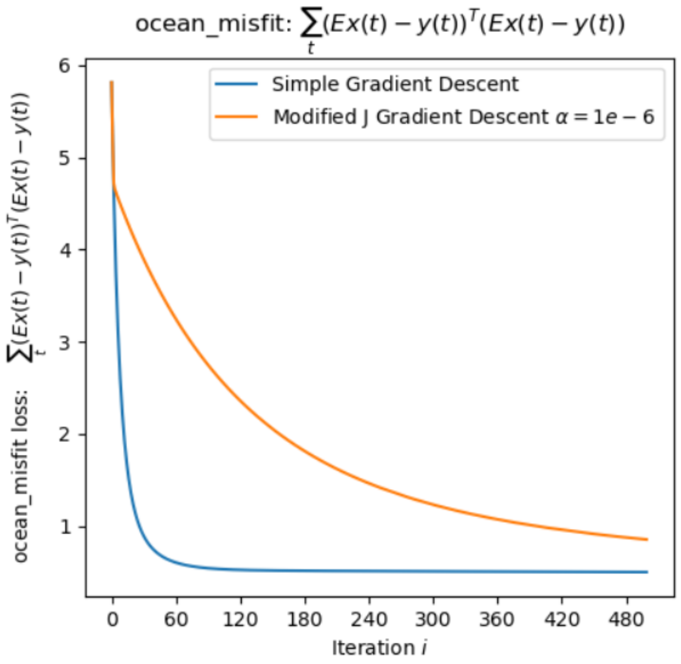


Figure 1: Modifying J seems to slow down our ocean fitting. This make sense, since we're no longer only minimizing J; we have a second term to prioritize, as well.

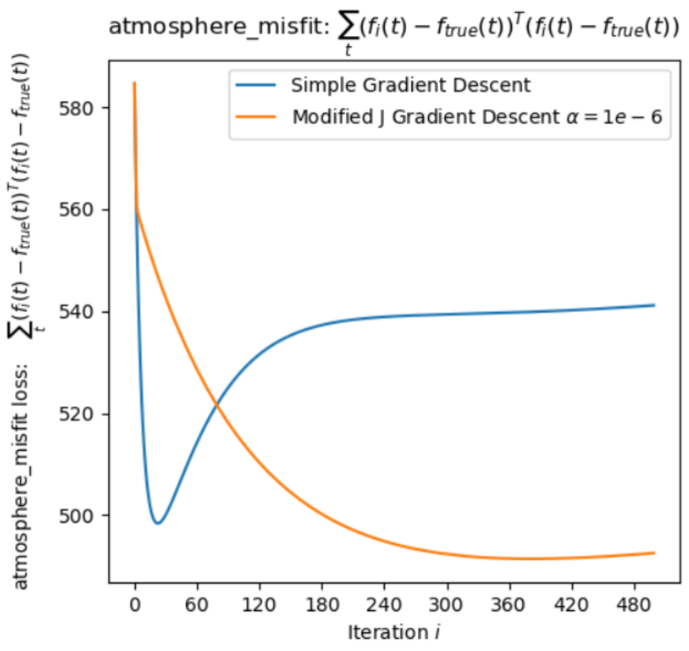


Figure 2: As we might expect, simple gradient descent begins overfitting if we run it for too long.

Modifying J seems to create a better atmospheric estimate!

It also seems to overfit more slowly: this could be because we're using a more realistic adjustment, or because Mahalanobis acts as a regularizer, penalizing larger magnitudes of α_i.

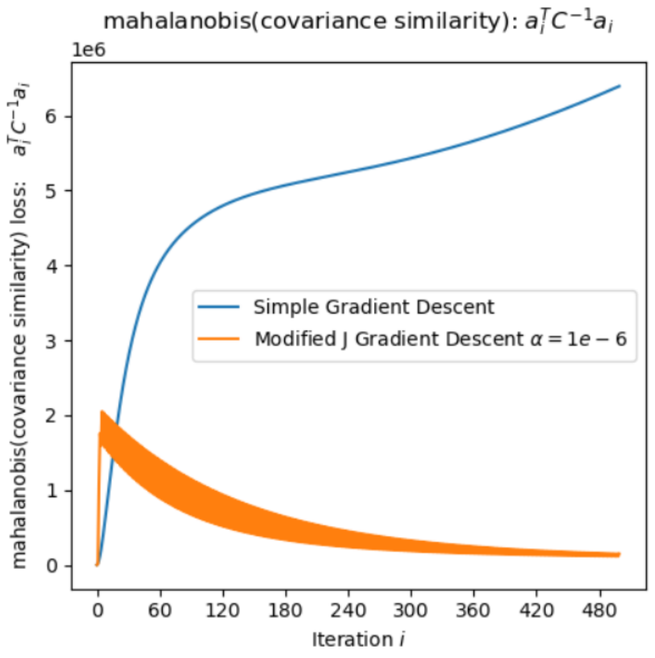


Figure 3: It makes sense that simple gradient descent would have increasing mahalanobis: there's no part of the procedure which penalizes it for creating an adjustment with an unrealistic covariance.

Strangely, the modified gradient descent seems to oscillate very quickly. Still, we can see that it clearly ends up with a much more realistic covariance.

6.1.2 Conclusions

In our case, where the true goal of optimizing the ocean simulation, is to get the best possible atmosphere estimate, modifying J seems to work better than simple gradient descent.

- This makes sense for the same reasons that motivated us to try this approach at all: we're penalizing our atmosphere for having a more unrealistic covariance. So, we might expect that to create a better atmosphere.

That said, it's somewhat unclear why we see such an aggressive oscillation on the Mahalanobis plot.

6.2 Dan Method

Next, we compare our previous methods (3.3 and 4.4.1) to the Dan method (4.4.2).

$$u_i = \delta\left(\frac{Cs_i}{s_i^\top Cs_i}\right) \tag{34}$$

6.2.1 Plots

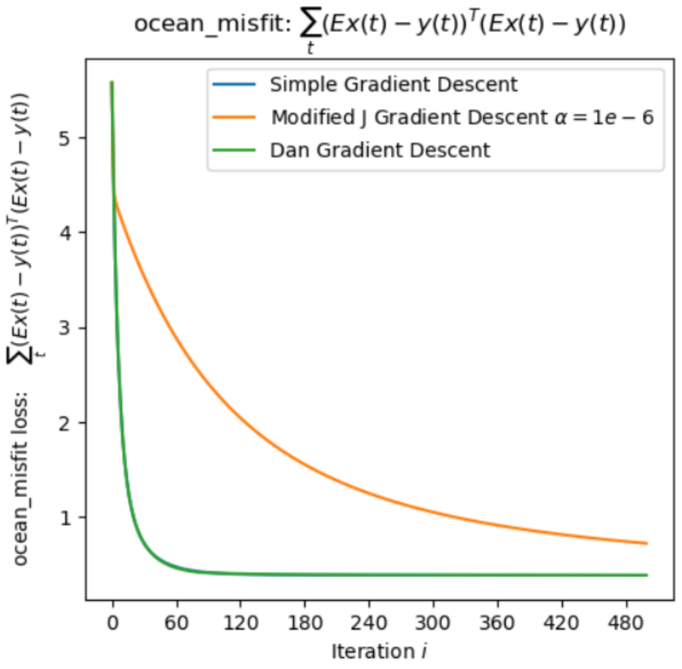


Figure 4: The Dan method performs almost exactly as well as simple gradient descent! This should make sense, since our optimization includes a constraint that J should decrease by the same amount as it would have normally.

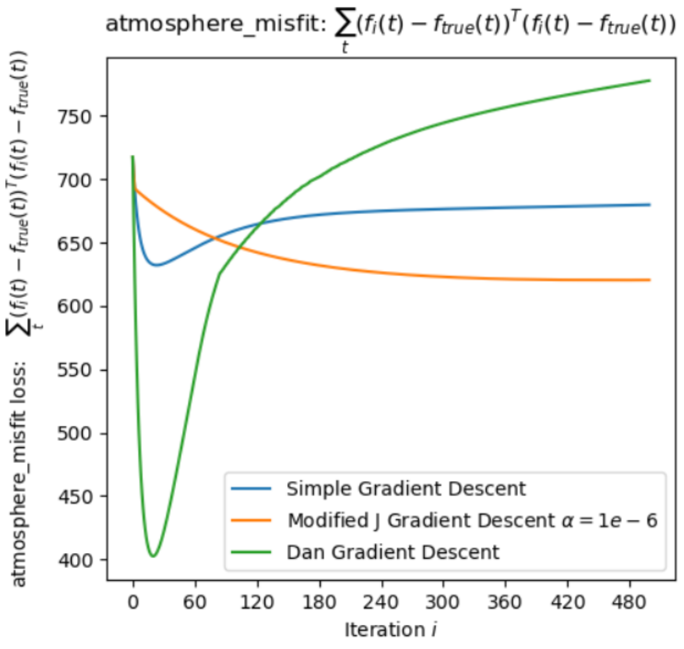


Figure 5: Dan's method creates a much better fit than either other method!

For unknown reasons, however, it suffers more from overfitting. Still, we can remedy this by using an early-stopping condition (stopping once J doesn't meaningfully decrease).

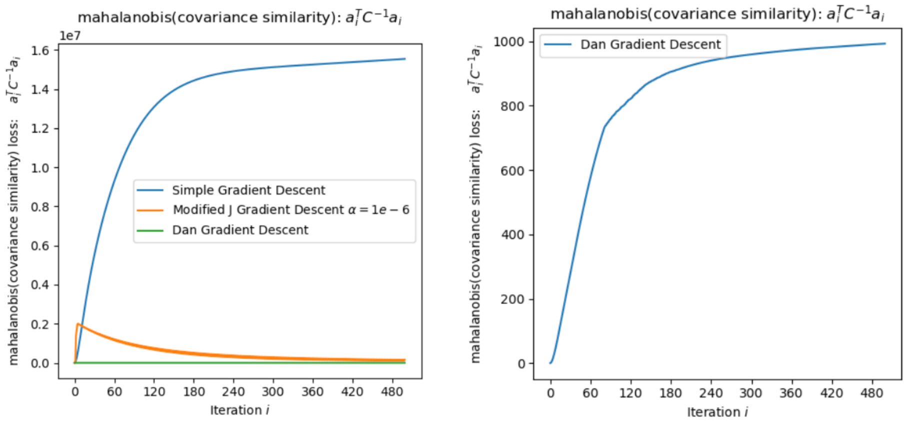


Figure 6: Dan's method keeps the Mahalanobis distance so low, that in our first plot, it's essentially invisible: it presumably does an excellent job at keeping the covariance close to C, suggesting that we have a very realistic adjustment.

The second plot only plots the Dan method, showing that, while Mahalanobis increases much less than the other methods (note that the vertical axis of the left plot is scaled by 1e7), it does still increase until converging, making a similar curve to simple gradient descent.

We should expect the Mahalanobis to increase from 0 regardless: even if we have the correct covariance structure, we're still increasing the magnitude of our control adjustment from 0.

6.2.2 Conclusions

Dan's method performs best by all of our visible measures, barring some strange overfitting issues.

It outpaces the modified-J approach, so we'll omit that one from further comparisons, for readability.

6.3 Dan Method Modified

Finally, we introduce the Dan Modified Method (4.4.3).

$$u_i = -a_i + (\delta + s_i^\top a_i)\left(\frac{Cs_i}{s_i^\top Cs_i}\right) \tag{35}$$

6.3.1 Plots

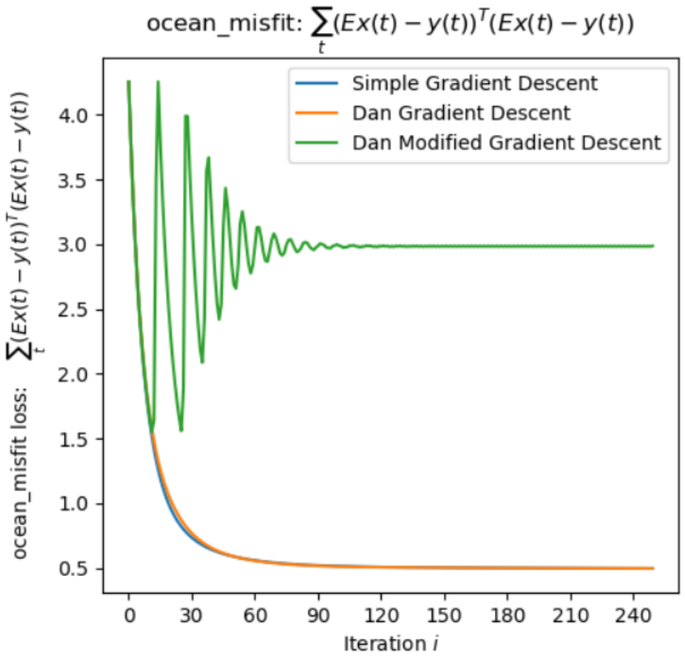


Figure 7: Dan modified starts out performing similarly to the regular Dan method before starting to aggressively oscillate... this is a serious problem.

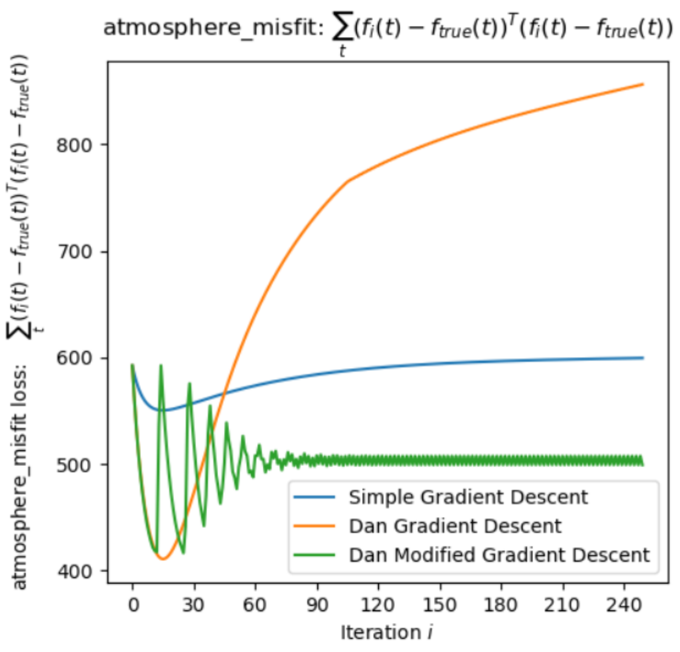


Figure 8: Same sort of pattern as before: very unstable behavior.

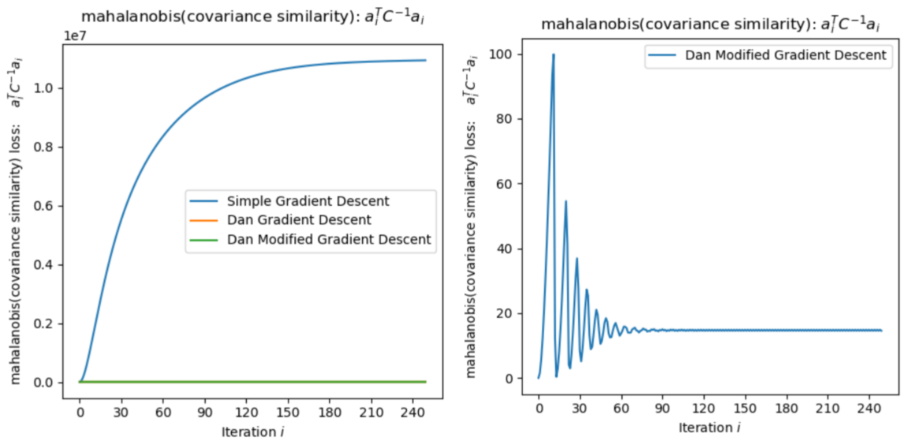


Figure 9: It seems that the mahalanobis distance is getting reset down to nearly 0, over and over again. Just as in the other cases, it seems to eventually converge on some particular mahalanobis value.

6.3.2 Conclusions

The Dan Modified method clearly fails: it starts out performing almost identical to the Dan method, and then abruptly becomes much worse. From there, it starts improving again, only to once again worsen. It repeatedly oscillates, until it converges on a worse result than Dan gradient descent's optimum.

But why does it fail?

6.4 Why does the "Dan Modified" Method Fail?

6.4.1 PCA Analysis

From the above plots, it's not entirely clear *how* the Dan Modified method is failing. Is our control adjustment oscillating over the same values? What happens when it breaks: why is it so abrupt?

The easiest way to test this would be if we could visualize the trajectory across iterations. Initially, this seems infeasible: our atmospheric vector is in a very high-dimensional space (a 32x32 grid becomes a length-1024 vector).

However, it's possible that most of the trajectory behavior is moving in only one or two dimensions: if so, then we can depict our trajectory in those dimensions, without losing information. We don't need to display dimensions that hardly change (have very low "variance"): if they're mostly constant, then we wouldn't learn anything from seeing them.

We can measure this using Principle Component Analysis (PCA): we determine which axes have the greatest variance, and project our data onto those axes.

- Notably, these axes don't have to be aligned with any of the original dimensions in our 1024-dim space: they can instead be a linear combination of our original dimensions.

This is only useful if we're right, and all of the variance can be explained by a few dimensions. As a rule of thumb, we want at least 95% of our variance to be explained by our first two axes, for those to fully describe our trajectory.

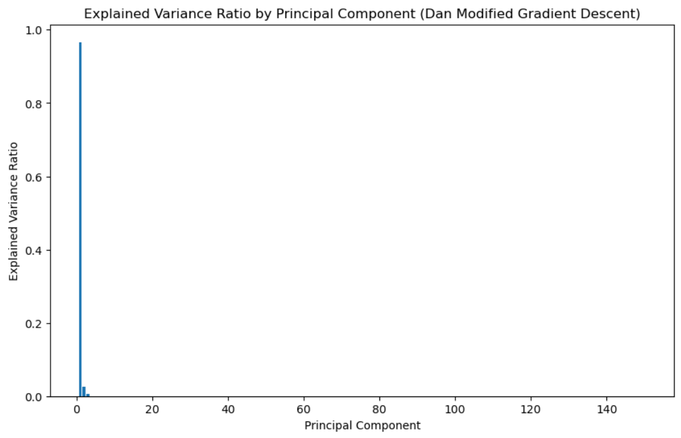


Figure 10: In this plot we see how much variance is explained by each component. Our first component describes 96.63% of the variance, and the second component describes 2.63% of it. So, our first two axes should be sufficient to accurately represent our trajectory.

If that's the case, let's see what result we get:

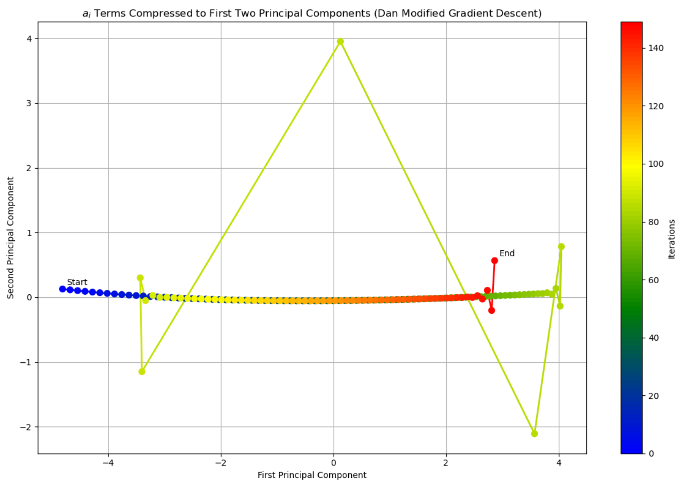


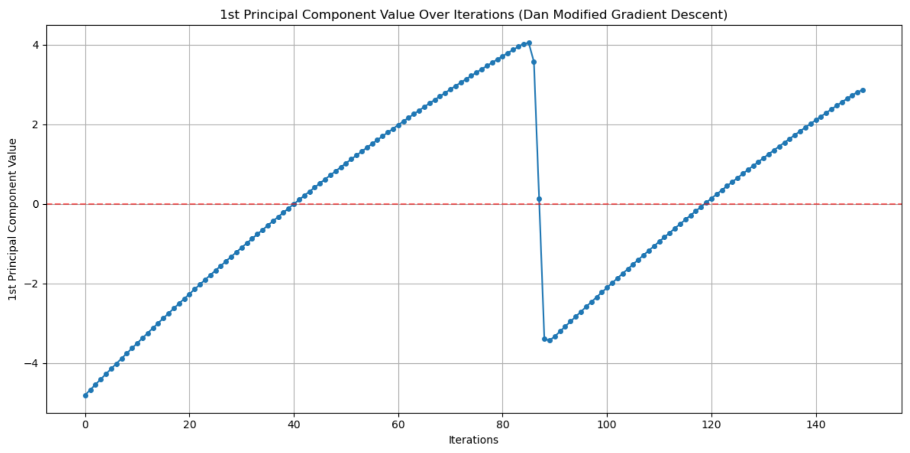
Figure 11: This plot is a bit difficult to read at first. But, if we follow the heatmap showing us the flow in time, it becomes a bit more clear:

Our plot starts our by optimizing, mostly along one axis: moving in, more or less, a straight line (left to right).

Then, it abruptly begins destabilizing, and hopping erratically, mostly perpendicular to the previous path. Once it restabilizes, we end up near where we first started.

Once it restabilizes, it essentially follows the same path that it did before.

We can plot each of our first two principle components to make this behavior clearer:



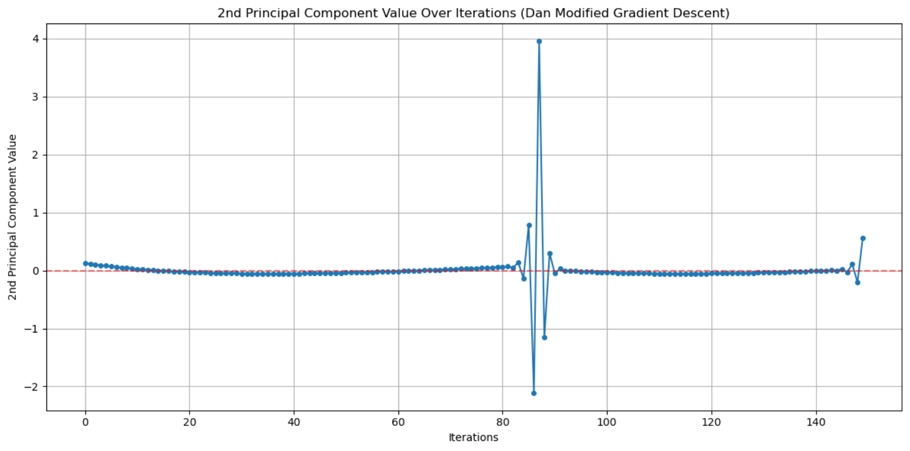


Figure 12: These plots are a bit easier to read, but they show the same sort of pattern as before: we move in a straight line along PCA 1, until the method destabilizes, hopping along PCA 2, and reversing progress on PCA 1.

Why don't we start at 0, if $a_0 = 0$? Because PCA subtracts the mean of all of our data points: so, our plot has been shifted so the mean of every point is 0.

6.4.2 "Forgetting" theory

In order to see why this method fails, let's return to the solution, from 4.4.2.

$$u_i = -a_i + (\delta + s_i^\top a_i)\left(\frac{Cs_i}{s_i^\top Cs_i}\right) \tag{36}$$

What happens to our adjustment when we apply this update to it?

$$a_{i+1} = a_i + u_i \tag{37}$$

$$a_{i+1} = a_i - a_i + (\delta + s_i^\top a_i)\left(\frac{Cs_i}{s_i^\top Cs_i}\right) \tag{38}$$

This method *completely eliminates* our old adjustment, a_i: subtracts it away.

We've essentially "forgotten" our old adjustment: it doesn't contribute to the new position of our adjustment.

$$a_{i+1} = (\delta + s_i^\top a_i)\Big(\frac{Cs_i}{s_i^\top Cs_i}\Big) \tag{39}$$

Here's how we can interpret this:

- After our previous timestep, we had some adjustment a_i.
- After our new timestep, our adjustment is pointing in a completely new direction: the direction of vector $-Cs_i$ (all other terms in the above equation are scalars, not vectors). Because a_i is completely cancelled out, we move in the $-Cs_i$ direction from the origin.

During our first timesteps, this works fine, because $-Cs_i$ is in the same direction as the previous a_i (as we can see from the previous plots, we're moving in a mostly-consistent direction).

We run into problems if Cs_i starts to change direction after a few timesteps: the further we are from the origin, the more our position changes if we were to rotate slightly. So, we take a very big step from the origin, in some new direction. Thus, the difference between a_i and a_{i+1} is very large, relative to what we expect for a typical gradient descent step.

This is problematic for several reasons:

- The bigger our step is, the less accurate our gradient will be: the gradient *locally* describes the direction of greatest increase.
- Cs_i is computed at the position a_i. But, when we take our new step proportional to $-Cs_i$, we're not starting at position a_i: we're starting from the origin, and then moving in the direction $-Cs_i$.
  - In other words, we're using gradient for position a_i, to determine our direction of motion from the origin.
  - The origin's gradient could very likely be in a different direction: we're using the **wrong gradient** for moving from the origin.

Once we've taken one big, incorrect step, we've moved into a new position, and our gradient will change direction even more. This causes us to take another, bigger step in an even more wrong direction. This repeats, self-amplifying, until the algorithm fails catastrophically.

6.4.3 Supporting evidence: $-Cs_i \cdot a_i$

Of course, to this point, this is only a guess: we need evidence.

This problem hinges on the idea that when the our algorithm breaks, it occurs because a_i and $-Cs_i$ are in different directions. This, of course, is only possible if a_i and $-Cs_i$ are in different directions at the point where our function breaks, and the same direction otherwise.

We can measure this by taking the normalized dot product between them:

$$\frac{-Cs_i \cdot a_i}{|Cs_i||a_i|} \tag{40}$$

The dot product is 1 if two vectors are in the same direction, 0 if they're perpendicular, and -1 if they're in directly opposing directions.

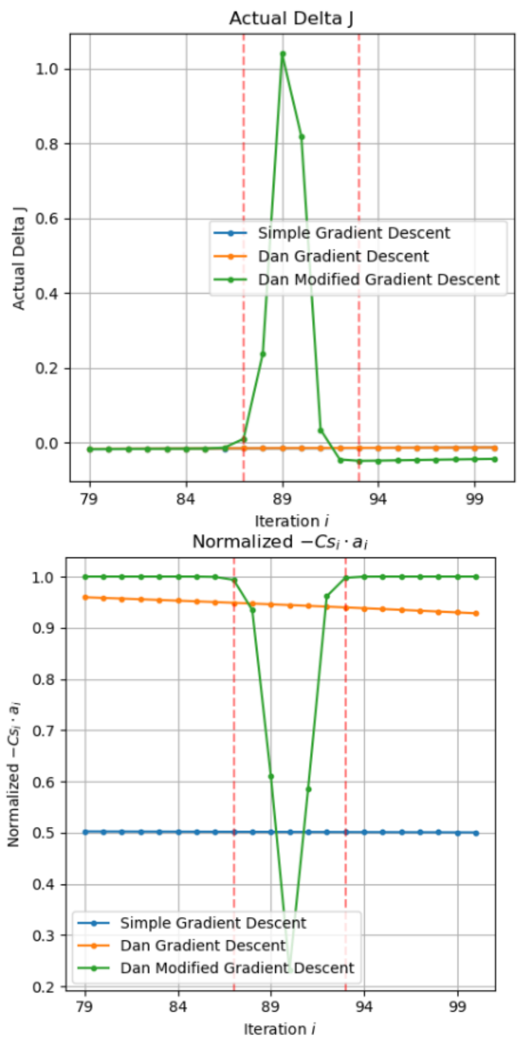


Figure 13: We can use the true ΔJ to see when our algorithm breaks: we're no longer successfully improving our ocean misfit; we're making it worse.

Notably, over the same region, we see exactly what we expected: whereas $-Cs_i$ and a_i were previously in the same direction, we start to run into problems once this is no longer the case.

We use red vertical lines to indicate this "failure region" of the Dan Modfied method.

6.4.4 PCA for comparing methods

An alternative explanation might be that Dan Modified happens to create adjustments that, eventually, enter some unusually unstable region of the adjustment space.

The first way we could check this is to compare the adjustments of the Dan Modified Method, to those of the other methods. How similar/different are they?

We can visualize this, once again, by using PCA. We combine all of the adjustments from all three of our methods (simple gradient, Dan, Dan Modified) into a single dataset, and take PCA over it.

When we compute the variance explained by each dimension, we once again find that our first two components describe $> 95\%$.

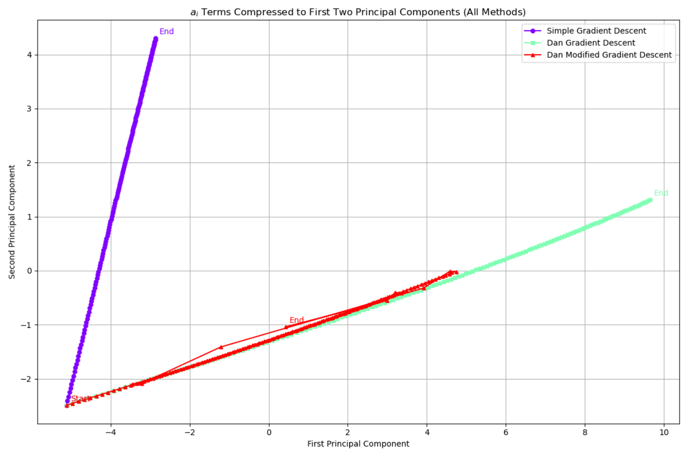


Figure 14: Our first two PCA components explain 83.5% and 15.2% of the variance, respectively.

When we compare these plots, we see that, at least visual inspection, the Dan method and Dan Modified method seem to take very similar trajectories.

Notably, simple gradient descent takes a much more different trajectory.

We're not done yet, though.

6.4.5 Switching Methods

There's a more sure way that we can test this theory. If Dan Modified is breaking because of the region of the adjustment space, then we should be able to place the Dan method and Dan Modified method in the exact same spot, and either they both break, or neither one breaks.

First, we'll test this by running the Dan modified method, and noting where it breaks. If we start from the adjustment a_i where the Dan Modified method breaks, and switch to the Dan method, then we can see whether it still breaks. If it doesn't, then the method, not the adjustment a_i, is the problem: they'll be using the exact same prior adjustment.

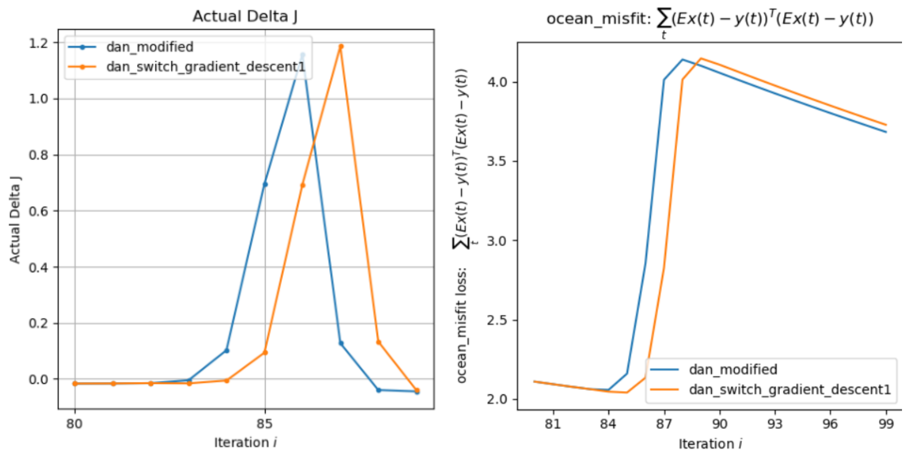


Figure 15: In the "danswitch" method, we used the normal Dan method for one timestep. During this timestep, our optimization proceeded as normal: it didn't break until we switched back to the Modified Dan method on the next timestep.

What if we use the Dan method for longer? Maybe it breaks, but it takes longer?

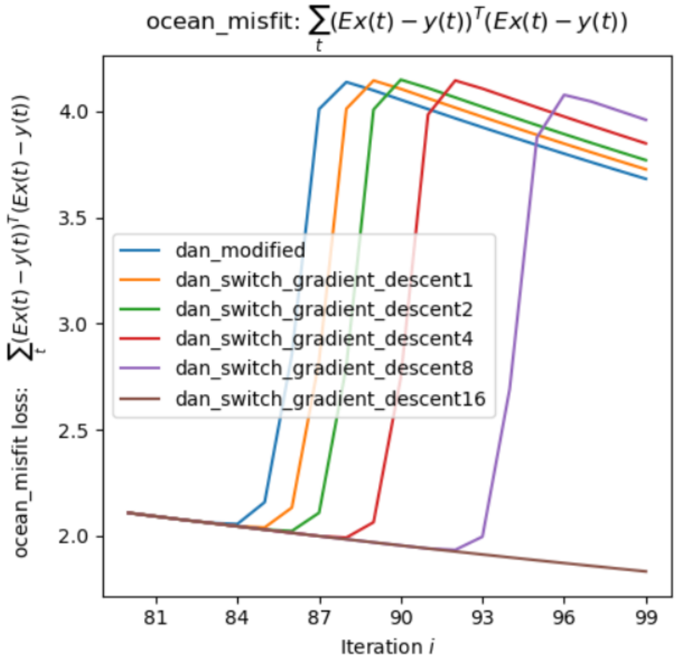


Figure 16: Each method is labelled with the number of iterations for which we switch to the Dan method. It seems that the Dan method doesn't break at all, but if we switch back to the Dan modified method, it'll break immediately.

This seems to support our theory. Another test would be to go the other way: start with the Dan Method, and switch to the Dan Modified method.

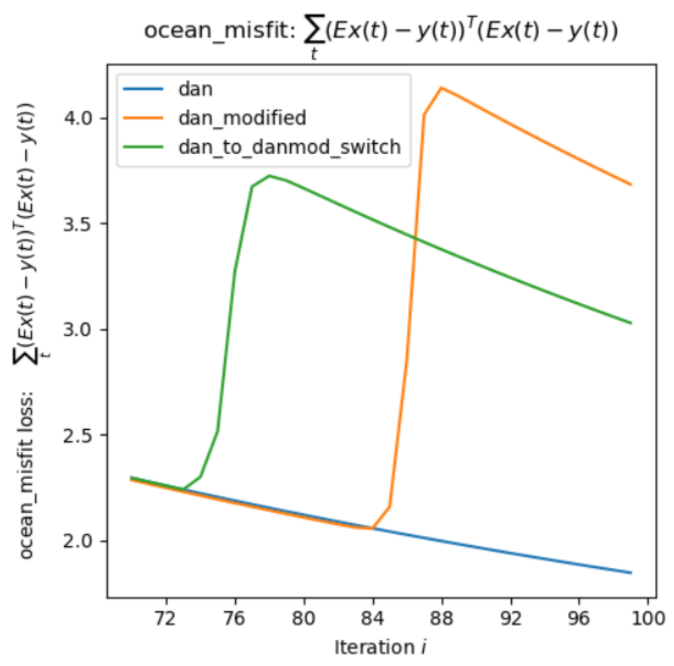


Figure 17: The results here are interesting: if we switch from Dan to Dan Modified, we can actually break our algorithm *faster* than if we just used the Modified Dan method. Why might that be?

Well, we're assuming that Dan Modified breaks because of an issue with $-Cs_i$ and a_i pointing in different directions.

So, let's check the dot product between the two:

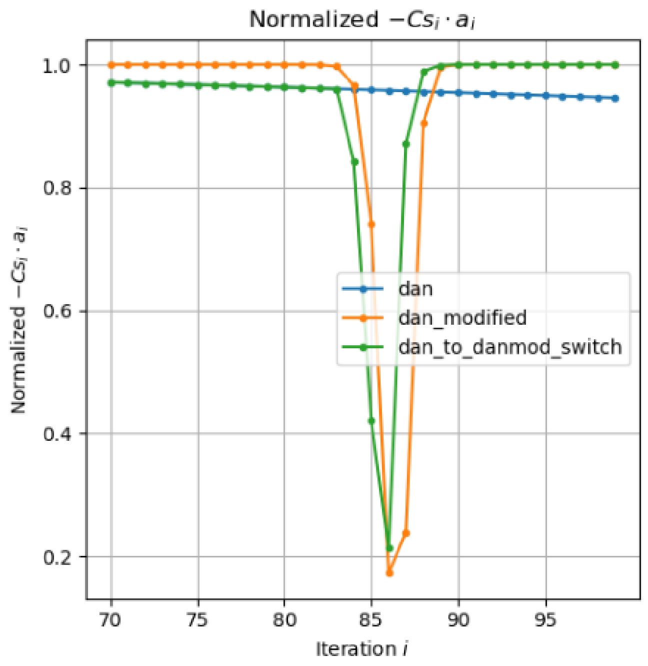


Figure 18: It seems that the simple Dan method doesn't generally have $-Cs_i$ and a_i pointing in the same direction: the normalized dot product is a bit less than 1.

This makes sense: the reason the Modified Dan Method has $-Cs_i$ and a_i pointing in the same direction, is that after each iteration, we completely replace a_i with a vector pointing towards $-Cs_i$. The same isn't true for the simple Dan method.

If we trust our theory for why Dan Modified breaks, this also explains the above behavior. If the Dan Modified method breaks because a_i and $-Cs$ are pointed in different directions, and the simple Dan method develops this property more quickly, then it would make sense for it to break when we switch to Modified Dan.

6.5 Is the Dan method really flawed at all?

It seems that the Modified Dan method is a dead end; even if we're incorrect about the reason why it breaks, we still know that it consistently does.

So, why did we decided to try the Modified Dan method? Wasn't there a problem with the simple Dan method? But when we run the simple Dan method, it seems to run well. In fact, the mahalanobis distance doesn't continuously increase like we thought it might: it actually begins to plateau.

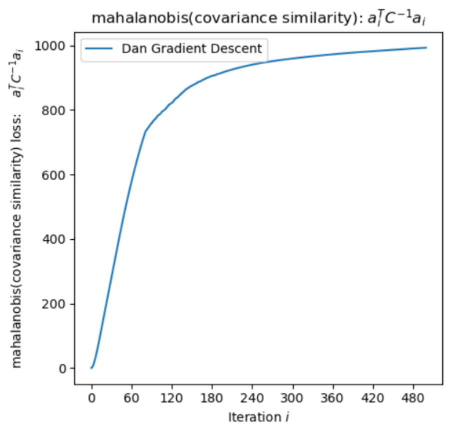


Figure 19: A reminder of how our Dan Method's mahalanobis distance increases.

6.5.1 Mahalanobis pressures Correlation, not Covariance

Eventually, we came to realize the mistake in our thinking: let's review the lagrangian for the Dan method (4.4.2).

$$\mathcal{L}(u_i) = \lambda(s_i^\top u_i - \delta) + u_i^\top C^{-1} u_i \tag{41}$$

Here, we are trying to minimize the mahalanobis distance. However, we misunderstood the purpose of the mahalanobis distance:

- At first, we thought of it as constraining u_i to have covariance C.
- *But*, Mahalanobis has a regularization effect as well: if we decrease the magnitude of u_i, that will also decrease the mahalanobis.

Suppose we have a vector z with covariance C. The vector $z/2$ would have a lower mahalanobis distance, even though the covariance is now $C/2$.

- In other words, the Mahalanobis distance pressures our covariance to be *proportional* to C, but it wants the vector magnitude to be as small as possible.

So, it encourages the "structure" that covariance C has, regardless of magnitude. We're not enforcing covariance: we're enforcing the **correlation** of our vector.

- Vector z_1 and z_2 having the same correlation is equivalent to having a *proportional* covariance: $V[z_1] = rV[z_2]$.

Why is this significant? Let's think of our previous concern:

- We were worried that each update u_i would have covariance C. As we add many updates, a_i will end up with a covariance much larger than C: it would have the right *correlation*, but it would be too large.

But this isn't how the mahalanobis constraint works: it pressures u_i to have a same covariance as implied by C. The covariance of u_i can be much smaller than C.

In fact, the above plot suggests that, as we run our gradient descent, u_i makes smaller and smaller adjustments to the covariance with later iterations. It seems to converge on some constant mahalanobis.

So, it seems that, while the Dan Modified method may not be functional, the Dan Method doesn't have the problem we thought it did.

Since each u_i should have roughly the right correlation, and thus a covariance *proportional* to C, then we should be adding to get a total adjustment a_i with covariance proportional to C, as well.

But does a_i converge on covariance C? Or some multiple of it? Let's investigate.

6.5.2 The Mahalanobis Distance we expect

We've been measuring our covariance similarity to C with the Mahalanobis distance. But as we just established, the Mahalanobis distance also penalizes magnitude:

- Having a mahalanobis distance of 0 doesn't mean you have the desired covariance: it means that $a_i = 0$.

So then, how do we know that our covariance is correct? Well, we need to find out what the mahalanobis would be, for a vector with the correct covariance.

Testing this is simple: we generate some random normal vector z with covariance C, and measure their mahalanobis.

- For our length-1024 vector, we find that the mahalanobis is roughly 1000.
- Is it the case that for a vector of length n, the average mahalanobis is n?
- Upon testing other examples, the answer is yes!

The reason why this occurs is discussed in 10.6.

More importantly, we can compare this "typical" mahalanobis value for a length-1024 vector, to the mahalanobis that the Dan Method converges to.

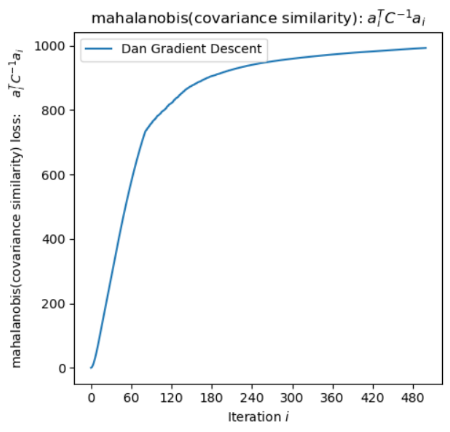


Figure 20: The Dan method converges to ≈ 1000, too!

This suggests that the Dan method likely converges on the correct covariance!

However, we can't be sure: Mahalanobis doesn't allow us to clearly separate "covariance similarity" from "magnitude".

- If we compare their magnitudes, the "typical" random length-1024 vector with covariance C (covariance matrix defined by 10.5) has magnitude ≈ 17, while our final a_i has magnitude ≈ 14.
- So, they have similar magnitudes: we could investigate further, but for now, we'll take this as sufficient evidence that the covariance of a_i (at least approximately) converges to the desired covariance C.

6.5.3 Conclusion

Our takeaways:

- The modified-J method appears to work better than simple gradient descent, but worse than Dan's Method.
- Dan's Method works best, and appears to cause our adjustment to approach the desired covariance.
- Dan's Modified Method does not work as intended, and is unlikely unnecessary.

7 Future Work

7.1 Extensions of our Toy Model

- Our atmosphere is current time-invariant. We could model a time-varying atmosphere.
- Our ocean currents are time-invariant as well, presently.
- We could test more loss functions, or try to figure out why J-modify doesn't perform as well as Dan's Method.
- Introduce more parameters: wind currents, for example.
- Our model was entirely ocean, on a flat grid, with uniform grid cell dimensions. We could introduce land masses, or reshape our model into a more realistic topology (like a low-resolution globe).
- We could advance our model beyond a first-order estimate.
- In the same way that we used covariance constraints on the atmosphere, we could introduce information about the covariance of the ocean state. This would allow us to infer about ocean states we do not observe.
- Our code is capable of using a separate covariance for the control, versus the control adjustment. We could see how this affects our performance.

7.2 Extensions Beyond our Toy Model

- The original goal of testing Dan's Method was to apply it to MITgcm. Dan's method could be implemented in that environment.
- Our model uses gradient descent, but we could use other optimization approaches more similar to MITgcm, like the BFGS algorithm.
- Our covariance matrix is manufactured: an extension should look into finding a more accurate covariance matrix, based on real data.
  - However, for a huge model like MITgcm, it would be very difficult to store this enormous matrix. So, we would need to find more space-efficient ways to store/compress this matrix.
  - One option is SVD, where we only retain the first n singular values.

7.3 Further investigation on our work

- When Dan's modified method fails...
  - How does this behavior relate to the eigenvectors of C? Can we characterize the failured based on projecting our adjustment onto the eigenvectors of C?

- Why does it consistently loop over the same region of space as it repeatedly fails? And why does it converge to a point in the middle of that region?
- Are there other ways to test our theory for why it fails?

8 References

[1] Dan Amrhein. Ocean state estimation with atmospheric adjustments constrained by prior covariance and observations. Unpublished manuscript, April 2024.

[2] Andrew M. Bradley. Pde-constrained optimization and the adjoint method. https://cs.stanford.edu/~ambrad/adjoint_tutorial.pdf, Oct 2019.

[3] Shumon Koga. Report: Project in jpl. Technical report, Jet Propulsion Laboratory (JPL), November 2017. Unpublished report.

9 Acknowledgments

I thank my mentor Ian Fenty, co-mentor Ou Wang, and external mentor Dan Amrhein.

I would also like to thank Caltech for providing resources through SFP, and NASA for providing a workspace, work materials, site access, etc.

I studied the adjoint with the help of several valuable resources, including [2], [3].

10 Appendices

10.1 Matrix Conventions

In this paper, we assume that vectors default to column-vector form.

$$z = \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_n \end{bmatrix} \tag{42}$$

Moreover, we use **denominator layout** for matrix derivatives take x and y to be scalars, and $\vec{x}$ and $\vec{y}$ to be vectors.

$$\frac{\partial y}{\partial \vec{x}} = \begin{bmatrix} \frac{\partial y}{\partial x_1} \\ \frac{\partial y}{\partial x_2} \\ \vdots \\ \frac{\partial y}{\partial x_n} \end{bmatrix} \tag{43}$$

$$\frac{\partial \vec{y}}{\partial x} = \begin{bmatrix} \frac{\partial y_1}{\partial x} & \frac{\partial y_2}{\partial x} & \cdots & \frac{\partial y_m}{\partial x} \end{bmatrix} \tag{44}$$

$$\frac{\partial \vec{y}}{\partial \vec{x}} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_2}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_1} \\ \frac{\partial y_1}{\partial x_2} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_m}{\partial x_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_1}{\partial x_n} & \frac{\partial y_2}{\partial x_n} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix} \tag{45}$$

10.2 Dan's Method Derivation

10.2.1 Constraints of Dan's Method

Dan's method can be interpreted as a modification to gradient descent.

- In its original writing, it appears to only reference a **single iteration**.

More accurately, it only discusses the total control adjustment u.

However, it only uses a single sensitivity s: since s is re-computed each GD iteration, it's most natural to treat this as a single iteration of GD.

In particular, we modify our **gradient update**, u_i, with **two constraints**.

First, we want to ensure that u_i productively improves our loss function.

Constraint 1

The **total change** in J over this iteration is equal to some constant δ.

- We'll use the approximation $\Delta J \approx u_i^\top s$.

$$u_i^\top s = \delta \tag{46}$$

But our real goal is to encourage u_i to have an expected covariance C:

The following technique works under the assumption that u_i has mean 0. We'll make this assumption.

Constraint 2

We want to make the **covariance** of u_i as close to C as possible.

- We measure this with the **mahalanobis distance**

$$u_i^\top C^{-1} u_i \tag{47}$$

- The larger it is, the further we are from having covariance C.

So, we want to **minimize** this distance.

We can combine these into a single loss function:

Key Equation 3

Our **loss function** $\mathcal{L}$ for finding our update u_i is given by:

$$\mathcal{L}(u_i) = 2\lambda\left(u_i^\top s - \delta\right) + u_i^\top C^{-1} u_i \tag{48}$$

This encourages u_i to have covariance C.

10.2.2 Optimizing over our constraints

How do we use this loss function from 48? We set two constraints: $d\mathcal{L}/du_i = 0$, and $d\mathcal{L}/d\lambda = 0$.

Because our derivatives equal 0, we divided by 2 before getting the below equations.

$$0 = \frac{d\mathcal{L}}{du_i} = \lambda s + C^{-1}u_i \tag{49}$$

$$0 = \frac{d\mathcal{L}}{d\lambda} = u_i^\top s - \delta \tag{50}$$

For our derivation below, we'll find the rearranged version more useful:

$$-C^{-1}u_i = \lambda s \tag{51}$$

$$\delta = u_i^\top s \tag{52}$$

10.2.3 Derivation

From here, we do some algebra.

From 51, we find

$$-C^{-1}u_i = \lambda s \qquad \Longrightarrow \qquad u_i = -C\lambda s$$

$$u_i = -C\lambda s \tag{53}$$

Plugging 53 into 52, we find:

$$\delta = u_i^\top s \qquad \Longrightarrow \qquad \delta = (-C\lambda s)^\top s = -\lambda s^\top C^\top s = -\lambda s^\top C s \qquad \Longrightarrow \qquad \delta = -\lambda s^\top C s \qquad \Longrightarrow \qquad \lambda = \frac{-\delta}{s^\top C s}$$

$$\lambda = \frac{-\delta}{s^\top C s} \tag{54}$$

Plugging 54 into 53, we get:

$$u_i = -C\lambda s \qquad \Longrightarrow \qquad u_i = -C\left(\frac{-\delta}{s^\top C s}\right)s \qquad \Longrightarrow \qquad u_i = \delta\left(\frac{Cs}{s^\top C s}\right)$$

This gives us our gradient update u_i:

Key Equation 4

According to **Dan's method**, we have computed our gradient update:

$$u_i = \delta\left(\frac{Cs}{s^\top Cs}\right) \tag{55}$$

10.3 Dan's Modified Method Derivation

Based on constraining a_{i+1}'s covariance instead of u_i, let's adjust our previous equations:

- We still want our **newest** update, u_i, to create a fixed change in J, δ.

$$u_i^\top s = \delta \tag{56}$$

- However, we want to encourage our **total** control adjustment, $a_i + u_i$, to have covariance C.

$$(a_i + u_i)^\top C^{-1}(a_i + u_i) \tag{57}$$

Constraint 5

We want to encourage the covariance of **control adjustment** $a_i + u_i$ to be close to C.

- We'll do this by minimizing the **mahalanobis distance**

$$\left(a_i + u_i\right)^\top C^{-1}\left(a_i + u_i\right) \tag{58}$$

We'll make a new loss function:

Key Equation 6

Our **lagrangian** $\mathcal{L}$ for finding our update u is given by:

$$\mathcal{L}(u_i) = 2\lambda\left(u_i^\top s - \delta\right) + \left(a_i + u_i\right)^\top C^{-1}\left(a_i + u_i\right) \tag{59}$$

This encourages $a_i + u_i$ to have covariance C.

10.3.1 Optimizing over constraints

We differentiate 59 just as we did before:

$$0 = \frac{d\mathcal{L}}{du_i} = \lambda s + C^{-1}(a_i + u_i) \tag{60}$$

$$0 = \frac{d\mathcal{L}}{d\lambda} = u_i^\top s - \delta \tag{61}$$

If we rearrange for utility, we get:

$$-C^{-1}(a_i + u_i) = \lambda s \tag{62}$$

$$\delta = u_i^\top s \tag{63}$$

10.3.2 Derivation

Once again, we manipulate some algebra.

From 62, we find:

$$-C^{-1}(a_i + u_i) = \lambda s \quad \Longrightarrow \quad a_i + u_i = -C\lambda s \quad \Longrightarrow \quad u_i = -u - \lambda C s$$

$$u_i = -a_i - \lambda C s \tag{64}$$

Plugging 64 into 63, we find

$$\delta = u_i^\top s \quad \Longrightarrow \quad \delta = \Big(-a_i - \lambda C s\Big)^\top s = -a_i^\top - \lambda s^\top C^\top s = -\Big(a_i^\top + \lambda s^\top C\Big) s = -\Big(a_i^\top s + \lambda s^\top C s\Big)$$

We can rearrange for λ:

$$-\delta = a_i^\top s + \lambda s^\top C s \quad \Longrightarrow \quad -\Big(\delta + a_i^\top s\Big) = \lambda s^\top C s \quad \Longrightarrow \quad \lambda = -\Big(\frac{\delta + a_i^\top s}{s^\top C s}\Big)$$

$$\lambda = -\Big(\frac{\delta + a_i^\top s}{s^\top C s}\Big) \tag{65}$$

Plugging 65 into 64, we get:

$$u_i = -a_i - C\lambda s \quad \Longrightarrow \quad u_i = -a_i - C\Big(\frac{-\delta - a_i^\top s}{s^\top C s}\Big) s \quad \Longrightarrow \quad u_i = -a_i + \Big(\delta + a_i^\top s\Big)\Big(\frac{Cs}{s^\top C s}\Big)$$

We have our gradient update u_i:

Key Equation 7

According to **Dan's modified method**, we have computed our gradient update:

$$u_i = -a_i + \Big(\delta + a_i^\top s\Big)\Big(\frac{Cs}{s^\top C s}\Big) \tag{66}$$

10.4 The Advection-Diffusion-Forcing Model, Discretized

Here, we briefly outline the discretization of our differential equation.

We simplify this by representing it in 1D. The full, 2D representation can be seen in the code.

Let c_i be the temperature of grid cell i.

10.4.1 Diffusion

We discretize the differential equation

$$\frac{\partial c}{\partial t} = K\nabla^2 c \tag{67}$$

Giving us

Note that Δx is the horizontal distance between two cells, or equivalently, the length of one cell.

$$\frac{\Delta c_i}{\Delta t} \approx K\Big(\frac{c_{i+1} + c_{i-1} - 2c_i}{(\Delta x)^2}\Big)$$

10.4.2 Forcing

The forcing term

$$\frac{\partial c}{\partial t} = F(f - c) \tag{68}$$

Doesn't require any discretization, beyond discretizing f_i and c_i in space.

$$\frac{\Delta c_i}{\Delta t} = F(f_i - c_i)$$

10.4.3 Advection

The advection expression

$$\frac{\partial c}{\partial t} = -\nu\nabla c \tag{69}$$

Has to be discretized a bit unusually, in order to avoid some numerical effects.

- If the current pushes right (positive), then heat only comes from the left; we don't care about temperature on the right.
- Vice versa if the current pushes left.

$$\frac{\Delta c_i}{\Delta t} \approx \Big(\max(\nu_i, 0)c_{i-1} + \min(\nu_i, 0)c_i - \max(\nu_{i+1}, 0)c_i - \min(\nu_{i+1}, 0)c_{i+1}\Big)/\Delta x$$

This expression may seem nonlinear, but because we assume our current to be time-independent, we can pre-compute each min and max, and turn it into a simple constant.

10.4.4 Affine Model

When we add all three of these expressions together, we can rearrange them to give us an affine model. Using matrix M and constant F, we get the function:

$$c(t+1) \approx Mc(t) + Ff \tag{70}$$

Which we use for both simulation and adjoint computation.

10.5 Our "gaussian decay" covariance assumption

We need to choose a particular covariance matrix C to work with. We chose a simple approximation: nearby cells should have a greater covariance.

We model this with a gaussian function.

$$C_{a,b} = \sigma^2_{a,b} = e^{-||a-b||^2/2s} \tag{71}$$

We combine all of these into our covariance matrix:

$$C = \begin{bmatrix} \sigma^2_{1,1} & \cdots & \sigma^2_{1,n} \\ \vdots & \ddots & \vdots \\ \sigma^2_{n,1} & \cdots & \sigma n, n \end{bmatrix} \tag{72}$$

10.6 The χ-squared distribution

Earlier, in 6.5.2, we noticed that, if a length-n vector has covariance C, its expected mahalanobis distance seems to be n. Why is that?

10.6.1 Informal Justification

First, we'll informally justify it. Let's start by consider what the mahalanobis distance *is*. For review:

$$z^\top C^{-1} z \tag{73}$$

This equation is a generalized version the 1d equation, where we compute z^2, and divide by the variance.

$$\left(\frac{z}{\sigma}\right)^2 \tag{74}$$

Since z is assumed to have mean 0, z/σ is a standard normal variable, $z/\sigma \sim \mathcal{N}(0,1)$.

We're *squaring a standard normal variable*: this is closely related to the definition of the **chi-squared distribution**.

In particular, χ^2_k is the sum of k squared standard normal variables.

Now, we remind ourselves that z is not actually a single real number: it's a vector of length n.

To make this informal argument easier, let's trust that C^{-1} "cancels out" the covariance C of the random normal vector z. So, we end up with w, a standard normal vector.

$$z^\top C^{-1} z = w^\top w = \sum_i w_i^2 \tag{75}$$

Each w_i is now a standard normal vector. So, we're summing n squared standard normal vectors: we have a chi-squared distribution, χ^2_n.

The expected value of this distribution is n: this is exactly what we observe.

10.6.2 Formally demonstrating this result

Our goal is to formally justify the step,

$$z^\top C^{-1} z = w^\top w \tag{76}$$

Above, we used the "square root" of variance, σ, to cancel out the variance of z in the 1D case. Here, we'll do something similar, and define a sort of "square root" for our matrix C.

Thus we introduce the Cholesky Decomposition:

- If a matrix is positive-definite symmetric (like any covariance matrix C), then it has a "cholesky decomposition into a matrix L, where

$$LL^\top = C \tag{77}$$

  For this proof, we do not care about the actual contents of L: we just need to know that it exists.

An important property of the cholesky decomposition (or any other "square root" of C):

- **Lemma**: if $w \sim \mathcal{N}(0, I)$, then $Lw \sim \mathcal{N}(0, C)$.

  Since we know that Lw is mean zero, we can get the covariance as:

$$\mathrm{Cov}(Lw) = E\left[(Lw)(Lw)^\top\right] \tag{78}$$

  If we rearrange:

$$E\left[(Lw)(Lw)^\top\right] = E\left[Lww^\top L^\top\right] = LE\left[ww^\top\right]L^\top \tag{79}$$

  We already know that w has covariance I. Since it has mean 0, $E\left[ww^\top\right] = \mathrm{Cov}(w) = I$.

$$\mathrm{Cov}(Lw) = LE\left[ww^\top\right]L^\top = LIL^\top = LL^\top = C \tag{80}$$

  Thus, $Lw \sim \mathcal{N}(0, C)$.

Now, we can use a trick: we know that $z \sim (0, C)$ has the same distribution as Lw. So, we'll substitute one for the other: $z^\top C^{-1} z$ is equivalent in distribution to $(Lw)^\top C^{-1}(Lw)$.

$$(Lw)^\top C^{-1}(Lw) = w^\top L^\top C^{-1} L w \tag{81}$$

We'll use the fact that $C^{-1} = (LL^\top)^{-1} = (L^\top)^{-1}L^{-1}$.

$$w^\top L^\top C^{-1} L w = w^\top L^\top (L^\top)^{-1} L^{-1} L w = w^\top w \tag{82}$$

We've shown that our mahalanobis distance is equivalent in distribution to $w^\top w$: in other words, equivalent in distribution to χ^2_n. Proof complete.

10.7 The Adjoint Method

The adjoint method allows us to compute dJ/df more efficiently. Here, we derive/justify the approach.

10.7.1 Setting Up/Motivation

For our derivation, it's easiest to focus on the atmosphere at a single timestep, even though our atmosphere is uniform. We choose an arbitrary timestep q.

Our goal is to modify our atmospheric forcing $f(q)$ to improve our simulation (in other words, reducing J). This can be best represented by asking, "how does modifying $f(q)$ affect J?" This question is answered by the derivative,

$$\frac{dJ}{df(q)}$$

We can use this to directly compute an adjustment to $f(q)$, to improve our estimate. So, this derivative is our goal.

10.7.2 Our Model

How does $f(q)$ affect J? It doesn't directly show up in the equation for J.

- Rather, it *indirectly* affects J, by modifying the (simulated) ocean state, $x(t)$.

This effect is represented by our equation for simulating forward in time:

$$x(t+1) = Mx(t) + Ff(t)$$

$f(q)$ influences the next state $x(q+1)$, which contributes to J. But, we're forgetting a second way that $f(q)$ can affect J: by affecting *future states*.

- While $f(q)$ only directly affects $x(q+1)$, we use $x(q+1)$ to compute $x(q+2)$. We can then use $x(q+2)$ to compute $x(q+3)$, and so on.
- So, $f(q)$ affects all of our future states!
- By affecting each of these states, $f(q)$ can affect J at $\tau - q$ different states.

We can account for all of these terms using the multivariable chain rule:

$$\frac{dJ}{df(q)} = \sum_{t=q+1}^{\tau} \frac{dx(q+1)}{df(q)} \cdot \frac{dx(t)}{dx(q+1)} \cdot \frac{\partial J}{\partial x(t)}$$

We know how to compute each of these terms: the first and third terms are known matrix derivatives, so we'll put them off until later.

It's useful to think of this in a second way: above, we've listed every way that $x(q+1)$ can affect J. We have a *total derivative* of J with respect to $x(q+1)$.

$$\frac{dJ}{df(q)} = \frac{dx(q+1)}{df(q)}\left(\sum_{t=q+1}^{\tau}\frac{dx(t)}{dx(q+1)}\cdot\frac{\partial J}{\partial x(t)}\right) = \frac{dx(q+1)}{df(q)}\left(\frac{dJ}{dx(q+1)}\right)$$

10.7.3 Redundant Calculations

This technique gets the job done, but it can be inefficient to use for multiple timesteps: we have a lot of duplicate calculations. Consider an example:

- $f(1)$ and $f(2)$ both affect $x(3)$, which in turn affects J. Thus, both equations require $\frac{dJ}{dx(3)}$.

$$\frac{dJ}{df(1)} = \frac{dx(2)}{df(1)}\left(\overbrace{\frac{dJ}{dx(2)}}^{\text{Total effect of } x(2)}\right) = \frac{dx(2)}{df(1)}\left(\overbrace{\frac{\partial J}{\partial x(2)}}^{x(2)\text{ effect by itself}} + \overbrace{\frac{dx(3)}{dx(2)}\,\color{red}{\frac{dJ}{dx(3)}}}^{x(2)\text{ effect via future timesteps}}\right)$$

$$\frac{dJ}{df(2)} = \frac{dx(3)}{df(1)}\left(\color{red}{\frac{dJ}{dx(3)}}\right)$$

10.7.4 The Adjoint Method: Base Case

It seems that, in the above case, it would make sense to compute $dJ/dx(3)$ first, so we can re-use it for computing $dJ/dx(2)$.

- But if we just showed that $dJ/dx(3)$ is used for twice, doesn't it make sense that the same is true for $dJ/dx(4)$?
  - If we use an identical argument to before, we could show that computing $dJ/dx(3)$ involves computing $dJ/dx(4)$.
  - So, we should handle $dJ/dx(4)$ first.

We can use the same logic over and over, going further forward in time: it seems we're reusing a lot of calculations!

The natural conclusion is for us to start with the very last timestep, $dJ/dx(\tau)$.

- Because there are no future timesteps, $x(\tau)$ can only affect J directly:

$$\frac{dJ}{dx(\tau)} = \frac{\partial J}{\partial x(\tau)}$$

10.7.5 The Adjoint Method: Recursion

Now, we can move one step **backwards** in time, using the equation we wrote above:

$$\frac{dJ}{dx(\tau-1)} = \overbrace{\frac{\partial J}{\partial x(\tau-1)}}^{x(\tau-1)\text{ effect by itself}} + \overbrace{\frac{dx(\tau)}{dx(\tau-1)}\frac{dJ}{dx(\tau)}}^{x(\tau-1)\text{ effect via }x(\tau)}$$

To make things clearer, we'll rename the variable we're recursively building up:

$$\lambda_t = \frac{dJ}{dx(t)}$$

Rewriting our equation:

$$\lambda_{\tau-1} = \frac{\partial J}{\partial x(\tau-1)} + \frac{dx(\tau)}{dx(\tau-1)}\lambda_\tau$$

We get something that looks like a **recursive** relation: $\lambda_{\tau-1}$ references the next element in the sequence, λ_τ. As we move further back in time, we find the exact same equation, confirming our suspicions. If we write it in general, we get:

$$\lambda_t = \begin{cases} \frac{\partial J}{\partial x(t)} + \frac{dx(t+1)}{dx(t)}\lambda_{t+1} & \text{if } t < \tau \\ \\ \frac{\partial J}{\partial x(t)} & t = \tau \end{cases}$$

These are our **adjoint variables**.

10.7.6 Using the adjoint

We can find our adjoint variables by moving backwards in time: we start by computing λ_τ, and begin decrementing through $t = \tau-1, \tau-2, ..., 2, 1$.

Once we've finished, it's easy to compute our final derivatives:

$$\frac{dJ}{df(q)} = \frac{dx(q+1)}{df(q)}\lambda_{q+1}$$

If we apply this to our model from 10.4 ($x(t+1) = Mx(t)+Ff(t)$), we find that $\frac{dx(t+1)}{dx(t)} = M^\top$

$$\lambda_t = \begin{cases} \frac{\partial J}{\partial x(t)} + M^\top\lambda_{t+1} & \text{if } t < \tau \\ \\ \frac{\partial J}{\partial x(t)} & t = \tau \end{cases}$$

If we work through the induction, we can simplify this to:

$$\lambda_k = \sum_{i=k}^{\tau} \left((M^{i-k})^\top \frac{\partial J}{\partial x(i)} \right)$$

And finally:

$$\frac{dJ}{df(q)} = F\lambda_{q+1} = F \sum_{i=q+1}^{\tau} \left((M^{i-q-1})^\top \frac{\partial J}{\partial x(i)} \right)$$

Notice that the last forcing, $f(\tau)$, actually has no effect on our loss: it would be applied to a future state $x(\tau+1)$, that doesn't exist.

- In the above equation, this would refer to some non-existent $\lambda_{\tau+1}$.

10.7.7 Why is the adjoint useful?

Something worth addressing:

Q: *Couldn't we have computed the answer in our original form, without invoking the adjoint? We could've just plugged values into the chain rule we started with.*

In this particular case, this is true. However, this is only simple, because our model takes on such a simple form, where we can multiply by $A^\top$ repeatedly to get our answer.

In many situations, our model can be too complex to get an analytical derivative. So, instead, we might use a more demanding approach, like **finite difference approximation**:

- Modify one variable of $f(q)$ and simulate the whole model, seeing how the loss changes.
- We repeat this process for each variable in $f(q)$, to get the overall derivative.
- Then, we have to repeat *all* of that, for every timestep q.

Using the adjoint method, we can significantly cut down on the work we have to do:

- First, we compute the adjoint variables λ_t: this requires computing our derivatives $\partial J/\partial x(t)$ and $\partial x(t+1)/\partial x(t)$.
  - $\partial J/\partial x(t)$ can be gotten directly from the loss function.
  - $\partial x(t+1)/x(t)$ only requires simulating one timestep forward, for each variable.

Since we have to simulate between each pair of timesteps t and $t+1$, this is equivalent to running through the whole model once (per variable in x).

Once we've done that, we don't need to run the whole simulation for each $f(q)$: we only have to run one timestep, to see how it affects $x(q+1)$.

We can think of this as "pre-simulating" the effect that our states have on the loss, so that we only have to see how $f(q)$ affects the first in that chain of timesteps: $x(q+1)$.

10.8 Code

Our code is spread across 8 jupyter notebooks, including tutorials justifying each function, and demonstrating how to use it.

Here, we provide the file `helper.py`, which strips away these additional materials and simply provides the functions.

```
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
from matplotlib import gridspec
from scipy.interpolate import griddata
from matplotlib.colors import ListedColormap
from scipy import sparse
import copy

#NASA Logo

def png_to_matrix(file_path):
    """
    Convert a PNG image to a NumPy matrix.

    Args:
    file_path (str): Path to the PNG file.

    Returns:
    numpy.ndarray: 2D matrix representing the image.
    """
    # Open the image using PIL
    img = Image.open(file_path)

    # Convert the image to grayscale if it's not already
    img = img.convert('L')

    # Convert the image to a NumPy array
    matrix = np.array(img)

    return matrix

file_path = 'ImageProcessingDemo128.png'
NASA_map = png_to_matrix(file_path)[::-1,::]



### Diffusion-Advection-Forcing Model ###

def make_M_2d_diffusion_advection_forcing(nr: int, nc: int, dt: float,
                                KX: np.ndarray, KY: np.ndarray,
                                DX_C: np.ndarray, DY_C: np.ndarray,
                                DX_G: np.ndarray, DY_G: np.ndarray,
                                VX: np.ndarray, VY: np.ndarray,
                                RAC: np.ndarray,
                                F : float,
                                cyclic_east_west:   bool = True,
                                cyclic_north_south: bool = False,
                                M_is_sparse=True):
    """
    Creates linear model M which can be used to forward-simulate a discrete approximation of a
    2D diffusion-advection-forcing model.

    c(t+1) = Mc(t) + F f(t)

    Spatially-variant case

    nr:     the number of rows of discrete cells.
    nc:     the number of cols of discrete cells.
    dt:     duration of a timestep

    (i,j) = (0,  0) is the southwesternmost cell
    (i,j) = (-1,-1) is the northeasternmost cell

    For the below definitions:

    KX:     the diffusivity constant matrix along the x-axis (between columns)
                KX[i,j]   gives diffusivity at the boundary between cells [i,j-1] and [i,j]

    KY:     the diffusivity constant matrix along the y-axis (between rows)
                KY[i,j]   gives diffusivity at the boundary between cells [i-1,j] and [i,j]

    DX_C:   the horizontal distance (x-axis) matrix between the centers cells in adjacent columns
                DX_C[i,j] gives the distance between the centers of cells [i,j-1] and [i,j]

    DY_C:   the vertical distance (y-axis) matrix between the centers of cells in adjacent rows
                DY_C[i,j] gives the distance between the centers of cells [i-1,j] and [i,j]

    DX_G:   the horizontal length (x-axis) matrix of a cell along one edge.
                DX_G[i,j] gives the length of the "south" side of cell [i,j]

    DY_G:   the vertical length (y-axis) matrix of a cell along one edge.
                DY_G[i,j] gives the length of the "west" side of cell [i,j]

    VX:     the velocity constant matrix along the x-axis (between columns)
                VX[i,j]   gives the velocity at the boundary between cells [i,j-1] and [i,j]

    VY:     the velocity constant matrix along the y-axis (between rows)
                VY[i,j]   gives the velocity at the boundary between cells [i-1,j] and [i,j]

    RAC:    the area of a cell.
                RAC[i,j]  gives the area of cell [i,j]

    F:     the forcing term constant. This is the same for all cells.

    cyclic_east_west:   if True, cell [i, 0] is east of cell [i,-1]

    cyclic_north_south: if True, cell [0, j] is north of cell [-1,j]

    M_is_sparse: if True, return a sparse matrix. If False, return a dense matrix.
    """

    if KX.shape !=   (nr, nc+1):
        raise ValueError("KX doesn't have the right shape for your dimensions!")
    if KY.shape !=   (nr+1, nc):
        raise ValueError("KY doesn't have the right shape for your dimensions!")
    if DX_C.shape != (nr, nc+1):
        raise ValueError("DX_C doesn't have the right shape for your dimensions!")
    if DY_C.shape != (nr+1, nc):
        raise ValueError("DY_C doesn't have the right shape for your dimensions!")
    if DX_G.shape != (nr+1, nc):
        raise ValueError("DX_G doesn't have the right shape for your dimensions!")
    if DY_G.shape != (nr, nc+1):
        raise ValueError("DX_G doesn’t have the right shape for your dimensions!")
    if VX.shape !=   (nr, nc+1):
        raise ValueError("VX doesn’t have the right shape for your dimensions!")
    if VY.shape !=   (nr+1, nc):
        raise ValueError("VY doesn’t have the right shape for your dimensions!")

    size = nr * nc

    if M_is_sparse:
        M = sparse.lil_matrix((size, size))
    else:
        M = np.zeros((size, size))

    beta = dt / RAC



    # Shorthand variables

    S = KX*DY_G/DX_C
    T = KY*DX_G/DY_C

    S_IJ, T_IJ     = S[:,   :-1], T[:-1,  :]
    S_IJP1, T_IP1J = S[:,   1:],  T[1:,   :]

    R = VX*DY_G
    Q = VY*DX_G

    R_IJ, Q_IJ     = R[:,   :-1], Q[:-1,  :]
    R_IJP1, Q_IP1J = R[:,   1:],  Q[1:,   :]

    # Contributions from diffusion (d)

    d_IP1_J = beta * T_IP1J
    d_IM1_J = beta * T_IJ
    d_I_JP1 = beta * S_IJP1
    d_I_JM1 = beta * S_IJ

    d_IJ = - d_IP1_J - d_IM1_J - d_I_JP1 - d_I_JM1

    # Contributions from advection (a)

    a_IM1_J = beta * np.maximum(Q_IJ, 0)
    a_IP1_J = - beta * np.minimum(Q_IP1J, 0)
    a_I_JM1 = beta * np.maximum(R_IJ, 0)
    a_I_JP1 = - beta * np.minimum(R_IJP1, 0)

    a_IJ = beta * (\
        np.minimum(Q_IJ, 0) - np.maximum(Q_IP1J, 0) + \
        np.minimum(R_IJ, 0) - np.maximum(R_IJP1, 0))


    #Create array to store indices

    c = np.zeros([nr,nc])
    c_indices = np.arange(len(c.ravel()))
    c_indices = np.array(np.reshape(c_indices, [nr, nc]))

    for i in range(nr): #y-axis (north, south)
        for j in range(nc): #x-axis (east, west)

            #Get current position
            ind_here = c_indices[i,j]

            # Currently we have no adjacent cells, we need to populate
                them
            ind_N = np.nan
            ind_E = np.nan
            ind_S = np.nan
            ind_W = np.nan

            # Get indices for each direction
            # south
            if i > 0:
                ind_S = c_indices[i-1, j]
            elif cyclic_north_south:
                ind_S = c_indices[-1, j]

            # north
            if i < nr-1:
                ind_N = c_indices[i+1, j]
            elif cyclic_north_south:
                ind_N = c_indices[0, j]

            # west
            if j > 0:
                ind_W = c_indices[i, j-1]
            elif cyclic_east_west:
                ind_W = c_indices[i, -1]

            # east
            if j < nc-1:
                ind_E = c_indices[i, j+1]
            elif cyclic_east_west:
                ind_E = c_indices[i, 0]

            # Now that we have our indices, we can fill in our matrix

            M[ind_here, ind_here] = 1 + d_IJ[i,j] + a_IJ[i,j] \
                                        - F #Forcing term is the same for all
                                            cells

            if np.isfinite(ind_W):
                # cell to the west
                M[ind_here, ind_W] = 0 + d_I_JM1[i,j] + a_I_JM1[i,j]
            if np.isfinite(ind_E):
                # cell to the east
                M[ind_here, ind_E] = 0 + d_I_JP1[i,j] + a_I_JP1[i,j]
            if np.isfinite(ind_N):
                # cell to the north
                M[ind_here, ind_N] = 0 + d_IP1_J[i,j] + a_IP1_J[i,j]
            if np.isfinite(ind_S):
                # cell to the south
                M[ind_here, ind_S] = 0 + d_IM1_J[i,j] + a_IM1_J[i,j]

    if sparse:
        M = M.tocsr()

    return M





### Simulate Model ###

def compute_linear_time_evolution(c0, M, saved_timesteps, duration,
        debug = False):
    """
    Compute linear time evolution of model: c(t+1) = Mc(t)

    Args:
    c0 (array): Initial state vector (a,1)
    M (array): Linear model matrix (a,a)
    saved_timesteps (list): Timesteps to save state
    duration (int): Number of timesteps to simulate
    debug (bool): If True, print progress every 10 steps

    Returns:
    saved_timesteps (list): List of timesteps where state was saved
    saved (list): List of state vectors at each saved timestep
    """
    r = np.zeros_like(c0)
    return compute_affine_time_evolution(c0, M, r, saved_timesteps,
        duration, debug)

def compute_linear_time_evolution_simple(c0, M, num_saved_timesteps,
    duration_per_saved_timestep,
        debug = False):
    """
    Compute linear time evolution of model: c(t+1) = Mc(t)

    Simplified with evenly spaced saved_timesteps.

    Args:
    c0 (array): Initial state vector (a,1)
    M (array): Linear model matrix (a,a)
    num_saved_timesteps (int): Number of saved_timesteps to save
    duration_per_saved_timestep (int): Number of timesteps between
        saved_timesteps
    debug (bool): If True, print progress every 10 steps

    Returns:
    list: 1D arrays representing saved_timesteps in time
    """
    saved_timesteps = [i*duration_per_saved_timestep for i in range(
        num_saved_timesteps)] # Evenly spaced saved_timesteps
    duration = num_saved_timesteps * duration_per_saved_timestep + 1 #
        Duration must be longer than the last saved timestep

    return compute_linear_time_evolution(c0, M, saved_timesteps, duration
        , debug)

def compute_affine_time_evolution(c0, M, r, saved_timesteps, duration,
        debug = False):
    """
    Compute affine time evolution model with affine term: c(t+1) = Mc(t)
        + r

    Args:
    c0 (array): Initial state vector (a,1)
    M (array): Linear model matrix (a,a)
    r (array): Affine term vector (a,1)
    saved_timesteps (list): Timesteps to save state
    duration (int): Number of timesteps to simulate
    debug (bool): If True, print progress every 10 steps

    Returns:
    saved_timesteps (list): List of timesteps where state was saved
    state_over_time (list): List of state vectors at each saved timestep
    """
    state_over_time=[]
    c = c0

    for i in range(duration): #Iterate over all timesteps
        if i%10 == 0 and debug:
            print(i)

        if i in saved_timesteps:
            state_over_time.append(c)
        c = M.dot(c) + r #Update the state

    return saved_timesteps, state_over_time

def compute_affine_time_evolution_simple(c0, M, r, num_saved_timesteps,
                                         duration_per_saved_timestep=1,
                                             debug = False):
    """
    Compute affine time evolution model with affine term: c(t+1) = Mc(t)
        + r

    Simplified with evenly spaced saved_timesteps.

    Args:
    c0 (array): Initial state vector (a,1)
    M (array): Linear model matrix (a,a)
    r (array): Affine term vector (a,1)
    num_saved_timesteps (int): Number of saved_timesteps to save
    duration_per_saved_timestep (int): Number of timesteps between
        saved_timesteps
    debug (bool): If True, print progress every 10 steps

    Returns:
    list: 1D arrays representing saved_timesteps in time
    """
    saved_timesteps = [i*duration_per_saved_timestep for i in range(
        num_saved_timesteps)] # Evenly spaced saved_timesteps
    duration = num_saved_timesteps * duration_per_saved_timestep + 1 #
        Duration must be longer than the last saved timestep

    return compute_affine_time_evolution(c0, M, r, saved_timesteps,
        duration, debug)
```

```python



### Plotting Heatmaps ###

def plot_multi_heatmap_time_evolution(saved_timesteps, many_states_over_time,
                                 nr, nc, titles, big_title,
                                 vmin=None, vmax=None, is_1d=False):
    """
    Display time evolution of multiple 1D or 2D arrays over time, side by side.

    Args:
    saved_timesteps (list or None): List of times that we save our state, or None if not shown
    many_states_over_time (list of lists): Each inner list contains 1D arrays of state at each timestep
    nr (int): Number of rows in each 2D array (or 1 for 1D state)
    nc (int): Number of columns in each 2D array (or length of 1D array)
    titles (list): Titles for each subplot
    big_title (str): Overall title for the entire plot
    vmin, vmax (float, optional): Min/max values for color scaling
    is_1d (bool): Whether the input state is 1D (True) or 2D (False)

    Displays heatmaps of several 1D or 2D arrays evolving over time.
    """
    # Reconstruct 1D arrays into 2D arrays
    if is_1d:
        plottable_states = [[state.reshape(1, -1) for state in state_over_time]
                            for state_over_time in many_states_over_time]
    else:
        plottable_states = [[state.reshape(nr, nc) for state in state_over_time]
                            for state_over_time in many_states_over_time]

    # Calculate global min and max for consistent color scaling if not provided
    if vmin is None:
        vmin = min(np.min(state) for state_over_time in many_states_over_time for state in state_over_time)
    if vmax is None:
        vmax = max(np.max(state) for state_over_time in many_states_over_time for state in state_over_time)

    # Create the heatmaps
    num_states = len(many_states_over_time)
    num_timesteps = len(many_states_over_time[0])  # Assume all state
        runs have the same number of timesteps

    # Calculate figure size and height ratios
    title_height = 0.5  # inches
    subplot_height = 2 if is_1d else 5  # inches
    total_height = title_height + (subplot_height * num_timesteps)
    fig_width = 5 * num_states + 1

    # Create figure with two subfigures
    fig = plt.figure(figsize=(fig_width, total_height))
    subfigs = fig.subfigures(2, 1, height_ratios=[title_height,
        subplot_height * num_timesteps])

    # Add the main title to the top subfigure
    subfigs[0].suptitle(big_title, fontsize=16)

    # Create gridspec for the bottom subfigure (plot grid)
    gs = subfigs[1].add_gridspec(num_timesteps, num_states + 1,
        width_ratios=[1]*num_states + [0.05])

    for i, state_over_time in enumerate(plottable_states):
        for j, state in enumerate(state_over_time):
            ax = subfigs[1].add_subplot(gs[j, i])
            im = ax.imshow(state, aspect='auto', cmap='coolwarm', vmin=
                vmin, vmax=vmax, origin='lower')

            # Set the title, including timestep if provided
            if saved_timesteps is None:
                ax.set_title(f'{titles[i]}')
            else:
                ax.set_title(f'{titles[i]}\nt={saved_timesteps[j]}')

            ax.set_xlabel('$x$')
            if not is_1d:
                ax.set_ylabel('$y$')
            else:
                ax.set_yticks([])

            # Add colorbar for each row
            if i == num_states - 1:  # Only for the last column
                cbar_ax = subfigs[1].add_subplot(gs[j, -1])
                plt.colorbar(im, cax=cbar_ax)

    plt.tight_layout()
    plt.show()
```

```python

def plot_1d_heatmap_time_evolution(saved_timesteps, state_over_time):
    """
    Displays the time evolution of a 1D state as a series of heatmaps.

    Args:
    saved_timesteps (list): List of time values corresponding to each
        state array
    state_over_time (list): List of 1D numpy arrays, each representing
        the state at a time saved

    Returns:
    None: Displays the plot using matplotlib

    """
    plot_multi_heatmap_time_evolution(
        saved_timesteps=saved_timesteps,
        many_states_over_time=[state_over_time],
        nr=1,
        nc=len(state_over_time[0]),
        titles=["Evolution in time"],
        big_title="Evolution in time"
    )



def plot_2d_heatmap_time_evolution(saved_timesteps, state_over_time, nr,
    nc, vmin = None, vmax = None):
    """
    Displays the time evolution of 2D state as a series of heatmaps.

    Args:
    saved_timesteps (list): List of time values corresponding to each
        state array in time
    state_over_time (list): List of 1D numpy arrays, each representing
        the state at a time saved
    nr (int): Number of rows in the 2D grid
    nc (int): Number of columns in the 2D grid
    vmin (float, optional): Minimum value for color scaling. If None,
        calculated from state.
    vmax (float, optional): Maximum value for color scaling. If None,
        calculated from state.

    Returns:
    None: Displays the plot using matplotlib
    """
    # We’re passing a single state over time, so we wrap it in another
```

```
        list
    plot_multi_heatmap_time_evolution(
        saved_timesteps=saved_timesteps,
        many_states_over_time=[state_over_time],
        nr=nr,
        nc=nc,
        titles=["Evolution in time"],
        big_title="Evolution in time",
        vmin=vmin,
        vmax=vmax
    )


def plot_multi_heatmap(many_states, nr, nc, titles, big_title, vmin=None,
     vmax=None):
    """
    Display heatmaps of multiple 2D states side by side for comparison.

    Args:
    many_states (list): List of 1D arrays to be reshaped into 2D
    nr, nc (int): Number of rows and columns for reshaping
    titles (list): Titles for each heatmap
    big_title (str): Overall title for the plot
    vmin, vmax (float, optional): Min/max values for color scaling

    Displays heatmaps side by side for comparison
    """
    # We're comparing at a single timestep, so we wrap each data array in
         its own list
    many_states_over_time = [[state] for state in many_states]

    plot_multi_heatmap_time_evolution(
        saved_timesteps=None,  # Single timestep
        many_states_over_time=many_states_over_time,
        nr=nr,
        nc=nc,
        titles=titles,
        big_title=big_title,
        vmin=vmin,
        vmax=vmax
    )



### Covariance functions ###

def compute_covariance_gaussian_dropoff(a, b, std_dev = 1):
    """
    Compute Gaussian covariance matrix with exponential dropoff.

    Calculates covariance using e^{ -d / 2s }, where d is squared
    Euclidean distance between a and b, and s is the standard deviation
        std_dev.

    Args:
    a, b (array-like): Input vectors, represents spatial coordinates
    std_dev (float): Standard deviation of gaussian dropoff (default=1)

    Returns:
    numpy.ndarray: Covariance matrix
    """

    d = np.linalg.norm(a - b, axis=-1)**2      #Distance between a and b
    covariance = np.exp( -d / (2*std_dev))       #Covariance matrix

    return covariance


def vector_of_2d_indices(nr, nc):
    """
    Convert 2D array of indices to a 1D vector representation containing
        the same indices.

    Args:
    nr (int): Number of rows
    nc (int): Number of columns

    Returns:
    numpy.ndarray: Shape (nr*nc, 2), each row is [row, col] index
    """
    y, x = np.mgrid[:nr, :nc] #Get meshgrid of all of our indices
    vector_form = np.column_stack((y.ravel(), x.ravel())) #Stack
    return vector_form


def compute_covariance_matrix_gaussian_dropoff(nr, nc, std_dev=1):
    """
    Compute covariance matrix with Gaussian dropoff for 2D grid.

    Compares all pairs of 2D indices and computes covariance using
    exponential dropoff function e^{ -d / 2s }, where
    - d is squared Euclidean distance between a and b
    - s is the standard deviation std_dev.

    Args:
    nr, nc (int): Number of rows and columns in 2D grid
    std_dev (float): Standard deviation for Gaussian dropoff (default=1)

    Returns:
    np.ndarray: Covariance matrix of size (nr*nc, nr*nc)
    """
    # Get all 2D indices as a 1D vector of (row, col) pairs
    index_vector = vector_of_2d_indices(nr, nc)

    # Prepare indices for broadcasting
    indices_i = index_vector[:, np.newaxis, :]  # Shape: (nr*nc, 1, 2)
    indices_j = index_vector[np.newaxis, :, :]  # Shape: (1, nr*nc, 2)

    # Compute covariance matrix
    covariance_matrix = compute_covariance_gaussian_dropoff(
        indices_i,
        indices_j,
        std_dev=std_dev
    )

    return covariance_matrix



### Generating Smooth Data ###

def add_random_circles(matrix, num_circles, radius, values):
    """
    Add random circles to a matrix for interesting initial conditions.

    Args:
    matrix (numpy.ndarray): 2D array to modify
    num_circles (int): Number of circles to add
    radius (int): Radius of circles
    values (list): Possible values for circles

    Returns:
    numpy.ndarray: Modified matrix with added circles
    """

    nr, nc = matrix.shape
    for _ in range(num_circles):
        center_r = np.random.randint(0, nr)
        center_c = np.random.randint(0, nc)
        r, c = np.ogrid[:nr, :nc]
        mask = ((r - center_r)**2 + (c - center_c)**2 <= radius**2)

        value = np.random.choice(values)
        matrix[mask] = value
    return matrix

def generate_random_vectors_mean_0_cov_C(nr, nc, C, num_vectors):
    """
    Generates random vectors from a normal distribution with mean 0 and
        covariance C.

    Args:
    nr, nc (int): Dimensions of the 2D grid
    C (np.ndarray): Covariance matrix
    num_vectors (int): Number of random vectors to generate

    Returns:
    tuple: (zs, Lzs)
        zs (list): Random normal vectors with mean 0 and covariance I
            z ~ N(0, I)
        Lzs (list): Random vectors with mean 0 and covariance C
            Lz ~ N(0, C)
    """
    size = nr * nc #Size of the grid
    zs = [np.random.randn(size, 1) for _ in range(num_vectors)] #Random
        normal vectors with mean 0 and covariance I

    # If z ~ N(0, I), then Lz ~ N(0, C)
    L = np.linalg.cholesky(C)
    Lzs = [L @ z for z in zs] #Random vectors with mean 0 and covariance
        C

    return zs, Lzs

def generate_random_vector_mean_0_cov_C(nr, nc, C):
    """
    Generates a random vector from a normal distribution with mean 0 and
        covariance C.

    Args:
    nr, nc (int): Dimensions of the 2D grid
    C (np.ndarray): Covariance matrix

    Returns:
    tuple: (z, Lz)
        z (np.ndarray): Random normal vector with mean 0 and covariance I
            z ~ N(0, I)
        Lz (np.ndarray): Random vector with mean 0 and covariance C
```

```python
            Lz ~ N(0, C)
    """
    zs, Lzs = generate_random_vectors_mean_0_cov_C(nr, nc, C, 1)
    return zs[0], Lzs[0]





def generate_true_and_first_guess_field_uniform_cov(C, nr, nc, gamma):
    """
    Generates two fields: a true field and a first-guess field, both with the same covariance C.

    Args:
    C (np.ndarray): Covariance matrix for both fields
    nr (int): Number of rows in the grid
    nc (int): Number of columns in the grid
    gamma (float): Fraction of the field shared between true and first-guess fields

    Returns:
    tuple: (true_field, first_guess_field)
        true_field (np.ndarray): Random field with covariance C
        first_guess_field (np.ndarray): Random field sharing a component with true_field, covariance C

    Notes:
    - f2 replaces f1 in the "first-guess" field to represent inaccuracies
    - The shared component (f0) represents the fraction of the field that is
      common between the true and first-guess fields
    """
    f0,f1,f2 = generate_random_vectors_mean_0_cov_C(nr, nc, C, 3)[1]

    true_field        = f0 * gamma + f1 * (1-gamma)
    first_guess_field = f0 * gamma + f2 * (1-gamma)

    return true_field, first_guess_field

def generate_true_and_first_guess_field(C_known, C_error, nr, nc):
    """
    Generates two fields: a true field and a first-guess field, both with specified covariances.

    Args:
    C_known (np.ndarray): Covariance matrix for the known part of the
```

```
        field
    C_error (np.ndarray): Covariance matrix for the error part of the
        field
    nr, nc (int): Dimensions of the 2D grid

    Returns:
    tuple: (true_field, first_guess_field)
        true_field (np.ndarray): Random field with covariance C_known +
            C_error
        first_guess_field (np.ndarray): Random field sharing a component
            with true_field, covariance C_known + C_error

    Notes:
    - f2 replaces f1 in the "first-guess" field to represent inaccuracies
    - The shared component (f0) represents the fraction of the field that
        is
      common between the true and first-guess fields
    """
    f0 = generate_random_vectors_mean_0_cov_C(nr, nc, C_known, 1)[1][0]
    f1, f2 = generate_random_vectors_mean_0_cov_C(nr, nc, C_error, 2)[1]

    true_field = f0 + f1
    first_guess_field = f0 + f2

    return true_field, first_guess_field


def generate_gaussian_field(n, nrv, ncv):
    """
    Randomly generates a 2D field composed of n Gaussian functions with
        distinct means and standard deviations.

    Args:
    n (int): Number of Gaussian functions to generate
    nrv (int): Number of rows in the field
    ncv (int): Number of columns in the field

    Returns:
    np.ndarray: A 2D array representing the generated Gaussian field

    Notes:
    - The field is made pseudo-periodic by creating three copies of each
        Gaussian function
    """
    mux = np.random.choice(ncv, n)
    muy = np.random.choice(range(2, nrv - 2), n)
    sigmax = np.random.uniform(1,ncv/4,n)
```

```python
    sigmay = np.random.uniform(1,nrv/4,n)

    #Combine all the gaussian functions to get the field

    v = np.zeros((nrv,ncv))
    for i in range(n):
        for x in range(ncv):
            for y in range(nrv):
                #We create three copies of our gaussian so that we get a pseudo-periodic field

                # Original Gaussian
                gauss  = np.exp(-((x-mux[i])**2/(2*sigmax[i]**2) + (y-muy[i])**2/(2*sigmay[i]**2)))
                # Shifted left
                gauss += np.exp(-((x-(mux[i]-ncv))**2/(2*sigmax[i]**2) + (y-muy[i])**2/(2*sigmay[i]**2)))
                # Shifted right
                gauss += np.exp(-((x-(mux[i]+ncv))**2/(2*sigmax[i]**2) + (y-muy[i])**2/(2*sigmay[i]**2)))

                v[y,x] += gauss

    return v


def generate_circular_field(v):
    """
    Generates a circular field by taking the gradient of the input field and rotating it by 90 degrees.

    Args:
    v (np.ndarray): Input 2D field

    Returns:
    tuple: (grad_v_x, grad_v_y)
        grad_v_x (np.ndarray): X-component of the circular field
        grad_v_y (np.ndarray): Y-component of the circular field
    """
    grad_v_y, grad_v_x = np.gradient(v)

    return -grad_v_y, grad_v_x


def create_random_model(nr, nc, dt, F,
                        num_gauss = 16,
                        DX_C = None, DY_C = None, DX_G = None, DY_G =
                                None, RAC = None,
                            cyclic_east_west=True, cyclic_north_south=False):
    """
    Creates a random model with a new velocity field and diffusivity
        field.

    - Velocity field is generated from a field of circular patterns, in
        order to create a field
    with low divergence.

    - Diffusivity field is generated randomly, with 0 diffusivity on the
        boundaries.

    Args:
    nr (int): Number of rows in the grid
    nc (int): Number of columns in the grid
    dt (float): Time step
    F (float): Forcing coefficient
    num_gauss (int, optional): Number of Gaussian functions for velocity
        field generation. Defaults to 16.
    DX_C, DY_C, DX_G, DY_G, RAC (np.ndarray, optional): Grid spacing and
        area parameters. If None, set to arrays of ones.
    cyclic_east_west (bool, optional): If True, applies cyclic conditions
         east-west. Defaults to True.
    cyclic_north_south (bool, optional): If True, applies cyclic
        conditions north-south. Defaults to False.

    Returns:
    tuple: (M, params)
        M (np.ndarray): Model matrix for 2D diffusion-advection-forcing
        params (dict): Dictionary of parameters used to create the model

    """
    # If none, just set everything to appropriately-sized array of 1's
    if DX_C is None:
        DX_C = np.ones((nr, nc+1))
    if DY_C is None:
        DY_C = np.ones((nr+1, nc))
    if DX_G is None:
        DX_G = np.ones((nr+1, nc))
    if DY_G is None:
        DY_G = np.ones((nr, nc+1))
    if RAC is None:
        RAC = np.ones((nr, nc))



    # Randomly generate diffusivities: must be positive
    KX = np.random.rand(nr, nc+1)
    KY = np.random.rand(nr+1, nc)
    KX = np.abs(KX)
    KY = np.abs(KY)

    # Randomly generate velocities as above
    gauss = generate_gaussian_field(num_gauss,nr+1,nc+1)
    VX, VY = generate_circular_field(gauss)

    # Create the model matrix
    params = {
        'nr': nr,
        'nc': nc,
        'dt': dt,
        'KX': KX,
        'KY': KY,
        'DX_C': DX_C,
        'DY_C': DY_C,
        'DX_G': DX_G,
        'DY_G': DY_G,
        'VX': 100*VX[:-1,:],
        'VY': 100*VY[:,:-1],
        'RAC': RAC,
        'F': F,
        'cyclic_east_west':  cyclic_east_west,
        'cyclic_north_south': cyclic_north_south
    }

    M = make_M_2d_diffusion_advection_forcing(**params)

    return M, params

def create_random_initial_ocean_state(nr, nc, C, num_circles, radius,
    values):
    """
    Creates a random initial ocean state with specified covariance matrix
        .

    Args:
    nr (int): Number of rows in the grid
    nc (int): Number of columns in the grid
    C (np.ndarray): Covariance matrix for the initial state
    num_circles (int): Number of random circles to add
    radius (int): Radius of circles
    values (list): Possible values for circles

    Returns:
    tuple: (z, Lz)
        z (np.ndarray): Random initial state with covariance C
        Lz (np.ndarray): Random initial state with covariance
    """

    z = np.random.rand(nr,nc)
    z = add_random_circles(z, num_circles, radius, values)
    z = z.reshape((nr*nc,1))

    L = np.linalg.cholesky(C)
    Lz = L @ z

    return z, Lz

def generate_world(nr, nc, dt, F, num_gauss=16, num_circles=20, radius=5,
     values=[2,-2], std_dev=2):
    """
    Generates a world with an ocean state, atmosphere, and model matrix.

    Args:
    nr (int): Number of rows in the grid
    nc (int): Number of columns in the grid
    dt (float): Time step
    F (float): Forcing parameter
    num_gauss (int, optional): Number of Gaussian functions for velocity
        field generation. Defaults to 16.
    num_circles (int, optional): Number of circles for ocean state
        generation. Defaults to 20.
    radius (int, optional): Radius of circles for ocean state generation.
         Defaults to 5.
    values (list, optional): Values of circles for ocean state generation
        . Defaults to [2,-2].
    std_dev (int, optional): Standard deviation for Gaussian dropoff.
        Defaults to 2.

    Returns:
    tuple: (C, c0, f, M)
        C (np.ndarray): Covariance matrix
        c0 (np.ndarray): Initial ocean state
        f (np.ndarray): Atmosphere
        M (np.ndarray): Model matrix
    """
    # Generate covariance matrix
    C = compute_covariance_matrix_gaussian_dropoff(nr, nc, std_dev)

    # Generate model matrix
    M, params = create_random_model(nr, nc, dt, F, num_gauss=num_gauss)

    # Generate initial ocean state
    _, c0 = create_random_initial_ocean_state(nr, nc, C, num_circles=
        num_circles, radius=radius, values=values)

    # Generate atmosphere
    _, f = generate_random_vector_mean_0_cov_C(nr, nc, C)

    return C, c0, f, M

#C, c0, f, M = generate_world(50, 50, 0.1, 1, num_gauss=16, num_circles
    =20, radius=5, values=[2,-2], std_dev=2)

# Get magnitude of f
#f = f/3
#f_mag = np.linalg.norm(f)
#print(f'Magnitude of f: {f_mag:.2f}')


### Observe field ###

def observe(real, sigma, num_observations):
    """
    Generates noisy observations of a true state at randomly selected
        indices.

    Args:
    real (np.ndarray): The true state of the system, as a 1D array
    sigma (float): The standard deviation of the observation noise
    num_observations (int): The number of observations to make

    Returns:
    tuple: (indices, observations)
        indices (np.ndarray): Array of randomly selected indices for
            observation
        observations (np.ndarray): Noisy observations of the true state
            at the selected indices

    Notes:
    - Observations are made by adding Gaussian noise to the true state
        values
    - The noise is generated as a 2D column vector
    - Indices are selected without replacement, ensuring unique
        observation points
    """
    # We randomly select which indices to observe
```

```python
    indices = np.random.choice(len(real), num_observations, replace=False
        )

    # We observe the true state plus Gaussian noise
    noise = np.random.normal(0, sigma, (num_observations, 1))

    observations = real[indices] + noise

    return indices, observations

def fill_nan_map_with_observations(indices, observations, nr, nc):
    """
    Maps observations to their corresponding positions in a 2D grid,
        filling the rest with NaNs.

    Args:
    indices (np.ndarray): Indices of the observations in the flattened
        grid
    observations (np.ndarray): Observed values
    nr (int): Number of rows in the grid
    nc (int): Number of columns in the grid

    Returns:
    np.ndarray: 2D array with observations at their corresponding
        positions and NaNs elsewhere
    """
    observed_state_2d = np.full((nr, nc), np.nan)
    observed_state_2d.flat[indices] = observations.flatten()
    return observed_state_2d

def interpolate_observation_map(observed_state_2d, extend=False):
    """
    Interpolates a 2D grid of observed values, to fill in NaN values (
        representing unobserved points).

    Args:
    observed_state_2d (np.ndarray): 2D array with observed values and
        NaNs
    extend (bool, optional): If True, extends the observed state by three
         copies horizontally. Defaults to False.

    Returns:
    np.ndarray: 2D array of interpolated values

    Notes:
    - If extend is True:
        - The observed state is extended by three copies horizontally
        - Interpolation is performed on the extended grid
        - The middle third of the interpolated result is returned
    - If extend is False:
        - Interpolation is performed on the original grid
    - Linear interpolation is used, with NaN values for points outside
        the convex hull of observations
    """
    nr, nc = observed_state_2d.shape

    if extend:
        # Extend state by three copies
        observed_state_2d = np.hstack((observed_state_2d,
            observed_state_2d, observed_state_2d))

    extended_nr, extended_nc = observed_state_2d.shape

    # Create grid coordinates
    x, y = np.meshgrid(np.arange(extended_nc), np.arange(extended_nr))

    # Find non-NaN indices and values
    observed_indices = np.where(~np.isnan(observed_state_2d.flatten()))
        [0]
    observed_values = observed_state_2d.flatten()[observed_indices]

    # Create points and grid for interpolation
    points = np.column_stack((x.flat[observed_indices], y.flat[
        observed_indices]))
    grid_x, grid_y = np.meshgrid(np.arange(extended_nc), np.arange(
        extended_nr))

    # Perform interpolation
    interpolated_2d = griddata(points, observed_values, (grid_x, grid_y),
         method='linear', fill_value=np.nan)

    if extend:
        # Extract the middle third
        return interpolated_2d[:, nc:2*nc]
    else:
        return interpolated_2d


def observe_over_time(ocean_states, sigma, num_obs_per_timestep, nr, nc):
    """
    Observes the ocean state at each timestep, and places those
        observations in
    a 2d array with NaNs for unobserved points.

    Args:
    ocean_states (list): List of ocean states at each timestep
    sigma (float): Standard deviation of observation noise
    num_obs_per_timestep (int): Number of observations per timestep
    nr (int): Number of rows in the grid
    nc (int): Number of columns in the grid

    Returns:
    list: List of observed ocean states at each timestep, each as a 2D
        array with NaNs for unobserved points

    """
    # Observe ocean state at each timestep
    indices_and_observations_over_time = [observe(ocean_state, sigma,
        num_obs_per_timestep)
                                          for ocean_state in
                                              ocean_states]

    # Place on a map: unobserved points are filled with NaN
    observed_state_over_time_2d = [fill_nan_map_with_observations(indices
        , observations_t, nr, nc)
                                  for indices, observations_t in
                                      indices_and_observations_over_time
                                      ]

    return observed_state_over_time_2d




### Compute Adjoints ###

def compute_Jt(xt_true, xt_guess):
    """
    Computes squared loss between two vectors at time t.

    Args:
    xt_true (np.ndarray): True state vector at time t
    xt_guess (np.ndarray): Guessed state vector at time t

    Returns:
    float: Squared loss, or 0 if no valid terms
    """

    # Sum over all valid terms, using numpy to treat nans as zeros
    result = np.nansum((xt_true - xt_guess)**2)
    if np.isnan(result):
```

```python
        return 0
    else:
        return result

def compute_J(x_true, x_guess):
    """
    Computes total squared loss between two vectors across all timesteps.

    Args:
    x_true (list): List of true state vectors at each timestep
    x_guess (list): List of guessed state vectors at each timestep

    Returns:
    float: Total squared loss across all timesteps
    """
    return np.sum([
                compute_Jt(x_true[i], x_guess[i]) for i in range(len(
                    x_true))]
                )

def compute_DJ_Dxt(xt_true, xt_guess):
    """
    Computes partial derivative of squared loss w.r.t. guessed state at
        time t.

    Args:
    xt_true (np.ndarray): True state vector at time t
    xt_guess (np.ndarray): Guessed state vector at time t

    Returns:
    np.ndarray: Partial derivative of loss, with NaNs treated as 0
    """
    return np.nan_to_num( 2*(xt_guess - xt_true), nan = 0 )


def compute_adjoints(DJ_Dx, dxtp1_dxt):
    """
    Computes adjoint variables for optimization using backwards-time
        recursion.

    Args:
    DJ_Dx (list): List of partial derivatives of loss w.r.t. state at
        each timestep
    dxtp1_dxt (list): List of total derivatives of next state w.r.t.
        current state at each timestep

    Returns:
```

```
    list: Adjoint variables for each timestep, in forward time order
    """

    tau = len(DJ_Dx)
    adjoints = [0] * tau # Initialize list of adjoints

    adjoints[tau-1] = DJ_Dx[tau-1]

    for t in range(tau-2, -1, -1):  # Backwards in time
        adjoint = DJ_Dx[t] + dxtp1_dxt[t].dot( adjoints[t+1] )

        adjoints[t] = adjoint

    return adjoints

def compute_dJ_df(M, F, observed_state_over_time,
    simulated_state_over_time):
    """
    Computes the gradient of the loss with respect to the forcing field f
         for the linear model:
    x(t+1) = Mx(t) + Ff

    Args:
    M (np.ndarray): Model matrix
    F (float): Forcing coefficient
    observed_state_over_time (list): List of observed states at each
        timestep
    simulated_state_over_time (list): List of simulated states at each
        timestep

    Returns:
    np.ndarray: Gradient of the loss with respect to the forcing field f
    """
    num_timesteps = len(observed_state_over_time)
    vec_length = len(observed_state_over_time[0])

    #Compute adjoints
    DJ_Dx = [compute_DJ_Dxt(observed_state_over_time[i],
        simulated_state_over_time[i])
             for i in range(num_timesteps)] # partial J / partial x(t)

    dxtp1_dxt = [M.T for i in range(num_timesteps-1)] #dx(t+1)/dx(t)

    adjoints = compute_adjoints(DJ_Dx, dxtp1_dxt) # dJ/dx(t) = lambda(t)

    # Compute gradient for each timestep: how f being applied at time t
        affects J
    dJ_dft = [ F * adjoint for adjoint in adjoints[1:] ] # dJ/df(t) = dx(
        t+1)/df(t) dJ/dx(t+1)
    dJ_dft.append(np.zeros((vec_length,1))) # dJ/df(tau) = 0

    #f is applied the same at all timesteps
    dJ_df = np.sum(dJ_dft, axis=0) # dJ/df = sum_t dJ/df(t)
    return dJ_df

### Gradient Descent ###

losses_template = { #Losses at each iteration
        "ocean_misfit": [],
        "atmosphere_misfit": [],
        "mahalanobis(covariance similarity)": [],
    }


def update_losses(losses, ocean_states_observed, ocean_states_simulated,
    f_guess, f_adjust, f_true, C_error):
    """
    Updates the losses dictionary with new loss values for ocean,
        atmosphere, and control adjustment.

    Args:
    losses (dict): Dictionary containing lists of loss values
    ocean_states_observed (list): List of observed ocean states
    ocean_states_simulated (list): List of simulated ocean states
    f_guess (np.ndarray): Initial guess for the atmospheric forcing field
    f_adjust (np.ndarray): Adjustment to the atmospheric forcing field
    f_true (np.ndarray): True atmospheric forcing field
    C_error (np.ndarray): Covariance matrix for the control error

    Returns:
    dict: Updated losses dictionary with new loss values appended
    """
    f_i = f_guess + f_adjust

    # Compute losses
    ocean_loss_i = compute_J(ocean_states_observed,
        ocean_states_simulated) # J_{ocean} = J
    atmos_loss_i = compute_Jt(f_true, f_i)
                                         # J_{atm} = misfit of atm

    mahal = f_adjust.T @ np.linalg.inv(C_error) @ f_adjust # C_error
        should be the covariance of our adjustment
    mahal_loss_i = np.linalg.norm(mahal)

    #Store losses
    losses["ocean_misfit"].append(ocean_loss_i)
    losses["atmosphere_misfit"].append(atmos_loss_i)
    losses["mahalanobis(covariance similarity)"].append(mahal_loss_i)

    return losses

possible_debug_vars = { #Debug variables to compute at each iteration
    "Norm of s_i": [],
    "Expected Delta J w simple gd": [],
    "Expected Delta J w update rule": [],
    "Norm of simple gd ui": [],
    "Norm of update rule ui": [],
    "Normalized dot product $a_i$ and $u_i$": [],
    "$s_i^T Cs_i$": [],
    "$a_i$": [],
    "Normalized $a_i^T s_i$": [],
    "Normalized $-Cs_i \cdot a_i$": [],
    "Norm of $a_i$": [],
    "Actual Delta J": []
}

def update_debug_vars(debug_vars, x0, M, F, f_guess, f_adjust,
                      C_known, C_error,
                      s, step_size, ui,
                      ocean_states_simulated, ocean_states_observed):
    """
    Updates the debug variables dictionary with various metrics for
        gradient descent analysis.

    Args:
    debug_vars (dict): Dictionary containing lists of debug variable
        values
    x0 (np.ndarray): Initial ocean state
    M (np.ndarray): Model matrix
    F (float): Forcing coefficient
    f_guess (np.ndarray): Initial guess for the atmospheric forcing field
    f_adjust (np.ndarray): Adjustment to the atmospheric forcing field
    C_known (np.ndarray): Covariance matrix for the known portion of the
        control
    C_error (np.ndarray): Covariance matrix for the control error
    s (np.ndarray): Gradient of the loss with respect to the forcing
        field
    step_size (float): Step size for gradient descent
    ui (np.ndarray): Update vector for the current iteration
    ocean_states_simulated (list): List of simulated ocean states
    ocean_states_observed (list): List of observed ocean states

    Returns:
    dict: Updated debug_vars dictionary with new values appended to each
        metric
    """
    #Initialize useful variables
    num_timesteps = len(ocean_states_observed)
    ui_simple_gd = -step_size * s
    delta = s.T @(ui_simple_gd)

    #Compute debug vars
    norm_s = np.linalg.norm(s)
    exp_delta_J_simple_gd = (s.T @ ui_simple_gd)[0,0]
    exp_delta_J_update_rule = (s.T @ ui)[0,0]
    norm_simple_ui = np.linalg.norm(ui_simple_gd)
    norm_ui = np.linalg.norm(ui)
    norm_dot_product = (f_adjust.T @ ui)[0,0] / (np.linalg.norm(f_adjust)
         * np.linalg.norm(ui))
    sTCs = (s.T @ C_error @ s)[0,0]
    ai = f_adjust
    normalized_aiTs = (f_adjust.T @ s)[0,0] / (np.linalg.norm(f_adjust) *
         np.linalg.norm(s))
    normalized_Csdotai = (f_adjust.T @ (C_error @ s))[0,0] / (np.linalg.
        norm(f_adjust) * np.linalg.norm(C_error @ s))
    norm_ai = np.linalg.norm(f_adjust)

    #Store debug vars
    debug_vars["Norm of s_i"].append(norm_s)
    debug_vars["Expected Delta J w simple gd"].append(
        exp_delta_J_simple_gd)
    debug_vars["Expected Delta J w update rule"].append(
        exp_delta_J_update_rule)
    debug_vars["Norm of simple gd ui"].append(norm_simple_ui)
    debug_vars["Norm of update rule ui"].append(norm_ui)
    debug_vars["Normalized dot product $a_i$ and $u_i$"].append(
        norm_dot_product)
    debug_vars["$s_i^T Cs_i$"].append(sTCs)
    debug_vars["$a_i$"].append(ai)
    debug_vars["Normalized $a_i^T s_i$"].append(normalized_aiTs)
    debug_vars["Normalized $-Cs_i \cdot a_i$"].append(-normalized_Csdotai
        )
    debug_vars["Norm of $a_i$"].append(norm_ai)

    #Handle last debug var: Actual Delta J
    f_new = f_guess + f_adjust + ui
    _, new_ocean_states_simulated = compute_affine_time_evolution_simple(
        x0, M, F*f_new, num_timesteps)
```

```python

    new_J = compute_J(ocean_states_observed, new_ocean_states_simulated)
    old_J = compute_J(ocean_states_observed, ocean_states_simulated)

    actual_delta_J = new_J - old_J

    debug_vars["Actual Delta J"].append(actual_delta_J)

    return debug_vars


def gradient_descent_template(M, F, f_true, f_guess, C_known, C_error,   # World parameters
                              x0, num_timesteps,                          # Simulation parameters
                              ocean_states_observed, num_iters, step_size,      # Optimization parameters
                              update_rule, update_params, disp=False):
    """
    Perform gradient descent to optimize the atmospheric forcing field.

    The step we take at each iteration is computed using a modified update rule, and extra parameters as necessary.

    Args:
        M: Model matrix
        F: Scalar constant for forcing
        f_true: True atmospheric forcing field
        f_guess: Initial guess for the atmospheric forcing field
        x0: Initial ocean state
        num_timesteps: Number of timesteps
        ocean_states_observed: Observed ocean states
        num_iters: Number of iterations
        step_size: Step size for gradient descent
        C_known: Covariance matrix for the known portion of the control
        C_error: Covariance matrix for the control error
        update_params: Function to compute the update rule to take at each iteration
        extra_params: Extra parameters to pass to the update rule
        disp: Flag to print information

    Returns:
        f: Optimized atmospheric forcing field
        losses: Dictionary of losses
            ocean_misfit: Ocean loss at each iteration
            atmosphere: Atmospheric loss at each iteration
            $a_iC^{-1}a_i$: Mahalanobis distance for the control
                adjustment at each iteration
    """
    size = f_guess.shape[0]
    f_adjust = np.zeros((size,1))

    losses = copy.deepcopy(losses_template)
    debug_vars = copy.deepcopy(possible_debug_vars)

    for i in range(num_iters):
        if i%10==0 and disp:
            print("Iteration", i)

        #Compute results of previous update rule
        f_i = f_guess + f_adjust #f_i = f_0 + a_i
        _, ocean_states_simulated = compute_affine_time_evolution_simple(
            x0, M, F*f_i, num_timesteps)


        # Compute and store losses
        losses = update_losses(losses, ocean_states_observed,
            ocean_states_simulated, f_guess, f_adjust, f_true, C_error)

        # Compute and store debug variables

        s = compute_dJ_df(M, F, ocean_states_observed,
            ocean_states_simulated)
        ui = update_rule(i, s, step_size, f_adjust, *update_params) #
            Update rule

        debug_vars = update_debug_vars(debug_vars, x0, M, F, f_guess,
            f_adjust,
                                            C_known, C_error,
                                            s, step_size, ui,
                                            ocean_states_simulated,
                                                ocean_states_observed)

        # Apply update rule to f_adjust

        f_adjust = f_adjust + ui



    return f_adjust, losses, debug_vars


def simple_gradient_update_rule(curr_iter, s, step_size, f_adjust):
```

```python
    """
    Computes the update step for simple gradient descent.

    Args:
    curr_iter (int): Current iteration number (unused in this function)
    s (np.ndarray): Gradient of the loss with respect to the forcing field
    step_size (float): Step size for gradient descent
    f_adjust (np.ndarray): Current adjustment to the forcing field (unused in this function)

    Returns:
    np.ndarray: Update step for the forcing field adjustment
    """
    return -step_size * s  # Just use the gradient of the loss

def simple_gradient_descent(M, F, f_true, f_guess, C_known, C_error,      # World parameters
                            x0, timesteps,                                                  # Simulation parameters
                            ocean_states_observed, num_iters, step_size,        # Optimization parameters
                            disp=False):                                                    # Optimization method
    """
    Implements simple gradient descent for optimizing the atmospheric forcing field.

    Args:
    M (np.ndarray): Model matrix
    F (float): Forcing coefficient
    f_true (np.ndarray): True atmospheric forcing field
    f_guess (np.ndarray): Initial guess for the atmospheric forcing field
    C_known (np.ndarray): Covariance matrix for the known portion of the control
    C_error (np.ndarray): Covariance matrix for the control error
    x0 (np.ndarray): Initial ocean state
    timesteps (int): Number of timesteps for simulation
    ocean_states_observed (list): List of observed ocean states
    num_iters (int): Number of iterations for gradient descent
    step_size (float): Step size for gradient descent
    disp (bool, optional): If True, display progress. Defaults to False.

    Returns:
    tuple: (f_adjust, losses, debug_vars)
```

```
        f_adjust (np.ndarray): Final adjustment to the atmospheric
            forcing field
        losses (dict): Dictionary of loss values over iterations
        debug_vars (dict): Dictionary of debug variables over iterations
    """
    return gradient_descent_template(M, F, f_true, f_guess, C_known,
        C_error,
                                     x0, timesteps,
                                     ocean_states_observed, num_iters,
                                         step_size,
                                     simple_gradient_update_rule, [],
                                         disp)



def cholesky_update_rule(curr_iter, s, step_size, f_adjust, C_error):
    """
    Computes the update step using Cholesky decomposition of the error
        covariance matrix.

    Args:
    curr_iter (int): Current iteration number (unused in this function)
    s (np.ndarray): Gradient of the loss with respect to the forcing
        field
    step_size (float): Step size for gradient descent
    f_adjust (np.ndarray): Current adjustment to the forcing field (
        unused in this function)
    C_error (np.ndarray): Covariance matrix for the control error

    Returns:
    np.ndarray: Update step for the forcing field adjustment

    Notes:
    Applies the Cholesky decomposition L of C_error to s, then rescales
        the result
    to match the original gradient's magnitude.
    """
    # Apply cholesky decomposition L : C_error = L @ L.T
    L = np.linalg.cholesky(C_error)
    cholesky_s = L @ s

    # Rescale so magnitude is the same
    rescaled_cholesky_s = cholesky_s * (np.linalg.norm(s) / np.linalg.
        norm(cholesky_s))
    step = - step_size * rescaled_cholesky_s

    return step
```

```python

def cholesky_gradient_descent(M, F, f_true, f_guess, C_known, C_error,          # World parameters
                              x0, timesteps,                                     # Simulation parameters
                              ocean_states_observed, num_iters, step_size,    # Optimization parameters
                              disp=False):                                       # Optimization method
    """
    Implements gradient descent using Cholesky decomposition for optimizing the atmospheric forcing field.

    Args:
    M (np.ndarray): Model matrix
    F (float): Forcing coefficient
    f_true (np.ndarray): True atmospheric forcing field
    f_guess (np.ndarray): Initial guess for the atmospheric forcing field
    C_known (np.ndarray): Covariance matrix for the known portion of the control
    C_error (np.ndarray): Covariance matrix for the control error
    x0 (np.ndarray): Initial ocean state
    timesteps (int): Number of timesteps for simulation
    ocean_states_observed (list): List of observed ocean states
    num_iters (int): Number of iterations for gradient descent
    step_size (float): Step size for gradient descent
    disp (bool, optional): If True, display progress. Defaults to False.

    Returns:
    tuple: (f_adjust, losses, debug_vars)
        f_adjust (np.ndarray): Final adjustment to the atmospheric forcing field
        losses (dict): Dictionary of loss values over iterations
        debug_vars (dict): Dictionary of debug variables over iterations
    """
    return gradient_descent_template(M, F, f_true, f_guess, C_known, C_error,
                                     x0, timesteps,
                                     ocean_states_observed, num_iters, step_size,
                                     cholesky_update_rule, [C_error], disp)


def cov_constraint_J_update_rule(curr_iter, s, step_size, f_adjust,
    C_error, weight_cov_term):
    """
    Computes the update step using a covariance constraint on the loss function.

    Args:
    curr_iter (int): Current iteration number (unused in this function)
    s (np.ndarray): Gradient of the loss with respect to the forcing field
    step_size (float): Step size for gradient descent
    f_adjust (np.ndarray): Current adjustment to the forcing field
    C_error (np.ndarray): Covariance matrix for the control error
    weight_cov_term (float): Weight for the covariance constraint term

    Returns:
    np.ndarray: Update step for the forcing field adjustment

    Notes:
    Adds a weighted covariance constraint term to the original gradient,
    then rescales the result to match the original gradient's magnitude.
    """
    cov_term_grad = 2 * np.linalg.inv(C_error) @ f_adjust  # Covariance term

    s_prime = s + weight_cov_term * cov_term_grad  # Gradient of J' with respect to the forcing field

    norm_s_prime = s_prime * (np.linalg.norm(s) / np.linalg.norm(s_prime))  # Rescale magnitude to match original

    return -step_size * norm_s_prime

def cov_constraint_J_gradient_descent(M, F, f_true, f_guess, C_known, C_error,       # World parameters
                                      x0, timesteps,                                  # Simulation parameters
                                      ocean_states_observed, num_iters, step_size,    # Optimization parameters
                                      weight_cov_term, disp=False):                   # Optimization method
    """
    Implements gradient descent with a covariance constraint for optimizing the atmospheric forcing field.

    Args:
    M (np.ndarray): Model matrix
    F (float): Forcing coefficient
    f_true (np.ndarray): True atmospheric forcing field
    f_guess (np.ndarray): Initial guess for the atmospheric forcing field
    C_known (np.ndarray): Covariance matrix for the known portion of the control
    C_error (np.ndarray): Covariance matrix for the control error
    x0 (np.ndarray): Initial ocean state
    timesteps (int): Number of timesteps for simulation
    ocean_states_observed (list): List of observed ocean states
    num_iters (int): Number of iterations for gradient descent
    step_size (float): Step size for gradient descent
    weight_cov_term (float): Weight for the covariance constraint term
    disp (bool, optional): If True, display progress. Defaults to False.

    Returns:
    tuple: (f_adjust, losses, debug_vars)
        f_adjust (np.ndarray): Final adjustment to the atmospheric forcing field
        losses (dict): Dictionary of loss values over iterations
        debug_vars (dict): Dictionary of debug variables over iterations
    """
    return gradient_descent_template(M, F, f_true, f_guess, C_known, C_error,
                                     x0, timesteps,
                                     ocean_states_observed, num_iters, step_size,
                                     cov_constraint_J_update_rule, [C_error, weight_cov_term], disp)

def dan_update_rule(curr_iter, s, step_size, f_adjust, C_error):
    """
    Computes the update step using Dan's method for improving the Mahalanobis distance.

    Args:
    curr_iter (int): Current iteration number (unused in this function)
    s (np.ndarray): Gradient of the loss with respect to the forcing field
    step_size (float): Step size for gradient descent
    f_adjust (np.ndarray): Current adjustment to the forcing field (unused in this function)
    C_error (np.ndarray): Covariance matrix for the control error

    Returns:
    np.ndarray: Update step for the forcing field adjustment
```

```python

    Notes:
    Modifies the gradient direction to improve the Mahalanobis distance of u_i (update rule) while
    maintaining the desired improvement in the loss function J.
    """
    ui_simple_gd = -step_size * s  # Pre-dan step
    delta = s.T @ (ui_simple_gd)   # Compute desired improvement of J

    new_vec = (C_error @ s) / (s.T @ C_error @ s)  # Direction modified to improve Mahalanobis distance

    return delta * new_vec

def dan_gradient_descent(M, F, f_true, f_guess, C_known, C_error,       # World parameters
                         x0, timesteps,                                  # Simulation parameters
                         ocean_states_observed, num_iters, step_size,   # Optimization parameters
                         disp=False):                                    # Optimization method
    """
    Implements gradient descent using Dan's method for optimizing the atmospheric forcing field.

    Args:
    M (np.ndarray): Model matrix
    F (float): Forcing coefficient
    f_true (np.ndarray): True atmospheric forcing field
    f_guess (np.ndarray): Initial guess for the atmospheric forcing field
    C_known (np.ndarray): Covariance matrix for the known portion of the control
    C_error (np.ndarray): Covariance matrix for the control error
    x0 (np.ndarray): Initial ocean state
    timesteps (int): Number of timesteps for simulation
    ocean_states_observed (list): List of observed ocean states
    num_iters (int): Number of iterations for gradient descent
    step_size (float): Step size for gradient descent
    disp (bool, optional): If True, display progress. Defaults to False.

    Returns:
    tuple: (f_adjust, losses, debug_vars)
        f_adjust (np.ndarray): Final adjustment to the atmospheric forcing field
        losses (dict): Dictionary of loss values over iterations
        debug_vars (dict): Dictionary of debug variables over iterations
    """
    return gradient_descent_template(M, F, f_true, f_guess, C_known, C_error,
                                     x0, timesteps,
                                     ocean_states_observed, num_iters, step_size,
                                     dan_update_rule, [C_error], disp)


def dan_modified_update_rule(curr_iter, s, step_size, f_adjust, C_error):
    """
    Computes the update step using a modified version of Dan's method for improving the Mahalanobis distance.

    Args:
    curr_iter (int): Current iteration number (unused in this function)
    s (np.ndarray): Gradient of the loss with respect to the forcing field
    step_size (float): Step size for gradient descent
    f_adjust (np.ndarray): Current adjustment to the forcing field
    C_error (np.ndarray): Covariance matrix for the control error

    Returns:
    np.ndarray: Update step for the forcing field adjustment

    Notes:
    Modifies the gradient direction to improve the Mahalanobis distance while
    maintaining the desired improvement in the loss function J. This version
    intends to improve Mahalanobis distance of a_i+u_i (f_adjust+gradient update), instead of just u_i.
    """
    ui_simple_gd = -step_size * s  # Pre-dan step
    delta = s.T @ (ui_simple_gd)   # Compute desired improvement of J

    new_vec = (C_error @ s) / (s.T @ C_error @ s)  # Direction modified to improve Mahalanobis distance
    vec_scale = delta + s.T @ f_adjust

    return -f_adjust + vec_scale * new_vec

def dan_modified_gradient_descent(M, F, f_true, f_guess, C_known, C_error,       # World parameters
                                  x0, timesteps,                                      # Simulation parameters
                                  ocean_states_observed, num_iters, step_size,   # Optimization parameters
                                  disp=False):                                                                    # Optimization method
    """
    Implements gradient descent using a modified version of Dan’s method for optimizing the atmospheric forcing field.

    Args:
    M (np.ndarray): Model matrix
    F (float): Forcing coefficient
    f_true (np.ndarray): True atmospheric forcing field
    f_guess (np.ndarray): Initial guess for the atmospheric forcing field
    C_known (np.ndarray): Covariance matrix for the known portion of the control
    C_error (np.ndarray): Covariance matrix for the control error
    x0 (np.ndarray): Initial ocean state
    timesteps (int): Number of timesteps for simulation
    ocean_states_observed (list): List of observed ocean states
    num_iters (int): Number of iterations for gradient descent
    step_size (float): Step size for gradient descent
    disp (bool, optional): If True, display progress. Defaults to False.

    Returns:
    tuple: (f_adjust, losses, debug_vars)
        f_adjust (np.ndarray): Final adjustment to the atmospheric forcing field
        losses (dict): Dictionary of loss values over iterations
        debug_vars (dict): Dictionary of debug variables over iterations
    """
    return gradient_descent_template(M, F, f_true, f_guess, C_known, C_error,
                                     x0, timesteps,
                                     ocean_states_observed, num_iters, step_size,
                                     dan_modified_update_rule, [C_error], disp)


### Gradient Descent Testing ###

def compare_gd_methods_once(M, F, f_true, f_guess, C_known, C_error,
                            x0, timesteps,
                            ocean_states_observed, num_iters, step_size,
                            methods, disp=False):
```

```
    """
    Runs multiple methods of gradient descent on the same dataset and
        compares their performance.

    Args:
    M (np.ndarray): Model matrix
    F (float): Forcing coefficient
    f_true (np.ndarray): True atmospheric forcing field
    f_guess (np.ndarray): Initial guess for the atmospheric forcing field
    C_known (np.ndarray): Covariance matrix for the known portion of the
        control
    C_error (np.ndarray): Covariance matrix for the control error
    x0 (np.ndarray): Initial ocean state
    timesteps (int): Number of timesteps for simulation
    ocean_states_observed (list): List of observed ocean states
    num_iters (int): Number of iterations for gradient descent
    step_size (float): Step size for gradient descent
    methods (list): List of gradient descent methods to compare. Each
        method is a list of the form
                    ["Method Name", method_func, extra_params]
    disp (bool, optional): If True, display progress. Defaults to False.

    Returns:
    dict: A dictionary where keys are method names and values are tuples
        containing:
          - f_adjust (np.ndarray): Final adjustment to the atmospheric
              forcing field
          - losses (dict): Dictionary of loss values over iterations
          - debug_vars (dict): Dictionary of debug variables over
              iterations

    Notes:
    This function applies each specified gradient descent method to the
        same initial conditions
    and dataset, allowing for direct comparison of their performance.
    """
    results = {}

    for method_name, method_func, extra_params in methods:
        if disp:
            print(f"Running method {method_name}")

        f_adjust, losses, debug_vars = gradient_descent_template(M, F,
            f_true, f_guess, C_known, C_error,
                                                    x0, timesteps,
                                                    ocean_states_observed
                                                        , num_iters,
                                                                     step_size,
                                                                 method_func,
                                                                     extra_params,
                                                                     disp)
        results[method_name] = (f_adjust, losses, debug_vars)

    return results

def compare_gd_methods_many_times(nr, nc, dt, F, gamma, sigma,
    num_obs_per_timestep,
                                      num_timesteps, num_iters, step_size,
                                      C_known, C_error, methods, num_runs,
                                          disp=False):
    """
    Create many different sets of data.
    For each one, we will run each of our gradient descent methods.
    Once we finish, we average losses across all runs.

    Args:
    nr (int): Number of rows in the grid
    nc (int): Number of columns in the grid
    dt (float): Time step
    F (float): Forcing parameter
    gamma (float): Proportion of the control vector that is correct
    sigma (float): Standard deviation of observation noise
    num_obs_per_timestep (int): Number of observations per timestep
    num_timesteps (int): Number of timesteps
    num_iters (int): Number of iterations of gradient descent
    step_size (float): Step size for gradient descent
    methods (list): List of gradient descent methods to compare
    num_runs (int): Number of times to run the whole optimization process
    disp (bool): If True, print progress

    Returns:
    dict: Dictionary containing averaged losses and debug variables for
        each method
    """

    # Initialize results dictionary
    losses =     {method[0]: copy.deepcopy(losses_template)
                for method in methods}
    debug_vars = {method[0]: copy.deepcopy(possible_debug_vars)
                  for method in methods}

    for run in range(num_runs):
        if disp:
            print(f"Run {run + 1}/{num_runs}")

        # Generate world
        C_control, x0, _, M = generate_world(nr, nc, dt, F)
        C_known, C_error = C_control * gamma, C_control * (1-gamma)

        f_true, f_guess = generate_true_and_first_guess_field(C_known,
            C_error, nr, nc)

        # Run the simulation with the true and guessed control vector
        _, real_state_over_time  = compute_affine_time_evolution_simple(
            x0, M, F*f_true,  num_timesteps)

        observed_state_over_time_2d = observe_over_time(
            real_state_over_time, sigma,
                                                               num_obs_per_timestep
                                                                   , nr,
                                                                   nc)

        observed_state_over_time = [np.reshape(observed_state_2d, (nr*nc,
             1))
                                    for observed_state_2d in
                                        observed_state_over_time_2d]

        # Run each method
        for method_name, method_func, extra_params in methods:
            if disp:
                print(f"  Method: {method_name}")

            results = compare_gd_methods_once(M, F, f_true, f_guess,
                C_known, C_error,
                                                x0, num_timesteps,
                                                observed_state_over_time,
                                                     num_iters, step_size
                                                    ,
                                                [[method_name,
                                                    method_func,
                                                    extra_params]], disp)

            # Include new losses
            for method_name, (_, method_losses, method_debug_vars) in
                results.items():
                for loss_name, loss_list in losses[method_name].items():
                    loss_list.append(method_losses[loss_name])

                for debug_name, debug_list in debug_vars[method_name].
                    items():
                    debug_list.append(method_debug_vars[debug_name])
```

```python

    # Average losses

    for method_name, method_losses in losses.items():
        for loss_name, loss_list in method_losses.items():
            losses[method_name][loss_name] = np.mean(loss_list, axis=0)

    for method_name, method_debug_vars in debug_vars.items():
        for debug_name, debug_list in method_debug_vars.items():
            debug_vars[method_name][debug_name] = np.mean(debug_list, axis=0)

    return losses, debug_vars

### Gradient Descent Visualization ###

def plot_losses(losses_many, num_obs_per_timestep, step_size, num_timesteps, num_iters, min_iter=None, max_iter=None):
    """
    Plots the losses for multiple gradient descent methods over iterations.

    Args:
    losses_many (dict): Dictionary of losses for each method.
                        Keys are method names, values are dictionaries containing losses.
    num_obs_per_timestep (int): Number of observations per timestep
    step_size (float): Step size used in gradient descent
    num_timesteps (int): Number of timesteps in the simulation
    num_iters (int): Number of iterations of gradient descent
    min_iter (int): Lowest plotted iter (default: None, plots from the beginning)
    max_iter (int): Highest plotted iter (default: None, plots until the end)

    Returns:
    None: This function displays the plot using matplotlib.pyplot.show()

    Notes:
    Creates a 2x2 grid of plots:
    1. Ocean misfit
    2. Atmosphere loss
    3. Control adjust Mahalanobis distance
    4. J' (combined loss for covariance constraint method, ocean loss for others)

    Each plot shows the evolution of the respective loss over iterations
```

```
        for all methods.
    """
    fig, axs = plt.subplots(2, 2, figsize=(10, 10))

    loss_funcs = ["$\sum_t (Ex(t)-y(t))^{T} (Ex(t)-y(t))$",
                  "$\sum_t (f_i(t)-f_{true}(t) )^{T} ( f_i(t)-f_{true}(t) )$",
                  "$a_i^T C^{-1} a_i$"]

    # Determine the range of iterations to plot
    min_iter = 0 if min_iter is None else max(0, min_iter)
    max_iter = num_iters if max_iter is None else min(num_iters, max_iter)
    plot_range = slice(min_iter, max_iter)

    for i, (loss_name, ax, func) in enumerate(zip(["ocean_misfit", "atmosphere_misfit", "mahalanobis(covariance similarity)"], axs.flatten(), loss_funcs)):
        for method_name, losses_dict in losses_many.items():
            ax.plot(range(min_iter, max_iter), losses_dict[loss_name][plot_range], label=method_name)
        ax.set_xlabel("Iteration $i$")
        ax.set_ylabel(loss_name+" loss:    "+func)
        ax.legend()
        ax.set_title(f"{loss_name}: "+func)

        # Set integer ticks on x-axis
        ax.xaxis.set_major_locator(plt.MaxNLocator(integer=True))

    # Fourth plot: ocean_misfit + mahalanobis if using covariance control adjust, just ocean otherwise
    ax = axs[1, 1]
    for method_name, losses_dict in losses_many.items():
        if method_name == r"Covariance Constraint J Gradient Descent":
            combined_loss = [o + c for o, c in zip(losses_dict["ocean_misfit"], losses_dict["mahalanobis(covariance similarity)"])]
            ax.plot(range(min_iter, max_iter), combined_loss[plot_range], label=method_name)
        else:
            ax.plot(range(min_iter, max_iter), losses_dict["ocean_misfit"][plot_range], label=method_name)

    ax.set_xlabel("Iteration $i$")
    ax.set_ylabel("J'")
    ax.legend()
    ax.set_title("J'")

    # Set integer ticks on x-axis for the fourth plot
    ax.xaxis.set_major_locator(plt.MaxNLocator(integer=True))

    fig.suptitle(f"Gradient Descent Variants: num_obs={
        num_obs_per_timestep}, step_size={step_size}, num_timesteps={
        num_timesteps}, num_iters={num_iters}")

    plt.tight_layout()
    plt.show()

def plot_debug(debug_vars, min_iter=None, max_iter=None, tickwidth=1,
    vlines = []):
    """
    Plot 9 chosen debug variables in a 3x3 grid for each method.

    Args:
    debug_vars (dict): Dictionary containing debug variables for each
        method
    min_iter (int, optional): Start index for plotting. If None, starts
        from the beginning.
    max_iter (int, optional): End index for plotting. If None, plots
        until the end.
    tickwidth (int): Width between ticks on x-axis
    vlines (list): List of vertical lines to add to the plot

    Returns:
    None: Displays the plot
    """
    # Determine the actual range of iterations
    all_iters = next(iter(debug_vars.values()))["Norm of s_i"]
    total_iters = len(all_iters)

    # Set min_iter and max_iter if they are None
    min_iter = 0 if min_iter is None else max(0, min_iter)
    max_iter = total_iters if max_iter is None else min(total_iters,
        max_iter)

    fig, axs = plt.subplots(3, 3, figsize=(15, 15))
    fig.suptitle(f"Debug Variables: Iterations {min_iter} to {max_iter}")

    # List of debug variables to plot
    plot_vars = [
        "Norm of s_i",
        "Expected Delta J w simple gd",
        "Expected Delta J w update rule",
        "Actual Delta J",
        "Norm of simple gd ui",
        "Norm of update rule ui",
        "Normalized dot product $a_i$ and $u_i$",
        "Normalized $-Cs_i \cdot a_i$",
        "Norm of $a_i$",
    ]

    for i, (var_name, ax) in enumerate(zip(plot_vars, axs.flatten())):
        for method_name, method_debug_vars in debug_vars.items():
            if var_name in method_debug_vars:
                plot_data = method_debug_vars[var_name][min_iter:max_iter
                    ]
                ax.plot(range(min_iter, max_iter), plot_data, ".-", label
                    =method_name)

        ax.set_xlabel("Iteration $i$")
        ax.set_ylabel(var_name)
        ax.legend()
        ax.set_title(var_name)
        ax.grid(True)

        # Set x-axis ticks to reflect the actual iteration numbers
        ticks = range(min_iter, max_iter, tickwidth)
        ax.set_xticks(ticks)
        ax.set_xticklabels(ticks)

        # Add vertical lines if specified
        for vline in vlines:
            if min_iter <= vline < max_iter:
                ax.axvline(x=vline, color='r', linestyle='--', alpha=0.5)

    plt.tight_layout()
    plt.show()
```