Finite Dimensional Optimal Control; Nonlinear Conjugate Gradient Methods and its Implementation in Python

Bachelor Thesis

submitted by

Stefan Kienle

at the University of Konstanz.

Department of Mathematics and Statistics

Evaluated by
Prof. Dr. Stefan Volkwein

Konstanz, Januar 2020
Abstract

In this Bachelor thesis an introduction of finite dimensional optimal control problems and a summary of nonlinear conjugate gradient methods are presented. A Python algorithm to solve a specific type of finite dimensional optimal control problem was developed as part of this thesis and is documented in this work.

Zusammenfassung

In dieser Bachelorarbeit wird eine Einführung in endlichdimensionale optimale Steuerungsprobleme präsentiert und die Literatur zu nichtlinearen konjugierten Gradientenmethoden zusammengefasst. Die beiden Themengebiete bilden die Basis für den im Rahmen dieser Abschlussarbeit entwickelten Python Algorithmus zum lösen eines speziellen Problemtyps. Dieser Algorithmus ist in dieser Arbeit dokumentiert.
# Contents

**Introduction**                                                                 1

1. **Finite Dimensional Optimal Control**                                          2
   1.1. Task of Optimal Control with Linear Constraint ........................................... 2
   1.1.1. Optimality System ...................................................................................... 3
   1.1.2. The Lagrange function ................................................................................ 6
   1.2. Task of Optimal Control with Nonlinear Constraint ......................................... 7
   1.2.1. Example subject to implementation in Python ............................................... 8
   1.2.2. Implemented Version .................................................................................. 9

2. **Nonlinear Conjugate Gradient Method**                                          11
   2.1. Linear Conjugate Gradient Method ..................................................................... 11
   2.2. Nonlinear Conjugate Gradient Method ................................................................ 13
   2.2.1. Fletcher-Reeves Method .............................................................................. 13
   2.2.2. Alternative Nonlinear-CG Methods ................................................................ 16
   2.2.3. Performance analysis .................................................................................. 17

3. **Implementation in Python**                                                     19
   3.1. Packages and Formats ..................................................................................... 19
   3.2. Task of Optimal Control .................................................................................. 20
   3.3. Nonlinear Conjugate Gradient Method ................................................................ 21
   3.4. Main File ........................................................................................................ 23

4. **Numerical Results**                                                           24
   4.1. Linear Constraint ......................................................................................... 24
   4.2. Nonlinear Constraint ..................................................................................... 25

5. **Conclusion**                                                                  27

A. **Notation**                                                                   30

B. **Python Codes**                                                               31
   B.1. Task of Finite Dimensional Optimal Control .................................................. 31
   B.2. Nonlinear Conjugate Gradient Method ................................................................ 39
C. Numerical Results

C.1. Discontinuous $y_s$ ........................................... 45
C.2. Continuous $y_s$, 1 ........................................... 46
C.3. Continuous $y_s$, 2 ........................................... 47
C.4. Continuous $y_s$, 3 ........................................... 48
C.5. Continuous $y_s$, 4 ........................................... 49
List of Figures

4.1. Linear Constraint; $d = 0$ and $a = 0$; linear equation system. 24
4.2. Linear Constraint; $d = 0$; nonlinear CG. 25
4.3. Nonlinear Constraint; nonlinear CG. 26
4.4. Nonlinear Constraint; behavior of $||\nabla \hat{J}(u_k)||, \hat{J}(u_k)$; nonlinear CG. 26
C.1. Discontinuous $y_s$; nonlinear constraint; nonlinear CG. 45
C.2. Discontinuous $y_s$; nonlinear constraint; behavior of $||\nabla \hat{J}(u_k)||, \hat{J}(u_k)$; nonlinear CG. 46
C.3. Continuous $y_s$, 1; nonlinear constraint; nonlinear CG. 46
C.4. Continuous $y_s$, 1; nonlinear constraint; behavior of $||\nabla \hat{J}(u_k)||, \hat{J}(u_k)$; nonlinear CG. 47
C.5. Continuous $y_s$, 2; nonlinear constraint; nonlinear CG. 47
C.6. Continuous $y_s$, 2; nonlinear constraint; behavior of $||\nabla \hat{J}(u_k)||, \hat{J}(u_k)$; nonlinear CG. 48
C.7. Continuous $y_s$, 3; nonlinear constraint; nonlinear CG. 48
C.8. Continuous $y_s$, 3; nonlinear constraint; behavior of $||\nabla \hat{J}(u_k)||, \hat{J}(u_k)$; nonlinear CG. 49
C.9. Continuous $y_s$, 4; nonlinear constraint; nonlinear CG. 49
C.10. Continuous $y_s$, 4; nonlinear constraint; behavior of $||\nabla \hat{J}(u_k)||, \hat{J}(u_k)$; nonlinear CG. 50
Introduction

In many mathematical applications large scale minimization tasks arise. Fast growing fields for such applications are Artificial Intelligence and Machine Learning. Another and not less important field is Optimal Control. Usually, the considered tasks need to be discretised before they can be solved numerically. Therefore, reasonable algorithms can be developed alongside the discrete problems. These discretised optimal control problems often take on a specific form, which is introduced in the first section of this thesis. Furthermore, in the first section optimality conditions are developed in order to provide the necessary theory for solving finite dimensional optimal control problems numerically. Conjugate gradient methods are highly efficient in solving large scale minimization tasks because they have comparatively little memory requirements. In the second section literature about conjugate gradient methods is summarized, keeping the linear version short and focusing on the nonlinear version.

The algorithm is implemented in Python. This programming language is heavily in use for all above mentioned application fields and provides a good basis to develop algorithms, because it is open source and already has rich packages to build on. In the third section the implemented algorithm is documented and discussed. Whereas, in the fourth section of this thesis numerical results of the algorithm for the finite dimensional task of optimal control are presented. The appendix contains the information about the used notation (A), the full codes and some more numerical results.
1. Finite Dimensional Optimal Control

In the first part of this section we closely follow ([10], Section 1.4). To ensure formal correctness, definitions, corollaries, lemmas and theorems are taken from the book and are freely translated into English. In the second part of this section an optimality system in case of a nonlinear constraint is developed. To complete the section one possible application is presented.

1.1. Task of Optimal Control with Linear Constraint

Given a cost functional \( J = J(y, u) \), \( J : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R} \) to be minimized; matrices \( A \in \mathbb{R}^{n \times n}, B \in \mathbb{R}^{n \times m} \) and \( U_{ad} \subset \mathbb{R}^m \), the admissible range for arguments \( u \). If \( U_{ad} \neq \mathbb{R}^m \), we assume \( U_{ad} \) has the form \( \{ u \in \mathbb{R}^m : u_a \leq u \leq u_b \} \) with \( u_a \leq u_b \in \mathbb{R}^m \) known vectors. The optimization task with linear constraint reads as follows:

\[
\begin{align*}
\min_{y,u} J(y, u) \quad \text{subject to} \quad Ay = Bu, \quad u \in U_{ad}.
\end{align*}
\]

(1.1)

Let additionally \( A \) be invertible. We then have

\[ y = A^{-1}Bu. \]

For any \( u \in \mathbb{R}^m \) there is an unique solution \( y \in \mathbb{R}^n \), since the linear mapping described by \( A \) is bijective due to the invertibility of \( A \). In other words: For any chosen \( u \) there is always exactly one corresponding \( y \). Therefore \( u \) is called control or control vector and \( y \) is called state or associated state vector.

We define \( S := A^{-1}B, \) to get the reduced cost functional \( \hat{J} \),

\[ \hat{J}(u) := J(Su, u) = J(y, u). \]

We now have a reduced problem on \( u \). This makes (1.1) to a optimization task.

\[
\begin{align*}
\min_u \hat{J}(u) \quad \text{subject to} \quad u \in U_{ad}.
\end{align*}
\]

(1.2)

**Definition 1.1.1.** We call a vector \( \bar{u} \in U_{ad} \) optimal control for the task (1.1), if \( \hat{J}(\bar{u}) \leq \hat{J}(u) \) applies to all \( u \in U_{ad} \). We call \( \bar{y} := S\bar{u} \) optimal state associated to \( \bar{u} \).
Corollary 1.1.2. If $J$ is continuous on $\mathbb{R}^n \times U_{ad}$, the set $U_{ad}$ is nonempty, bounded and closed; and the matrix $A$ is invertible. Then at least one optimal control exists for (1.1).

Proof: Since $J$ is continuous on $\mathbb{R}^n \times U_{ad}$ it directly follows that $\hat{J}$ is continuous on $U_{ad}$. As $U_{ad}$ is a bounded and closed set of the finite dimensional space $\mathbb{R}^m$, it is compact. According to the Weierstrass extreme value theorem ([4], Folgerung 10.9) there exists a $\bar{u} \in U_{ad}$ satisfying $J(\bar{u}) = \min_{u \in U_{ad}} J(u)$. □

1.1.1. Optimality System

From here on we make the additional assumption that $J$ is continuously partial differentiable with respect to $y$ and $u$. The differentiability of $\hat{J}$ follows directly. In the following we consider the necessary first-order optimality conditions and develop an optimality system based on them.

Theorem 1.1.3. Let $\bar{u}$ be an optimal control for the problem (1.2) and $\hat{J}$ continuously differentiable on the convex set $U_{ad}$. Then $\bar{u}$ satisfies the variational inequality

$$\hat{J}'(\bar{u})(u - \bar{u}) \geq 0 \quad \forall u \in U_{ad}. \quad (1.3)$$

Conversely, if $\bar{u} \in U_{ad}$ is a solution of the variational inequality (1.3) and $\hat{J}$ a convex function then $\bar{u}$ is a solution to the problem (1.2).

Proof: For any $u \in U_{ad}$ we consider the linear combination

$$u(t) = \bar{u} + t(u - \bar{u}) \quad \text{where} \quad t \in (0, 1].$$

Because of the convexity of $U_{ad}$ it holds that $u(t) \in U_{ad}$. Since $\bar{u}$ is optimal, it follows that $\hat{J}(u(t)) \geq \hat{J}(\bar{u})$ and further

$$\frac{1}{t}(\hat{J}(\bar{u} + t(u - \bar{u})) - \hat{J}(\bar{u})) \geq 0.$$ 

For $t \searrow 0$ we get the variational inequality

$$\hat{J}'(\bar{u})(u - \bar{u}) = \lim_{t \searrow 0} \frac{\hat{J}(\bar{u} + t(u - \bar{u})) - \hat{J}(\bar{u})}{t} \geq 0.$$ 

Reverse: Since $\hat{J}$ is a convex function for any $u \in U_{ad}$ it holds that ([11], Satz 6.3)

$$\hat{J}(u) - \hat{J}(\bar{u}) \geq \hat{J}'(\bar{u})(u - \bar{u}) \geq 0.$$ 

Therefore, $\bar{u}$ is a optimal control for (1.2). □
The derivative $\dot{J}'(\bar{u})$ of (1.3) is calculated as follows:

$$\dot{J}'(\bar{u})h = D_yJ(S\bar{u}, \bar{u})Sh + D_uJ(S\bar{u}, \bar{u})h$$

$$= (\nabla_yJ(\bar{y}, \bar{u}), A^{-1}Bh)_{\mathbb{R}^n} + (\nabla_uJ(\bar{y}, \bar{u}), h)_{\mathbb{R}^m}$$

$$= (B^T(A^T)^{-1}\nabla_yJ(\bar{y}, \bar{u}) + \nabla_uJ(\bar{y}, \bar{u}), h)_{\mathbb{R}^m}.$$ 

Thus, the variational inequality (1.3) takes on the form:

$$(B^T(A^T)^{-1}\nabla_yJ(\bar{y}, \bar{u}) + \nabla_uJ(\bar{y}, \bar{u}), u - \bar{u})_{\mathbb{R}^m} \geq 0 \quad \forall u \in U_{ad}. \quad (1.4)$$

**Adjoint Variable and Reduced Gradient**

The adjoint variable is introduced to avoid the high numerical cost of computing inverse matrices. The adjoint variable is defined as $\bar{p} := (A^T)^{-1}\nabla_yJ(\bar{y}, \bar{u})$. Thus, we can compute $\bar{p}$ from the following equation system:

$$A^T\bar{p} = \nabla_yJ(\bar{y}, \bar{u}). \quad (1.5)$$

**Definition 1.1.4.** Equation (1.5) is called adjoint equation. The solution to the adjoint equation, $\bar{p}$, is called adjoint variable associated with $(\bar{y}, \bar{u})$.

Using $\bar{y} = S\bar{u}$, the gradient of $\dot{J}$ simplifies to

$$\nabla \dot{J}(\bar{u}) = B^T\bar{p} + \nabla_uJ(\bar{y}, \bar{u}).$$

The gradient, $\nabla \dot{J}(\bar{u})$, is called (on $u$) reduced gradient. At any $u \in U_{ad}$, where $y = Su$ and $A^T\bar{p} = \nabla_yJ(y, u)$ hold, the directional derivative of $\dot{J}$ in direction $h$ is given by

$$\dot{J}'(\bar{u})h = (B^T\bar{p} + \nabla_uJ(\bar{y}, \bar{u}), h)_{\mathbb{R}^m}.$$ 

**Lemma 1.1.5.** Suppose $A$ is invertible, $\dot{J}$ is continuously differentiable on $U_{ad}$, $\bar{u}$ is an optimal control for (1.2) and $\bar{y}$ the associated optimal state. Then, there is a unique adjoint variable satisfying (1.5). Moreover, the variational inequality holds

$$(B^T\bar{p} + \nabla_uJ(\bar{y}, \bar{u}), u - \bar{u})_{\mathbb{R}^m} \geq 0 \quad \forall u \in U_{ad}. \quad (1.6)$$

**Proof:** Since $A$ is invertible and $\bar{p}$ is defined as $\bar{p} := (A^T)^{-1}\nabla_yJ(\bar{y}, \bar{u})$, it follows directly that there exists a unique solution for adjoint equation (1.5). Inequality (1.6) follows from the above observations. □

In total we are now able to define the following optimality system for the three unknown vectors $\bar{y}$, $\bar{u}$ und $\bar{p}$.
1.1: Task of Optimal Control with Linear Constraint

\begin{align*}
(I) \quad Ay &= Bu, \quad u \in U_{ad}, \\
(II) \quad A^T p &= \nabla_y J(y, u), \\
(III) \quad (B^T p + \nabla_u J(y, u), v - u)_{\mathbb{R}^m} &\geq 0 \quad \forall v \in U_{ad}
\end{align*}

**Remark 1.1.6.** The usual procedure to use the above optimality system in iterative applications for finding the optimal control and its associated optimal state is to:

1. Solve $Ay = Bu$ for $y$;
2. Solve $A^T p = \nabla_y J(y, u)$ for $p$;
3. Compute the gradient $B^T p + \nabla_u J(y, u)$.

**No Restriction of $u$**

Suppose $U_{ad} = \mathbb{R}^m$. Because $u - \bar{u}$ can take on any value $h \in \mathbb{R}^m$ from (1.6) follows that

$$B^T \bar{p} + \nabla_u J(\bar{y}, \bar{u}) = 0$$

needs to hold.

**Example 1.1.7.** Consider the quadratic cost functional with $y_s \in \mathbb{R}^n$ and $\lambda \geq 0$

$$J(y, u) = \alpha \frac{1}{2} ||y - y_s||^2 + \lambda \frac{1}{2} ||u||^2,$$

where $y_s$ is a known vector. The gradients are

$$\nabla_y J(y, u) = \alpha (y - y_s) \quad \text{and} \quad \nabla_u J(y, u) = \lambda u.$$

We obtain an optimality system which takes on the form of a linear equation system.

\begin{align*}
(I) \quad Ay &= Bu, \quad u \in U_{ad}, \\
(II) \quad A^T p &= \alpha (y - y_s), \\
(III) \quad B^T p + \lambda u &= 0
\end{align*}

An intuitive approach to find the optimal control and its associated optimal state is to
solve the linear equation system:

\[
\begin{pmatrix}
\lambda I_m & 0_{m \times n} & B^T \\
0_{n \times m} & \alpha I_{n \times n} & -A^T \\
-B & A & 0_{n \times n}
\end{pmatrix}
\begin{pmatrix}
u \\
y \\
p
\end{pmatrix}
=
\begin{pmatrix}
0_{m \times 1} \\
\alpha y_s \\
0_{n \times 1}
\end{pmatrix}.
\]

If we consider the reduced cost functional

\[
\hat{J}(u) = \frac{\alpha}{2}||Su - y_s||^2 + \frac{\lambda}{2}||u||^2,
\]

where \( S := A^{-1}B \), then the gradient is given by

\[
\nabla \hat{J}(u) = (\alpha S^T S + \lambda I_m)u - \alpha S^T Sy_s.
\]

Since the reduced cost functional is convex \(^1\) and \( \alpha S^T S + \lambda I_m \) is positive definite and symmetric \(^2\), another approach is to solve \((\alpha S^T S + \lambda I_m)u = \alpha S^T Sy_s \). Because of the kind properties of the example, the Cholesky method or the linear conjugate gradient method would be effective to find solutions for \( u \).

### 1.1.2. The Lagrange function

Since (1.1) is a minimization problem subject to constraints it is worth considering the Lagrange function.

**Definition 1.1.8.** The function \( L : \mathbb{R}^{2n+m} \rightarrow \mathbb{R} \) given as

\[
L(y, u, p) := J(y, u) - (Ay - Bu, p)_{\mathbb{R}^n}
\]

is called Lagrange function.

Consider the partial derivatives of the Lagrange function

\[
\begin{align*}
\nabla_y L(y, u, p) &= \nabla_y J(y, u) - A^T p \\
\nabla_u L(y, u, p) &= \nabla_u J(y, u) + B^T p \\
\nabla_p L(y, u, p) &= Ay - Bu.
\end{align*}
\]

The adjoint equation (1.5) is equivalent to \( \nabla_y L(\bar{y}, \bar{u}, \bar{p}) = 0 \). The variational inequality (1.6) is equivalent to \((\nabla_u L(\bar{y}, \bar{u}, \bar{p}), u - \bar{u})_{\mathbb{R}^m} \geq 0 \) for all \( u \in U_{ad} \). Equations (1.5) and (1.6) can be set up by differentiating the Lagrange function with respect to \( y \) respectively \( u \).

\(^1\)For any \( h \in \mathbb{R}^m \setminus \{0\} \) it holds that \( h^T \nabla^2 \hat{J}(u)h = h^T(\alpha S^T S + \lambda I_m)h = \alpha h^T S^T Sh + h^T I_m h = \alpha |Sh|^2 + \lambda |h|^2 \geq 0 \).

\(^2\)It holds, that \((\alpha S^T S + \lambda I_m)^T = (\alpha S^T S + \lambda I_m)\).
1.2: Task of Optimal Control with Nonlinear Constraint

For the problem without equality constraint,

\[ \min_{y, u} L(y, u, \bar{p}), \quad u \in U_{ad}, \quad y \in \mathbb{R}^n, \]  

(1.7)

the necessary optimality conditions are satisfied by \((\bar{y}, \bar{u})\). However, to solve (1.7) we have to know \(\bar{p}\).

1.2. Task of Optimal Control with Nonlinear Constraint

Consider the nonlinear optimization problem

\[ \min_{y, u} J(y, u) \quad \text{subject to} \quad Ay + F(y) = Bu, \quad u \in U_{ad}, \]  

(1.8)

where \(J, A\) are the same as in (1.1), \(B\) describes a continuous mapping and \(F: \mathbb{R}^n \to \mathbb{R}^n\); is continuously differentiable function such that \(Ay + F(y) - Bu = 0\) has a unique solution \(y\) for any \(u \in U_{ad}\). Furthermore, we define \(G(y) := Ay + F(y)\) and restrict \(F\) such that \(G'(y)\) is invertible for any \(y \in \mathbb{R}^n\). Then the inverse function theorem ([4], Satz 14.1) gives us the continuous differentiability of \(G^{-1}\) for any \(y \in \mathbb{R}^n\).

Almost all of the above results still hold for the nonlinear case, because we impose the restriction that \(Ay + F(y) - Bu = 0\) has a unique solution \(y\) for any \(u \in U_{ad}\). The only change is the gradient of the reduced cost functional and therefore the optimality system.

In order to derive the optimality system we consider the Lagrangian,

\[ L(y, u, p) := J(y, u) - (Ay + F(y) - Bu, p)_{\mathbb{R}^n}, \]

and its partial derivatives:

\[ \nabla_y L(y, u, p) = \nabla_y J(y, u) - (A + F'(y))^T p, \]
\[ \nabla_u L(y, u, p) = \nabla_u J(y, u) + B^T p, \]
\[ \nabla_p L(y, u, p) = Ay + F(y) - Bu. \]

The optimality system is set up from the partial derivatives of the Lagrangian.

\[
\begin{align*}
(\text{I}) & \quad Ay + F(y) - Bu = 0, \quad u \in U_{ad} \\
(\text{II}) & \quad (A + F'(y))^T p = \nabla_y J(y, u) \\
(\text{III}) & \quad (B^T p + \nabla_u J(y, u), v - u)_{\mathbb{R}^m} \geq 0 \quad \forall v \in U_{ad}
\end{align*}
\]

(1.9)

To verify the optimality system we need to show that an \(\bar{u}\) satisfying the system (1.9) meets the necessary first-order optimality condition \(\hat{J}'(\bar{u})(u - \bar{u}) \geq 0 \quad \forall u \in U_{ad}\).
The reduced cost functional takes on the form $\hat{J}(u) = J((G^{-1} \circ B)(u), u)$. Since $Ay + F(y) = Bu$ has a unique solution for any $u$, we can write $y = (G^{-1} \circ B)(u)$. Thus, for the left-hand side of (II), it holds that:

$$(A + F'(y))^T p = (G'(y))^T p = (G'((G^{-1} \circ B)(u)))^T p.$$  \hfill (1.10)

The derivative of $\hat{J}$ in direction $h$ computes as follows:

$$\hat{J}'(u)h = D_y J((G^{-1} \circ B)(u), u)(G^{-1} \circ B)(u)h + D_u J((G^{-1} \circ B)(u), u)h$$

$$= ((G^{-1} \circ B)'(u))^T \nabla_y J((G^{-1} \circ B)(u), u) + \nabla_u J((G^{-1} \circ B)(u), u)h_{\mathbb{R}^m}.$$  

It holds that $((G^{-1} \circ B)'(u))^T \nabla_y J((G^{-1} \circ B)(u), u) = B^T p$, if $p$ satisfies (II).

**Proof:** By using (1.10) in the first equality and the chain rule in the second last equality we obtain

$$(G^{-1} \circ B)'(u))^T \nabla_y J((G^{-1} \circ B)(u), u) = (G'((G^{-1} \circ B)(u)))^T p$$

$$= [(G \circ G^{-1} \circ B)'(u)]^T p = B^T p.$$  \hfill $\square$

Thus, $\bar{u}$ satisfying the optimality system satisfies

$$\hat{J}'(\bar{u})(u - \bar{u}) \geq 0 \quad \forall u \in U_{ad}.$$  

### 1.2.1. Example subject to implementation in Python

Let $a, d$ and $\xi$ be in $\mathbb{R}$ and $a, d \geq 0$. Consider the nonlinear ordinary differential equation:

$$-\ddot{y}(x) + ay(x) + dy^3(x) = \dot{u}(x), \quad x \in (0, \xi), \quad (1.11)$$

with some boundary conditions

$$\dot{y}(0) = 0 \quad \text{and} \quad \dot{y}(X) = 0.$$  \hfill (1.12)

A finite difference approximation of the above system leads to a discrete form, $Ay + F(y) = Bu$ (([9], Section 2.2) and ([7], Chapter 1,2)). The matrices $A, B$ and the function $F$ will be defined in the section 'Implemented Version' (1.2.2).

**Remark 1.2.1.** Equation (1.11) can be interpreted as the one dimensional version of the semilinear elliptic partial differential equation:

$$-\Delta \dot{y}(x, t) + a(t)\dot{y}(x, t) + d(t)\dot{y}^3(x, t) = \dot{u}(x, t), \quad x \in (0, \xi), \quad t \in (0, T), \quad (1.13)$$
with boundary conditions
\[ \hat{y}(0, t) = 0 \quad \text{and} \quad \hat{y}(X, t) = 0. \] (1.14)

A two dimensional finite difference approximation of the Laplace operator again leads to a matrix representation (([9], Section 2.2) and ([7], Chapter 3)). So the above equation \( Ay + F(y) = Bu \) is applicable as a discrete version for a variety of (partial) differential equations.

Let us come back to our example. We are interested to compute \( \hat{y} \) because solving (1.11). We are able to control \( \hat{u}(x) \) at every \( x \in (0, \xi) \). In addition, we would like \( \hat{y} \) to follow some slope, described by \( \hat{y}_s \), as good as possible.

Suppose, that \( \hat{u}, \hat{y}_s, \hat{y} \in L^2(0, \xi) \). Further suppose, we can provide that \( \hat{u} \) can take on any value in \( \mathbb{R} \), such that \( U_{ad} = \mathbb{R} \). But we want to find \( \hat{u}(x) \), where \( ||\hat{u}||_{L^2(0,\xi)} \). In order to meet our wishes we define a cost functional \( J \):

\[ J(y, u) = \frac{\alpha}{2} ||\hat{y} - \hat{y}_s||^2_{L^2(0,\xi)} + \frac{\lambda}{2} ||\hat{u}||^2_{L^2(0,\xi)}. \] (1.15)

In the discrete version of (1.15) (in section ‘Implemented Version’), the \( L^2(0,\xi) \)-norm is replaced by the standard Euclidean norm for simplicity.

The resulting optimization problem is given by:

\[ \min_{\hat{y}, \hat{u}} J(\hat{y}, \hat{u}) \quad \text{subject to} \quad -\hat{y}''(x) + a\hat{y}(x) + d\hat{y}^3(x) = \hat{u}(x), \quad x \in (0, \xi). \] (1.16)

In the next section the discrete version of the example is provided.

### 1.2.2. Implemented Version

Input parameters are: \( a, d \in \mathbb{R}^n \geq 0, y_s \in \mathbb{R}^n, \alpha \geq 0, \lambda \geq 0, \xi \in \mathbb{R} \) and \( n \in \mathbb{N} \).

The interval \([0,\xi]\) is discretized by \( n + 2 \) grid points \( 0 = x_0 < x_1 < \cdots < x_n < x_{n+1} = \xi \) with equal distance between neighboring points \( \Delta x_n = \frac{\xi}{n+2} \). The vectors \( y, y_s \) and \( u \) approximate the functions from (1.15) the following way: \( y_i \approx \hat{y}(x_i) \) for \( i \leq n \in \mathbb{N} \) (analog for \( \hat{y}_s \) and \( \hat{u} \)).

A finite difference approximation of (1.11) is given by \( Ay + F(y) = Bu \), where the function \( F \) is defined as

\[ F : \mathbb{R}^n \to \mathbb{R}^n; y \mapsto \begin{pmatrix} d_1 y_1^2 \\ \vdots \\ d_n y_n^2 \end{pmatrix}, \quad F'(y) = \begin{pmatrix} 3d_1 y_1^2 & 0 & \cdots & 0 \\ 0 & \ddots & \cdots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & 3d_n y_n^2 \end{pmatrix}, \]
and matrices $A$ and $B$ are defined as

$$A = \frac{1}{\Delta x^2_n} \begin{pmatrix} 2 + a_1 & -1 & 0 & \cdots & \cdots & \cdots & 0 \\ -1 & 2 + a_2 & -1 & 0 & \cdots & \cdots & \vdots \\ 0 & -1 & 2 + a_3 & -1 & 0 & \cdots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & -1 & 2 + a_{n-2} & -1 & 0 \\ 0 & \cdots & \cdots & -1 & 2 + a_{n-1} & -1 & \vdots \\ 0 & \cdots & \cdots & \cdots & 0 & -1 & 2 + a_n \end{pmatrix}$$

and $B = I_n$.

The optimization task is:

$$\min_{y,u} J(y, u) := \frac{\alpha}{2} ||y - y_s||^2 + \frac{\lambda}{2} ||u||^2 \quad \text{subject to} \quad Ay + F(y) = Bu, \quad u \in \mathbb{R}^n. \quad (1.17)$$

The resulting optimality system is given by

1. \quad (I) $Ay + F(y) - Bu = 0,$
2. \quad (II) $(A + F'(y))^T p = \alpha(y - y_s),$
3. \quad (III) $B^T p + \nabla_u J(y, u) = 0.$
2. Nonlinear Conjugate Gradient Method

Since the linear conjugate gradient (CG) method is effective solving equations like $Ax = b$, where $A \in \mathbb{R}^{n \times n}$, $A^T = A \succ 0$ and $b \in \mathbb{R}^n$; it can be used to compute the gradient or, if the cost functional is quadratic, even the solution of the finite dimensional optimal control problem. Therefore the first part of this section states the essentials of the method. The linear CG method is in general not applicable for the general case (1.5). Thus, a natural extension for finding a solution to (1.5) is the nonlinear version of the conjugate gradient method. The second part of this section gives a short summary of the literature on nonlinear versions of the conjugate gradient method and compares them to their performance minimizing the Rosenbrock function.

2.1. Linear Conjugate Gradient Method

The linear conjugate gradient method is used to minimize strictly convex quadratic functions like

$$f(x) = \frac{1}{2}x^T Ax - b^T x + c,$$

where $A \in \mathbb{R}^{n \times n}$, $A^T = A \succ 0$, $b \in \mathbb{R}^n$ and $c \in \mathbb{R}$.

Because of $\nabla^2 f(x) = A \succ 0$, (2.1) is strictly convex. Thus, the necessary condition $\nabla f(x^*) = 0 \iff Ax^* - b = 0$ is sufficient. Such that the optimization task broke down to solve the linear equation system $Ax = b$. The linear CG method minimizes $f$ by successively minimizing it along the individual directions in an $A$-conjugate set\(^1\). Given $x_0 \in \mathbb{R}^n$ and an $A$-conjugate set the algorithm generates a sequence $\{x_k\}$ by setting $x_{k+1} = x_k + \alpha_k p_k$, where $\alpha_k$ is the (exact) one-dimensional minimizer of $f$ along the ray $x_k + \alpha p_k$. It is given by $\alpha_k = -\frac{\nabla f(x_k)^T p_k}{p_k^T A p_k}$.\(^2\) Below is a pseudo code, which is a bit more economical than the conventional textbook algorithm (See ([8], Chapter 5, Algorithm 5.1)). Further, following ([12], Kapitel 6, Bemerkung 6.3), the formula to compute $\beta_k$ is changed. And the exact step length $\alpha_k$ is computed with a different formula ([12], Kapitel 6.2). The presented pseudo code is based on ([12], Kapitel 6.2, Algorithm 6.1).

\(^1\)See ([8], p.103) A set of nonzero vectors $\{p_0, p_1, \ldots, p_l\}$ is said to be conjugate with respect to the symmetric positive definite matrix $A$ if $p_i^T A p_j = 0$ for all $i \neq j$.

\(^2\)See ([8], p.56)
Require $x_0 \in \mathbb{R}^n$;
Choose $\varepsilon_{rel} \geq \varepsilon_{abs} > 0$ and $\maxiter$ (maximum number of iterations);
Set $\nabla f_0 = \nabla f(x_0)$, $p_0 = -\nabla f_0$, $counter = 0$;
while $counter < \maxiter$:
    $d_k = Ap_k$ #product is needed twice
    $\alpha_k = \frac{||\nabla f_k||^2}{p_k^T d_k}$ #exact stepsize
    $x_{k+1} = x_k + \alpha_k p_k$ #next iteration
    $\nabla f_{k+1} = \nabla f_k + \alpha_k d_k$ #next gradient
    if $||\nabla f_{k+1}||_2 \geq \varepsilon_{rel} ||\nabla f_0||_2 + \varepsilon_{abs}$ then:
        $\beta_k = \frac{(\nabla f_{k+1} - \nabla f_k)^T \nabla f_{k+1}}{||\nabla f_k||^2_2}$ #needed for $p_{k+1}$
        $p_{k+1} = -\nabla f_{k+1} + \beta_k p_k$ #next $A$-conjugate direction
    else:
        $counter = counter + 1$
end

text:

\textbf{Remark 2.1.1.} 1. ([8], Theorem 5.1): For any $x_0 \in \mathbb{R}^n$ the sequence $\{x_k\}$ generated by the conjugate direction algorithm converges to the solution $x^*$ of the linear system $Ax = b$ in at most $n$ steps.

2. ([8], Theorem 5.4): If $A$ has only $r$ distinct eigenvalues, then the linear CG-iteration will terminate at the solution in at most $r$ iterations.

3. If some eigenvalues are clustered, then the algorithm is likely to give a good estimation after (distinct eigenvalues + eigenvalue clusters) iterations ([8], p.117). Furthermore, the algorithm terminates after $m$ iterations, if $b$ can be written as linear combination of maximum $m$ eigenvectors and the algorithm is initialized with $0 = x_0 \in \mathbb{R}^n$ ([12], Kapitel 6.2).

4. ([8], Theorem 5.5): If $A$ has eigenvalues $\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n$, we have that
   \[ ||x_{k+1} - x^*||_A^2 \leq \left( \frac{\lambda_{n-k} - \lambda_1}{\lambda_{n-k} + \lambda_1} \right) ||x_0 - x^*||_A^2. \]

5. It can be shown that
   \[ ||x_k - x^*||_A^2 \leq 2 \left( \frac{\sqrt{\kappa(A)} - 1}{\sqrt{\kappa(A)} + 1} \right)^k \leq ||x_0 - x^*||_A. \]
   This bound often leads to an overestimation of the error ([8], p.117).
2.2: Nonlinear Conjugate Gradient Method

6. Preconditioning is a technique to improve the distribution of the eigenvalues. It is heavily problem dependent, but can increase the performance of the algorithm significantly. See ([8] Chapter 5.1), ([9], Chapter 9) on preconditioning.

2.2. Nonlinear Conjugate Gradient Method

The aim is to make use of the conjugate gradient method not only for strictly convex functions, but for general differentiable functions. There are only two changes of the algorithm in the transition from the linear-CG to the nonlinear-CG, namely:

1. As the function to be minimized is not strictly convex anymore, there is no formula for the exact step length. The step size, $\alpha_k$ needs to be approximated by a line search algorithm. Later we will see that it must be a Wolfe-Powell line search using the strong Wolfe conditions.

2. The gradient of $f$ at $x_{k+1}$ cannot be written dependent on $\nabla f_k$ anymore. A function evaluation or an approximation of $\nabla f(x_{k+1})$ is required instead.

The first part of this section considers the Fletcher-Reeves method following ([8], chapter 5.2). The second part provides a compilation of alternative nonlinear-CG methods. The third part gives a short numerical comparison of the methods.

2.2.1. Fletcher-Reeves Method

For this section we assume

i. The level set $L := \{x | f(x) \leq f(x_0)\}$ is bounded;

ii. the objective function $f$ is twice continuously differentiable.

Under the above assumptions mentioned above, ([8], Lemma 3.1) ensures the existence of a step length $\alpha_k$ satisfying the strong Wolfe conditions. A pseudo code of the Fletcher-Reeves Method is given below (based on ([8], Algorithm 5.4)):

| Require $x_0 \in \mathbb{R}^n$; |
| Choose $\varepsilon > 0$ and maxiter; |
| Set $\nabla f_0 = \nabla f(x_0)$, $p_0 = -\nabla f_0$, $\text{counter} = 0$; |
| while $\nabla f_k > \varepsilon$ and counter < maxiter |
| call linesearch to determine $\alpha_k$ |
| $x_{k+1} = x_k + \alpha_k p_k$ #next iteration |
| calculate $\nabla f_{k+1}$ |
| $\beta_{k+1} = \frac{||\nabla f_{k+1}||^2}{||\nabla f_k||^2}$ #needed for $p_{k+1}$ |
| $p_{k+1} = -\nabla f_{k+1} + \beta_{k+1} p_k$ #next descent direction |
| $k = k+1$ |
| end |
If the objective function is strictly convex and \( \alpha_k \) the exact step length, then the algorithm is equivalent to the linear CG method.

Since the algorithm does not make use of the exact step length, we lose the property

\[
\nabla f_k^T p_k = \nabla f_k^T ( -\nabla f_k + \beta_k^{FR} p_{k-1} ) = -\|\nabla f_k\|^2 + \beta_k^{FR} \nabla f_k^T p_{k-1},
\]

it is not possible to ensure \( p_k \) to be a descent direction without demanding certain criteria for the line search algorithm. The following quotes ([8], Lemma 5.6):

**Lemma 2.2.1.** Suppose that FR-Algorithm is implemented with a step length \( \alpha_k \) that satisfies the strong Wolfe conditions

\[
\begin{align*}
 f(x_k + \alpha_k + p_k) &\leq c_1 \alpha_k \nabla f_k^T p_k \quad (2.2) \\
 |\nabla f(x_k + \alpha_k p_k)^T p_k| &\leq -c_2 \nabla f_k^T p_k \quad (2.3)
\end{align*}
\]

\( 0 < c_1 < c_2 < \frac{1}{2} \). Then the method generates descent directions \( p_k \) that satisfy the following inequalities:

\[
- \frac{1}{1-c_2} \leq \frac{\nabla f_k^T p_k}{\|\nabla f_k\|^2} \leq \frac{2c_2-1}{1-c_2} < 0 \quad \text{for all} \quad k = 0, 1, \ldots \quad (2.4)
\]

*Proof:* See ([8], Lemma 5.6). \( \square \)

Since the function \( g: \mathbb{R} \to \mathbb{R}; x \mapsto \frac{2x-1}{1-x} \) is monotonically increasing on the interval \([0, \frac{1}{2}]\) and it holds that \( g(0) = -1 \) and \( g(\frac{1}{2}) = 0 \), an algorithm satisfying the strong Wolfe conditions ensures \( p_k \) to be a descent direction. A Wolfe-Powell line search algorithm can satisfy these requirements. For a pseudo code, see ([11], Algorithm 9.3).

**Remark 2.2.2.** At this point the greatest weakness of the FR algorithm should be pointed out. It may happen that the algorithm repeatedly generates long sequences of unproductive iterates. To see that, let \( \theta_k \) denote the angle between the steepest descent direction \( -\nabla f_k \) and \( p_k \). It is defined by \( \cos \theta_k = \frac{-\nabla f_k^T p_k}{\|\nabla f_k\| \|p_k\|} \). Suppose \( p_k \) is almost orthogonal to \( -\nabla f_k \), then \( \cos \theta_k \approx 0 \). By multiplying both sides of (2.4) with \( \|\nabla f_k\| \|p_k\| \), we obtain

\[
- \frac{1}{1-c_2} \leq \cos \theta_k \leq \frac{2c_2-1}{1-c_2}, \quad \text{for all} \quad k = 0, 1, \ldots
\]

This inequality implies

\[
\cos \theta_k \approx 0 \quad \text{if and only if} \quad \|\nabla f_k\| < < \|p_k\|. \quad (2.5)
\]

\( ^3 \)If \( \nabla f_k^T p_{k-1} > 0 \) and \( \beta_k^{FR} \nabla f_k^T p_{k-1} \geq \|\nabla f_k\|^2 \), then \( p_k \) is no descent direction.
Dependent on the objective function the strong Wolfe conditions can lead to a very tiny $\alpha_k$ such that $x_{k+1} = x_k + \alpha_k p_k \approx x_k$, if $\cos \theta_k \approx 0$. For a continuously differentiable objective function this means $f_{k+1} \approx f_k$ implying $\nabla f_{k+1} \approx \nabla f_k$ and therefore $\beta^{FR}_{k+1} \approx 1$. Ultimately, we have $p_{k+1} = -\nabla f_{k+1} + \beta^{FR}_{k+1} p_k \approx p_k$ and thus a long sequence of unproductive iterates will be generated.

### Global Convergence

For convergence analysis it is necessary to additionally assume

iii. the objective function $f$ is Lipschitz continuously differentiable in some open neighborhood $N$ of $L$.

Zoutendijk’s theorem is essential for convergence analysis of any algorithms using Wolfe conditions. It is used later in this work and in the proof of global convergence of the FR-algorithm. It is stated below ([8], Theorem 3.2).

**Theorem 2.2.3.** (Zoutendijk) Consider any iteration of the form $x_{k+1} = x_k + \alpha_k p_k$, where $p_k$ is a descent direction and $\alpha_k$ satisfies the Wolfe conditions

$$ f(x_k + \alpha_k p_k) \leq c_1 \alpha_k \nabla f_k^T p_k \quad (2.6) $$

$$ \nabla f(x_k + \alpha_k p_k)^T p_k \geq c_2 \nabla f_k^T p_k \quad (2.7) $$

$0 < c_1 < c_2 < 1$. Suppose that $f$ is bounded below in $\mathbb{R}^n$ and that $f$ is continuously differentiable in an open set $N$ containing the level set $L = \{ x | f(x) \leq f(x_0) \}$, where $x_0$ is the starting point of the iteration. Also assume that the gradient $\nabla f$ is Lipschitz continuous on $N$. Then

$$ \sum_{k \geq 0} \cos^2 \theta_k \| \nabla f_k \|^2 < \infty. \quad (2.8) $$

**Proof:** See ([8], Theorem 3.2). \hspace{1cm} □

The following theorem reflects ([8], Theorem 5.7) respectively ([2], Theorem 2) in meaning.

**Theorem 2.2.4.** Suppose that assumptions (i.) and (ii.) hold, and that the FR algorithm is implemented with a line search that satisfies the strong Wolfe conditions (2.2), (2.3), with $0 < c_1 < c_2 < \frac{1}{2}$. Then

$$ \liminf_{k \to \infty} \| \nabla f_k \| = 0 \quad (2.9) $$

**Proof:** See ([8], Theorem 5.7). \hspace{1cm} □
2.2.2. Alternative Nonlinear-CG Methods

The global convergence result above can be extended to any choice of $\beta_k$ satisfying $|\beta_k| \leq \beta_k^{FR}$. In general it follows from Zoutendijk’s condition that $\liminf_{k \to \infty} ||\nabla f_k|| = 0$, if one can show that there exist constants $c_4, c_5 > 0$ such that

$$\cos \theta_k \geq c_4 \frac{||\nabla f_k||}{||p_k||} \cdot \frac{||\nabla f_k||}{||p_k||} \geq c_5 > 0 \quad k = 1, 2, \ldots \quad ([8], p.130).$$

As pointed out in remark (2.2.2), the FR algorithm tends to perform a sequence of unproductive steps once the descent direction is almost orthogonal to the steepest descent direction. To overcome this property we can incorporate a restart in the FR algorithm. That is periodically setting $\beta_k^{FR}$ to zero. By doing so we erase old, possibly unbeneicial information and "restart" the algorithm with the steepest descent direction.

**Lemma 2.2.5.** Any nonlinear conjugate gradient algorithm which updates the descent direction like $p_{k+1} = -\nabla f_{k+1} + \beta_{k+1} p_k$ converges, if the algorithm restarts periodically.

**Proof:** Let $k_1, k_2$, and so on denote the iterations on which restarts occur, meaning $\beta_{k_i} = 0$ for $i \in \mathbb{N}$. Since

$$\cos^2 \theta_{k_i} = \frac{(-\nabla f_k p_k)^2}{(||\nabla f_k||||p_k||)^2} = \frac{(||\nabla f_k||^2)^2}{(||\nabla f_k||^2)^2} = 1,$$

from Zoutendijk’s theorem follows

$$\sum_{k=k_1, k_2, \ldots} ||\nabla f_k||^2 < \infty \Leftrightarrow \liminf_{k \to \infty} ||\nabla f_k|| = 0. \quad \square$$

For the remaining of this section we define two restart parameters. At first $r_p$, which is defined to be any prime number and second $tol$, a tolerance which can be used to have quicker restarts in case of a bad descent direction. Reasonable values would be $r_p = 23$ and $tol = 10^{-3}$.

**Fletcher-Reeves Method with Restart**

A variant of the Fletcher-Reeves algorithm with periodic restart is given by:

$$\beta_{k+1}^{FRR} = \begin{cases} 0 & \text{if } k + 1 \in \{i \star r_p \mid i \in \mathbb{N}\} \lor |\beta_k^{FR} - 1| < tol \\ \beta_{k+1}^{FR} & \text{else} \end{cases} \quad (2.10)$$
Polak-Ribiere Method and Variants

The Polak-Ribiere method ([8], p.122) is an important variant of the FR method. The only adjustment of the FR algorithm is to define the parameter $\beta$ as follows:

$$
\beta_{k+1}^{PR} = \frac{\nabla f_{k+1}^T (\nabla f_{k+1} - \nabla f_k)}{||\nabla f_k||^2}.
$$

(2.11)

One desirable property is that $\beta_{k+1}^{PR} \approx 0$, if the descent direction $p_k$ is almost orthogonal to the steepest descent direction. Thus, the algorithm automatically performs a restart after a bad direction. Although it holds that $\beta_{k+1}^{PR} = \beta_{k+1}^{FR}$, if the objective function $f$ is strongly convex and the line search is exact. It is not possible to proof global convergence ([8], Theorem 5.8).

But we have global convergence for the following variant often referred to as PR+ algorithm ([8], Section 5.2):

$$
\beta_{k+1}^{PR+} = \max\{\beta_{k+1}^{PR}, 0\}.
$$

(2.12)

As mentioned above, the following variant is also global convergent:

$$
\beta_k = \begin{cases} 
-\beta_k^{FR} & \text{if } \beta_k^{PR} < -\beta_k^{FR} \\
\beta_k^{PR} & \text{if } |\beta_k^{PR}| \leq \beta_k^{FR} \\
0 & \text{if } |\beta_k^{PR}| \leq \beta_k^{FR} \land |\beta_k^{PR}| \leq tol \\
\beta_k^{FR} & \text{if } \beta_k^{PR} > \beta_k^{FR}
\end{cases}
$$

(2.13)

Other variants of the CG method

For the two variants below we define $\hat{y}_k := \nabla f_{k+1} - \nabla f_k$. Dai and Yuan ([3]) proposed the following formula to update $\beta$:

$$
\beta_{k+1}^{DY} = \frac{||\nabla f_{k+1}||^2}{\hat{y}_k^T p_k}.
$$

(2.14)

Hager and Zhang ([5]) proposed:

$$
\beta_{k+1}^{N} = \left( \hat{y}_k - 2p_k \frac{||\hat{y}_k||^2}{\hat{y}_k^T p_k} \right)^T \nabla f_{k+1} \frac{||\hat{y}_k||^2}{\hat{y}_k^T p_k}.
$$

(2.15)

2.2.3. Performance analysis

The table below compares the above nonlinear CG methods on their performance minimizing the Rosenbrock function,

$$
f(x) = (1 - x_1)^2 + 100(x_1 - x_2^2)^2,
$$

(2.16)
where $x = (x_1, x_2) \in \mathbb{R}^2$. Additionally, a steepest descent method is considered. It is realised by setting $\beta = 0$ in every iteration.

The used line search algorithm is based on the 'wolfe' algorithm from the 'oppy'-library ([14]).

|       | Iterations | Solution               | $||\nabla f(x_{sol})||_2$ | Mean$(f_k - f_{k-1})$ | Median$(f_k - f_{k-1})$ |
|-------|------------|------------------------|---------------------------|------------------------|--------------------------|
| FR    | 525        | $(1.00000086, 1.00000172)$ | $9.538 \times 10^{-7}$   | 18,857,637.72          | $6.078 \times 10^{-3}$  |
| FRR   | 144        | $(0.99999901, 0.99999802)$ | $8.976 \times 10^{-7}$   | 68,751,804.17          | $8.844 \times 10^{-4}$  |
| PR    | 1415       | $(1.00000002, 1.00000004)$ | $9.980 \times 10^{-7}$   | 6,996,650.04           | $2.314 \times 10^{-15}$ |
| PR+   | 1550       | $(1.00000027, 1.00000054)$ | $9.980 \times 10^{-7}$   | 6,387,264.39           | $1.170 \times 10^{-15}$ |
| FR-PR | 125        | $(0.99999971, 0.99999941)$ | $5.950 \times 10^{-7}$   | 79,202,078.41          | $3.423 \times 10^{-8}$  |
| DY    | 1515       | $(0.9999992, 0.9999984)$  | $7.383 \times 10^{-7}$   | 6,534,824.95           | $3.132 \times 10^{-3}$  |
| N     | 705        | $(1.00000076, 1.00000153)$ | $8.919 \times 10^{-7}$   | 14,042,921.70          | $1.871 \times 10^{-8}$  |
| GRAD  | 458,030    | $(1.00000078, 1.00000156)$ | $9.784 \times 10^{-7}$   | 21,614.88              | $5.851 \times 10^{-5}$  |

Table 2.1.: Wolfe line search parameters: $c_1 = 10^{-4}$, $c_2 = 0.4$, init = 1 and amax = 30.

Nonlinear CG parameters: $eps = 10^{-6}$, termination condition, and $x_0 = (100,500)$, initial guess.

Clearly, no conclusion can be drawn on which formula to update $\beta$ is the best. More sophisticated numerical evaluations can be found in ([1]) or ([8], Chapter 5, Table 5.1). The purpose of these results is to illustrate, that there may be a significant advantage of nonlinear conjugate gradient methods compared to the steepest descent method. And typically, more than half of all steps toward the minimum are teeny tiny. In the later numerical results the 'FR_PR' update formula is used because it seems quite effective and stable.
3. Implementation in Python

Python is an open source software and thus whenever a written code is publicly accessible, anybody within the Python community can use or optimize it. The advantage of Python compared to other scientific computing software is that it is open source and widely in use. This probably arises from its comparatively simple superstructure. In the proceeding basic knowledge about Python is assumed. The Python surface Spyder was used.

3.1. Packages and Formats

In practice the matrices from (1.8) are very large, \( 500 \times 500 \) for example, and sparsely populated. The 'sparsematrix'-format only stores values different from zero to improve performance in case of sparse matrices. A rich toolbox including a building kit\(^1\) and many linear algebra operations\(^2\) can be found in the SciPy Library package.

In general scientific computing in Python builds on three packages ([13]):

1. **NumPy\(^3\)**, the fundamental package for numerical computation. It defines the numerical array and matrix types and basic operations on them.

2. **Scipy Library\(^4\)**, a collection of numerical algorithms and domain-specific toolboxes, including signal processing, optimization, statistics, and much more.

3. **Matplotlib\(^5\)**, a mature and popular plotting package that provides publication-quality 2-D plotting, as well as rudimentary 3-D plotting.

The packages can be imported and used, illustrated on the `numpy` functions "array" and "norm", the following way:

```python
import numpy as np  # import the whole package
from numpy.linalg import norm  # import a single function from numpy
x = np.array([1, 2])
normx = norm(x)
```

In case an self-written file should be imported into another file, it needs to be stored in the same folder.

\(^1\)https://docs.scipy.org/doc/scipy/reference/sparse.html
\(^2\)https://docs.scipy.org/doc/scipy/reference/sparse.linalg.htmlmodule-scipy.sparse.linalg
\(^3\)https://numpy.org/
\(^4\)https://www.scipy.org/scipylib/index.html
\(^5\)https://matplotlib.org/
3.2: Task of Optimal Control

The implementation of the finite dimensional optimal control problem in (1.2.2) consists of two files: 'costfunctional' and 'costfunctional_method_library'. The latter just stores functions to build the required matrices in sparse format and is imported in 'costfunctional' as 'cml'.

As the user can choose input parameters, the problem needs to be initialised. This is accomplished, by using a Python 'class' that restores all input parameters and creates and stores all information resulting from them. In (3.4), the use of the 'class' will be illustrated. For detailed documentation of the 'class' see Appendix (B.1).

Since solving the optimal control problem comes down to minimize the reduced cost functional, implementing the task of optimal control comes down to define functions that compute the reduced cost function and the reduced gradient. Both need the solution $y$, for given $u$, of the constraint $Ay + F(y) = Bu$ to be computed. In order to avoid unnecessary computation of $y$, only one function is implemented. The user can choose if she wants to compute: only the value of the cost functional ('ind = 0'), only the reduced gradient ('ind = 1') or both ('ind = 2'). The function requires a vector argument 'u' (control vector), a data-object which contains all information about (1.2.2) and an indicator, 'ind', specifying the output.

```python
def CFandRG(u, data, ind = 0):
    A Newton-CG\(^6\) computes the solution $y$ for given $u$ of the equation $Phi := Ay + F(y) - Bu = 0$.

    $y$, count = NewtonSolver_cg_own(data.Phi, data.dyPhi, data.y0_nw, u, data.eps_nw, 100)

    If 'ind = 0' is set, then the cost functional will be evaluated and the function will return a 'dictionary' containing the value of the cost functional and the to $u$ associated state $y$.

    if ind == 0:
        cf = ((data.alpha/2) * (np.linalg.norm(y - data.yT))**2
             + (data.lamb /2) * (np.linalg.norm(u))**2)
        return {'func': cf, 'state': y}

    If rather 1 or 2 is set for 'ind', the solution to the adjoint equation is calculated by a CG-Algorithm\(^7\). Also the reduced gradient gets computed.

    if ind >= 1:
        c = data.alpha*(y - data.yT)
        p, info = cgo.cg(sc.transpose(data.dyPhi(y)), c, data.y0_cg)
```

\(^6\)See (B.1), $NewtonSolver_{-cg\_own}$.  
\(^7\)See (B.1), $cg$. 

---

20
rg = data.B_T * p + data.lamb * u

If 'ind = 1', then the function returns a 'dictionary' containing the reduced gradient, the to u associated state y and the solution to the adjoint equation. If 'ind = 2', then additionally the value of the cost functional gets computed. The function returns a 'dictionary' containing the same values as in the 'ind = 1' case and moreover the value of the cost functional.

if ind == 1:
    return {'rgrad': rg, 'adjoint': p}
if ind == 2:
    cf = ((data.alpha/2) * (np.linalg.norm(y - data.yT))**2
          + (data.lamb/2)*np.linalg.norm(u))**2
    return {'func': cf, 'state': y, 'adjoint': p, 'rgrad': rg}

If the user fails to choose a valid value for 'ind', the algorithm returns with an error message.

return print('Error: wrong input for "ind"')

3.3. Nonlinear Conjugate Gradient Method

The nonlinear conjugate gradient method consists of the file 'NLCG', containing the method itself, and three supplementary files containing the wolfe-powell line search algorithms. Namely: 'oppy_unconOpt_wolfe_copy1', 'linesearch' and 'oppy_unconOpt_wolfe'. The 'oppy_unconOpt_wolfe_copy1' file is based on the 'wolfe' method that can be found in the 'oppy'-library ([14]). Some modifications to fit the problem were made but the general structure has not changed. A pseudo code can be found in ([11], page 39). The file 'linesearch' contains the SciPy Library line search algorithms. Again there were made minor changes but not influencing any computation steps. It is based on ([8], Section 3.5). The file 'oppy_unconOpt_wolfe' is the original 'wolfe' line search from the 'oppy'-library. In order to be able to use the method for a function like 'CFandRG'(discussed above in (3.2)) and functions where the function value and the gradient of the function are calculated separately, the input parameter 'dfhandle' is also used as key indicating what input is given.

def nlcg(fhandle, x0, dfhandle, data = None, ind = 0, \\method = 'FR_PR', maxiter = 10e3, eps = 1e-3, \
    rp1 = 23, rp2 = 1e-4, disp = False, fargs=()):

If 'dfhandle' is set to 'None', then 'fhandle' needs to be a function like 'CFandRG', and 'data' is required as input parameter for 'CFandRG'. The other non-optional input pa-
3.3: Nonlinear Conjugate Gradient Method

Parameters are a function handle, ‘fhandle’, and ‘x0’, the starting point for the iterations. It can be chosen which line search algorithm is used by the input parameter ‘ind’ (0: oppy library; else: Scipy Library). The formula used to compute $\beta$ can be changed in the 5<sup>th</sup> input argument. All formulas from (2.2) can be chosen. Maximum number of iterations, ‘maxiter’, is per default set to $10^3$. The input ‘eps’ sets the termination condition and is set to 0.001 per default. The two following inputs, ‘rp1’ and ‘rp2’, are restart parameters and per default set to the values used in the code example. If the user wants to see the proceedings while the algorithm is at work, ‘disp’ can be set ‘True’. Additional input arguments for ‘fhandle’ respectively ‘dfhandle’ can be stored in ‘fargs’. Dependent on ‘method’, the formula to compute $\beta$ is generated the following way (illustrated on (2.10)):

```python
elif method == 'FRR':
    def bta(gk, gk1, norm_gk, norm_gk1, p, tol, rp, \ 
               current_iterate):
        if type(k/rp) == int:
            beta = 0
        else:
            betaFR = norm_gk1**2 / norm_gk**2
            if abs(betaFR - 1) < 0.0001:
                beta = 0
            else:
                beta = betaFR
        return beta
```

Before the loop is entered the function handle and its derivative are redefined inside the function; illustrated on the derivative:

```python
def df(x):
    if dfhandle == None:
        l = fhandle(x, data, 1)['rgrad']
    else:
        l = dfhandle(x, *fargs)
    return l
```

The remaining code follows the pseudo code in (2.2.1). The full code can be found in the Appendix (B.2).

One particularity about the Scipy Wolfe Powell is that it sometimes does not converge and as consequence returns ‘None’ as step size. If that happens the ‘oppy’ line search is used.

```python
if ind == 0:
    if dfhandle == None:
        #modified oppy wolfe line search
```
3.4. Main File

In the 'main' file we need to import the two files 'costfunctional' and 'NLCG'. Then the parameters: $n$, $m$, $a$, $d$, $T$, $\alpha$, $\lambda$ and $y_s$ (vectors need to be column vectors) from (1.2.2), $y_0_{nw}$, initial guess for 'Newton Solver', $y_0_{cg}$, initial guess for the 'CG', and $\epsilon_{nw}$, tolerance for 'Newton Solver', need to be set. Having set the parameters, we can use the class 'Data' from 'costfunctional' to initialise the optimal control problem (1.2.2):

```python
data = Data(n, m, T, ys, alpha, lamb, a, d, y0_nw, y0_cg, 
            eps_nw, eps_ncg)
```

To start the nonlinear CG, we need an initial guess and a tolerance for the termination condition.

```python
def_newton_cg = 1e-4 #tolerance
import numpy as np
u = np.ones((n,1)) #initial guess
```

Then we use the nonlinear CG method to get the optimal control.

```python
x_sol, counter = nlcg(CFandRG, u, None, data, 1, method = 'FR_PR', 
                       maxiter = 150, eps = def_newton_cg, disp = True)
```

And finally we get the associated state to the optimal control by the 'Newton Solver' (needs to be imported from 'costfunctional').

```python
y, count = NewtonSolver_cg_own(data.Phi, data.dyPhi, data.y0_nw, 
                                 x_sol, data.eps_nw, 5000)
```
4. Numerical Results

In this section numerical results of the implemented nonlinear conjugate gradient method, minimizing (1.2.2), are presented. In order to provide some comparability an initial parameter set is used for all results. Only parameters that are different from the initial set are explicitly mentioned in each result.

Initial parameter set: \( n = 200, \ m = n, \ X = 12, \ y_s(x) = \begin{cases} 12 - x & x \in [6, 12], \\ 0 & \text{else} \end{cases}, \ \alpha = 99, \lambda = 0.1, \ a = 5, \ d = 0.5, \ y_{0\_nw} = 0, \ y_{0\_cg} = 0, \ e_{ps\_nw} = 1e - 10, \ e_{ps\_ncg} = 1e - 5 \) and the is initial guess for the nonlinear CG is \( u = 0 \).

4.1. Linear Constraint

First we consider the linear case, meaning \( d = 0 \). As already pointed out, we can solve this problem by solving a linear equation system. For the first result the ‘numpy’ function ‘linalg.solve’ is used to solve the linear equation system defined in (1.1.7) and \( a = 0 \) was set.

The value of the cost functional and the norm of the gradient in the solution was computed in less than a second and is given below.

\[
\hat{J}(\bar{u}) \approx 1884.2972 \quad \text{and} \quad \|\nabla \hat{J}(\bar{u})\| \approx 8.83840e - 10
\]
4.2: Nonlinear Constraint

With the change in $a$ removed and the implemented nonlinear gradient method used, the following result was obtained.

![Graph of optimal state and optimal control](image)

(a) Slope of associated state $y$ to optimal control $u$ and target slope $y_s$.

(b) Slope of optimal control $u$.

Figure 4.2.: Linear Constraint; $d = 0$; nonlinear CG.

The value of the cost functional and the norm of the gradient at the solution were computed in about 7.65 seconds and the algorithm performed 53 iterations.

$$
\hat{J}(\bar{u}) \approx 3722.2001 \quad \text{and} \quad ||\nabla \hat{J}(\bar{u})|| \approx 8.6351e-06 \quad (4.2)
$$

Solving the linear equation system in this case took a good second and led to a slightly better solution. The norm of the gradient at the solution is approximately $2.2137e - 12$. Since there are many highly sophisticated methods to solve such 'linear' problems and a rather naive approach to solve the considered problem performs better than the implemented algorithm, it is recommended to use one of those for such problems.

4.2. Nonlinear Constraint

The result of solving the optimal control problem (1.2.2) using the initial parameter set from above is presented below. The formula to compute beta was chosen to be (2.13) (Fletcher-Reeves Polak-Ribiere).

The value of the cost functional and the norm of the gradient at the solution were computed in about 35.4 seconds and the algorithm performed 52 iterations.

$$
\hat{J}(\bar{u}) \approx 9533.8796 \quad \text{and} \quad ||\nabla \hat{J}(\bar{u})|| \approx 8.3454e - 06. \quad (4.3)
$$

In all results of this section the optimal state $y$ fails to follow the exact slope of $y_s$. This behavior is expected due to the constraint: $y$ is an approximation of a twice differentiable function. However, $y$ follows the slope of $y_s$ better when either $\alpha$ gets bigger or $\lambda$ gets
4.2: Nonlinear Constraint

(a) Slope of associated state $y$ to optimal control $u$ and target slope $y_s$.

(b) Slope of optimal control $u$.

Figure 4.3.: Nonlinear Constraint; nonlinear CG.

(a) Behavior of $||\nabla \hat{J}(u_k)||$ at every iteration.

(b) Behavior of $\hat{J}(u_k)$ at every iteration.

Figure 4.4.: Nonlinear Constraint; behavior of $||\nabla \hat{J}(u_k)||$, $\hat{J}(u_k)$; nonlinear CG.

A too small $\lambda$, e.g. $\lambda = 0.0001$, makes the optimization task very difficult. Then $u$ has more freedom to follow any slope and therefore it is increasingly difficult for the algorithm to converge towards the minimum.

The most important property of conjugate gradient methods is their low memory requirement. With the current state of the art, this makes the algorithm extremely attractive for large scale problems like optimal control problems. Unfortunately, the line search algorithm may need many function evaluations, what can make the nonlinear CG horribly slow. Therefore, it is essential to make as little as possible function evaluations inside the algorithm; to make sure function evaluations are computed very quick and to make sure the line search does not take too much time. In the considered example function evaluations are heavily dependent on the ‘Newton-CG’ method (to solve the constraint $Ay + F(y) = Bu$).

A quick convergence of the method builds on a fast working, highly accurate CG method which then leads to local quadratic convergence of the ‘Newton’ method.
5. Conclusion

The implemented Python algorithm is able to solve finite dimensional optimal control problems with nonlinear constraint in reasonable time. Conjugate gradient methods, linear and nonlinear, are well suitable for such problems. Due to the low memory requirements they perform well on large scale problems. They can not only be applied beneficially in the context of optimal control, but in general for large scale optimization tasks.

Implementing the algorithm is demanding. If the algorithm does not behave as expected, there may be many possible sources of the mistake. But Python is open source, and there are already working algorithms that can help identify errors and provide ideas for improving your own algorithm.

In the sense of open source, the presented algorithm is a first version and must be modified or optimized if it is to be used successfully in more difficult applications.
Bibliography


[14] https://www.mathematik.uni-konstanz.de/volkwein/python/, last access on 24.01.2020
A. Notation

- \((u,v)_{\mathbb{R}^m} = \sum_{i=1}^{m} u_i v_i\) \(\mathbb{R}^m\)-scalar product
- \(|\cdot|\) Euclidean norm
- \(T\) transpose
- \(0_{n \times m}\) \(n \times m\) zero matrix
- \(I_n\) \(n \times n\) unit matrix

Derivatives of a function, \(f : \mathbb{R}^m \to \mathbb{R}\), are denoted as:

- \(D_i = \frac{\partial}{\partial x_i}\), \(D_x = \frac{\partial}{\partial x}\) partial derivatives
- \(f'(x) = (D_1 f(x), \ldots, D_m f(x))\) derivative (evaluated at \(x\))
- \(\nabla f(x) = f'(x)^T\) Gradient (evaluated at \(x\))
- \(\nabla^2 f(x) = \begin{pmatrix} D_{11} f(x) & \cdots & D_{1m} f(x) \\ \vdots & \ddots & \vdots \\ D_{m1} f(x) & \cdots & D_{mm} f(x) \end{pmatrix}\) Hessian matrix (evaluated at \(x\))

For functions \(f = f(x,y) : \mathbb{R}^m \times \mathbb{R}^n \to \mathbb{R}\), \(D_x f\) is the row vector of the partial derivatives with respect to \(x_1, \ldots, x_m\). The corresponding column vector is \(\nabla_x f\).

- \(\nabla f_k = \nabla f(x_k)\)
- \(f_k = f(x_k)\)
- \(A \succ 0\) matrix \(A\) is positive definite
- \(|x|_A^2 = x^T A x\) weighted norm (\(A\) is matrix)
- \(\kappa(A) = |A||A^{-1}|_2\) Euclidean condition number (\(|A|_2\) matrix 2-norm)
- \(|\cdot|_{L^2(0, \xi)}\) \(L^2\)-norm
- \(1e - 2 = 0.01\)
B. Python Codes

B.1. Task of Finite Dimensional Optimal Control

File 'costfunctional':

```python
import numpy as np
import scipy as sc
import scipy.sparse
from scipy.sparse.linalg import aslinearoperator
import costfunctional_method_library as cml

class Data:
    
    ""
    Object to save data from the above finite dimensional optimal control problem. To initialise, it needs:
    
    n: discretisation points of y on the interval [0, T]
    m: discretisation points of u
    T: upper bound of the interval [0, T]
    yT, alpha, lamb: parameters for the cost functional
    a, d: parameters for A*y + F(y) = B*u
    y0_nw: starting point for Newton iterations
    y0_cg: starting point for CG iterations
    eps_nw: tolerance for Newton
    
    After initialising you have in addition to the above parameters:
    
    delta_t: distance between discretisation points
    A: negative finite dimensional Laplacian with parameters
    a1, . . . , an added to the diagonal
    A_1: A as linear operator
    B: (now identity matrix; can easily be changed in file
    costfunctional_method_library)
    B_1: B as linear operator
    B_T: B transposed and as linear operator
    Fu: Function, F:R^n -> R^n; y|-->(d1y1**3, . . . , dynyn**3)
    """
```
B.1: Task of Finite Dimensional Optimal Control

dFu: derivative of F

\( \Phi: \text{Function, } \Phi: (R^n \times R^m) \rightarrow R^n; \)
\( (y, u) \rightarrow A*y + F(y) - B*u \)

\( \partial \Phi: \text{partial derivative of } \Phi \text{ with respect to } y \)

```python
def __init__(self, n, m, T, yT, alpha, lamb, a, d, y0_nw, y0_cg, eps_nw):
    # define parameters
    self.n = n
    self.m = m
    self.T = T
    self.yT = yT
    self.alpha = alpha
    self.lamb = lamb
    self.a = a
    self.d = d
    self.y0_nw = y0_nw
    self.y0_cg = y0_cg
    self.eps_nw = eps_nw

    # define distance between discretisation points
    self.delta_t = ((n+2)**(-1)) * T

    # define matrices as linear operators
    if np.all(self.a) == 0:
        self.A = (-self.delta_t**(-2)) * cml.FDLaplacian(n)
    else:
        self.A = ((-self.delta_t**(-2)) * cml.FDLaplacian(n))
        + cml.diagonal(a)
    self.A_1 = aslinearoperator(self.A)
    self.B = cml.B(n,n)
    self.B_1 = aslinearoperator(self.B)
    self.B_T = aslinearoperator(sc.transpose(cml.B(n,n)))

    # define function F
    self.Fu = lambda y: d*(y**3)
    self.dFu = lambda y: cml.dF(y, d, self.n)
```
B.1: Task of Finite Dimensional Optimal Control

```python
# define constraint
if np.all(self.d) == 0:
    self.Phi = lambda y, u: self.A_1 * y - self.B_1 * u
    self.dyPhi = lambda y: self.A
else:
    self.Phi = lambda y, u: self.A_1 * y + self.Fu(y) - self.B_1 * u
    self.dyPhi = lambda y: self.A + self.dFu(y)

def CFandRG(u, data, ind = 0):
    ""
    Computes the cost functional and the reduced gradient
    ""
    Parameter
    ______
    u
    Argument (control)
    data
    Object which has the whole data of the system
    ind
    Indicator specifying output:
    0: calculate the solution of the state and the value of the cost functional
    1: calculate the solution of the state, the adjoint equation and the reduced gradient
    2: calculate the solution of the state, the adjoint equation, the value of the cost functional and reduced gradient
    Return
    _____
    dict
    Dictionary containing (dependent on ind!):
    [ 'func' ]: function value
    [ 'state' ]: associated state vector
    [ 'adjoint' ]: solution to the adjoint equation
    [ 'rgrad' ]: reduced gradient
    ""
    y, count = NewtonSolver_cg_own(data.Phi, data.dyPhi, \
    data.y0_nw, u, data.eps_nw, 100)
```
if ind == 0:
    cf = ((data.alpha/2) * (np.linalg.norm(y - data.yT))**2
         + (data.lamb/2)*(np.linalg.norm(u))**2)
    return {'func': cf, 'state': y}
if ind >= 1:
    c = data.alpha*(y - data.yT)
    p, info = cg(sc.transpose(data.dyPhi(y)), c, data.y0_cg)
    rg = data.B_T * p + data.lamb * u
    if ind == 1:
        return {'rgrad': rg, 'adjoint': p}
    if ind == 2:
        cf = ((data.alpha/2) * (np.linalg.norm(y - data.yT))**2
             + (data.lamb/2)*(np.linalg.norm(u))**2)
        return {'func': cf, 'state': y, 'adjoint': p, 'rgrad': rg}
return print('Error: wrong input for "ind"')

# = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
# Newton Solver using CG-method
# = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

def NewtonSolver_cg_own(fhandle, dfhandle, initialguess, u, eps, maxit):
    
    """ Parameter
    __________
    fhandle
        function handle that takes a ndarray as argument and gives
        output a ndarray with same dimension as the argument
dfhandle
        function handle that takes a ndarray as argument and ouputs
        the derivative of fhandle in form of sparse matrix or ndarray
    initialguess
        starting point
eps
        tolerance for the termination condition
maxit
        maximum number of iterations performed in the loop below
    Return
    ______
    x
        solution
count
    # number of iterations
###
# set parameters for the loop
x = initialguess
count = 0
# Termination Condition based on the Gradient at the current iterate and maximum number of iterations
while np.linalg.norm(fhandle(x,u)) >= eps and count < maxit:
    f = -fhandle(x,u)
    # calculate the derivative as sparse Matrix
    A = dfhandle(x)
    # use the CG-Method to solve the Newton Equation
    d, info = cg(A, f, x)
    # set next iterate
    x = x + d
    # count the number of iterations for the termination condition
    count += 1
return x, count

# = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
# Conjugate Gradient Method
# = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
def cg(A, b, x0, eps_rel = 1e-16, eps_abs = 1e-16, maxiter = 10**5):
    ###
    Parameter
    __________
    A
    Ndarray/Sparse which represent A in Ax = b
    b
    Ndarray which represent b in Ax = b
    x0
    the initial vector where the method should start
    eps_rel
        (optional) relative tolerance for termination condition
    eps_abs
        (optional) absolute tolerance for termination condition
    maxiter
        (optional) maximum number of iterations
    Return
x

solution x
counter

number of iterations

# check if A is sparse or dense Matrix
if sc.sparse.issparse(A) == True:
    A_1 = aslinearoperator(A)
    def Ax(x):
        Ax = A_1 * x
        return Ax
elif type(A) == np.ndarray:
    def Ax(x):
        Ax = A.dot(x)
        return Ax
else:
    # test if the input is a function which gives the action of Ax
    test = A(b)
    if len(test) == len(b):
        def Ax(x):
            Ax = A(x)
            return Ax
    else:
        print('wrong input for A')
        return

# set parameters for the loop
grad_1 = Ax(x0) - b
grad_0 = grad_1
normGrad_0 = np.linalg.norm(grad_0)
x = x0
d = -grad_1
counter = 0

# Check if the initial guess already is an approximate solution
if np.linalg.norm(grad_0) < eps_abs:
    print('Initial guess already satisfies termination condition. The normed gradient evaluatet at the initial guess is:')
    np.linalg.norm(grad_0))
else:
B.1: Task of Finite Dimensional Optimal Control

```python
# termination condition maximum number of iterations
while counter < maxiter:
    # calculate the matrix vector product A*d and the norm
    # of the Gradient at the current iterate;
    # both used twice afterwards
    p = Ax(d)
    nomGrad_1 = np.linalg.norm(grad_1)
    # calculate stepsize
    t = np.max(0, (nomGrad_1**2)/np.dot((d.T),p))
    # set next iterate
    x = x + t*d
    # set next gradient
    grad_2 = grad_1 + t*p
    # termination condition based on relative and
    # absolute tolerance
    if np.linalg.norm(grad_2) >= eps_rel*normGrad_0 + eps_abs:
        # beta for finding the next A-conjugate direction
        beta = ((np.dot((grad_2 - grad_1).T,grad_2))*
            ((nomGrad_1**2))**(-1))
        # update the gradient
        grad_1 = grad_2
        # set next A-conjugate direction
        d = -grad_1 + beta*d
    else:
        # set counter +1
        counter +=1
        return x, counter
    # set counter +1
    counter += 1
return x, counter
```

File: 'costfunctional_method_library':

```python
import numpy as np
import scipy as sc
import scipy.sparse

# Creates Matrices used in 'costfunctional'
```
def FDLaplacian(n, sparse = 'csr'):
    #main diagonal
    D_M = -2*sc.sparse.eye(n, n, format = 'csc')
    #other Diagonals
    D_U = sc.sparse.eye(n, n, 1, format = 'csc')
    D_L = sc.sparse.eye(n, n, -1, format = 'csc')
    A = D_M + D_U + D_L
    #transfer into the chosen sparse
    if sparse == 'coo':
        A = A.tocoo()
    elif sparse == 'csc':
        A = A.tocsc()
    elif sparse == 'csr':
        A = A.tocsr()
    else:
        print('Wrong choice of sparse. Choose: "coo","csc" or "csr"')
    return A

def B(n, m, sparse = 'csr'):
    #    A = sc.sparse.eye(n, m, -1, format = 'csr')
    A = sc.sparse.eye(n, m, 0, format = 'csr')
    #transfer into the chosen sparse
    if sparse == 'coo':
        A = A.tocoo()
    elif sparse == 'csc':
        A = A.tocsc()
    elif sparse == 'csr':
        A = A.tocsr()
    else:
        print('Wrong choice of sparse. Choose: "coo","csc" or "csr"')
    return A

def diagonal(a, sparse = 'csr'):
    n = len(a)
    diag = np.array([0])
    A = sc.sparse.spdiags(a.T, diag, n, n, format = 'csr')
    #transfer into the chosen sparse
    if sparse == 'coo':
        A = A.tocoo()
elif sparse == 'csc':
    A = A.tocsc()
elif sparse == 'csr':
    A = A.tocsr()
else:
    print('Wrong choice of sparse. Choose: "coo","csc" or "csr"')
return A

def dF(y, d, n, sparse = 'csr'):
    df_1 = 3*(d*(y**2))
    diag = np.array([0])
    A = sc.sparse.spdiags(df_1.T, diag, n, n, format = 'csr')
    
    # transfer into the chosen sparse
    if sparse == 'coo':
        A = A.tocoo()
    elif sparse == 'csc':
        A = A.tocsc()
    elif sparse == 'csr':
        A = A.tocsr()
    else:
        print('Wrong choice of sparse. Choose: "coo","csc" or "csr"')
    return A

B.2. Nonlinear Conjugate Gradient Method

File 'NLCG':

```python
import numpy as np
from oppy_unconOpt_wolfe_copy1 import wolfe as wolfe_oc
from linesearch import line_search_wolfe2
from oppy_unconOpt_wolfe import wolfe

def nlcg(fhandle, x0, dfhandle, data = None, ind = 0, \
    method = 'FR_PR', maxiter = 10e3, eps = 1e-3, \
```
B.2: Nonlinear Conjugate Gradient Method

\[
\text{rp1} = 23, \text{rp2} = 1e^{-4}, \text{disp} = \text{False}, \text{fargs} = ():\n\]

```python
### Parameter

- **fhandle**
  - function handle or
  - function in optimal control setting where it can be
    indicated if only the function or function and
    reduced gradient is computed
  - (needs a data-object as input)

- **x0**
  - initial point

- **dfhandle**
  - function handle computing the derivative of fhandle or None if
    fhandle is a function in optimal control setting

- **data** (optional)
  - object containing the whole information about
    optimal control problem

- **ind** (optional; default value: 0)
  - Indicator specifying wolfe line search:
    - 0: oppy library wolfe linesearch
    - 1: scipy wolfe line search and sometimes oppy library

- **method** (optional; default: 'FR_PR')
  - Choose a nonlinear conjugate gradient method:
    - 'FR': Fletcher Reeves
    - 'FRR': Fletcher Reeves with restart
    - 'PR': Polak Ribiere
    - 'PR+': See Bachelor Thesis, section 'Alternative Nonlinear–CG Methods'
    - 'FR_PR': Fletcher Reeves Polak Ribiere
    - 'DY': See Bachelor Thesis, section 'Alternative Nonlinear–CG Methods'
    - 'N': See Bachelor Thesis, section 'Alternative Nonlinear–CG Methods'

- **maxit** (optional; default: 10e3)
  - maximum number of iterations performed in the loop below

- **eps** (optional; default: 1e-3)
  - termination condition for loop
B.2: Nonlinear Conjugate Gradient Method

rp1 (optional; default: 23)
restart parameter used in some formulas to compute beta

rp2 (optional; default: 1e−4)
restart parameter used in some formulas to compute beta

disp (optional; default: False)
display information about the process in work

fargs (optional)
contains additional arguments for fhandle and dfhandle

Return

x
solution
k
number of iterations

if dfhandle == None and data == None:
   print('Error: at least one of the input arguments "dfhandle" \n      or "data" needs to be not "None"!'

#formulas to update beta

#Fletcher–Reeves–Polak–Ribiere
if method == 'FR_PR':
   def bta(gk, gk1, norm_gk, norm_gk1, p, tol, rp, 
      current_iterate):
      betaFR = norm_gk1**2 / norm_gk**2
      betaPR = np.dot(gk1.T, (gk1 - gk)) / norm_gk**2
      if betaPR < −betaFR :
         betaFRPR = −betaFR
      elif abs(betaPR) <= betaFR and abs(betaPR) > 0.0001:
         betaFRPR = betaPR
      elif abs(betaPR) <= betaFR and abs(betaPR) <= 0.0001:
         betaFRPR = 0
      elif betaPR > betaFR:
         betaFRPR = betaFR
      else:
         betaFRPR = 0
   return betaFRPR

#Fletcher–Reeves

elif method == 'FR':
    def bta(gk, gk1, norm_gk, norm_gk1, p, tol, rp, current_iterate):
        betaFR = norm_gk1 ** 2 / norm_gk ** 2
        return betaFR
#Fletcher−Reeves with restart
elif method == 'FRR':
    def bta(gk, gk1, norm_gk, norm_gk1, p, tol, rp, current_iterate):
        if type(k/rp) == int:
            beta = 0
        else:
            betaFR = norm_gk1 ** 2 / norm_gk ** 2
            if abs(betaFR - 1) < 0.0001:
                beta = 0
            else:
                beta = betaFR
        return beta
#Polak−Ribiere
elif method == 'PR':
    def bta(gk, gk1, norm_gk, norm_gk1, p, tol, rp, current_iterate):
        betaPR = np.dot(gk1.T, (gk1 - gk)) / norm_gk ** 2
        return betaPR
#modified Polak−Ribiere
elif method == 'PR+':
    def bta(gk, gk1, norm_gk, norm_gk1, p, tol, rp, current_iterate):
        betaPR = np.dot(gk1.T, (gk1 - gk)) / norm_gk ** 2
        betaPR = max(0, betaPR)
        return betaPR
#Dai−Yuan
elif method == 'DY':
    def bta(gk, gk1, norm_gk, norm_gk1, p, tol, rp, current_iterate):
        yhat = gk1 - gk
        betaDY = norm_gk1 ** 2 / np.dot(yhat.T, p)
        return betaDY
#Hager−Zhang
elif method == 'N':
    def bta(gk, gk1, norm_gk, norm_gk1, p, tol, rp, \
             current_iterate):
        yhat = gk1 - gk
        betaN = np.dot((yhat - 2*p*((np.linalg.norm(yhat)**2) /
                          (np.dot(yhat.T, p))))).T, (gk1 / np.dot(yhat.T, p)))
        return betaN
else:
    print('No valid input for "method" used. \n          Please use one of the available options')

def f(x):
    if dfhandle == None:
        z = fhandle(x, data, 0)['func']
    else:
        z = fhandle(x, *fargs)
    return z

def df(x):
    if dfhandle == None:
        l = fhandle(x, data, 1)['rgrad']
    else:
        l = dfhandle(x, *fargs)
    return l

#Set parameters to start the loop
x = x0
Gk = df(x)
p = -gk
k = 0
norm_gk = np.linalg.norm(gk)

#termination conditions for the algorithm
while norm_gk > eps and k < maxiter:
    #choose line search according to input 'ind'
    if ind == 0:
        #if optimal control setting use modified line search
        if dfhandle == None:
            #modified oppy line search
            t, flag = wolfe_oc(fhandle, x, p, data, None)
        else:
            ...
#original oppy line search
    t, flag = wolfe(f, df, x, p, None)

else:
    #slightly modified scipy wolfe line search
    t = line_search_wolfe2(f, df, x, p, gk)
    t = t[0]
    #if scipy line search does not converge use oppy wolfe
    if t == None:
        if dfhandle == None:
            t, flag = wolfe_oc(fhandle, x, p, data, None)
        else:
            t, flag = wolfe(f, df, x, p, None)

    #update current point
    x = x + t*p
    #compute updated gradient
    gk1 = df(x)
    #compute the euclidean norm of the gradient
    norm_gk1 = np.linalg.norm(gk1)
    #update beta
    beta = bta(gk, gk1, norm_gk, norm_gk1, p, rp2, rp1, k)
    #compute the next conjugate descent direction
    p = -gk1 + beta*p
    #memorise computed values for use in 'bta'
    gk = gk1
    norm_gk = norm_gk1
    #update counter
    k += 1
    #display information
    if disp:
        print('func:', f(x), 'NormGrad:', norm_gk, 
              'Iteration:', k, 'Beta:', beta, 'Stepsize:', t)

return x, k
C. Numerical Results

C.1. Discontinuous $y_s$

Parameter set: $n = 1000$, $m = n$, $X = 12$, $y_s(x) = \begin{cases} 
0 & x \in (0, 3) \\
12 & x \in [3, 6) \\
-12 & x \in [6, 9) \\
0 & x \in [9, 12) 
\end{cases}$, $\alpha = 99$, $\lambda = 0.01$, $a = 5$, $d = 0.5$, $y_{0\_nw} = 0$, $y_{0\_cg} = 0$, $\text{eps\_nw} = 1e - 10$, $\text{eps\_ncg} = 1e - 4$ and the initial guess for the nonlinear CG is $u = 0$.

The value of the cost functional and the norm of the gradient at the solution were computed in 107 iterations.

$$\hat{J}(\bar{u}) \approx 731689.2056 \quad \text{and} \quad \|\nabla \hat{J}(\bar{u})\| \approx 6.6693e - 05 \quad \text{(C.1)}$$

![Figure C.1.: Discontinuous $y_s$; nonlinear constraint; nonlinear CG.](image)
C.2: Continuous $y_s$, 1

Parameter set: $n = 1000$, $m = n$, $X = 12$, $y_s(x) = \begin{cases} x^3 - 18x & x \in (0, 6) \\ 144 - x^2 & x \in [6, 12) \end{cases}$, $\alpha = 99$, $\lambda = 0.01$, $a = 5$, $d = 0.5$, $y_{0\_nw} = 0$, $y_{0\_cg} = 0$, $\text{eps\_nw} = 1e - 10$, $\text{eps\_ncg} = 1e - 4$ and the is initial guess for the nonlinear CG is $u = 0$.

The value of the cost functional and the norm of the gradient at the solution were computed in 20 iterations.

$\hat{J}(\bar{u}) \approx 98559456.4006 \quad \text{and} \quad ||\nabla \hat{J}(\bar{u})|| \approx 8.8166e - 05 \quad (C.2)$

(a) Slope of associated state $y$ to optimal control $u$ and target slope $y_s$.

(b) Slope of optimal control $u$.

Figure C.3.: Continuous $y_s$, 1; nonlinear constraint; nonlinear CG.
C.3: Continuous $y_s$, 2

Parameter set: $n = 500$, $m = n$, $X = 12$, $y_s(x) = (x^3 - 12x^2)/100$, $\alpha = 99$, $\lambda = 1$, $a = 5$, $d = 0.5$, $y_0_{nw} = 0$, $y_0_{cg} = 0$, $eps_{nw} = 1e - 10$, $eps_{ncg} = 1e - 4$ and the is initial guess for the nonlinear CG is $u = 0$.

The value of the cost functional and the norm of the gradient at the solution were computed in 12 iterations.

$$\hat{J}(\bar{u}) \approx 19809.1469 \quad \text{and} \quad ||\nabla \hat{J}(\bar{u})|| \approx 5.3224e - 05 \quad (C.3)$$

(a) Slope of associated state $y$ to optimal control $u$ and target slope $y_s$.

(b) Slope of optimal control $u$.

Figure C.5.: Continuous $y_s$, 2; nonlinear constraint; nonlinear CG.
Figure C.6.: Continuous $y_s$, 2; nonlinear constraint; behavior of $||\nabla \hat{J}(u_k)||$, $\hat{J}(u_k)$; nonlinear CG.

Figure C.7.: Continuous $y_s$, 3; nonlinear constraint; nonlinear CG.

**C.4. Continuous $y_s$, 3**

Parameter set: $n = 500$, $m = n$, $X = 12$, $y_s(x) = (x^3 - 12x^2)/100$, $\alpha = 99.9$, $\lambda = 0.01$, $a = 5$, $d = 0.5$, $y_0\_nw = 0$, $y_0\_cg = 0$, $eps\_nw = 1e - 10$, $eps\_ncg = 1e - 4$ and the initial guess for the nonlinear CG is $u = 0$.

The value of the cost functional and the norm of the gradient at the solution were computed in 63 iterations.

$$\hat{J}(\bar{u}) \approx 402.5197 \quad \text{and} \quad ||\nabla \hat{J}(\bar{u})|| \approx 8.2769e - 05 \quad (C.4)$$
C.5. Continuous $y_s$, 4

Parameter set: $n = 500$, $m = n$, $X = 12$, $y_s(x) = (x^3 - 12x^2)/100$, $\alpha = 99.99999$, $\lambda = 0.00001$, $a = 5$, $d = 0.5$, $y_{0\_nw} = 0$, $y_{0\_cg} = 0$, $eps\_nw = 1e-10$, $eps\_ncg = 1e-4$ and the is initial guess for the nonlinear CG is $u = 0$.

The value of the cost functional and the norm of the gradient at the solution were computed in 145 iterations.

\[
\hat{J}(\bar{u}) \approx 0.4094 \quad \text{and} \quad ||\nabla \hat{J}(\bar{u})|| \approx 9.3062e-05 \quad \text{(C.5)}
\]

Figure C.8.: Continuous $y_s$, 3; nonlinear constraint; behavior of $||\nabla \hat{J}(u_k)||$, $\hat{J}(u_k)$; nonlinear CG.

Figure C.9.: Continuous $y_s$, 4; nonlinear constraint; nonlinear CG.
C.5: Continuous $y_s$, 4

(a) Behavior of $||\nabla \hat{J}(u_k)||$ at every iteration.

(b) Behavior of $\hat{J}(u_k)$ at every iteration.

Figure C.10.: Continuous $y_s$, 4; nonlinear constraint; behavior of $||\nabla \hat{J}(u_k)||$, $\hat{J}(u_k)$; nonlinear CG.