Basics of Adjoints and Algorithmic Differentiation (AD)

Shreyas Sunil Gaikwad
ECCO Hackweek

October 15, 2024

The University of Texas at Austin
Oden Institute for Computational Engineering and Sciences

Utility of gradients

As we will see soon, 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.

These gradients have many uses.

- Sensitivity Analysis (**Identifying causal pathways, explaining variability**; Ichiro's talk today)
- Optimization or Data Assimilation (**ECCO products**; see Matt Goldberg's talk tomorrow)
- Uncertainty Quantification
- Design Ocean Climate Observing Systems (e.g. Nora Loose's work)

Example: 2-step non-linear simulation and sensitivity wrt initial state

Consider a 2-step non-linear model. Our QoI $\mathcal{J}$ is an explicit function of the final state $\mathbf{m}_2$ and we want to evaluate its sensitivity to the initial state $\mathbf{m}_0$.

$$\mathcal{J} = \mathcal{J}(\mathbf{m}_2) \text{ s.t. } \mathbf{m}_2 = A_2(\mathbf{m}_1) = A_2(A_1(\mathbf{m}_0))$$

Specifically in the example on the right,

$$\begin{aligned}
\text{Init} : \mathbf{m}_0 &= [m_{x0}, m_{y0}] = [1, 1] \\
1^{\text{st}} \text{ line} : \mathbf{m}_1 &= [m_{x1}, m_{y1}] = A_1(\mathbf{m}_0) = \left[m_{x0}^2 + m_{y0}, m_{y0}\right] \\
2^{\text{nd}} \text{ line} : \mathbf{m}_2 &= [m_{x2}, m_{y2}] = A_2(\mathbf{m}_1) = \left[m_{x1}, m_{x1} + m_{y1}^2\right] \\
3^{\text{rd}} \text{ line} : \mathcal{J} &= m_{x2} + m_{y2}
\end{aligned}$$

```python
def forward(mx = 1.0, my = 1.0):

    # m1 = A1(m0), A1 only modifies x-component
    mx = mx**2 + my

    # m2 = A2(m1), A2 only modifies y-component
    my = mx + my**2

    # J = J(m2)
    J = mx + my

    return J
```

Figure: Non-linear forward model code

Naive method for computing gradients: Finite differences

```
def forward(mx = 1.0, my = 1.0):

    # m1 = A1(m0), A1 only modifies x-component
    mx = mx**2 + my

    # m2 = A2(m1), A2 only modifies y-component
    my = mx + my**2

    # J = J(m2)
    J = mx + my

    return J
```

Figure: Non-linear forward model code

To compute how sensitive $\mathcal{J}$ is to perturbations in $\mathbf{m}_0$, we can use finite differences for each component of $\mathbf{m}_0$ one at a time.

$$g_x \approx \frac{\mathcal{J}(\mathbf{m}_0 + \epsilon \hat{\mathbf{e}}_x) - \mathcal{J}(\mathbf{m}_0)}{\epsilon} + \mathcal{O}(\epsilon)$$

$$g_y \approx \frac{\mathcal{J}(\mathbf{m}_0 + \epsilon \hat{\mathbf{e}}_y) - \mathcal{J}(\mathbf{m}_0)}{\epsilon} + \mathcal{O}(\epsilon)$$

```
def finite_diff_grad(mx = 1.0, my = 1.0, eps = 1.e-7):

    J = forward(mx, my)

    # Perturb in x-direction
    mx_perturb = mx + eps
    Jx_perturb = forward(mx_perturb, my)
    gx = (Jx_perturb - J) / eps

    # Perturb in y-direction
    my_perturb = my + eps
    Jy_perturb = forward(mx, my_perturb)
    gy = (Jy_perturb - J) / eps

    return np.array([gx, gy])

finite_diff_grad()
```

```
array([4.00000021, 4.00000011])
```

Figure: Finite differences, note the small errors in the gradients.

Summary

Finite differences are a $\mathcal{O}(\mathrm{N})$ **non-linear forward runs** method to get the sensitivity gradient.

A better method: tangent linear model or linearized forward model

Consider a 2-step non-linear model. Our QoI $\mathcal{J}$ is an explicit function of the final state $\mathbf{m}_2$ and we want to evaluate its sensitivity to the initial state $\mathbf{m}_0$.

$$\mathcal{J} = \mathcal{J}(\mathbf{m}_2) \text{ s.t. } \mathbf{m}_2 = A_2(\mathbf{m}_1) = A_2(A_1(\mathbf{m}_0))$$

What is a Jacobian in a simple example?

Think back to high school calculus. Linearizing $y = x^2$ gives $dy = 2xdx$ that is linear in dx, dy. The Jacobian in this case is $2x$, a scalar since x, y are scalars. If x, y are vectors, the Jacobian is a matrix.

Linearizing the non-linear forward model ($\mathbf{B}_i$ are Jacobian matrices for non-linear operators A_i),

$$d\mathcal{J} = \nabla_{\mathbf{m}_2}\mathcal{J} \cdot d\mathbf{m}_2 \text{ s.t. } d\mathbf{m}_2 = \mathbf{B}_2(\mathbf{m}_1)\mathbf{B}_1(\mathbf{m}_0)d\mathbf{m}_0$$

Mathematically, we are just computing the directional derivatives in the direction $d\mathbf{m}_0$! Replace $d\mathbf{m}_0$ with $\hat{\mathbf{e}}_i$ to compute the i^{th} component of the gradient.

$$\mathbf{g}_i = \nabla_{\mathbf{m}_2}\mathcal{J}^T\mathbf{B}_2(\mathbf{m}_1)\mathbf{B}_1(\mathbf{m}_0)\hat{\mathbf{e}}_i$$

Summary

Tangent linear models is a $\mathcal{O}(\mathbf{N})$ **linearized** and **non-linear** forward runs method to get the gradient.

Line by line tangent linear model code

Forward model	Forward code
$\mathbf{m}_1 = A_1(\mathbf{m}_0) = [m_{x0}^2 + m_{y0}, m_{y0}]$	`mx = mx**2 + my`
$\mathbf{m}_2 = A_2(\mathbf{m}_1) = [m_{x1}, m_{x1} + m_{y1}^2]$	`my = mx + my**2`
$\mathcal{J} = m_{x2} + m_{y2}$	`J = mx + my`

TLM model	TLM code
$d\mathbf{m}_1 = \mathbf{B}_1(\mathbf{m}_0)d\mathbf{m}_0 = [2m_{x0}dm_{x0} + dm_{y0}, dm_{y0}] \implies \begin{bmatrix} dm_{x1} \\ dm_{y1} \end{bmatrix} = \begin{bmatrix} 2m_{x0} & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} dm_{x0} \\ dm_{y0} \end{bmatrix}$	`dmx = 2*mx*dmx + dmy`
$d\mathbf{m}_2 = \mathbf{B}_2(\mathbf{m}_1)d\mathbf{m}_1 = [dm_{x1}, dm_{x1} + 2m_{y1}dm_{y1}] \implies \begin{bmatrix} dm_{x2} \\ dm_{y2} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 2m_{y1} \end{bmatrix} \begin{bmatrix} dm_{x1} \\ dm_{y1} \end{bmatrix}$	`dmy = dmx + 2*my*dmy`
$d\mathcal{J} = \nabla_{\mathbf{m}_2}\mathcal{J}^T d\mathbf{m}_2 = \begin{bmatrix} 1 & 1 \end{bmatrix} \begin{bmatrix} dm_{x2} \\ dm_{y2} \end{bmatrix} = dm_{x2} + dm_{y2} \implies \begin{bmatrix} dm_{x2} \\ dm_{y2} \\ dJ \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} dm_{x2} \\ dm_{y2} \\ dJ \end{bmatrix}$	`g[i] = dmx + dmy`

(I'm taking some liberties with the notation for simplicity and conciseness. Technically, the first two Jacobian matrices could also be 4×4 instead of 2×2 depending on how you look at it.)

Code for Tangent Linear Model

```
def forward(mx = 1.0, my = 1.0):

    # m1 = A1(m0), A1 only modifies x-component
    mx = mx**2 + my

    # m2 = A2(m1), A2 only modifies y-component
    my = mx + my**2

    # J = J(m2)
    J = mx + my

    return J
```

Figure: Non-linear forward model code

Replace $d\mathbf{m}_0$ with $\hat{\mathbf{e}}_x$ to compute the x-component of the gradient (and same for y).

$$g_x = \nabla_{\mathbf{m}_2}\mathcal{J}^T\mathbf{B}_2(\mathbf{m}_1)\mathbf{B}_1(\mathbf{m}_0)\hat{\mathbf{e}}_x$$

$$g_y = \nabla_{\mathbf{m}_2}\mathcal{J}^T\mathbf{B}_2(\mathbf{m}_1)\mathbf{B}_1(\mathbf{m}_0)\hat{\mathbf{e}}_y$$

```
def TLM_grad(mx = 1.0, my = 1.0):

    mx_orig = mx
    my_orig = my

    # x-component of gradient
    dmx = 1.0
    dmy = 0.0

    dmx = 2*mx*dmx + dmy # dm1 = B1(m0)*dm0
    dmy = dmx + 2*my*dmy # dm2 = B2(m1)*dm1
    gx  = dmx + dmy      # dJ = gx

    # y-component of gradient
    dmx = 0.0
    dmy = 1.0

    dmx = 2*mx*dmx + dmy # dm1 = B1(m0)*dm0
    dmy = dmx + 2*my*dmy # dm2 = B2(m1)*dm1
    gy  = dmx + dmy      # dJ = gy

    return np.array([gx, gy])

TLM_grad()
```

```
array([4., 4.])
```

What is an adjoint operator?

For our purposes in the Euclidean space, just the transpose!

For any linear operator $\mathbf{A}$ in the Euclidean space, there always exists an adjoint operator that is the transpose of $\mathbf{A}$, such that the following identities hold.

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

Quick takeaway

When you see adjoints, think transpose of some matrix!

Using the adjoint operator to get the gradients

Consider a 2-step non-linear model. Our QoI $\mathcal{J}$ is an explicit function of the final state $\mathbf{m}_2$ and we want to evaluate its sensitivity to the initial state $\mathbf{m}_0$.

$$\mathcal{J} = \mathcal{J}(\mathbf{m}_2) \text{ s.t. } \mathbf{m}_2 = A_2(\mathbf{m}_1) = A_2(A_1(\mathbf{m}_0))$$

Taking derivative w.r.t $\mathbf{m}_0$ and using the chain rule, we get

$$\nabla_{\mathbf{m}_0}\mathcal{J}(\mathbf{m}_2) = \mathbf{B}_1(\mathbf{m}_0)^T\mathbf{B}_2(\mathbf{m}_1)^T\nabla_{\mathbf{m}_2}\mathcal{J}(\mathbf{m}_2)$$

Using the adjoint operator to get the gradients

Consider a 2-step non-linear model. Our QoI $\mathcal{J}$ is an explicit function of the final state $\mathbf{m}_2$ and we want to evaluate its sensitivity to the initial state $\mathbf{m}_0$.

$$\mathcal{J} = \mathcal{J}(\mathbf{m}_2) \text{ s.t. } \mathbf{m}_2 = A_2(\mathbf{m}_1) = A_2(A_1(\mathbf{m}_0))$$

Taking derivative w.r.t $\mathbf{m}_0$ and using the chain rule, we get

$$\nabla_{\mathbf{m}_0}\mathcal{J}(\mathbf{m}_2) = \mathbf{B}_1(\mathbf{m}_0)^T\mathbf{B}_2(\mathbf{m}_1)^T\nabla_{\mathbf{m}_2}\mathcal{J}(\mathbf{m}_2)$$

Takeaway

The forward model flows from $\mathbf{m_0} \rightarrow \mathbf{m_1} \rightarrow \mathbf{m_2} \rightarrow \mathcal{J}$, but the adjoint model flows in reverse i.e. $\mathcal{J} \rightarrow \mathbf{m_2} \rightarrow \mathbf{m_1} \rightarrow \mathbf{m_0}$, passing information backwards.

Using the adjoint operator to get the gradients

Consider a 2-step non-linear model. Our QoI $\mathcal{J}$ is an explicit function of the final state $\mathbf{m}_2$ and we want to evaluate its sensitivity to the initial state $\mathbf{m}_0$.

$$\mathcal{J} = \mathcal{J}(\mathbf{m}_2) \text{ s.t. } \mathbf{m}_2 = A_2(\mathbf{m}_1) = A_2(A_1(\mathbf{m}_0))$$

Taking derivative w.r.t $\mathbf{m}_0$ and using the chain rule, we get

$$\nabla_{\mathbf{m}_0}\mathcal{J}(\mathbf{m}_2) = \mathbf{B}_1(\mathbf{m}_0)^T\mathbf{B}_2(\mathbf{m}_1)^T\nabla_{\mathbf{m}_2}\mathcal{J}(\mathbf{m}_2)$$

Takeaway

The forward model flows from $\mathbf{m_0} \to \mathbf{m_1} \to \mathbf{m_2} \to \mathcal{J}$, but the adjoint model flows in reverse i.e. $\mathcal{J} \to \mathbf{m_2} \to \mathbf{m_1} \to \mathbf{m_0}$, passing information backwards.

Takeaway

Adjoint models can give us the complete gradient in one forward pass and one reverse pass! This is extremely efficient. We need $\mathbf{m}_0, \mathbf{m}_1, \mathbf{m}_2$ in the reverse order of how we compute them. We need to either store them during our forward pass for use during the reverse pass or recompute them.

Line by line adjoint code

TLM model	TLM code
$\mathbf{dm}_1 = \mathbf{B}_1(\mathbf{m}_0)\mathbf{dm}_0 = [2m_{x0}dm_{x0} + dm_{y0}, dm_{y0}] \implies \begin{bmatrix} dm_{x1} \\ dm_{y1} \end{bmatrix} = \begin{bmatrix} 2m_{x0} & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} dm_{x0} \\ dm_{y0} \end{bmatrix}$	`dmx = 2*mx*dmx + dmy`
$\mathbf{dm}_2 = \mathbf{B}_2(\mathbf{m}_1)\mathbf{dm}_1 = [dm_{x1}, dm_{x1} + 2m_{y1}dm_{y1}] \implies \begin{bmatrix} dm_{x2} \\ dm_{y2} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 2m_{y1} \end{bmatrix} \begin{bmatrix} dm_{x1} \\ dm_{y1} \end{bmatrix}$	`dmy = dmx + 2*my*dmy`
$d\mathcal{J} = \nabla_{\mathbf{m}_2}\mathcal{J}^T \mathbf{dm}_2 = \begin{bmatrix} 1 & 1 \end{bmatrix} \begin{bmatrix} dm_{x2} \\ dm_{y2} \end{bmatrix} = dm_{x2} + dm_{y2} \implies \begin{bmatrix} dm_{x2} \\ dm_{y2} \\ dJ \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} dm_{x2} \\ dm_{y2} \\ dJ \end{bmatrix}$	`g[i] = dmx + dmy`

Transpose!!!

Note that the matrices in here are the transpose of the Jacobian matrices in TLM.

Adjoint model	Adjoint code
$\frac{\partial J}{\partial J} = 1$	`Jb = 1`

$\nabla_{\mathbf{m}_2}\mathcal{J} = [1, 1] \implies \begin{bmatrix} m^*_{x2} \\ m^*_{y2} \\ J^* \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} m^*_{x2} \\ m^*_{y2} \\ J^* \end{bmatrix}$	`mxb = mxb + Jb`
	`myb = myb + Jb`
	`Jb = 0`

$\nabla_{\mathbf{m}_1}\mathcal{J} = \mathbf{B}_2(\mathbf{m}_1)^T \nabla_{\mathbf{m}_2}\mathcal{J} \implies \begin{bmatrix} m^*_{x1} \\ m^*_{y1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & 2m_{y1} \end{bmatrix} \begin{bmatrix} m^*_{x2} \\ m^*_{y2} \end{bmatrix}$	`mxb = mxb + myb` `myb = 2*my*myb`
$\nabla_{\mathbf{m}_0}\mathcal{J} = \mathbf{B}_1(\mathbf{m}_0)^T \nabla_{\mathbf{m}_1}\mathcal{J} \implies \begin{bmatrix} m^*_{x0} \\ m^*_{y0} \end{bmatrix} = \begin{bmatrix} 2m_{x0} & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} m^*_{x1} \\ m^*_{y1} \end{bmatrix}$	`myb = mxb + myb` `mxb = 2*mx*mxb`

Code for the adjoint model

```
def forward(mx = 1.0, my = 1.0):

    # m1 = A1(m0), A1 only modifies x-component
    mx = mx**2 + my

    # m2 = A2(m1), A2 only modifies y-component
    my = mx + my**2

    # J = J(m2)
    J = mx + my

    return J
```

Figure: Non-linear forward model code

N-dimensional gradient in a single forward and reverse pass.

$$\nabla_{\mathbf{m}_0}\mathcal{J}(\mathbf{m}_2) = \mathbf{B}_1(\mathbf{m}_0)^T\mathbf{B}_2(\mathbf{m}_1)^T\nabla_{\mathbf{m}_2}\mathcal{J}(\mathbf{m}_2)$$

Summary

Adjoint models give us the complete gradient in one forward and reverse pass! We need $\mathbf{m}_0, \mathbf{m}_1, \mathbf{m}_2$ in the reverse order of how we compute them. We need to either store them during our forward pass for use during the reverse pass or recompute them.

```
def adj_store_grad(mx = 1.0, my = 1.0):

    tape = []

    tape.append([mx, my]) # Store m0
    mx = mx**2 + my

    tape.append([mx, my]) # Store m1
    my = mx + my**2

    J  = mx + my

    mxb = 0.0
    myb = 0.0
    Jb  = 1.0 # dJ/dJ = 1.0

    # m2b = dJ/dm2
    mxb = mxb + Jb
    myb = myb + Jb
    Jb  = 0.0

    # m1b = B2(m1)^T * m2b
    mx, my = tape.pop()
    mxb = mxb + myb
    myb = 2*my*myb

    # m0b = B1(m0)^T * m1b
    mx, my = tape.pop()
    myb = mxb + myb
    mxb = 2*mx*mxb

    # g = dJ/dm0 = m0b
    return np.array([mxb, myb])

adj_store_grad()
```

```
array([4., 4.])
```

Hopefully, I have convinced you that this is tedious to do by hand for just 3 lines of code.

Hopefully, I have convinced you that this is tedious to do by hand for just 3 lines of code.

You have to redo a lot of things if the forward model/list of controls changes.

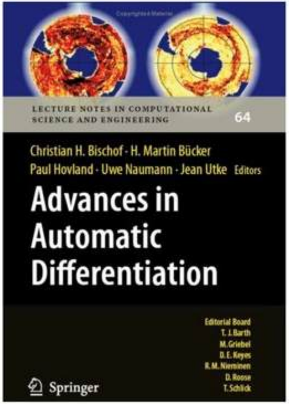
Hopefully, I have convinced you that this is tedious to do by hand for just 3 lines of code.

You have to redo a lot of things if the forward model/list of controls changes.

MITgcm has hundreds of thousands of lines of code.

Why Automatic/Algorithmic Differentiation (AD)?

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)

Why Automatic/Algorithmic Differentiation (AD)?

- Inaccuracy of finite differences (**Need exact derivatives**)
- Dimensionality of the problem is too large (**Need efficient/cheap gradients**)
- Code changes frequently (**Error-free and timely updates to adjoint code**)
- Changing set of controls (**Ability to auto-generate adjoint code for specific controls**)

Open source AD tool Tapenade

- Tapenade is an Open-Source Automatic Differentiation (AD) Engine developed at INRIA, France.
- Developers – Laurent Hascoët and Valérie Pascual
- It takes in as input a source code and list of independent and dependent variables.
- Outputs a differentiated code that evaluates the gradients of the dependent variables with respect to the independent variables.
- It's freely accessible and actively maintained.

Tapenade vs TAF: store-all vs recompute-all

- TAF achieves its performance through the targeted insertion of directives for storing variables.
- TAF is by default recompute-all.
- MITgcm has around 3,000 TAF store directives.
- Labor-intensive and error-prone adjoint code performance tuning.

Tapenade vs TAF: store-all vs recompute-all

- TAF achieves its performance through the targeted insertion of directives for storing variables.
- TAF is by default recompute-all.
- MITgcm has around 3,000 TAF store directives.
- Labor-intensive and error-prone adjoint code performance tuning.

- Tapenade is by default store-all.
- This simplifies the implementation of efficient, AD-compatible code with Tapenade.

Tapenade vs TAF: `tutorial_global_oce_biogeo`

- $\mathcal{J}$: globally integrated air-sea flux of CO_2 on the final time-step.
- Control: initial SST (`theta`).
- Relative difference between TAF and Tapenade results is negligible.

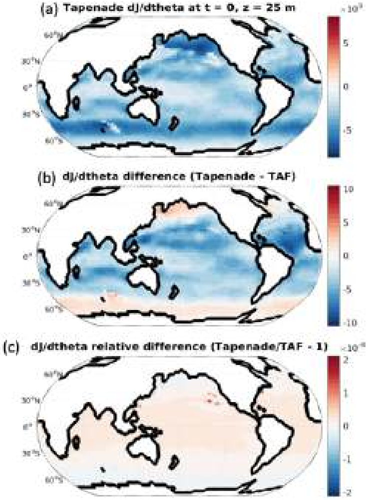


Figure: from Gaikwad et. al (2024)

Tapenade vs TAF: Checkpointing for time loops

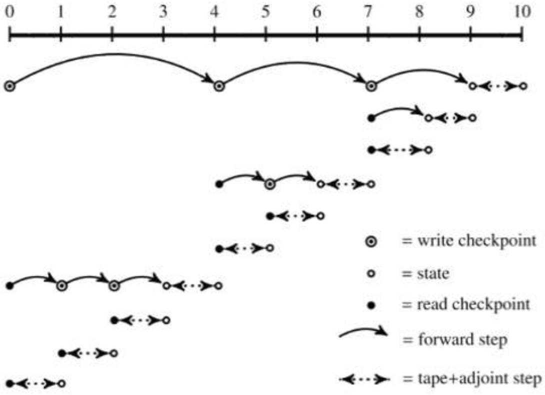


Figure: from Gaikwad et. al (2024), binomial checkpointing in Tapenade, optimal in number of recomputations.

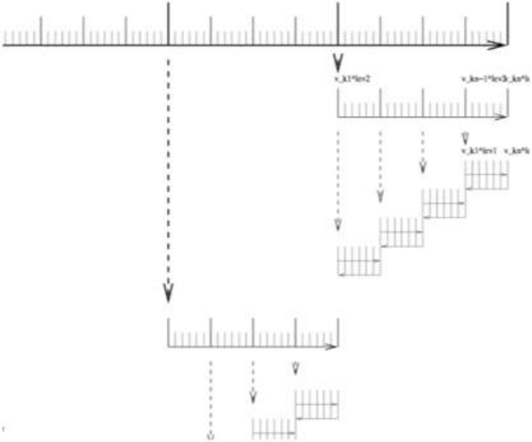


Figure: from MITgcm documentation, multi-level checkpointing with TAF.

Tapenade: New profiling tool

Table 3: Timing comparison for Tapenade (TAP) against TAF. Time in seconds. The runs are performed on Intel Xeon CPU E5-2695 v3 nodes (2.30 GHz clock rate, 35.84 MB L3 cache, 63.3 GB memory). Checkpointing intervals were optimized for the TAF runs to hold the maximum number of timesteps in memory; multi-level checkpointing was required only for `global_ocean.cs32x15`.

Experiment	Forward	TAF	TAP	TAP/TAF
global_with_exf	4.0	15.8	35.4	2.2
global_ocean.cs32x15	30.7	173.4	1112.2	6.4
global_oce_biogeo	26.4	53.4	~~165.0~~	~~3.1~~
20% additional stacksize compared to own			102.3	1.9
150% additional stacksize compared to own			76.3	1.4

Figure: from Gaikwad et. al (2024), updated values are using checkpointing optimization using the profiling tool, as in Hascoet et. al (2024).

Tapenade: New profiling tool

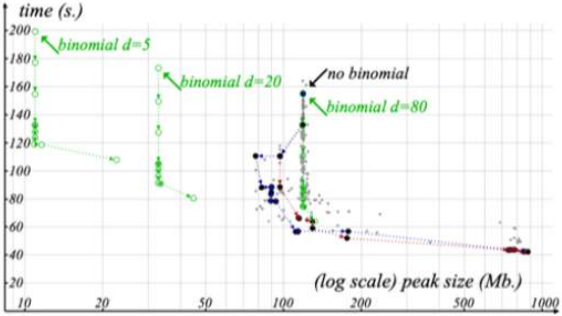


Figure: from Hascoet et. al (2024), profiling for `halfpipe_streamice`.

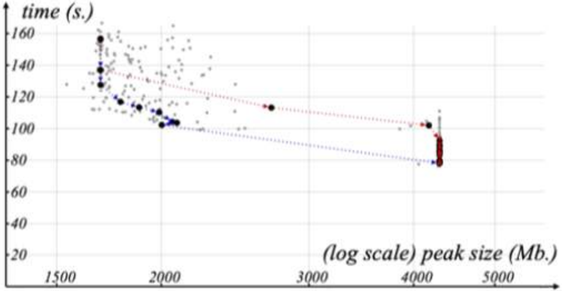


Figure: from Hascoet et. al (2024), profiling for `tutorial_global_oce_biogeo`.

Tapenade: Current capabilities

Verification Experiments

tutorial_tracer_adjsens	global_with_exf
tutorial_global_oce_biogeo	halfpipe_streamice
isomip	labsea
global_ocean.cs32x15	global_oce_biogeo_bling
tutorial_barotropic_gyre	tutorial_baroclinic_gyre

Packages

exch2	cd_code	kpp	gfd	streamice	ptracers
gmredi	monitor	tapenade	exf	shelfice	thsice
cal	cost	ctrl	seaice	dic	grdchk
diagnostics	gchem	ggl90	ecco	profiles	mnc

Tapenade: Future Outlook

- Tapenade is easy to use, actively maintained, and free
- Tapenade can generate tangent linear code, unlike OpenAD
- Tapenade has a new profiling capability for error-free optimization of memory vs recomputations.
- List of future updates documented in Issue 735 on MITgcm Github.

We need more people to use Tapenade for further improvement.
If interested, please check out tutorials on AD on the Hackweek website!