Simple automatic derivatives in numerical simulation and shape optimization

R. A. E. Mäkinen
Department of Mathematical Information Technology
University of Jyväskylä, FINLAND
E-mail: rainom@mit.jyu.fi

INTRODUCTION

Elementary example

Numerical solution of the nonlinear equation

$$f(x) := x - \cos x = 0.$$

Methods based on function values (sign!) only:

- bisection method, ...

Methods based on successive linearization of function f:

- Newton's method $f(x) \approx f(\hat{x}) + f'(\hat{x})(x - \hat{x})$
- Secant method $f(x) \approx f(\hat{x}) + \frac{f(\hat{x}) - f(\hat{\hat{x}})}{\hat{x} - \hat{\hat{x}}}(x - \hat{x})$
- Fixed point method $f(x) \approx x - \cos \hat{x}$ (no slope info used)

Do we still need derivatives in industrial applications ??

- In e.g. shape optimization, state variables are highly implicit and **complicated** functions of design variables
- Exact derivatives are **expensive** in terms of man-hours. Approximate derivatives are expensive in terms of CPU-time.
- Derivative based (optimization) methods are **not robust** enough in complicated industrial problems.

Model problem

Domain $\Omega(\boldsymbol{a})$ is defined by a vector of geometric parameters $\boldsymbol{a} = (a_1, \ldots, a_n)$. The domain is occupied by incompressible fluid which is modeled by the Navier–Stokes equations:

$$-\mu\Delta\vec{u} + \nabla p + \rho(\vec{u}\cdot\nabla)\vec{u} = \rho\vec{g}, \qquad \nabla\cdot\vec{u} = 0 \quad \text{in } \Omega(\boldsymbol{a}). \tag{1}$$

We wish to control the fluid flow by changing the shape of $\Omega(\boldsymbol{a})$. Mathematically this can be formulated as the following combined simulation and optimization problem:

$$\min_{\boldsymbol{a}\in U^{ad}} \left\{ J(\boldsymbol{a}, \vec{u}(\boldsymbol{a})) = \int_{\Omega(\boldsymbol{a})} |\vec{u}(\boldsymbol{a}) - \vec{u}_d|^2 \, dx \right\}, \tag{2}$$

where $\vec{u}(\boldsymbol{a})$ is the the solution of (1) for the given control parameter $\boldsymbol{a}$.

After FEM discretization of (1) we obtain the following nonlinear programming problem

$$\min_{\boldsymbol{a}\in U^{ad}} \{\mathcal{J}(\boldsymbol{a}) = J(\boldsymbol{a},\boldsymbol{u}(\boldsymbol{a}))\} \tag{3}$$

subject to

$$\boldsymbol{K}(\boldsymbol{a};\boldsymbol{u}(\boldsymbol{a}))\,\boldsymbol{u}(\boldsymbol{a}) = \boldsymbol{f}(\boldsymbol{u}(\boldsymbol{a})). \tag{4}$$

Assume that the mappings $\boldsymbol{a} \mapsto \boldsymbol{u}(\boldsymbol{a})$ and $(\boldsymbol{a},\boldsymbol{u}) \mapsto J(\boldsymbol{a},\boldsymbol{u})$ are differentiable.

- Principally it would be possible to treat $\boldsymbol{u}$ and $\boldsymbol{a}$ independent variables and iterate them simultaneously ("one shot method").
- One shot method is not convenient if one wants to utilize existing general purpose simulation codes. Usually the control variable $\boldsymbol{a}$ and the state variable $\boldsymbol{u}$ are iterated separately on two levels.

Choose initial guess $\boldsymbol{a}^{(0)},\ k := 0$

repeat

Choose initial guess $\boldsymbol{u}_0(\boldsymbol{a}^{(k)}),\ i := 0$

repeat

Determine $\Delta\boldsymbol{u}_i$

$\boldsymbol{u}_{i+1}(\boldsymbol{a}^{(k)}) := \boldsymbol{u}_i(\boldsymbol{a}^{(k)}) + \Delta\boldsymbol{u}_i$ (iterate state solution)

until $|\Delta\boldsymbol{u}_i|$ is small

Determine $\Delta\boldsymbol{a}^{(k)}$

$\boldsymbol{a}^{(k+1)} := \boldsymbol{a}^{(k)} + \Delta\boldsymbol{a}^{(k)}$ (improve design)

until $|\Delta\boldsymbol{a}^{(k)}|$ is small

Several ways to realize this abstract algorithm!

System (4) is usually solved using iteration

$$\boldsymbol{P}(\overline{\boldsymbol{u}})\,\Delta\boldsymbol{u} = \boldsymbol{f}(\overline{\boldsymbol{u}}) - \boldsymbol{K}(\overline{\boldsymbol{u}})\overline{\boldsymbol{u}}.$$

Natural choices for the preconditioner $\boldsymbol{P}(\overline{\boldsymbol{u}})$ are $\boldsymbol{K}(\overline{\boldsymbol{u}})$ (“Picard” linearization) and $\frac{\partial \boldsymbol{r}(\overline{\boldsymbol{u}})}{\partial \boldsymbol{u}}$ (exact or “Newton” linearization).

There are two fundamentally different approaches to realize the optimization loop. In classical descent methods the correction is computed from

$$\boldsymbol{Q}(\overline{\boldsymbol{a}})\,\Delta\boldsymbol{a} = -\nabla_{\boldsymbol{a}}\mathcal{J}(\overline{\boldsymbol{a}}),$$

where $\overline{\boldsymbol{a}}$ a the approximate design from the previous optimization iteration and $\boldsymbol{Q}(\overline{\boldsymbol{a}})$ is a Hessian approximation. In stochastic optimization methods (like Genetic Algorithms, GA) no gradient information is used to generate a set of new candidates for approximate optimum design.

1. Solve outer (optimization) problem using GA. Solve inner (state) problem using Picard iteration

$$\boldsymbol{K}(\bar{\boldsymbol{u}})\Delta\boldsymbol{u} = -\boldsymbol{r}(\bar{\boldsymbol{u}}).$$

+ robustness (GA has the potential to find global minimum instead of local one)
+ ease of implementation (no gradients or Jacobians are needed).
+ Moreover, stochastic optimization methods can be applied to more general optimization problems involving discrete variables or nonsmooth cost functions.
– Costly in terms of CPU-time. Picard iteration converges slowly also near the exact solution. As a rule GA needs hundreds or thousands cost function evaluations.
– No natural stopping criterion for stochastic optimization methods $\Longrightarrow$ assessment of accuracy may be difficult in practical problems

2. Solve outer problem using a descent method. Solve inner problem using the Newton method

$$\frac{\partial \boldsymbol{r}(\overline{\boldsymbol{u}})}{\partial \boldsymbol{u}} \Delta \boldsymbol{u} = -\boldsymbol{r}(\overline{\boldsymbol{u}}).$$

+ Accuracy and efficiency (super-linear or even quadratic convergence).

– Not robust: Newton method may fail to converge from a given initial guess, descent methods may find only a local minimum.

– Implementation of gradient and Jacobian computations.

3. Solve outer problem with a combination of GA and descent methods. Solve inner problem with a combination of Picard and Newton linearization.

+ This hybrid approach obviously shares the good properties of methods 1 and 2, namely robustness and reasonable efficiency (GA is used only to locate very rough approximation of minimum).

– Implementation of gradient and Jacobian computations.

Gradients and Jacobians are useful in large-scale simulation and optimization!

But...

People in industry not willing to spent days or weeks to program and to debug derivatives!!!

Difference approximations of partial derivatives, like

$$\frac{\partial \mathcal{J}(\boldsymbol{a}^0)}{\partial a_i} \approx \frac{\mathcal{J}(\boldsymbol{a}^0 + \delta \boldsymbol{e}^{(i)}) - \mathcal{J}(\boldsymbol{a}^0)}{\delta},$$

- way to supply derivative information without laborious programming
- expensive in large-scale problems.
- suffer accuracy problems
- but usually better than no derivatives at all !

EXACT DERIVATIVES USING AUTOMATIC DIFFERENTIATION (AD)

AD is **not**

- finite difference approximation of derivatives
- symbolic differentiation of a single expression using symbolic manipulation programs like Mathematica or Maple.

AD **is** a technique for augmenting computer programs with derivative computations without human intervention.

- Based on: every computer program executes a sequence of elementary arithmetic operations + the chain rule of derivative calculus
- Can be implemented e.g. using the operator overloading technique (Fortran 95, C++)

A more simple model problem:

$$J(\boldsymbol{a},u) = \int_{\Omega(\boldsymbol{a})} (u - u_d)^2 \, dx, \tag{5}$$

where $u = u(\boldsymbol{a})$ solves the 2D quasilinear scalar diffusion equation

$$\nabla \cdot (\kappa(u)\nabla u) + f(u) = 0 \quad \text{in } \Omega(\boldsymbol{a}). \tag{6}$$

FEM discretization:

$$\int_{\Omega} (k(u_h)\nabla u_h \cdot \nabla \varphi_j - f(u_h)\varphi_j) \, dx = 0, \quad j = 1, \ldots, N. \tag{7}$$

Our aim is to offer:

- Fully automatic exact Jacobian with respect to $\boldsymbol{u}$
- ”Semi-automatic” exact Jacobian with respect to design parameters

State problem (7) in matrix form

$$\boldsymbol{r}(\boldsymbol{a};\boldsymbol{u}) \equiv \boldsymbol{K}(\boldsymbol{a};\boldsymbol{u})\,\boldsymbol{u} - \boldsymbol{f}(\boldsymbol{a};\boldsymbol{u}) = 0.$$

Partial derivatives with respect to design:

$$\frac{\partial \mathcal{J}(\boldsymbol{a})}{\partial a_k} = \frac{\partial J(\boldsymbol{a},\boldsymbol{u})}{\partial a_k} - \boldsymbol{p}^{\mathrm{T}} \left(\frac{\partial \boldsymbol{r}(\boldsymbol{a};\boldsymbol{u})}{\partial a_k} \right),$$

where $\boldsymbol{p}$ solves the adjoint equation

$$\left(\frac{\partial \boldsymbol{r}(\boldsymbol{a};\boldsymbol{u})}{\partial \boldsymbol{u}} \right)^{\mathrm{T}} \boldsymbol{p} = \nabla_{\boldsymbol{u}} J(\boldsymbol{a},\boldsymbol{u}).$$

- The direct application of the 'forward mode' of AD to calculate e.g. $N \times N$ matrix $\partial \boldsymbol{r}/\partial \boldsymbol{u}$ is not computationally efficient.
- We will utilize the **sparsity** of the derivatives with respect to $\boldsymbol{u}$ and to work on the element level.

The global residual vector $\boldsymbol{r}(\boldsymbol{a};\boldsymbol{u})$ is obtained by the standard assembly process

$$\boldsymbol{r}(\boldsymbol{a};\boldsymbol{u}) = \sum_{e}^{\text{ass.}} \boldsymbol{r}^e(\boldsymbol{a};\boldsymbol{u}^e),$$
$$\boldsymbol{r}^e(\boldsymbol{a};\boldsymbol{u}^e) \equiv \boldsymbol{K}^e(\boldsymbol{a};\boldsymbol{u}^e)\boldsymbol{u}^e - \boldsymbol{f}^e(\boldsymbol{a};\boldsymbol{u}^e) \qquad \text{(local residual)}$$

The Integral-type cost functional is clearly separable, too:

$$J(\boldsymbol{a};u_h) = \sum_e \int_{\Omega_e} (u_h - u_d)^2\, dx = \sum_e J_e(\boldsymbol{a};\boldsymbol{u}^e).$$

- Need to apply automatic differentiation only to **local** contributions $\boldsymbol{r}^e(\boldsymbol{a};\boldsymbol{u}^e) \in \mathbb{R}^m$ and $J_e(\boldsymbol{a},\boldsymbol{u}^e)$. Here m is the number of **element** degrees of freedoms.
- Thus one needs to differentiate only scalars and "small" vectors with respect to "small" number of independent variables $a_1,\ldots,a_n$ and $u_1^e,\ldots,u_m^e$.

Assume that the Taylor expansion of nodal positions with respect to geometric parameters is available:

$$\boldsymbol{X}(:,i) = \boldsymbol{G}^{(i)}\boldsymbol{a} + \boldsymbol{c}^{(i)}, \quad i = 1,2.$$

- independent on the state problem
- matrices $\boldsymbol{G}^{(i)}$ can be produced by AD too !

Evaluate partial Jacobian of $\boldsymbol{r}$ with respect to $\boldsymbol{a}$ using the **chain rule**:

$$\frac{\partial \boldsymbol{r}}{\partial \boldsymbol{a}} = \frac{\partial \boldsymbol{r}}{\partial \boldsymbol{X}(:,1)}\boldsymbol{G}^{(1)} + \frac{\partial \boldsymbol{r}}{\partial \boldsymbol{X}(:,2)}\boldsymbol{G}^{(2)}.$$

Straightforward implementation using operator overloading AD

```
TYPE(AD_type):: rlok, ulok, Jaclok, uk, graduk, ...
do e=1,Number_of_elements
   ulok = u(ind(e,:))
   JacLok = AD_derivative( rlok, ulok )
   do k=1,Number_of_Gauss_points
      do j=1,n
         uk = uk + ulok(j)*phi(j)
         graduk = graduk + ulok(j)*gradphi(j)
      end do
      do i=1,n
         rlok(i)=rlok(i)+w(k)*(kappa(uk)*graduk.dot.gradphi(i))-f*phi(i))
      end do
   end do
   Call Assemble( Jac, JacLok )
   Call Assemble( res, rlok )
end do
```

- The concepts of the software should follow closely the language of mathematics. This offers a general and multidisciplinary framework.
- The software engineering concepts that the user needs to know should be kept at minimum. This lowers the user treshold and keeps the focus on modeling.
- The low-level details of numerical methods should be hidden from the user. The user should be able to control the key aspects of the numerical methods, however.
- The derivatives needed for linearizations and design sensitivities should be computed automatically by the software.
- Defining a system of coupled models should be as natural as defining the models one at the time. Too often, linking the software components describing the separate models are difficult and time-consuming.

Eliminate loops and indices:

```
...
TYPE(SysVector_type):: r, u, ...
TYPE(SysMatrix_type):: Jac, ...
Jac = AD_Derivative( r, u )
do while ( Integrate( Omega ) )
   phi = BasisFunction()
   r = kappa(u) *( grad(u) .dot. grad(phi) ) - f * phi
end do
```

Advantages:

- Model implemented almost in the language of the weak form of the P.D.E.
- more operator overloading based code can be inlined $\Longrightarrow$ better run-time performance

Numerrin 2.0 user interface (Fortran 95):

```
DO iter=1,itmax
   r = 0.0_wp; drda = 0.0_wp; drdu = 0.0_wp
   drdu = DefaultDerivative( r, u )
   drda = ShapeDerivative( r, G )
   DO WHILE ( Integrate(mesh) )
      phi => GetBasisFunctionPtr()
      r = ( (k(u)*grad(u)) .DOT. grad(phi) ) - f(u)*phi
   END DO
   du = LU( drdu, r )
   u = u - du
   if ( norm2(r) < toler ) exit
END DO
!
dJdu = DefaultDerivative( J, u )
dJ = ShapeDerivative( J, G )
J = 0.0_wp
DO WHILE ( Integrate(mesh) )
   phi => GetBasisFunctionPtr()
   J = (u-u_d)**2*phi
END DO
p = LU( drdu, dJdu, trans=.TRUE. )
CALL SysMatVec( drda, p, pt_x_drda, trans=.TRUE.)
dcost = dJ - pt_x_drda
```

Advances and disadvantages of using Fortran 95

- Almost seamless communication with classical fortran 77 software (LAPACK, optimization codes, ...)
- Array syntax, complex arithmetic,... is part of the language
- Fortran 95 lacks support for some essential concepts of object oriented programming (constructor/destructor, inheriation) $\Longrightarrow$ e.g. separate derived type definitions:

  Fortran 95:

  ```
  TYPE(Matrix_t):: A
  A = InitializeMatrix(m,n)
  ```

  C++:

  ```
  Matrix_t A(m,n);
  ```

- ”Non-typical” properties of F95 are bug-ridden in many compilers (Not uncommon phenomen with C++, too!)

APPLICATIONS

Multigrid solution of a Non-equilibrium radiation diffusion problem

(R. Glowinski & Jari Toivanen)

Gray approximation for thermal radiation diffusion:

$$\frac{\partial E}{\partial t} - \nabla \cdot (\kappa_1(T,E)\nabla E) = \sigma(T)(T^4 - E)$$
$$\frac{\partial T}{\partial t} - \nabla \cdot (\kappa_2(T)\nabla T) = -\sigma(T)(T^4 - E)$$
$$\sigma(T) = \frac{z^3}{T^3}, \quad \kappa_1(T,E) = \frac{1}{3\sigma(T) + \frac{E}{|\nabla E|}}, \quad \kappa_2(T) = kT^{5/2}$$

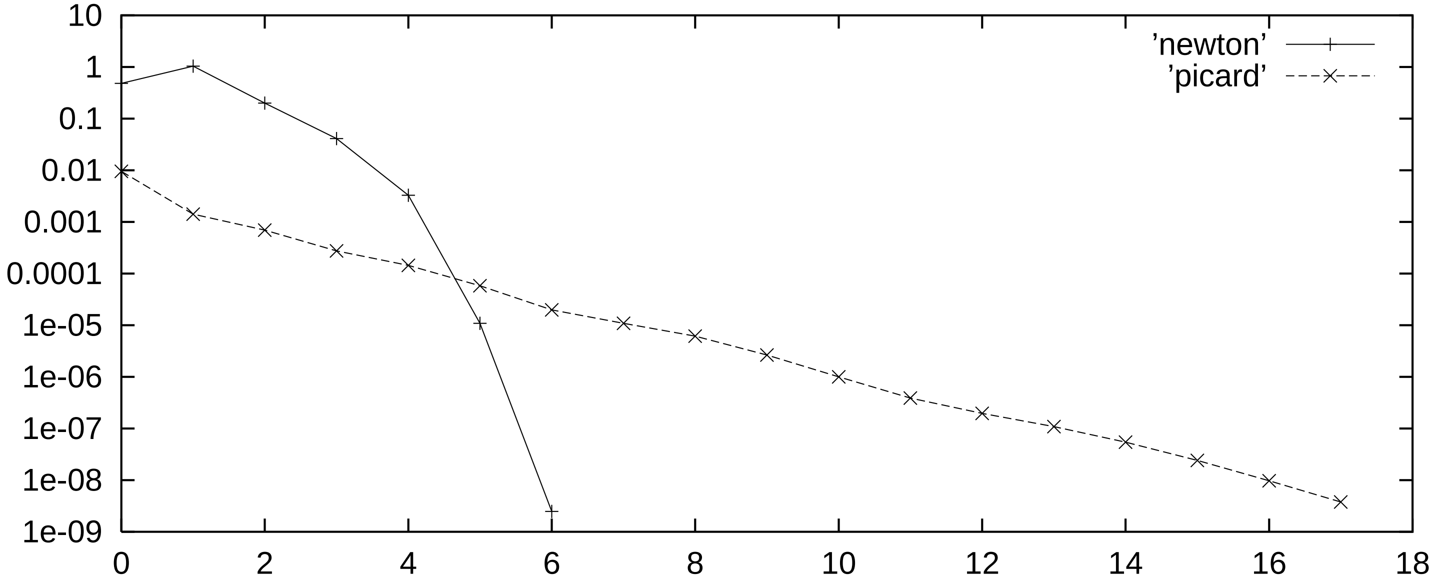
(E is the radiation energy, T material temperature, z atomic mass number)

Fortran 90 implementation, Jacobian for Newton's method computed using AD.

Coupled heat equation and incompressible Navier–Stokes equations

(Janne Martikainen)

- 2D, Stabilized FEM, Picard or Newton+AD, (C++ implementation)
- Linear systems solved using Martikainen's Algebraic Multigrid solver for problems arising from FEM discretization of general coupled PDE systems
- Re ≈ 100, NDOFS$\approx 15,000$, total CPU-time 2 minutes (P 1.2 GHz)



Simulation of turbulent flow in an expanding channel

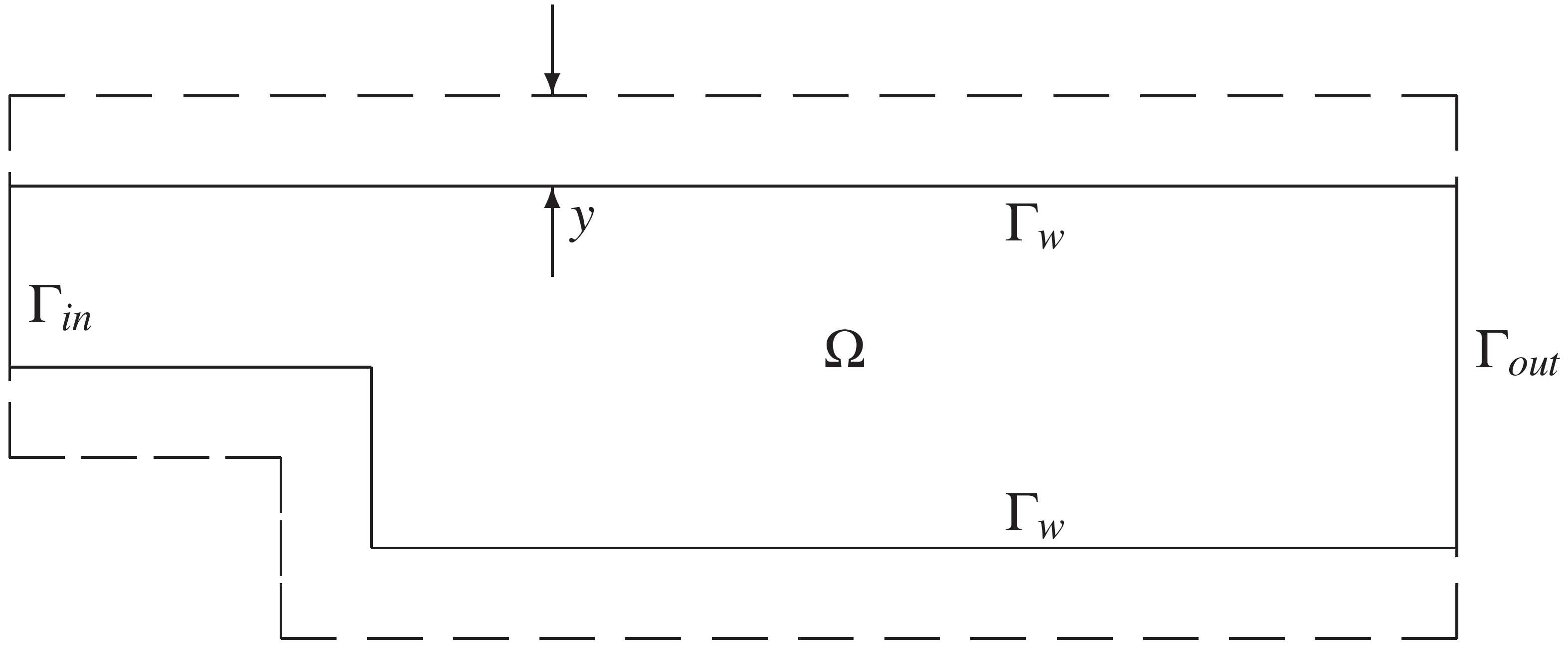


Figure 1: Physical domain – – – Computational domain Ω ——

Reynolds averaged Navier–Stokes equations with the k-ε turbulence model in computational domain Ω:

$$
\begin{aligned}
-\nabla\cdot\boldsymbol{\sigma}+\rho(\vec{u}\cdot\nabla)\vec{u}&=\rho\vec{g}\\
\nabla\cdot\vec{u}&=0\\
-\nabla\cdot((\mu_0+c_1\hat{\mu})\nabla k)+\rho\vec{u}\cdot\nabla k&=\hat{\mu}\Phi-\rho\varepsilon\\
-\nabla\cdot((\mu_0+c_2\hat{\mu})\nabla\varepsilon)+\rho\vec{u}\cdot\nabla\varepsilon&=C_1\hat{\mu}\frac{\varepsilon}{k}\Phi-C_2\rho\frac{\varepsilon^2}{k}\\
\hat{\mu}&=C_\mu\rho k^2/\varepsilon.
\end{aligned}
$$

Nonstandard boundary conditions on Γ_w:

$$
\sigma_t=-\operatorname{sign}(u_t)\rho(u^\star)^2,\quad u_n=0,\quad k=(u^\star)^2/\sqrt{C_\mu},\quad \varepsilon=\frac{(u^\star)^3}{\kappa y}.
$$

Here $u^\star$ is solved from the nonlinear equation (Ψ given wall law function)

$$
u_t=u^\star\Psi(y^+),\quad y^+=\frac{\rho u^\star y}{\mu}.
$$

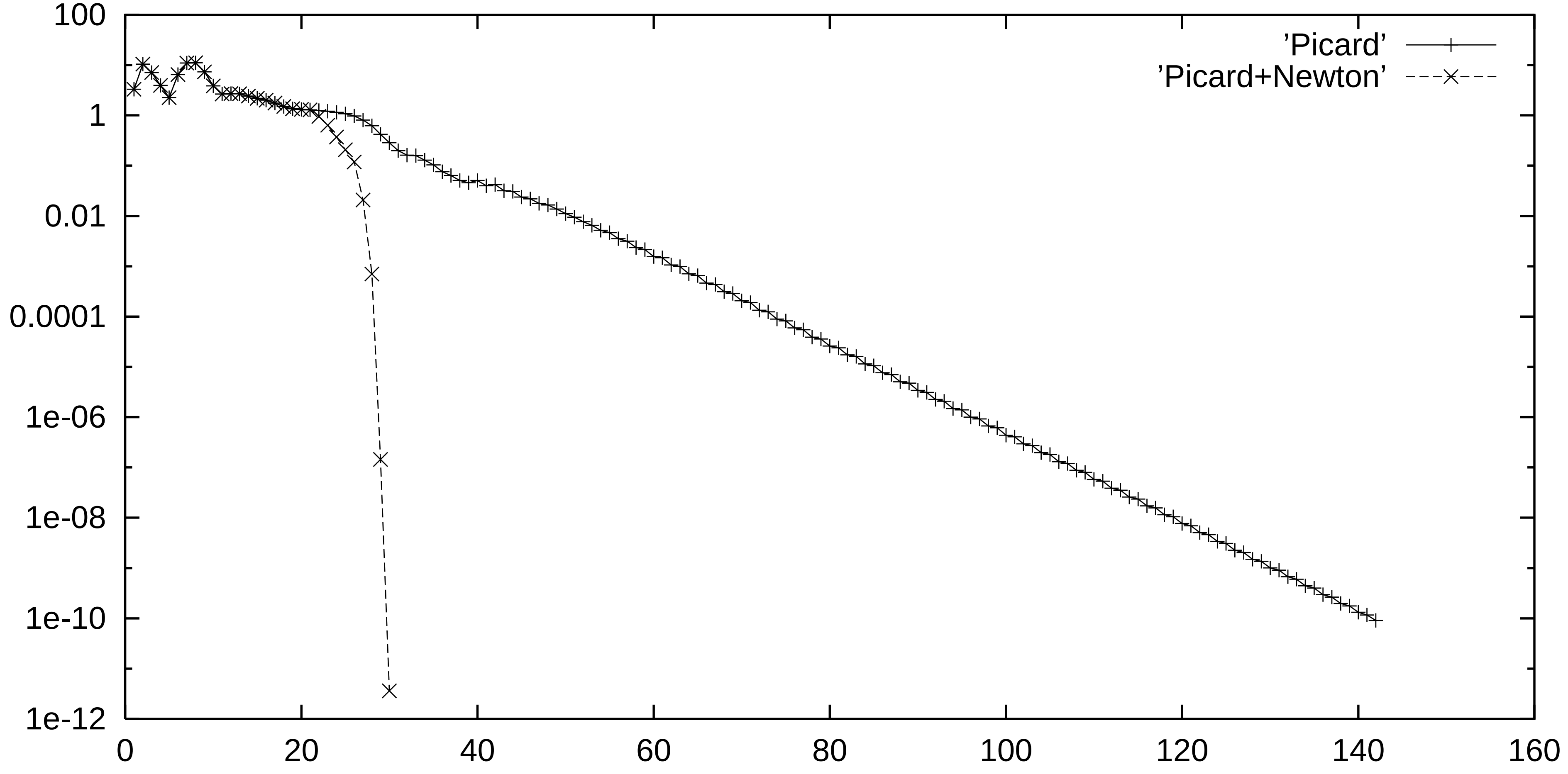


Figure 2: Residual norm vs. iterations (Re $= 10^6$, 2300 elements, Franca–Frey stabilization)

Shape optimization in spreading process

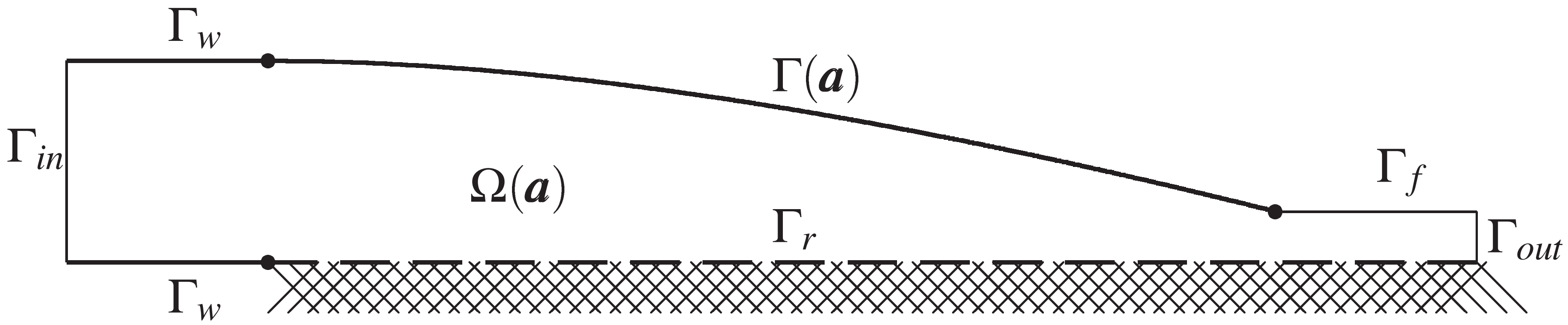


Figure 3: Problem geometry

The non-Newtonian ”power-law” fluid flow in the device $\Omega(\boldsymbol{a})$ is modeled by the equations

$$-\nabla\cdot\boldsymbol{\sigma}+\rho\vec{u}\cdot\nabla\vec{u}=\rho\vec{g}$$

$$\nabla\cdot\vec{u}=0$$

$$\boldsymbol{\sigma}=\mu(\nabla\vec{u}+\nabla\vec{u}^{\mathrm{T}})-pI,\quad \mu=k_0\Phi^{\frac{n-1}{2}}.$$

On the inflow boundary and on the solid walls both velocity components are prescribed. On the free fluid surface Γ_f the vertical velocity component is prescribed to zero. On Γ_r the tangential velocity component and the normal forces are prescribed:

$$u_t = u_R \quad \sigma_n = \lambda u_n + f_0 \quad \text{on } \Gamma_{out}.$$

The aim is to control the filtration profile (normal velocity) on Γ_r by changing the shape of the back wall $\Gamma(\boldsymbol{a})$ of the device to minimize the functional

$$\mathcal{J}(\boldsymbol{a}) = \int_{\Gamma_r} |u_n - \hat{u}_n|^2 \, dx.$$

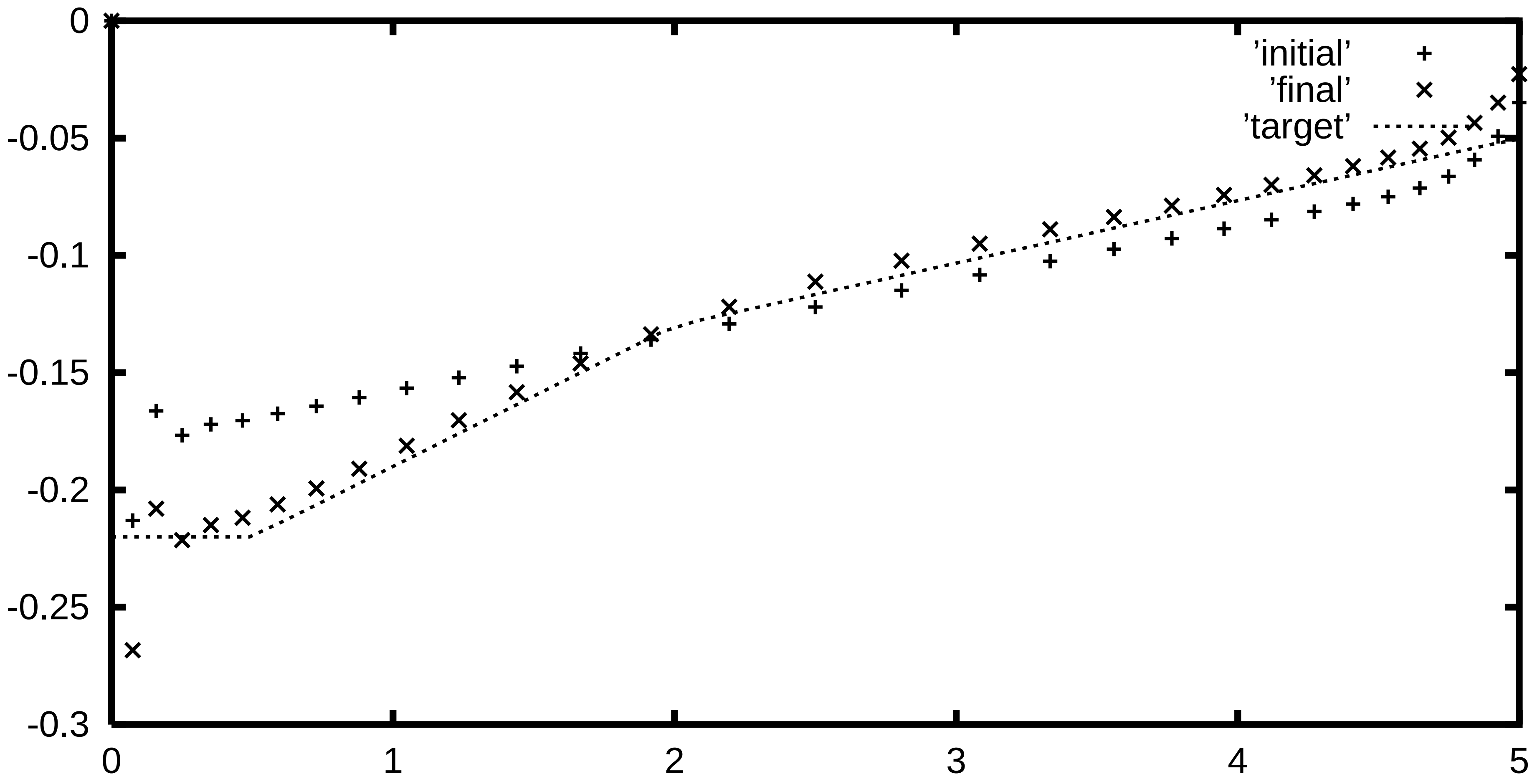


Figure 4: Outflow profiles on Γ_r

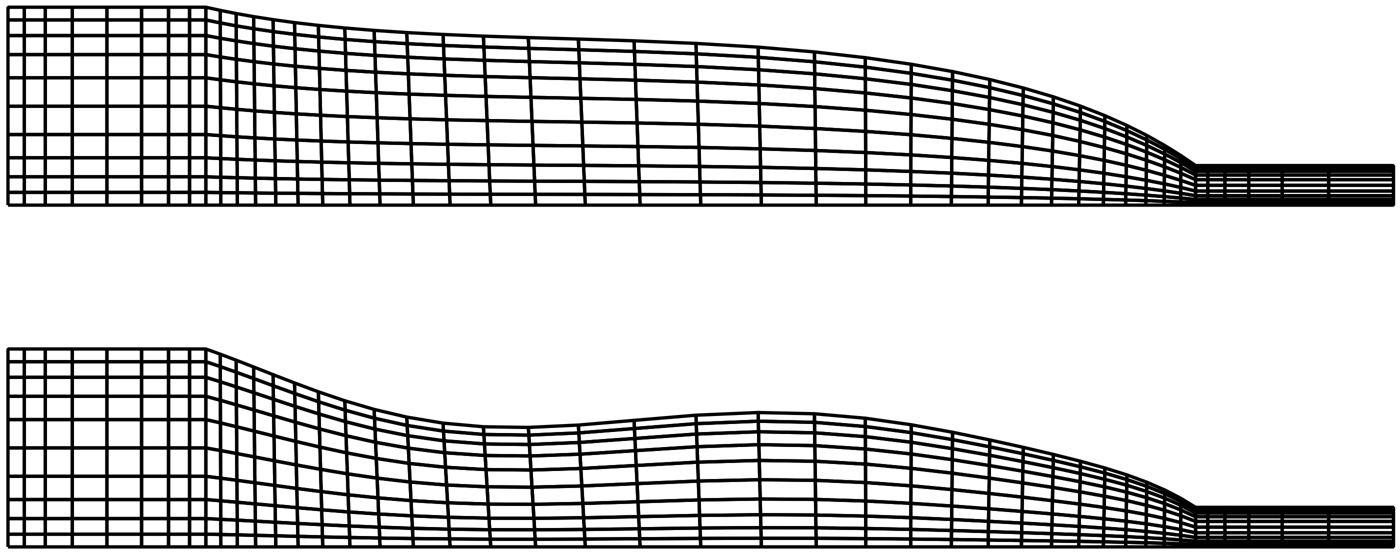


Figure 5: Initial (upper) and optimum (below) designs. $\Gamma(\boldsymbol{a})$ parameterized using a Bezier curve with 6 control points. Optimum shape found after **15** quasi-Newton iterations

OPEN PROBLEMS AND FURTHER STUDY

- More automatic Mesh generation and design parametrization instead of current ad hoc solutions ?
- Automatic derivatives of unstructured meshes with respect to design parameters ?
- Own modeling language instead of Fortran 95 (like FreeFrem) ?
- Web-based user interface to increase portability (like WWW-Nimbus) ?

CONCLUSIONS

- Derivatives are useful in analysis and optimization of P.D.E. systems.

  Derivative-free methods for robustness – derivative based methods for accuracy.

- Derivatives of (intermediate) quantities are not physically interesting in general.

  Derivative based methods will be attractive in industrial applications only if derivatives can be produced easily.

- AD is a technique to obtain exact derivatives without laborious and error-prone hand-coding.

  We have proposed a simple (= don't bother with 'forward' or 'reverse', call trees, etc.) but reasonably efficient way to apply AD in FEM based simulation and optimization.

- Every new numerical software should include at least a simple implementation of automatic differentiation!