An AD Approach using F90 and Tapenade

D. P. Jones
Queen Mary, University of London

November, 2009

Introduction

- At Queen Mary Univ., AD (adj) is being applied to two CFD codes:
- The first is an in-house code for simulating compressible flow,
- and the second code is commercial, from ESI, for solving multi-physics problems.
- Both codes are written in F90, using a large proportion of the language features.
- So far, AD has been successfully applied to the in-house code.
- Work is underway on the commercial code.

Creating the Adjoint Program

- The entire processes of creating the differentiated source code is controlled by the Makefile.
- The process breaks up into the following steps:
  1. Modify: prepare the source code to be read by Tapenade.
  2. Compile and Link*: test that the code being passed to Tapenade is complete and syntactically correct.
  3. *Differentiate*: pass the source code to Tapenade with directives to create the differentiated source code.
  4. Modify AD: perform any necessary changes to the AD source code.
  5. *Build*: create executable from the primal and differentiated code.

Prerequisites

- The primal must be written within the subset of the language supported by the AD tool.
- Procedures containing dynamic allocation ought not be differentiated.
- Constructors/Destructors will be required for handling dynamic variables used for obtaining the adjoint.
- All dependencies ought to be available (as source) to the AD tool of a procedure to be differentiated.
- The primal may require modifying before differentiating.

A Note on Arguments

- Generally, the best way to pass arguments is explicitly, via the argument list.
- By declaring variables explicitly, the `INTENT()` attribute can be used.
- This can be tedious when many variables are required, though for arrays their sizes can be omitted.
- If there are many variables which belong to a structure, the structure may be passed, though a dependency is introduced via a `USE` statement.
- Parameters are commonly passed implicitly since their intent is already clear.

Arguments Example

```
subroutine calc_force(geom,prop,obj)

  ! inactive variables:
  use param_m, only: wp, g

  ! active variables: independent & dependent
  use struc_m, only: geom_t, prop_t, obj_t
  type(geom_t)::geom
  type(prop_t)::prop
  type(obj_t)::obj

  ! active variables: intermediate
  real(wp)::mass

  mass = prop%den * geom%vol
  obj%force = mass * g
end subroutine
```

primal

```
SUBROUTINE CALC_FORCE_D(geom,geomd, prop, obj,objd)
  USE PARAM_M_D, ONLY : wp, g

  USE STRUC_M_D, ONLY : geom_t, prop_t, obj_t, &
    geom_t_d, prop_t, obj_t_d
  TYPE(GEOM_T) :: geom
  TYPE(GEOM_T_D) :: geomd
  TYPE(PROP_T) :: prop
  TYPE(OBJ_T) :: obj
  TYPE(OBJ_T_D) :: objd

  REAL(wp) :: mass
  REAL(wp) :: massd

  massd = prop%den*geomd%vol
  mass = prop%den*geom%vol
  objd%force = g*massd
  obj%force = mass*g
END SUBROUTINE
```

differential

Modification Scripts

- At present, shell scripting is used to automate the modification process.
- Syntax is examined and modified via Grep/Sed utilities and via C preprocessor directives.
- This approach is inadequate: transformation is slow and the script is difficult to program.
- Tokenising/parsing/reconstruction utilities would help the writing of such modification scripts.

Modification Example From ESI

- The source code is divided into modules declaring data and subroutines.
- Active variables are obtained via a look-up function rather than being passed explicitly.
- Tapenade is unable to differentiate such code so modifications must be performed.
- The present way of dealing with the problem is to place the active variables in the argument list and comment any associated pointer functions.

Initial and Modified Source

The initial code is shown along with its modified version:

```
subroutine sol_scalar()
  use activevars_m, only: lookup
  use scalar_m, only: iphi
  real,dimension(:),pointer::phi

  phi => lookup(iphi)
  phi = ...
  nullify(phi)
end subroutine
```

initial source

```
subroutine sol_scalar(phi)
  use activevars_m, only: lookup
  use scalar_m, only: iphi
  real,dimension(:),pointer::phi

  ! phi => lookup(iphi)
  phi = ...
  ! nullify(phi)
end subroutine
```

modified source

Differentiation and Final Modification

The modified code is differentiated, then the differentiated code is further modified:

```
subroutine sol_scalar_d(phi, phid)
  use activevars_m_d, only: lookup
  use scalar_m_d, only: iphi
  real,dimension(:),pointer::phid
  real,dimension(:),pointer::phi

  ! phi => lookup(iphi)
  phid = ...
  phi = ...
  ! nullify(phi)
end subroutine
```

differentiated source

```
subroutine sol_scalar_d()
  use activevars_m_d, only: lookup
  use scalar_m_d, only: iphi, phid
  real,dimension(:),pointer::phid
  real,dimension(:),pointer::phi

  phid => lookup(iphid)
  phi => lookup(iphi)
  phid = ...
  phi = ...
  nullify(phid)
  nullify(phi)
end subroutine
```

modified differentiated source

Compiling and Linking of Primal Code

- This step is not necessary, though during development is helpful. It serves two purposes:
  1. Compilation is performed to test whether the modifications have produced valid code and
  2. to check that all dependencies are accounted for in the source code, the objects are linked to a driver program to create an executable.

Arguments to Differentiate

- For larger codes, usually only a section of (say) a solver requires differentiating, not the entire solver algorithm. This requires knowledge of what to differentiate with respect to.
- At the top level the inputs and outputs are usually obvious: an array of design variables and a scalar objective. However at lower levels, the inputs and outputs to differentiate is not obvious.
- A useful test then is to differentiate the top level, even though it is not required, and examine how the lower level routines have been handled.

Argument Intent

- For a given routine; suppose it is understood which differential is required and thus which arguments to deal with, care still must be taken in defining the argument list.
- Each argument in the primal must be checked as to whether it is an input, output or both. From this knowledge the differentiation command is set.
- Never specify an argument as both input and output to Tapenade if in the code it is not. Handle summations/duplications externally.

Argument Summation Example

```
call init_flow(alp[in], q[out])
do iter = 1, n
  q0 = q
  call calc_res(q0[in], r[out])
  call upd_vars(q0[in], r[in], q[out])
end do
call calc_lift(q[in], cl[out])
```

primal

```
call calc_lift_b(qb[out], clb[in])
do iter = 1, n
  call upd_vars_b(q0b[out], rb[out], qb[in])
  qb = q0b
  call calc_res_b(q0b[out], rb[in])
  qb = qb + q0b
end do
call init_flow_b(alpb[out], qb[in])
```

adjoint

```
-head 'calc_res(q0)\(r) upd_vars(q0 r)\(q)'
```

Fortran

- To make use of the features of F90, such as argument checking, modulation, use of structures, etc, Chapman recommends all procedures are programmed within the scope of modules.
- To arrange a code in a modular way, dependency of data and procedures must be ordered, avoiding any circular references.
- Since many of the features in F90 are supported by Tapenade, these should be exploited.
- Make use of intrinsic functions and vector operators to make code easier to read and optimise.
- Certain constructs must be avoided in any code passed to Tapenade, such as open ended DO loops.

Code Structure

```
module base_m

  integer,parameter::wp=8
  real(wp),parameter::pi=
  real(wp),parameter::small=

contains

  subroutine cross_prod(x,y)
    real(wp)::x(3),y(3)
    ...
  end subroutine
end module
```

base_m.f90

```
module solver_m
  use base_m

  type::eqn_t
    real(wp),dimension(:),<A>::ap
    real(wp),dimension(:),<A>::su
    real(wp),dimension(:),<A>::phi
  end type
  type(eqn_t)::u,v,w

contains

  subroutine solve_eqn()
    ...
    call cross_prod(u,v)
    ...
  end subroutine
end module
```

solver_m.f90

```
program main
  use solver_m

  call alloc_eqn(u,v,w)
  call primal()

contains

  subroutine primal()
    call init_eqn()
    do i=1,n_iter
      call solve_eqn()
    end do
    call output_eqn()
  end subroutine
end program
```

main.f90

Differentiated Code Structure

```
module base_m_b

  integer,parameter::wp=8
  real(wp),parameter::pi=
  real(wp),parameter::small=

contains

  subroutine cross_prod(x,y)
    real(wp)::x(3),y(3)
    ...
  end subroutine

  subroutine cross_prod_b(x,xb,y,yb)
    real(wp)::x(3),y(3)
    real(wp)::xb(3),yb(3)
    ...
  end subroutine
end module
```

base_m_b.f90

```
module solver_m_b
  use base_m_b

  type::eqn_t
    real(wp),dimension(:),<A>::ap
    real(wp),dimension(:),<A>::su
    real(wp),dimension(:),<A>::phi
  end type
  type(eqn_t)::u,v,w

  type::eqn_t_b
    real(wp),dimension(:),<A>::su
    real(wp),dimension(:),<A>::phi
  end type
  type(eqn_t_b)::ub,vb,wb

contains

  subroutine solve_eqn()
    ...
    call cross_prod(u,v)
  end subroutine

  subroutine solve_eqn_b()
    ...
    call cross_prod_b(u,ub,u,vb)
  end subroutine
end module
```

solver_m_b.f90

```
program main
  use solver_m_b

  call alloc_eqn(u,v,w)
  call primal()

  call alloc_eqnb(ub,vb,wb)
  call adjoint()

contains

  subroutine primal()
    ...
    call solve_eqn()
  end subroutine

  subroutine adjoint()
    ...
    call solve_eqn_b()
  end subroutine
end program
```

main.f90

Tools

C Preprocessor

- In subroutines, macros can be used to define which lines are necessary to create a primal, and which to use for differentiating.
- Through the Makefile the macros are set and are used to define the build procedure.
- To use cpp on an F90 file:

```
cpp -DMACRO file_cpp.f90 |
  sed -e '/^#/d' -e '/^$/d' > file.f90
```

- Multi-pass preprocessing may be useful.

Tapenade: File Handling

- To differentiate a procedure, it plus all its dependencies must be passed to the differentiation tool.
- For typical codes, knowing (or finding) the dependencies can be slightly tedious, so the use of modules greatly ease the problem; simply look to see which modules are used.
- Upon differentiating, the output code will contain the original plus any differentiated code within new modules.
- File structure will remain intact if one module-per-file arrangement is used.
- The new modules have the same name as the original plus a suffix indicating the type of differentiation.

Tapenade: Usage

- Tapenade has the feature which enable many routines to be differentiated in one call and is of the from `tapenade -head 'cross_prod(x,y)\(z) solve_eqn(u)\(u)'`
- A limitation of this method is that there is no way of multiply differentiating a procedure.
- The possibility of defining the resultant name of the differentiated routine would be helpful; i.e. `-head 'resid(q)\(r)>resid1 resid(x n)\(r)>resid2'`
- Also, the possibility of stating the mode of differentiation for each routine within the string would make the build process neat.

Makefile

- Typical makefiles do not depend on the order the object list, since any undefined calls are resolved at the link stage.
- With modules, the object list must be ordered, i.e.
  `OBJ = base_m.o solver_m.o main.o`
- Often the files being passed to Tapenade are in several directories. To simplify defining the call list, `VPATH` is set beforehand and the automatic dependency variable `$^` in the Makefile is used.