ALGORITHMIC DIFFERENTIATION (AD)
PART I

Ian Fenty
Shreyas Gaikwad

Goals for this talk

- Part I (Theory)
  - Reintroduce adjoints in a slightly different context
  - Hopefully show a different way of looking at adjoints
- Part 2 (Example)
  - Work through how Algorithmic Differentiation (AD) actually works in theory and practice
  - Apply these concepts to a simple 1D climate model

Utility of gradients

The adjoint operator can help us get gradients for our quantity of interest (QoI) with respect to any independent controls such as initial and boundary conditions, model parameters, etc.

- Sensitivity Analysis
- State Estimation
- Uncertainty Quantification
- Optimal Experimental Design (what is the most optimal location for new sensors for maximum new information gain)

MATHEMATICAL NOTATION

- **A: Matrix**
- **x: vector**
- *A and f: (potentially) non-linear functions*
- J and z: Scalar

Non-Linear Forward Model

This is the model we are working with. It could be an ocean model like the MITgcm or an ice sheet model like SICOPOLIS or ISSM.

In a general sense, the model can be expressed as a non-linear function.

$$\mathbf{y} = A(\mathbf{x})$$
$$J = f(\mathbf{y})$$

- $\mathbf{x}$ is a (uncertain) vector of “controls” (initial / boundary conditions, model parameters)
- A is a non-linear model (time-stepping ocean / ice-sheet model)
- $\mathbf{y}$ is the final model state
- J is a scalar **quantity of interest (QoI)**, could be some model-data misfit, could be something like projected sea level rise, or transport quantities like the AMOC. It is a function of the final state **y**.

How to get the gradients of J with respect to $\mathbf{x}$?

Method 1: Perturb one x at a time (Finite Differences)

$$\mathbf{y} = A(\mathbf{x})$$
$$J = f(\mathbf{y})$$

Let's perturb one component of **x** at a time and observe the resulting change in J.

Perturb the i-th component of **x**: $\mathbf{x}_i^+ = \mathbf{x} + \varepsilon \cdot \mathbf{e}_i$, where e_i is a unit vector in the i-th direction and ε is a small scalar. That will change the value of **y** to $\mathbf{y}_i^+$ and J to J_i^+.

$$\partial J/\partial x_i \approx [J_i^+ - J] / \varepsilon$$
$$\partial J/\partial x_i \approx [f(\mathbf{y}_i^+) - f(\mathbf{y})] / \varepsilon$$
$$\partial J/\partial x_i \approx [f(A(\mathbf{x}_i^+)) - f(A(\mathbf{x}))] / \varepsilon$$
$$\partial J/\partial x_i = [f(A(\mathbf{x}_i^+)) - f(A(\mathbf{x}))] / \varepsilon + O(\varepsilon)$$

Method 1: Finite Differences (Perturb one x at a time)

$$\mathbf{y} = A(\mathbf{x});\ J = f(\mathbf{y})$$

$$\mathbf{x}_i^+ = \mathbf{x} + \varepsilon \cdot \mathbf{e}_i$$

$$\partial J/\partial x_i \approx [f(A(\mathbf{x}_i^+)) - f(A(\mathbf{x}))] / \varepsilon + O(\varepsilon)$$

Drawbacks:

- Requires N+1 calls of non-linear forward model for N-dimensional gradient wrt **x.**
- Doesn't scale well (N ~ 0.5 billion "controls" for ECCO).
- Always has an error term proportional to some power of ε.
- Choice of ε is hard: too large and the response might not be linear, too small and it leads to numerical round-off errors.
- In practice, for different choices of ε, the values of the gradient can vary wildly.

Method 1: Finite Differences Example

- Finite Differences (Method 1) uses the non-linear forward model as a black box and just perturbs the input and uses non-linear functional evaluations to approximate the directional derivative.

$$y = x,\ J = y^2$$

$$\begin{aligned} dJ/dx &\approx ((x+\varepsilon)^2 - x^2)/\varepsilon \\ &= (x^2 + 2x\varepsilon + \varepsilon^2 - x^2)/\varepsilon \\ &= (2x\varepsilon + \varepsilon^2)/\varepsilon \\ &= 2x + \varepsilon \text{ (slightly off)} \end{aligned}$$

Method 2: Tangent Linear Model (Linearized Forward Model)

We again have: $\mathbf{y} = A(\mathbf{x})$ and $J = f(\mathbf{y})$

We compute the directional derivative of **y** with respect to **x** using the tangent linear model. Let $\delta\mathbf{x}$ be a small perturbation in **x**. The corresponding perturbation in **y** is given by:

$$\delta\mathbf{y} = \partial A/\partial\mathbf{x} \cdot \delta\mathbf{x}$$

$$\therefore\ \delta\mathbf{y} = \mathbf{A(x)} \cdot \delta\mathbf{x}$$

The matrix in purple is the Jacobian or the **Tangent Linear Model (TLM)**. Then propagate $\delta\mathbf{y}$ to get the change in J:

$$\delta J = \partial f/\partial\mathbf{y} \cdot \delta\mathbf{y}$$

$$\therefore\ \delta J = \partial f/\partial\mathbf{y} \cdot \mathbf{A(x)} \cdot \delta\mathbf{x}$$

This gives the **directional derivative of J along** $\delta\mathbf{x}$. To compute full gradient $\partial J/\partial\mathbf{x}$, repeat for each unit vector $\delta\mathbf{x} = \mathbf{e}_i$

Method 2: Tangent Linear Model

$$\mathbf{y} = A(\mathbf{x})$$
$$J = f(\mathbf{y})$$

HARD TO GET

$$\delta\mathbf{y} = \mathbf{A}(\mathbf{x}) \cdot \delta\mathbf{x}$$
$$\delta J = \partial f/\partial\mathbf{y} \cdot \delta\mathbf{y}$$

TRIVIAL

To compute full gradient $\partial J/\partial\mathbf{x}$, repeat for each unit vector $\delta\mathbf{x} = \mathbf{e}_i$

Characteristics:

- Precise.
- One call of TLM takes roughly twice as long as a non-linear forward model call.
- Requires N calls of the TLM for N-dimensional gradient wrt **x**.
- Doesn’t scale well (N ~ 0.5 billion “controls” for ECCO).

Method 2: Tangent Linear Model Example

- TLM (Method 2) employs the chain rule to linearize the non-linear forward model line-by-line. It then propagates small perturbations through the linearized model equations to get the precise directional derivative.

$y = x$, $J = y^2$

$\therefore \delta y = \delta x$ and $\delta J = 2y\delta y$

In 1D, only one unit vector $\delta x = 1$.

$\therefore \delta y = 1$ and $\delta J = 2y = 2x$

$\therefore 2x$ is our directional derivative (precise).

Reminder that for Method 1 our result was $2x + \varepsilon$ (slightly off).

Method 3: Adjoint Model (Transpose of TLM)

We again have: $\mathbf{y} = A(\mathbf{x})$ and $J = f(\mathbf{y})$. Let's employ the chain rule

$$\partial J/\partial x_i = \sum_j \partial J/\partial y_j \cdot \partial y_j/\partial x_i$$

$$\therefore \partial J/\partial x_i = \sum_j \partial y_j/\partial x_i \cdot \partial J/\partial y_j \text{ (both terms are scalars)}$$

$$\therefore \partial J/\partial x_i = \sum_j \partial(A(\mathbf{x}))_j/\partial x_i \cdot \partial J/\partial y_j$$

$$\therefore \nabla_{\mathbf{x}} J = (\partial A/\partial \mathbf{x})^T \cdot \nabla_{\mathbf{y}} J$$

$$\therefore \nabla_{\mathbf{x}} J = \mathbf{A(x)}^T \cdot \nabla_{\mathbf{y}} J$$

The matrix in purple is the transpose of the Tangent Linear Model (TLM), known as the **Adjoint Model**. You can **compute the entire gradient in one adjoint model pass**!

Method 3: Adjoint Model

$$\mathbf{y} = A(\mathbf{x});\ J = f(\mathbf{y})$$

$$\nabla_{\mathbf{x}} J = (\partial A / \partial \mathbf{x})^{T} \cdot \nabla_{\mathbf{y}} J = \mathbf{A}(\mathbf{x})^{T} \cdot \nabla_{\mathbf{y}} J$$

HARD TO GET   TRIVIAL

Characteristics:

- Precise.
- Gradient computed in one adjoint pass.
- For reasons we will soon discuss, it can be 5-100 times slower than the non-linear forward model (still better than running the non-linear forward model billions of times).

Why is Algorithmic Differentiation (AD) necessary?

- The chain rule has to be propagated across your entire code to get the derivatives (Remember $A(\mathbf{x})$ represents of your entire codebase).
- The MITgcm is hundred of thousands of lines of code.
- One change in the non-linear forward model could mean several changes in the TLM or adjoint depending on how the chain rule changes. It is thus error-prone and tedious to do this manually.
- Example:

```
# Compute a,b,c upstream
x=1
z= f(a,b,c)
y=x^2
```

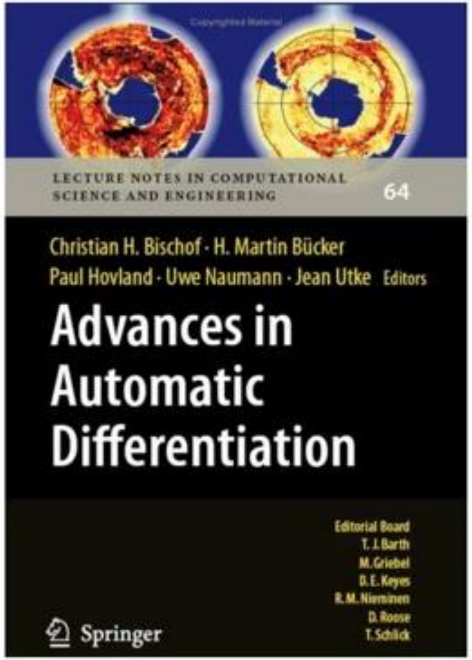
```
# Compute a,b,c upstream
x=1, dx = 1
z= f(a,b,c)
δy=2xδx
```

```
# Compute a,b,c upstream
x=1
z= f(a,b,c)
y=x^2 + z
```

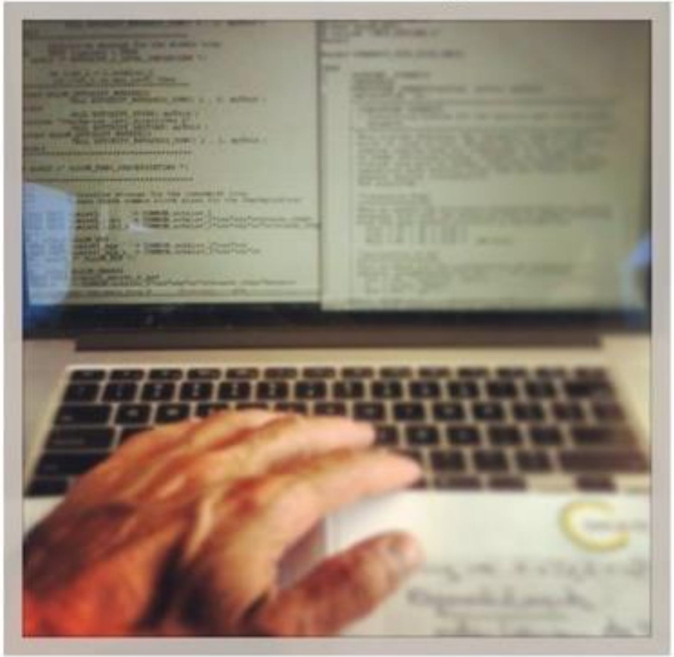
```
# Compute a,b,c upstream
# Compute δa,δb,δc upstream
x=1, δx = 1
z= f(a,b,c),
δz = f'(a,b,c,δa,δb,δc)
δy=2xδx + δz
```

Why is Algorithmic Differentiation (AD) necessary?

Generating and maintaining the adjoint of a state-of-the-art ocean GCM



hand-written adjoint



Automatic Differentiation

Giering & Kaminski (1998); Marotzke et al. (1999); Heimbach et al. (2005); Utke et al. (2007); Griewank & Walther (2008)

Some more nuances (looking inside A)

- To keep things readable, let's assume $\mathbf{x}$ is just initial conditions of an ocean model (no model parameters or boundary conditions). Let's assume N time steps.

$$\mathbf{y} = A(\mathbf{x}) = A_N(\ldots(A_2(A_1(\mathbf{x}))))$$

- For notational convenience,

$$\mathbf{x}_1 = A_1(\mathbf{x}),$$
$$\mathbf{x}_i = A_i(\mathbf{x}_{i-1}) \; i = 2, \ldots, N\text{-}1$$
$$\mathbf{y} = A_N(\mathbf{x}_{N-1})$$

- The Tangent Linear Model is given by:

$$\delta\mathbf{y} = \mathbf{A}(\mathbf{x}) \cdot \delta\mathbf{x} = \mathbf{A}_N(\mathbf{x}_{N-1})\ldots\mathbf{A}_2(\mathbf{x}_1)\cdot\mathbf{A}_1(\mathbf{x}) \cdot \delta\mathbf{x}$$

- The Adjoint Model is given by:

$$\nabla_{\mathbf{x}}\mathbf{J} = \mathbf{A}(\mathbf{x})^T \cdot \nabla_{\mathbf{y}}\mathbf{J} = \mathbf{A}_1(\mathbf{x})^T\cdot\mathbf{A}_2(\mathbf{x}_1)^T\ldots\mathbf{A}_N(\mathbf{x}_{N-1})^T \cdot \nabla_{\mathbf{y}}\mathbf{J}$$

Some more nuances (looking inside *A*)

- The Tangent Linear Model is given by:

ORDER OF COMPUTATION

$$\delta \mathbf{y} = \mathbf{A}(\mathbf{x}) \cdot \delta\mathbf{x} = \mathbf{A}_N(\mathbf{x}_{N-1})\ldots\mathbf{A}_2(\mathbf{x}_1)\cdot\mathbf{A}_1(\mathbf{x}) \cdot \delta\mathbf{x}$$

- The Adjoint Model is given by:

ORDER OF COMPUTATION

$$\nabla_{\mathbf{x}} J = \mathbf{A}(\mathbf{x})^T \cdot \nabla_{\mathbf{y}} J = \mathbf{A}_1(\mathbf{x})^T\cdot\mathbf{A}_2(\mathbf{x}_1)^T\ldots\mathbf{A}_N(\mathbf{x}_{N-1})^T \cdot \nabla_{\mathbf{y}} J$$

- The sequence of operations above is in our hands.
- If we go from left to right, we have to keep doing matrix-matrix operations, if both matrices are MxM and we have N matrices, that’s about NM^3 operations.
- If we go from right to left, that is always hit the vector on the right with a matrix first, we are only doing about NM^2 operations.
- N ~ 250,000 time steps and M ~ 500,000,000 parameters for ECCO
- **10^31 vs 10^23 floating point operations**, that’s a significant difference!

Some more nuances (looking inside *A*)

- The Tangent Linear Model is given by:

ORDER OF COMPUTATION

$$\delta \mathbf{y} = \mathbf{A}(\mathbf{x}) \cdot \delta \mathbf{x} = \mathbf{A}_N(\mathbf{x}_{N-1})\mathbf{....}\mathbf{A}_2(\mathbf{x}_1)\cdot\mathbf{A}_1(\mathbf{x}) \cdot \delta \mathbf{x}$$

- We need **x** first, followed by $\mathbf{x}_2$, $\mathbf{x}_3$, …, $\mathbf{x}_{N-1}$
- Also notice that the 1st timestep's adjoint matrix is hitting the vector first and then all the other timesteps' matrices hit it in the "expected" order.
- This is perfectly fine, since this is the natural order of computation in our non-linear forward model anyways.

Some more nuances (looking inside *A*)

- The Adjoint Model is given by:

ORDER OF COMPUTATION

$$\nabla_{\mathbf{x}} J = \mathbf{A}(\mathbf{x})^T \cdot \nabla_{\mathbf{y}} J = \mathbf{A}_1(\mathbf{x})^T \cdot \mathbf{A}_2(\mathbf{x}_1)^T \ldots \mathbf{A}_N(\mathbf{x}_{N-1})^T \cdot \nabla_{\mathbf{y}} J$$

- We need $\mathbf{x}_{N-1}$ first, followed by $\mathbf{x}_{N-2}$, $\mathbf{x}_{N-3}$, …,$\mathbf{x}_2$, $\mathbf{x}_2$, $\mathbf{x}$.
- This is reverse of the natural order of computation in our non-linear forward model or our general understanding of dependencies forward in time.
- Also notice that the N-th timestep's adjoint matrix is hitting the vector first and then all the other timesteps' matrices hit it in the reverse order.
- The adjoint model runs backwards in time!

Some more nuances (looking inside *A*)

- The Adjoint Model is given by:

ORDER OF COMPUTATION

$$\nabla_{\mathbf{x}} J = \mathbf{A}(\mathbf{x})^T \cdot \nabla_{\mathbf{y}} J = \mathbf{A}_1(\mathbf{x})^T \cdot \mathbf{A}_2(\mathbf{x}_1)^T \ldots \mathbf{A}_N(\mathbf{x}_{N-1})^T \cdot \nabla_{\mathbf{y}} J$$

- The adjoint model runs backwards in time!
- (Not shown here) The sign of the advection operator just reverses for the adjoint. If you were looking at an adjoint movie of the Gulf Stream it would be flowing from north to south.
- **INTUITION**: The Gulf Stream is sending “information” to the poles, and you as a detective are watching the movie in reverse, tracing the "influence trail" back from the poles to figure out which earlier states or regions mattered most.

Storage using a tape (stack)

- The Adjoint Model is given by:

$$\nabla_{\mathbf{x}} J = \mathbf{A(x)}^{\mathrm{T}} \cdot \nabla_{\mathbf{y}} J = \mathbf{A}_{1}(\mathbf{x})^{\mathrm{T}} \cdot \mathbf{A}_{2}(\mathbf{x}_{1})^{\mathrm{T}} \ldots \mathbf{A}_{N}(\mathbf{x}_{N-1})^{\mathrm{T}} \cdot \nabla_{\mathbf{y}} J$$

- We need the states in the reverse order when computing the adjoint.
- We have three options:
  - Run forward model, store all states (**i.e. all the x's**) in memory and retrieve in reverse order (Tapenade by default).
  - Recompute the state we need from scratch every time we take one step to the left (TAF by default).
  - Something hybrid (checkpointing, classic tradeoff between memory and computation time).

Why Tangent Linear Model?

- It would seem that the Tangent linear model is not useful when you have the adjoint model to compute the gradient in one go.
- However, it has its uses:
  - Validation of the adjoint model (ideally should agree to around machine precision!)
  - Second-order optimization methods (Hessian contains the tangent linear model)
  - Uncertainty quantification (Hessian is inverse of the posterior covariance matrix)

ALGORITHMIC DIFFERENTIATION (AD)
PART II

Ian Fenty
Shreyas Gaikwad

Budyko-Sellers Energy Balance Model

Let's now work with a small climate model!

Budyko-Sellers Model

- Observed temperature gradient between the equator and poles is a result of
    - Incoming solar insolation (tries to make equator much warmer than the poles)
    - Outgoing longwave radiation (local energy loss, damps temperature increases)
    - Heat transport by winds and oceans (tries to flatten the temperature gradient by flowing from equator to poles)
        - COOL the tropics
        - WARM the poles.
- We are looking to model the equator-to-pole temperature difference.

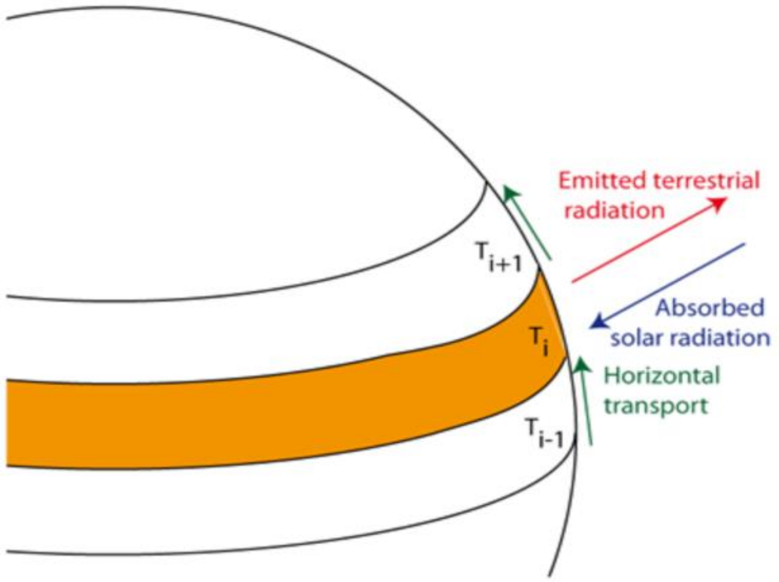


Figure 3.3: Representation of a one-dimensional EBM for which the temperature T_i is averaged over a band of longitude.

http://www.climate.be/textbook/chapter3_node6.xml

Budyko-Sellers Model local energy budget for each latitude

$$\frac{\partial E(\phi)}{\partial t} = \text{ASR}(\phi) - \text{OLR}(\phi) - \frac{1}{2\pi a^2 \cos\phi} \frac{\partial \mathcal{H}}{\partial \phi}$$

- ASR: Absorbed solar radiation
- OLR: Outgoing longwave radiation
- H is the meridional heat transport

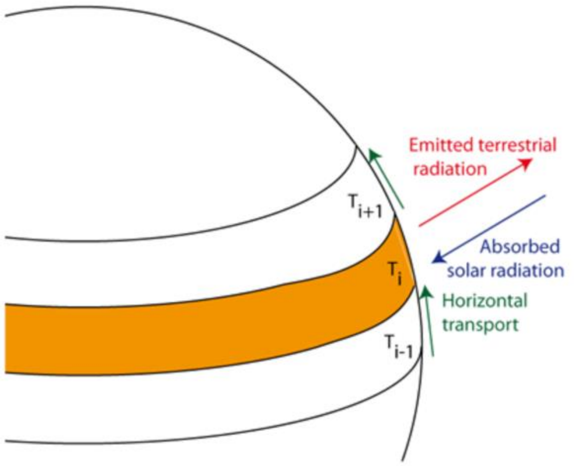


Figure 3.3: Representation of a one-dimensional EBM for which the temperature T_i is averaged over a band of longitude.

http://www.climate.be/textbook/chapter3_node6.xml

Modeling the heat transport

Let's now formally introduce a parameterization that **approximates the heat transport as a down-gradient diffusion process**:

$$\mathcal{H}(\phi) \approx -2\pi a^2 \cos\phi \, D \, \frac{\partial T_s}{\partial \phi}$$

With D a parameter for the **diffusivity** or **thermal conductivity** of the climate system, a number in W m^{-2} ºC^{-1}.

The value of D will be chosen to match observations – i.e. tuned.

Budyko-Sellers Model local energy budget for each latitude

$$\frac{\partial E(\phi)}{\partial t} = \mathrm{ASR}(\phi) - \mathrm{OLR}(\phi) + \boxed{\frac{D}{\cos\phi}\frac{\partial}{\partial\phi}\left(\cos\phi\,\frac{\partial T_s}{\partial\phi}\right)}$$

- ASR: Absorbed solar radiation
- OLR: Outgoing longwave radiation
- The third term is diffusion in spherical coordinates.

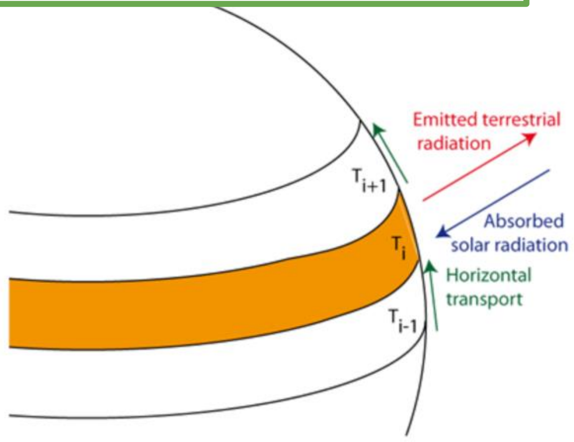


Figure 3.3: Representation of a one-dimensional EBM for which the temperature T_i is averaged over a band of longitude.

http://www.climate.be/textbook/chapter3_node6.xml

Modeling the energy content

- Most of the heat is in the oceans.
- Surface temperature is a good proxy for the heat content of the ocean column

$$E(\phi) = C(\phi)\, T_s(\phi)$$

- C is the effective heat capacity of the ocean column. Function of latitude depending on fraction of ocean vs land can vary.
- Ts is the surface temperature.

Budyko-Sellers Model local energy budget for each latitude

$$\boxed{C(\phi)\frac{\partial T_s}{\partial t}} = \mathrm{ASR}(\phi) - \mathrm{OLR}(\phi) + \boxed{\frac{D}{\cos\phi}\frac{\partial}{\partial\phi}\left(\cos\phi\,\frac{\partial T_s}{\partial\phi}\right)}$$

- ASR: Absorbed solar radiation
- OLR: Outgoing longwave radiation
- The third term is heat diffusion in spherical coordinates.

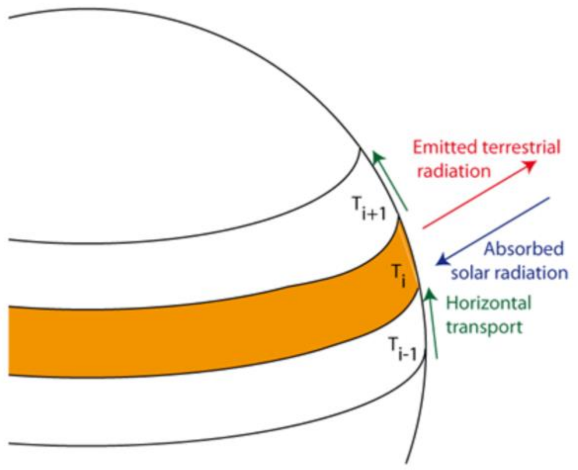


Figure 3.3: Representation of a one-dimensional EBM for which the temperature T_i is averaged over a band of longitude.

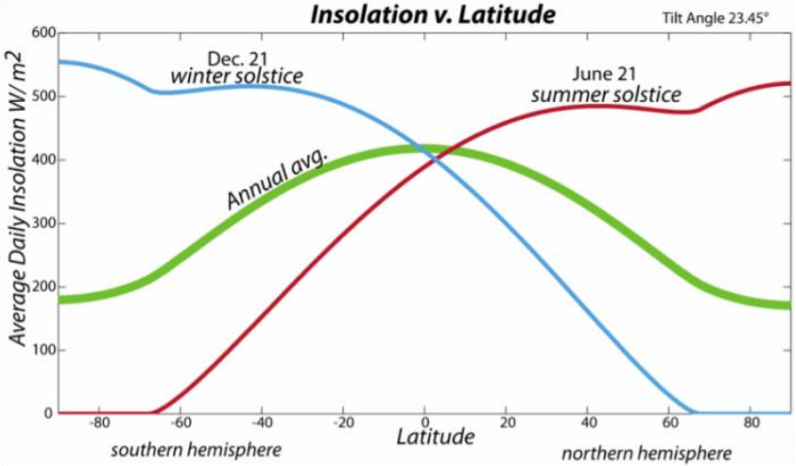
http://www.climate.be/textbook/chapter3_node6.xml

Modeling the Radiation terms

- Albedo depends linearly on the temperature
- Incoming heat Q depends on the latitude (sine of the latitude actually)
- Stefan-Boltzmann law for outgoing longwave radiation

$$ASR = [1 - \alpha(T_s)]\, Q(\phi)$$

$$OLR = \epsilon \sigma T_s^4$$



Budyko-Sellers Model local energy budget for each latitude

$$\boxed{C(\phi)\frac{\partial T_s}{\partial t}} = \boxed{[1-\alpha(T_s)]\,Q(\phi)} - \boxed{\epsilon\sigma T_s^4} + \boxed{\frac{D}{\cos(\phi)}\frac{\partial}{\partial\phi}\left(\cos(\phi)\frac{\partial T_s}{\partial\phi}\right)}$$

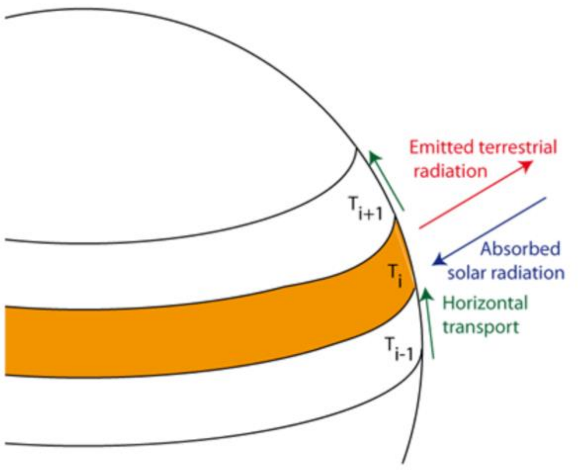


Figure 3.3: Representation of a one-dimensional EBM for which the temperature T_i is averaged over a band of longitude.

http://www.climate.be/textbook/chapter3_node6.xml

Let’s look at a pseudo-code

Initialize surface temperature to be a constant 290K (17C)

T(i) = 290 for all i in the latitude grid

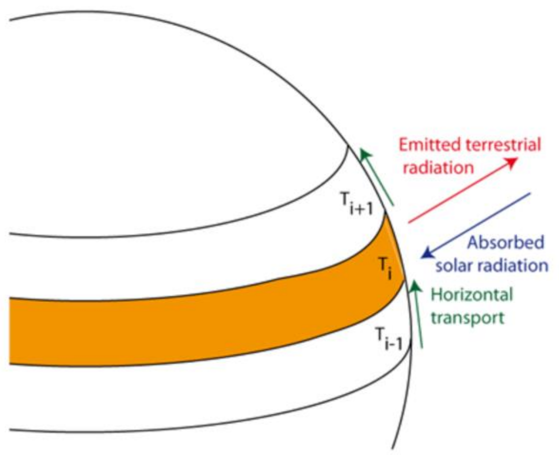


Figure 3.3: Representation of a one-dimensional EBM for which the temperature T_i is averaged over a band of longitude.

http://www.climate.be/textbook/chapter3_node6.xml

Initialize surface temperature to be a constant 290K (17C)

T(i) = 290 for all i in the latitude grid

for t = 1..N

Incoming radiative flux [time-invariant, constant albedo assumed]

Fin(i) = sx(i)*(1.d0-alpha_const(i)) + **xxs (control)**

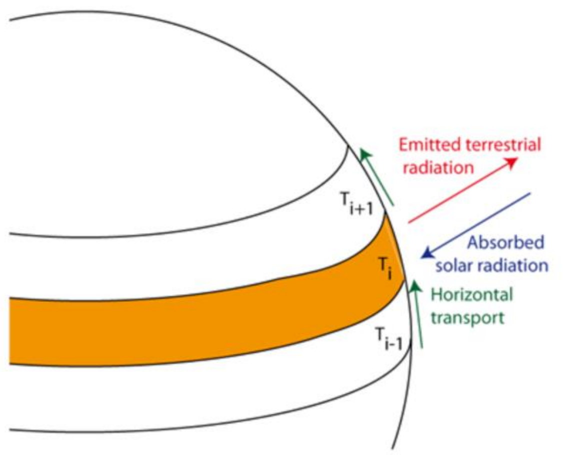


Figure 3.3: Representation of a one-dimensional EBM for which the temperature T_i is averaged over a band of longitude.

http://www.climate.be/textbook/chapter3_node6.xml

Initialize surface temperature to be a constant 290K (17C)

T(i) = 290 for all i in the latitude grid

for t = 1..N

Incoming radiative flux [time-invariant, constant albedo assumed]

Fin(i) = sx(i)*(1.d0-alpha_const(i)) + **xxs (control)**

Outgoing radiation

Fout(i) = epsilon*sigma* T(i)**4

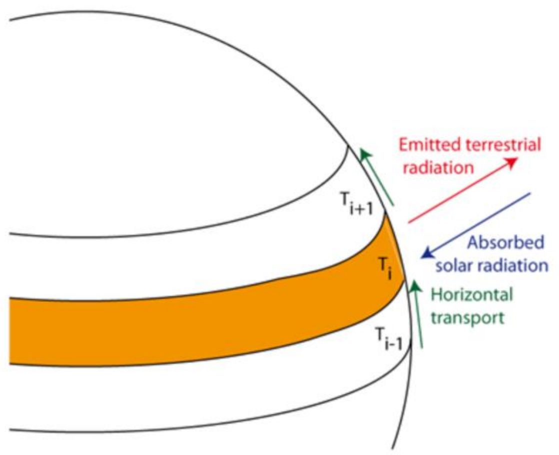


Figure 3.3: Representation of a one-dimensional EBM for which the temperature T_i is averaged over a band of longitude.

http://www.climate.be/textbook/chapter3_node6.xml

Initialize surface temperature to be a constant 290K (17C)

T(i) = 290 for all i in the latitude grid

for t = 1..N

Incoming radiative flux [time-invariant, constant albedo assumed]

Fin(i) = sx(i)*(1.d0-alpha_const(i)) + **xxs (control)**

Outgoing radiation

Fout(i) = epsilon*sigma* T(i)**4

Meridional flux (simplified for brevity, central difference scheme)

Fdiff(i) = D*[[T(i+1) - 2T(i) +T(i-1)]] / dx**2

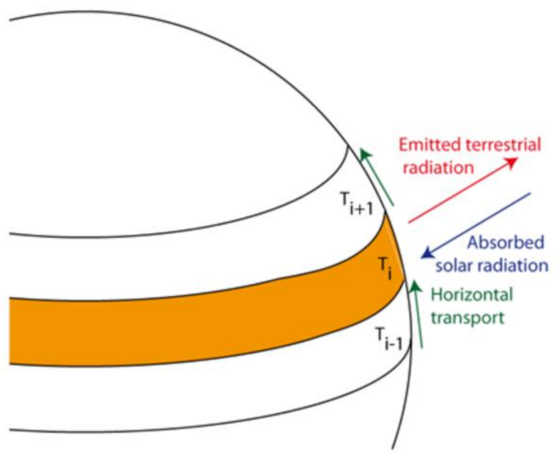


Figure 3.3: Representation of a one-dimensional EBM for which the temperature T_i is averaged over a band of longitude.

http://www.climate.be/textbook/chapter3_node6.xml

Initialize surface temperature to be a constant 290K (17C)

T(i) = 290 for all i in the latitude grid

for t = 1..N

Incoming radiative flux [time-invariant, constant albedo assumed]

Fin(i) = sx(i)*(1.d0-alpha_const(i)) + **xxs (control)**

Outgoing radiation

Fout(i) = epsilon*sigma* T(i)**4

Meridional flux (simplified for brevity, central difference scheme)

Fdiff(i) = D*[[T(i+1) - 2T(i) +T(i-1)]] / dx**2

Update T (Assume C = 1)

Tnew(i) = T(i)+ dt*[Fin(i)-Fout(i)+Fdiff(i)]

end

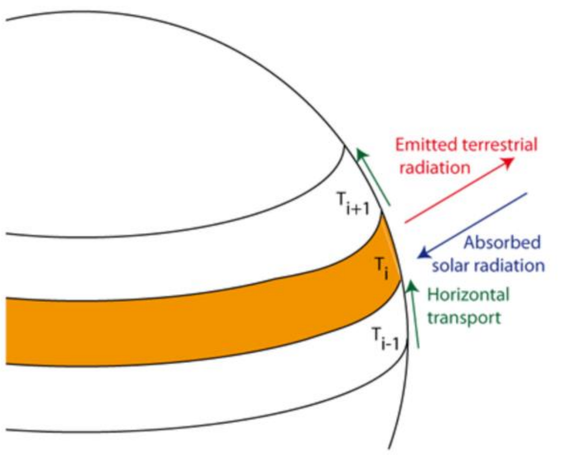


Figure 3.3: Representation of a one-dimensional EBM for which the temperature T_i is averaged over a band of longitude.

http://www.climate.be/textbook/chapter3_node6.xml

Initialize surface temperature to be a constant 290K (17C)

T(i) = 290 for all i in the latitude grid

for t = 1..N

Incoming radiative flux [time-invariant, constant albedo assumed]

Fin(i) = sx(i)*(1.d0-alpha_const(i)) + **xxs (control)**

Outgoing radiation

Fout(i) = epsilon*sigma* T(i)**4

Meridional flux (simplified for brevity, central difference scheme)

Fdiff(i) = D*[[T(i+1) - 2T(i) +T(i-1)]] / dx**2

Update T (Assume C = 1)

Tnew(i) = T(i)+ dt*[Fin(i)-Fout(i)+Fdiff(i)]

end

J = f(Tnew) = Tnew(equator) - Tnew(poles)

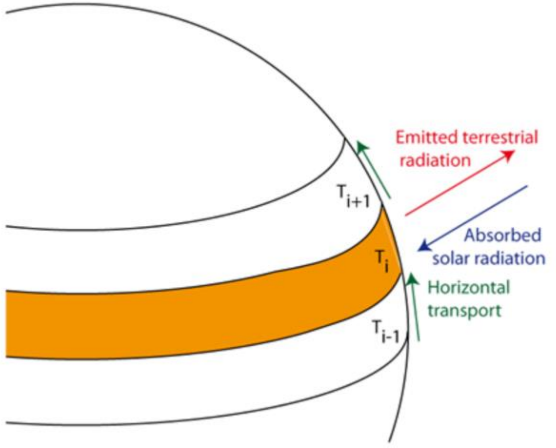


Figure 3.3: Representation of a one-dimensional EBM for which the temperature T_i is averaged over a band of longitude.

http://www.climate.be/textbook/chapter3_node6.xml

Before we go into the real code

Let’s differentiate a line of code by hand

Initialize surface temperature to be a constant 290K (17C)

T(i) = 290 for all i in the latitude grid

for t = 1..N

Incoming radiative flux [time-invariant, constant albedo assumed]

Fin(i) = sx(i)*(1.d0-alpha_const(i)) + **xxs (control)**

Outgoing radiation

Fout(i) = epsilon*sigma* T(i)**4

Meridional flux (simplified for brevity, central difference scheme)

Fdiff(i) = D*[[T(i+1) - 2T(i) +T(i-1)]] / dx**2

Update T (Assume C = 1)

Tnew(i) = T(i)+ dt*[Fin(i)-Fout(i)+Fdiff(i)]

end

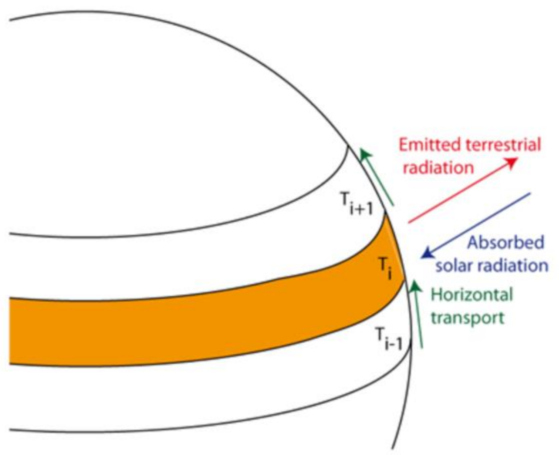


Figure 3.3: Representation of a one-dimensional EBM for which the temperature T_i is averaged over a band of longitude.

http://www.climate.be/textbook/chapter3_node6.xml

J = f(Tnew) = Tnew(equator) - Tnew(poles)

Outgoing radiation

Fout(i) = epsilon*sigma* T(i)**4

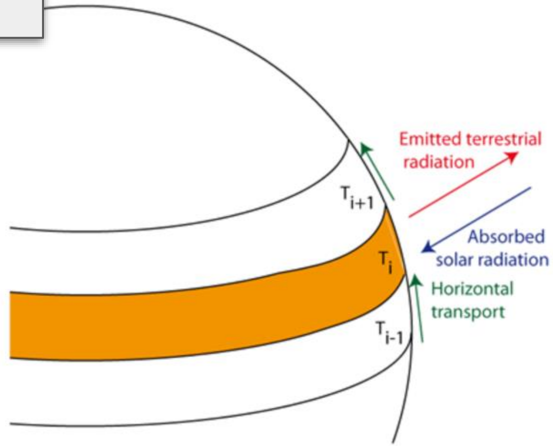


Figure 3.3: Representation of a one-dimensional EBM for which the temperature T_i is averaged over a band of longitude.

http://www.climate.be/textbook/chapter3_node6.xml

REMINDER ON WHAT TLM DOES

- TLM (Method 2) employs the chain rule to linearize the non-linear forward model line-by-line. It then propagates small perturbations through the linearized model equations to get the precise directional derivative.

$y = x$, $J = y^2$

$\therefore \delta y = \delta x$ and $\delta J = 2y\delta y$

In 1D, only one unit vector $\delta x = 1$.

$\therefore \delta y = 1$ and $\delta J = 2y = 2x$

$\therefore$ $2x$ is our directional derivative (precise).

AD tool's modification of source code to generate TLM

Source Code (T(i) coming from upstream)

Fout(i) = epsilon*sigma* **T(i)**4**

AD generated TLM Code (T(i), T_tl(i) coming from upstream)

Fout(i) = epsilon*sigma* **T(i)**4** (OG non-linear forward code)

Fout_tl(i) = 4*epsilon*sigma* **T(i)**3*T_tl(i)** (Propagating pertubations)

Observations:

- For many lines of non-linear forward code, you have 2 lines in the TLM code each.
- Some lines may not be differentiated because there's nothing "active" in them.
- The TLM code has slightly less than 2x the lines of the non-linear forward code.
- It is thus slightly less than 2x times slower.

NOTE: var_tl means δvar in terms of the math.

AD tool's modification of source code to generate TLM

AD generated TLM Code (T(i), T_tl(i) coming from upstream)

Fout(i) = epsilon*sigma* **T(i)**4** (OG non-linear forward code)

Fout_tl(i) = 4*epsilon*sigma* **T(i)**3***T_tl(i)** (Propagating pertubations)

Matrix form of TLM

$$\begin{bmatrix} \text{Fout\_tl(i)} \\ \text{T\_tl(i)} \end{bmatrix}_{new} = \begin{bmatrix} 0 & 4*\text{epsilon}*\text{sigma}*T_{old}(i)**3 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \text{Fout\_tl(i)} \\ \text{T\_tl(i)} \end{bmatrix}_{old}$$

HELPFUL TIP: Think of every single line of forward code as its own mini-model.

NOTE: var_tl means δvar in terms of the math.

REMINDER ON WHAT THE ADJOINT MODEL IS

$$\mathbf{y} = A(\mathbf{x});\ J = f(\mathbf{y})$$

$$\nabla_{\mathbf{x}} J = (\partial A / \partial \mathbf{x})^T \cdot \nabla_{\mathbf{y}} J = \mathbf{A}(\mathbf{x})^T \cdot \nabla_{\mathbf{y}} J$$

HARD TO GET

TRIVIAL

Characteristics:

- Precise.
- Gradient computed in one adjoint pass.
- Can be 5-100 times slower than the non-linear forward model (still better than running the non-linear forward model millions of times).

AD tool's modification of source code to generate Adjoint

Source Code

Fout(i) = epsilon*sigma* **T(i)**4**

Matrix form of Adjoint (Transpose of TLM)

$$\begin{vmatrix} \textbf{Fout\_ad(i)} \\ \textbf{T\_ad(i)} \end{vmatrix}_{new} = \begin{vmatrix} 0 & 0 \\ 4*epsilon*sigma*\textbf{T}_{\textbf{old}}\textbf{(i)**3} & 1 \end{vmatrix} \begin{vmatrix} \textbf{Fout\_ad(i)} \\ \textbf{T\_ad(i)} \end{vmatrix}_{old}$$

Transpose of TLM matrix i.e. A^T

NOTE:

- You should always read var_ad in your mind as ∂J/∂var (and not as something proportional or equivalent to var itself).
- Always write down the tangent linear model first and then take the transpose to get the adjoint.

AD tool's modification of source code to generate Adjoint

LET'S LOOK A BIT MORE CLOSELY

$$\begin{vmatrix} \textbf{Fout\_ad(i)} \\ \textbf{T\_ad(i)} \end{vmatrix}_{new} = \begin{vmatrix} 0 & 0 \\ 4*epsilon*sigma*\mathbf{T_{old}(i)**3} & 1 \end{vmatrix} \begin{vmatrix} \textbf{Fout\_ad(i)} \\ \textbf{T\_ad(i)} \end{vmatrix}_{old}$$

- We know that the adjoint runs reverse in time, so if the do loop in the forward code ran from 1 to N, the adjoint runs from N to 1.
- This means we actually don't have $\mathbf{T_{old}(i)}$!
- We need to store it when we are running the forward code then and then retrieve it for use here (Tapenade). Or we have to compute it somehow (TAF).

AD tool's modification of source code to generate Adjoint

$$\begin{vmatrix} \textbf{Fout\_ad(i)} \\ \textbf{T\_ad(i)} \end{vmatrix}_{new} = \begin{vmatrix} 0 & 0 \\ 4*epsilon*sigma*\mathbf{T_{old}(i)**3} & 1 \end{vmatrix} \begin{vmatrix} \textbf{Fout\_ad(i)} \\ \textbf{T\_ad(i)} \end{vmatrix}_{old}$$

Source Code

```
Do t = 1, T
   Do i = 1, N
       … BEFORE …
       Fout(i) = epsilon*sigma* T(i)**4
       … AFTER (T(i) updated)…
   End do
End do
```

AD generated Adjoint Code

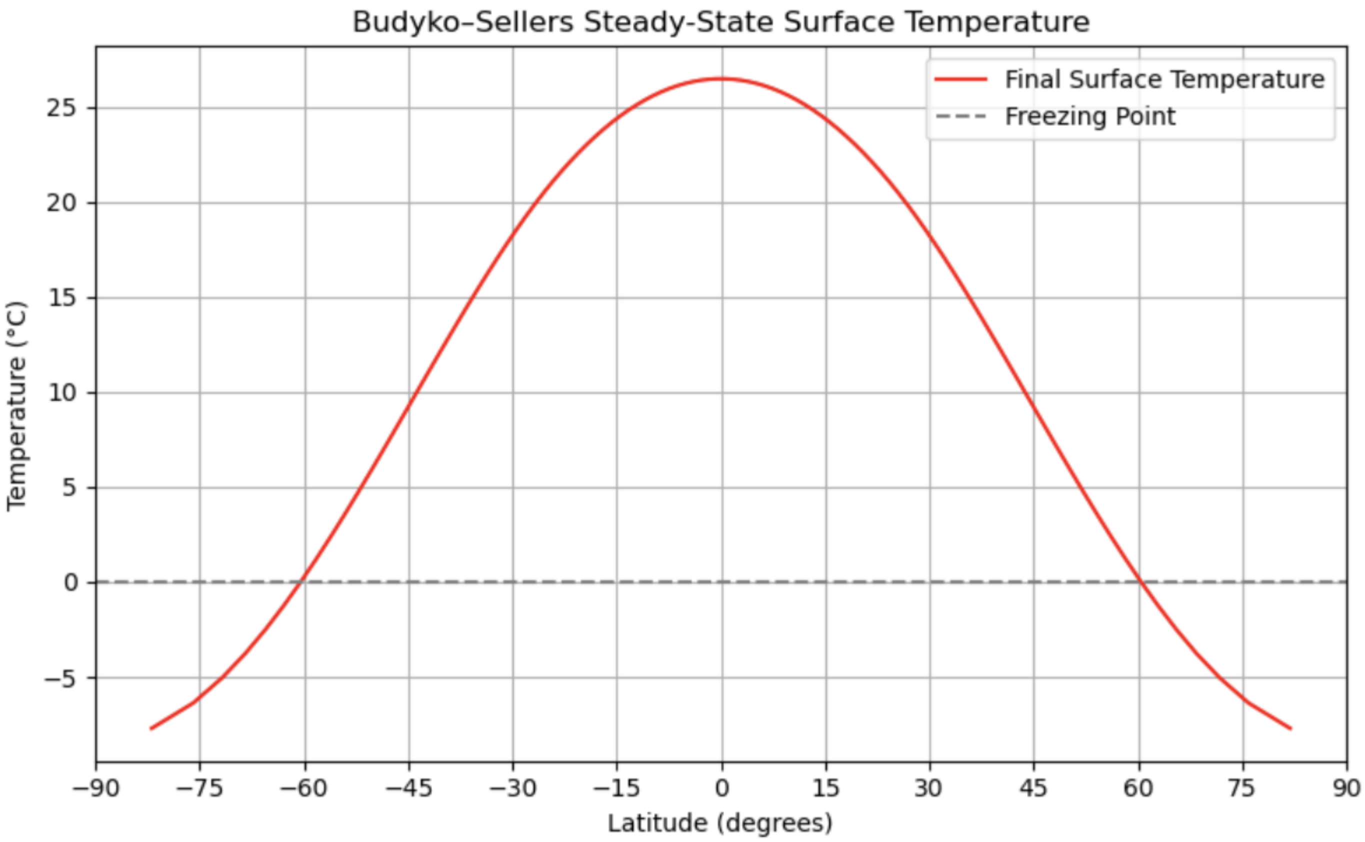
```
Do t = 1, T
  Do i = 1, N
       … BEFORE …
       Fout(i) = epsilon*sigma* T(i)**4
       STORE(T(i)) # save T_old
       … AFTER  (T updated)…
  End do
End do
Do t = T, 1, -1
  Do i = N, 1, -1
       … AFTER_AD (T_new is available but we need T_old) …
            RETRIEVE(T(i)) # get T_old
            T_ad(i) = 1*T_ad(i) + 4*epsilon*sigma*T(i)**3*Fout_ad(ad)
            Fout_ad(i) = 0*T_ad(i)+0*Fout_ad(ad) = 0
       … BEFORE_AD …
  End do
End do
```

The sequence matters!!

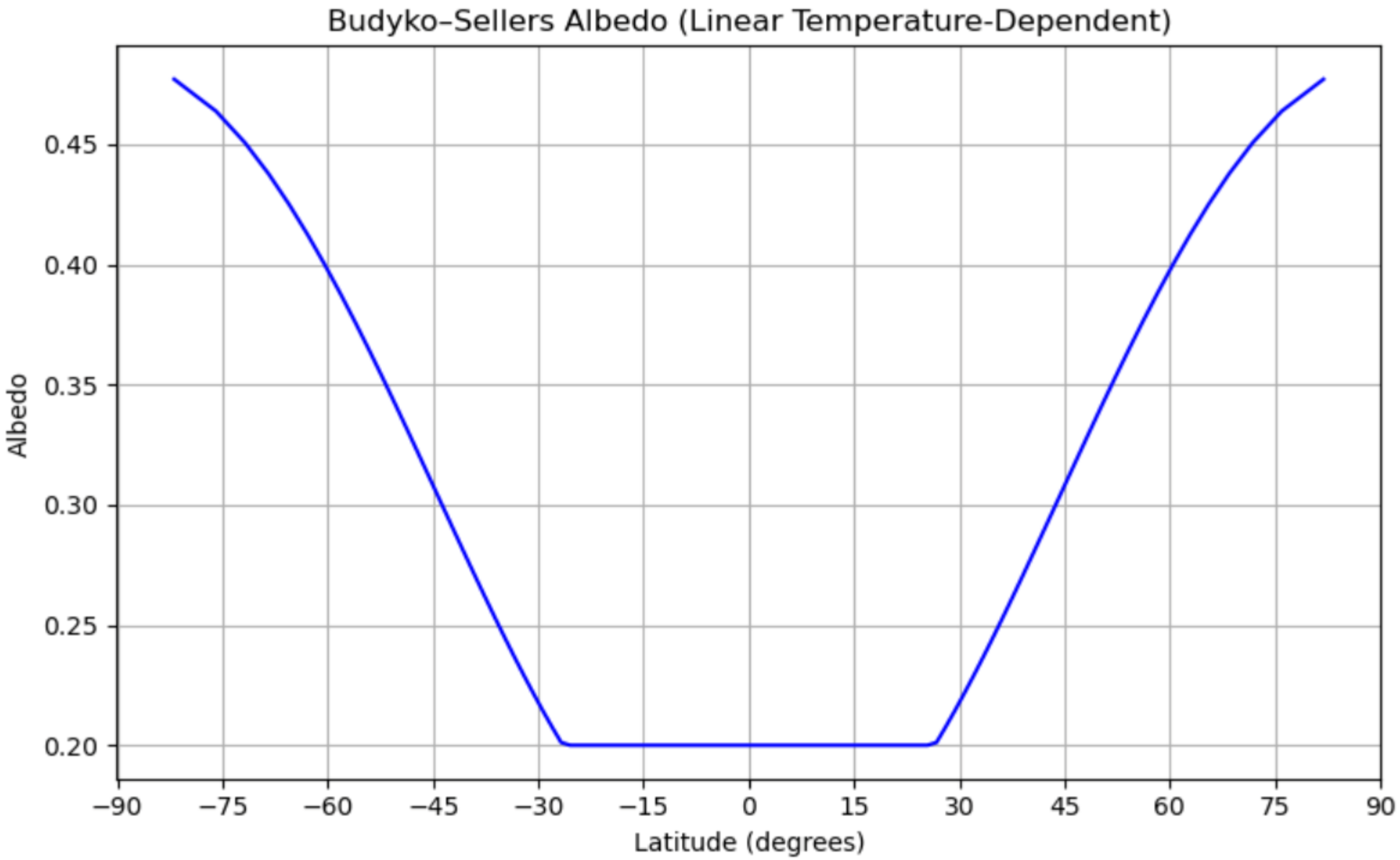
Differentiating Budyko-Sellers model in Fortran-77

Using TAF and Tapenade

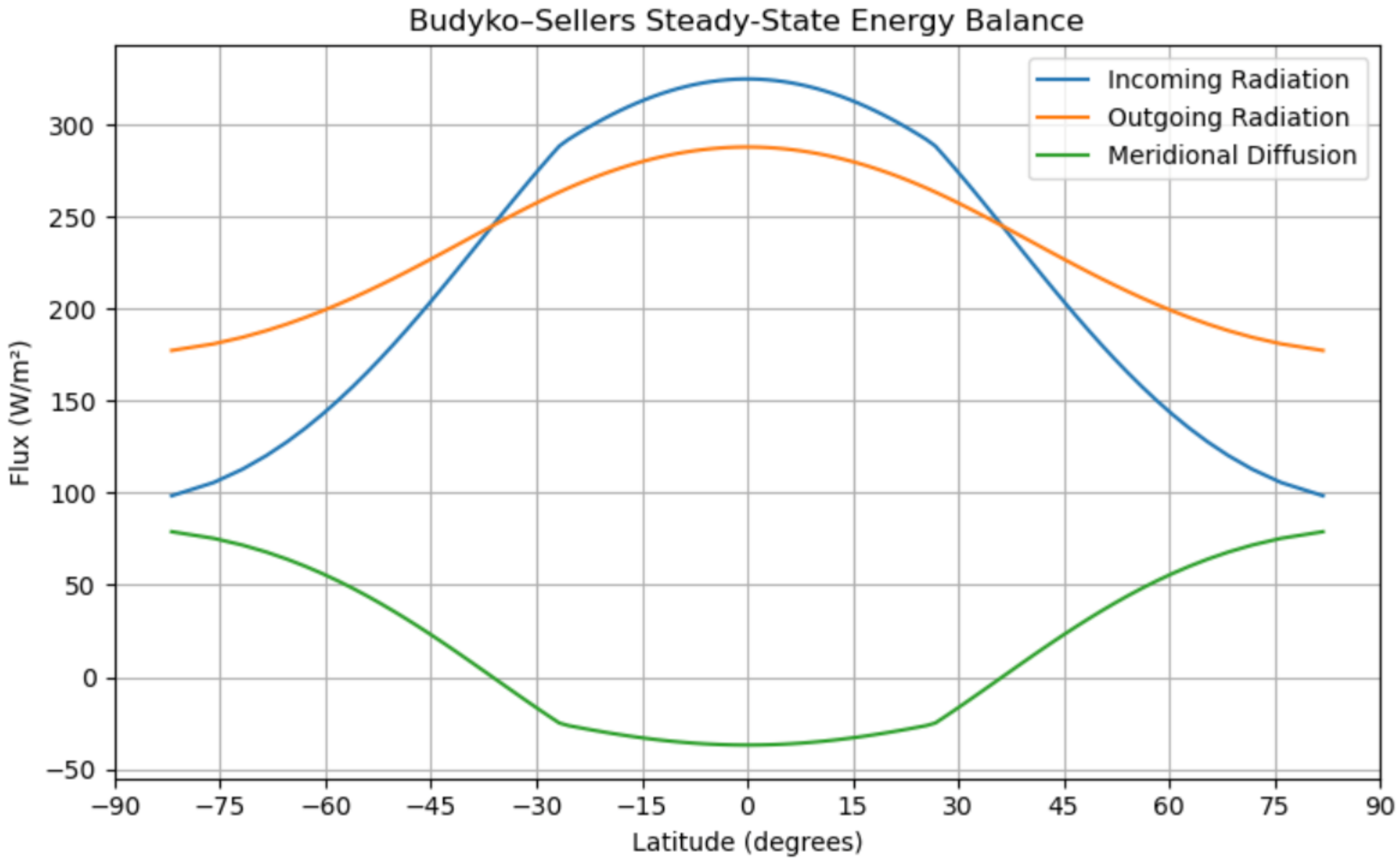
Forward model results (E2P delta_T = 34 C)



Forward model results (E2P delta_T = 34 C)



Forward model results (E2P delta_T = 34 C)



Task 1: Validate our gradients

Using TAF and Tapenade

Finite differences

budyko_sellers takes in vector xxs (control) and computes scalar J (QoI)

$$\partial J/\partial x_i \approx [f(A(x_i^+)) - f(A(x))] / \varepsilon + O(\varepsilon)$$

```
C      open a file to save gradients (dJ/dx)
       open(unit=111,file='dJdX_from_finite_differences.txt')
       J = 0.0d0

       DO I = 1, N
C          perturb one element of xxs
           XXS(I) = EPS

           call budyko_sellers( XXS, J )

C          reset perturbation to zero
           XXS(I) = 0.
           print *, 'values of gradient for I = ', I, ': ',
     &               (J-J_ORIG)/EPS
           write(unit=111,fmt='(F24.17,A)') (J-J_ORIG)/EPS
       END DO

       close(unit=111)
```

Tangent linear code

- budyko_sellers takes in vector xxs (control) and computes scalar J (QoI)
- budyko_sellers_tl takes in vector xxs (control), xxs_tl and computes scalar J (QoI), J_tl

Let's look at the line we differentiated by hand earlier, in the generated code budyko_sellers_tl

```
fout_tl(i) = epsilon*sigma*4*t(i)**3*t_tl(i)
fout(i) = epsilon*sigma*t(i)**4
```

Fout(i) = epsilon*sigma* **T(i)**4** (OG non-linear forward code)

Fout_tl(i) = 4*epsilon*sigma* **T(i)**3***T_tl(i)** (Propagating pertubations)

Tangent linear code

- budyko_sellers takes in vector xxs (control) and computes scalar J (QoI)
- budyko_sellers_tl takes in vector xxs (control), xxs_tl and computes scalar J (QoI), J_tl

Let’s look at the do loop to compute the gradient

```
C     open a file to save gradients (dJ/dx)
      open(unit=111,file='dJdX_from_tangent_linear.txt')

      J = 0.0d0

      DO I = 1, N
C         perturb one element of xxs
          XXS_TL(I) = 1.

          call budyko_sellers_tl( XXS, XXS_TL, J, J_TL )

C         reset perturbation to zero
          XXS_TL(I) = 0.

          print *, 'values of gradient for I = ', I, ': ', J_TL
          write(unit=111,fmt='(F24.17,A)') J_TL
      END DO

      close(unit=111)
```

Tapenade adjoint code

```
    DO iter=1,max_iter
      DO i=1,n
```

Do t = 1, T
Do i = 1, N

Tapenade adjoint code

```
DO iter=1,max_iter
  DO i=1,n
fout(i) = epsilon*sigma*t(i)**4
```

Do t = 1, T

Do i = 1, N

… BEFORE …

Fout(i) = epsilon*sigma* **T(i)**4**

Tapenade adjoint code

```
DO iter=1,max_iter
  DO i=1,n
fout(i) = epsilon*sigma*t(i)**4
CALL PUSHREAL8(t(i))
    ENDDO
ENDDO
```

Do t = 1, T

Do i = 1, N

… BEFORE …

Fout(i) = epsilon*sigma* **T(i)**4**

STORE(T(i)) # save T_old

… AFTER (T updated)…

End do

End do

Tapenade adjoint code

```
      DO iter=1,max_iter
        DO i=1,n
fout(i) = epsilon*sigma*t(i)**4
CALL PUSHREAL8(t(i))
        ENDDO
      ENDDO
DO iter=max_iter,1,-1
  DO i=n,1,-1
```

Do t = 1, T
Do i = 1, N
… BEFORE …
Fout(i) = epsilon*sigma* **T(i)**4**
STORE(T(i)) # save T_old
… AFTER (T updated)…
End do
End do
Do t = T, 1, -1
Do i = N, 1, -1

Tapenade adjoint code

```
DO iter=1,max_iter
  DO i=1,n
fout(i) = epsilon*sigma*t(i)**4
CALL PUSHREAL8(t(i))
  ENDDO
ENDDO
DO iter=max_iter,1,-1
  DO i=n,1,-1
CALL POPREAL8(t(i))
```

```
Do t = 1, T
  Do i = 1, N
      … BEFORE …
      Fout(i) = epsilon*sigma* T(i)**4
      STORE(T(i)) # save T_old
      … AFTER  (T updated)…
  End do
End do
Do t = T, 1, -1
  Do i = N, 1, -1
      … AFTER_AD (T_new is available but we need T_old) …
          RETRIEVE(T(i)) # get T_old
```

Tapenade adjoint code

```
DO iter=1,max_iter
  DO i=1,n
fout(i) = epsilon*sigma*t(i)**4
CALL PUSHREAL8(t(i))
  ENDDO
ENDDO
DO iter=max_iter,1,-1
  DO i=n,1,-1
CALL POPREAL8(t(i))
t_ad(i) = t_ad(i) + 4*t(i)**3*epsilon*sigma*fout_ad(i)
fout_ad(i) = 0.D0
  ENDDO
ENDDO
```

Do t = 1, T
  Do i = 1, N
… BEFORE …
Fout(i) = epsilon*sigma* **T(i)**4**
STORE(T(i)) # save T_old
… AFTER (T updated)…
  End do
End do
Do t = T, 1, -1
  Do i = N, 1, -1
… AFTER_AD (T_new is available but we need T_old) …
RETRIEVE(T(i)) # get T_old
T_ad(i) = 1***T_ad(i) +**
4*epsilon*sigma***T(i)**3*****Fout_ad(ad)**
Fout_ad(i) = 0*T_ad(i)+0***Fout_ad(ad) = 0**
… BEFORE_AD …
  End do
End do

Tapenade adjoint code

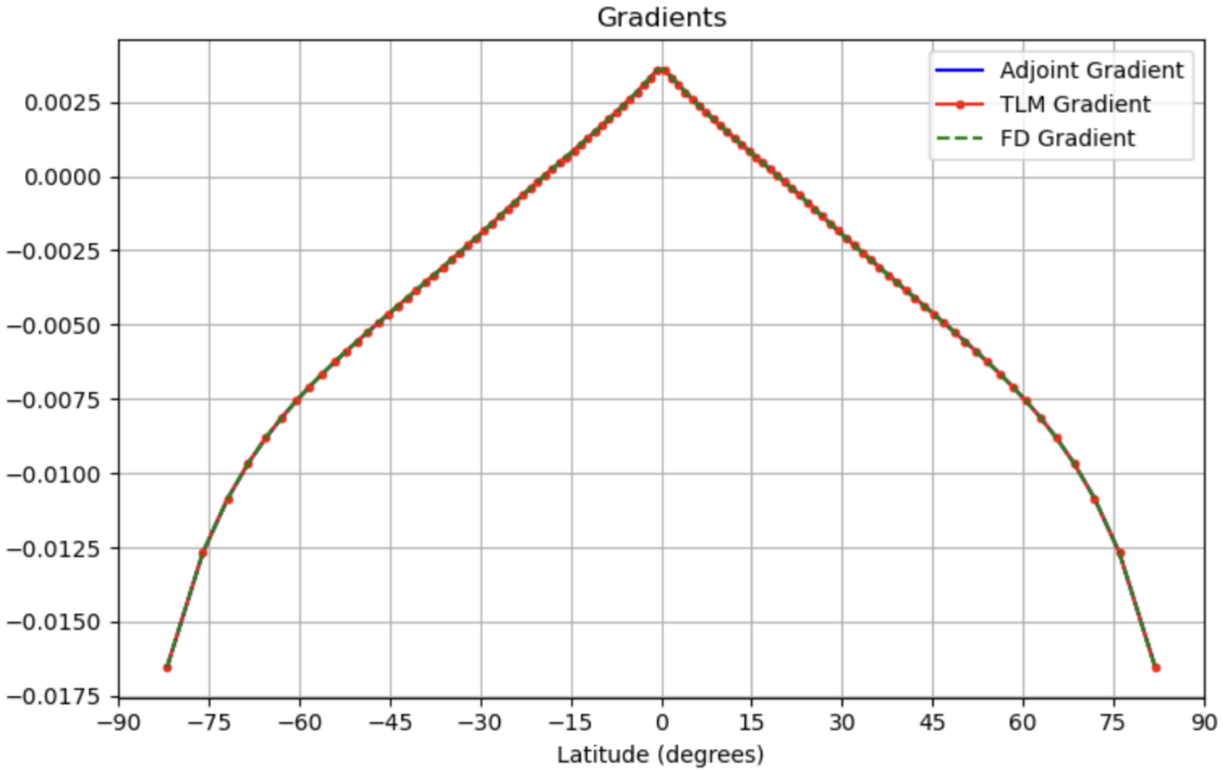
- budyko_sellers takes in vector xxs (control) and computes scalar J (QoI)
- budyko_sellers_ad takes in vector xxs (control), J_ad and computes scalar J (QoI), xxs_ad
- budyko_sellers_ad computes the gradient in one pass, no do loop needed.

```
C     initialize with J_AD = 1
      J_AD = 1.
      call budyko_sellers_ad( XXS, XXS_AD, J, J_AD )

      DO I = 1, N
          print *, 'gradient of J w.r.t. XXS ', I, XXS_AD(i)
      END DO
```

- You should always read var_ad in your mind as ∂J/∂var (and not as something proportional or equivalent to var itself).
- J_ad = 1 is therefore just ∂J/∂J = 1. Needed to spinup the adjoint.

Gradients


Gradients
Adjoint Gradient
TLM Gradient
FD Gradient
0.0025
0.0000
−0.0025
−0.0050
−0.0075
−0.0100
−0.0125
−0.0150
−0.0175
−90
−75
−60
−45
−30
−15
0
15
30
45
60
75
90
Latitude (degrees)

FD vs TLM gradients

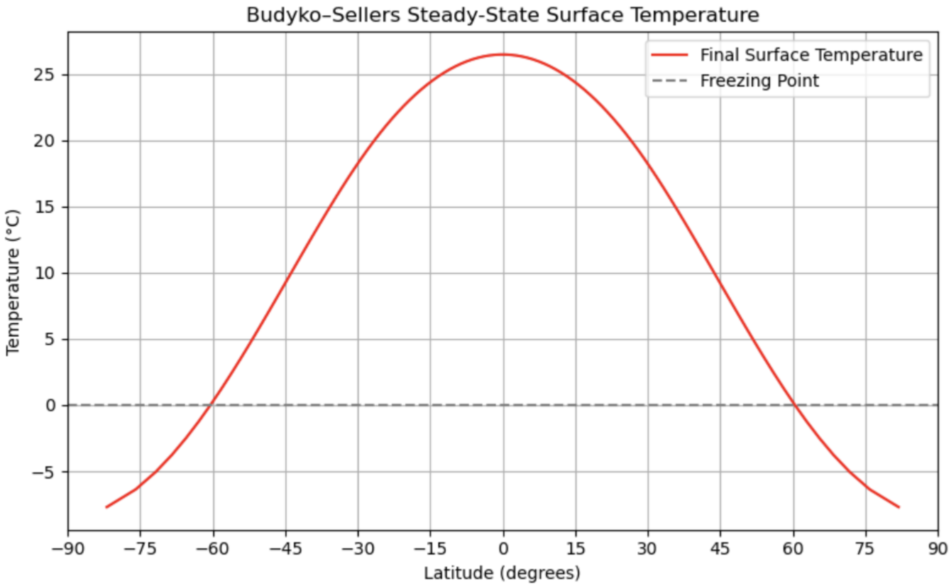
-0.01653916508417628	-0.01653915597657857
-0.01267075637159216	-0.01267075135869183
-0.01084356648216779	-0.01084356336210099
-0.00966999629781104	-0.00966999402492560
-0.00880994893114733	-0.00880994712615732
-0.00813005409592375	-0.00813005279620890
-0.00756494163618162	-0.00756494069765643
-0.00707800206133326	-0.00707800122798549
-0.00664681922705905	-0.00664681864135141
-0.00625676203594308	-0.00625676183507197
-0.00589781217694750	-0.00589781184493842
-0.00556284893217323	-0.00556284867184190
-0.00524666192040816	-0.00524666186681259
-0.00494534781795133	-0.00494534767571454
-0.00465592513078169	-0.00465592528633375
-0.00437608340140598	-0.00437608347310064
-0.00410400767638958	-0.00410400808184414
-0.00383826135207586	-0.00383826140119467
-0.00357769556618080	-0.00357769584727457
-0.00332139064907432	-0.00332139093827167
-0.00306860593104386	-0.00306860644248948
-0.00281874643205245	-0.00281874698792770
-0.00257133438318727	-0.00257133494258996
-0.00232598887671662	-0.00232598936094236
-0.00208240915412558	-0.00208240944556558
-0.00184036056579226	-0.00184036141488459
-0.00159966786346462	-0.00159966797194391
-0.00136019934094743	-0.00136019978189429
-0.00111507268162899	-0.00111507279026799
-0.00087638791613382	-0.00087638828094106
-0.00064489384278301	-0.00064489468136643

Adjoint vs TLM gradients

-0.01653915597657863	-0.01653915597657857
-0.01267075135869208	-0.01267075135869183
-0.01084356336210131	-0.01084356336210099
-0.00966999402492580	-0.00966999402492560
-0.00880994712615741	-0.00880994712615732
-0.00813005279620923	-0.00813005279620890
-0.00756494069765668	-0.00756494069765643
-0.00707800122798573	-0.00707800122798549
-0.00664681864135159	-0.00664681864135141
-0.00625676183507226	-0.00625676183507197
-0.00589781184493867	-0.00589781184493842
-0.00556284867184213	-0.00556284867184190
-0.00524666186681274	-0.00524666186681259
-0.00494534767571469	-0.00494534767571454
-0.00465592528633384	-0.00465592528633375
-0.00437608347310064	-0.00437608347310064
-0.00410400808184417	-0.00410400808184414
-0.00383826140119473	-0.00383826140119467
-0.00357769584727464	-0.00357769584727457
-0.00332139093827176	-0.00332139093827167
-0.00306860644248954	-0.00306860644248948
-0.00281874698792772	-0.00281874698792770
-0.00257133494259004	-0.00257133494258996
-0.00232598936094237	-0.00232598936094236
-0.00208240944556556	-0.00208240944556558
-0.00184036141488467	-0.00184036141488459
-0.00159966797194393	-0.00159966797194391
-0.00136019978189429	-0.00136019978189429
-0.00111507279026801	-0.00111507279026799
-0.00087638828094107	-0.00087638828094106
-0.00064489468136644	-0.00064489468136643
-0.00041946030152067	-0.00041946030152066
-0.00019904064044680	-0.00019904064044680
0.00001733535649476	0.00001733535649475
0.00023057714647199	0.00023057714647199
0.00044154289729336	0.00044154289729335

State Estimate

- For D=0.6, the equator-to-pole temperature difference is 34.16 deg Celcius.
- We know that the real value of the equator-to-pole temperature difference is 45 deg Celcius.



Basics of State Estimation

- Drive a model-data misfit to zero using optimization methods.
- In this case, our cost function is (m: Model, d: Data)

$$J = (\Delta T_m - \Delta T_d)^2$$

- Our independent variable is the diffusion coefficient D.
- We use steepest descent to drive this cost function down to 0.

```
      DIFF = 0.6D0 + XXS
```

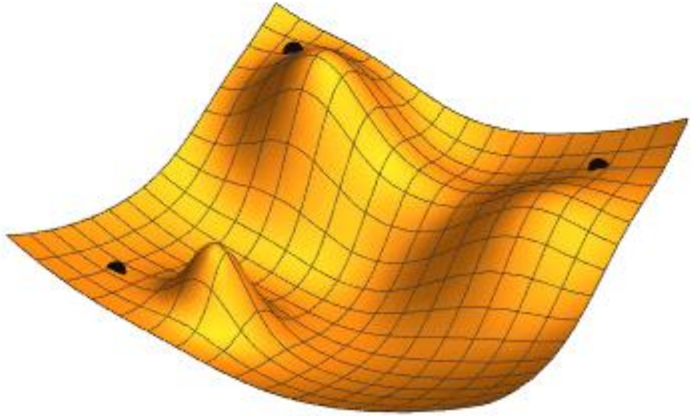
```
      DO ITER_GD = 1, MAX_ITER_GD
C       initialize with J_AD = 1
        J_AD = 1.
        XXS_AD = 0.0D0
        call budyko_sellers_ad( XXS, XXS_AD, J, J_AD )

        XXS = XXS - ETA*XXS_AD

        call budyko_sellers(XXS, J)

      END DO
```

Steepest descent



```
T_pole:     265.450    | T_equator:     299.613    | ICE_LINE IN DEGREES IN NORTHERN HEMISPHERE:
GradDes I:       0     | Cost J:     58.717483     | XXS:    0.000000
```

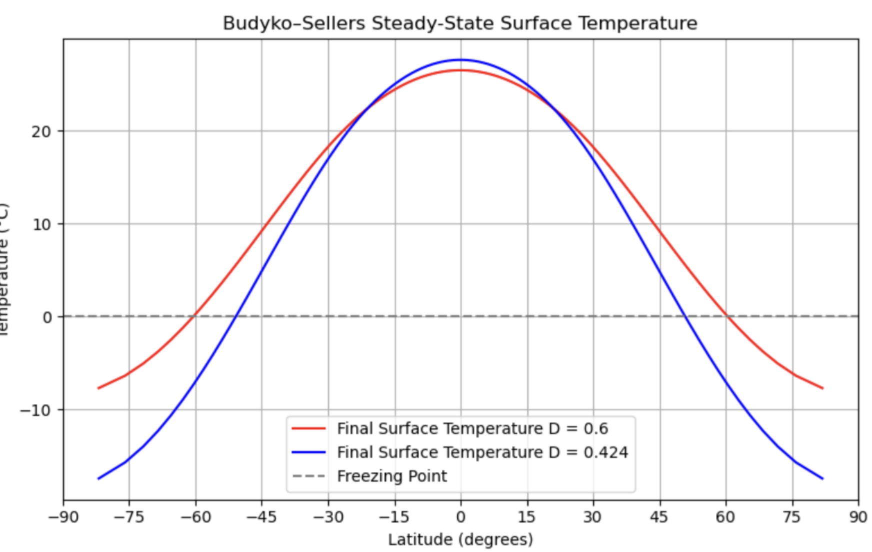
```
T_pole:     261.136    | T_equator:     300.157    | ICE_LINE IN DEGREES IN NORTHERN HEMISPHERE:
GradDes I:       2     | Cost J:     17.872432     | XXS:   -0.093027
```

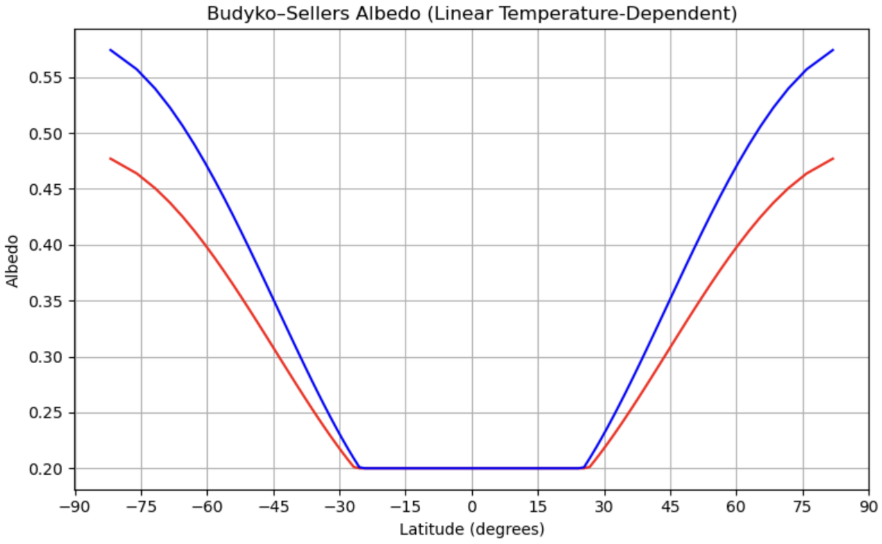
```
T_pole:     256.338    | T_equator:     300.672    | ICE_LINE IN DEGREES IN NORTHERN HEMISPHERE:
GradDes I:       5     | Cost J:      0.221896     | XXS:   -0.168211
```

```
T_pole:     255.730    | T_equator:     300.729    | ICE_LINE IN DEGREES IN NORTHERN HEMISPHERE:
GradDes I:      10     | Cost J:      0.000001     | XXS:   -0.176125
```

```
T_pole:     255.729    | T_equator:     300.729    | ICE_LINE IN DEGREES IN NORTHERN HEMISPHERE:
GradDes I:      11     | Cost J:      0.000000     | XXS:   -0.176135
```

Tuned results





Thank you!

A glimpse into utility of TLM and adjoint for UQ

A glimpse into Uncertainty Quantification

$$\mathcal{J} = \frac{1}{2}(\mathbf{y} - \mathbf{d})^T \Gamma_{\mathrm{data}}^{-1} (\mathbf{y} - \mathbf{d}) = \frac{1}{2}(A(\mathbf{x}) - \mathbf{d})^T \Gamma_{\mathrm{data}}^{-1} (A(\mathbf{x}) - \mathbf{d})$$

A glimpse into Uncertainty Quantification

$$\mathcal{J} = \frac{1}{2}(\mathbf{y} - \mathbf{d})^T \Gamma_{\text{data}}^{-1} (\mathbf{y} - \mathbf{d}) = \frac{1}{2}(A(\mathbf{x}) - \mathbf{d})^T \Gamma_{\text{data}}^{-1} (A(\mathbf{x}) - \mathbf{d})$$

$$\mathbf{g}(\mathbf{x}) = \frac{\partial \mathcal{J}}{\partial \mathbf{x}} = \mathbf{A}(\mathbf{x})^T \Gamma_{\text{data}}^{-1} (A(\mathbf{x}) - \mathbf{d})$$

A glimpse into Uncertainty Quantification

$$\mathcal{J} = \frac{1}{2}(\mathbf{y} - \mathbf{d})^T \Gamma_{\text{data}}^{-1} (\mathbf{y} - \mathbf{d}) = \frac{1}{2}(A(\mathbf{x}) - \mathbf{d})^T \Gamma_{\text{data}}^{-1} (A(\mathbf{x}) - \mathbf{d})$$

$$\mathbf{g}(\mathbf{x}) = \frac{\partial \mathcal{J}}{\partial \mathbf{x}} = \mathbf{A}(\mathbf{x})^T \Gamma_{\text{data}}^{-1} (A(\mathbf{x}) - \mathbf{d})$$

$$\mathbf{H}(\mathbf{x}) = \frac{\partial^2 \mathcal{J}}{\partial \mathbf{x}^2} = \mathbf{A}(\mathbf{x})^T \Gamma_{\text{data}}^{-1} \mathbf{A}(\mathbf{x})$$

A glimpse into Uncertainty Quantification

$$\mathcal{J} = \frac{1}{2}(\mathbf{y}-\mathbf{d})^T\Gamma_{\text{data}}^{-1}(\mathbf{y}-\mathbf{d}) = \frac{1}{2}(A(\mathbf{x})-\mathbf{d})^T\Gamma_{\text{data}}^{-1}(A(\mathbf{x})-\mathbf{d})$$

$$\mathbf{g}(\mathbf{x}) = \frac{\partial \mathcal{J}}{\partial \mathbf{x}} = \mathbf{A}(\mathbf{x})^T\Gamma_{\text{data}}^{-1}(A(\mathbf{x})-\mathbf{d})$$

$$\mathbf{H}(\mathbf{x}) = \frac{\partial^2 \mathcal{J}}{\partial \mathbf{x}^2} = \mathbf{A}(\mathbf{x})^T\Gamma_{\text{data}}^{-1}\mathbf{A}(\mathbf{x}) + \text{SCARY 3rd order tensor}$$

A glimpse into Uncertainty Quantification

$$\mathcal{J} = \frac{1}{2}(\mathbf{y}-\mathbf{d})^T\Gamma_{\text{data}}^{-1}(\mathbf{y}-\mathbf{d}) = \frac{1}{2}(A(\mathbf{x})-\mathbf{d})^T\Gamma_{\text{data}}^{-1}(A(\mathbf{x})-\mathbf{d})$$

$$\mathbf{g}(\mathbf{x}) = \frac{\partial \mathcal{J}}{\partial \mathbf{x}} = \mathbf{A}(\mathbf{x})^T\Gamma_{\text{data}}^{-1}(A(\mathbf{x})-\mathbf{d})$$

$$\mathbf{H}(\mathbf{x}) = \frac{\partial^2 \mathcal{J}}{\partial \mathbf{x}^2} = \mathbf{A}(\mathbf{x})^T\Gamma_{\text{data}}^{-1}\mathbf{A}(\mathbf{x}) + \text{SCARY 3rd order tensor} \times (A(\mathbf{x})-\mathbf{d})$$

A glimpse into Uncertainty Quantification

$$\mathcal{J} = \frac{1}{2}(\mathbf{y}-\mathbf{d})^T \Gamma_{\text{data}}^{-1}(\mathbf{y}-\mathbf{d}) = \frac{1}{2}(A(\mathbf{x})-\mathbf{d})^T \Gamma_{\text{data}}^{-1}(A(\mathbf{x})-\mathbf{d})$$

$$\mathbf{g}(\mathbf{x}) = \frac{\partial \mathcal{J}}{\partial \mathbf{x}} = \mathbf{A}(\mathbf{x})^T \Gamma_{\text{data}}^{-1}(A(\mathbf{x})-\mathbf{d})$$

$$\mathbf{H}(\mathbf{x}) = \frac{\partial^2 \mathcal{J}}{\partial \mathbf{x}^2} = \mathbf{A}(\mathbf{x})^T \Gamma_{\text{data}}^{-1}\mathbf{A}(\mathbf{x}) + \text{SCARY 3rd order tensor} \times (A(\mathbf{x})-\mathbf{d})$$

Adjoint

TLM

A glimpse into Uncertainty Quantification

$$\mathcal{J} = \frac{1}{2}(\mathbf{y} - \mathbf{d})^T \Gamma_{\text{data}}^{-1} (\mathbf{y} - \mathbf{d}) = \frac{1}{2}(A(\mathbf{x}) - \mathbf{d})^T \Gamma_{\text{data}}^{-1} (A(\mathbf{x}) - \mathbf{d})$$

$$\mathbf{g}(\mathbf{x}) = \frac{\partial \mathcal{J}}{\partial \mathbf{x}} = \mathbf{A}(\mathbf{x})^T \Gamma_{\text{data}}^{-1} (A(\mathbf{x}) - \mathbf{d})$$

$$\mathbf{H}(\mathbf{x}) = \frac{\partial^2 \mathcal{J}}{\partial \mathbf{x}^2} = \mathbf{A}(\mathbf{x})^T \Gamma_{\text{data}}^{-1} \mathbf{A}(\mathbf{x}) + \text{SCARY 3rd order tensor} \times \cancelto{0}{(A(\mathbf{x}) - \mathbf{d})}$$

Adjoint

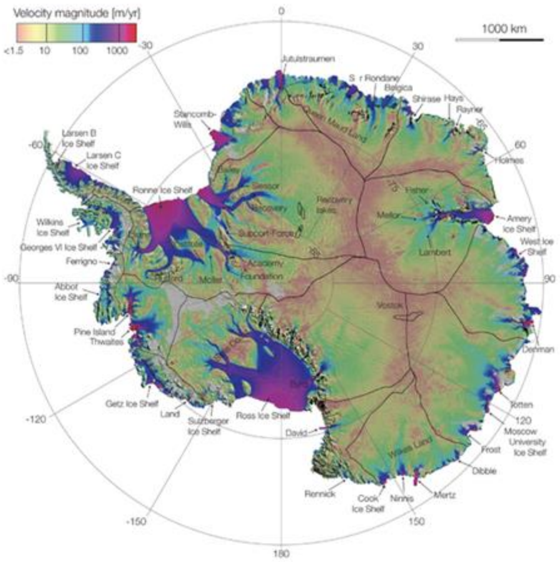
TLM

Gauss-Newton approximation

Example: Antarctic Ice Sheet basal sliding coefficient (Isaac et. al (2015))

Objective

For a steady-state Stokes model for ice sheet flow, can the basal sliding coefficient below the Antarctic ice sheet be constrained using InSAR surface velocity observations?



Observed surface velocity (Rignot et al., 2011)

Figure: Surface velocity data from Rignot et al. 2011.

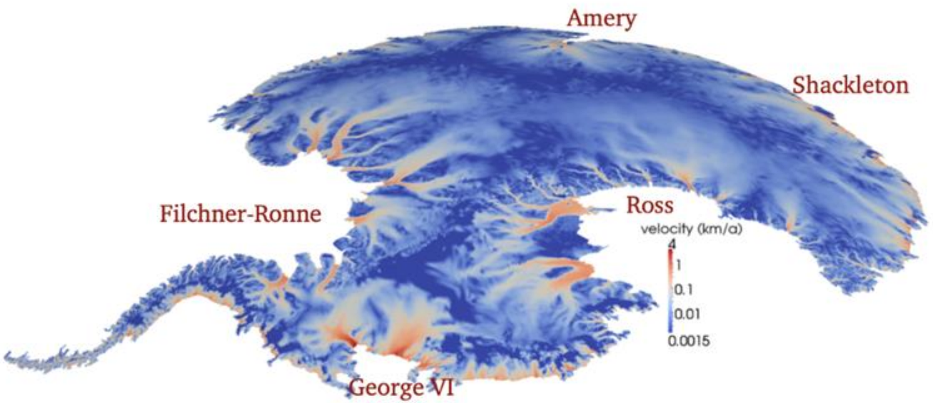


Figure: InSAR surface velocity data.

Example: Antarctic Ice Sheet basal sliding coefficient (Isaac et. al (2015))

Deterministic solution

Optimal inferred basal sliding coefficient field that reduces model-data misfit.

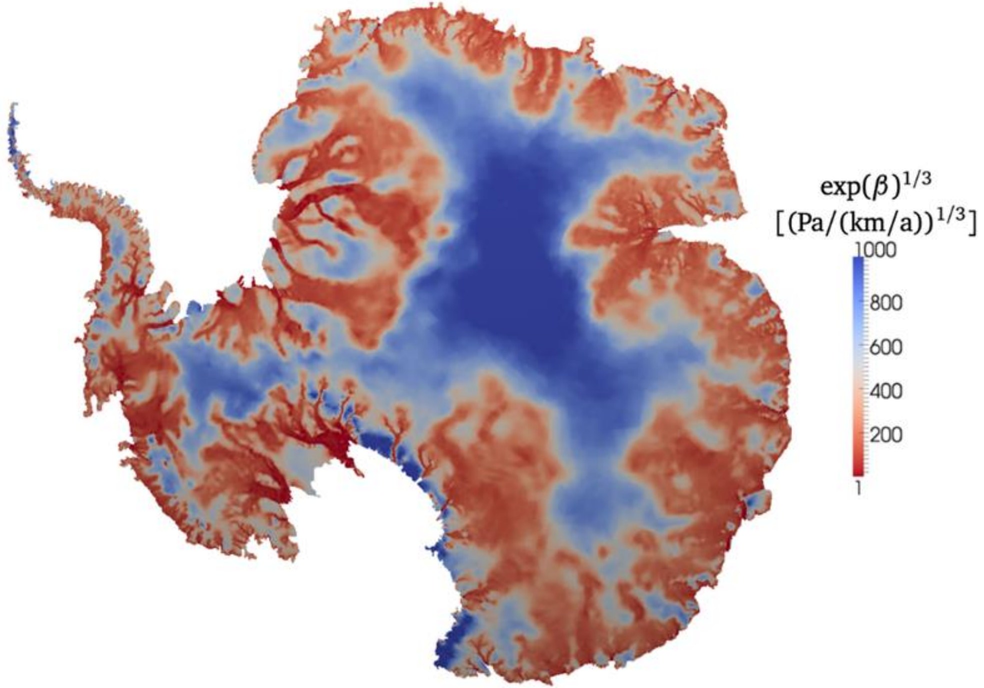


Figure: Inferred basal sliding coefficient

Example: Antarctic Ice Sheet basal sliding coefficient (Isaac et. al (2015))

Uncertainty Quantification (Bayesian probabilistic perspective)

Confidence in our inferred parameter field.

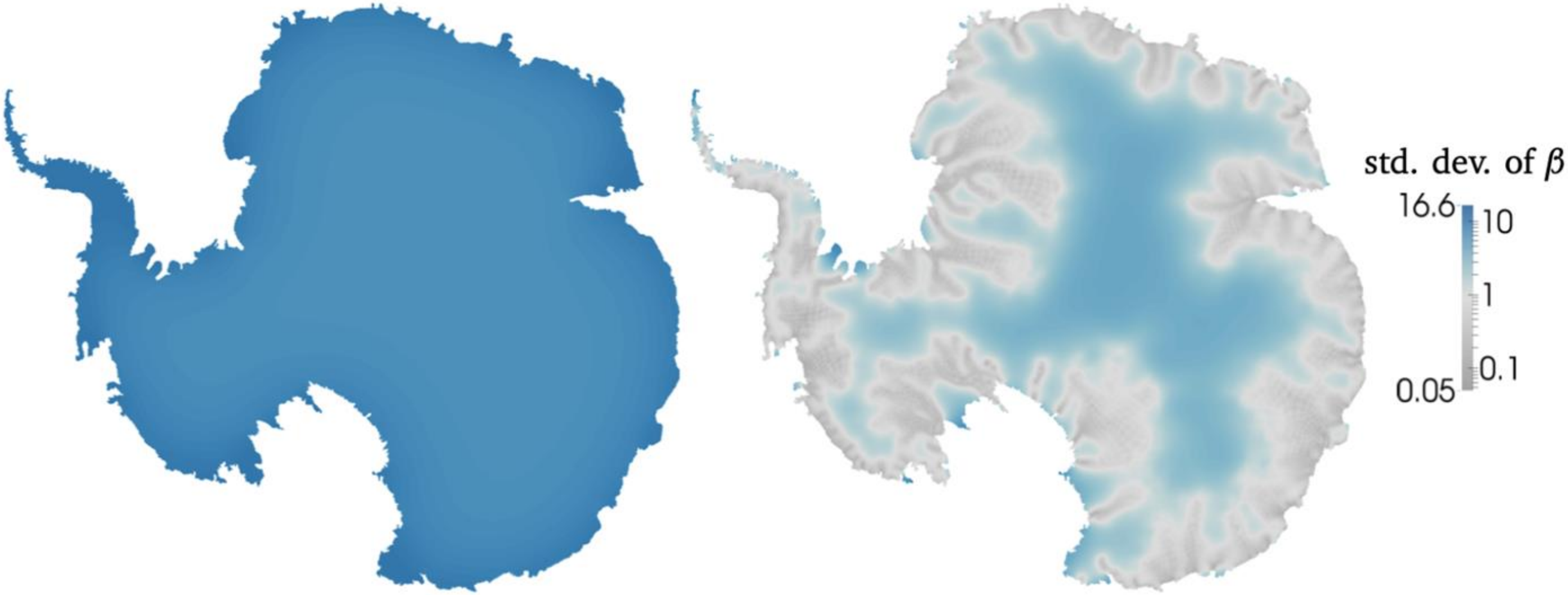


Figure: Prior and posterior point-wise marginal uncertainties