An Interface Between Optimization and Application for the Numerical Solution of Optimal Control Problems

Matthias Heinkenschloss
Rice University
and
Luís N. Vicente
Universidade de Coimbra

An interface between the application problem and the nonlinear optimization algorithm is proposed for the numerical solution of distributed optimal control problems. By using this interface, numerical optimization algorithms can be designed to take advantage of inherent problem features like the splitting of the variables into states and controls and the scaling inherited from the functional scalar products. Further, the interface allows the optimization algorithm to make efficient use of user provided function evaluations and derivative calculations.

Categories and Subject Descriptors: G.4. [**Mathematical Software**]: User Interfaces

General Terms: Algorithms, Design

Additional Key Words and Phrases: Optimization, simulation, optimal control

1. INTRODUCTION

This paper is concerned with the implementation of optimization algorithms for the solution of smooth discretized optimal control problems. The problems under consideration can be written as

$$\begin{aligned} \min \ & f(y,u) \\ \text{s.t.} \ & c(y,u) = 0, \\ & \underline{y} \le y \le \overline{y}, \\ & \underline{u} \le u \le \overline{u} \end{aligned} \tag{1}$$

This work was supported in part by the DoE under Grant DE-FG03-95ER25257, by the AFOSR under Grant F49620–93–1–0280, by Centro de Matemática da Universidade de Coimbra, FCT, and Praxis XXI 2/2.1/MAT/346/94.
Name: Matthias Heinkenschloss
Address: Department of Computational and Applied Mathematics, Rice University, Houston, TX 77005–1892, USA, E-Mail: `heinken@caam.rice.edu`
Name: Luís N. Vicente
Address: Departamento de Matemática, Universidade de Coimbra, 3000 Coimbra, Portugal, E-Mail: `lvicente@mat.uc.pt`

or

$$\begin{aligned} \min\ & \widehat{f}(u) = f(y(u), u) \\ \text{s.t.}\ & \underline{y} \le y(u) \le \overline{y}, \\ & \underline{u} \le u \le \overline{u}. \end{aligned} \tag{2}$$

Here u represents the control, y and $y(u)$ represent the state, and $c(y,u) = 0$ represents the state equation. If the implicit function theorem is applicable, the state equation $c(y,u) = 0$ defines a function $y(\cdot)$ of u and in this case the problem (1) can be reduced to (2). We note that (1) and (2) are related, but they are not necessarily equivalent. If for given u the equation $c(y,u) = 0$ has more than one solution, the implicit function theorem will select one solution branch $y(u)$, provided the assumptions of the implicit function theorem are satisfied. Hence, the feasible set of (2) is contained in (1) but the feasible sets are not necessarily equal.

Examples of optimal control problems of the form (1) or (2) are given, e.g., in Borggaard and Burns [1997], Chen and Hoffmann [1991], Cliff, Heinkenschloss, and Shenoy [1997], Friedman and Hu [1998], Gunzburger, Hou, and Svobotny [1993], Handagama and Lenhart [1998], Ito and Kunisch [1996], Kupfer and Sachs [1992], Lions [1971], Neittaanmäki and Tiba [1994].

Discretized optimal control problems are large scale nonlinear programming problems with a particular structure. Structure arises from the partitioning of the variables in controls u and states y, from the underlying infinite dimensional problem, and from the discretization. Since these problems are nonlinear programming problems, they can in principle be solved using existing nonlinear programming solvers such as LANCELOT [Conn et al. 1992], LOQO [Shanno and Vanderbei 1997; Vanderbei 1998], or SNOPT [Gill et al. 1997]. These solvers, as well as other general nonlinear programming solvers, require user provided subroutines that evaluate the objective function and its gradient and the constraint function and its Jacobian matrix. Thus, they require the evaluation of $f(x)$, $\nabla f(x)$, $c(x)$, $c'(x)$ given $x = (y,u)$ for problem (1) and the evaluation of $\widehat{f}(u)$, $\nabla \widehat{f}(u)$, $y(u)$, $y'(u)$ given u for problem (2). If derivative information is not available, then general nonlinear programming solvers typically have an option that allows the approximation of first-order derivatives by finite differences. For several discretized optimal control problems the application of general nonlinear programming solvers has been done. This approach, however, has severe limitations.

Reasons that limit the applicability of codes for general nonlinear programming problems to discretized optimal control problems are the following:

i) If discretizations are refined, the problems (1) and (2) tend to exhaust available memory if Jacobians are stored (see, e.g., the numerical tests performed by Mittelmann and Maurer [1998]).

ii) The description of the optimal control problems in the sparse formats required by some general nonlinear programming solvers can be very difficult. This is, e.g., the case for optimal control problems governed by partial differential equations which are discretized using existing finite element packages. The incorporation of application dependent linear solvers such as multigrid methods for the solution of linearized PDE state equations is very difficult or even impossible.

iii) The view of the discretized optimal control problem as a finite dimensional nonlinear programming problem ignores the underlying infinite dimensional problem structure. Instead of a mesh-independent convergence behavior one can often observe a deterioration of the convergence as the discretization is refined due to improper scaling and artificial ill-conditioning.

In practice, optimization methods that have been proven successful for general nonlinear programming problems are tailored to a specific class of discretized optimal control problems, often optimal control problems governed by ODEs and DAEs (see, e.g., Betts [1997], Petzold, Rosen, Gill, Jay, and Park [1997], Schulz [1997], and Varvarezos, Biegler, and Grossmann [1994]). More often, many new developments in optimization methods are incorporated late or not at all into solution approaches for optimal control problems. In fact, the gradient method is still frequently used for the solution of unconstrained optimal control problems min $\widehat{f}(u)$.

We believe that the gap between optimization methods and their application to optimal control problems can be narrowed and in many cases even be closed by the provision of an interface between optimization algorithms and optimal control applications. The purpose of this paper is to develop a framework for such an interface. We assume that $\mathcal{Y}$, $\mathcal{U}$, and Λ are finite dimensional Hilbert spaces of dimension n_y, n_u, and n_y, respectively. These Hilbert spaces can be identified with $I\!R^{n_y}, I\!R^{n_u}$, and $I\!R^{n_y}$, respectively, but are equipped with scalar products $\langle\cdot,\cdot\rangle_{\mathcal{Y}}$, $\langle\cdot,\cdot\rangle_{\mathcal{U}}$, and $\langle\cdot,\cdot\rangle_{\Lambda}$. The functions

$$\begin{aligned} f : \mathcal{Y} \times \mathcal{U} &\to I\!R, \\ c : \mathcal{Y} \times \mathcal{U} &\to \Lambda, \end{aligned}$$

are assumed to be at least once differentiable. In some of our discussions we will also use second derivatives of f and c. This Hilbert space structure allows us to incorporate scaling information into the problem description which is important for a mesh-independent convergence behavior of the optimization algorithms. The interface explores the structure of the problem arising from the partitioning of variables into states and controls. In particular, this will be important for the description of derivative information. Solution of systems involving partial Jacobians of $c(y, u)$ are left to the application. This ensures that application dependent linearized state equation solvers such as multigrid methods can be used. The interface takes into account that linear system solves and function and derivative evaluations provided by the application are done inexactly. For example, this allows iterative linear system solvers on the application side and sensitivity and adjoint computations based on the infinite dimensional problem, not on the discretized problem which might require expensive mesh sensitivities [Borggaard and Burns 1997].

The interface can be used to implement a variety of optimization algorithms for (1) and (2), including conjugate gradient methods, Newton and quasi–Newton methods, augmented Lagrangian methods, sequential quadratic programming methods, and interior–point methods. To deal with storage limitations, matrix-free representations of linear operators in Hilbert spaces and problem scaling, Jacobians and Hessians are not passed to the optimizer as matrices, but only their application to a given vector is available through our interface. Thus, typically the above optimization algorithms have to be implemented in a matrix–free fashion using iterative methods such as the conjugate–gradient method or GMRES for the solution of linear systems within the optimization. We believe this interface is particularly useful for, but not restricted to problems governed by partial differential equations.

The description of our interface is conceptual and not tied to a specific programming language. We have used implementations of this interface in Fortran 77 and MATLAB[1] to solve a variety of optimal control problems, most of which are governed by partial differential equations. Other implementations of this interface are possible. For example, in C++ our interface could be implemented elegantly using the Hilbert Class Library (HCL) of Gockenbach and Symes [1997] (see also [Gockenbach et al. 1997]). HCL is a collection of C++ classes that implement mathematical objects such as vectors, linear and nonlinear operators and some algorithms for solving linear operator equations and unconstrained minimization problems. HCL is broader in scope than the interface proposed here and provides all the ingredients for a concrete implementation of our interface in C++. Our goal is to describe the information that needs to be exchanged between the application and derivative based optimization algorithms for the specific problem class (1) and (2). The description of our interface is based on the mathematical language used for optimal control problems and the notation used in (1) and (2). Our interface is not tied to a specific programming language.

This paper is structured as follows. In Section 2 we approach the scaling of the problem and illustrate its use in a few particular instances. Section 3 addresses the calculation of derivatives using sensitivity and adjoint equation methods. The optimization-application interface that we propose for the numerical solution of distributed optimal control problems is described in detail in Section 4. A few code fragments corresponding to parts of known optimization algorithms are given in Section 5.1 to illustrate the use of this interface and Section 5.2 illustrates the interface from an optimal control point of view. Section 6 discusses limitations and extensions of our framework.

2. SCALING OF THE PROBLEM

The scalar products $\langle\cdot,\cdot\rangle_{\mathcal{Y}}$, $\langle\cdot,\cdot\rangle_{\mathcal{U}}$, and $\langle\cdot,\cdot\rangle_{\Lambda}$ induce a scaling into the problem that is important for the performance of the optimization algorithms. The scalar products influence the computation of the gradients and other derivatives, they influence the definition of adjoints, and they are present in all subtasks that require scalar products, such as quasi-Newton updates and Krylov subspace methods. We will describe their influence on the gradient computation here and defer the discussion of their effect on Hessians, adjoints, and quasi-Newton updates to Appendix A.

The partial gradients of f are defined by the relations

$$\begin{aligned} \lim_{h_y\to 0} |f(y+h_y,u)-f(y,u)-\langle\nabla_y f(y,u),h_y\rangle_{\mathcal{Y}}|/\|h_y\|_{\mathcal{Y}} &= 0, \\ \lim_{h_u\to 0} |f(y,u+h_u)-f(y,u)-\langle\nabla_u f(y,u),h_u\rangle_{\mathcal{U}}|/\|h_u\|_{\mathcal{U}} &= 0. \end{aligned} \qquad (3)$$

In finite dimensions all norms are equivalent and, thus, the choice of norms in the denominators in (3) do not influence the definition of the gradient. The choice of the scalar product in the numerator, however, does.

[1]MATLAB is a registered trademark of The MathWorks, Inc., `info@mathworks.com`, `http://www.mathworks.com`.

It will be illustrative to study the effect of the scalar products on the gradient computation in more detail. Each scalar product on $\mathbb{R}^k$ can be identified with a symmetric positive definite matrix and we therefore write

$$\langle y, v\rangle_{\mathcal{Y}} = y^\top T_y v, \tag{4}$$

$$\langle u, w\rangle_{\mathcal{U}} = u^\top T_u w, \tag{5}$$

$$\langle \lambda, c\rangle_{\Lambda} = \lambda^\top T_\lambda c, \tag{6}$$

where $T_y, T_\lambda \in \mathbb{R}^{n_y \times n_y}$ and $T_u \in \mathbb{R}^{n_u \times n_u}$ are symmetric positive definite matrices. We emphasize that this is done for illustration only. The weighting matrices are never directly accessed, but only the value of a scalar product for two given vectors is needed.

If $\langle y, v\rangle_{\mathcal{Y}} = y^\top v$, $\langle u, w\rangle_{\mathcal{U}} = u^\top w$, then (3) yields

$$\nabla_y f(y,u) = {}_{\mathrm{e}}\nabla_y f(y,u), \quad \nabla_u f(y,u) = {}_{\mathrm{e}}\nabla_u f(y,u),$$

where

$$\begin{aligned} {}_{\mathrm{e}}\nabla_y f(y,u) &= \left(\tfrac{\partial}{\partial y_1} f(y,u), \ldots, \tfrac{\partial}{\partial y_{n_y}} f(y,u)\right)^\top, \\ {}_{\mathrm{e}}\nabla_u f(y,u) &= \left(\tfrac{\partial}{\partial u_1} f(y,u), \ldots, \tfrac{\partial}{\partial u_{n_u}} f(y,u)\right)^\top \end{aligned}$$

denote the gradient with respect to the Euclidean scalar products, i.e., the vectors of first-order partial derivatives.

Now we consider the two scalar products (4) and (5). From

$${}_{\mathrm{e}}\nabla_y f(y,u)^\top v = \left(T_y^{-1}\,{}_{\mathrm{e}}\nabla_y f(y,u)\right)^\top T_y v = \langle T_y^{-1}\,{}_{\mathrm{e}}\nabla_y f(y,u), v\rangle_{\mathcal{Y}} \quad \forall v,$$

and (3) we can see that

$$\nabla_y f(y,u) = T_y^{-1}\,{}_{\mathrm{e}}\nabla_y f(y,u). \tag{7}$$

Similarly,

$$\nabla_u f(y,u) = T_u^{-1}\,{}_{\mathrm{e}}\nabla_u f(y,u). \tag{8}$$

The representations (7) and (8) of the gradients can also be interpreted differently. Since T_y and T_u are symmetric positive definite, we can write them as the product of two symmetric positive definite matrices, $T_y = \left(T_y^{1/2}\right)^2$ and $T_u = \left(T_u^{1/2}\right)^2$. Now, we can define $\widetilde{y} = T_y^{1/2} y$, $\widetilde{u} = T_u^{1/2} u$, and $\widetilde{f}(\widetilde{y}, \widetilde{u}) = f(T_y^{-1/2}\widetilde{y}, T_u^{-1/2}\widetilde{u})$. If we compute the first-order partial derivatives of $\widetilde{f}$, then

$$\nabla_{\widetilde{y}} \widetilde{f}(\widetilde{y}, \widetilde{u}) = T_y^{-1/2}\,{}_{\mathrm{e}}\nabla_y f(y,u), \qquad \nabla_{\widetilde{u}} \widetilde{f}(\widetilde{y}, \widetilde{u}) = T_u^{-1/2}\,{}_{\mathrm{e}}\nabla_u f(y,u).$$

If we scale these vectors by $T_y^{-1/2}$ and $T_u^{-1/2}$, respectively, then we obtain (7) and (8). See also [Dennis and Schnabel 1983, Ch. 7].

3. DERIVATIVE COMPUTATIONS: ADJOINTS AND SENSITIVITIES

Sensitivity and adjoint equation approaches are used to compute derivative information in optimal control problems. In this section, we briefly describe what these approaches are in the context of this paper and how they can be used in derivative computations.

We consider the problem

$$\begin{aligned} &\min \; f(y,u) \\ &\text{s.t. } \; c(y,u) = 0 \end{aligned} \tag{9}$$

with associated Lagrangian

$$\ell(y,u,\lambda) = f(y,u) + \langle \lambda, c(y,u)\rangle_\Lambda \tag{10}$$

and the associated reduced problem

$$\min \widehat{f}(u) = f(y(u),u). \tag{11}$$

In (11) the function $y(\cdot)$ is defined via the implicit function theorem as a solution of $c(y,u) = 0$. We assume that the assumptions of the implicit function theorem applied to $c(y,u) = 0$ are satisfied.

Typically, sensitivity and adjoint equation approaches are used to compute the gradient and second-order derivative information for $\widehat{f}$. However, the same issues also arise for certain first and second order derivative computations related to the problem (9). The main purpose of this section is to show the commonalities in these approaches for (9) and (11) and to establish a common framework that can be used in many optimization algorithms for (9) and (11) and in fact for (1) and (2). For more discussions on sensitivity and adjoint equation approaches we refer to the literature. See, e.g., the collection [Borggaard et al. 1998].

In this section we use the sensitivity and adjoint equation approaches to compute the gradient and second-order derivative information for $\widehat{f}$ and ℓ. The fact that $\widehat{f}$ and f are objective functions is not important. It is only important that $\widehat{f} : \mathcal{U} \to I\!\!R$ depends on the implicit function $y(u)$. In general the sensitivity and adjoint equation approaches are needed when derivative information of a function $\widehat{h} : \mathcal{U} \to I\!\!R$ is computed that is of the form $\widehat{h}(u) = h(y(u),u)$. Thus most of what is said in the following also applies in this context. In particular, if additional constraints $d(y,u) = 0$ and $\widehat{d}(u) = d(y(u),u) = 0$, $d : \mathcal{Y} \times \mathcal{U} \to I\!\!R^k$ are present in (9) or (11), respectively, then the derivations in this section can be applied to the component functions $\widehat{d}_i$ or the Lagrangian $f(y,u) + \langle \lambda, c(y,u)\rangle_\Lambda + \mu^\top d(y,u)$.

3.1 First-order derivatives

Under the assumptions of the implicit function theorem the derivative of the implicitly defined function $y(\cdot)$ is given as the solution of

$$c_y(y(u),u)y'(u) = -c_u(y(u),u). \tag{12}$$

This equation is called the *sensitivity equation* and its solution is called the sensitivity of y. We can now compute the gradient of $\widehat{f}$:

$$\begin{aligned} \langle \nabla \widehat{f}(u), v\rangle_{\mathcal{U}} &= \langle \nabla_y f(y(u),u), y'(u)v\rangle_{\mathcal{Y}} + \langle \nabla_u f(y(u),u), v\rangle_{\mathcal{U}} \\ &= \langle \nabla_y f(y(u),u), -c_y(y(u),u)^{-1}c_u(y(u),u)v\rangle_{\mathcal{Y}} + \langle \nabla_u f(y(u),u), v\rangle_{\mathcal{U}} \\ &= \langle -\big(c_y(y(u),u)^{-1}c_u(y(u),u)\big)^* \nabla_y f(y(u),u) + \nabla_u f(y(u),u), v\rangle_{\mathcal{U}}. \end{aligned}$$

Hence,

$$\nabla \widehat{f}(u) = -\Big(c_y(y(u),u)^{-1}c_u(y(u),u)\Big)^* \nabla_y f(y(u),u) + \nabla_u f(y(u),u). \tag{13}$$

The formula (13) is used in *the sensitivity equation approach* to compute the gradient. First, the sensitivity matrix

$$S(y,u) \;=\; c_y(y(u),u)^{-1}c_u(y(u),u)$$

is computed and then the gradient is formed using (13).

To introduce the adjoint equation approach, we rewrite the formula (13) for the gradient as follows:

$$\nabla\widehat{f}(u) \;=\; -c_u(y(u),u)^*\big(c_y(y(u),u)^*\big)^{-1}\nabla_y f(y(u),u) + \nabla_u f(y(u),u).$$

Thus one can compute the adjoint variables $\lambda(u)$ by solving the adjoint equation

$$c_y(y(u),u)^*\lambda(u) \;=\; -\nabla_y f(y(u),u) \tag{14}$$

and then compute the gradient using

$$\nabla\widehat{f}(u) = c_u(y(u),u)^*\lambda(u) + \nabla_u f(y(u),u). \tag{15}$$

This is *the adjoint equation approach* to compute the gradient.

Traditionally, the sensitivity equation approach and the adjoint equation approach have been used in the context of the reduced problem (11). However, the same techniques are also needed to compute derivative information for the solution of (9).

Consider the Lagrangian (10). Its partial gradients are

$$\nabla_y\ell(y,u,\lambda) = \nabla_y f(y,u) + c_y(y,u)^*\lambda, \qquad \nabla_u\ell(y,u,\lambda) = \nabla_u f(y,u) + c_u(y,u)^*\lambda.$$

We see that $\nabla_y\ell(y,u,\lambda) = 0$ corresponds to the adjoint equation

$$c_y(y,u)^*\lambda = -\nabla_y f(y,u). \tag{16}$$

If we define $\lambda(y,u)$ as the solution of (16), then

$$\nabla_u\ell(y,u,\lambda)|_{\lambda=\lambda(y,u)} = \nabla_u f(y,u) - c_u(y,u)^*(c_y(y,u)^*)^{-1}\nabla_y f(y,u).$$

In particular,

$$\nabla\widehat{f}(u) = \nabla_u\ell(y,u,\lambda)|_{y=y(u),\lambda=\lambda(u)}.$$

With

$$W(y,u) = \begin{pmatrix} -c_y(y,u)^{-1}c_u(y,u) \\ I_{n_u} \end{pmatrix}$$

we can write

$$\nabla_u\ell(y,u,\lambda)|_{\lambda=\lambda(y,u)} = W(y,u)^*\begin{pmatrix}\nabla_y f(y,u)\\ \nabla_u f(y,u)\end{pmatrix}$$

and

$$\nabla\widehat{f}(u) = W(y,u)^*\begin{pmatrix}\nabla_y f(y,u)\\ \nabla_u f(y,u)\end{pmatrix}\Big|_{y=y(u)}.$$

An optimization algorithm applied to the solution of (9) may require the evaluation of the Lagrangian $f(y,u)+\langle\lambda(y,u),c(y,u)\rangle_\Lambda$, where $\lambda(y,u)$ is the solution of (16). If the adjoint equation approach is used for the derivatives, the adjoint variables $\lambda(y,u)$ can be calculated. If only the sensitivities $c_y(y,u)^{-1}c_u(y,u)$ and their adjoints are provided, adjoint variables cannot be computed from (16). In such a situation we can evaluate the corresponding value of the Lagrangian by solving the *linearized state equation*

$$c_y(y,u)s \;=\; -c(y,u) \tag{17}$$

and by using the relation

$$\begin{aligned}\langle \lambda(y,u), c(y,u)\rangle_\Lambda &= -\langle (c_y(y,u)^*)^{-1}\nabla_y f(y,u), c(y,u)\rangle_\Lambda \\ &= -\langle \nabla_y f(y,u), c_y(y,u)^{-1}c(y,u)\rangle_{\mathcal{Y}} \,. \end{aligned}\tag{18}$$

The introduction of $W(y,u)$ which plays an important role in solution methods for (16) allows an elegant and compact notation for the first-order derivatives and, as we will see in the following, for the second-order derivatives. It also localizes the use of the sensitivity equation approach and the adjoint equation approach in the derivative calculations. In all derivative computations, the sensitivity equation approach or the adjoint equation approach is only needed to evaluate the application of $W(y,u)$ and $W(y,u)^*$ onto vectors. For example, the computation of the y-component z_y of $z = W(y,u)d_u$ is done in two steps:

$$\begin{array}{lrcl} \text{Compute} & v_y &=& -c_u(y,u)d_u. \\ \text{Solve} & c_y(y,u)z_y &=& v_y. \end{array}$$

If the sensitivities $S(y,u) = c_y(y,u)^{-1}c_u(y,u)$ are known, then $z_y = -S(y,u)d_u$. The equation $c_y(y,u)z_y = v_y$ is a generalized linearized state equation, cf. (17). Similarly, for given d the matrix-vector product $z = W(y,u)^*d$, $d = (d_y, d_u)$, is computed successively as follows:

$$\begin{array}{lrcl} \text{Solve} & c_y(y,u)^*v_y &=& -d_y. \\ \text{Compute} & v_u &=& c_u(y,u)^*v_y. \\ \text{Compute} & z &=& v_u + d_u. \end{array}$$

Again, if the adjoint of the sensitivities $S(y,u) = c_y(y,u)^{-1}c_u(y,u)$ is known, then $z = -S(y,u)^*d_y + d_u$. The equation $c_y(y,u)^*v_y = -d_y$ is a generalized adjoint equation, cf. (16).

3.2 Second-order derivatives

The issue of sensitivities and adjoints not only arises in gradient calculations, but also in Hessian computations. The Hessian of the Lagrangian

$$\nabla^2_{xx}\ell(y,u,\lambda) = \begin{pmatrix} \nabla^2_{yy}\ell(y,u,\lambda) & \nabla^2_{yu}\ell(y,u,\lambda) \\ \nabla^2_{uy}\ell(y,u,\lambda) & \nabla^2_{uu}\ell(y,u,\lambda) \end{pmatrix} \tag{19}$$

and the reduced Hessian

$$\widehat{H}(y,u) = W(y,u)^* \begin{pmatrix} \nabla^2_{yy}\ell(y,u,\lambda) & \nabla^2_{yu}\ell(y,u,\lambda) \\ \nabla^2_{uy}\ell(y,u,\lambda) & \nabla^2_{uu}\ell(y,u,\lambda) \end{pmatrix} W(y,u)\Big|_{\lambda=\lambda(y,u)} \tag{20}$$

play an important role. Both matrices (19) and (20) are important in algorithms based on the sequential quadratic programming (SQP) approach [Fletcher 1987, Ch. 12]. Moreover, it is known, see, e.g., [Dennis et al. 1998; Heinkenschloss 1996], that the Hessian of the reduced functional in (11) is given by

$$\nabla^2 \widehat{f}(u) = \widehat{H}(y(u), u).$$

We note that the computation of (19) and (20) requires knowledge of the adjoint variables λ. In many algorithms, these are computed via the adjoint equations (16). If only the sensitivities $c_y(y,u)^{-1}c_u(y,u)$ and their adjoints are provided, adjoint variables cannot be computed from (16). If no estimate for λ is available, then the operators in (19) and (20) cannot be computed. In cases in which $\nabla_y f(y,u) \approx 0$ for (y,u) near the solution, one may set $\lambda = \lambda(y,u) \approx 0$, cf. (16). This leads to the approximations

$$\nabla^2_{xx}\ell(y,u,\lambda) \approx \begin{pmatrix} \nabla^2_{yy}f(y,u) & \nabla^2_{yu}f(y,u) \\ \nabla^2_{uy}f(y,u) & \nabla^2_{uu}f(y,u) \end{pmatrix} \tag{21}$$

and

$$\widehat{H}(y,u) \approx W(y,u)^* \begin{pmatrix} \nabla^2_{yy}f(y,u) & \nabla^2_{yu}f(y,u) \\ \nabla^2_{uy}f(y,u) & \nabla^2_{uu}f(y,u) \end{pmatrix} W(y,u). \tag{22}$$

The situation $\nabla_y f(y,u) \approx 0$ often arises in least squares functionals $f(y,u) = \frac{1}{2}\|y-y_d\|^2_{\mathcal{Y}} + \frac{\gamma}{2}\|u\|^2_{\mathcal{U}}$, where y_d is some desired state. In this case $\nabla_y f(y,u) = y - y_d$ and if the given data y_d can be fitted well, then $\nabla_y f(y,u) \approx 0$. In this case, the approximation (22) is the Gauss-Newton approximation to the Hessian $\nabla^2 \widehat{f}(u)$, provided $y = y(u)$.

The Hessian $\nabla^2 \widehat{f}(u)$ of the reduced objective can also be computed by using second-order sensitivities. In this approach one applies the chain rule to $\nabla \widehat{f}(u) = y'(u)^* \nabla_y f(y(u),u) + \nabla_u f(y(u),u)$ and one computes the second-order derivatives of $y(u)$ by applying the implicit function theorem to (12). Unlike (19) and (20), this approach avoids the explicit use of Lagrange multipliers.

We let $H(y,u,\lambda)$ be the Hessian $\nabla^2_{xx}\ell(y,u,\lambda)$ or an approximation thereof. If conjugate-gradient like methods are used to solve subproblems, then Newton-based optimization methods for (11) or reduced SQP-based optimization methods for (9) require the computation of some of the quantities

$$H(y,u,\lambda)s, \quad \langle s, H(y,u,\lambda)s\rangle_{\mathcal{X}}, \quad W(y,u)^* H(y,u,\lambda)s,$$
$$W(y,u)^* H(y,u,\lambda) W(y,u) s_u, \quad \langle s_u, W(y,u)^* H(y,u,\lambda) W(y,u) s_u\rangle_{\mathcal{U}}$$

for given $s = (s_y, s_u)$ and s_u.

Often, one does not approximate the Hessian $\nabla^2_{xx}\ell(y,u,\lambda)$, but the reduced Hessian. This is, e.g., the case if a quasi-Newton method is used to solve (11) or a reduced SQP method is used to solve (9). If $\widehat{H}(y,u) \approx W(y,u)^* \nabla^2_{xx}\ell(y,u,\lambda) W(y,u)$, then this approximation fits into the previous framework in which the full Hessian is approximated by setting

$$H(y,u,\lambda) = \begin{pmatrix} 0 & 0 \\ 0 & \widehat{H}(y,u) \end{pmatrix}. \tag{23}$$

If $H(y,u,\lambda)$ is given by (23), then the definition of $W(y,u)$ implies the equalities

$$H(y,u,\lambda)s = \begin{pmatrix} 0 \\ \widehat{H}(y,u)s_u \end{pmatrix},$$
$$\langle s, H(y,u,\lambda)s\rangle_{\mathcal{X}} = \langle s_u, \widehat{H}(y,u)s_u\rangle_{\mathcal{U}}, \quad W(y,u)^* H(y,u,\lambda)s = \widehat{H}(y,u)s_u,$$
$$W(y,u)^* H(y,u,\lambda) W(y,u) s_u = \widehat{H}(y,u)s_u,$$
$$\langle s_u, W(y,u)^* H(y,u,\lambda) W(y,u) s_u\rangle_{\mathcal{U}} = \langle s_u, \widehat{H}(y,u)s_u\rangle_{\mathcal{U}}.$$

4. USER INTERFACE

Table 1 lists the functions or subroutines that are part of the user interface. In this section, we will describe the calling sequences of these functions or subroutines using Matlab syntax. The input parameters appear in parenthesis after the name of the function or subroutine whereas the output parameters are displayed in brackets. Of course, the interface is not language specific and Matlab is used for illustration only. The main purpose is to show what information needs to be passed from the application routines to the optimizer. We do not promote a specific language for the implementation of this information transfer.

Not all interface routines listed in Table 1 are needed in the implementation of all optimization algorithms. For example, if quasi-Newton updates are used to approximate second-order derivative information, the subroutine `hs_exact` is not used and if the optimization problem formulation (1) is used, then `state` is not needed.

Table 1. User provided subroutines.

a. Adjoint and sensitivity equation approaches

`fval`	evaluate $f(y,u)$
`cval`	evaluate $c(y,u)$
`lcval`	evaluate $c_y(y,u)s_y + c_u(y,u)s_u + c(y,u)$
`state`	solve $c(y,u) = 0$ for fixed u
`linstate`	solve $c_y(y,u)s_y = -c_u(y,u)s_u - c(y,u)$
`yprod`	compute $\langle y_1, y_2\rangle_{\mathcal{Y}}$
`uprod`	compute $\langle u_1, u_2\rangle_{\mathcal{U}}$
`lprod`	compute $\langle \lambda_1, \lambda_2\rangle_{\Lambda}$
`hs_exact`	compute $\nabla^2_{xx}\ell(y,u,\lambda)s$
`xnew`	(re)activate a new iterate

b. Adjoint equation approach

`adjoint`	solve $c_y(y,u)^*\lambda = -\nabla_y f(y,u)$
`adjval`	evaluate $c_y(y,u)^*\lambda + \nabla_y f(y,u)$
`grad`	evaluate $c_u(y,u)^*\lambda + \nabla_u f(y,u)$

c. Sensitivity equation approach

`sens`	compute $S(y,u)v$
`sensa`	compute $S(y,u)^*v$
`fgrad`	compute $\nabla_y f(y,u)$ and $\nabla_u f(y,u)$

More details about the user provided subroutines will be given in the following sections. All user provided subroutines return a variable `iflag`, all user provided subroutines but `xnew` have an input parameter `tol`, and all user provided subroutines have an input parameter `user_parms`. The return variable `iflag` indicates whether the required task could be performed. On return, the `iflag` should be set as follows:

`iflag` $= 0$: The required task could be performed.

`iflag` > 0 : The required task could not be performed.

`iflag` < 0 : The required task could be performed, but the results are not ideal.

If `iflag` > 0 during the execution of the optimization algorithm, the optimization algorithm can return with an error message providing the value of `iflag` and the place in the optimization code where the error occurred. A negative value of `iflag` can be used to communicate, e.g., (near) singularity of matrices, or other potentially serious events that fall short of fatal errors. If `iflag` < 0 during the execution of the optimization algorithm, the optimization algorithm can issue a warning that contains the value of `iflag` and the place in the optimization code where the error occurred.

The input parameter `tol` can be used to control inexactness. Often in practical applications the state equation, the linearized state and the adjoint equations are solved using iterative linear system solvers. Moreover, the derivatives of f and c may be approximated by finite differences. In such situations user provided information will never be exact and an optimization algorithm has to adapt to this situation. In fact, allowing inexact, but less expensive function and derivative information could lead to more efficient optimization algorithms, provided this inexactness is controlled properly. An example are inexact Newton methods for large scale problems [Nash and Sofer 1996, Ch. 12]. The input parameter `tol` allows the optimization algorithm to control the inexactness.

Finally, all subroutines have an input parameter `user_parms` that allows to pass problem specific information such as physical parameters or weighting coefficients in the objective function. These parameters are never accessed by the optimizer. In a MATLAB implementation `user_parms` could be a structure array.

4.1 Vectors

The Hilbert spaces $\mathcal{Y}$, $\mathcal{U}$, and Λ are finite dimensional and can be identified with $\mathbb{R}^{n_y}, \mathbb{R}^{n_u}, \mathbb{R}^{n_y}$, respectively. Thus vectors in these spaces can in principle be stored as arrays. In many cases, however, other representations such as derived types in Fortran 90/95, objects in C++, or structure arrays in MATLAB might be more useful. In this case the user also has to provide functions that create and initialize a vector with zeros, create a vector and copy an existing vector into this newly created one, multiply a vector by a scalar, and add a scalar multiple of one vector to another vector. Each of these functions has to be provided for vectors in $\mathcal{Y}$, $\mathcal{U}$, and Λ, i.e., each of these functions has to be provided three times.

4.2 User provided functions used in the adjoint and sensitivity equation approaches

`fval` Given y and u evaluate $f(y,u)$. The generic function is

```
[ f, iflag ] = fval( y, u, tol, user_parms )
```

`cval` Given y and u evaluate $c(y,u)$. The generic function is

```
[ c, iflag ] = cval( y, u, tol, user_parms )
```

`lcval` Given y, u, s_y, s_u, and `tol` approximately evaluate the linearized constraints

$$c_y(y,u)\, s_y + c_u(y,u)\, s_u + c(y,u),$$

i.e., compute l_c such that

$$\left\| l_c - \Big(c_y(y,u)\, s_y + c_u(y,u)\, s_u + c(y,u) \Big) \right\|_\Lambda \le \texttt{tol}.$$

The generic function is

```
[ lc, iflag ] = lcval( y, u, sy, su, tol, user_parms )
```

`state` Given u, an initial approximation y_i, and `tol` compute an approximate solution y_s to the state equation $c(y,u) = 0$, i.e., compute y_s such that

$$\left\| c(y_s, u) \right\|_\Lambda \le \texttt{tol}.$$

The generic function is

```
[ ys, iflag ] = state( yi, u, tol, user_parms )
```

`linstate` Given y, u, s_u, c, and `tol` compute an approximate solution s_y of the linearized state equation

$$c_y(y,u)\, s_y + c_u(y,u)\, s_u + c = 0\,,$$

i.e., compute s_y such that

$$\left\| c_y(y,u)\, s_y + c_u(y,u)\, s_u + c \right\|_\Lambda \le \texttt{tol}.$$

Particular cases of the previous task are the following ones:
Given y, u, c, and `tol` compute an approximate solution s_y of the linearized state equation $c_y(y,u)\, s_y + c = 0$. Given y, u, s_u, and `tol` compute an approximate solution s_y of the linearized state equation $c_y(y,u)\, s_y + c_u(y,u)\, s_u = 0$. The generic function is

```
[ sy, iflag ] = linstate( y, u, su, c, job, tol, user_parms )
```

In an optimization algorithm `linstate` might be called with $c = 0$ or $s_u = 0$. In these cases the linearized state equation simplifies. The parameter `job` is used to communicate this to the user supplied `linstate` so that the user can take advantage of these special cases. The parameter `job` has the following meaning:
`job = 1`: Solve $c_y(y,u)s_y + c_u(y,u)s_u + c = 0$ for s_y.
`job = 2`: Solve $c_y(y,u)s_y + c = 0$ for s_y.
If `job = 2`, then `su` is a dummy variable and should not be referenced in `linstate`.

`job = 3`: Solve $c_y(y,u)s_y + c_u(y,u)s_u = 0$ for s_y.
If `job = 3`, then `c` is a dummy variable and should not be referenced in `linstate`.

`yprod` Given y_1 and y_2 evaluate the scalar product $\langle y_1, y_2 \rangle_{\mathcal{Y}}$. The generic function is `[ yp, iflag ] = yprod( y1, y2, tol, user_parms )`

`uprod` Given u_1 and u_2 evaluate the scalar product $\langle u_1, u_2 \rangle_{\mathcal{U}}$. The generic function is `[ up, iflag ] = uprod( u1, u2, tol, user_parms )`

`lprod` Given λ_1 and λ_2 evaluate the scalar product $\langle \lambda_1, \lambda_2 \rangle_{\Lambda}$. The generic function is `[ lp, iflag ] = lprod( lambda1, lambda2, tol, user_parms )`

`hs_exact` Given y, u, λ, s_y, and s_u compute the product of the Hessian of the Lagrangian $\nabla^2_{xx}\ell(y,u,\lambda)$ times the vector $s = (s_y, s_u)$. The generic function name is

```
[ hsy, hsu, iflag ]
      = hs_exact( y, u, lambda, sy, su, tol, ind, user_parms )
```

The input variables are the y-component `y`, the u-component `u`, the Lagrange multiplier `lambda`, the y- and u-component `sy` and `su` of the vector s, a dummy variable `tol` (this variable is included to make the parameter lists of the Hessian functions uniform, but is not used in this case), and an indicator `ind`:

`ind = 0`: `sy` and `su` are nonzero.

`ind = 1`: `sy` is zero. In this case the vector `sy` may never be referenced.

`ind = 2`: `su` is zero. In this case the vector `su` may never be referenced.

The return variables are the y- and the u-component `hsy` and `hsu` of $\nabla^2_{xx}\ell(y,u,\lambda)\, s$, and the error flag `iflag`.
Instead of $\nabla^2_{xx}\ell(y,u,\lambda)$, one can also use approximations of $\nabla^2_{xx}\ell(y,u,\lambda)$ such as (21). In particular, the input parameter `lambda` provided by the optimizer may not be the solution of (14) or (16) but a suitable approximation.

In many of the above interface functions, the input list contains a parameter `job`. This is included to identify special cases that in some applications may be executed more efficiently than the general task. The following interface function `xnew` is also added to allow more efficient implementations and to improve monitoring. In many applications a considerable overhead, such as the computation of stiffness matrices or the adaptation of grids is associated with function or gradient evaluations. Often, these computations only depend on the iterate (y,u). If (y,u) is unchanged, these computations do not need to be redone, regardless of how many function or derivative evaluations at this point are computed. In this case it may be desirable to do these computations only once per iterate and change these quantities only if the iterate changes. Moreover, if one knows that a certain point $x = (y,u)$ is only used temporarily, one may decide to keep the information corresponding to the point x that one will return to, rather than recomputing it when one returns. The purpose of `xnew` is to communicate the change of $x = (y,u)$ to the application. The optimization algorithm should call `xnew` whenever the argument $x = (y,u)$ changes. Another application of `xnew` is the storage of intermediate information. For example, the user may wish to record the development of iterates, or to stop the optimization algorithm and to restart it at a later time. In this situation the subroutine `xnew` can be used to store intermediate information on hard disk.

`xnew` The subroutine `xnew` activates, or reactivates an iterate. The generic function is `[ iflag ] = xnew( iter, y, u, new, user_parms )`.
After the call to `xnew` the pair (y, u) passed to `xnew` is used as the argument in all functions until the next call to `xnew`. The input parameter `new` is passed to help the user to control the action taken by `xnew`. The following is a set of possible options for this input parameter.
`new = 'init'` : Initialize with (y, u) as the current iterate. `xnew` has never been called before.
`new = 'current_it'`: (y, u) is the current iterate.
`new = 'react_it'`: (y, u) is reactivated as the current iterate.
`new = 'trial_it'`: (y, u) is a candidate for the next iterate. `xnew` has never been called with (y, u) before.
`new = 'new_it'`: (y, u) will be the next iterate. `xnew` has been called with (y, u) and option `new = 'trial_it'` before.
`new = 'temp'`: (y, u) is only used temporarily. Usually only one or two function evaluations are made with argument (y, u).
Since in `xnew` vital information, like stiffness matrices or grids, may be computed, `xnew` also returns `iflag`.

As we have mentioned before, the options for `new` depend on the particular optimization algorithm. The set of values for `new` above will be useful in a trust-region or a line-search framework [Dennis and Schnabel 1983], [Nash and Sofer 1996]. Trust-region algorithms generate steps (s_y, s_u) and evaluate functions at the trial iterate $(y + s_y, u + s_u)$ (`new = 'trial_it'`). Depending on some criteria, the trial iterate $(y + s_y, u + s_u)$ will become the new iterate (`new = 'new_it'`), or it will be rejected and (y, u) will remain the current iterate (`new = 'react_it'`). For the use of `xnew` in a simple line-search algorithm see Section 5.1. The option `new = 'temp'` will be useful, for example, in finite difference approximations.

The settings above are motivated by a trust-region algorithm. In other optimization algorithms more or fewer settings may be useful. For example, the steepest descent algorithm in Section 5.1 requires fewer settings. Therefore, the actual settings for `new` depend on the particular optimization algorithm and should be described in the documentation of each individual optimization algorithm.

4.3 User provided functions used only in the adjoint equation approach

`adjoint` Given y, u, and `tol` compute an approximate solution λ of the adjoint equation

$$c_y(y,u)^*\lambda + \nabla_y f(y,u) = 0\,,$$

i.e., compute λ such that

$$\left\| c_y(y,u)^*\lambda + \nabla_y f(y,u) \right\|_{\mathcal{Y}} \le \texttt{tol}.$$

A slightly more general task is the following:
Given y, u, f_y, and `tol` compute an approximate solution λ of the generalized

adjoint equation

$$c_y(y,u)^*\lambda + f_y = 0\,.$$

Here f_y is an arbitrary vector and not necessarily the gradient of the objective with respect to y. Since the gradient $\nabla_y f(y,u)$ often has a particular structure, e.g., has many zero entries, the equation $c_y(y,u)^*\lambda + \nabla_y f(y,u) = 0$ might be solved more efficiently than the equation $c_y(y,u)^*\lambda + f_y = 0$ with a generic vector f_y. The generic function is

```
[ lambda, iflag ] = adjoint( y, u, fy, job, tol, user_parms )
```

The parameter `job` specifies which equation has to be solved.

`job = 1`: Solve $c_y(y,u)^*\lambda + \nabla_y f(y,u) = 0$ for λ.

If `job = 1`, then `fy` is a dummy variable and should not be referenced in `adjoint`.

`job = 2`: Solve $c_y(y,u)^*\lambda + f_y = 0$ for λ.

`adjval` Given y, u, λ, and `tol` approximately evaluate the residual of the adjoint equation

$$c_y(y,u)^*\lambda + \nabla_y f(y,u)\,,$$

i.e., compute the vector a such that

$$\left\| a - \Big(c_y(y,u)^*\lambda + \nabla_y f(y,u)\Big) \right\|_{\mathcal{Y}} \le \texttt{tol}\,.$$

The generic function is

```
[ adj, iflag ] = adjval( y, u, lambda, tol, user_parms )
```

`grad` Given y, u, λ, and `tol` approximately evaluate the reduced gradient

$$c_u(y,u)^*\lambda + \nabla_u f(y,u)\,,$$

i.e., compute g such that

$$\left\| g - \Big(c_u(y,u)^*\lambda + \nabla_u f(y,u)\Big) \right\|_{\mathcal{U}} \le \texttt{tol}\,.$$

A slightly more general task is the following: Given y, u, λ, f_u, and `tol` approximately compute

$$c_u(y,u)^*\lambda + f_u\,.$$

Here f_u is an arbitrary vector and not necessarily the gradient of the objective with respect to u. Again, we distinguish between the two cases because $\nabla_u f(y,u)$ is often a very simple vector. The generic function is

```
[ g, iflag ] = grad( y, u, lambda, fu, job, tol, user_parms )
```

The parameter `job` specifies which expression has to be evaluated.

`job = 1`: Compute $c_u(y,u)^*\lambda + \nabla_u f(y,u)$.

If `job = 1`, then `fu` is a dummy variable and should not be referenced in `grad`.

`job = 2`: Compute $c_u(y,u)^*\lambda + f_u$.

4.4 User provided functions used only in the sensitivity equation approach

`fgrad` Given y, u, and `tol` compute approximate partial gradients $\nabla_y f(y,u)$ and $\nabla_u f(y,u)$ of f, i.e., compute f_y and f_u such that

$$\left\|\nabla_y f(y,u) - f_y\right\|_{\mathcal{Y}} \le \texttt{tol}, \quad \left\|\nabla_u f(y,u) - f_u\right\|_{\mathcal{U}} \le \texttt{tol}.$$

The generic function is

```
[ fy, fu, iflag ] = fgrad( y, u, job, tol, user_parms )
```

The parameter `job` specifies which partial gradient has to be computed and is included to allow the optimization algorithm to take advantage of special cases. It has the following meaning:
`job = 1`: Compute $\nabla_y f(y,u)$.
`job = 2`: Compute $\nabla_u f(y,u)$.
`job = 3`: Compute $\nabla_y f(y,u)$ and $\nabla_u f(y,u)$.

`sensa` Given y, u, and `tol` compute

$$z = c_u(y,u)^* \left(c_y(y,u)^*\right)^{-1} v$$

approximately, i.e., compute z such that

$$\left\| z - c_u(y,u)^* \left(c_y(y,u)^*\right)^{-1} v \right\|_{\mathcal{U}} \le \texttt{tol}.$$

The generic function is

```
[ z, iflag ] = sensa( y, u, v, tol, user_parms )
```

`sens` Given y, u, and `tol` compute

$$z = c_y(y,u)^{-1} c_u(y,u) v$$

approximately, i.e., compute z such that

$$\left\|c_y(y,u)z - c_u(y,u)v\right\|_{\Lambda} \le \texttt{tol} \quad \text{or} \quad \left\|z - c_y(y,u)^{-1}c_u(y,u)v\right\|_{\mathcal{Y}} \le \texttt{tol}.$$

The generic function is

```
[ z, iflag ] = sens( y, u, v, tol, user_parms )
```

4.5 Stopping criteria

The output parameter `iflag` could also be used to implement user supplied stopping tests that augment standard convergence tests based on gradient norms, function values, step norms, or iteration numbers. In addition to these standard convergence tests, the user could implement stopping criteria based on quantities computed within the user supplied subroutines and force the optimization algorithm to return by setting $\texttt{iflag} > 0$.

A sensible place to implement an application dependent stopping criterion could be in the user suplied subroutine `xnew`.

4.6 Consistency and derivative checks

For the adjoint equation approach. In exact arithmetic, the adjoints have to satisfy

$$\begin{aligned}
\langle c_y(y,u)s_y, \lambda\rangle_\Lambda &= \langle s_y, c_y(y,u)^*\lambda\rangle_\mathcal{Y}, & \forall\, s_y, \lambda,\\
\langle c_u(y,u)s_u, \lambda\rangle_\Lambda &= \langle s_u, c_u(y,u)^*\lambda\rangle_\mathcal{U}, & \forall\, s_u, \lambda,\\
\langle c_y(y,u)^{-1}c, s_y\rangle_\mathcal{Y} &= \langle c, (c_y(y,u)^{-1})^* s_y\rangle_\Lambda, & \forall\, c, s_y.
\end{aligned}$$

If inexact solvers are used with tolerances as described in the previous section, then

$$\begin{aligned}
\langle c_y(y,u)s_y, \lambda\rangle_\Lambda - \langle s_y, c_y(y,u)^*\lambda\rangle_\mathcal{Y} &= \mathcal{O}(\mathtt{tol}), & \forall\, s_y, \lambda,\\
\langle c_u(y,u)s_u, \lambda\rangle_\Lambda - \langle s_u, c_u(y,u)^*\lambda\rangle_\mathcal{U} &= \mathcal{O}(\mathtt{tol}), & \forall\, s_u, \lambda,\\
\langle c_y(y,u)^{-1}c, s_y\rangle_\mathcal{Y} - \langle c, (c_y(y,u)^{-1})^* s_y\rangle_\Lambda &= \mathcal{O}(\mathtt{tol}), & \forall\, c, s_y.
\end{aligned}$$

Derivative computations can be checked using finite differences. If only the user provided functions described in Sections 4.2 and 4.3 are to be used for these checks, then not all derivatives can be accessed. For example, $c_u(y,u)^*$ is never computed explicitly. Using the functions in Sections 4.2 and 4.3, one can perform the checks

$$\begin{aligned}
\big\| c_y(y,u)s_y - \frac{1}{\alpha}\big(c(y+\alpha s_y, u) - c(y,u)\big)\big\|_\Lambda &= \mathcal{O}(\alpha), & (24)\\
\big\| c_u(y,u)s_u - \frac{1}{\alpha}\big(c(y, u+\alpha s_u) - c(y,u)\big)\big\|_\Lambda &= \mathcal{O}(\alpha), & (25)\\
\langle c_y(y,u)^*\lambda, s_y\rangle_\mathcal{Y} - \frac{1}{\alpha}\big(\langle c(y+\alpha s_y,u),\lambda\rangle_\Lambda - \langle c(y,u),\lambda\rangle_\Lambda\big) &= \mathcal{O}(\alpha),\\
\langle c_u(y,u)^*\lambda, s_u\rangle_\mathcal{U} - \frac{1}{\alpha}\big(\langle c(y,u+\alpha s_u),\lambda\rangle_\Lambda - \langle c(y,u),\lambda\rangle_\Lambda\big) &= \mathcal{O}(\alpha),\\
\big|\langle \nabla_y f(y,u), s_y\rangle_\mathcal{Y} - \frac{1}{\alpha}\big(f(y+\alpha s_y,u) - f(y,u)\big)\big| &= \mathcal{O}(\alpha), & (26)\\
\big|\langle \nabla_u f(y,u), s_u\rangle_\mathcal{U} - \frac{1}{\alpha}\big(f(y,u+\alpha s_u) - f(y,u)\big)\big| &= \mathcal{O}(\alpha). & (27)
\end{aligned}$$

For the sensitivity equation approach. Similarly, one can check user provided information for the sensitivity equation approach. With the user provided functions described in Sections 4.2 and 4.4 one can perform the consistency check

$$\langle S(y,u)s_u, s_y\rangle_\mathcal{Y} - \langle s_u, S(y,u)^* s_y\rangle_\mathcal{U} = \mathcal{O}(\mathtt{tol}), \quad \forall\, s_y, s_u$$

and the finite difference checks (24)-(25) and (26)-(27).

5. EXAMPLES

5.1 Examples of optimization algorithms

In this section we provide code or code fragments for some optimization algorithms to illustrate the use of the interface. To keep our illustration simple, we make no use of the return flag `iflag` and simply assume that all requested operations can be performed. Moreover, we do not address the control of inaccuracy and we simply carry `tol` along without ever modifying it. What to do in an optimization algorithm if certain application information can not be computed and how to control the inexactness are important and interesting questions. The answers to these questions belong in a article on optimization algorithms and are beyond the scope of this paper. Again, we use MATLAB syntax for illustration. In particular we assume that all vectors are arrays so that we can use existing arithmetic operators for the addition of vectors and scalar multiplication (see Section 4.1).

The first example is the steepest descent method with Armijo line search rule for the solution of the reduced problem (11). Depending on whether the sensitivity equation approach or the adjoint equation approach is used the gradient is computed by (13) or by (15). In this example, u is the unknown variable and y is a function of u. As a consequence, only u is passed to `xnew` and the variable y is only used as a dummy argument.

```
  ...
% Loop k: a current iterate u is given and the corresponding
%         solution y of the state equation has been computed.
%
%  Compute the gradient W(y(u),u)*gradf(y(u),u) of the reduced
%  function.
   if der_cal == 'adjoints'
%    Solve the adjoint equation.
     [lambda, iflag] = adjoint( y, u, zeros(size(y)), 1, tol, user_parms );
%
%    Compute the reduced gradient.
     [rgrad, iflag] = grad( y, u, lambda, zeros(size(u)), 1, tol, user_parms );
   elseif der_cal == 'sensitivities'
%    Compute the gradient of f wrt y and u.
     [grady, gradu, iflag] = fgrad( y, u, 3, tol, user_parms );
%
%    Compute the reduced gradient.
     [z, iflag]  = sensa( y, u, grady, tol, user_parms );
     rgrad       = -z + gradu;
   end
%
%  Compute step size t.
   t = 1;
   [gradnrm2, iflag] = uprod( rgrad, rgrad, tol, user_parms );
   succ = 0;
   while( succ == 0 )
%     Compute trial iterate (y is a dummy variable).
      unew = u - t*rgrad;
      [iflag] = xnew( iter, y, unew, 'trial_it', user_parms );
%
%     Solve the state equation.
      [ynew, iflag] = state( y, unew, tol, user_parms );
%
%     Evaluate objective function.
      [fnew, iflag] = fval( ynew, unew, tol, user_parms );
%
%     Check step size criterion.
      if( fnew - f <= -1.e-4 * t * gradnrm2 )
          succ = 1;
      end
```

```
%
%     Reduce the step size.
      t = 0.5 * t;
   end
%
%  Set new iterate.
   y       = ynew;
   u       = unew;
   f       = fnew
   [iflag] = xnew( iter, y, u, 'new_it', user_parms );
%
%  End of loop k.
   ...
```

As our second example, we consider a simple version of a reduced SQP method with no strategy for globalization. See, e.g., [Heinkenschloss 1996, Alg. 2.1]. At a given point (y, u), the SQP method computes a solution of

$$\widehat{H}(y,u)s_u = -W(y,u)^*\nabla f(y,u),$$

where $\widehat{H}(y,u)$ is the reduced Hessian or an approximation thereof (see (20)) and then a solution of

$$c_y(y,u)s_y = -c(y,u) - c_u(y,u)s_u.$$

The following code fragment illustrates the use of the user interface to implement the reduced SQP method.

```
  ...
% A new iterate (y,u) has been computed before and xnew has been called.
%
% Compute the reduced gradient W(y,u)*gradf(y,u).
  if der_cal == 'adjoints'
%    Solve the adjoint equation.
     [lambda, iflag] = adjoint( y, u, zeros(size(y)), 1, tol, user_parms );
%
%    Compute the reduced gradient.
     [rgrad, iflag] = grad( y, u, lambda, zeros(size(u)), 1, tol, user_parms );
  elseif der_cal == 'sensitivities'
%    Compute the gradient of f wrt y and u.
     [grady, gradu, iflag] = fgrad( y, u, 3, tol, user_parms );
%
%    Compute the reduced gradient.
     [z, iflag]  = sensa( y, u, grady, tol, user_parms );
     rgrad       = -z + gradu;
  end
%
  Compute the value of c(y,u).
  [c, iflag] = cval( y, u, tol, user_parms );
%
% Compute the norms of c and rgrad squared.
  [rgradnrm2, iflag] = uprod( rgrad, rgrad, tol, user_parms );
  [cnrm2, iflag]     = lprod( c, c, tol, user_parms );
```

```
%
% Termination criterion.
  if( sqrt(rgradnrm2) < gtol & sqrt(cnrm2) < ctol )
      return
  end
%
% Compute su.
  ...
%
% Compute sy.
  [sy, iflag] = linstate( y, u, su, c, 1, tol, user_parms );
%
% Set the new iterate.
  y       = y + sy;
  u       = u + su;
  [iflag] = xnew( iter, y, u, 'current_it', user_parms );
  ...
```

One possible merit function to globalize the SQP method is the augmented Lagrangian:

$$f(y,u) + \langle \lambda(y,u), c(y,u) \rangle_\Lambda + \rho \|c(y,u)\|^2_\Lambda,$$

where ρ is a positive penalty parameter. The following code fragment describes the use of the interface to compute the value of the augmented Lagrangian function. The calculation of the scalar product $\langle \lambda(y,u), c(y,u) \rangle_\Lambda$ by the sensitivity equation approach is shown in (18).

```
  ...
  Compute the values of f(y,u) and c(y,u).
  [f, iflag] = fval( y, u, tol, user_parms );
  [c, iflag] = cval( y, u, tol, user_parms );
%
  if der_cal == 'adjoints'
%    Solve the adjoint equation.
     [lambda, iflag] = adjoint( y, u, zeros(size(y)), 1, tol, user_parms );
%
     [ctlambda, iflag] = lprod( lambda, c, tol, user_parms );
  elseif der_cal == 'sensitivities'
%    Solve the linearized state equation.
     [sy, iflag] = linstate( y, u, zeros(size(u)), c, 2, tol, user_parms );
%
%    Compute the gradient of f wrt y.
     [grady, gradu, iflag] = fgrad( y, u, 1, tol, user_parms );
%
     [ctlambda, iflag] = yprod( grady, sy, tol, user_parms );
  end
%
% Compute the norm of c squared.
  [cnrm2, iflag] = lprod( c, c, tol, user_parms );
%
% Compute the value of the augmented Lagrangian function.
  augLag = f + ctlambda + rho * cnrm2;
```

...

The next example concerns the implementation of limited memory BFGS updates for the approximation of $\nabla^2_{xx}\ell(y,u,\lambda)$. We set

$$s_i = \begin{pmatrix} (s_y)_i \\ (s_u)_i \end{pmatrix}, \quad v_i = \begin{pmatrix} (v_y)_i \\ (v_u)_i \end{pmatrix}, \quad \langle s_i, v_i\rangle_{\mathcal{X}} = \langle (s_y)_i, (v_y)_i\rangle_{\mathcal{Y}} + \langle (s_u)_i, (v_u)_i\rangle_{\mathcal{U}},$$

where $(s_y)_i = y_{i+1} - y_i$, $(s_u)_i = u_{i+1} - u_i$, $(v_y)_i = \nabla_y\ell(y_{i+1}, u_{i+1}, \lambda_{i+1}) - \nabla_y\ell(y_i, u_i, \lambda_i)$, $(v_u)_i = \nabla_u\ell(y_{i+1}, u_{i+1}, \lambda_{i+1}) - \nabla_u\ell(y_i, u_i, \lambda_i)$. If $\langle s_{k-1}, v_{k-1}\rangle_{\mathcal{X}} \neq 0$ and if the Hessian approximation H_{k-1} is invertible, then the inverse of the BFGS update is given as

$$\begin{aligned} H_k^{-1} &= (I_{n_x} - \rho_{k-1} s_{k-1} \otimes v_{k-1})\, H_{k-1}^{-1}\, (I_{n_x} - \rho_{k-1} v_{k-1} \otimes s_{k-1}) \\ &\quad + \rho_{k-1} s_{k-1} \otimes s_{k-1}, \end{aligned} \tag{28}$$

where $n_x = n_y + n_u$ and $\rho_{k-1} = 1/\langle s_{k-1}, v_{k-1}\rangle_{\mathcal{X}}$. See, e.g., [Nocedal 1980]. Given x and w, $x \otimes w$ is defined by $(x \otimes w)z = \langle w, z\rangle_{\mathcal{X}}\, x$. See the appendix.

The equation (28) leads to a limited storage BFGS (L-BFGS), by using the recursion L times and replacing H_{k-L}^{-1} by

$$H_{k-L}^{-1} \to \begin{pmatrix} \frac{1}{\gamma_y} I_{n_y} & 0 \\ 0 & \frac{1}{\gamma_u} I_{n_u} \end{pmatrix}.$$

The computation of $H_k^{-1}g$, where H_k is the L-BFGS matrix can be done in a efficient way following [Matties and Strang 1979; Nocedal 1980] or [Byrd et al. 1994]. We demonstrate the computation of $z = H_k^{-1}g$, where $g = (g_y, g_u)$ is a given vector and H_k^{-1} is the L-BFGS approximation of $\nabla^2_{xx}\ell(y,u,\lambda)$ using our interface and the recursive formula given in [Matties and Strang 1979] and [Nocedal 1980, p. 779]. The integer L denotes the number of vector pairs s_i, v_i stored. The last character in the variable name indicates whether the quantity corresponds to the $\mathcal{Y}$ space or to the $\mathcal{U}$ space. Otherwise, the naming of variables and the structure of the algorithm follows [Nocedal 1980, p. 779]. For simplicity, we assume that $k > L$.

```
...
for i = L-1:-1:0
    j = i + k - L;
    [vtsy, iflag] = yprod( vy(j), sy(j), tol, user_parms );
    [vtsu, iflag] = uprod( vu(j), su(j), tol, user_parms );
    rho(j)        = 1 / ( vtsy + vtsu );
    [gtsy, iflag] = yprod( gy, sy(j), tol, user_parms );
    [gtsu, iflag] = uprod( gu, su(j), tol, user_parms );
    alpha(i)      = ( gtsy + gtsu ) * rho(j);
    gy            = gy - alpha(i) * vy(j);
    gu            = gu - alpha(i) * vu(j);
end
gy = gy / gammay;
gu = gu / gammau;
for i = 0:L-1
    j = i + k - L;
    [gtvy, iflag] = yprod( gy, vy(j), tol, user_parms );
    [gtvu, iflag] = uprod( gu, vu(j), tol, user_parms );
```

```
        beta(i)       = ( gtvy + gtvu ) * rho(j);
        gy            = gy + ( alpha(i) - beta(i) ) * sy(j);
        gu            = gu + ( alpha(i) - beta(i) ) * su(j);
    end
    ...
```

5.2 Example of an optimal control problem

Examples that illustrate the use of this interface from an application perspective are given in [Cliff et al. 1997], [Heinkenschloss and Vicente 1999] and, with less detail, in [Cliff et al. 1998]. The numerical computations in those papers were performed using an implementation of this interface in MATLAB or Fortran 77.

We discuss the optimal control problem from [Cliff et al. 1997] in more detail. The state equations in this problem model the steady flow of an inviscid fluid in a duct. They are a simplified version of the one-dimensional Euler equations. The goal is to find a shape of the duct, represented by u, so that the flow velocity, denoted by y, matches a desired velocity indicated by the superscript d. The boundary conditions in this example are so that the flow exhibits a shock at y_s. Therefore, it is useful to split the state equation into an equation left of the shock and an equation right of the shock. In the notation of this paper the infinite dimensional problem corresponding to (1) is given by

$$\min\ f(y,u) = \frac{1}{2}\int_0^{y_s} (y_L(x) - y^d(x))^2\,dx + \frac{1}{2}\int_{y_s}^1 (y_R(x) - y^d(x))^2\,dx$$

subject to the equality constraints

$$\begin{gathered} (h(y_L))_x + g(y_L, u_L) = 0, \quad x \in [0, y_s], \\ (h(y_R))_x + g(y_R, u_R) = 0, \quad x \in [y_s, 1], \\ h(y_L(y_s)) = h(y_R(y_s)), \\ y_L(0) = y_{\text{in}}, \quad y_R(1) = y_{\text{out}}, \end{gathered}$$

and to the inequality constraints

$$0 \le y_s \le 1, \quad \underline{u} \le u_L(x), u_R(x) \le \overline{u},$$

where $y = (y_L, y_R, y_s) \in W^{1,\infty}(0,1) \times W^{1,\infty}(0,1) \times I\!R$ and $u = (u_L, u_R) \in L^2(0,1) \times L^2(0,1)$. The conditions $h(y_L(y_s)) = h(y_R(y_s))$ are the Rankine–Hugoniot conditions. Here y_L, y_R denote the velocity of the fluid left and right of the shock location y_s and u_L, u_R are related to the cross sectional area of the duct left and right of the shock. The superscript d indicates the desired velocity profile. The functions h and g are given by $h(y) = y + \bar{H}/y$, $g(y,u) = u(\bar{\gamma} y - \bar{H}/y)$ with $\bar{\gamma} = 1/6$ and $\bar{H} = 1.2$.

For the discretization of the optimal control problem we use a cell centered grid. The subinterval $[0, y_s]$ left of the shock is subdivided into N_L equidistant subintervals of length $h_L = y_s/N_L$, the subinterval $[y_s, 1]$ right of the shock is subdivided into N_R equidistant subintervals of length $h_R = (1 - y_s)/N_R$. The point x_i denotes the midpoint of the ith cell: $x_i = (i - \frac{1}{2})h_L$, $i = 1, \ldots, N_L$, $x_i = y_s + (i - \frac{1}{2} - N_L)h_R$, $i = N_L + 1, \ldots, N_L + N_R$.

The objective function is discretized using the midpoint rule which leads to the discretized objective

$$f(y,u) = \frac{1}{2}\sum_{i=1}^{N_L} h_L\,(y_i - y^d(x_i))^2 + \frac{1}{2}\sum_{i=N_L+1}^{N_L+N_R} h_R\,(y_i - y^d(x_i))^2.$$

The derivatives in the equality constraints of the infinite dimensional problem are discretized using forward and backward differences. This leads to a set of $N_L + N_R + 1$ equality constraints $c(y,u) = 0$, where

$$c_i(y,u) = \begin{cases} N_L\,(h(y_i) - h(y_{i-1})) + y_s\, g(y_i,u_i) & i = 1,\dots,N_L, \\ N_R\,(-h(y_{i+1}) + h(y_i)) + (y_s - 1)\, g(y_i,u_i) & i = N_L+1,\dots,N_L+N_R, \\ h(y_{N_L}) - h(y_{N_L+1}) & i = N_L + N_R + 1 \end{cases}$$

and $y_0 = y_{\text{in}}$, $y_{N_L+N_R+1} = y_{\text{out}}$. The discretized states are $y = (y_1,\dots,y_{N_L+N_R},y_s)^\top$ and the discretized controls are $u = (u_1,\dots,u_{N_L+N_R})^\top$. We will use the notation $\lambda = (\lambda_1,\dots,\lambda_{N_L+N_R},\lambda_s)^\top$ for the adjoint variables.

We now discuss a few interface functions in more detail.

5.2.1 *Linearized state equation.* The partial Jacobian $c_y(y,u)$ is a bordered matrix given by

$$c_y(y,u) = \begin{pmatrix} B_L(y,u) & 0 & e_L(y,u) \\ 0 & B_R(y,u) & e_R(y,u) \\ d_L(y)^\top & d_R(y)^\top & 0 \end{pmatrix},$$

where $B_L(y,u) \in I\!\!R^{N_L \times N_L}$ is a lower bidiagonal matrix, $B_R(y,u) \in I\!\!R^{N_R \times N_R}$ is a upper bidiagonal matrix, and $e_L(y,u), d_L(y) \in I\!\!R^{N_L}$, $e_R(y,u), d_R(y) \in I\!\!R^{N_R}$. The partial Jacobian

$$c_u(y,u) = \begin{pmatrix} D_L(y,u) & 0 \\ 0 & D_R(y,u) \\ 0 & 0 \end{pmatrix},$$

is a $(N_L + N_R + 1) \times (N_L + N_R)$ 'diagonal' matrix with diagonal entries given by $(D_L(y,u))_{ii} = y_s\, g_u(y_i,u_i)$, $i = 1,\dots,N_L$, and $(D_R(y,u))_{ii} = (y_s - 1)\, g_u(y_i,u_i)$, $i = N_L+1,\dots,N_L+N_R$.

Given c, y, u and s_u, `linstate` requires the solution of $c_y(y,u)s_y = -c_u(y,u)s_u - c$. The structure of $c_y(y,u)$ can be used to solve this system using a Schur complement approach.

5.2.2 *Scalar products.* Even though this discretized optimal control problem is a rather small dimensional nonlinear programming problem, the computations in [Cliff et al. 1997] have shown that the choice of the scalar product can noticeably influence the performance of an optimization algorithm and the quality of the solution. For this problem suitable scalar products are

$$\begin{aligned} \langle y_1, y_2\rangle_{\mathcal{Y}} &= \textstyle\sum_{i=1}^{N_L} \frac{1}{N_L}\,(y_1)_i(y_2)_i + \sum_{i=N_L+1}^{N_L+N_R} \frac{1}{N_R}\,(y_1)_i(y_2)_i + (y_1)_s(y_2)_s, \\ \langle u_1, u_2\rangle_{\mathcal{U}} &= \textstyle\sum_{i=1}^{N_L} \frac{1}{N_L}\,(u_1)_i(u_2)_i + \sum_{i=N_L+1}^{N_L+N_R} \frac{1}{N_R}\,(u_1)_i(u_2)_i, \\ \langle \lambda_1, \lambda_2\rangle_{\Lambda} &= \textstyle\sum_{i=1}^{N_L} \frac{1}{N_L}\,(\lambda_1)_i(\lambda_2)_i + \sum_{i=N_L+1}^{N_L+N_R} \frac{1}{N_R}\,(\lambda_1)_i(\lambda_2)_i + (\lambda_1)_s(\lambda_2)_s. \end{aligned}$$

Given y_1 and y_2, `yprod` requires the evaluation of $\langle y_1, y_2\rangle_{\mathcal{Y}}$. The interface functions `uprod` and `lprod` are defined analogously.

5.2.3 *Adjoint equation.* The adjoint c_y^* of the partial Jacobian c_y depends on the scalar products $\langle\cdot,\cdot\rangle_{\mathcal{Y}}$, $\langle\cdot,\cdot\rangle_\Lambda$ (see Appendix A). For the scalar products specified above we find that

$$c_y^*(y,u) = \begin{pmatrix} B_L(y,u)^\top & 0 & \tilde{d}_L(y) \\ 0 & B_R(y,u)^\top & \tilde{d}_R(y) \\ \tilde{e}_L(y,u)^\top & \tilde{e}_R(y,u)^\top & 0 \end{pmatrix},$$

where $\tilde{d}_L(y) = N_L\; d_L(y)$, $\tilde{d}_R(y) = N_R\; d_R(y)$, $\tilde{e}_L(y,u) = e_L(y,u)/N_L$, $\tilde{e}_R(y,u) = e_R(y,u)/N_R$.

Given y, u, the interface function `adjoint` requires the solution of $c_y^*(y,u)\lambda = -\nabla_y f(y,u)$, if `job` $=$ `1`. Here $\nabla_y f(y,u)$ is the partial gradient of the objective function with respect to y. This partial gradient depends on the scalar product $\langle\cdot,\cdot\rangle_{\mathcal{Y}}$ (see Section 2). In our case it is given by

$$\begin{aligned} \nabla_y f(y,u) \;=\; & \Big(y_s(y_1 - y^d(x_1)), \ldots, y_s(y_{N_L} - y^d(x_{N_L})), \\ & (1-y_s)(y_{N_L+1} - y^d(x_{N_L+1})), \ldots, (1-y_s)(y_{N_L+N_R} - y^d(x_{N_L+N_R})), \\ & \partial_{y_s} f(y,u) \Big)^\top . \end{aligned}$$

The partial derivative $\partial_{y_s} f(y,u)$ is a lengthy expression because all grid points x_i depend on the shock location y_s. We omit it here. If `job` $=$ `2`, then we have also given f_y and `adjoint` requires the solution of $c_y^*(y,u)\lambda = -f_y$. In both cases we can use a Schur complement approach to solve the system.

6. LIMITATIONS AND EXTENSIONS

In the previous section we have illustrated how some optimization tasks can be implemented using our interface. We have used our interface to implement a class of affine-scaling interior-point optimization algorithms [Dennis et al. 1998] for the solution of

$$\begin{aligned} \min\;\; & f(y,u) \\ \text{s.t.}\;\; & c(y,u) = 0, \\ & \underline{u} \le u \le \overline{u} \end{aligned} \tag{29}$$

in Fortran 77 and Matlab . However, our interface is certainly not sufficient to implement all optimization algorithms for the solution of (29) or the more complicated problem (1). For example, the types of constraints may limit the applicability of the interface. In particular the presence of inequality constraints poses interesting questions. For example, for infinite dimensional problems of the type (29) with controls u in L^p, $p \ge 1$, the inequality constraints are point-wise constraints and are associated with the Banach space L^∞. The use of the Hilbert space $\mathcal{U} = L^2$ in this context seems questionable. We have obtained quite good numerical results with our algorithms in [Dennis et al. 1998] for solving (29) if in this situation we select $\mathcal{U} = L^2$. These numerical observations are supported by the theory in [Ulbrich et al. 1997]. In general, however, the pure Hilbert space structure underlying our interface (and others) does not seem sufficient. If one assumes that vectors are stored as arrays, the vectors must be small enough so that they can be held in-core. This is problematic for problems with time-dependent partial differential equations or problems with large data sets such as those arising in seismic inversion. Sometimes such optimization problems can be reformulated by elimination of part of the constraints $c(y,u)=0$ so that the resulting problem is significantly smaller. Additionally, functions like those implemented in HCL [Gockenbach et al. 1997] are needed to accomplish tasks like vector additions, if vectors cannot be stored in-core, but have to be stored on, say, hard disk. See also Section 4.1.

Besides the above mentioned limitations, we believe the interface presented in this paper is very useful. It can be used to implement a large number of algorithms for a significant class of optimal control problems. For instance, any problem of the form

$$\begin{aligned} \min \ & f(w,u) \\ \text{s.t. } & d(w,u)=0, \quad g(w,u)\geq 0, \\ & \underline{w}\leq w\leq \overline{w}, \\ & \underline{u}\leq u\leq \overline{u} \end{aligned}$$

can be reformulated as problem (1) by setting $y=(w,s)$ with $g(w,u)-s=0$. In this case the nonsingularity of $d_w(w,u)$ would imply the nonsingularity of $c_y(y,u)$.

We expect that the functions in this interface will be contained in interfaces developed to handle the very large scale problems mentioned above. The interface serves an important theoretical purpose in the use of structure for algorithmic design. By using this interface or some of its features, optimization algorithm designers are forced to separate optimization and application tasks within the algorithms.

ACKNOWLEDGMENTS

The authors greatfully acknowledge the careful reading and the constructive comments of the referees.

APPENDIX

A. SCALING OF THE PROBLEM

We continue the discussion in Section 2 and describe the influence of the scalar products on the computation of Hessians, adjoints, and quasi-Newton updates.

Influence of the scalar products on derivative computations. The partial Hessians are defined by

$$\begin{aligned} \lim_{h_y\to 0}\|\nabla_y f(y+h_y,u)-\nabla_y f(y,u)-\nabla^2_{yy}f(y,u)h_y\|_{\mathcal{Y}}/\|h_y\|_{\mathcal{Y}} &= 0, \\ \lim_{h_u\to 0}\|\nabla_y f(y,u+h_u)-\nabla_y f(y,u)-\nabla^2_{yu}f(y,u)h_u\|_{\mathcal{Y}}/\|h_u\|_{\mathcal{U}} &= 0, \\ \lim_{h_y\to 0}\|\nabla_u f(y+h_y,u)-\nabla_u f(y,u)-\nabla^2_{uy}f(y,u)h_y\|_{\mathcal{U}}/\|h_y\|_{\mathcal{Y}} &= 0, \\ \lim_{h_u\to 0}\|\nabla_u f(y,u+h_u)-\nabla_u f(y,u)-\nabla^2_{uu}f(y,u)h_u\|_{\mathcal{U}}/\|h_u\|_{\mathcal{U}} &= 0. \end{aligned} \tag{30}$$

The partial derivatives of c are defined by

$$\begin{aligned} \lim_{h_y\to 0}\|c(y+h_y,u)-c(y,u)-c_y(y,u)h_y\|_{\mathcal{Y}}/\|h_y\|_{\mathcal{Y}} &= 0, \\ \lim_{h_u\to 0}\|c(y,u+h_u)-c(y,u)-c_u(y,u)h_u\|_{\mathcal{U}}/\|h_u\|_{\mathcal{U}} &= 0. \end{aligned} \tag{31}$$

Because of the equivalency of norms in finite dimensions the Hessians are the first-order partial derivatives of the gradients (which depend on the scalar product) and the partial Jacobians of c are the matrices of first-order partial derivatives. Thus, the choice of the scalar product does not influence (30) and (31) directly. They are important, however, when inexact derivative information is allowed. Inexactness has to be measured in the appropriate norm.

If the scalar products are given by (4), (5), then the Hessians are given by

$$\nabla^2_{yy} f(y,u) = T_y^{-1} {}_{\mathrm{e}}\nabla^2_{yy} f(y,u), \quad \nabla^2_{yu} f(y,u) = T_y^{-1} {}_{\mathrm{e}}\nabla^2_{yu} f(y,u),$$
$$\nabla^2_{uy} f(y,u) = T_u^{-1} {}_{\mathrm{e}}\nabla^2_{uy} f(y,u), \quad \nabla^2_{uu} f(y,u) = T_u^{-1} {}_{\mathrm{e}}\nabla^2_{uu} f(y,u),$$

where ${}_{\mathrm{e}}\nabla^2$ is used to denote the matrices of second-order partial derivatives. Note that the partial Hessians $\nabla^2_{yy} f(y,u)$ and $\nabla^2_{uu} f(y,u)$ are symmetric with respect to the scalar products (4) and (5), respectively, and that $\langle \nabla^2_{yu} f(y,u) w, v\rangle_{\mathcal{Y}} = \langle w, \nabla^2_{uy} f(y,u) v\rangle_{\mathcal{U}}$. See also the following discussion on adjoints.

Influence of the scalar products on adjoint computations. The adjoints of c_y and c_u are defined by the relations

$$\begin{aligned} \langle c_y(y,u)^*\lambda, v\rangle_{\mathcal{Y}} &= \langle \lambda, c_y(y,u)v\rangle_\Lambda \quad \forall \lambda, v, \\ \langle c_u(y,u)^*\lambda, w\rangle_{\mathcal{U}} &= \langle \lambda, c_u(y,u)w\rangle_\Lambda \quad \forall \lambda, w. \end{aligned} \tag{32}$$

With the scalar products (4), (5), (6), and the adjoint relations (32) we find that

$$\langle \lambda, c_y(y,u)v\rangle_\Lambda = \lambda^\top T_\lambda c_y(y,u) v = (T_y^{-1} c_y(y,u)^\top T_\lambda \lambda)^\top T_y v = \langle c_y(y,u)^*\lambda, v\rangle_{\mathcal{Y}} \quad \forall \lambda, v.$$

Thus

$$c_y(y,u)^* = T_y^{-1} c_y(y,u)^\top T_\lambda.$$

Similarly,

$$c_u(y,u)^* = T_u^{-1} c_u(y,u)^\top T_\lambda.$$

Influence of the scalar products on quasi-Newton updates. Given u and v in $\mathcal{U}$, we define the linear operator $u \otimes v$ on $\mathcal{U}$ by $(u \otimes v)w = (\langle v, w\rangle_{\mathcal{U}})\, u$. Thus, if $\langle v, w\rangle_{\mathcal{U}} = v^\top w$, then $u \otimes v = uv^\top$. If $\langle v, w\rangle_{\mathcal{U}} = v^\top T_u w$ with T_u symmetric positive definite, then $u \otimes v = uv^\top T_u$.

We consider the BFGS update (see [Dennis and Schnabel 1983, Ch. 9] or [Gruver and Sachs 1980]) in the u component to illustrate the influence of this scaling onto the quasi-Newton update. We assume that y is fixed. The BFGS update is given by

$$H_+ = H + \frac{(\nabla_u f(y,u_+) - \nabla_u f(y,u)) \otimes (\nabla_u f(y,u_+) - \nabla_u f(y,u))}{\langle \nabla_u f(y,u_+) - \nabla_u f(y,u), s\rangle_{\mathcal{U}}} - \frac{Hs \otimes Hs}{\langle Hs, s\rangle_{\mathcal{U}}}.$$

If $\langle v, w\rangle_{\mathcal{U}} = v^\top w$, then $\nabla_u f(y,u) = {}_{\mathrm{e}}\nabla_u f(y,u)$ and we obtain the standard BFGS update [Dennis and Schnabel 1983, Ch. 9]. If $\langle v, w\rangle_{\mathcal{U}} = v^\top T_u w$, then $\nabla_u f(y,u) = T_u^{-1} {}_{\mathrm{e}}\nabla_u f(y,u)$ and

$$\begin{aligned} H_+ &= H + \frac{T_u^{-1}({}_{\mathrm{e}}\nabla_u f(y,u_+) - {}_{\mathrm{e}}\nabla_u f(y,u))({}_{\mathrm{e}}\nabla_u f(y,u_+) - {}_{\mathrm{e}}\nabla_u f(y,u))^\top}{({}_{\mathrm{e}}\nabla_u f(y,u_+) - {}_{\mathrm{e}}\nabla_u f(y,u))^\top s} \\ &\quad - \frac{Hs(T_u Hs)^\top}{s^\top T_u H s}. \end{aligned}$$

Influence of the scalar products on Krylov subspace methods. The use of weighted scalar products in conjugate-gradient methods is equivalent to a preconditioning with the inverse of the weighting matrix. This is described, e.g., in the work by Axelsson [1994, Sec. 11.2.6] or Gutknecht [1993].

REFERENCES

Axelsson, O. 1994. *Iterative Solution Methods.* Cambridge University Press, Cambridge, London, New York.

Betts, J. T. 1997. SOCS sparse optimal control software. Technical report, The Boeing Company, P.O. Box 3707, M/S 7L-21, Seattle, WA 98124-2207.

Borggaard, J. and Burns, J. 1997. A PDE sensitivity equation method for optimal aerodynamic design. *Journal of Computational Physics*, 366–384.

Borggaard, J., Burns, J., Cliff, E., and Schreck, S. Eds. 1998. *Computational Methods for Optimal Design. Proceedings of the AFSOR Workshop on Optimal Design and Control, Arlington, VA, 30-September – 3-October 1997*, Progress in Systems and Control Theory (Basel, Boston, Berlin, 1998). Birkhäuser Verlag.

Byrd, R. H., Nocedal, J., and Schnabel, R. B. 1994. Representations of quasi-Newton matrices and their use in limited memory methods. *Math. Programming 63*, 129–156.

Chen, Z. and Hoffmann, K.-H. 1991. Numerical solutions of the optimal control problem governed by a phase field model. In W. Desch, F. Kappel, and K. Kunisch Eds., *Optimal Control of Partial Differential Equations* (Basel, Boston, Berlin, 1991). Birkhäuser Verlag.

Cliff, E. M., Heinkenschloss, M., and Shenoy, A. 1997. An optimal control problem for flows with discontinuities. *Journal of Optimization Theory and Applications 94*, 273–309.

Cliff, E. M., Heinkenschloss, M., and Shenoy, A. 1998. Airfoil design by an all–at–once method. *International Journal for Computational Fluid Mechanics 11*, 3–25.

Conn, A. R., Gould, N. I. M., and Toint, P. L. 1992. *LANCELOT: A FORTRAN package for large scale nonlinear optimization with simple bounds.* Springer Series in Computational Mathematics, Vol. 17. Springer Verlag, Berlin, Heidelberg, New York.

Dennis, J. E., Heinkenschloss, M., and Vicente, L. N. 1998. Trust–region interior–point algorithms for a class of nonlinear programming problems. *SIAM J. Control Optim. 36*, 1750–1794.

Dennis, J. E. and Schnabel, R. B. 1983. *Numerical Methods for Nonlinear Equations and Unconstrained Optimization.* Prentice-Hall, Englewood Cliffs, N. J. Republished by SIAM, Philadelphia, 1996.

Fletcher, R. 1987. *Practical Methods of Optimization* (Second ed.). John Wiley & Sons, Chichester.

Friedman, A. and Hu, B. 1998. Optimal control of a chemical vapor deposition reactor. *J. of Optimization Theory and Applications 97*, 623–644.

Gill, P. E., Murray, W., and Saunders, M. A. 1997. SNOPT 5.3: A Fortran package for large–scale nonlinear programming. Numerical Analysis Report 97–5, Department of Mathematics, University of California, San Diego, La Jolla, CA.

Gockenbach, M. S., Petro, M. J., and Symes, W. W. 1997. C++ classes for linking optimization with complex simulation. To appear in *ACM Transactions on Mathematical Software*. `http://www.trip.caam.rice.edu/ txt/tripinfo/abstracts_list.html`.

Gockenbach, M. S. and Symes, W. W. 1997. The Hilbert class library. `http://www.trip.caam.rice.edu/ txt/hcldoc/html/index.html`.

Gruver, W. A. and Sachs, E. W. 1980. *Algorithmic Methods In Optimal Control.* Pitman, London.

Gunzburger, M. D., Hou, L. S., and Svobotny, T. P. 1993. Optimal control and optimization of viscous, incompressible flows. In M. D. Gunzburger and R. A. Nicolaides Eds., *Incompressible Computational Fluid Dynamics* (Cambridge, New York, 1993), pp. 109–150. Cambridge University Press.

GUTKNECHT, M. H. 1993. Changing the norm in conjugate gradient type algorithms. *SIAM J. Numer. Analysis 30*, 40–56.

HANDAGAMA, N. AND LENHART, S. 1998. Optimal control of a PDE/ODE system modeling a gas-phase bioreactor. In M. A. HORN, G. SIMONETT, AND G. WEBB Eds., *Mathematical Models in Medical and Health Sciences* (Nashville, TN, 1998). Vanderbilt University Press.

HEINKENSCHLOSS, M. 1996. Projected sequential quadratic programming methods. *SIAM J. Optim. 6*, 373–417.

HEINKENSCHLOSS, M. AND VICENTE, L. N. 1999. Numerical solution of semielliptic optimal control problems using SQP based optimization algorithms. Technical report, Department of Computational and Applied Mathematics, Rice University. In preparation.

ITO, K. AND KUNISCH, K. 1996. Augmented Lagrangian-SQP methods for nonlinear optimal control problems of tracking type. *SIAM J. Control and Optimization 34*, 874–891.

KUPFER, F.-S. AND SACHS, E. W. 1992. Numerical solution of a nonlinear parabolic control problem by a reduced SQP method. *Comput. Optim. and Appl. 1*, 113–135.

LIONS, J. L. 1971. *Optimal Control of Systems Governed by Partial Differential Equations.* Springer Verlag, Berlin, Heidelberg, New York.

MATTIES, H. AND STRANG, G. 1979. The solution of nonlinear finite element equations. *Internat. J. Numer. Methods Engrg. 14*, 1613–1626.

MITTELMANN, H. D. AND MAURER, H. 1998. Interior-point methods for solving elliptic control problems with control and state constraints: boundary and distributed control. Technical report, Department of Mathematics, Arizona State University. `http://plato.la.asu.edu/papers.html`.

NASH, S. G. AND SOFER, A. 1996. *Linear and Nonlinear Programming.* McGraw-Hill, New York.

NEITTAANMÄKI, P. AND TIBA, D. 1994. *Optimal Control of Nonlinear Parabolic Systems. Theory, Algorithms, and Applications.* Marcel Dekker, New York, Basel, Hong Kong.

NOCEDAL, J. 1980. Updating quasi–Newton matrices with limited storage. *Math. Comp. 35*, 773–782.

PETZOLD, L., ROSEN, J. B., GILL, P. E., JAY, L. O., AND PARK, K. 1997. Numerical optimal control of parabolic PDEs using DASOPT. In L. BIEGLER, T. COLEMAN, A. CONN, AND F. SANTOSA Eds., *Large Scale Optimization with Applications, Part II: Optimal Design and Control*, IMA Volumes in Mathematics and its Applications, Vol. 93 (Berlin, Heidelberg, New-York, 1997), pp. 271–300. Springer Verlag.

SCHULZ, V. 1997. Solving discretized optimization problems by partially reduced SQP methods. *Computing and Visualization in Science 1*, 2, 83–96.

SHANNO, D. AND VANDERBEI, R. J. 1997. An inter-point methods for nonconvex nonlinear programming. SOR97–21, Statistics and Operations Research, Princeton University, Princeton, NJ 08544. To appear in Computational Optimization and Applications.

ULBRICH, M., ULBRICH, S., AND HEINKENSCHLOSS, M. 1997. Global convergence of trust-region interior-point algorithms for infinite-dimensional nonconvex minimization subject to pointwise bounds. Technical Report TR-97-04, Department of Computational and Applied Mathematics, Rice University. To appear in SIAM J. Control and Optimization.

VANDERBEI, R. J. 1998. LOQO user's manual version 3.10. SOR97–8, Statistics and Operations Research, Princeton University, Princeton, NJ 08544. To appear in Optimization Methods and Software, `http://www.princeton.edu/~rvdb`.

VARVAREZOS, D. K., BIEGLER, L. T., AND GROSSMANN, I. E. 1994. Multiperiod design optimization with SQP decomposition. *Computers Chem. Engng. 18*, 579–595.