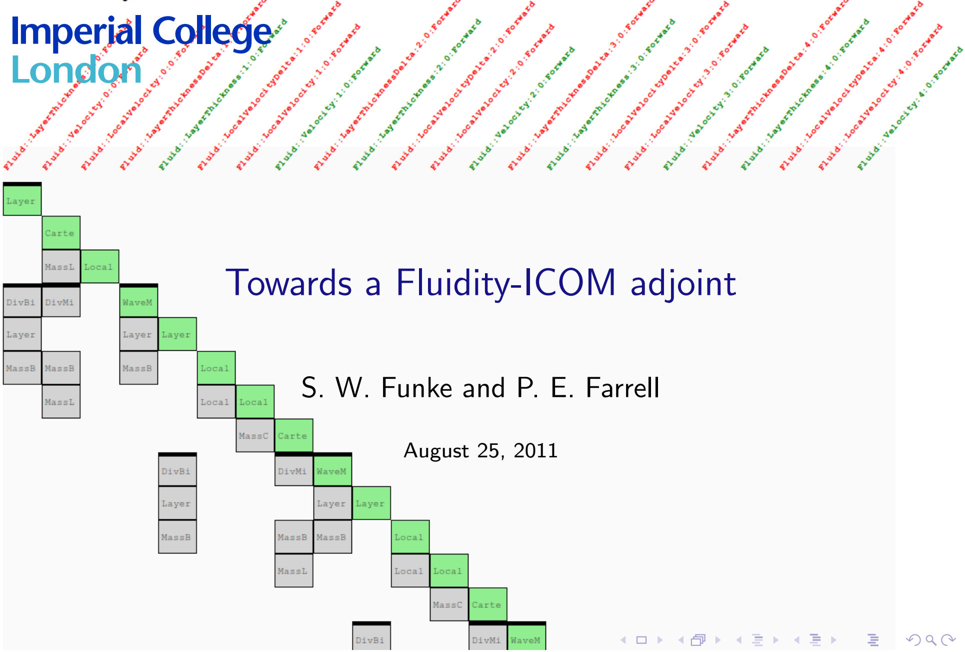
Imperial College
London


Fluid::LayerThickness:0:0:Forward
Fluid::Velocity:0:0:Forward
Fluid::LocalVelocity:0:0:Forward
Fluid::LayerThicknessDelta:1:0:Forward
Fluid::LayerThickness:1:0:Forward
Fluid::LocalVelocityDelta:1:0:Forward
Fluid::LocalVelocity:1:0:Forward
Fluid::Velocity:1:0:Forward
Fluid::LayerThicknessDelta:2:0:Forward
Fluid::LayerThickness:2:0:Forward
Fluid::LocalVelocityDelta:2:0:Forward
Fluid::LocalVelocity:2:0:Forward
Fluid::Velocity:2:0:Forward
Fluid::LayerThicknessDelta:3:0:Forward
Fluid::LayerThickness:3:0:Forward
Fluid::LocalVelocityDelta:3:0:Forward
Fluid::LocalVelocity:3:0:Forward
Fluid::Velocity:3:0:Forward
Fluid::LayerThicknessDelta:4:0:Forward
Fluid::LayerThickness:4:0:Forward
Fluid::LocalVelocityDelta:4:0:Forward
Fluid::LocalVelocity:4:0:Forward
Fluid::Velocity:4:0:Forward
Layer
Carte
MassL
Local
DivBi
DivMi
WaveM
MassB
MassC

Towards a Fluidity-ICOM adjoint

S. W. Funke and P. E. Farrell

August 25, 2011

AMCG

Why care about adjoints?

Three ways to adjoint a model

Introduction to `libadjoint`

Why care about adjoints?

The adjoint provides an efficient way to compute the gradient of a diagnostic model output with respect to model input parameters.

Applications

- Sensitivity analysis
- Data assimilation
- Parameter estimation
- Design optimisation
- Goal based error adaptivity
- Inverse problems

The Fluidity-ICOM model

Properties of `Fluidity-ICOM` that need to be taken into account in adjoint development:

- Solves the Navier-Stokes, Stokes, Shallow Water equations, Burgers equation, ...
- Written in Fortran 90/2003, C, C++, Python.
- Uses modern Fortran language features (derived types, dynamic memory allocation, operator overloading).
- PETSc is used for the linear solves.

Imperial College London

Three ways to adjoint a model

$$\begin{tikzcd}
\text{Continuous equations} \arrow[r] & \text{Discretised equations} \arrow[r] & \text{Source code}
\end{tikzcd}$$

Three ways to adjoint a model

$$\begin{tikzcd}
\text{Continuous equations} \arrow[r] \arrow[d, "\text{pen+paper}"] & \text{Discretised equations} \arrow[r] & \text{Source code} \\
\text{Continuous adjoint} & &
\end{tikzcd}$$

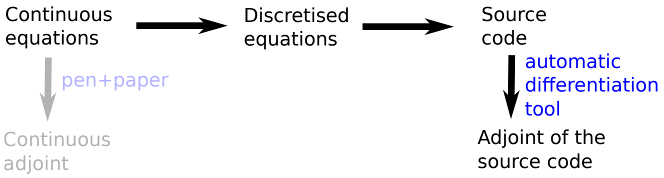
The continuous adjoint is ...

... a PDE that can be solved in a similar manner to the forward model.

- No problems with numerical schemes in the forward discretisation.
- Inconsistent gradient.

Imperial College
London

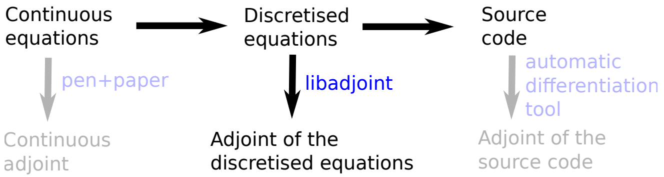
Three ways to adjoint a model



Adjoint of the source code

- Can (theoretically) be obtained automatically from the source code.
- Available AD tools are proprietary and/or support only a subset of the programming language.

Three ways to adjoint a model


Continuous equations
Discretised equations
Source code
pen+paper
libadjoint
automatic differentiation tool
Continuous adjoint
Adjoint of the discretised equations
Adjoint of the source code

Adjoint of the discretised equations

- Consistent gradient.
- Circumvents many problems with AD approach.
- ~~Is complicated to develop.~~ `Libadjoint` facilitates the development.

Imperial College London

Example: Burgers equation

The Burgers equation without viscosity has the form:

$$\frac{\partial u}{\partial t} + u \cdot \nabla u = 0.$$

The (explicit) discretisation with one nonlinear iteration per time step yields:

$$\underbrace{-(M + \Delta t A(u^n))}_{:=T(u^n)} u^n + M u^{n+1} = 0,$$

where M is the mass matrix, A is the discretised advection operator and Δt is the time step. We linearise the advection term using the velocity at the previous time step.

Example: Burgers equation

Three time steps can be written as a block matrix:

$$\underbrace{\begin{pmatrix} I & & & \\ T(u^0) & M & & \\ & T(u^1) & M & \\ & & T(u^2) & M \end{pmatrix}}_{A(u)} \underbrace{\begin{pmatrix} u^0 \\ u^1 \\ u^2 \\ u^3 \end{pmatrix}}_{u} = \underbrace{\begin{pmatrix} u_{\text{init}} \\ 0 \\ 0 \\ 0 \end{pmatrix}}_{b}$$

Therefore the adjoint equation reads:

$$\underbrace{\begin{pmatrix} I^* & \left(T(u^0) + \frac{\partial T}{\partial u^0}\Big|_{u^0} u^0\right)^* & & \\ & M^* & \left(T(u^1) + \frac{\partial T}{\partial u^1}\Big|_{u^1} u^1\right)^* & \\ & & M^* & \left(T(u^2) + \frac{\partial T}{\partial u^2}\Big|_{u^2} u^2\right)^* \\ & & & M^* \end{pmatrix} \begin{pmatrix} \lambda^0 \\ \lambda^1 \\ \lambda^2 \\ \lambda^3 \end{pmatrix} = \frac{\partial J}{\partial u}\Big|_{(u,\vec{m})}}_{\left(A(u)+\frac{\partial A(u)}{\partial u}u\right)^* \lambda = \frac{\partial J(u)}{\partial u}}$$

Libadjoint

libadjoint is a library that facilitates the adjoint model development.

The fundamental idea of libadjoint

A model is a sequence of linear solves.

The adjoint model development with libadjoint is subdivided in two steps:

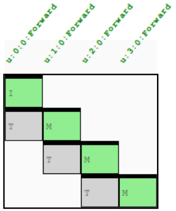
1. Annotation
2. Callback registration

Step 1: Annotation

- `libadjoint` provides a set of library calls with which a model may be annotated.
- Each linear solve is annotated to record what is being computed, what operators are acting on previously computed values, and what nonlinear dependencies those operators have.

This information is sufficient to describe the discretisation matrix of the forward model...

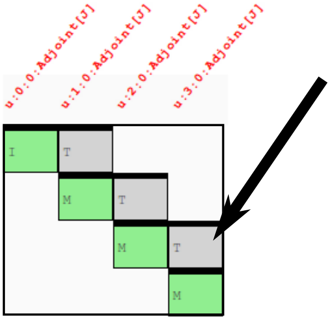
$$\begin{pmatrix} I & & & \\ -T(u^0) & M & & \\ & -T(u^1) & M & \\ & & -T(u^2) & M \end{pmatrix} \longrightarrow$$

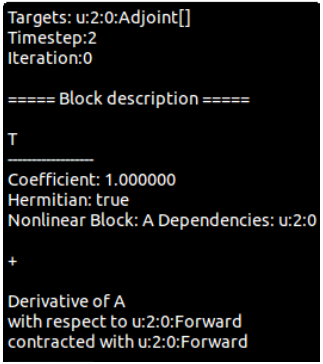


Imperial College London

Step 1: Annotation

...and so libadjoint can derive the associated adjoint system:




```
Targets: u:2:0:Adjoint[]
Timestep:2
Iteration:0

===== Block description =====

T
----------------
Coefficient: 1.000000
Hermitian: true
Nonlinear Block: A Dependencies: u:2:0

+

Derivative of A
with respect to u:2:0:Forward
contracted with u:2:0:Forward
```

Imperial College
London

Step 2. Register function callbacks

- `libadjoint` offers the facility to register function callbacks for computing the (Hermitian and derivative) action of the operator mentioned in the annotation.
- It also offers the facility to record a variable (in memory, on disk, or as a promise to compute it when necessary) mentioned in the annotation.

With these, it is possible for `libadjoint` to automatically assemble the adjoint equation as and when it is to be solved.

Strengths

- It is possible to adjoint models which use modern language features.
- External libraries, I/O, dynamic memory allocation, etc. are no longer problematic.
- The approach meshes well with AD: AD can be applied to the assembly routines only and its output interfaced with `libadjoint`, letting AD be applied where it is strong and letting `libadjoint` tie it all together.
- An advanced checkpointing strategy is implemented in `libadjoint`: `Revolve`[1].

[1] A. Griewank, A. Walther, revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation

Drawbacks

- Annotating the model must be done by hand, whereas a sufficiently advanced AD tool should not require user intervention (in theory).
- This is mitigated by the fact that whenever the forward model is changed, the AD tool must be reapplied, along with any hand-modifications; but if the model is annotated well, it will not be necessary to change the annotation.

Debugging tools

`libadjoint` offers debugging tools throughout the whole development process of the adjoint model:

- Visual output of the annotated model (forward and adjoint).
- Replay of the forward model through `libadjoint`. The forward model and the replay model should give exactly the same result.
- Automatic check for the correctness of the Hermitian implementation of the registered operators.
- Automatic check for the correctness of the registered derivative callbacks of nonlinear operators.

With these tools errors can be localised quickly and they allow a "trial and error" development approach.

Current status

The following equation sets have been adjointed:

- the linear shallow water equations.
- the non-linear Burgers equation.

A powerful framework for expressing CFD/GFD optimisation problems within the `Fluidity-ICOM` framework has been developed and applied to some simple examples (assimilating initial conditions from observations of the shallow water system).

We are now developing the adjoint for the non-linear Navier-Stokes solver... more about that in one year!