PSyclone autodiff module

Release 0.0.1

Julien Remy

Sep 06, 2023

TABLE OF CONTENTS

CHAPTER

ONE

INTRODUCTION

PSyclone `autodiff` module is PSyclone's **prototype** implementation of source-to-source *automatic differentiation (AD)*. It takes generic Fortran code and applies automatic differentiation in *forward-mode (tangent)* or *reverse-mode (adjoint)*.

It is inspired by Tapenade (see Hascoet and Pascual[1] and[2]), which is also used to perform numerical tests of the transformations, and OpenAD (see Utke *et al.*[3]).

The general approach and transformations rules were adapted from Griewank and Walther[4].

This module was created as a M1 internship project in the AIRSEA team of Inria Grenoble.

[1] Laurent Hascoet and Valérie Pascual. The tapenade automatic differentiation tool: principles, model, and specification. *ACM Transactions on Mathematical Software (TOMS)*, 39(3):1–43, 2013.

[2] Laurent Hascoët and Valérie Pascual. *Tapenade 2.1 user's guide*. PhD thesis, INRIA, 2004.

[3] Jean Utke, Uwe Naumann, Mike Fagan, Nathan Tallent, Michelle Strout, Patrick Heimbach, Chris Hill, and Carl Wunsch. Openad/f: a modular open-source tool for automatic differentiation of fortran codes. *ACM Transactions on Mathematical Software (TOMS)*, 34(4):1–36, 2008.

[4] Andreas Griewank and Andrea Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM, 2008.

CHAPTER

TWO

AUTOMATIC DIFFERENTIATION (AD)

Simply put, the aim of automatic differentiation (AD) is to automatically obtain the derivatives of somes variables output by an existing program with respect to some of its input variables.

It avoids resorting to symbolic differentiation, which is error-prone when done manually and quickly of excessive complexity when applied automatically, or finite differences, which are inexact.

To gain an intuition of the way this is achieved, consider 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 $(+, \times, /, \exp, \text{etc.})$.

Let us denote:

- independent variables: $(x_i)_i, i \in \{1, 2, ..., n\}$,
- dependent variables: $(y_j)_j, j \in \{1, 2, ..., m\}$,
- intermediate values: $(v_k)_k$ which may or not be assigned to variables in the program,
- relation: $i \prec j$ if v_j depends on v_i, *eg.* below $1 \prec 5$.
- predecessors: $u_j := (v_i)_{i \prec j}$ *eg.* below $u_5 = \begin{pmatrix} v_1 \\ v_2 \end{pmatrix}$
- operation: $\varphi_j : u_j \mapsto v_j$ *eg.* below $v_5 = \varphi_5(u_5)$

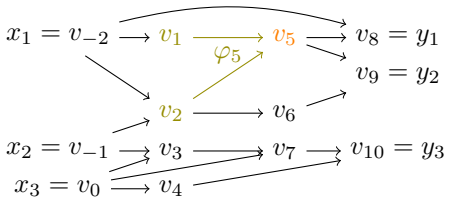

Fig. 1: Example program

Since all programs can be reduced to sequential elemental operations in this fashion, automatic differentiation allows to compute $\dfrac{\partial y_j}{\partial x_i}(x_1, \ldots, x_n)$ by differentiating operations $\varphi_k : u_k \mapsto v_k$ and using the chain rule.

It comes in two main flavors, usually called *forward- or tangent-mode* and *reverse- or adjoint-mode*, which differ in the way substitutions are performed in the chain rule, which partial derivatives are computed as a result and the order in which statements in the original program are differentiated by the AD transformation.

2.1 Forward- or tangent-mode

Using the notations introduced *above*, forward-mode automatic differentiations allows to compute all derivatives w.r.t. a **single** independent variable $d \in (x_i)_i$.

Let us denote the derivatives w.r.t. d as

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

such that the chain rule writes

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

Forward-mode automatic differentation is equivalent to applying substitutions in the order indicated by the arrow in

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

Initializing $\dot{d} = 1$ and $\dot{v}_k = 0, \forall v_k \neq d$,

we obtain in a **single** evaluation $\left(\frac{\partial y_j}{\partial d}(x_1, \dots x_n)\right)_j = J((x_i)_i)(0 \dots \dot{d} \dots 0)^T$

where J is the Jacobian matrix $J = \nabla \begin{pmatrix} y_1(x_1, \dots, x_n) \\ \vdots \\ y_m(x_1, \dots, x_n) \end{pmatrix}$.

2.1.1 Advantages and inconvenients

Forward-mode is easy to implement as derivatives can be computed in the same order of computation as that of the original program.

If there are less independent than dependent variables, its complexity is lower than that of the reverse- or adjoint-mode. But frequently, and maybe even more so in ocean and atmosphere models, the number of inputs greatly exceeds the number of outputs, requiring many repeated evaluations, one for each input or independent variable to differentiate with respect to.

2.2 Reverse- or adjoint-mode

Using the notations introduced *above*, reverse-mode automatic differentiations allows to compute all derivatives of a **single** dependent variable $z \in (y_j)_j$.

Let us denote the adjoints w.r.t. z as

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

such that the chain rule writes

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

where bold font is used to highlight how the value of the adjoint $\bar{v}_i$ depends on **successors** of v_i.

Reverse-mode automatic differentation is equivalent to applying substitutions in the order indicated by the arrow in

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

Initializing $\bar{z} = 1$ and $\bar{v}_k = 0, \forall v_k \neq z$,

we obtain in a **single** evaluation $\left(\frac{\partial z}{\partial x_i}(x_1, \ldots, x_n)\right)_i = \nabla^T z(x_1, \ldots, x_n) = (0 \ldots \bar{z} \ldots 0) J(x_1, \ldots, x_n)$.

2.2.1 Advantages and inconvenients

Reverse-mode is quite a lot more complicated to implement than forward-mode as adjoints need to be computed in the reversed order of computation compared to that of the original program as illustated in the *example below*.

If there are less dependent than independent variables, as is often the case, its complexity is lower than that of the forward- or tagent-mode.

However, when some variables are overwritten in the program, reverse-mode also requires running the original program and recording overwritten values, and eventually some the results of some operations, when they appear in the computations of some adjoints. This add further complications compared to forward-mode and requires using a persistent "tape", which needs to be kept in memory, or recomputing values as many times as they are required.

2.2.2 A simple example in reverse-mode with non-linearities

Let us consider the simple computations displayed below and illustate how to compute the adjoints $\bar{x}_1 = \frac{\partial z}{\partial x_1}$ and $\bar{x}_2 = \frac{\partial z}{\partial x_2}$ for a chosen dependent variable $z \in \{y_1, y_2\}$. Initialize with $\forall i, \bar{x}_i = 0, \forall k, \bar{v}_k = 0$ and choose $(\bar{y}_1 = 1, \bar{y}_2 = 0)$ **or** $(\bar{y}_1 = 0, \bar{y}_2 = 1)$ to obtain the adjoints.

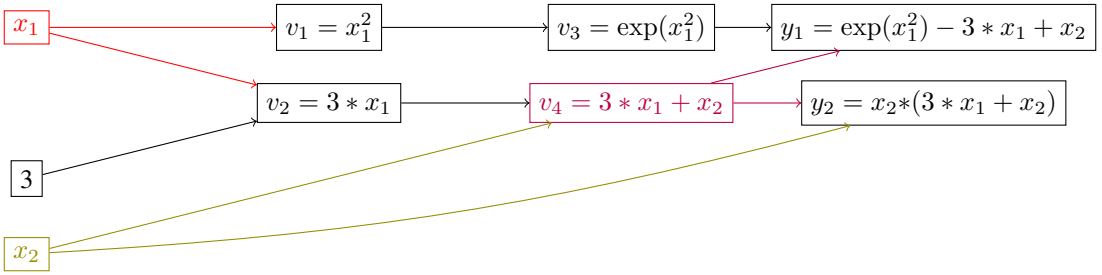

Fig. 2: Simple program example

Notice that the adjoint of variables appearing as operands in the original computations (top) are incremented in the reverse-mode ones (bottom). Moreover, non-linearities in the original occasion the presence of operation results/ non-adjoint variables in the adjoint computations, which could be either recomputed or recorded and restored from a tape.

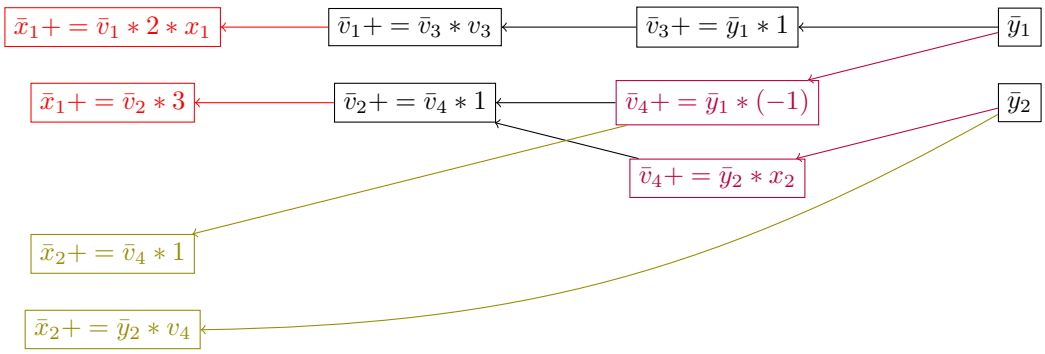

Fig. 3: Reverse-mode example

CHAPTER

THREE

GETTING STARTED

3.1 Download

PSyclone `autodiff` module is hosted on GitHub: https://github.com/JulienRemy/PSyclone/tree/automatic_differentiation.

It is currently an **experimental protoype**.
The latest version is the `automatic_differentiation` branch.

To download it, clone the repository using

```
$ git clone https://github.com/JulienRemy/PSyclone.git
```

3.2 Environment and dependencies

Please follow the instructions regarding environments and dependencies at https://psyclone.readthedocs.io/en/stable/getting_going.html.

This module also requires NumPy, which can be installed using `pip`:

```
$ pip install numpy
```

The tutorials also require Jupyter Notebook, which can be installed using `pip`:

```
$ pip install jupyter
```

3.3 Installing

PSyclone and its `autodiff` module can then be installed using `pip`:

```
$ cd <PSYCLONE_HOME>
$ pip install [--user] .
```

or using `setup.py`:

```
$ cd <PSYCLONE_HOME>
$ python setup.py install
```

3.4 Tutorial

See the src/psyclone/autodiff/tutorials/ directory for a Jupyter Notebook tutorial detailling the use of the module in reverse-mode.

To open it using Jupyter Notebook:

```
$ jupyter-notebook tuto1.ipynb
```

CHAPTER

FOUR

IMPLEMENTATION

This module performs source-to-source automatic differentiation of a target routine, in which dependent variables are differentiated with respect to independent variables.

This is implemented in PSyclone by parsing the source code file containing the target routine, and eventually the routines it calls, transforming it into a PSyIR AST and applying automatic differentiation transformations in either *forward-mode* or *reverse-mode* to the nodes thus obtained. The resulting PSyIR tree is then written to Fortran source code.

4.1 Implemented features

For now, only Fortran subroutines *ie.* neither functions nor programs can be transformed. The implementation only deals with **scalar** variables, which is to say that subroutines containing arrays, either as local variables or as arguments cannot yet be transformed. The statements the target routine (and eventual routines it calls) may contain are :

- assignments (PSyIR `Assignment` nodes),
- calls to subroutines (PSyIR `Call` nodes).

These statements may contain unary and binary linear or non-linear operations (PSyIR `UnaryOperation` and `BinaryOperation` nodes).

An optional `verbose` mode is available, which is especially useful when examining the transformed statements and routines in reverse-mode.

Basic simplification and substitution rules can be applied as an optional postprocessing step to shorten the transformed code and improve its readability.

Reverse-mode transformations store overwritten values using a *"tape"* that is implemented as a static array, rather than a LIFO stack as in many implementations, so that the transformed routines may (someday) be parallelized and/or offloaded to GPU.

Also in reverse-mode, three types of *reversal schedules* are available:

- *split reversal schedules*,
- *joint reversal schedules*,
- *"link" reversal schedules* specifying strong or weak links for all calling-called pairs of routines.

4.2 Missing features

What has **not** been implemented includes:

- functions and programs,
- differentiating called routines that are not found in the same file (or `Container` node) as the target routine,
- nary operations,
- loops,
- control flow,
- array variables and arguments,
- activity analysis (dependence DAG),
- to-be-recorded (TBR) analysis,
- taping operations results to reduce the computational complexity of the adjoint,
- and much more.

CHAPTER

FIVE

REVERSE-MODE AD (ADJOINT)

The adjoint of the Fortran source code is constructed using a source-to-source and line-by-line approach, transforming the *target (to be transformed) routine* into three different routines, one of which computes the adjoints of the variables by reversing the order of computation of the target routine.

This is implemented in PSyclone by parsing the source code file containing the target routine, and eventually the routines it calls, transforming it into a PSyIR AST and applying *reverse-mode automatic differentiation transformations* to the nodes thus obtained. The resulting PSyIR tree is then written to Fortran source code.

5.1 Reverse-mode transformations

Several transformations, to be applied on PSyIR nodes, have been implemented. In reverse-mode, all of them follow the naming convention `ADReverse[PSyIRNodeSubclass]Trans`. The one of most interest for the user is the *ADReverseContainerTrans* class and its `apply` method.

After parsing the Fortran code file containing the target routine, an `ADReverseContainerTrans` instance should be applied to it to perform automatic differentiation. The `ADReverseContainerTrans.apply` method in turn applies an `ADReverseRoutineTrans` to the target routine, which goes line-by-line through the statements found in the `Routine` node, applying `ADReverse[PSyIRNodeSubclass]Trans` to the statements, etc.

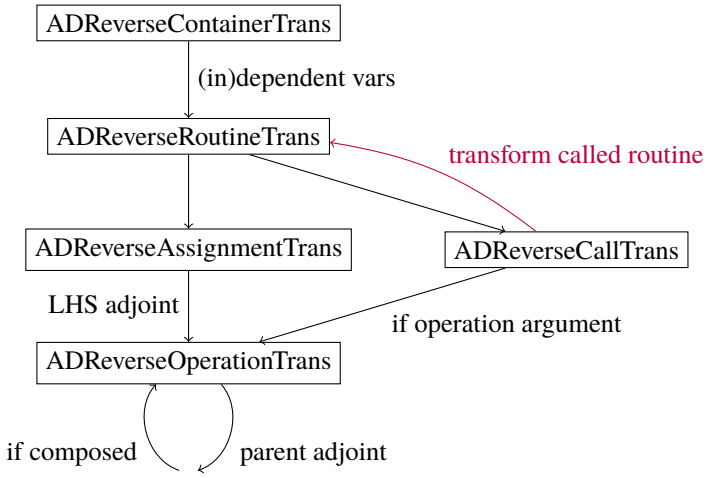

Fig. 1: Reverse-mode AD transformation call graph

5.1.1 Container transformation

class psyclone.autodiff.transformations.**ADReverseContainerTrans**

A class for automatic differentation transformation of Container nodes in reverse-mode. This is the transformation to apply on the PSyIR AST generated from a source.

apply(*container*, *routine_name*, *dependent_vars*, *independent_vars*, *reversal_schedule*, *options=None*)

Applies the transformation, returning a new container with routine definitions for both motions using the reverse-mode of automatic differentiation.

Options:
- bool 'jacobian': whether to generate the Jacobian routine. Defaults to False.
- bool 'verbose' : toggles explanatory comments. Defaults to False.
- bool 'simplify': True to apply simplifications after applying AD transformations. Defaults to True.
- int 'simplify_n_times': number of time to apply simplification rules to BinaryOperation nodes. Defaults to 5.
- bool 'inline_operation_adjoints': True to inline all possible operation adjoints definitions. Defaults to True.

Parameters

- **container** (psyclone.psyir.nodes.Container) – Container Node to the transformed.
- **routine_name** (*Str*) – name of the Routine to be transformed.
- **dependent_vars** (*List[Str]*) – list of dependent variables names to be differentiated.
- **independent_vars** (*List[str]*) – list of independent variables names to differentiate with respect to.
- **reversal_schedule** (psyclone.autodiff.ADReversalSchedule) – reversal schedule for routined called inside the one to transform (and inside them, etc.).
- **options** (*Optional[Dict[Str, Any]]*) – a dictionary with options for transformations, defaults to None.

Returns
a copied and modified container with all necessary Routine definitions.

Return type
psyclone.psyir.nodes.Container

As can be seen, the required arguments include the PSyIR [File]Container node obtained by parsing and transforming the source code, the **names** of the *target routine*, *dependent variables* (to be differentiated) and *independent variables* (to differentiate with respect to), as well as the *reversal schedule* and eventual transformation options.

The transformation returns a PSyIR Container node containing four Routine definitions for:

- the *advancing* (original) motion,
- the *recording* motion, which records overwritten values to the tape,
- the *returning* motion, which recovers values from the tape and computes the adjoints of the independent variables,
- the *reversing* motion, which combines the two precedent recording and returning motions and is the one to call in order to differentiate.

If some other routine is called by the target one, the returned `Container` node also contains four definitions for its different motions.

Target routine

The target routine is the Fortran routine in which to differentiate the *dependent variables* with respect to the *independent variables*. The routines it may call will also be differentiated iff they can be found in the `[File]Container` being transformed.

Dependent variables

The dependent variables are those to differentiate. Their intent in the target routine must be compatible with their values being returned, *ie.* they cannot be intent(in) arguments of the target routine.

Independent variables

The independent variables are those to differentiate with respect to. Their intent in the target routine must be compatible with their values being provided as arguments, *ie.* they cannot be intent(out) arguments of the target routine.

Reversal schedules

Reversal schedules (see Griewank and Walther[1] chapter 12.2, p.265) specify the way a transformed routine may call other transformed routines. They are implemented as 3 subclasses of `ADReversalSchedule`.

As an example let us consider the target routine `foo` calling subroutines `bar` and `qux`.

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

And let us denote:

- advancing □ routine the original,
- recording ▷ routine the one recording values to the tape,
- returning ◁ routine the one computing the adjoints,
- reversing ▷◁ routine the one combining the two preceding. Call it to differentiate.

Split reversal schedule

In split reversal, children (called) routines follow the recording or returning motion of their parent (calling) routine.

By doing so, the computational complexity is kept as low as possible but values stored to the tape in the recording motion of the called routines need to be kept until they are called in returning motion, thus using a possibly large amount of memory.

[1] Andreas Griewank and Andrea Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM, 2008.

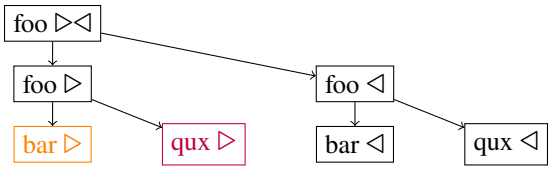

Fig. 2: Split reversal schedule

Joint reversal schedule

In joint reversal, all children (called) routines advance without recording when their parent (calling) routine is recording and reverse (record then immendiatly return) when their parent routine is returning.

On the one hand, this reversal schedule uses a smaller tape overall, as the values used in adjoining the called routines do no need to be stored longer than for them to be reversed. On the other hand, called subroutines computations are repeated, with increases the computational complexity of the adjoint program.

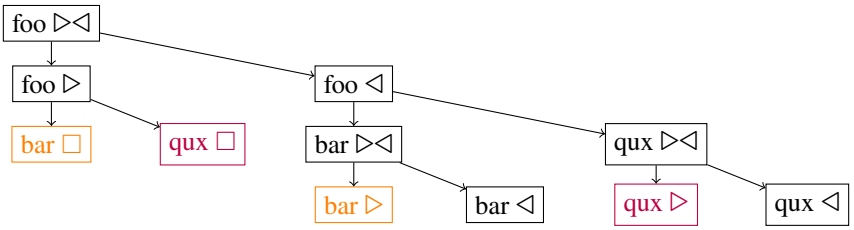

Fig. 3: Joint reversal schedule

"Link" reversal schedule

A third possibility is to specify *strong* or *weak* links for each caller-called pair of routines, where strong links behave as in split reversal and weak links as in joint reversal.

Below is an illustration of our toy example with `foo-bar` a strong link and `foo-qux` a weak link.

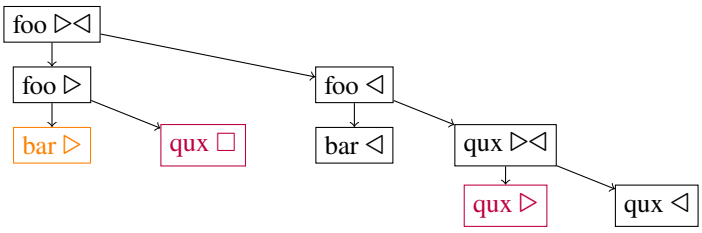

Fig. 4: Link reversal schedule with foo-bar a strong link and foo-qux a weak link

5.2 Value tape

Prevalues of overwritten variables are recorded and restored from a *value tape*, implemented as a static array. The transformations themselves employ an `ADValueTape` to generate recording and restoring statements to and from the value tape array.

class psyclone.autodiff.tapes.**ADValueTape**(*name*, *datatype*)

A class for recording and recovering function values in reverse-mode automatic differentiation. The **prevalues** of references are recorded. Based on static arrays storing a single type of data rather than a LIFO stack. Provides methods to create the PSyIR assignments for recording and restoring operations.

Parameters

- **name** – name of the value_tape (after a prefix).
- **datatype** (Union[psyclone.psyir.symbols.ScalarType, psyclone.psyir.symbols.ArrayType]) – datatype of the elements of the value_tape.

Raises

- **TypeError** – if name is of the wrong type.
- **TypeError** – if datatype is of the wrong type.
- **NotImplementedError** – if datatype is not of type 'ScalarType'.

record(*reference*)

Add the reference as last element of the value_tape and return the Assignment node to record the prevalue to the tape.

Parameters

reference (psyclone.psyir.nodes.Reference) – reference whose prevalue should be recorded.

Raises

- **TypeError** – if reference is of the wrong type.
- **TypeError** – if the intrinsic of reference's datatype is not the same as the intrinsic of the value_tape's elements datatype.
- **NotImplementedError** – if the reference's datatype is ArrayType.

Returns

an Assignment node for recording the prevalue of the reference as last element of the value_tape.

Return type

psyclone.psyir.nodes.Assignment

restore(*reference*)

Restore the last element of the value_tape if it is the symbol argument and return the Assignment node to restore the prevalue to the variable.

Parameters

reference (psyclone.psyir.symbols.DataSymbol) – reference whose prevalue should be restored from the value_tape.

Raises

TypeError – if reference is of the wrong type.

Returns
an Assignment node for restoring the prevalue of the reference from the last element of the value_tape.

Return type
`psyclone.psyir.nodes.Assignment`

5.3 Generating adjoints

The transformations applied to generate adjoints are detailled below. They mostly follow the guidelines found in Griewank and Walther[Page 13, 1] chapter 6.2, pp.125-126.

Internally, the transformations used are `ADReverseAssignmentTrans`, `ADReverseOperationTrans` and `ADReverseCallTrans`, depending on the PSyIR node being transformed. These all return two separate lists of PSyIR statements, used respectively in extending the recording and returning routines being generated.

5.3.1 Adjoints of operations

Adjoints of unary operations

Advancing motion	Recording motion	Returning motion

`f = +x`	`value_tape(i) = f`
	`f = +x`
		`f = value_tape(i)`
		`x_adj = x_adj + f_adj`
		`f_adj = 0.0`

`f = -x`	`value_tape(i) = f`
	`f = -x`
		`f = value_tape(i)`
		`x_adj = x_adj - f_adj`
		`f_adj = 0.0`

`f = SQRT(x)`	`value_tape(i) = f`
	`f = SQRT(x)`
		`f = value_tape(i)`
		`x_adj = x_adj + f_adj / (2 * SQRT(x))`
		`f_adj = 0.0`

`f = EXP(x)`	`value_tape(i) = f`
	`f = EXP(x)`
		`f = value_tape(i)`
		`x_adj = x_adj + f_adj * EXP(x)`
		`f_adj = 0.0`

`f = LOG(x)`	`value_tape(i) = f`
	`f = LOG(x)`
		`f = value_tape(i)`
		`x_adj = x_adj + f_adj / x`
		`f_adj = 0.0`

`f = LOG10(x)`	`value_tape(i) = f`
	`f = LOG10(x)`
		`f = value_tape(i)`
		`x_adj = x_adj + f_adj / (x * LOG(10.0))`
		`f_adj = 0.0`

`f = COS(x)`	`value_tape(i) = f`
	`f = COS(x)`
		`f = value_tape(i)`
		`x_adj = x_adj - f_adj * SIN(x)`
		`f_adj = 0.0`

`f = SIN(x)`	`value_tape(i) = f`
	`f = SIN(x)`
		`f = value_tape(i)`
		`x_adj = x_adj + f_adj * COS(x)`
		`f_adj = 0.0`

`f = TAN(x)`	`value_tape(i) = f`
	`f = TAN(x)`
		`f = value_tape(i)`
		`x_adj = x_adj + f_adj * (1.0 + TAN(x) ** 2)`
		`f_adj = 0.0`

`f = ACOS(x)`	`value_tape(i) = f`
	`f = ACOS(x)`
		`f = value_tape(i)`
		`x_adj = x_adj - f_adj / SQRT(1.0 - x ** 2)`
		`f_adj = 0.0`

`f = ASIN(x)`	`value_tape(i) = f`
	`f = ASIN(x)`
		`f = value_tape(i)`
		`x_adj = x_adj + f_adj / SQRT(1.0 - x ** 2)`
		`f_adj = 0.0`

`f = ATAN(x)`	`value_tape(i) = f`
	`f = ATAN(x)`
		`f = value_tape(i)`
		`x_adj = x_adj + f_adj / (1.0 + x ** 2)`
		`f_adj = 0.0`

`f = ABS(x)`	`value_tape(i) = f`
	`f = ABS(x)`
		`f = value_tape(i)`
		`x_adj = x_adj + f_adj * (x / ABS(x))`
		`f_adj = 0.0`

Note: some of these adjoints computations, explicitly those for `SQRT`, `EXP`, `TAN` and `ABS`, could reuse the (post)value of `f` before restoring its prevalue from the value tape rather than recompute it (see Griewank and Walther[Page 13, 1] table 4.8, p.68). This is not implemented yet.

Adjoints of binary operations

Advancing motion	Recording motion	Returning motion

`f = x + y`	`value_tape(i) = f`
	`f = x + y`
		`f = value_tape(i)`
		`x_adj = x_adj + f_adj`
		`y_adj = y_adj + f_adj`
		`f_adj = 0.0`

`f = x - y`	`value_tape(i) = f`
	`f = x - y`
		`f = value_tape(i)`
		`x_adj = x_adj + f_adj`
		`y_adj = y_adj - f_adj`
		`f_adj = 0.0`

`f = x * y`	`value_tape(i) = f`
	`f = x * y`
		`f = value_tape(i)`
		`x_adj = x_adj + f_adj * y`
		`y_adj = y_adj + f_adj * x`
		`f_adj = 0.0`

`f = x / y`	`value_tape(i) = f`
	`f = x / y`
		`f = value_tape(i)`
		`x_adj = x_adj + f_adj / y`
		`y_adj = y_adj - f_adj * x / y ** 2`
		`f_adj = 0.0`

`f = x ** y`	`value_tape(i) = f`
	`f = x ** y`
		`f = value_tape(i)`
		`x_adj = x_adj + f_adj * y * x ** (y - 1)`
		`y_adj = y_adj + f_adj * x ** y * LOG(x)`
		`f_adj = 0.0`

Note: some of these adjoints computations, explicitly those for `/` and `**` could reuse the (post)value of `f` before restoring its prevalue from the value tape rather than recompute it (see Griewank and Walther[Page 13, 1] table 4.8, p.68). This is not implemented yet.

The cases detailled above are the simpler ones, of assigning the result of an operation to a variable.

When composed operations are present, an adjoint variable is declared for the adjoint of the operation itself and used to increment the adjoints of its operands.

The transformation option `inline_operation_adjoints` allows the user to choose whether these operation adjoints should be substituted in further computations of adjoints as a postprocessing step, iff they only appear once on the RHS of an assignment.

As an example, consider the following computation involving composed operations and the associated adjoints computations, without and with substitution. *Note*: taping assignments are omitted below.

Composed operation	Adjoints, without substitution	Adjoints, with substitution

`f = EXP(x) + z`	`op_adj = f_adj`
	`z_adj = z_adj + f_adj`
	`x_adj = x_adj + op_adj * EXP(x)`
	`f_adj = 0.0`
		`z_adj = z_adj + f_adj`
		`x_adj = x_adj + f_adj * EXP(x)`
		`f_adj = 0.0`

Adjoints of iterative assignments

In the case of iterative assignments *ie.* where the LHS variable of the assignment is also present on the RHS, additional care must be taken to avoid incorrect computations of the LHS adjoint by assigning to it last rather than incrementing its value as in the general case detailled above (see Griewank and Walther[Page 13, 1] chapter 5.1, p.93).

As an example consider the following adjoint:

Advancing motion	Recording motion	Returning motion

`f = 2 * f + x`	`value_tape(i) = f`
	`f = 2 * f + x`
		`f = value_tape(i)`
		`x_adj = x_adj + f_adj`
		`f_adj = f_adj * 2`

5.3.2 Adjoints of calls to subroutines

The adjoints of calls to subroutines depend on the *reversal schedule* that is used.

Whether the prevalues of the arguments are recorded and restored from the tape depend on their intent in the called subroutine, which determines whether their value might be overwritten by it or not.

Operations as subroutine call arguments are also transformed.

Split reversal schedule

Advancing motion	Recording motion	Returning motion
`call func(x, y)`	`[value_tape(i) = x]`	`[x = value_tape(i)]`
	`[value_tape(i + 1) = y]`	`[y = value_tape(i + 1)]`
	`call func_recording(x,y)`	`call func_returning(x, x_adj, y, y_adj)`

Joint reversal schedule

Advancing motion	Recording motion	Returning motion
`call func(x, y)`	`[value_tape(i) = x]`	`[x = value_tape(i)]`
	`[value_tape(i + 1) = y]`	`[y = value_tape(i + 1)]`
	`call func(x,y)`	`call func_reversing(x, x_adj, y, y_adj)`

CHAPTER

SIX

FORWARD-MODE AD (TANGENT)

The derivative of the Fortran source code is constructed using a source-to-source and line-by-line approach, transforming the target routine into a tangent routine, which computes the derivatives of the dependent variables with respect to the independent variables.

This is implemented in PSyclone by parsing the source code file containing the target routine, and eventually the routines it calls, transforming it into a PSyIR AST and applying *forward-mode automatic differentiation transformations* to the nodes thus obtained. The resulting PSyIR tree is then written to Fortran source code.

6.1 Forward-mode transformations

All forward-mode AD transformations, to be applied to PSyIR nodes, follow the naming convention `ADForward[PSyIRNodeSubclass]Trans`. The one users should use is *ADForwardContainerTrans* class and its `apply` method.

6.1.1 Container transformation

After parsing the Fortran code file containing the target routine, an `ADForwardContainerTrans` instance should be applied to it to perform automatic differentiation. This in turn applies an `ADForwardRoutineTrans` to the target routine, which goes line-by-line through the statements found in the `Routine` node, applying `ADForward[PSyIRNodeSubclass]Trans` to the statements, etc.

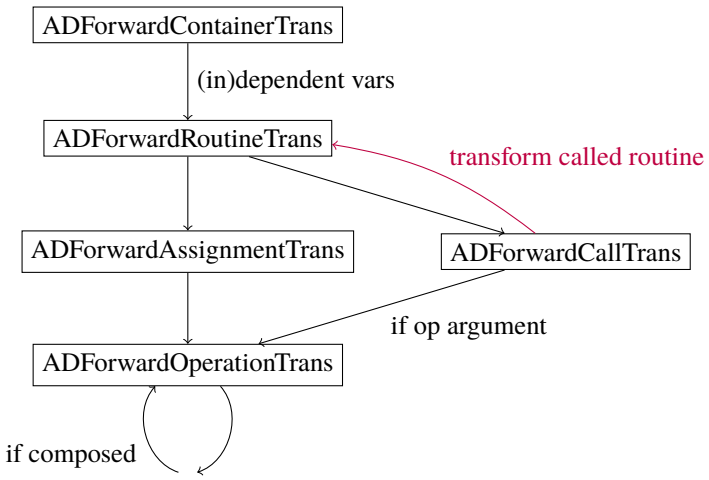

Fig. 1: Forward-mode AD transformation call graph

class psyclone.autodiff.transformations.**ADForwardContainerTrans**

A class for automatic differentation transformation of Container nodes in foward-mode. This is the transformation to apply on the PSyIR AST generated from a source.

apply(*container*, *routine_name*, *dependent_vars*, *independent_vars*, *options=None*)

Applies the transformation, returning a new container with routine definitions using the forward-mode of automatic differentiation.

Options:

- bool 'jacobian': whether to generate the Jacobian routine. Defaults to False.

- bool 'verbose' : toggles explanatory comments. Defaults to False.

- bool 'simplify': True to apply simplifications after applying AD transformations. Defaults to True.

- int 'simplify_n_times': number of time to apply simplification rules to BinaryOperation nodes. Defaults to 5.

Parameters

- **container** (psyclone.psyir.nodes.Container) – Container Node to the transformed.
- **routine_name** (*Str*) – name of the Routine to be transformed.
- **dependent_vars** (*List[Str]*) – list of dependent variables names to be differentiated.
- **independent_vars** (*List[Str]*) – list of independent variables names to differentiate with respect to.
- **options** (*Optional[Dict[Str, Any]]*) – a dictionary with options for transformations, defaults to None.

Returns

a copied and modified container with all necessary Routine definitions.

Return type

psyclone.psyir.nodes.Container

For descriptions of the arguments, see the relevant sections of *ADReverseContainerTrans*: *target routine*, *dependent variables* (to be differentiated) and *independent variables* (to differentiate with respect to).

The transformation returns a PSyIR `Container` node containing two `Routine` definitions for:

- the original target routine,
- the transformed tangent routine, which computes of the required derivatives.

If some other routine is called by the target one, the returned `Container` node also contains the original and tangent definitions for it.

6.2 Generating derivatives

The transformations applied to generate derivatives are detailled below. They mostly follow the guidelines found in Griewank and Walther[1] chapter 6.2, p.123.

6.2.1 Derivatives of unary operations

Original	Transformed
`f = +x`	`f_d = x_d` `f = +x`
`f = -x`	`f_d = -x_d` `f = -x`
`f = SQRT(x)`	`f_d = x_d / (2 * SQRT(x))` `f = SQRT(x)`
`f = EXP(x)`	`f_d = EXP(x) * x_d` `f = EXP(x)`
`f = LOG(x)`	`f_d = x_d / x` `f = LOG(x)`
`f = LOG10(x)`	`f_d = x_d / (x * LOG(10.0))` `f = LOG10(x)`
`f = COS(x)`	`f_d = (-SIN(x)) * x_d` `f = COS(x)`
`f = SIN(x)`	`f_d = COS(x) * x_d` `f = SIN(x)`
`f = TAN(x)`	`f_d = (1.0 + TAN(x) ** 2) * x_d` `f = TAN(x)`
`f = ACOS(x)`	`f_d = -x_d / SQRT(1.0 - x ** 2)` `f = ACOS(x)`
`f = ASIN(x)`	`f_d = x_d / SQRT(1.0 - x ** 2)` `f = ASIN(x)`
`f = ATAN(x)`	`f_d = x_d / (1.0 + x ** 2)` `f = ATAN(x)`
`f = ABS(x)`	`f_d = x / ABS(x) * x_d` `f = ABS(x)`

[1] Andreas Griewank and Andrea Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM, 2008.

6.2.2 Derivatives of binary operations

Advancing motion	Recording motion
`f = x + y`	`f_d = x_d + y_d` `f = x + y`
`f = x - y`	`f_d = x_d - y_d` `f = x - y`
`f = x * y`	`f_d = x_d * y + y_d * x` `f = x * y`
`f = x / y`	`f_d = (x_d - y_d * x / y) / y` `f = x / y`
`f = x ** y`	`f_d = x_d * (y * x ** (y - 1)) + y_d * (x ** y * LOG(x))` `f = x ** y`

6.2.3 Derivatives of calls to subroutines

Original	Transformed
`call func(x, y)`	`call func_tangent(x, x_d, y, y_d)`

INDEX

A

R