DEPARTMENT OF COMPUTATIONAL AND APPLIED MATHEMATICS
RICE UNIVERSITY
TECHNICAL REPORT TR08-05
(REVISED JUNE 26, 2012, AND JAN. 16, 2013)

NUMERICAL SOLUTION OF IMPLICITLY CONSTRAINED OPTIMIZATION PROBLEMS *

MATTHIAS HEINKENSCHLOSS †

Abstract. Many applications require the minimization of a smooth function $\widehat{f}:\mathbb{R}^{n_u}\to\mathbb{R}$ whose evaluation requires the solution of a system of nonlinear equations. This system represents a numerical simulation that must be run to evaluate $\widehat{f}$. This system of nonlinear equations is referred to as an implicit constraint.

In principle $\widehat{f}$ can be minimized using the steepest descent method or Newton-type methods for unconstrained minimization. However, for the practical application of derivative based methods for the minimization of $\widehat{f}$ one has to deal with many interesting issues that arise out of the presence of the implicit constraints that must be solved to evaluate $\widehat{f}$. This article studies some of these issues, ranging from sensitivity and adjoint techniques for derivative computation to implementation issues in Newton-type methods. A discretized optimal control problem governed by the unsteady Burgers equation is used to illustrate the ideas.

The material in this article is accessible to anyone with knowledge of Newton-type methods for finite dimensional unconstrained optimization. Many of the concepts discussed in this article extend to and are used in areas such as optimal control and PDE constrained optimization.

Key words. Unconstrained minimization, implicit constraints, adjoints, sensitivities, Newton's method, nonlinear programming, optimal control, Burgers equation.

AMS subject classifications. 49M37, 65K05, 90C53, 90C55

1. Introducton.

We are interested in the solution of

$$\min_{u\in U}\widehat{f}(u), \tag{1.1}$$

where U is a closed convex subset of $\mathbb{R}^{n_u}$, such as $U=\mathbb{R}^{n_u}$ or $U=[-1,1]^{n_u}$, and $\widehat{f}:U\to\mathbb{R}$ is a smooth function. The numerical solution of (2.1) using gradient-based and Newton-type methods is discussed in most courses on Numerical Analysis (at least for the case $U=\mathbb{R}^{n_u}$) and in courses on Optimization. Many textbooks such as [12, 22, 26] provide an excellent introduction into these methods. We investigate their application in the case where the evaluation of objective function $\widehat{f}$ requires the solution of a system of nonlinear equations. This situation arises in many science and engineering applications in which the evaluation of the objective function involves a simulation. We refer to the system of nonlinear equations (the simulation) as an implicit constraint. In theory standard optimization algorithms, such as those discussed in the textbooks [12, 22, 26] can be applied to the solution of (1.1). However, the practical application of these methods quickly leads to interesting questions related to

- gradient and Hessian computations for objective functions $\widehat{f}$ whose evaluation involves the solution of an implicit constraint,
- software design issues arising in the implementation of gradient based methods for the solution of (1.1),
- development of optimization algorithms for problems with inexact function and derivative information.

*This research was supported in part by NSF grant DMS-0511624 and AFOSR grant FA9550-06-1-0245

†Department of Computational and Applied Mathematics, MS-134, Rice University, 6100 Main Street, Houston, TX 77005-1892 (heinken@rice.edu).

In this paper we will formulate these questions more carefully and explore answers.

As an example for an optimization problem (1.1) in which the evaluation of the objective function requires the solution of a system of nonlinear equations, consider the design of an airplane wing. Assume that we want to determine the shape of the wing to optimize the performance of the aircraft. We measure the performance of the aircraft by the ratio of lift over drag. If we assume that the wing can be represented using a combination of surfaces parameterized by $u = (u_1, \ldots, u_{n_u})$, then we arrive at an optimization problem in u. We want to find the shape of the wing represented by u such that the ratio of lift over drag is maximized. Of course, we can convert the maximization problems into a minimization problem by minimizing the negative ratio of lift over drag. However, to compute the ratio of lift over drag for a given wing shape specified by u, we need to solve a complex system of differential equations, the Navier-Stokes equations, to obtain the velocity and pressure of the air flowing around the aircraft. From the velocity and pressure we are then able to compute lift and drag. Thus the optimization problem which is of the form (1.1) and which on the surface looks relatively simple is actually quite complicated because of the simulation, here the solution of the Navier-Stokes equations, required to evaluate the objective function.

How does the presence of the simulation required for the evaluation of the objective function (1.1) impact the application of gradient based optimization algorithms for the solution of (1.1)? We will explore answers to this question in a less complicated situation than that of the wing design problem described before. We assume that the simulation by a system of nonlinear algebraic equations. This setting allows us to explore the solution of (1.1) using basic results from real analysis, such the implicit function theorem and basic numerical optimization methods, such as the steepest descent method or Newton's method. In many applications, the simulation is described by a systems of (partial) differential equations. After discretization of the differential equations, one obtains a system of (nonlinear) algebraic equations and the setting of this paper can be applied. Additionally this setting exposes us to many concepts that one also counters in, e.g., optimal control problems and optimal design problems governed by (partial) differential equations.

The Matlab codes used to solve the examples in this paper can be downloaded from

http://www.caam.rice.edu/∼heinken/software

2. Problem Formulation. We are interested in optimization problems (2.1) in which the evaluation of $\widehat{f}$ requires the solution of a system of nonlinear equations. More precisely, we assume that

$$\widehat{f}(u) = f(y(u), u), \tag{2.1}$$

where $y(u) \in \mathbb{R}^{n_y}$ is the solution of an equation

$$c(y, u) = 0. \tag{2.2}$$

Here

$$f : \mathbb{R}^{n_y \times n_u} \to \mathbb{R}, \quad c : \mathbb{R}^{n_y \times n_u} \to \mathbb{R}^{n_y}$$

are given functions.

To distinguish between the implicit function which is defined as the solution of (2.2) and a vector in $\mathbb{R}^{n_y}$, we use the notation $y(\cdot)$ to denote the implicit function and y to denote a vector in $\mathbb{R}^{n_y}$. Furthermore, we use subscripts y and u to denote partial derivatives. For example $c_y(y, u) \in \mathbb{R}^{n_y \times n_y}$ is the partial Jacobian of the function c with respect to y and $\nabla_u f(y, u) \in \mathbb{R}^{n_u}$ is the partial gradient of the function f with respect to u

To make the formulation (1.1), (2.1), (2.2) rigorous, we make the following assumptions.

ASSUMPTION 2.1.

- *For all $u \in U$ there exists a unique $y \in \mathbb{R}^{n_y}$ such that $c(y,u) = 0$.*
- *There exists an open set $D \subset \mathbb{R}^{n_y \times n_u}$ with $\{(y,u) : u \in U,\ c(y,u) = 0\} \subset D$ such that f and c are twice continuously differentiable on D.*
- *The inverse $c_y(y,u)^{-1}$ exists for all $(y,u) \in \{(y,u) : u \in U,\ c(y,u) = 0\}$.*

Under these assumptions there exists a twice continuously differentiable function

$$y : \mathbb{R}^{n_u} \to \mathbb{R}^{n_y}$$

defined by

$$c(y(u),u) = 0.$$

Note that our Assumptions 2.1 are stronger than those required in the implicit function theorem. The standard assumptions of the implicit function theorem, only guarantee the local existence of the implicit function $y(\cdot)$ and the differentiability of this function.

We call (1.1), (2.1), (2.2) an implicitly constrained optimization problem because the solution of (2.2) is invisible to the optimization algorithm. Of course, in principle one can formulate (1.1), (2.1), (2.2) as an equality constrained optimization problem. In fact, since y is tied to u via the implicit equation (2.2), we could just include this equation into the problem formulation and reformulate (1.1), (2.1), (2.2) as

$$\begin{array}{ll} \min & f(y,u), \\ \text{s.t.} & c(y,u) = 0, \\ & u \in U. \end{array} \tag{2.3}$$

In (2.3), the optimization variables are $y \in \mathbb{R}^{n_y}$ and $u \in \mathbb{R}^{n_u}$. The formulation (2.3) can have significant advantages over (1.1), (2.1), (2.2), but in many applications the formulation of the optimization problem as a constrained problem may not be possible, for example, because of the huge size of y, which in applications can easily be many millions. We will return to the issue of solving the implicitly constrained problem (1.1), (2.1), (2.2) versus the solving the constrained problem (2.3) later. First, we focus on the solution of (1.1), (2.1), (2.2).

There are many algorithms for the solution of (1.1). See, e.g., the textbooks [7, 12, 22, 26]. We state a simple version of the Newton-Conjugate Gradient method for solving (1.1) with $U = \mathbb{R}^{n_u}$. The Newton equation $\nabla^2 \widehat{f}(u_k) s_k = -\nabla \widehat{f}(u_k)$ is solved approximately using the conjugate gradient (CG) method. The CG method is truncated if the Newton system residual is sufficiently small, more precisely

$$\|\nabla^2 \widehat{f}(u_k) s_k + \nabla \widehat{f}(u_k)\|_2 \leq \eta_k \|\nabla \widehat{f}(u_k)\|_2,$$

$\eta_k \in (0,1)$, or if a direction of negative curvature is detected. Once the direction s_k is computed, a simple Armijo line-search procedure is used to compute the step-size α_k. See, e.g., [22, 26] for more details.

ALGORITHM 2.2 (Newton-CG Method with Armijo Line-Search).

1. *Given u_0 and* gtol > 0. *Set $k = 0$.*
2. *Compute $\nabla \widehat{f}(u_k)$.*
3. *If $\|\nabla \widehat{f}(u_k)\| <$* gtol *stop.*
4. *Compute $\nabla^2 \widehat{f}(u_k)$.*
5. *Apply the CG method to compute an approximate solution of the Newton equation $\nabla^2 \widehat{f}(u_k) s_k = -\nabla \widehat{f}(u_k)$ (we use i as the iteration index in the CG method):*

5.1. *Set* $\eta_k \in (0,1)$, $s_k = 0$ *and* $p_{k,0} = r_{k,0} = -\nabla \widehat{f}(u_k)$.
5.2. *For* $i = 0,1,2,\ldots$ *do*
i. *If* $\|r_{k,i}\|_2 < \eta_k \|r_{k0}\|_2$ *goto 5.3.*
ii. *Compute* $q_{k,i} = \nabla^2 \widehat{f}(u_k) p_i$.
iii. *If* $p_{k,i}^T q_{k,i} < 0$ *goto 5.3.*
iv. $\gamma_{k,i} = \|r_{k,i}\|^2 / p_{k,i}^T q_{k,i}$.
v. $s_k = s_k + \gamma_{k,i} p_{k,i}$.
vi. $r_{k,i+1} = r_{k,i} - \gamma_{k,i} q_{k,i}$.
vii. $\beta_{k,i} = \|r_{k,i+1}\|^2 / \|r_{k,i}\|^2$.
viii. $p_{k,i+1} = r_{k,i+1} + \beta_{k,i} p_{k,i}$.
5.3. *If* $i = 0$ *set* $s_k = -\nabla \widehat{f}(u_k)$.
6. *Perform Armijo line-search.*
6.1. *Set* $\alpha_k = 1$ *and evaluate* $f(u_k + \alpha_k s_k)$.
6.2. *While* $f(u_k + \alpha_k s_k) > f(u_k) + 10^{-4} \alpha_k s_k^T \nabla \widehat{f}(u_k)$ *do*
i. *Set* $\alpha_k = \alpha_k / 2$ *and evaluate* $f(u_k + \alpha_k s_k)$.
7. *Set* $u_{k+1} = u_k + \alpha_k s_k$, $k \leftarrow k+1$. *Goto 2.*

Newton-CG Algorithm 2.2 requires the computation of gradients $\nabla \widehat{f}(u_k)$ and the application of Hessians $\nabla^2 \widehat{f}(u_k)$ to vectors p_i. We will discuss how to accomplish these tasks in the following two sections.

3. Gradient Computations. Under Assumption 2.1, the implicit function theorem guarantees the differentiability of $y(\cdot)$. The Jacobian of $y(\cdot)$ is the solution of

$$c_y(y,u)|_{y=y(u)} y_u(u) = -c_u(y,u)|_{y=y(u)}. \tag{3.1}$$

To simplify the notation we write $c_y(y(u),u)$ and $c_u(y(u),u)$ instead of $c_y(y,u)|_{y=y(u)}$ and $c_u(y,u)|_{y=y(u)}$, respectively. With this notation, we have

$$y_u(u) = -c_y(y(u),u)^{-1} c_u(y(u),u). \tag{3.2}$$

The derivative $y_u(u)$ is also called the *sensitivity* (of y with respect to u).

Since $y(\cdot)$ is differentiable, the function $\widehat{f}$ is differentiable and its gradient is given by

$$\begin{aligned} \nabla \widehat{f}(u) &= y_u(u)^T \nabla_y f(y(u),u) + \nabla_u f(y(u),u) \\ &= -c_u(y(u),u)^T c_y(y(u),u)^{-T} \nabla_y f(y(u),u) + \nabla_u f(y(u),u). \end{aligned} \tag{3.3}$$

Note that if we define the matrix

$$W(y,u) = \begin{pmatrix} -c_y(y,u)^{-1} c_u(y,u) \\ I \end{pmatrix}, \tag{3.4}$$

then

$$W(y(u),u) = \begin{pmatrix} y_u(u) \\ I \end{pmatrix} \tag{3.5}$$

and the gradient of $\widehat{f}$ can be written as

$$\nabla \widehat{f}(u) = W(y(u),u)^T \nabla_x f(y(u),u). \tag{3.6}$$

The matrix $W(y,u)$ will play a role later.

Equation (3.3) suggests the following method for computing the gradient.

ALGORITHM 3.1 (Gradient Computation Using Sensitivities).

1. *Given u, solve $c(y,u)=0$ for y (if not done already). Denote the solution by $y(u)$.*
2. *Compute the sensitivities $S=y_u(u)$ by solving $c_y(y(u),u)\,S=-c_u(y(u),u)$.*
3. *Compute $\nabla\widehat{f}(u)=S^T\nabla_y f(y(u),u)+\nabla_u f(y(u),u)$.*

The computation of the sensitivity matrix S requires the solution of n_u systems of linear equations $c_y(y(u),u)\,S=-c_u(y(u),u)$, all of which have the same system matrix but different right hand sides. If n_u is large this can be expensive. The gradient computation can be executed more efficiently since for the computation of $\nabla\widehat{f}(u)$ we do not need S, but only the application of S^T to $\nabla_y f(y(u),u)$. If we revisit (3.3), we can define $\lambda(u)=-c_y(y(u),u)^{-T}\nabla_y f(y(u),u)$, or, equivalently, we can define $\lambda(u)\in\mathbb{R}^{n_y}$ as the solution of

$$c_y(y(u),u)^T\lambda=-\nabla_y f(y(u),u). \tag{3.7}$$

In optimization problems (2.1), (2.2) arising from discretized optimal control problems, the system (3.7) are called the (discrete) adjoint equations and $\lambda(u)$ is the (discrete) adjoint. With this quantity, the gradient can now be written as

$$\nabla\widehat{f}(u)=\nabla_u f(y(u),u)+c_u(y(u),u)^T\lambda(u), \tag{3.8}$$

which suggests the so-called adjoint equation method for computing the gradient.

ALGORITHM 3.2 (Gradient Computation Using Adjoints).

1. *Given u, solve $c(y,u)=0$ for y (if not done already).*
2. *Solve the adjoint equation $c_y(y(u),u)^T\lambda=-\nabla_y f(y(u),u)$ for λ. Denote the solution by $\lambda(u)$.*
3. *Compute $\nabla\widehat{f}(u)=\nabla_u f(y(u),u)+c_u(y(u),u)^T\lambda(u)$.*

The gradient computation using the adjoint equation method can also be expressed using the Lagrangian

$$L(y,u,\lambda)=f(y,u)+\lambda^T c(y,u) \tag{3.9}$$

corresponding to the constraint problem (2.3). Using the Lagrangian, the equation (3.7) can be written as

$$\nabla_y L(y,u,\lambda)|_{y=y(u),\lambda=\lambda(u)}=0. \tag{3.10}$$

Moreover, (3.8) can be written as

$$\nabla\widehat{f}(u)=\nabla_u L(y,u,\lambda)|_{y=y(u),\lambda=\lambda(u)}. \tag{3.11}$$

The adjoint equations (3.7) or (3.10) are easy to write down in this abstract setting, but (hand) generating a code to set up and solve the adjoint equations can be quite a different matter. This will become somewhat apparent when we discuss a simple optimal control example in Section 6. The following observation can be used to generate some checks that indicate the correctness of the adjoint code. Assume that we have a code that for given u computes the solution y of $c(y,u)=0$. Often it is not too difficult to derive from this a code that for given r computes the solution s of $c_y(y,u)s=r$. If λ solves the adjoint equation $c_y(y,u)^T\lambda=-\nabla_y f(y,u)$, then

$$-s^T\nabla_y f(y,u)=s^T c_y(y,u)^T\lambda=r^T\lambda \tag{3.12}$$

must hold.

4. Hessian Computations. Since we assume f and c to be twice continuously differentiable, the function $\widehat{f}$ is twice continuous differentiable. The Hessian of $\widehat{f}$ can be computed from (3.11). In fact, we have already computed the derivative of $y(\cdot)$ in (3.2) using the implicit function theorem. Analogously we can apply the implicit function theorem to (3.7) or equivalently (3.10) to compute the derivative of $\lambda(\cdot)$. Differentiating (3.10) gives

$$\begin{aligned}\nabla_{yy}L(y,u,\lambda)|_{y=y(u),\lambda=\lambda(u)}\; y_u(u) + \nabla_{yu}L(y,u,\lambda)|_{y=y(u),\lambda=\lambda(u)} \\ +\nabla_{y\lambda}L(y,u,\lambda)|_{y=y(u),\lambda=\lambda(u)}\,\lambda_u(u) = 0.\end{aligned}$$

If we use $\nabla_{y\lambda}L(y,u,\lambda) = c_y(y,u)^T$ and (3.2) in the previous equation we find that

$$\begin{aligned}\lambda_u(u) = c_y(y(u),u)^{-T}\big[\nabla_{yy}L(y(u),u,\lambda(u))\,c_y(y(u),u)^{-1}c_u(y(u),u) \\ -\nabla_{yu}L(y(u),u,\lambda(u))\big].\end{aligned} \tag{4.1}$$

To simplify the expression, we have used the notation $\nabla_{yy}L(y(u),u,\lambda(u))$ instead of $\nabla_{yy}L(y,u,\lambda)|_{y=y(u),\lambda=\lambda(u)}\; y_u(u)$ and analogous notation for the other derivatives of L. We will continue to use this notation in the following.

Now we can compute the Hessian of $\widehat{f}$ by differentiating (3.11),

$$\begin{aligned}\nabla^2\widehat{f}(u) = \nabla_{uy}L(y(u),u,\lambda(u))\,y_u(u) + \nabla_{uu}L(y(u),u,\lambda(u)) \\ +\nabla_{u\lambda}L(y(u),u,\lambda(u))\,\lambda_u(u).\end{aligned} \tag{4.2}$$

If we insert (4.1) and (3.2) into (4.2) and observe that $\nabla_{u\lambda}L(y(u),u,\lambda(u)) = c_u(y(u),u)$ the Hessian can be written as

$$\begin{aligned}\nabla^2\widehat{f}(u) &= c_u(y(u),u)^T c_y(y(u),u)^{-T}\nabla_{yy}L(y(u),u,\lambda(u))\,c_y(y(u),u)^{-1}c_u(y(u),u) \\ &\quad -c_u(y(u),u)^T c_y(y(u),u)^{-T}\nabla_{yu}L(y(u),u,\lambda(u)) \\ &\quad -\nabla_{uy}L(y(u),u,\lambda(u))\,c_y(y(u),u)^{-1}c_u(y(u),u) + \nabla_{uu}L(y(u),u,\lambda(u)) \\ &= W(y(u),u)^T \begin{pmatrix} \nabla_{yy}L(y(u),u,\lambda(u)) & \nabla_{yu}L(y(u),u,\lambda(u)) \\ \nabla_{uy}L(y(u),u,\lambda(u)) & \nabla_{uu}L(y(u),u,\lambda(u)) \end{pmatrix} W(y(u),u).\end{aligned} \tag{4.3}$$

Obviously the identities (4.3) can be used to compute the Hessian. However, in many cases, the computation of the Hessian is too expensive. In that case optimization algorithms, such as the Newton-CG Algorithm 2.2 , that only require the computation of Hessian–times–vector products $\nabla^2\widehat{f}(u)v$ can be used. Using the equality (4.3) Hessian–times–vector products can be computed as follows.

ALGORITHM 4.1 (Hessian–Times–Vector Computation).

1. *Given u, solve $c(y,u) = 0$ for y (if not done already). Denote the solution by $y(u)$.*
2. *Solve the adjoint equation $c_y(y(u),u)^T\lambda = -\nabla_y f(y(u),u)$ for λ (if not done already). Denote the solution by $\lambda(u)$.*
3. *Solve the equation $c_y(y(u),u)\,w = c_u(y(u),u)v$.*
4. *Solve the equation*
   $c_y(y(u),u)^T\,p = \nabla_{yy}L(y(u),u,\lambda(u))w - \nabla_{yu}L(y(u),u,\lambda(u))v$.
5. *Compute*
   $\nabla^2\widehat{f}(u)v = c_u(y(u),u)^T\,p - \nabla_{uy}L(y(u),u,\lambda(u))w + \nabla_{uu}L(y(u),u,\lambda(u))v$.

Hence, if $y(u)$ and $\lambda(u)$ are already known, then the computation of $\nabla^2\widehat{f}(u)v$ requires the solution of two linear equations. One similar to the linearized state equation, Step 3, and one similar to the adjoint equation, Step 4.

We conclude this section with an observation concerning the connection between the Newton equation $\nabla^2\widehat{f}(u)\,s_u = -\nabla\widehat{f}(u)$ or the Newton–like equation $\widehat{H}\,s_u = -\nabla\widehat{f}(u)$ and the

solution of a quadratic program. These observations also emphasize the connection between the implicitly constrained problem (2.1) and the nonlinear programming problem (2.3).

THEOREM 4.2. *Let $c_y(y(u),u)$ be invertible and let $\nabla^2\widehat{f}(u)$ be symmetric positive semidefinite. The vector s_u solves the Newton equation*

$$\nabla^2\widehat{f}(u)\,s_u = -\nabla\widehat{f}(u) \tag{4.4}$$

if and only if (s_y,s_u) with $s_y = c_y(y(u),u)^{-1}c_u(y(u),u)s_u$ solves the quadratic program

$$\begin{aligned} \min \quad & \begin{pmatrix} \nabla_y f(y,u) \\ \nabla_u f(y,u) \end{pmatrix}^T \begin{pmatrix} s_y \\ s_u \end{pmatrix} + \tfrac{1}{2}\begin{pmatrix} s_y \\ s_u \end{pmatrix}^T \begin{pmatrix} \nabla_{yy}L(y,u,\lambda) & \nabla_{yu}L(y,u,\lambda) \\ \nabla_{uy}L(y,u,\lambda) & \nabla_{uu}L(y,u,\lambda) \end{pmatrix}\begin{pmatrix} s_y \\ s_u \end{pmatrix}, \\ \textit{s.t.} \quad & c_y(y,u)s_y + c_u(y,u)s_u = 0, \end{aligned} \tag{4.5}$$

where $y = y(u)$ and $\lambda = \lambda(u)$.

Proof. Every feasible point for (4.5) obeys

$$\begin{pmatrix} s_y \\ s_u \end{pmatrix} = \begin{pmatrix} c_y(y(u),u)^{-1}c_u(y(u),u)s_u \\ s_u \end{pmatrix} = W(y(u),u)\,s_u.$$

Thus, using (3.6) and (4.3), we see that (4.5) is equivalent to

$$\min_{s_u} s_u^T\nabla\widehat{f}(u) + \tfrac{1}{2}s_u^T\nabla^2\widehat{f}(u)\,s_u. \tag{4.6}$$

The desired result now follows from the equivalence of (4.5) and (4.6). □

Similarly, one can show the following result.

THEOREM 4.3. *Let $c_y(y(u),u)$ be invertible and let $\widehat{H}\in\mathbb{R}^{n_u\times n_u}$ be a symmetric positive semidefinite matrix. The vector s_u solves the Newton–like equation*

$$\widehat{H}\,s_u = -\nabla\widehat{f}(u), \tag{4.7}$$

if and only if (s_y,s_u) with $s_y = c_y(y(u),u)^{-1}c_u(y(u),u)s_u$ solves the quadratic program

$$\begin{aligned} \min \quad & \begin{pmatrix} \nabla_y f(y,u) \\ \nabla_u f(y,u) \end{pmatrix}^T \begin{pmatrix} s_y \\ s_u \end{pmatrix} + \tfrac{1}{2}\begin{pmatrix} s_y \\ s_u \end{pmatrix}^T \begin{pmatrix} 0 & 0 \\ 0 & \widehat{H} \end{pmatrix}\begin{pmatrix} s_y \\ s_u \end{pmatrix}, \\ \textit{s.t.} \quad & c_y(y,u)s_y + c_u(y,u)s_u = 0, \end{aligned} \tag{4.8}$$

where $y = y(u)$ and $\lambda = \lambda(u)$.

5. Gauss-Newton. In this section we assume that f is the form

$$f(y,u) = \tfrac{1}{2}\|Qy-d\|_2^2 + R(u) \tag{5.1}$$

where $Q\in\mathbb{R}^{m\times n_y}$ is a given matrix, $d\in\mathbb{R}^m$ is a given vector, and $R:\mathbb{R}^{n_u}\to\mathbb{R}$ is a twice continuously differentiable function. This type of objective function arises in data fitting problems, were Qy are observations of the system state, d are data, and $R(u)$ is a regularization term.

The reduced objective function corresponding to (5.1) is

$$\widehat{f}(u) = \tfrac{1}{2}\|Qy(u)-d\|_2^2 + R(u), \tag{5.2}$$

where $y(u)$ is the unique solution of $c(y,u)=0$. The Gauss-Newton method minimizes $\widehat{f}$ by solving a sequence of quadratic problems

$$\min_{s_u} \tfrac{1}{2}\|Qy_u(u)s_u + Qy(u) - d\|_2^2 + \nabla R(u)^T s_u + \tfrac{1}{2}s_u^T\nabla^2 R(u)\,s_u. \tag{5.3}$$

We have

$$\begin{aligned}
&\tfrac{1}{2}\|Qy_u(u)s_u + Qy(u) - d\|_2^2 + \nabla R(u)^T s_u + \tfrac{1}{2} s_u^T \nabla R(u)\, s_u \\
&= \tfrac{1}{2}\|Qy(u) - d\|_2^2 + (Qy(u) - d)^T Qy_u(u)s_u + \tfrac{1}{2} s_u^T y_u(u) Q^T Q y_u(u)\, s_u \\
&\quad + \nabla R(u)^T s_u + \tfrac{1}{2} s_u^T \nabla^2 R(u)\, s_u.
\end{aligned}$$

The Gauss-Newton approximation of the Hessian $\nabla^2 \widehat{f}(u)$ is

$$\widehat{G}(u) = y_u(u) Q^T Q y_u(u) + \nabla^2 R(u). \tag{5.4}$$

Note that $\widehat{G}(u)$ is obtained from $\nabla^2 \widehat{f}(u)$ in (4.3) by replacing $L(y,u,\lambda) = \frac{1}{2}\|Qy - d\|_2^2 + R(u) + \lambda^T c(y,u)$ with $L(y(u),u,0) = \frac{1}{2}\|Qy - d\|_2^2 + R(u)$. Note that if $Qy = d$, then the Lagrange multiplier $\lambda = -c_y(y,u)^{-1}\nabla_y f(y,u) = -c_y(y,u)^{-1} Q^T (Qy - d) = 0$, i.e., for zero residual problems, the Gauss-Newton Hessian approximation is equal to the Hessian.

If we insert (3.2) into (5.4) the Gauss-Newton approximation of the Hessian can be written as

$$\begin{aligned}
\nabla^2 \widehat{f}(u) &= c_u(y(u),u)^T c_y(y(u),u)^{-T} Q^T Q\, c_y(y(u),u)^{-1} c_u(y(u),u) + \nabla^2 R(u) \\
&= W(y(u),u)^T \begin{pmatrix} Q^T Q & 0 \\ 0 & \nabla^2 R(u) \end{pmatrix} W(y(u),u).
\end{aligned} \tag{5.5}$$

The Gauss–Newton–Hessian–times–vector products can be computed as follows. (Step 2 is left empty to facilitate comparison with Algorithm 4.1, see below.)

ALGORITHM 5.1 (Gauss–Newton–Hessian–Times–Vector Computation).

1. *Given u, solve $c(y,u) = 0$ for y (if not done already). Denote the solution by $y(u)$.*
2. *(Nothing needs to be done in this step.)*
3. *Solve the equation $c_y(y(u),u)\, w = c_u(y(u),u)v$.*
4. *Solve the equation $c_y(y(u),u)^T\, p = Q^T Q w$.*
5. *Compute $\widehat{G}(u)v = c_u(y(u),u)^T\, p + \nabla^2 R(u)v$.*

Note that $\nabla_{yy} L(y(u),u,0) = Q^T Q$, $\nabla_{yu} L(y(u),u,0) = 0$, $\nabla_{uu} L(y(u),u,0) = 0$, and $\nabla_{uu} L(y(u),u,0) = \nabla^2 R(u)$. If we compare Algorithms 4.1 and 5.1, then we see that Gauss–Newton–Hessian–times–vector product is computed by using Algorithm 4.1 with $\lambda(u) = 0$.

Analogously to Theorem 4.2 we can show the following result.

THEOREM 5.2. *Let $c_y(y(u),u)$ be invertible. The vector s_u solves the Gauss-Newton subproblem*

$$\min_{s_u} \tfrac{1}{2}\|Qy_u(u)s_u + Qy(u) - d\|_2^2 + \nabla R(u)^T s_u + \tfrac{1}{2} s_u^T \nabla^2 R(u)\, s_u \tag{5.6}$$

if and only if (s_y, s_u) with $s_y = c_y(y(u),u)^{-1} c_u(y(u),u)s_u$ solves the quadratic program

$$\begin{aligned}
\min \quad & \begin{pmatrix} Q^T(Qy-d) \\ \nabla R(u) \end{pmatrix}^T \begin{pmatrix} s_y \\ s_u \end{pmatrix} + \tfrac{1}{2} \begin{pmatrix} s_y \\ s_u \end{pmatrix}^T \begin{pmatrix} Q^T Q & 0 \\ 0 & \nabla^2 R(u) \end{pmatrix} \begin{pmatrix} s_y \\ s_u \end{pmatrix}, \\
\text{s.t.} \quad & c_y(y,u)s_y + c_u(y,u)s_u = 0,
\end{aligned} \tag{5.7}$$

where $y = y(u)$.

6. Optimal Control of Burgers' Equation.

6.1. The Infinite Dimensional Problem. We demonstrate the gradient and Hessian computation using an optimal control problems governed by the so-called Burgers' equation. The Burgers equation can be viewed as the Navier-Stokes equations in one space dimension and it was introduced by Burgers [9, 10]. We first state the optimal control problem in the differential equation setting and then introduce a simple discretization to arrive at a finite dimensional problem of the type (2.1).

We want to minimize

$$\min_u \tfrac{1}{2}\int_0^T \int_0^1 (y(x,t)-z(x,t))^2 + \omega u^2(x,t)dxdt, \tag{6.1a}$$

where y is the solution of

$$\begin{aligned} \tfrac{\partial}{\partial t}y(x,t) - \nu\tfrac{\partial^2}{\partial x^2}y(x,t) + \tfrac{\partial}{\partial x}y(x,t)y(x,t) &= r(x,t)+u(x,t) && (x,t)\in(0,1)\times(0,T),\\ y(0,t) = y(1,t) &= 0 && t\in(0,T),\\ y(x,0) &= y_0(x) && x\in(0,1), \end{aligned} \tag{6.1b}$$

where $z:(0,1)\times(0,T)\to\mathbb{R}$, $r:(0,1)\times(0,T)\to\mathbb{R}$, and $y_0:(0,1)\to\mathbb{R}$ are given functions and $\omega,\nu>0$ are given parameters. The parameter $\nu>0$ is also called the viscosity and the differential equation (6.1b) is known as the (viscous) Burgers' equation. The problem (6.1) is studied, e.g., in [25, 31]. As we have mentioned earlier, (6.1) can be viewed as a first step towards solving optimal control problems governed by the Navier-Stokes equations [2, 17].

In this context of (6.1) the function u is called the control, y is called the state, and (6.1b) is called the state equation. We do not study the infinite dimensional problem (6.1), but instead consider a discretization of (6.1).

6.2. Problem Discretization. To discretize (6.1) in space, we use piecewise linear finite elements. For this purpose, we multiply the differential equation in (6.1b) by a sufficiently smooth function φ which satisfies $\varphi(0)=\varphi(1)=0$. Then we integrate both sides over $(0,1)$, and apply integration by parts. This leads to

$$\begin{aligned} &\frac{d}{dt}\int_0^1 y(x,t)\varphi(x)dx + \nu\int_0^1 \frac{\partial}{\partial x}y(x,t)\frac{d}{dx}\varphi(x)dx + \int_0^1 \frac{\partial}{\partial x}y(x,t)y(x,t)\varphi(x)dx\\ &= \int_0^1 (r(x,t)+u(x,t))\varphi(x)dx. \end{aligned} \tag{6.2}$$

Now we subdivide the spatial interval $[0,1]$ into n subintervals $[x_{i-1},x_i]$, $i=1,\ldots,n$, with $x_i=ih$ and $h=1/n$. We define piecewise linear ('hat') functions

$$\varphi_i(x) = \begin{cases} h^{-1}(x-(i-1)h) & x\in[(i-1)h,ih]\cap[0,1],\\ h^{-1}(-x+(i+1)h) & x\in[ih,(i+1)h]\cap[0,1],\\ 0 & \text{else} \end{cases} \qquad i=0,\ldots,n, \tag{6.3}$$

which satisfy $\varphi_j(x_j)=1$ and $\varphi_j(x_i)=0$, $i\neq j$.

We approximate y and u by functions of the form

$$y_h(x,t) = \sum_{j=1}^{n-1} y_j(t)\varphi_j(x) \tag{6.4}$$

and

$$u_h(x,t) = \sum_{j=0}^{n} u_j(t)\varphi_j(x). \tag{6.5}$$

We set

$$\vec{y}(t) = (y_1(t),\ldots,y_{n-1}(t))^T \quad \text{and} \quad \vec{u}(t) = (u_0(t),\ldots,u_n(t))^T,$$

If we insert the approximations (6.4), (6.5) into (6.2) and require (6.2) to hold for for $\varphi = \varphi_i$, $i = 1,\ldots,n-1$, then we obtain the system of ordinary differential equations

$$M_h \frac{d}{dt}\vec{y}(t) + A_h\vec{y}(t) + N_h(\vec{y}(t)) + B_h\vec{u}(t) = r_h(t), \quad t \in (0,T), \tag{6.6}$$

where $M_h, A_h \in \mathbb{R}^{(n-1)\times(n-1)}$, $B_h \in \mathbb{R}^{(n-1)\times(n+1)}$, $r_h(t) \in \mathbb{R}^{n-1}$, and $N_h(\vec{y}(t)) \in \mathbb{R}^{n-1}$ are matrices or vectors with entries

$$\begin{aligned}
(M_h)_{ij} &= \int_0^1 \varphi_j(x)\varphi_i(x)dx,\\
(A_h)_{ij} &= \nu \int_0^1 \frac{d}{dx}\varphi_j(x)\frac{d}{dx}\varphi_i(x)dx,\\
(B_h)_{ij} &= -\int_0^1 \varphi_j(x)\varphi_i(x)dx,\\
(N_h(\vec{y}(t)))_i &= \sum_{j=1}^{n-1}\sum_{k=1}^{n-1}\int_0^1 \frac{d}{dx}\varphi_j(x)\varphi_k(x)\varphi_i(x)dx\, y_k(t)y_j(t),\\
(r_h(t))_i &= \int_0^1 r(x,t)\varphi_i(x)dx.
\end{aligned}$$

If we insert (6.4), (6.5) into (6.1), we obtain

$$\int_0^T \frac{1}{2}\vec{y}(t)^T M_h\vec{y}(t) + (g_h(t))^T\vec{y}(t) + \frac{\omega}{2}\vec{u}(t)^T Q_h\vec{u}(t)dt + \int_0^T\int_0^1 \frac{1}{2}\hat{y}^2(x,t)dxdt,$$

where $M_h \in \mathbb{R}^{(n-1)\times(n-1)}$ is defined as before and $Q_h \in \mathbb{R}^{(n+1)\times(n+1)}$, $g_h(t) \in \mathbb{R}^{(n-1)}$ are a matrix and vector with entries

$$\begin{aligned}
(Q_h)_{ij} &= \int_0^1 \varphi_j(x)\varphi_i(x)dx,\\
(g_h(t))_i &= -\int_0^1 z(x,t)\varphi_i(x)dx.
\end{aligned}$$

Thus a semi–discretization of the optimal control problem (6.1) is given by

$$\min_{\vec{u}} \int_0^T \frac{1}{2}\vec{y}(t)^T M_h\vec{y}(t) + (g_h(t))^T\vec{y}(t) + \frac{\omega}{2}\vec{u}(t)^T Q_h\vec{u}(t)dt, \tag{6.7a}$$

where $\vec{y}(t)$ is the solution of

$$\begin{aligned}
M_h\tfrac{d}{dt}\vec{y}(t) + A_h\vec{y}(t) + N_h(\vec{y}(t)) + B_h\vec{u}(t) &= r_h(t), \quad t \in (0,T),\\
\vec{y}(0) &= \vec{y}_0,
\end{aligned} \tag{6.7b}$$

where $\vec{y}_0 = (y_0(h),\ldots,y_0(1-h))^T$.

Using the definition (6.3) of φ_i, $i = 0,\dots,n$, it is easy to compute that

$$M_h = \frac{h}{6}\begin{pmatrix} 4 & 1 & & \\ 1 & 4 & 1 & \\ \ddots & \ddots & \ddots & \\ & 1 & 4 & 1 \\ & & 1 & 4 \end{pmatrix} \in \mathbb{R}^{(n-1)\times(n-1)}, \quad A_h = \frac{\nu}{h}\begin{pmatrix} 2 & -1 & & \\ -1 & 2 & -1 & \\ \ddots & \ddots & \ddots & \\ & -1 & 2 & -1 \\ & & -1 & 2 \end{pmatrix} \in \mathbb{R}^{(n-1)\times(n-1)},$$

$$N_h(\vec{y}(t)) = \frac{1}{6}\begin{pmatrix} y_1(t)y_2(t) + y_2^2(t) \\ -y_1^2(t) - y_1(t)y_2(t) + y_2(t)y_3(t) + y_3^2(t) \\ \vdots \\ -y_{i-1}^2(t) - y_{i-1}(t)y_i(t) + y_i(t)y_{i+1}(t) + y_{i+1}^2(t) \\ \vdots \\ -y_{n-3}^2(t) - y_{n-3}(t)y_{n-2}(t) + y_{n-2}(t)y_{n-1}(t) + y_{n-1}^2(t) \\ -y_{n-2}^2(t) - y_{n-2}(t)y_{n-1}(t) \end{pmatrix} \in \mathbb{R}^{n-1}$$

and

$$B_h = -\frac{h}{6}\begin{pmatrix} 1 & 4 & 1 & & \\ & 1 & 4 & 1 & \\ & \ddots & \ddots & \ddots & \\ & & 1 & 4 & 1 \\ & & & 1 & 4 & 1 \end{pmatrix} \in \mathbb{R}^{(n-1)\times(n+1)}, \quad Q_h = \frac{h}{6}\begin{pmatrix} 2 & 1 & & \\ 1 & 4 & 1 & \\ \ddots & \ddots & \ddots & \\ & 1 & 4 & 1 \\ & & 1 & 2 \end{pmatrix} \in \mathbb{R}^{(n+1)\times(n+1)}.$$

To approximate the integrals arising in the definition of $r_h(t)$ and $g_h(t)$ we apply the composite trapezoidal rule. This yields

$$(r_h(t))_i = h\, r(ih,t), \qquad (g_h(t))_i = h\, \hat{y}(ih,t).$$

Later we also need the Jacobian $N_h'(\vec{y}(t)) \in \mathbb{R}^{(n-1)\times(n-1)}$, which is shown in Figure 6.1 .

To discretize the problem in time, we use the Crank-Nicolson method. We let

$$0 = t_0 < t_1 < \ldots < t_{N+1} = T$$

and we define

$$\Delta t_i = t_{i+1} - t_i, \quad i = 0,\dots,N.$$

We also introduce

$$\Delta t_{-1} = \Delta t_{N+1} = 0.$$

The fully discretized problem is given by

$$\min_{\vec{u}_0,\dots,\vec{u}_{N+1}} \sum_{i=0}^{N+1} \frac{\Delta t_{i-1} + \Delta t_i}{2}\left(\frac{1}{2}\vec{y}_i^T M_h \vec{y}_i + (g_h)_i^T \vec{y}_i + \frac{\omega}{2}\vec{u}_i^T Q_h \vec{u}_i\right), \tag{6.8a}$$

where $\vec{y}_1,\dots,\vec{y}_{N+1}$ is the solution of

$$\begin{aligned} &\Big(M_h + \frac{\Delta t_i}{2}A_h\Big)\vec{y}_{i+1} + \frac{\Delta t_i}{2}N_h(\vec{y}_{i+1}) + \frac{\Delta t_i}{2}B_h\vec{u}_{i+1} \\ &+\Big(-M_h + \frac{\Delta t_i}{2}A_h\Big)\vec{y}_i + \frac{\Delta t_i}{2}N_h(\vec{y}_i) + \frac{\Delta t_i}{2}B_h\vec{u}_i \\ &-\frac{\Delta t_i}{2}\Big(r_h(t_i) + r_h(t_{i+1})\Big) = 0, \quad i = 0,\dots,N, \end{aligned} \tag{6.8b}$$

$$
N_h'(\vec{y}(t)) = \frac{1}{6}
\begin{pmatrix}
y_2(t) & y_1(t)+2y_2(t) & & & & \\
-2y_1(t)-y_2(t) & y_3(t)-y_1(t) & y_2(t)+2y_3(t) & & & \\
\ddots & \ddots & \ddots & \ddots & & \\
 & -2y_{i-1}(t)-y_i(t) & y_{i+1}(t)-y_{i-1}(t) & y_i(t)+2y_{i+1}(t) & & \\
 & \ddots & \ddots & \ddots & \ddots & \\
 & & -2y_{n-3}(t)-y_{n-2}(t) & y_{n-1}(t)-y_{n-3}(t) & y_{n-2}(t)+2y_{n-1}(t) & \\
 & & & -2y_{n-2}(t)-y_{n-1}(t) & -y_{n-2}(t) &
\end{pmatrix}
$$

FIG. 6.1. *The Jacobian $N_h'(\vec{y}(t))$*

and $\vec{y}_0$ is given. We denote the objective function in (6.8a) by $\widehat{f}$ and we set

$$u = (\vec{u}_0^T, \ldots, \vec{u}_{N+1}^T)^T.$$

We call u the control, $y = (\vec{y}_1^T, \ldots, \vec{y}_{N+1}^T)^T$ the state, and (6.8b) is called the (discretized) state equation.

Like with many applications, the verification that (6.8) satisfies the Assumptions 2.1, especially the first and third one, is difficult. If the set U of admissible controls u is constrained in a suitable manner and if the parameters ν, h, Δt_i are chosen properly, then it is possible to verify Assumptions 2.1. We ignore this issue and continue as if Assumptions 2.1 are valid for (6.8). In our numerical experiments indicate that this is fine for our problem setting. We also note that our simple Galerkin finite element method in space produces only meaningful results if the mesh size h is sufficiently small (relative to the viscosity ν and size of the solution y). Otherwise the computed solution exhibits spurious oscillations. Again, for our parameter settings, our discretization is sufficient.

Since the Burgers' equation (6.8b) is quadratic in $\vec{y}_{i+1}$, the computation of $\vec{y}_{i+1}$, $i = 0, \ldots, N$, requires the solution of system of nonlinear equations. We apply Newton's method to compute the solution $\vec{y}_{i+1}$ of (6.8b). We use the computed state $\vec{y}_i$ at the previous time step as the initial iterate in Newton's method.

6.3. Gradient and Hessian Computation. The fully discretized problem (6.8) is of the form (1.1), (2.1), (2.2). To compute gradient and Hessian information we first set up the Lagrangian corresponding to (6.8), which is given by

$$\begin{aligned}
&L(\vec{y}_1, \ldots, \vec{y}_{N+1}, \vec{u}_0, \ldots, \vec{u}_{N+1}, \vec{\lambda}_1, \ldots, \vec{\lambda}_{N+1}) \\
&= \sum_{i=0}^{N+1} \frac{\Delta t_{i-1} + \Delta t_i}{2} \left(\frac{1}{2} \vec{y}_i^T M_h \vec{y}_i + (g_h)_i^T \vec{y}_i + \frac{\omega}{2} \vec{u}_i^T Q_h \vec{u}_i \right) \\
&\quad + \sum_{i=0}^{N} \vec{\lambda}_{i+1}^T \Big[\Big(M_h + \frac{\Delta t_i}{2} A_h \Big) \vec{y}_{i+1} + \frac{\Delta t_i}{2} N_h(\vec{y}_{i+1}) + \frac{\Delta t_i}{2} B_h \vec{u}_{i+1} \\
&\qquad + \Big(-M_h + \frac{\Delta t_i}{2} A_h \Big) \vec{y}_i + \frac{\Delta t_i}{2} N_h(\vec{y}_i) + \frac{\Delta t_i}{2} B_h \vec{u}_i \\
&\qquad - \frac{\Delta t_i}{2} \Big(r_h(t_i) + r_h(t_{i+1}) \Big) \Big].
\end{aligned} \tag{6.9}$$

The adjoint equations corresponding to (3.7) are obtained by setting the partial derivatives with respect to y_i of the Lagrangian (6.9) to zero and are given by

$$\begin{aligned}
\Big(M_h + \tfrac{\Delta t_N}{2} A_h + \tfrac{\Delta t_N}{2} N_h'(\vec{y}_{N+1}) \Big)^T \vec{\lambda}_{N+1} &= -\tfrac{\Delta t_N}{2} (M_h \vec{y}_{N+1} + (g_h)_{N+1}), \\
\Big(M_h + \tfrac{\Delta t_{i-1}}{2} A_h + \tfrac{\Delta t_{i-1}}{2} N_h'(\vec{y}_i) \Big)^T \vec{\lambda}_i &= -\Big(-M_h + \tfrac{\Delta t_i}{2} A_h + \tfrac{\Delta t_i}{2} N_h'(\vec{y}_i) \Big)^T \vec{\lambda}_{i+1} \\
&\quad - \tfrac{\Delta t_{i-1} + \Delta t_i}{2} (M_h \vec{y}_i + (g_h)_i), \quad i = N, \ldots, 1,
\end{aligned} \tag{6.10}$$

where $N_h'(\vec{y}_i)$ denotes the Jacobian of $N_h(\vec{y}_i)$. (Recall that $\Delta t_{N+1} = 0$.) Given the solution of (6.10), the gradient of the objective function $\widehat{f}$ can be obtained by computing the partial

derivatives with respect to u_i of the Lagrangian (6.9). The gradient is given by

$$\nabla_u \widehat{f}(u) = \begin{pmatrix} \omega \frac{\Delta t_0}{2} Q_h \vec{u}_0 + \frac{\Delta t_0}{2} B_h^T \vec{\lambda}_1 \\ \omega \frac{\Delta t_0 + \Delta t_1}{2} Q_h \vec{u}_1 + B_h^T (\frac{\Delta t_0}{2} \vec{\lambda}_1 + \frac{\Delta t_1}{2} \vec{\lambda}_2) \\ \vdots \\ \omega \frac{\Delta t_{N-1} + \Delta t_N}{2} Q_h \vec{u}_N + B_h^T (\frac{\Delta t_{N-1}}{2} \vec{\lambda}_N + \frac{\Delta t_N}{2} \vec{\lambda}_{N+1}) \\ \omega \frac{\Delta t_N}{2} Q_h \vec{u}_{N+1} + \frac{\Delta t_N}{2} B_h^T \vec{\lambda}_{N+1} \end{pmatrix}. \tag{6.11}$$

(Recall that $\Delta t_{-1} = \Delta t_{N+1} = 0$.)

We summarize the gradient computation using adjoints in the following algorithm.

ALGORITHM 6.1 (Gradient Computation Using Adjoints).

1. *Given $\vec{u}_0, \ldots, \vec{u}_{N+1}$, and $\vec{y}_0$ compute $\vec{y}_1, \ldots, \vec{y}_{N+1}$ by solving*

$$\begin{aligned} &\Big(M_h + \frac{\Delta t_i}{2} A_h\Big) \vec{y}_{i+1} + \frac{\Delta t_i}{2} N_h(\vec{y}_{i+1}) \\ &= -\Big(-M_h + \frac{\Delta t_i}{2} A_h\Big) \vec{y}_i - \frac{\Delta t_i}{2} N_h(\vec{y}_i) - \frac{\Delta t_i}{2} B_h(\vec{u}_{i+1} + \vec{u}_i) + \frac{\Delta t_i}{2} \Big(r_h(t_i) + r_h(t_{i+1})\Big), \end{aligned}$$

   for $i = 0, \ldots, N$.

2. *Compute $\vec{\lambda}_{N+1}, \ldots, \vec{\lambda}_1$ by solving*

$$\begin{aligned} \Big(M_h + \frac{\Delta t_N}{2} A_h + \frac{\Delta t_N}{2} N_h'(\vec{y}_{N+1})\Big)^T \vec{\lambda}_{N+1} &= -\frac{\Delta t_N}{2} (M_h \vec{y}_{N+1} + (g_h)_{N+1}), \\ \Big(M_h + \frac{\Delta t_{i-1}}{2} A_h + \frac{\Delta t_{i-1}}{2} N_h'(\vec{y}_i)\Big)^T \vec{\lambda}_i &= -\Big(-M_h + \frac{\Delta t_i}{2} A_h + \frac{\Delta t_i}{2} N_h'(\vec{y}_i)\Big)^T \vec{\lambda}_{i+1} \\ &\quad - \frac{\Delta t_{i-1} + \Delta t_i}{2} (M_h \vec{y}_i + (g_h)_i), \end{aligned}$$

   for $i = N, \ldots, 1$.

3. *Compute $\nabla_u \widehat{f}(u)$ from* (6.11).

Of course, if we have computed the solution $\vec{y}_1, \ldots, \vec{u}_{N+1}$ of the discretized Burgers equation (6.8b) for the given $\vec{u}_0, \ldots, \vec{u}_{N+1}$ already, then we can skip step 1 in Algorithm 6.1. Furthermore, we can assemble the components of the gradient $\nabla_u \widehat{f}(u)$ that depend on $\vec{\lambda}_{i+1}$ immediately after it has been computed. This way we do not have to store all $\vec{\lambda}_1, \ldots, \vec{\lambda}_{N+1}$.

We conclude by adapting Algorithm 4.1 to our problem. Since the the objective function (6.8a) is quadratic and the implicit constraints (6.8b) are quadratic in y and linear in u, most of the second derivative terms are zero. The multiplication of the Hessian $\nabla_u^2 \widehat{f}(u)$ times vector v computation can be performed using the following algorithm. In step 4 of the following algorithm we use that $N_h(y)$ is quadratic. Hence $\frac{d}{dy}(N_h'(\vec{y})^T \vec{\lambda}) \vec{w} = N_h'(\vec{w})^T \vec{\lambda}$.

ALGORITHM 6.2 (Hessian–Times–Vector Computation).

1. *Given $\vec{u}_1, \ldots, \vec{u}_{N+1}$, and $\vec{y}_0$ compute $\vec{y}_1, \ldots, \vec{y}_{N+1}$ by solving*

$$\begin{aligned} &\Big(M_h + \frac{\Delta t_i}{2} A_h\Big) \vec{y}_{i+1} + \frac{\Delta t_i}{2} N_h(\vec{y}_{i+1}) \\ &= -\Big(-M_h + \frac{\Delta t_i}{2} A_h\Big) \vec{y}_i - \frac{\Delta t_i}{2} N_h(\vec{y}_i) - \frac{\Delta t_i}{2} B_h(\vec{u}_{i+1} + \vec{u}_i) + \frac{\Delta t_i}{2} \Big(r_h(t_i) + r_h(t_{i+1})\Big), \end{aligned}$$

   for $i = 0, \ldots, N$ (if not done already).

2. *Compute* $\vec{\lambda}_{N+1},\dots,\vec{\lambda}_1$ *by solving*

$$\Big(M_h+\frac{\Delta t_N}{2}A_h+\frac{\Delta t_N}{2}N_h'(\vec{y}_{N+1})\Big)^T\vec{\lambda}_{N+1}=-\frac{\Delta t_N}{2}(M_h\vec{y}_{N+1}+(g_h)_{N+1}),$$

$$\Big(M_h+\frac{\Delta t_{i-1}}{2}A_h+\frac{\Delta t_{i-1}}{2}N_h'(\vec{y}_i)\Big)^T\vec{\lambda}_i=-\Big(-M_h+\frac{\Delta t_i}{2}A_h+\frac{\Delta t_i}{2}N_h'(\vec{y}_i)\Big)^T\vec{\lambda}_{i+1}$$
$$-\frac{\Delta t_{i-1}+\Delta t_i}{2}(M_h\vec{y}_i+(g_h)_i),$$

for $i=N,\dots,1$ *(if not done already).*

3. *Compute* $\vec{w}_1,\dots,\vec{w}_{N+1}$ *from*

$$\Big(M_h+\frac{\Delta t_i}{2}A_h+\frac{\Delta t_i}{2}N_h'(\vec{y}_{i+1})\Big)\vec{w}_{i+1}=-\Big(-M_h+\frac{\Delta t_i}{2}A_h+\frac{\Delta t_i}{2}N_h'(\vec{y}_i)\Big)\vec{w}_i+\frac{\Delta t_i}{2}B_h(\vec{v}_i+\vec{v}_{i+1}),$$

$i=0,\dots,N$, *where* $\vec{w}_0=0$.

4. *Compute* $\vec{p}_{N+1},\dots,\vec{p}_1$ *by solving*

$$\Big(M_h+\frac{\Delta t_N}{2}A_h+\frac{\Delta t_N}{2}N_h'(\vec{y}_{N+1})\Big)^T\vec{p}_{N+1}=\frac{\Delta t_N}{2}M_h\vec{w}_{N+1}+\frac{\Delta t_N}{2}N_h'(\vec{w}_{N+1})^T\vec{\lambda}_{N+1},$$

$$\Big(M_h+\frac{\Delta t_{i-1}}{2}A_h+\frac{\Delta t_{i-1}}{2}N_h'(\vec{y}_i)\Big)^T\vec{p}_i=-\Big(-M_h+\frac{\Delta t_i}{2}A_h+\frac{\Delta t_i}{2}N_h'(\vec{y}_i)\Big)^T\vec{p}_{i+1}$$
$$+\frac{\Delta t_{i-1}+\Delta t_i}{2}M_h\vec{w}_i+N_h'(\vec{w}_i)^T\Big(\frac{\Delta t_{i-1}}{2}\vec{\lambda}_i+\frac{\Delta t_i}{2}\vec{\lambda}_{i+1}\Big),$$

for $i=N,\dots,1$.

5. *Compute*

$$\nabla^2\widehat{f}(u)v=\begin{pmatrix}\omega\frac{\Delta t_0}{2}Q_h\vec{v}_0+\frac{\Delta t_0}{2}B_h^T\vec{p}_1\\ \omega\frac{\Delta t_0+\Delta t_1}{2}Q_h\vec{v}_1+B_h^T(\frac{\Delta t_0}{2}\vec{p}_1+\frac{\Delta t_1}{2}\vec{p}_2)\\ \vdots\\ \omega\frac{\Delta t_{N-1}+\Delta t_N}{2}Q_h\vec{v}_N+B_h^T(\frac{\Delta t_{N-1}}{2}\vec{p}_N+\frac{\Delta t_N}{2}\vec{p}_{N+1})\\ \omega\frac{\Delta t_N}{2}Q_h\vec{v}_{N+1}+\frac{\Delta t_N}{2}B_h^T\vec{p}_{N+1}\end{pmatrix}.$$

The objective function in (6.8a) is of the form (5.1) with

$$Q^TQ=\begin{pmatrix}\frac{\Delta t_{-1}+\Delta t_0}{2}M_h & & \\ & \ddots & \\ & & \frac{\Delta t_N+\Delta t_{N+1}}{2}M_h\end{pmatrix},\quad Q^Td=\begin{pmatrix}-(g_h)_0\\ \cdots\\ -(g_h)_{N+1}\end{pmatrix}$$

and

$$R(\vec{u}_0,\dots,\vec{u}_{N+1})=\frac{\omega}{2}\sum_{i=0}^{N+1}\vec{u}_i^TQ_h\vec{u}_i.$$

Hence we can apply the Gauss-Newton method. The Gauss-Newton-times-vector products are computed by the following algorithm. The difference between Algorithm 6.2 and Algorithm 6.3 below is in step 4. Algorithm 6.3 is obtained from Algorithm 6.2 by replacing the $\vec{\lambda}_i$'s in step 4 by zero.

ALGORITHM 6.3 (Gauss-Newton-Hessian–Times–Vector Computation).

1. *Given* $\vec{u}_1,\ldots,\vec{u}_{N+1}$, *and* $\vec{y}_0$ *compute* $\vec{y}_1,\ldots,\vec{y}_{N+1}$ *by solving*

$$\begin{aligned}&\Big(M_h+\frac{\Delta t_i}{2}A_h\Big)\vec{y}_{i+1}+\frac{\Delta t_i}{2}N_h(\vec{y}_{i+1})\\&=-\Big(-M_h+\frac{\Delta t_i}{2}A_h\Big)\vec{y}_i-\frac{\Delta t_i}{2}N_h(\vec{y}_i)-\frac{\Delta t_i}{2}B_h(\vec{u}_{i+1}+\vec{u}_i)+\frac{\Delta t_i}{2}\Big(r_h(t_i)+r_h(t_{i+1})\Big),\end{aligned}$$

   for $i=0,\ldots,N$ *(if not done already).*

2. *Compute* $\vec{\lambda}_{N+1},\ldots,\vec{\lambda}_1$ *by solving*

$$\begin{aligned}\Big(M_h+\frac{\Delta t_N}{2}A_h+\frac{\Delta t_N}{2}N_h'(\vec{y}_{N+1})\Big)^T\vec{\lambda}_{N+1}&=-\frac{\Delta t_N}{2}(M_h\vec{y}_{N+1}+(g_h)_{N+1}),\\\Big(M_h+\frac{\Delta t_{i-1}}{2}A_h+\frac{\Delta t_{i-1}}{2}N_h'(\vec{y}_i)\Big)^T\vec{\lambda}_i&=-\Big(-M_h+\frac{\Delta t_i}{2}A_h+\frac{\Delta t_i}{2}N_h'(\vec{y}_i)\Big)^T\vec{\lambda}_{i+1}\\&\quad-\frac{\Delta t_{i-1}+\Delta t_i}{2}(M_h\vec{y}_i+(g_h)_i),\end{aligned}$$

   for $i=N,\ldots,1$ *(if not done already).*

3. *Compute* $\vec{w}_1,\ldots,\vec{w}_{N+1}$ *from*

$$\Big(M_h+\frac{\Delta t_i}{2}A_h+\frac{\Delta t_i}{2}N_h'(\vec{y}_{i+1})\Big)\vec{w}_{i+1}=-\Big(-M_h+\frac{\Delta t_i}{2}A_h+\frac{\Delta t_i}{2}N_h'(\vec{y}_i)\Big)\vec{w}_i+\frac{\Delta t_i}{2}B_h(\vec{v}_i+\vec{v}_{i+1}),$$

   $i=0,\ldots,N$, *where* $\vec{w}_0=0$.

4. *Compute* $\vec{p}_{N+1},\ldots,\vec{p}_1$ *by solving*

$$\begin{aligned}\Big(M_h+\frac{\Delta t_N}{2}A_h+\frac{\Delta t_N}{2}N_h'(\vec{y}_{N+1})\Big)^T\vec{p}_{N+1}&=\frac{\Delta t_N}{2}M_h\vec{w}_{N+1},\\\Big(M_h+\frac{\Delta t_{i-1}}{2}A_h+\frac{\Delta t_{i-1}}{2}N_h'(\vec{y}_i)\Big)^T\vec{p}_i&=-\Big(-M_h+\frac{\Delta t_i}{2}A_h+\frac{\Delta t_i}{2}N_h'(\vec{y}_i)\Big)^T\vec{p}_{i+1}\\&\quad+\frac{\Delta t_{i-1}+\Delta t_i}{2}M_h\vec{w}_i,\end{aligned}$$

   for $i=N,\ldots,1$.

5. *Compute*

$$\nabla^2\widehat{f}(u)v=\begin{pmatrix}\omega\frac{\Delta t_0}{2}Q_h\vec{v}_0+\frac{\Delta t_0}{2}B_h^T\vec{p}_1\\\omega\frac{\Delta t_0+\Delta t_1}{2}Q_h\vec{v}_1+B_h^T(\frac{\Delta t_0}{2}\vec{p}_1+\frac{\Delta t_1}{2}\vec{p}_2)\\\vdots\\\omega\frac{\Delta t_{N-1}+\Delta t_N}{2}Q_h\vec{v}_N+B_h^T(\frac{\Delta t_{N-1}}{2}\vec{p}_N+\frac{\Delta t_N}{2}\vec{p}_{N+1})\\\omega\frac{\Delta t_N}{2}Q_h\vec{v}_{N+1}+\frac{\Delta t_N}{2}B_h^T\vec{p}_{N+1}\end{pmatrix}.$$

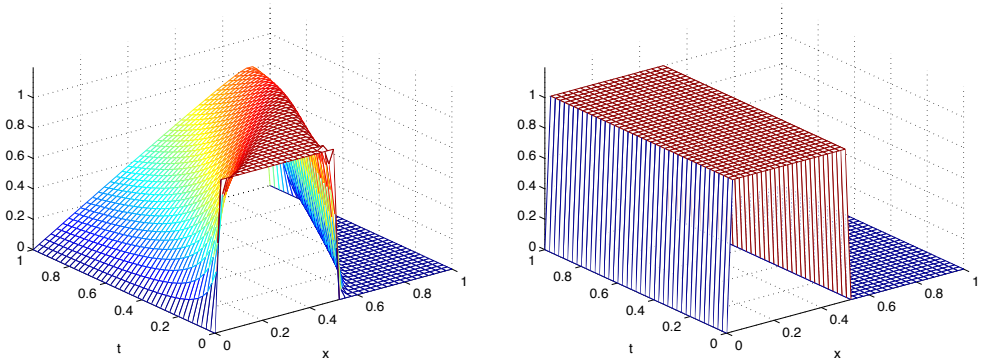
6.4. A Numerical Example. We consider the optimal control problem (6.8) with data $T = 1$, $\omega = 0.05$, $\nu = 0.01$, $r = 0$,

$$y_0(x) = \begin{cases} 1 & x \in (0, \frac{1}{2}], \\ 0 & \text{else}, \end{cases}$$

and $z(x,t) = y_0(x)$, $t \in (0,T)$ (cf. [23]). For the discretization we use $n_x = 80$ spatial subintervals and 80 time steps, i.e., $\Delta t = 1/80$.

The solution y of the discretized Burgers' equation (6.8b) with $u(x,t) = 0$ as well as the desired state z are shown in Figure 6.2 .

FIG. 6.2. *Solution of Burgers' equation with $u = 0$ (no control) (left) and desired state z (right)*



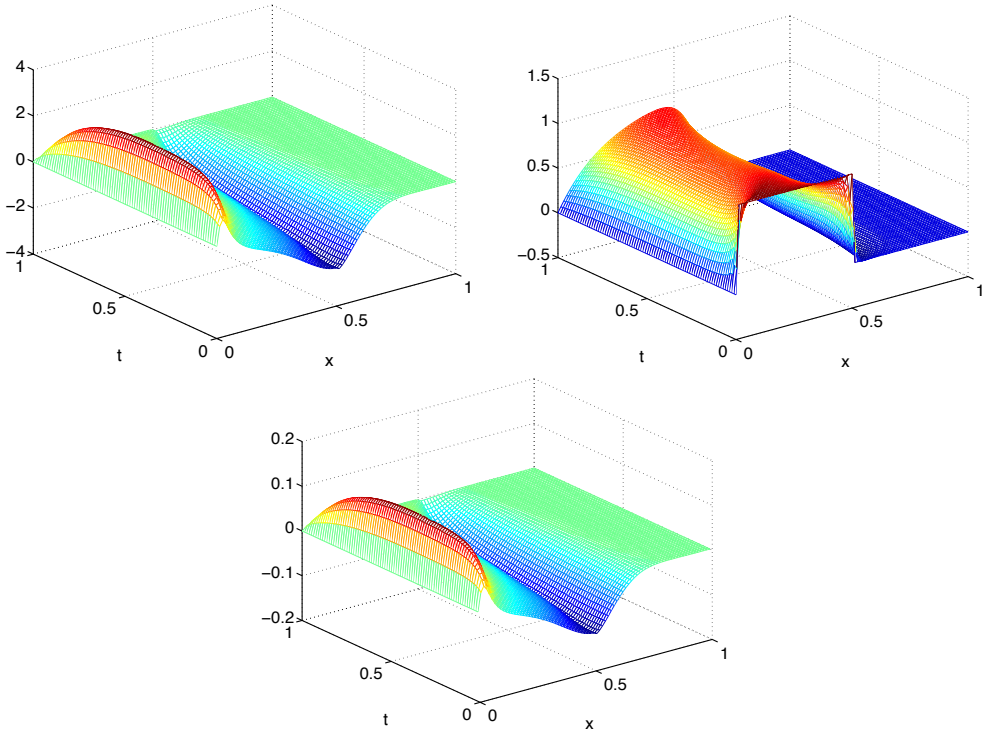
The solution u of the optimal control problem (6.1), (6.1b), the corresponding solution $y(u_*)$ of the discretized Burgers' equation (6.1b) and the solution $\lambda(u_*)$ of (6.10) are plotted in Figure 6.3 .

The convergence history of the Newton–CG method with Armijo-line search applied to (6.8a) is shown in Table 6.1 . We use the Newton–CG Algorithm 2.2 with gtol $= 10^{-8}$ and $\eta_k = \min\{0.01, \|\nabla \widehat{f}(u_k)\|_2\}$.

TABLE 6.1
Performance of a Newton-CG method applied to the solution of (6.1)

k	$\widehat{f}(u_k)$	$\|\nabla \widehat{f}(u_k)\|_2$	$\|s_k\|_2$	α_k	#CG iters
0	$-8.320591e-02$	$3.056462e-03$	$1.350236e+02$	0.5	8
1	$-1.752788e-01$	$7.293242e-04$	$3.511393e+01$	1.0	10
2	$-1.861746e-01$	$9.073135e-05$	$4.239564e+00$	1.0	16
3	$-1.863410e-01$	$1.697294e-06$	$9.011109e-02$	1.0	23
4	$-1.863411e-01$	$1.061131e-09$			

FIG. 6.3. *Optimal control u_* (upper left), corresponding solution $y(u_*)$ of Burgers' equation (upper right) and corresponding Lagrange multipliers $\lambda(u_*)$ (bottom)*



6.5. Checkpointing. In Algorithm 6.1 we note that the state equation is solved forward for the $\vec{y}_i$'s while the adjoint equation is solved backward for the $\vec{\lambda}_i$'s. Moreover the states $\vec{y}_{N+1},\dots,\vec{y}_1$ are needed for the computation of the adjoints $\vec{\lambda}_{N+1},\dots,\vec{\lambda}_1$. If the size of the state vectors $\vec{y}_i$ is small enough so that all states $\vec{y}_1,\dots,\vec{y}_{N+1}$ can be held in the computer memory, this dependence does not pose a difficulty. However, for many problems, such as flow control problems governed by the unsteady Navier-Stokes equations, the states are too large to hold the entire state history in computer memory. In this case one needs to apply so-called checkpointing techniques.

With checkpointing one trades memory for state re-compuations. In a simple scheme one keeps not every state $\vec{y}_0,\vec{y}_1,\dots,\vec{y}_{N+1}$, but only every Mth state $\vec{y}_0,\vec{y}_M,\dots,\vec{y}_{N+1}$ (here we assume that $N+1$ is an integer multiple of M). In the computation of the adjoint variables $\vec{\lambda}_i$ for $i\in\{kM+1,\dots,(k+1)M-1\}$ and some $k\in\{0,\dots,(N+1)/M\}$ one needs $\vec{y}_i$, which has not been stored. Therefore, one uses the stored $\vec{y}_{kM}$ to re-compute $\vec{y}_{kM+1},\dots,\vec{y}_{(k+1)M-1}$.

ALGORITHM 6.4 (Gradient Computation Using Adjoints and Simple Checkpointing). *Let N and M be such that $N+1$ is an integer multiple of M.*

1. *Given $\vec{u}_0,\dots,\vec{u}_{N+1}$, and $\vec{y}_0$. Store $\vec{y}_0$.*
   1.1. *For $k=0,\dots,(N+1)/M-1$ solve*

$$\Big(M_h+\frac{\Delta t_i}{2}A_h\Big)\vec{y}_{i+1}+\frac{\Delta t_i}{2}N_h(\vec{y}_{i+1})$$
$$=-\Big(-M_h+\frac{\Delta t_i}{2}A_h\Big)\vec{y}_i-\frac{\Delta t_i}{2}N_h(\vec{y}_i)-\frac{\Delta t_i}{2}B_h(\vec{u}_{i+1}+\vec{u}_i)+\frac{\Delta t_i}{2}\Big(r_h(t_i)+r_h(t_{i+1})\Big),$$

   for $i=kM,\dots,(k+1)M-1$.
   1.2. *Store $\vec{y}_{(k+1)M}$.*
2. *Adjoint computation.*
   2.1. *Compute $\vec{\lambda}_{N+1}$ by solving*

$$\Big(M_h+\frac{\Delta t_N}{2}A_h+\frac{\Delta t_N}{2}N_h'(\vec{y}_{N+1})\Big)^T\vec{\lambda}_{N+1}=-\frac{\Delta t_N}{2}(M_h\vec{y}_{N+1}+(g_h)_{N+1}).$$

   Add the $\vec{\lambda}_{N+1}$ contribution to the appropriate entries of $\nabla_u\widehat{f}(u)$.
   2.2. *For $k=(N+1)/M-1,\dots,0$*
      2.2.1 *Re-compute $\vec{y}_{kM+1},\dots,\vec{y}_{(k+1)M-1}$ from the stored $\vec{y}_{kM}$ by solving*

$$\Big(M_h+\frac{\Delta t_i}{2}A_h\Big)\vec{y}_{i+1}+\frac{\Delta t_i}{2}N_h(\vec{y}_{i+1})$$
$$=-\Big(-M_h+\frac{\Delta t_i}{2}A_h\Big)\vec{y}_i-\frac{\Delta t_i}{2}N_h(\vec{y}_i)-\frac{\Delta t_i}{2}B_h(\vec{u}_{i+1}+\vec{u}_i)+\frac{\Delta t_i}{2}\Big(r_h(t_i)+r_h(t_{i+1})\Big),$$

      for $i=kM,\dots,(k+1)M-1$.
      2.2.2 *Compute $\vec{\lambda}_{(k+1)M-1},\dots\vec{\lambda}_{kM}$ by solving*

$$\Big(M_h+\frac{\Delta t_{i-1}}{2}A_h+\frac{\Delta t_{i-1}}{2}N_h'(\vec{y}_i)\Big)^T\vec{\lambda}_i=-\Big(-M_h+\frac{\Delta t_i}{2}A_h+\frac{\Delta t_i}{2}N_h'(\vec{y}_i)\Big)^T\vec{\lambda}_{i+1}$$
$$-\frac{\Delta t_{i-1}+\Delta t_i}{2}(M_h\vec{y}_i+(g_h)_i),$$

      for $i=(k+1)M-1,\dots,kM$.
      After $\vec{\lambda}_i$ has been computed add the $\vec{\lambda}_i$ contribution to the appropriate entries of $\nabla_u\widehat{f}(u)$.

Note that for $k = (N+1)/M - 1$ one really does not need to recompute the states $\vec{y}_{N+2-M}, \ldots, \vec{y}_N$ in step 2.2.1, since they are the last states computed in step 1.1. and should be stored there. Algorithm 6.4 requires storage for $(N+1)/M + 1$ vectors $\vec{y}_0, \vec{y}_M, \ldots, \vec{y}_{N+1}$, for $M-1$ vectors $\vec{y}_{kM+1}, \ldots, \vec{y}_{(k+1)M-1}$ computed in step 2.2.1, and for one vector $\vec{\lambda}_i$. This simple checkpointing scheme has been used in [6] for the solution of an optimal control problem governed by the unsteady Navier-Stokes equation.

The simple checkpointing scheme used in Algorithm 6.4 is not optimal in the sense that given a certain memory size to store state information it uses too many state re-computations. The issue of optimal checkpointing is studied in the context of Automatic Differentiation (see also Section 8). The so-called reverse mode automatic differentiation is closely related to gradient computations via the adjoint method. We refer to [15, Sec. 4] for more details. The papers [16, 13] discuss implementations of checkpointing schemes and the paper [20] discusses the use of checkpointing schemes for adjoint based gradient computations in optimal control problem governed by the unsteady Navier-Stokes equation.

7. Optimization. In the previous sections we have discussed the computation of gradient and Hessian information for the implicitly constrained optimization problem (1.1), (2.1), (2.2). Thus it seems we should be able to apply a gradient based optimization algorithm, like the Newton–CG Algorithm 2.2 to solve the problem. In fact, in the previous section we have used the Newton–CG Algorithm 2.2 to solve the discretized optimal control problem (6.8). However, there are important issues left to be dealt with. These are perhaps not so obvious when one deals with the algorithms in the previous sections 'on paper', but they become apparent when one actually as to implement the algorithms.

7.1. Implicit Constraints.

7.1.1. Avoiding Recomputations of y and λ. If we look at the Newton–CG Algorithm 2.2 we see that in each iteration k we have to compute a gradient $\nabla\widehat{f}(u_k)$, we have to apply the Hessian $\nabla^2\widehat{f}(u_k)$ to a number of vectors, and we have to evaluate the function $\widehat{f}$ at some trial points. In a Matlab implementation of Newton–CG Algorithm 2.2 one may require the user to supply three functions

```
function [f] = fval(u, usr_par)
function [g] = grad(u, usr_par)
function [Hv] = Hessvec(v, u, usr_par)
```

that evaluate the objective function $\widehat{f}(u)$, evaluate the gradient $\nabla\widehat{f}(u)$, and evaluate the Hessian-times-vector product $\nabla^2\widehat{f}(u)v$, respectively. The last argument `usr_par` is included to allow he user to pass problem specific parameters to the functions.

Now, if we look at Algorithms 3.1, 3.2, and 4.1 we see that the computation of $\nabla\widehat{f}(u)$ and $\nabla^2\widehat{f}(u)v$ all require the computation of $y(u)$. Furthermore, the computation of $\nabla^2\widehat{f}(u)v$ requires the computation of $\lambda(u)$. Since the computation $y(u)$ can be expensive, we want to reuse an already computed $y(u)$ rather than to recompute $y(u)$ every time `fval`, `grad`, or `Hessvec` is called. Similarly we want to reuse $\lambda(u)$ which has to be computed as part of the gradient computation in Algorithm 3.2 during subsequent calls of `Hessvec`. Of course, if u changes, we must recompute $y(u)$ and $\lambda(u)$. How can we do this?

If we know precisely what is going on in our optimization algorithm, then $y(u)$ and $\lambda(u)$ can be reused. For example, if we use the Newton–CG Algorithm 2.2, then we know that $\widehat{f}(u_k)$ is evaluated before $\nabla\widehat{f}(u_k)$ is computed. Moreover, we know that Hessian-times-vector products $\nabla^2\widehat{f}(u_k)v$ computed only after $\nabla\widehat{f}(u_k)$ is computed. Thus, in this case, when `fval` is called, we compute $y(u_k)$ and store it to make it available for reuse in subsequent calls to `grad` and `Hessvec`. Similarly, if the gradient is implemented via Algorithm 3.2, then when `grad` is called we compute $\lambda(u_k)$ and store it to make it available for reuse in subsequent calls

to `Hessvec`. This strategy works only because we know that the functions `fval`, `grad`, or `Hessvec` are called in the right order. If the optimization is changed such that, say $\nabla\widehat{f}(u_k)$ is computed before $\widehat{f}(u_k)$, the optimization algorithm will fail because it is no longer interfaced correctly with our problem.

We need to find a way that allows us to separate the optimization algorithm (which doesn't need and shouldn't need to know about the fact that the evaluation of our objective function depends on the implicit function $y(u)$) from the particular optimization problem, but allows us to avoid unnecessary recomputations of $y(u)$ and $\lambda(u)$. Such software design issues are extremely important for the efficient implementation of optimization algorithms in which function evaluations may involve expensive simulations. We refer to [3, 4, 5, 18, 27, 29], for more discussions on such issues. In our Matlab implementation we deal with this issue by expanding our interface between optimization algorithm and application slightly.

In our Matlab implementation, we require the user to supply a function

```
function [usr_par] = unew(u, usr_par)
```

The function `unew` is called by the optimization algorithm whenever u has been changed and before any of the three functions `fval`, `grad`, or `Hessvec` are called. In our context, whenever `unew` is called with argument u we compute $y(u)$ and store it to make it available for reuse in subsequent calls to `fval`, `grad` and `Hessvec`. If the implementer of the optimization algorithm changes the algorithm and, say requires the computation of $\nabla\widehat{f}(u_k)$ before the computation of $\widehat{f}(u_k)$ then she/he needs to ensure that `unew` is called with argument u_k before `grad` is called. This change of the optimization algorithm does not need to be communicated to the user of the optimization algorithm. The interface would still work. We use this interface in our Matlab implementation of the Newton–CG Algorithm 2.2 and of a limited memory BFGS method which are available at

http://www.caam.rice.edu/∼heinken/software

The introduction of `unew` enables us to separate the optimization form the application and to avoid unnecessary recomputations of $y(u)$ and $\lambda(u)$. It is not totally satisfactory, however, since it requires that the optimization algorithm developer implements the use of `unew` correctly and it requires the application person not to accidentally overwrite information between two calls of `unew`. These requirements become the more difficult to fulfill the more complex the optimization algorithm and applications become. The papers mentioned above discuss other approaches when C++ instead of Matlab is used.

7.1.2. Inexact Function and Derivative Information. The evaluation of the objective function (2.1) requires the solution of the system of equations (2.2). If c is nonlinear in y, then (2.2) typically must be solved using iterative methods, for example using Newton's method. Consequently, in practice we are not able to compute $y(u)$, but only an approximation $\widetilde{y}_\varepsilon(u)$ that satisfies $\|c(\widetilde{y}_\varepsilon(u),u)\| \le \varepsilon$, where $\varepsilon > 0$ can be selected by the user via the choice of the stopping tolerance of the iterative method applied to (2.2).

Of course, since in practice we can only compute an approximation $\widetilde{y}(u)$ of $y(u)$ we can never compute the objective function $\widehat{f}$ in (2.1) and its derivatives exactly. Instead of $\widehat{f}(u)$, $\nabla\widehat{f}(u)$, $\nabla^2\widehat{f}(u)v$ we can only compute approximations $\widehat{f}_\varepsilon(u) = f(\widetilde{y}_\varepsilon(u),u)$, $\nabla\widehat{f}_\varepsilon(u)$, and $\nabla^2\widehat{f}_\varepsilon(u)v$.

In our numerical solution of the optimal control problem (6.8) we have to solve the nonlinear equations in (6.8b) for $\vec{y}_{i+1}$, $i = 0,\dots,N$. We do this by applying Newton's method. As the initial guess for $\vec{y}_{i+1}$ we use the computed solution $\vec{y}_i$ in the previous time step. We stop the Newton iteration when the residual is less than $10^{-2}\min\{h^2,\Delta t^2\}$. In our example, the computed solution $\vec{y}_i$ at the previous time step is a good approximation for the solu-

tion $\vec{y}_{i+1}$ of (6.8b) and we only need one, at most two Newton steps to reduce the residual below $10^{-2}\min\{h^2,\Delta t^2\}$. We use the computed function and derivative information $\widehat{f}_\varepsilon(u) = f(\widetilde{y}_\varepsilon(u),u)$, $\nabla\widehat{f}_\varepsilon(u)$, and $\nabla^2\widehat{f}_\varepsilon(u)v$ as if it was exact. Since the computed solution $\vec{y}_{i+1}$ is a very good approximation to the exact solution of (6.8b), the inexactness in the computed function and derivative information is small relative to the required stopping tolerance $\|\nabla\widehat{f}(u)\|_2 < \text{gtol}$ when $\text{gtol} = 10^{-8}$, which was used to generate the Table 6.1. However, if we set $\text{gtol} = 10^{-12}$, the Newton–CG Algorithm 2.2 produces the output shown in (7.1). We see that the gradient norm and the step norm are hardly reduced between iterations 4 and 5. The line-search fails in iteration 5 because no sufficient decrease could be detected after 55 reduction of the trial step size α_5 (see Step 6.2 in the Newton–CG Algorithm 2.2). If in the Newton iteration for the solution of (6.8b) we reduce the residual stopping tolerance to $10^{-5}\min\{h^2,\Delta t^2\}$, then the Newton–CG Algorithm 2.2 converges in 5 iterations, see Table 7.2.

TABLE 7.1
Performance of a Newton-CG method with $\text{gtol} = 10^{-12}$ *applied to the solution of* (6.1). *The systems* (6.8b) *are solved with a residual stopping tolerance of* $10^{-2}\min\{h^2,\Delta t^2\}$

k	$\widehat{f}(u_k)$	$\|\nabla\widehat{f}(u_k)\|_2$	$\|s_k\|_2$	α_k	#CG iters
0	$-8.320591e-02$	$3.056462e-03$	$1.350236e+02$	$5.00e-01$	8
1	$-1.752788e-01$	$7.293242e-04$	$3.511393e+01$	$1.00e+00$	10
2	$-1.861746e-01$	$9.073135e-05$	$4.239564e+00$	$1.00e+00$	16
3	$-1.863410e-01$	$1.697294e-06$	$9.011109e-02$	$1.00e+00$	23
4	$-1.863411e-01$	$1.061131e-09$	$4.866485e-05$	$7.63e-06$	36
5	$-1.863411e-01$	$1.061122e-09$	$4.866448e-05$	F	36

TABLE 7.2
Performance of a Newton-CG method with $\text{gtol} = 10^{-12}$ *applied to the solution of* (6.1). *The systems* (6.8b) *are solved with a residual stopping tolerance of* $10^{-5}\min\{h^2,\Delta t^2\}$

k	$\widehat{f}(u_k)$	$\|\nabla\widehat{f}(u_k)\|_2$	$\|s_k\|_2$	α_k	#CG iters
0	$-8.320590e-02$	$3.056462e-03$	$1.350237e+02$	$5.00e-01$	8
1	$-1.752752e-01$	$7.294590e-04$	$3.511488e+01$	$1.00e+00$	10
2	$-1.861738e-01$	$9.070177e-05$	$4.239663e+00$	$1.00e+00$	15
3	$-1.863401e-01$	$1.696622e-06$	$9.009389e-02$	$1.00e+00$	23
4	$-1.863401e-01$	$1.031566e-09$	$4.663490e-05$	$1.00e+00$	37
5	$-1.863401e-01$	$4.666573e-16$			

In the simple problem (6.8) we are able to solve the implicit constraints (6.8b) rather accurately. Consequently, even for an optimization stopping tolerance $\text{gtol} = 10^{-8}$ (which arguably is small for our discretization of (6.1)) the Newton–CG Algorithm 2.2 converges. In other applications the inexactness in the solution of the implicit equation will affect the optimization algorithm even for coarser stopping tolerances gtol.

The 'hand-tuning' of stopping tolerances for the implicit equation and the optimization algorithm is, of course, very unsatisfactory. Ideally one would like an optimization algorithm

that selects these automatically and allows more inexact and therefore less expensive solves of the implicit equation at the beginning of the optimization iteration. One difficulty is that one cannot compute the error in function and derivative information, but one can usually only provide an asymptotic estimate of the form $|\widehat{f}_\varepsilon(u) - \widehat{f}(u)| = O(\varepsilon)$.

There are approaches to handle inexact function and derivative information in optimization algorithms. For example, a general approach to this problem is presented in the book [28]. Additionally, Section 10.6 in [11] describes an approach to adjust the accuracy of function values and derivatives in a trust-region method (see also the references in that section). Handling inexactness in optimization algorithms to increase the efficiency of the overall algorithm by using rough, inexpensive function and derivative information whenever possible while maintaining the robustness of the optimization algorithm are important research problems. Although approaches exist, more work remains to be done.

7.2. Constrainted Optimization. One may wonder why we have treated (1.1), (2.1), (2.2) as an implicitly constrained problem rather than using (2.3). Clearly the explicitly constrained formulation (2.3) has several advantages:
1) Often the problem (2.3) is well-posed, even if the constraint $c(y,u) = 0$ has multiple or no solutions y for some u.
2) The inexactness in function and derivative information that we have discussed in the previous section and that arises out of the solution of $c(y,u) = 0$ for y is no longer an issue, since y and u are both optimization variables in (2.3) and no implicit function has to be computed.
3) Finally, optimization algorithms for (2.3), such as sequential quadratic programming (SQP) methods do not have to maintain feasibility throughout the iteration. This can lead to large gains in efficiency of SQP methods for (2.3) over Newton-type methods for the implicitly constrained problem (1.1), (2.1), (2.2).
If possible, the formulation (2.3) should be chosen over (1.1), (2.1), (2.2). However, in many applications the number of y variables is so huge that it is infeasible to keep all in memory. This is for example the case for problems in which $c(y,u) = 0$ corresponds to the discretization of time dependent partial differential equations in 3D. (Our 1D example problem in Section 6 is a baby sibling of such problems.)

Constrained optimization problems of the type (2.3) can be solved using SQP methods. We mention a few ingredients of SQP methods for the solution of (2.3) with $U = \mathbb{R}^{n_u}$ to point out the relation between SQP methods for (2.3) and Newton-type methods for the implicitly constrained problem (1.1), (2.1), (2.2). More details on SQP methods can be found in [26].

SQP methods compute a solution of (2.3) with $U = \mathbb{R}^{n_u}$ by solving a sequence of quadratic programming (QP) problems

$$\begin{aligned} \min \quad & \begin{pmatrix} \nabla_y f(y,u)^T \\ \nabla_u f(y,u) \end{pmatrix}^T \begin{pmatrix} s_y \\ s_u \end{pmatrix} + \tfrac{1}{2} \begin{pmatrix} s_y \\ s_u \end{pmatrix}^T \begin{pmatrix} \nabla_{yy} L(y,u,\lambda) & \nabla_{yu} L(y,u,\lambda) \\ \nabla_{uy} L(y,u,\lambda) & \nabla_{uu} L(y,u,\lambda) \end{pmatrix} \begin{pmatrix} s_y \\ s_u \end{pmatrix}, \\ \text{s.t.} \quad & c_y(y,u)s_y + c_u(y,u)s_u = -c(y,u), \end{aligned} \tag{7.1}$$

where H is the Hessian of the Lagrangian (3.9),

$$H = \begin{pmatrix} \nabla_{yy} L(y,u,\lambda) & \nabla_{yu} L(y,u,\lambda) \\ \nabla_{uy} L(y,u,\lambda) & \nabla_{uu} L(y,u,\lambda) \end{pmatrix}$$

or a replacement thereof. In so-called reduced SQP methods one uses

$$H = \begin{pmatrix} 0 & 0 \\ 0 & \widehat{H} \end{pmatrix}.$$

The QP (7.1) is almost identical to the QPs (4.5) and (4.8) arising in Newton-type methods for the implicitly constrained problem (1.1), (2.1), (2.2). In the QPs (4.5) and (4.8), $y = y(u)$ and $\lambda = \lambda(u)$ and the right hand side of the constraint is $c(y(u),u) = 0$. This indicates that one step of an SQP method for (2.3) may not be computationally more expensive than one step of a Newton type method for (1.1), (2.1), (2.2). However, SQP methods profit from the decoupling of the variables y and u and can be significantly more efficient than Newton type method for (1.1), (2.1), (2.2) because the latter compute iterates that are on the constraint manifold.

8. Automatic Differentiation. In Section 6.3 we have 'hand coded' the gradient and Hessian-vector multiplication for our example program (6.8). This can be very time consuming. Fortunately, one can use Automatic Differentiation in this process. 'Automatic Differentiation (AD) is a set of techniques based on the mechanical application of the chain rule to obtain derivatives of a function given as a computer program' [1]. We have already come across AD in our discussion of checkpointing. For more information of how AD works we refer to [26, Sec. 8.2] and [14, 15]. The Community Portal for Automatic Differentiation [1] contains links to other AD resources, including software tools.

9. Differential Equation Constraints. In many applications, including our simple model problem (6.1), the governing equations are (partial) differential equations. After discretization of the differential equations, one obtains a system of (nonlinear) algebraic equations and the techniques discussed in this paper can be applied. However, it also possible to extend the techniques discussed in this paper so that hey are applicable to problems (1.1), (2.1), (2.2) posed in infinite dimensional function spaces. We refer to the books [8, 21] and references cited therein for optimization (optimal control) problems governed by ordinary differential equations and to the books [19, 30, 24] and references cited therein for optimization (optimal control) problems governed by partial differential equations.

REFERENCES

[1] *Community portal for automatic differentiation*. http://www.autodiff.org.

[2] F. ABERGEL AND R. TEMAM, *On some control problems in fluid mechanics*, Theoretical and Computational Fluid Dynamics, 1 (1990), pp. 303–325.

[3] R. A. BARTLETT, *Thyra linear operators and vectors: Overview of interfaces and support software for the development and interoperability of abstract numerical algorithms*, Tech. Rep. SAND 2007-5984, Sandia National Laboratories, 2007.

[4] R. A. BARTLETT, S. S. COLLIS, T. COFFEY, D. DAY, M. HEROUX, R. HOEKSTRA, R. HOOPER, R. PAWLOWSKI, E. PHIPPS, D. RIDZAL, A. SALINGER, H. THORNQUIST, AND J. WILLENBRING, *ASC vertical integration milestone*, Tech. Rep. SAND 2007-5839, Sandia National Laboratories, 2007.

[5] R. A. BARTLETT, B. G. VAN BLOEMEN WAANDERS, AND M. A. HEROUX, *Vector reduction/transformation operators*, ACM Trans. Math. Software, 30 (2004), pp. 62–85.

[6] M. BERGGREN, *Numerical solution of a flow-control problem: Vorticity reduction by dynamic boundary action*, SIAM J. Scientific Computing, 19 (1998), pp. 829–860.

[7] D. P. BERTSEKAS, *Nonlinear Programming*, Athena Scientific, Belmont, Massachusetts, 1995.

[8] A. BRYSON AND Y. HO, *Applied Optimal Control*, Hemisphere, New York, 1975.

[9] J. M. BURGERS, *Application of a model system to illustrate some points of the statistical theory of free turbulence*, Nederl. Akad. Wetensch., Proc., 43 (1940), pp. 2–12.

[10] ———, *A mathematical model illustrating the theory of turbulence*, in Advances in Applied Mechanics, R. von Mises and T. von Kármán, eds., Academic Press Inc., New York, N. Y., 1948, pp. 171–199.

[11] A. R. CONN, N. I. M. GOULD, AND P. L. TOINT, *Trust–Region Methods*, SIAM, Philadelphia, 2000.

[12] J. E. DENNIS, JR. AND R. B. SCHNABEL, *Numerical Methods for Nonlinear Equations and Unconstrained Optimization*, SIAM, Philadelphia, 1996.

[13] M. S. GOCKENBACH, D. R. REYNOLDS, P. SHEN, AND W. W. SYMES, *Efficient and automatic implementation of the adjoint state method*, ACM Trans. Math. Softw., 28 (2002), pp. 22–44.

[14] A. GRIEWANK, *Evaluating Derivatives. Principles and Techniques of Algorithmic Differentiation*, Frontiers in Applied Mathematics, SIAM, Philadelphia, 2000.

[15] ———, *A mathematical view of automatic differentiation*, in Acta Numerica 2003, A. Iserles, ed., Cambridge University Press, Cambridge, London, New York, 2003, pp. 321–398.

[16] A. GRIEWANK AND A. WALTHER, *Algorithm 799: revolve: An implementation of checkpointing for the reverse or adjoint mode of computational differentiation*, ACM Trans. Math. Softw., 26 (2000), pp. 19–45.

[17] M. D. GUNZBURGER, *Perspectives in Flow Control and Optimization*, SIAM, Philadelphia, 2003.

[18] M. HEINKENSCHLOSS AND L. N. VICENTE, *An interface between optimization and application for the numerical solution of optimal control problems*, ACM Transactions on Mathematical Software, 25 (1999), pp. 157–190.

[19] M. HINZE, R. PINNAU, M. ULBRICH, AND S. ULBRICH, *Optimization with Partial Differential Equations*, vol. 23 of Mathematical Modelling, Theory and Applications, Springer Verlag, Heidelberg, New York, Berlin, 2009.

[20] M. HINZE, A. WALTHER, AND J. STERNBERG, *An optimal memory-reduced procedure for calculating adjoints of the instationary Navier-Stokes equations*, Optimal Control Applications and Methods, 27 (2006), pp. 19–40.

[21] J. JAHN, *Introduction to the Theory of Nonlinear Optimization*, Springer Verlag, Berlin, Heidelberg, New York, third ed., 2007.

[22] C. T. KELLEY, *Iterative Methods for Optimization*, SIAM, Philadelphia, 1999.

[23] K. KUNISCH AND S. VOLKWEIN, *Control of Burger's equation by a reduced order approach using proper orthogonal decomposition*, Journal of Optimization Theory and Applications, 102 (1999), pp. 345–371.

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

[25] H. V. LY, K. D. MEASE, AND E. S. TITI, *Distributed and boundary control of the viscous Burgers' equation*, Numer. Funct. Anal. Optim., 18 (1997), pp. 143–188.

[26] J. NOCEDAL AND S. J. WRIGHT, *Numerical Optimization*, Springer Verlag, Berlin, Heidelberg, New York, second ed., 2006.

[27] A. D. PADULA, *Software Design for Simulation Driven Optimization*, PhD thesis, Department of Computational and Applied Mathematics, Rice University, Houston, TX, 2005. Available as CAAM TR05–11.

[28] E. POLAK, *Optimization:Algorithms and Consistent Approximations*, Applied Mathematical Sciences, Vol. 124, Springer Verlag, Berlin, Heidelberg, New-York, 1997.

[29] W. W. SYMES, A. D. PADULA, AND S. D. SCOTT, *A software framework for the abstract expression of coordinate-free linear algebra and optimization algorithms*, tr05–12, Dept. of Computational and Applied Mathematics, Rice University, Houston, Texas, 2005.

[30] F. TRÖLTZSCH, *Optimal Control of Partial Differential Equations: Theory, Methods and Applications*, vol. 112 of Graduate Studies in Mathematics, American Mathematical Society, Providence, RI, 2010.

[31] S. VOLKWEIN, *Distributed control problems for the Burgers equation*, Comput. Optim. Appl., 18 (2001), pp. 115–140.