Automatic differentiation in PSyclone

A prototype using reverse-mode AD

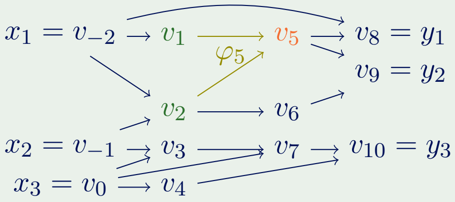
Team meeting
July 24, 2023
Edited slide 8, July 28

Julien REMY (M1 student - Martin's trainee)
julien.remy@grenoble-inp.org

Automatic differentation

Program

Suppose we have a program computing return values of variables y_j from values of arguments x_i through intermediate values v_k, where each value is obtained from its direct predecessors through *elemental* operations $(+, \times, /, \exp, \text{etc.})$.



Notations

Relation: $i \prec j$ if v_j depends on v_i, *eg.* $1 \prec 5$.

Predecessors: $u_j := (v_i)_{i \prec j}$ *eg.* $u_5 = \begin{pmatrix} v_1 \\ v_2 \end{pmatrix}$

Operation: $\varphi_j : u_j \mapsto v_j$ *eg.* $v_5 = \varphi_5(u_5)$

AD: Obtain $\dfrac{\partial y_j}{\partial x_i}(x_1, \ldots, x_n)$ automatically, by differentiating operations $\varphi_k : u_k \mapsto v_k$.

Forward mode Tangent linear

Derivatives

For **one *independent variable*** d chosen among the **argument** variables, say $d = x_1$,

let
$$\dot{v}_i = \frac{\partial v_i}{\partial d}$$

$$\overleftarrow{\dot{v}_{k+2} = \frac{\partial v_{k+2}}{\partial v_{k+1}} \underbrace{\left(\frac{\partial v_{k+1}}{\partial v_k} \dot{v}_k \right)}_{\dot{v}_{k+1}}}$$

The chain rule gives

Easy to implement.

$$\dot{v}_j = \sum_{i \prec j} \frac{\partial \varphi_j}{\partial v_i}(u_j)\dot{v}_i$$

Initialize $\dot{d} = 1$,
and $\dot{v}_k = 0, \forall v_k \neq d$,
get $\left(\frac{\partial y_j}{\partial d}(x_1, \ldots x_n) \right)_j$

Issue

We obtain $J((x_i)_i)(0 \ldots \dot{d} \ldots 0)^T$: all the derivatives $(\dot{y}_j)_j$ wrt a **single** $d \in (x_i)_i$.
Interesting if $\#\{x_i\}_i \ll \#\{y_j\}_j$ but usually there are many arguments, few results.

Reverse mode Adjoint

Adjoints

For **one *dependent variable*** z chosen among the **return** variables, say $z = y_1$, denote

$$\bar{v}_i = \frac{\partial z}{\partial v_i}$$

$$\overrightarrow{\underbrace{\left(\bar{v}_k \frac{\partial v_k}{\partial v_{k-1}}\right)}_{\bar{v}_{k-1}} \frac{\partial v_{k-1}}{\partial v_{k-2}}} = \bar{v}_{k-2}$$

The chain rule gives

In practice, increment.

$$\bar{v}_i = \sum_{j \succ i} \bar{v}_j \frac{\partial \varphi_j}{\partial v_i}(\overset{?}{u_j})$$

Initialize $\bar{z} = 1$,
and $\bar{v}_k = 0, \forall v_k \neq z$,
get $\left(\frac{\partial z}{\partial x_i}(x_1, \ldots, x_n)\right)_i$

Remark

We obtain the adjoints $(\bar{x}_i)_i$ of **all arguments** x_i for a single differentiated $z \in (y_j)_j$.
Compute $\nabla^T z(x_1, \ldots, x_n) = (0 \ldots \bar{z} \ldots 0) J(x_1, \ldots, x_n)$ in one evaluation!

Reverse mode AD A simple math example, with non linearities

x_1

$v_1 = x_1^2$

$v_3 = \exp(x_1^2)$

$y_1 = \exp(x_1^2) - 3 * x_1 + x_2$

$v_2 = 3 * x_1$

$v_4 = 3 * x_1 + x_2$

$y_2 = x_2 * (3 * x_1 + x_2)$

3

x_2

Initialize with $\forall i, \bar{x}_i = 0, \quad \forall k, \bar{v}_k = 0$ and choose $(\bar{y}_1 = 1, \bar{y}_2 = 0)$ **or** $(\bar{y}_1 = 0, \bar{y}_2 = 1)$

$\bar{x}_1 += \bar{v}_1 * 2 * x_1$

$\bar{v}_1 += \bar{v}_3 * v_3$

$\bar{v}_3 += \bar{y}_1 * 1$

$\bar{y}_1$

$\bar{x}_1 += \bar{v}_2 * 3$

$\bar{v}_2 += \bar{v}_4 * 1$

$\bar{v}_4 += \bar{y}_1 * (-1)$

$\bar{y}_2$

$\bar{v}_4 += \bar{y}_2 * x_2$

$\bar{x}_2 += \bar{v}_4 * 1$

$\bar{x}_2 += \bar{y}_2 * v_4$

What about programs? First issue

Overwrites and non-returned variables

Consider the following Fortran code and its naive and **wrong** reverse-mode AD:

```
b = sin(a)
a = x + 1 ! overwrite a
c = cos(a)
! now reverse
```

```
! here a == x + 1
a_adj = a_adj + c_adj * (-sin(a));    c_adj = 0.0
x_adj = x_adj + a_adj * 1;            a_adj = 0.0
a_adj = a_adj + b_adj * cos(a) ! wrong!
b_adj = 0.0
```

Taping

Record and restore the values of overwritten variables to a "tape".
Usually a LIFO stack. But then we can't parallelize. Instead, I'm using (static) arrays.
Other possibility: recompute values, optional checkpointing.

What about programs? First issue

Overwrites, with taping

Recording routine

```
b = sin(a)
tape(i) = a
a = x + 1 ! overwrite a
c = cos(a)
```

Returning routine

```
! here a == x + 1
a_adj = a_adj + c_adj * (-sin(a));     c_adj = 0.0
x_adj = x_adj + a_adj * 1;             a_adj = 0.0
a = tape(i)
a_adj = a_adj + b_adj * cos(a);        b_adj = 0.0
```

Taping results of operations

Can also be used for the results of composed operations. Not implemented yet.
PSyclone: Operation nodes have no datatypes for now, especially shapes (for vectors).

What about programs? Edited July 28 Second issue

Iterative assignments

Consider the following Fortran code and its naive and **wrong** reverse-mode AD:

```
a = 2 * a + x                a_adj = a_adj + a_adj * 2 ! wrong! ā ↛ 3ā
! now reverse                x_adj = x_adj + a_adj * 1 ! wrong too!
```

Solution: deal with the LHS adjoint last, edited July 28

```
a = 2 * a + x                x_adj = x_adj + a_adj * 1 ! correct
! now reverse                a_adj = a_adj * 2 ! correct
```

Reversal schedules 1/2

Nested calls

Suppose we want to differentiate the subroutine foo, which calls bar and qux.

```
subroutine foo(x, y)
   call bar(x, y)
   call qux(x, y)
end subroutine foo
```

Reversal schedules

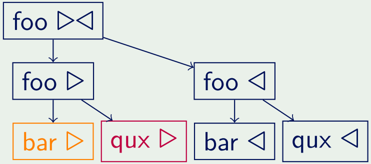
Let us call and denote :

- advancing $\square$ routine the original,
- recording $\triangleright$ routine the one recording values to the tape,
- returning $\triangleleft$ routine the one computing the adjoints,
- reversing $\triangleright\triangleleft$ routine the one combining the two preceding. Call it to differentiate.

Reversal schedules 2/2

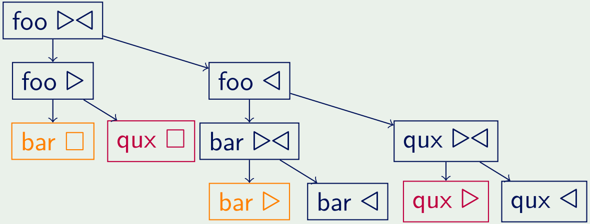
Joint reversal

Larger tape in memory, but computations are only run once.



Split reversal

Smaller tape but computations are repeated.



"Link" reversal

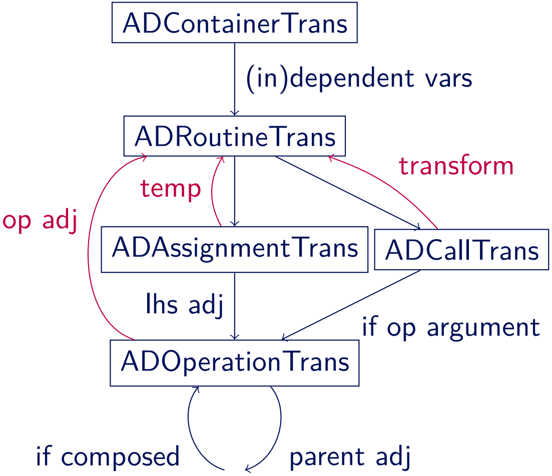
Specify whether to use a split or joint reversal for each (caller → called) pair.

Reverse-mode AD in PSyclone - psyclone.autodiff

For now:

- scalars only,
- recording, returning, reversing and "Jacobian" (scalars $\mapsto$ scalars) routines,
- subroutines only, calls to subroutines,
- joint, split and "link" reversals,
- adjoints are all double precision,
- no activity analysis (dependence DAG),
- no control on what is recorded (TBR),
- quite naive implementation:
  - operations are always recomputed, eg. $v = 1/u$ or $v = \exp(u)$ don't use v,
  - adjoints for every operation, even unary.

Transformations call graph:



Examples of transformed programs 1/2

```
subroutine bar_rec(a, b, value_tape_bar)
    double precision, intent(in) :: a
    double precision, intent(out) :: b
    double precision, dimension(3), intent(out) :: value_tape_bar
    double precision :: c
    c = 2
    value_tape_bar(1) = c
    c = 4
    b = 3
    value_tape_bar(2) = b
    b = b + c * a
    value_tape_bar(3) = c
end subroutine bar_rec
```

```
subroutine bar_rev(a, a_adj, b, b_adj)
    double precision, intent(in) :: a
    double precision, intent(out) :: b
    double precision, intent(inout) :: b_adj
    double precision, intent(inout) :: a_adj
    double precision :: c
    double precision, dimension(3) :: value_tape_bar
    call bar_rec(a, b, value_tape_bar)
    call bar_ret(a, a_adj, b, b_adj, value_tape_bar)
end subroutine bar_rev
```

(Verbose mode)

```
subroutine bar_ret(a, a_adj, b, b_adj, value_tape_bar)
    double precision, intent(in) :: a
    double precision, intent(inout) :: b
    double precision, intent(inout) :: a_adj
    double precision, intent(inout) :: b_adj
    double precision, dimension(3), intent(in) :: value_tape_bar
    double precision :: c
    double precision :: c_adj
    double precision :: temp_b_adj
    double precision :: op_adj
    c_adj = 0.0
    c = value_tape_bar(3)
    b = value_tape_bar(2)
    ! Adjoining b = b + c * a, iterative
    temp_b_adj = b_adj
    b_adj = 0.0
    ! Adjoining b + c * a
    b_adj = b_adj + temp_b_adj * 1
    op_adj = temp_b_adj * 1
    ! Adjoining c * a
    c_adj = c_adj + op_adj * a
    a_adj = a_adj + op_adj * c  ! Finished adjoining b = b + c * a
    ! Adjoining b = 3
    b_adj = 0.0  ! Finished adjoining b = 3
    c = value_tape_bar(1)
    ! Adjoining c = 4
    c_adj = 0.0  ! Finished adjoining c = 4
    ! Adjoining c = 2
    c_adj = 0.0  ! Finished adjoining c = 2
end subroutine bar_ret
```

Examples of transformed programs 2/2

```fortran
subroutine foo_rec(x, w, f, g, value_tape_foo)
    double precision, intent(in) :: x
    double precision, intent(in) :: w
    double precision, intent(out) :: f
    double precision, intent(out) :: g
    double precision, dimension(4), intent(out) :: value_tape_foo
    f = w ** 2
    g = 3 * x
    value_tape_foo(1) = f
    call bar_rec(x + 2 * f, f, value_tape_foo(2:))
end subroutine foo_rec
subroutine foo_ret(x, x_adj, w, w_adj, f, f_adj, g, g_adj, value_tape_foo)
    double precision, intent(in) :: x
    double precision, intent(in) :: w
    double precision, intent(inout) :: f
    double precision, intent(inout) :: g
    double precision, intent(inout) :: x_adj
    double precision, intent(inout) :: w_adj
    double precision, intent(inout) :: f_adj
    double precision, intent(inout) :: g_adj
    double precision, dimension(4), intent(in) :: value_tape_foo
    double precision :: op_adj
    double precision :: op_adj_1
    f = value_tape_foo(1)
    ! Adjoining call bar(x + 2 * f, f)
    op_adj = 0.0
    call bar_ret(x + 2 * f, op_adj, f, f_adj, value_tape_foo(2:))
    ! f is output so overwritten
    f_adj = 0.0
    ! Adjoining x + 2 * f
    x_adj = x_adj + op_adj * 1
    op_adj_1 = op_adj * 1
    ! Adjoining 2 * f
    f_adj = f_adj + op_adj_1 * 2  ! Finished adjoining call bar(x + 2 * f, f)
    ! Adjoining g = 3 * x
    x_adj = x_adj + g_adj * 3
    g_adj = 0.0  ! Finished adjoining g = 3 * x
    ! Adjoining f = w ** 2
    w_adj = w_adj + f_adj * (2 * w ** (2 - 1))
    f_adj = 0.0  ! Finished adjoining f = w ** 2
end subroutine foo_ret
```

```fortran
! Independent variables as columns: ['x', 'w'].
! Dependent variables as rows: ['f', 'g'].
!
! df/dx df/dw
! dg/dx dg/dw
subroutine foo_jacobian(f, g, x, w, J_foo)
    double precision, intent(out) :: f
    double precision, intent(out) :: g
    double precision, intent(in) :: x
    double precision, intent(in) :: w
    double precision, dimension(2,2), intent(out) :: J_foo
    double precision :: f_adj
    double precision :: g_adj
    double precision :: x_adj
    double precision :: w_adj
    f_adj = 1.0
    g_adj = 0.0
    x_adj = 0.0
    w_adj = 0.0
    call foo_rev(x, x_adj, w, w_adj, f, f_adj, g, g_adj)
    J_foo(1,1) = x_adj
    J_foo(2,1) = w_adj
    g_adj = 1.0
    f_adj = 0.0
    x_adj = 0.0
    w_adj = 0.0
    call foo_rev(x, x_adj, w, w_adj, f, f_adj, g, g_adj)
    J_foo(1,2) = x_adj
    J_foo(2,2) = w_adj
end subroutine foo_jacobian
```

Next

Coming soon

- Numerical comparisons with AutoGrad / JAX (Tapenade?)
- Finish writing unitary tests, more general ones

Some day...

- Vectors, matrices, functions
- Activity analysis ($\leftrightarrow$ dependence DAG)
- TBR (To Be Recorded) analysis
- Loops: are run backward
  - except if trivial: record the loop indices, run-time length of the tape
- Control flow: record the boolean values of conditions? Tag the branches?

Questions?
Remarks?
Ideas?