# AD Master Class 5

## Bootstrapping validated adjoints on real-world codes

**Viktor Mosenkis**

viktor.mosenkis@nag.co.uk

*27 August 2020*

nag® Experts in numerical algorithms and HPC services

# AD Masterclass Schedule and Remarks

## AD Masterclass Schedule

- ☐ 30 July 2020 | Why the need for Algorithmic Differentiation?
- ☐ 6 August 2020 | How AD works
- ☐ 13 August 2020 | Testing and validation
- ☐ 20 August 2020 | Pushing performance using SIMD vectorization
- ☐ 27 August 2020 | Bootstrapping validated adjoints on real-world codes

## Remarks

- ☐ Please submit your questions via the questions panel at any time during this session, these will be addressed at the end.
- ☐ A recording of this session, along with the slides will be shared with you in a day or two.

# Dialogue

We want this webinar series to be interactive (even though it's hard to do)

- We want your feedback, we want to adapt material to your feedback
- Please feel free to contact us via email to ask questions at any time
- We'd love to reach out offline, discuss what's working, what to spend more time on
- For some orgs, may make sense for us to do a few bespoke sessions

Blog:
https://www.nag.com/blog/algorithmic-differentiation-masterclass-series-2

# Outcomes

This session is about getting an adjoint to run at all, with minimal user effort. This is a necessary first step before you start doing more advanced adjoint techniques, which we use (in combination with validating adjoint code) to check that our advanced adjoint techniques are working.

In this session we will discuss following automatic techniques to avoid running out of memory when computing the adjoints

- write tape to disk
- compressing the vector of adjoints
- Jacobian pre-accumulation

# Understanding tape structure in modern AD-tool

Modern operator overloading AD-tools use so called partial(s) tape instead of opcode tape. During tape recording the local partial derivatives of language intrinsics (statements) are computed and stored. During tape interpretation only fused multiply adds needs to be performed to compute the adjoints.

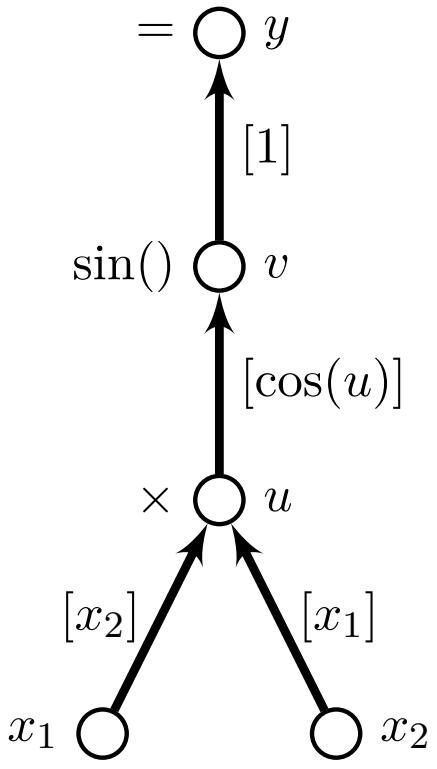
- faster tape interpretation
  - no switch statement as in opcode tape
  - local partial derivatives are computed once
  - pre-accumulation of statements (basic blocks) possible
- tape and vector of adjoints can be separated
- tape is accessed sequentially
- vector of adjoints is accessed randomly

# Understanding tape structure in modern AD-tool

$F : \mathbb{R}^2 \to \mathbb{R}, \quad y = \sin(x_1 \cdot x_2)$

After recording the tape

| Tape | | Vec of Adj |
|---|---|---|
| Indx | Partial deriv. | |
| 0 | $(1, \cdot)$ | $\bar{x_1}$ |
| 1 | $(1, \cdot)$ | $\bar{x_2}$ |
| 2 | $(x_2, 0),\ (x_1, 1)$ | $\bar{u}$ |
| 3 | $(\cos(u), 2)$ | $\bar{v}$ |
| 4 | $(1, 3)$ | $\bar{y}$ |

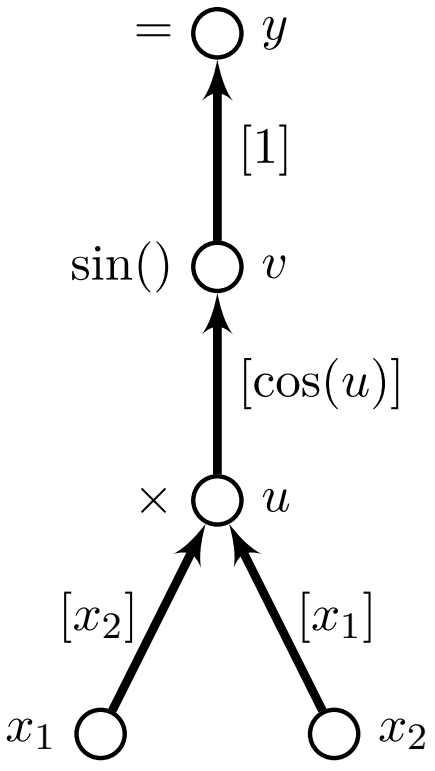


# Understanding tape structure in modern AD-tool

$F : \mathbb{R}^2 \to \mathbb{R}, \quad y = \sin(x_1 \cdot x_2)$

Tape interpretation

| Tape | | Vec of Adj |
|---|---|---|
| Indx | Partial deriv. | |
| 0 | $(1, \cdot)$ | $\bar{x_1}$ |
| 1 | $(1, \cdot)$ | $\bar{x_2}$ |
| 2 | $(x_2, 0), (x_1, 1)$ | $\bar{u}$ |
| 3 | $(\cos(u), 2)$ | $\bar{v}$ |
| 4 | $(1, 3)$ | $\bar{y} = 1$ |



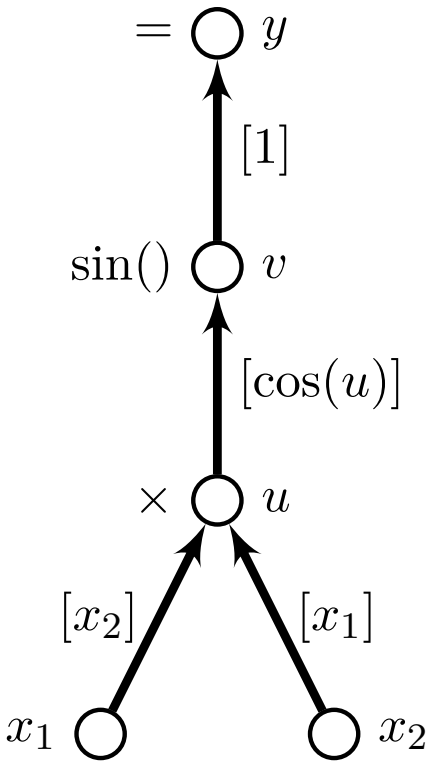

Set $\bar{y} = 1.0$ and start tape interpretation

# Understanding tape structure in modern AD-tool

$F : \mathbb{R}^2 \to \mathbb{R}, \quad y = \sin(x_1 \cdot x_2)$

Tape interpretation

| Tape | | Vec of Adj |
|---|---|---|
| Indx | Partial deriv. | |
| 0 | $(1, \cdot)$ | $\bar{x_1}$ |
| 1 | $(1, \cdot)$ | $\bar{x_2}$ |
| 2 | $(x_2, 0), (x_1, 1)$ | $\bar{u}$ |
| 3 | $(\cos(u), 2)$ | $\bar{v} += 1 \cdot \bar{y}$ |
| $\to$ 4 | $(1, 3)$ | $\bar{y} = 1$ |



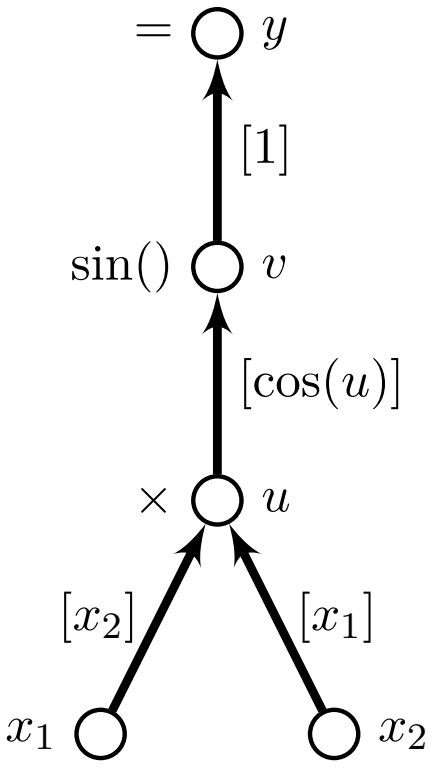

Set $\bar{y} = 1.0$ and start tape interpretation

# Understanding tape structure in modern AD-tool

$F : \mathbb{R}^2 \to \mathbb{R}, \quad y = \sin(x_1 \cdot x_2)$

Tape interpretation

| Tape | | Vec of Adj |
|---|---|---|
| Indx | Partial deriv. | |
| 0 | $(1, \cdot)$ | $\bar{x_1}$ |
| 1 | $(1, \cdot)$ | $\bar{x_2}$ |
| 2 | $(x_2, 0), (x_1, 1)$ | $\bar{u} += \cos(u) \cdot \bar{v}$ |
| $\to$ 3 | $(\cos(u), 2)$ | $\bar{v}$ |
| 4 | $(1, 3)$ | $\bar{y}$ |



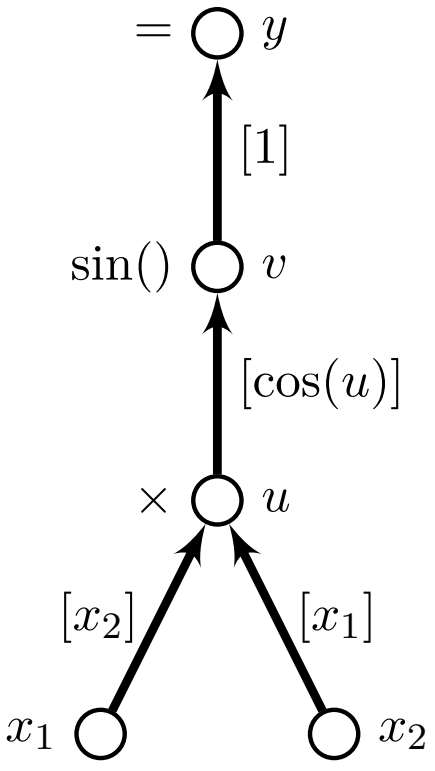

Set $\bar{y} = 1.0$ and start tape interpretation

# Understanding tape structure in modern AD-tool

$F : \mathbb{R}^2 \to \mathbb{R}, \quad y = \sin(x_1 \cdot x_2)$

Tape interpretation

| Tape | | Vec of Adj |
|---|---|---|
| Indx | Partial deriv. | |
| 0 | $(1, \cdot)$ | $\bar{x_1} += x_2 \cdot \bar{u}$ |
| 1 | $(1, \cdot)$ | $\bar{x_2} += x_1 \cdot \bar{u}$ |
| $\to$ 2 | $(x_2, 0)$, $(x_1, 1)$ | $\bar{u}$ |
| 3 | $(\cos(u), 2)$ | $\bar{v}$ |
| 4 | $(1, 3)$ | $\bar{y}$ |




Set $\bar{y} = 1.0$ and start tape interpretation

# Writing tape to disk

Instead of storing the tape in the main memory, tape can be written to disk

- disk space is cheaper compared to main memory
- significantly more storage available
- slower access
- random access is problematic
  - ☐ tape is accessed sequentially
  - ☐ vector of adjoint is accessed randomly

# Writing tape to disk in dco/c++

dco/c++ has two types of tapes

- **Blob** - Memory of the specified size is allocated and used for storing the tape without bound checks
- **Chunk** - The tape grows in chunks up to the physical memory bound.

Writing tape to file/disk is only supported with **Chunk** tape.

- Tape dynamically allocates chunks of memory of specified size
- Filled chunks result in offloading to disk followed by creation of new chunks within the previously allocated memory
- Chunks are read from disk during tape interpretation
- Vector of adjoints is not offloaded to disk

# Writing tape to disk in dco/c++

```
1  template <typename T>
2  void foo(int& n, T* x, T& y){ ... }
3
4  int main(){
5    dco::ga1s<double>::type *x, y;
6
7    dco::tape_options o;
8    //Enable writing tape to disk
9    o.write_to_file()=true;
10   DCO_MODE::global_tape=DCO_TAPE_T::create(o);
11   ...
12   foo(n, x, y);
13   ...
14 }
```

Vector of adjoints can use significant amount of memory!

# Compressing vector of adjoints

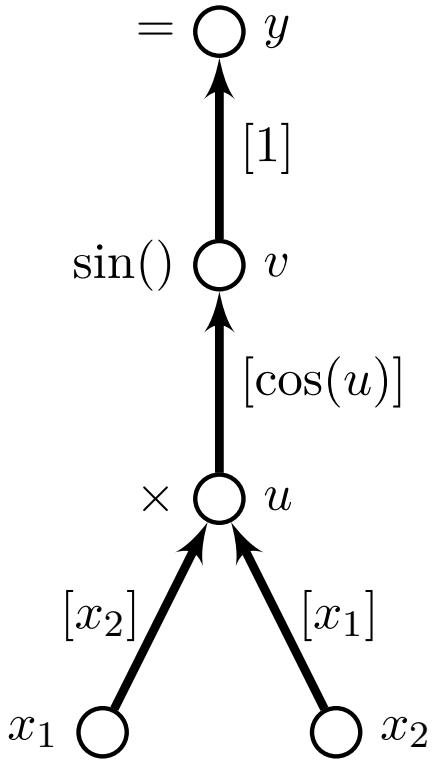
- Naive way of implementing vector of adjoints is to have a distinct adjoint memory location for each entry in the tape.
  - fast, no need to search for the required adjoint vector entries
  - high memory requirements, vector of adjoints has the same number of entries as the tape.
- Compress the vector of adjoints by analysing the maximum number of required distinct adjoint memory locations.
  - During interpretation, adjoint memory, which is no longer required, is reused
  - Especially useful for iterative algorithms (e.g. time iteration)
  - Requires many modulo operations during interpretation. Slower than naive approach
  - Works nicely with writing tape to disk
  - Uses less memory than naive approach. Trading computation time for memory.

# Compressing vector of adjoints

$F : \mathbb{R}^2 \to \mathbb{R}, \quad y = \sin(x_1 \cdot x_2)$

After recording the tape

| Tape | | Vec of Adj |
|---|---|---|
| Indx | Partial deriv. | |
| 0 | $(1, \cdot)$ | $\ldots$ |
| 1 | $(1, \cdot)$ | $\ldots$ |
| 2 | $(x_2, 0)$, $(x_1, 1)$ | $\ldots$ |
| 3 | $(\cos(u), 2)$ | |
| 4 | $(1, 3)$ | |



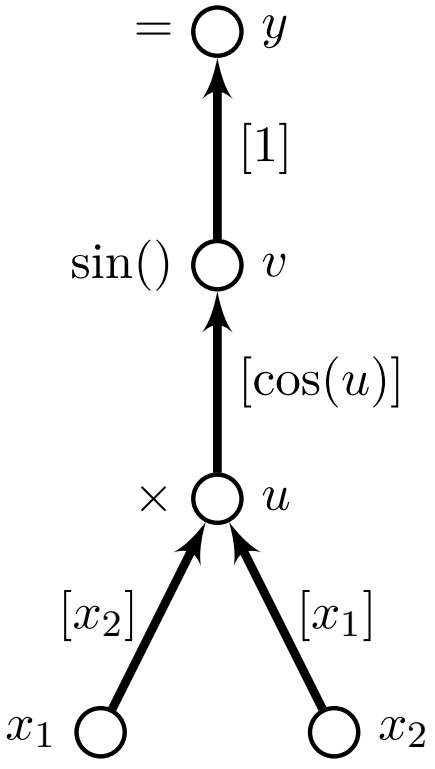

In our example we require only three distinct adjoint memory locations

# Understanding tape structure in modern AD-tool

$F : \mathbb{R}^2 \to \mathbb{R}, \quad y = \sin(x_1 \cdot x_2)$

Tape interpretation

| Tape | | Vec of Adj |
|---|---|---|
| Indx | Partial deriv. | |
| 0 | $(1, \cdot)$ | $\ldots$ |
| 1 | $(1, \cdot)$ | $\bar{y} = 1$ |
| 2 | $(x_2, 0)$, $(x_1, 1)$ | $\ldots$ |
| 3 | $(\cos(u), 2)$ | |
| 4 | $(1, 3)$ | |




Set $\bar{y} = 1.0$ and start tape interpretation

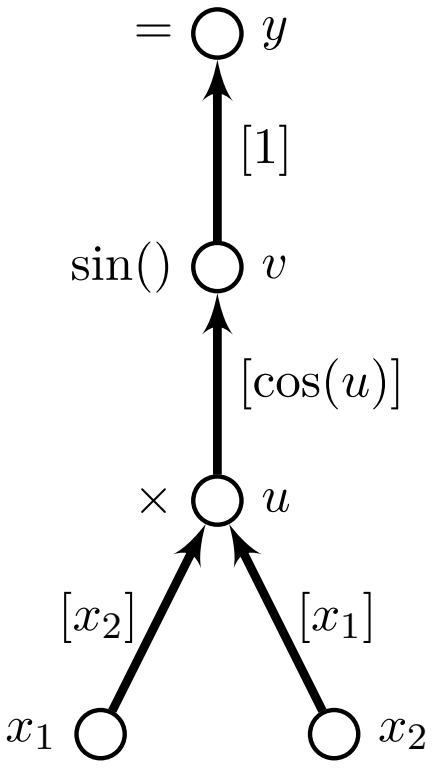
$4 \bmod 3 = 1$

# Compressing vector of adjoints

$F : \mathbb{R}^2 \to \mathbb{R}, \quad y = \sin(x_1 \cdot x_2)$

Tape interpretation

| Tape | | Vec of Adj |
|---|---|---|
| Indx | Partial deriv. | |
| 0 | $(1, \cdot)$ | $\bar{v} += 1 \cdot \bar{y}$ |
| 1 | $(1, \cdot)$ | $\bar{y}$ |
| 2 | $(x_2, 0), (x_1, 1)$ | $\cdots$ |
| 3 | $(\cos(u), 2)$ | |
| $\to$ 4 | $(1, 3)$ | |



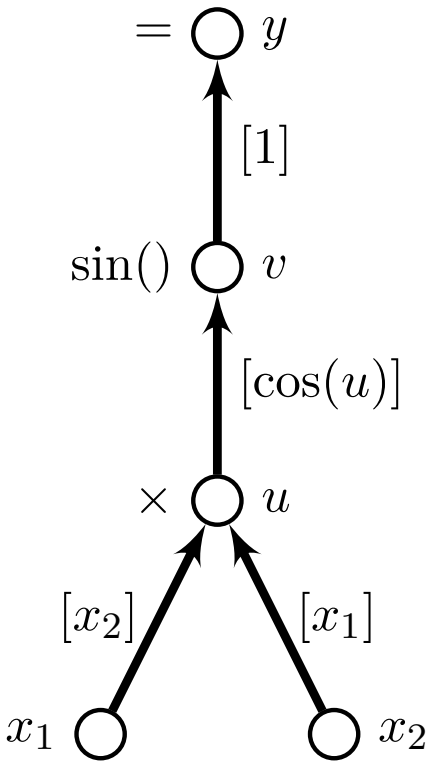

$3 \bmod 3 = 0$

# Compressing vector of adjoints

$F : \mathbb{R}^2 \to \mathbb{R}, \quad y = \sin(x_1 \cdot x_2)$

Tape interpretation

| Tape | | Vec of Adj |
|---|---|---|
| Indx | Partial deriv. | |
| 0 | $(1, \cdot)$ | $\bar{v}$ |
| 1 | $(1, \cdot)$ | $\bar{y}$ |
| 2 | $(x_2, 0), (x_1, 1)$ | $\bar{u} += \cos(u) \cdot \bar{v}$ |
| $\to$ 3 | $(\cos(u), 2)$ | |
| 4 | $(1, 3)$ | |



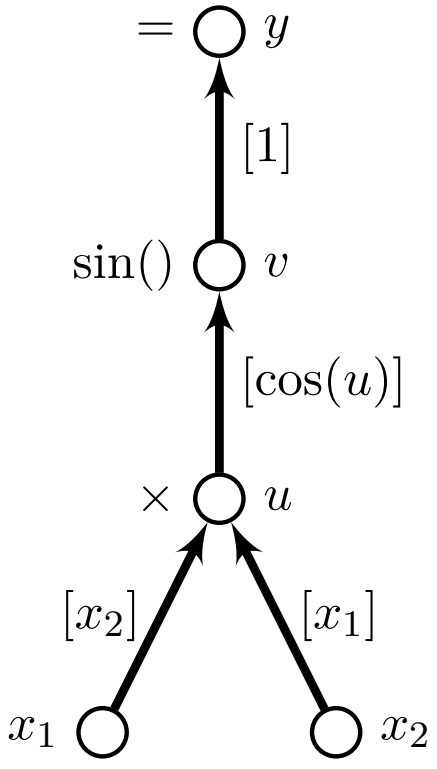

$2 \bmod 3 = 2$

# Compressing vector of adjoints

$F : \mathbb{R}^2 \to \mathbb{R}, \quad y = \sin(x_1 \cdot x_2)$

Tape interpretation

| Tape | | Vec of Adj |
|---|---|---|
| Indx | Partial deriv. | |
| 0 | $(1, \cdot)$ | $\bar{x_1} += x_2 \cdot \bar{u}$ |
| 1 | $(1, \cdot)$ | $\bar{x_2} += x_1 \cdot \bar{u}$ |
| $\to$ 2 | $(x_2, 0)$, $(x_1, 1)$ | $\bar{u}$ |
| 3 | $(\cos(u), 2)$ | |
| 4 | $(1, 3)$ | |




$1 \bmod 3 = 1$

$0 \bmod 3 = 0$

# Compressing vector of adjoints with dco/c++

```
1  template <typename T>
2  void foo(int& n, T* x, T& y){
3     ...
4  }
5
6  int main(){
7     //replaced data type
8     dco::ga1s_mod<double>::type *x, y;
9     ...
10    DCO_MODE::global_tape=DCO_TAPE_T::create();
11    ...
12    foo(n, x, y);
13    ...
14 }
```

Without writing tape to disk

Replace `ga1s<double>::type` with `ga1s_mod<double>::type`

# Compressing vector of adjoints with dco/c++

```
1  template <typename T>
2  void foo(int& n, T* x, T& y){ ... }
3  int main(){
4    //replaced data type
5    dco::ga1s_mod<double>::type *x, y;
6
7    dco::tape_options o;
8    //Enable writing tape to disk
9    o.write_to_file()=true;
10   DCO_MODE::global_tape=DCO_TAPE_T::create(o);
11   ...
12   foo(n, x, y);
13   ...
14 }
```

With writing tape to disk

Replace `ga1s<double>::type` with `ga1s_mod<double>::type`

# Reducing the tape size with Jacobian pre-accumulation

**Basic idea**

- compute the Jacobian for a specific part of the code during tape recording
- insert the Jacobian into the tape instead of recording the tape for this part of the code

**Remarks**

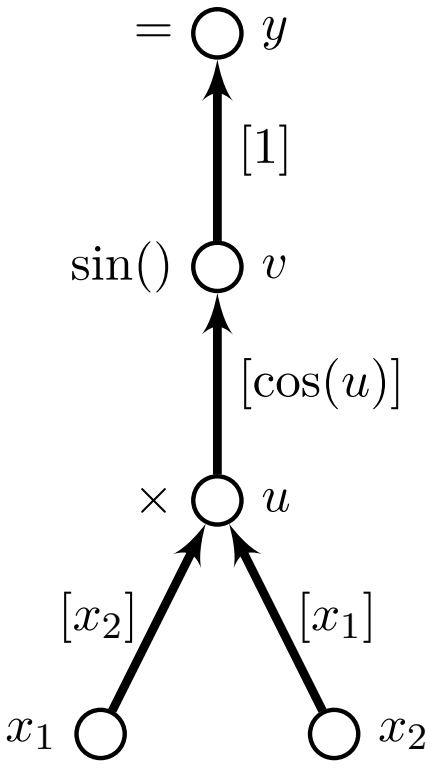
- computation of the Jacobian can be expensive, depending on the size of the input and output dimension of the underlying function
- can be implemented with small development overhead
- you must identify all active outputs of the pre-accumulated code.

# Reducing the tape size with Jacobian pre-accumulation

$F : \mathbb{R}^2 \to \mathbb{R}, \quad y = \sin(x_1 \cdot x_2)$

After recording the tape

| Tape | | Vec of Adj |
|---|---|---|
| Indx | Partial deriv. | |
| 0 | $(1, \cdot)$ | $\bar{x_1}$ |
| 1 | $(1, \cdot)$ | $\bar{x_2}$ |
| 2 | $(x_2, 0), (x_1, 1)$ | $\bar{u}$ |
| 3 | $(\cos(u), 2)$ | $\bar{v}$ |
| 4 | $(1, 3)$ | $\bar{y}$ |



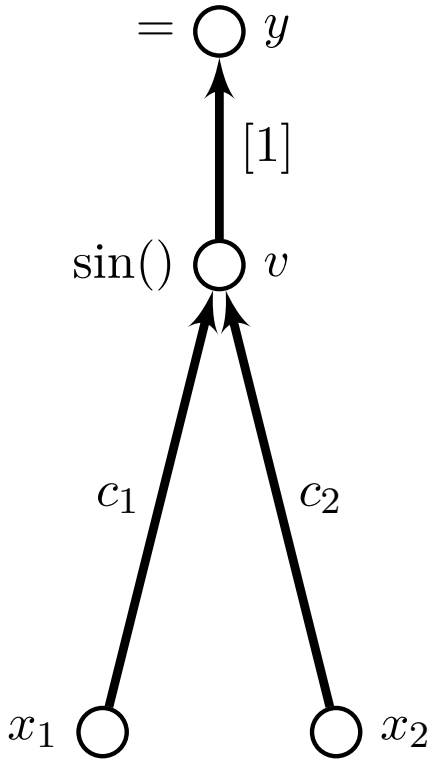

Pre-compute the Jacobian for $\sin(x_1, x_2)$

# Reducing the tape size with Jacobian pre-accumulation

$F : \mathbb{R}^2 \to \mathbb{R}, \quad y = \sin(x_1 \cdot x_2)$

After recording the tape

| Tape | | Vec of Adj |
|---|---|---|
| Indx | Partial deriv. | |
| 0 | $(1, \cdot)$ | $\bar{x_1}$ |
| 1 | $(1, \cdot)$ | $\bar{x_2}$ |
| 2 | $(c_1, 0)$, $(c_2, 1)$ | $\bar{v}$ |
| 3 | $(1, 3)$ | $\bar{y}$ |




Pre-compute the Jacobian for $\sin(x_1, x_2)$.

$c_1 = \cos(u) \cdot x_2, \qquad c_2 = \cos(u) \cdot x_1$

# Jacobian pre-accumulation with dco/c++

- easy to use interface
- pre-accumulation is performed through tape interpretation of the corresponding code
- almost no slowdown for pre-accumulation functions with one output
- number of tape interpretations correspond to the number of outputs of the pre-accumulated function
- you must register all active outputs of the function
- parallel edges are currently not merged (expected in the next version of dco/c++)

# Jacobian pre-accumulation with dco/c++

```
1  template <typename T>
2  void foo(int& n, T* x, T& y){ ... }
3
4  int main(){
5    ...
6    DCO_M::jacobian_preaccumulator_t jp(DCO_M::global_tape);
7    for (size_t i = 0; i < num_mcpath; i++) {
8      jp.start();
9      f(n, x, y);
10     jp.register_output(y);
11     jp.finish();
12   }
13   ...
14 }
```

# Jacobian pre-accumulation: register all outputs

```
1  template <typename T>
2  void foo(const T& x, T& y, T& z){
3   y = sin(x);
4   z = cos(x);
5  }
6  int main(){
7     ...
8     DCO_M::jacobian_preaccumulator_t jp(DCO_M::global_tape);
9     jp.start();
10    f(x, y, z);
11    jp.register_output(y);
12    jp.finish();
13    ...
14   }
```

Wrong result as output `z` is not registered

# Which parts should be pre-accumulated

- the tape size of the code should fit into the main memory
- the tape size of the code should be big enough to see the difference
- you should be able to identify all outputs of the code you are trying to pre-accumulate
- the number of outputs should be small. Otherwise big overhead due to several tape interpretations

# Summary

- Writing tape to file
  - brute force method using disk to store the tape (slowdown)
  - adjoint vector should not be written to disk.
  - compression methods should be used to reduce size of the adjoint vector
- Compressing adjoint vector
  - significantly reduces the memory occupied by the adjoint vector
  - saves $> 90\%$ of adjoint vector and 20%-30% of overall (tape + adjoint vector) memory use
  - only decent slowdown
- Jacobian pre-accumulation
  - can significantly reduce the tape size
  - slow down is number for outputs is greater than one
  - difficult to find a suitable code part
  - must identify all outputs of the code

# AD Masterclass Series: **Advanced Adjoint Techniques**

- **Checkpointing and external functions**: Manipulating the DAG
- **Checkpointing and external functions**: Injecting symbolic information
- **Monte Carlo**
- **Computing Hessians**
- **Guest lecture by Prof Uwe Naumann**: Adjoint Code Design Patterns applied to Monte Carlo
- **Guest lecture by Prof Uwe Naumann**: Advanced AD topics in Machine Learning

Do let us know of any topics you'd like us to spend time on. If we can't fit it into the second Masterclass series, or if you can't wait until then, we can always try to schedule a private discussion. Please get in touch with Jacques - jacques@nag.co.uk.

You will see a survey on your screen after exiting from this session.
We would appreciate your feedback.

We are now moving on the Q&A Session