CONSISTENT APPROXIMATIONS, AUTOMATIC DIFFERENTIATION AND DOMAIN DECOMPOSITION FOR OPTIMAL SHAPE DESIGN

OLIVIER PIRONNEAU AND NICOLAS DICESARE
(Dedicated to J.L.Lions)

Abstract. Our purpose is to use mesh adaptation for accuracy and Domain Decomposition Methods (DDM) for parallelization. To speed up the computations we wish to embed the mesh refinement loop within the optimization loop and for this we shall use the theory of Consistent Approximation (CA). However the sensitivity analysis becomes quite involved and so we will use AD (Automatic Differentiation). Consequently this paper is about mixing several new developments of Optimal Shape Design with a DDM approach.

1 Introduction

Consider the shape optimization problem
\[
\min_{\Omega} \{ J(\Omega, u) : a(u, \hat{u}) = (L, \hat{u}) \quad \forall \hat{u} \in V(\Omega) ; \ u \in V(\Omega) \}
\]
where the boundary of \( \Omega \) is the design parameter. Our objective is to use mesh adaptation, DDM and the optimization iterations to reduce the computing time. Automatic Differentiation will be used as a tool to simplify the implementation and we will show that an in house library based on operator overloading for AD is quite sufficient for these two dimensional problems.

The plan of the paper is as follows. First we will recall the theory of Consistent Approximations (CA) of E.Polak[10]. Then we will apply it to a boundary control problem discretized by the Finite Element method (see for ex. Ciarlet [4]) with a Schwarz algorithm (see for ex. P.L. Lions [8]) on two overlapping domains. We will then present our Automatic Differentiation (AD) software written in C++ with expression templates and finally show a result on a simple academic problem.
2 Consistent Approximations

Consider an optimization problem and its discrete approximation with a discretization parameter $h$:

\[(\mathcal{P}) \quad \min_{z \in O} J(z) \quad \text{Optimality function } \theta(z)\]

\[(\mathcal{P}_h) \quad \min_{z_h \in O_h} J_h(z_h) \quad \text{Approximated function } \theta_h(z)\]

Consider a sequence $h_n \downarrow 0$, then $\{O_{h_n}\}$ must be such that $O_{h_n} \subset O_{h_{n+1}} \subset O \subset \mathcal{H}$, where $\mathcal{H}$ is some Hilbert space and $\langle \cdot, \cdot \rangle$ its scalar product.

Consider the following algorithm to solve $\mathcal{P}$:

2.1 Algorithm

1. Choose $z^0, \epsilon^0, \beta \in ]0, 1[$. Set $n = 0, \epsilon = \epsilon^0, h = h_0$

2. Compute $z^n_m$ performing $m$ iterations of a descent algorithm on $\mathcal{P}_h$ from starting point $z^n$ so as to achieve $\theta_h(z^n_m) > -\epsilon$

3. Set $\epsilon = \beta \epsilon, h = h_{n+1}, z^{n+1}_m = z^n_m, n = n + 1$ and go to Step 2.

Theorem 1 If $\mathcal{P}_h$ is a consistent approximation (defined below) of $\mathcal{P}$, then the above algorithm converges, in the sense that every accumulation point of $\{z_n\}_n$ is a zero of the optimality function of $\mathcal{P}$.

2.2 Definitions

Following Polak[10] we introduce the following concepts:

2.2.1 Epi-convergence

$\{\mathcal{P}_h\}$ epi-converge to $\mathcal{P}$, $\mathcal{P}_h \rightarrow \mathcal{P}$, if :

- $\forall z \in O, \exists \{z_h\} : z_h \in O_h, \text{such that } z_h \rightarrow z$ (consistency)

- if $\{z_h\}_h, z_h \in O_h, z_h \rightarrow \bar{z}$ then $\bar{z} \in O$ (density).

- and $J_h(z_h) \rightarrow J(z)$, as $h \rightarrow 0$ (continuity in $h$)

2.2.2 Optimality functions

A function $\theta : O \rightarrow R$ is an optimality function for $\mathcal{P}$ if,

- $\theta$ is sequentially upper-semicontinuous.

- $\theta(z) \leq 0$ for all $z \in O$. $\theta(\bar{z}) = 0 \forall \bar{z} \in O$ local minimizer for $\mathcal{P}$.

An optimality function $\theta_h$ for $\mathcal{P}_h$ is defined similarly.
2.2.3 Consistent Approximation

The pairs \( \{ \mathcal{P}_h, \theta_h \} \) are consistent approximations to \((\mathcal{P}, \theta)\), if

- \( \mathcal{P}_h \) epi-converge \( \mathcal{P} \).
- \( \forall \{ z_h \in O_h \}_h \), \( z_h \rightarrow z \) then \( \lim_{h \to 0} \theta_h(z_h) \leq \theta(z) \) (h-\( \frac{1}{2} \)continuity).

3 Application to a Control Problem

Consider the boundary control problem \((D\) is a given subset of \(\Omega)\)

\[
\min_{v \in L^2(\Gamma)} \{ J(v) = \int_D (u - u_d)^2 : u - \Delta u = f \text{ in } \Omega, \frac{\partial u}{\partial n} |_\Gamma = v \}.
\]

Optimality conditions are derived as follows:

\[
\delta J = 2(u - u_d, \delta u) + o() = (p, \delta v)_\Gamma
\]

where \( p - \Delta p = 2(u - u_d)1_D \frac{\partial p}{\partial n} |_\Gamma = 0 \)

Let us use a finite Element Method with \( u \in V_h \) continuous piecewise linear on the triangles of a triangulation of \(\Omega\):

\[
\min_{v \in L^2_h} J_h(v) = \|u - u_d\|_D^2 : (u, w) + (\nabla u, \nabla w) = (f, w) = (v, w)_\Gamma \forall w \in V_h
\]

Then the discrete optimality conditions are

\[
\delta J = (p_h, \delta v)_\Gamma \text{ where } (p_h, v) + (\nabla p_h, \nabla v) = 2(u - u_d, v)_D \forall w \in V_h
\]

3.1 The Schwarz Algorithm

Let \( \Omega = \Omega_1 \cup \Omega_2 \), let \( \Gamma = \partial \Omega \) and \( \Gamma_{ij} = \partial \Omega_i \cap \Omega_j \). Assume \( D \subset \Omega_1 \) for clarity. The multiplicative Schwarz algorithm for the Laplace equation starts from a guess \( u_1^0, u_2^0 \) and computes the solution of

\[-\Delta u = f \text{ in } \Omega, \quad u|_\Gamma = u_\Gamma
\]

as the limit in \( n \) of \( u_i^n, i = 1, 2 \) defined by

\[
-\Delta u_{1}^{n+1} = f \text{ in } \Omega_1, \quad u_{1}^{n+1}|_{\Gamma \cap \Omega_1} = u_\Gamma, \quad u_{1}^{n+1}|_{\Gamma_{12}} = u_2^n
\]

\[
-\Delta u_{2}^{n+1} = f \text{ in } \Omega_2, \quad u_{2}^{n+1}|_{\Gamma \cap \Omega_2} = u_\Gamma, \quad u_{2}^{n+1}|_{\Gamma_{21}} = u_1^n
\]

(1)
3.2 Discrete Optimality conditions

The discretized problem is

$$\min_{v \in L^h} J^N(v) = \|u^N - u_d\|^2_D : \quad u^0_i = 0, \quad n = 1..N \quad \forall w \in V_h$$

$$u^n_i|_{\partial \Omega_{ij}} = u^{n-1}_j \quad (u^n_i, w)_\Omega + (\nabla u^n_i, \nabla w)_\Omega = (f, w)_\Omega + (v, w)_\Omega$$

where $N$ is the number of Schwarz iterations. The exact discrete optimality conditions of the discrete problems are (see Lions[7] for details):

$$p^N - \Delta p^N = 2(u^N - u_d)1_D \quad p^{N-1} - \Delta p^{N-1} = 0 \quad p^N_{\Gamma_{ij}} = p^N$$

These are very difficult to implement because we must store all intermediate functions generated by the Schwarz algorithm and integrate the system for $p^n$ in the reverse order.

A possible remedy is to use some sort of penalty method, such as

$$\min_{v \in L^h} J^f(v) = \|u - u_d\|^2_D + \frac{1}{\epsilon} \|u_1 - u_2\|^2_{\Omega_1 \cap \Omega_2} :$$

$$(u_i - f)_\Omega + (\nabla u_i, \nabla w)_\Omega = \pm (v, w)_{\Gamma_{ij}} \quad \forall w \in V_h(\Omega_i)$$

but the choice of $\epsilon$ is difficult (see also [9] for alternatives). So here we will use the theory of consistent approximations; we will not try to compute the exact optimality functions but only an approximate one.

Define

$$\theta_h = -\|p_h\|_{\Gamma}$$

where $p_h$ is computed by $N$ iterations of the Schwarz algorithm.

3.3 Hypothesis of the Theorem

Recall that the continuous problem is

$$\min_{v \in L^2(\Gamma)} \{ J(v) = \|(u - u_d)\|^2_D : \quad (u, w) + (\nabla u, \nabla w) = (f, w) + (v, w)_{\Gamma} \}$$

$$\theta = -\|p\|_{0, \Gamma} \quad (p, w) + (\nabla p, \nabla w) = 2(u - u_d, w)_D \quad \forall w \in H^1(\Omega)$$

and that the discrete problem is

$$\min_{v \in L^h} J^N_h(v) = \|u^N - u_d\|^2_D : \quad u^0_i = 0, \quad n = 1..N \quad \forall w \in V_h$$

$$u^n_i|_{\partial \Omega_{ij}} = u^{n-1}_j \quad (u^n_i, w)_\Omega + (\nabla u^n_i, \nabla w)_\Omega = (f, w)_\Omega + (v, w)_\Omega$$

$$\theta^N_h = -\|p_h\|_{\Gamma}$$

Notice that $\theta$ being the proportional to the norm of the gradient of the cost function, so it is an optimality function: $\theta(u) = 0$ is equivalent to $u = u_d$.

However in the discrete case $\theta^N_h(u^N_h)$ is not the gradient of the discrete cost function, nevertheless it is an approximate optimality function because

$$\theta^N_h(u^N_h) = 0 \quad \Rightarrow \quad u^N_h = \Pi_h u_d \quad \Rightarrow \quad J^N_h(v) = 0$$

where $\Pi_h$ is the $L^2$-projector on $V_h(\Omega_h)$. 


1. Continuity of $J$: $v^m \to v \iff J(v^m) \to J(v)$ because $v \to u$ is continuous from $L^2(\Gamma)$ to $H^1(\Omega)$.

2. Inclusion: $L_{h/2}$ for instance is included in $L_h$ if it is obtained by adding more points on boundary (and if $\Gamma$ is polygonal).

3. Continuity of $\theta, \theta^N_h$

   \[
   v^m \xrightarrow{L^2(\Gamma)} v, \quad u^m \xrightarrow{H^1(\Omega)} u \quad \Rightarrow \quad p^m \to p \\
   v^m_j \xrightarrow{R} v_j, \quad u^m_h \xrightarrow{H^1(\Omega)} u_h \Rightarrow p^m_h \to p_h.
   \]

4. Consistency: Set $H = (h, \frac{1}{N})$

   $\forall v \in L^2(\Gamma), \exists \{v_H\}, \quad v_H \to v : u_H \to u, \quad p_H \to p.$

5. Continuity in $H$ at 0

   $v_H \to v \Rightarrow u_H \to u \Rightarrow J_H(u_H) \to J(u) \Rightarrow p_H|_{\Gamma} \to p$.

### 3.4 Numerical example

We propose the following example: let $B(r)$ be the circle of center (-0.2,-0.2) and of radius $r$. Let $C$ be the circle of center (1.4,0.4) and of radius 0.4. We have chosen

\[
\Omega_1 = \{x, y : x^2 + y^2 < 1\} \quad \Omega_2 = \{0, 2\} \times \{0, 1\}
\]

and $D = B(0.2), \quad E = B(0.4), \quad u_d = 1_D, \quad f = 1_E$

![Figure 1: Four of the triangulations generated by the automatic mesh generator based on the edge size at the boundary, as it decreases](image1)

![Figure 2b The final solution](image2)
Figure 2: The error log-plot when starting with a coarse mesh and when starting with a fine mesh. Most of the optimization is done with the coarse mesh and so the computing time is considerably smaller. Two curves are for the criteria versus the iteration number, these do not converge to zero. The two other curves are the norms of the adjoint $p$ on $\partial C$.

4 Home made Automatic Differentiation

Principle

Consider the problem of computing $J'(u)$ at $u=2.3$ where

$$J'(u) \quad \text{where} \quad \begin{align*}
x &= 2u(u + 1) \\
y &= x + \sin(u) \\
J &= x \ast y
\end{align*}$$

Recipe: Below each line of code add its differentiated form

$$\begin{align*}
x &= 2 \ast u \ast (u + 1) \\
dx &= 2 \ast u \ast du + 2 \ast du \ast (u + 1) \\
y &= x + \sin(u) \\
dy &= dx + \cos(u) \ast du \\
J &= x \ast y \\
dJ &= dx \ast y + x \ast dy
\end{align*}$$

The derivative will be obtained by running this program with $u = 2.3$, $dv = 1$, and $dx = dy = 0$ at start time.

Loops and branching statements are not more complicated. Indeed an if statement

$$\text{A; if ( bool ) then B else C; D;}$$

is in fact 2 programs

$$\text{A; B; D; } \Rightarrow \text{ A';B'; D;D'; A; C; D; } \Rightarrow \text{ A';A‘; C;C‘; D;D‘;}$$

which can be recombined into

$$\text{A;A‘; if ( bool ) then B;B‘ else C;C‘; D;D‘}$$

Similarly a loop statement like
A; for i:=1 to 3 do B(i); D;

is in fact

A; B(1); B(2); B(3); D; => A; A'; B(1); B'(1); ...; B(3); B'(3); D; D';

which is also recombined into

A; A'; for i:=1 to 3 do { B(i); B'(i); } D; D'

4.1 Principle of programming

Each differentiable variable stores in fact two numbers: its value and the value of its derivative. So we may replace all variables by an array of size 2. Hence the program becomes

```c
float y[2], x[2], u[2];
// x = 2 u (u+1)
x[0] = 2 * u[0] * (u[0] + 1);
// dx = 2 u du + 2 du (u+1)
x[1] = 2 * u[0] * u[1] + 2 * u[1] * (u[0] + 1);
y[0] = x[0] + sin(u[0]);
y[1] = x[1] + cos(u[0]) * u[1];
J[0] = x[0] * y[0];
J[1] = x[1] * y[0] + x[0] * y[1];
```

Now, following [6] we create a C++ class whereby each variable contains the array of size two just introduced and we redefine the standard operations of linear algebra by giving our own definition such as the one used below for the multiplication:

```c
class dfloat{
public:
    float v[2];
friend dfloat& operator * (dfloat& a, dfloat& b)
    {
        dfloat c;
        c.v[0] = a.v[0] * b.v[0];
        return c;
    }
};
```

Now all we have to do is to change all “float” variables into `dfloat` variables and indicate that the derivatives are to be taken with respect to `v` at `v = v_0`.

```c
void main () {    dfloat x, u;
    u.init(2.3, 1); // Derivative at u0=2.3 requested
    x = 2 * u * u + 1;
    ... }
```
4.2 Partial Derivatives

When there are several parameters the method remains the same essentially. For instance consider \((u_1, u_2) \rightarrow J(u_1, u_2)\) defined by the program

\[
\begin{align*}
y_1 &= l_1(u_1, u_2) \\
y_2 &= l_2(u_1, u_2, y_1) \\
J &= l_3(u_1, u_2, y_1, y_2)
\end{align*}
\]

Apply the same recipe

\[
\begin{align*}
y_1 &= l_1(u_1, u_2) \\
dy_1 &= \partial_{u_1} l_1(u_1, u_2) dx_1 + \partial_{u_2} l_1(u_1, u_2) dx_2 \\
y_2 &= l_2(u_1, u_2, y_1) \\
dy_2 &= \partial_{u_1} l_2 dx_1 + \partial_{u_2} l_2 dx_2 + \partial_{y_1} l_2 dy_1 \\
J &= l_3(u_1, u_2, y_1, y_2) \\
dJ &= \partial_{u_1} l_3 dx_1 + \partial_{u_2} l_3 dx_2 + \partial_{y_1} l_3 dy_1 + \partial_{y_2} l_3 dy_2
\end{align*}
\]

Run the program twice, a) with \(dx_1 = 1, dx_2 = 0\), b) with \(dx_1 = 0, dx_2 = 1\).

Or duplicate the lines \(dy_i = \) and evaluate both at once with \(dx_1 = \delta_{ij}\).

The C++ implementation is done with a class construct of the type

```cpp
template <int N> class dfloat{
public:
    float v[N];
friend dfloat& operator * (dfloat& a, dfloat& b)
    {
        dfloat c; c.v[0] = a.v[0] * b.v[0];
        for(int i=1;i<N;i++)
            c.v[i] = a.v[i] * b.v[0] + a.v[0] * b.v[i];
        return c;
    }
};
```

It is better to use a template class because \(N\), which is the number of parameters, should not be fixed at a default value. However \(N\) may not be known at compile time (in our case the number of discretization point on the boundary) and then dynamic arrays are needed

```cpp
class dfloat{
public:
    float* v; int N;
friend dfloat& operator * (dfloat& a, dfloat& b)
    {
        dfloat c; c.v=new float[N];
        c.v[0] = a.v[0] * b.v[0];
        for(int i=1;i<N;i++)
            c.v[i] = a.v[i] * b.v[0] + a.v[0] * b.v[i];
        delete [] c.v;
        return c;
    }
};
```
The problem then is that large number of temporary arrays are created dynamically, like c.v here, and that takes a lot of time. An optimization can be found in Dicesare[2] which uses Expression Templates and reduces considerably the creation of temporaries so as to arrive at performance similar to those obtained with the template class library explained above. Still to the user the simplicity of the class library is kept and it is extremely easy to link any C program to these libraries for a sensitivity analysis with respect to a few parameters, say less than 50.

4.3 Reverse Mode

The reference in terms of efficiency for the computation of partial derivatives is the so called “reverse mode”.

Consider a simple model problem where \( J \) is a function of two parameters and is computed by a program that uses two intermediate variables:

\[
\begin{align*}
y_1 &= l_1(u_1, u_2) \\
y_2 &= l_2(u_1, u_2, y_1) \\
J &= l_3(u_1, u_2, y_1, y_2)
\end{align*}
\]

Let us build the Lagrangian by associating to each intermediate variable a dual or adjoint variable \( p \), except for the last one which is taken equal to 1:

\[
L = p_1[y_1 - l_1(x)] + p_2[y_2 - l_2(x, y_1)] + J - l_3(x, y_1, y_2)
\]  

(2)

Stationarity with respect to \( y_2, y_1 \) (in that order) gives

\[
\begin{align*}
0 &= p_2 - \frac{\partial l_3}{\partial y_2}(x, y_1, y_2) \\
0 &= p_1 - p_2 \frac{\partial l_2}{\partial y_1}(x, y_1) - \frac{\partial l_3}{\partial y_1}(x, y_1, y_2)
\end{align*}
\]

This gives \( p_2 \) first and then \( p_1 \) and then \( J'_x \) is

\[
\frac{\partial J}{\partial u_i} = p_1 \frac{\partial l_1}{\partial u_i} + p_2 \frac{\partial l_2}{\partial u_i} + \frac{\partial l_3}{\partial u_i}
\]

The difference with the direct approach is that whatever the number of intermediate variables the adjoint variables \( p_i \) are evaluated once only and so the complexity of the computation is much less. On the other hand the method requires a symbolic manipulation of the program itself and so it is not easy to implement as a class library (see Auber[1]).

There are currently two implementation of the reverse mode, ADOL-C[5] and Odyssee[11]. Implementation of the direct mode is easy with template class using fixed N but not so with expression templates. The interested user may retrieve our implementation “fadbad” in http://www.ann.jussieu.fr/dicesare. A comparison with ADOL-C is available in [2].
5 An Academic Optimal Shape Example

Consider the problem of recovering the position of a disk $B(x_0, y_0)$ of radius $r$ and center $(x_0, y_0)$ which gave $u_d$ solution of

$$-\Delta u = f \text{ in } \Omega \setminus B(x_0, y_0) \quad u|_{\partial B(x_0, y_0)} = 0$$

with Dirichlet or Neumann conditions on $\partial \Omega$. We set

$$\min_{x_0, y_0} \{ J(x_0, y_0) = \int_D |u - u_d|^2 : \text{subject to (3)} \}$$

The problem is discretized by the finite element method of order 1. Analytical derivation of the two partial derivatives of $J$ is quite complex. On the other hand it is very easy to link the C-code to the fad library to obtain these derivatives. A gradient method with fixed step size turned out to be very fast, as shown below on Figure 3 and 4. Computations where done with $r = 0.2$ (the height of the rectangle is 1), $f = 10I_C$ where $C$ has same center as $B$ and radius 0.3 and Dirichlet conditions 0 on the left incomplete circle and 1 or the vertical right side and homogeneous Neumann on the horizontal sides. The observation domain is in the rectangular part of the domain: $\Omega \cap \{ x^2 + y^2 > 1 \}$. The results are shown on Figures 3 and 4.

![Figure 3: Top: target state $u_d$ and computed state after 4 iterations. Bottom: intermediate results at iteration 1, 2 and 3.](image)

**Remark** All computations have been performed with freefem+[3].

References


Figure 4: Convergence curve for $J$, $|\frac{\partial J}{\partial x_0}|$, $|\frac{\partial J}{\partial y_0}|$, and the boundary indicator $(|u|^2 + |\frac{\partial u}{\partial n}|^2)_{B(x_0,y_0)}$.


