Available online at www.sciencedirect.com

ScienceDirect

IFAC PapersOnLine 51-20 (2018) 331–336

Sensitivity Analysis for Nonlinear Programming in CasADi ⋆

Joel A. E. Andersson James B. Rawlings

Department of Chemical and Biological Engineering,
University of Wisconsin-Madison, USA
(email: jandersson@wisc.edu, james.rawlings@wisc.edu)

Abstract: We present an extension of the CasADi numerical optimization framework that allows arbitrary order NLP sensitivities to be calculated automatically and efficiently. The approach, which can be used together with any NLP solver available in CasADi, is based on a sparse QR factorization and an implementation of a primal-dual active set method. The whole toolchain is freely available as open-source software and allows generation of thread-safe, self-contained C code with small memory footprint. We illustrate the toolchain using three examples; a sparse QP, an optimal control problem and a parameter estimation problem.

Keywords: NLP, sensitivity, active set methods, code generation, algorithmic differentiation

1. INTRODUCTION

Consider the parametric nonlinear program (NLP):

$$\begin{aligned} &\underset{x \in \mathbb{R}^{n_x}}{\text{minimize}} \quad f(x,p) \\ &\text{subject to } x^{\text{lb}} \leq x \leq x^{\text{ub}}, \quad g(x,p) = 0, \end{aligned} \tag{1}$$

where $\mathbb{R}^{n_x}$ is the decision variable and $p \in \mathbb{R}^{n_p}$ is given.

When the objective function $f(x,p)$ and constraint function $g(x,p)$ are sufficiently smooth and the problem is well-posed (cf. Fiacco and Ishizuka, 1990), sensitivity analysis can be used to study how the optimal x depends on small perturbations in p. In particular, we wish to perform a *forward sensitivity analysis*, which for (1) amounts to calculating the Jacobian-times-vector product:

$$\hat{x} = \frac{\partial x(p)}{\partial p}\, \hat{p}, \tag{2}$$

for a seed vector $\hat{p} \in \mathbb{R}^{n_p}$.

We also consider *adjoint sensitivity analysis*, which amounts to calculating the vector-times-Jacobian product:

$$\bar{p} = \left[\frac{\partial x(p)}{\partial p}\right]^{\text{T}} \bar{x}, \tag{3}$$

for a seed vector $\bar{x} \in \mathbb{R}^{n_x}$.

As we shall see in Section 2, both forward and adjoint sensitivity analysis can be efficiently and accurately calculated by applying the implicit function theorem (IFT) to the Karush Kuhn Tucker (KKT) conditions of the NLP.

NLP sensitivity analysis is of great practical interest for optimization practitioners. Applications include the calculation of confidence intervals and covariance matrices for parameter estimation problems as well as *advance step* strategies to nonlinear model predictive control (NMPC).

This paper presents a practical approach to NLP sensitivity analysis, implemented within the CasADi numerical optimization framework, where sensitivity equations of arbitrary order are automatically generated and solved using the symbolic representation of the NLP already available within CasADi. Unlike the barrier sensitivity approach, implemented in the code sIPOPT (Pirnay et al., 2012), the implemented approach makes no attempt to reuse the factorization used to solve the nonlinear program. Instead, it starts with the primal-dual solution returned from any NLP solver available in CasADi, determines the active set and then proceeds to evaluate the sensitivity equations.

The decoupling of the NLP solution and the sensitivity calculation makes it possible to solve the NLP with a quasi-Newton method, e.g. Generalized Gauss-Newton, but still use exact second derivatives to calculate sensitivities.

The entire toolchain presented here, including the sparse direct linear solver and the active set QP method has been implemented to allow execution in self-contained, generated C code. The generated C code performs all calculations with user-provided work vectors, which means no dynamic memory allocation, thread-safe execution and limited static memory allocation. The small memory footprint makes the code suitable for embedded applications. Furthermore, since active set changes do not affect the control flow, cache misses are limited.

1.1 CasADi

CasADi (Andersson et al., 2018) is an open-source modeling environment for numerical optimization, available for C++, Python, MATLAB and Octave. In particular, CasADi offers a flexible approach to solve numerical optimal control problems. CasADi lets the user to construct structurally complex symbolic expressions, which can include systems of nonlinear equation or initial-value problems in ordinary or differential-algebraic equations. New expressions for derivatives can then be calculated accurately and efficiently using a state-of-the-art implementation of algorithmic differentiation (AD).

⋆ The authors would like to thank Johnson Controls International Inc, for generous support that has made this work possible.

2405-8963 © 2018, IFAC (International Federation of Automatic Control) Hosting by Elsevier Ltd. All rights reserved.
Peer review under responsibility of International Federation of Automatic Control.
10.1016/j.ifacol.2018.11.055

1.2 Notation

Operations such as $x \leq y$ or $\min(x, y)$, are understood to be elementwise unless otherwise stated. Vectors are column vectors and we use MATLAB notation to denote concatenation: $[x; y; z] = [x^{\mathrm{T}}, y^{\mathrm{T}}, z^{\mathrm{T}}]^{\mathrm{T}}$. Elementwise multiplication is denoted $a * b$ and scalar product is $a \cdot b = a^{\mathrm{T}}\, b$.

2. NLP SENSITIVITY EQUATIONS

NLP solvers in CasADi use the following extension of (1), which includes bounds on $g(x, p)$ and an artificial variable $\tilde{p}$, equal to p:

$$\begin{array}{ll} \underset{x \in \mathbb{R}^{n_x},\, \tilde{p} \in \mathbb{R}^{n_p}}{\text{minimize}} & f(x, \tilde{p}) \\ \text{subject to} & \tilde{p} = p, \quad x^{\mathrm{lb}} \leq x \leq x^{\mathrm{ub}}, \quad g^{\mathrm{lb}} \leq g(x, \tilde{p}) \leq g^{\mathrm{ub}}. \end{array} \tag{4}$$

We introduce Lagrange multipliers λ_x^{lb}, λ_x^{ub}, λ_g^{lb}, λ_g^{ub} and λ_p. The Lagrangian function is defined as follows:

$$\begin{aligned} \tilde{\mathcal{L}}(x, \tilde{p}, \lambda_x^{\mathrm{lb}}, \lambda_x^{\mathrm{ub}}, \lambda_g^{\mathrm{lb}}, \lambda_g^{\mathrm{ub}}, \lambda_p) = f(x, \tilde{p}) + \lambda_p \cdot (\tilde{p} - p) \\ -\lambda_x^{\mathrm{lb}} \cdot (x - x^{\mathrm{lb}}) - \lambda_x^{\mathrm{ub}} \cdot (x^{\mathrm{ub}} - x) \\ -\lambda_g^{\mathrm{lb}} \cdot (g(x, p) - g^{\mathrm{lb}}) - \lambda_g^{\mathrm{ub}} \cdot (g^{\mathrm{ub}} - g(x, \tilde{p})). \end{aligned} \tag{5}$$

The KKT conditions for (4) are:

$$\nabla_x \tilde{\mathcal{L}}(x, \tilde{p}, \lambda_x^{\mathrm{lb}}, \lambda_x^{\mathrm{ub}}, \lambda_g^{\mathrm{lb}}, \lambda_g^{\mathrm{ub}}, \lambda_p) = 0, \tag{6a}$$

$$\nabla_p \tilde{\mathcal{L}}(x, \tilde{p}, \lambda_x^{\mathrm{lb}}, \lambda_x^{\mathrm{ub}}, \lambda_g^{\mathrm{lb}}, \lambda_g^{\mathrm{ub}}, \lambda_p) = 0, \tag{6b}$$

$$\tilde{p} = p, \quad x - x^{\mathrm{lb}} \geq 0, \quad x^{\mathrm{ub}} - x \geq 0, \tag{6c}$$

$$g(x, \tilde{p}) - g^{\mathrm{lb}} \geq 0, \quad g^{\mathrm{ub}} - g(x, \tilde{p}) \geq 0, \tag{6d}$$

$$\lambda_x^{\mathrm{lb}} \geq 0, \quad \lambda_x^{\mathrm{ub}} \geq 0, \quad \lambda_g^{\mathrm{lb}} \geq 0, \quad \lambda_g^{\mathrm{ub}} \geq 0, \tag{6e}$$

$$\lambda_x^{\mathrm{lb}} * (x - x^{\mathrm{lb}}) = 0, \quad \lambda_x^{\mathrm{ub}} * (x^{\mathrm{ub}} - x) = 0, \tag{6f}$$

$$\lambda_g^{\mathrm{lb}} * (g(x, \tilde{p}) - x^{\mathrm{lb}}) = 0, \quad \lambda_g^{\mathrm{ub}} * (g^{\mathrm{ub}} - g(x, \tilde{p})) = 0, \tag{6g}$$

where we assume strict complementary in (6f) and (6g).

To simplify the equations, let us introduce $\lambda_x := \lambda_x^{\mathrm{ub}} - \lambda_x^{\mathrm{lb}}$ and $\lambda_g := \lambda_g^{\mathrm{ub}} - \lambda_g^{\mathrm{lb}}$. We redefine the Lagrangian function with $\tilde{p}$ and all linear terms eliminated:

$$\mathcal{L}(x, p, \lambda_g) = f(x, p) + \lambda_g \cdot g(x, p). \tag{7}$$

This leads to the following, equivalent KKT formulation:

$$\nabla_x \mathcal{L}(x, p, \lambda_g) + \lambda_x = 0, \tag{8a}$$

$$\nabla_p \mathcal{L}(x, p, \lambda_g) + \lambda_p = 0, \tag{8b}$$

$$x^{\mathrm{lb}} \leq x \leq x^{\mathrm{ub}}, \quad g^{\mathrm{lb}} \leq g(x, p) \leq g^{\mathrm{ub}}, \tag{8c}$$

$$I_x^0\, \lambda_x + I_x^{\mathrm{ub}}\, x^{\mathrm{ub}} + I_x^{\mathrm{lb}}\, x^{\mathrm{lb}} - I_x\, x = 0, \tag{8d}$$

$$I_g^0\, \lambda_g + I_g^{\mathrm{ub}}\, g^{\mathrm{ub}} + I_g^{\mathrm{lb}}\, g^{\mathrm{lb}} - I_g\, g(x, p) = 0, \tag{8e}$$

where $I_{\bullet}^{\mathrm{lb}}$ and $I_{\bullet}^{\mathrm{ub}}$ are diagonal matrices with zeros and ones denoting inactive and active constraints, respectively. Furthermore, $I_{\bullet} := I_{\bullet}^{\mathrm{lb}} + I_{\bullet}^{\mathrm{ub}}$ and $I_{\bullet}^0 := I - I_{\bullet}$ indicate where either and neither bound is active, respectively.

Let us use (8a) to eliminate λ_x and write (8d) and (8e) as an implicit function $F(z, q) \Leftrightarrow z = G(q)$ with unknown $z := [x; \lambda_g]$ and the parameter $q := [x^{\mathrm{lb}}; x^{\mathrm{ub}}; g^{\mathrm{lb}}; g^{\mathrm{ub}}; p]$:

$$F(z, q) = \begin{bmatrix} I_x^0\, \nabla_x \mathcal{L}(x, p, \lambda_g) + I_x\, x - I_x^{\mathrm{ub}}\, x^{\mathrm{ub}} - I_x^{\mathrm{lb}}\, x^{\mathrm{lb}} \\ I_g\, g(x, p) - I_g^0\, \lambda_g - I_g^{\mathrm{ub}}\, g^{\mathrm{ub}} - I_g^{\mathrm{lb}}\, g^{\mathrm{lb}} \end{bmatrix} = 0. \tag{9}$$

We also have $y := [\lambda_x; \lambda_p]$, given by (8a) and (8b).

When the active set is known, the derivatives of G, if they exist, are given by the implicit function theorem:

$$\frac{\partial G}{\partial q} = -\left[\frac{\partial F}{\partial z}\right]^{-1} \frac{\partial F}{\partial q} \tag{10}$$

with the Jacobian given by:

$$\frac{\partial F}{\partial z} = \begin{bmatrix} I_x^0\, \nabla_x^2 \mathcal{L}(x, p, \lambda_g) + I_x & I_x^0 \left[\dfrac{\partial g}{\partial x}(x, p)\right]^{\mathrm{T}} \\ I_g\, \dfrac{\partial g}{\partial x}(x, p) & -I_g^0 \end{bmatrix}. \tag{11}$$

We also have

$$\frac{\partial F}{\partial q} = \begin{bmatrix} -I_x^{\mathrm{lb}} & -I_x^{\mathrm{ub}} & 0 & 0 & I_x^0\, \dfrac{\partial}{\partial p} \nabla_x \mathcal{L}(x, p, \lambda_g) \\ 0 & 0 & -I_g^{\mathrm{lb}} & -I_g^{\mathrm{ub}} & I_g\, \dfrac{\partial}{\partial p} g(x, p) \end{bmatrix}. \tag{12}$$

The complete Jacobian defined in (10) may be large prohibitively expensive to calculate. We can however use this expression to derive Jacobian-times-vector products, that can be calculated at the expense of only a linear system solve.

2.1 Forward sensitivity analysis

To derive an expression for the forward sensitivities $\hat{z} = [\hat{x}; \hat{\lambda}_g]$, we conceptually multiply (10) from the right by $\hat{q} := [\hat{x}^{\mathrm{lb}}; \hat{x}^{\mathrm{ub}}; \hat{g}^{\mathrm{lb}}; \hat{g}^{\mathrm{ub}}; \hat{p}]$:

$$\begin{aligned} \hat{z} &= \frac{\partial G}{\partial q}\, \hat{q} = -\left[\frac{\partial F}{\partial z}\right]^{-1} \frac{\partial F}{\partial q}\, \hat{q} \\ &= \left[\frac{\partial F}{\partial z}\right]^{-1} \begin{bmatrix} I_x^{\mathrm{lb}}\, \hat{x}^{\mathrm{lb}} + I_x^{\mathrm{ub}}\, \hat{x}^{\mathrm{ub}} - I_x^0\, \dfrac{\partial}{\partial p} \nabla_x \mathcal{L}(x, p, \lambda_g)\, \hat{p} \\ I_g^{\mathrm{lb}}\, \hat{g}^{\mathrm{lb}} + I_g^{\mathrm{ub}}\, \hat{g}^{\mathrm{ub}} - I_g\, \dfrac{\partial}{\partial p} g(x, p)\, \hat{p} \end{bmatrix} \end{aligned} \tag{13}$$

Note that $\frac{\partial g}{\partial p}\, \hat{p}$ can be efficiently calculated using forward mode AD and $\frac{\partial \nabla_x \mathcal{L}}{\partial p}\, \hat{p}$ can be calculated using forward over reverse mode AD.

2.2 Adjoint sensitivity analysis

To derive an expression for the adjoint sensitivities, we conceptually multiply the transpose of (10) from the right with $\bar{z} := [\bar{x}; \bar{\lambda}_g]$. If we first solve the linear system

$$\begin{bmatrix} \bar{\beta}_x \\ \bar{\beta}_g \end{bmatrix} = \left[\frac{\partial F}{\partial z}\right]^{-\mathrm{T}} \bar{z}, \tag{14}$$

and calculate $\bar{\beta}_p$ using (reverse over) reverse mode AD:

$$\bar{\beta}_p = \left[\frac{\partial}{\partial p} \nabla_x \mathcal{L}(x, p, \lambda_g)\right]^{\mathrm{T}} I_x^0\, \bar{\beta}_x + \left[\frac{\partial}{\partial p} g(x, p)\right]^{\mathrm{T}} I_g\, \bar{\beta}_g, \tag{15}$$

we can obtain $\bar{q} = [\bar{x}^{\mathrm{lb}}; \bar{x}^{\mathrm{ub}}; \bar{g}^{\mathrm{lb}}; \bar{g}^{\mathrm{ub}}; \bar{p}]$ from

$$\bar{q} = \left[\frac{\partial G}{\partial q}\right]^{\mathrm{T}} \bar{z} = -\left[\frac{\partial F}{\partial q}\right]^{\mathrm{T}} \begin{bmatrix} \bar{\beta}_x \\ \bar{\beta}_g \end{bmatrix} = \begin{bmatrix} I_x^{\mathrm{lb}}\, \bar{\beta}_x \\ I_x^{\mathrm{ub}}\, \bar{\beta}_x \\ I_g^{\mathrm{lb}}\, \bar{\beta}_g \\ I_g^{\mathrm{ub}}\, \bar{\beta}_g \\ -\bar{\beta}_p \end{bmatrix}. \tag{16}$$

2.3 Implementation in CasADi

The sensitivity equations (13) and (16) have been implemented in CasADi 3.4 and are generated whenever the user attempts to differentiate an NLP solver instance within the framework. The sensitivity equations, including the linear system solves, have been implemented entirely using differentiable CasADi expressions. This fact means that the resulting expressions can, in turn, be efficiently differentiated providing a means to accurately calculate derivative information to *any* order.

In the following, we discuss two steps of critical importance that have been left out of the discussion so far, namely how to factorize and solve the sparse KKT system (Section 3) and how to determine the active set (Section 4).

3. A SPARSE DIRECT LINEAR SOLVER

As already mentioned, we do not reuse existing factorizations of the KKT system. This differs from sIPOPT (Pirnay et al., 2012), which reuses the Bunch-Kaufman LDL^{T} factorization in IPOPT (Wächter and Biegler, 2006). The decoupling of NLP solution and sensitivity calculation means that the NLP can be solved with quasi-Newton methods such as Generalized Gauss-Newton, without impacting the correctness of the calculated sensitivities. The NLP can also be solved in a higher-dimensional space, known to improve the rate of local convergence and region of attraction (Albersmeyer and Diehl, 2010).

CasADi supports embedding sparse linear systems into symbolic expressions, with automatic calculation of derivatives. We use this support and, for the purposes of this paper, extend it with a new sparse direct linear solver. The solver is based on a sparse, *left-looking* Householder-based QR factorization (cf. Davis, 2006). For a sparse matrix A, which we assume to be square for the purposes of this paper, the method calculates a factorization of the form:

$$A = {P_{\mathrm{R}}}^{\mathrm{T}} \, Q \, R \, P_{\mathrm{C}} \tag{17}$$

where P_{R} is a row permutation, P_{C} is a column permutation, Q is an orthogonal matrix and R is an upper triangular matrix. For reasons of efficiency, matrix Q is never formed explicitly, but represented by (V, β), as described in (Davis, 2006). Here, the columns of the sparse matrix V contain of the Householder vectors and β is a vector.

Our implementation divides the factorization into two steps; a *symbolic factorization*, implemented in CasADi's `Sparsity` class with routines derived from (Davis, 2006) and a *numeric factorization*, in CasADi's C/C++ runtime.

Given the sparsity pattern of A, the symbolic factorization calculates the permutation matrices P_{R} and P_{C} as well as the sparsity patterns of V and R. With sparsity pattern, we refer to the location of the structural nonzeros, which may or may not be numerically zero. CasADi represents sparsity patterns using compressed column storage (CCS) (cf. eg. Davis, 2006). To find P_{C}, we use the fill-reducing *approximate minimal degree* algorithm (Amestoy et al., 1996), applied to $A^{\mathrm{T}} A$. Our implementation is derived from `cs_amd`, listed in (Davis, 2006, Chapter 7). P_{R} as well as the sparsity patterns of V and R are found using a Householder-based left-looking algorithm derived from `cs_qr`, listed in (Davis, 2006, Chapter 5).

The numerical factorization is implemented in about 250 lines of templated C++ code, designed to allow automatic transformation to self-contained C code. This code includes the actual numerical factorization, i.e. the calculation of the numerical values for β, V and R, as well as solving the (optionally transposed) linear system and handling singularity. If a diagonal entry in R falls below some threshold, a vector v with $v^{\mathrm{T}} v = 1$ can be calculated such that $A \, v \approx 0$. This feature is used to recover from singular KKT matrices in the active set solver described in Section 4.

4. DETERMINING THE ACTIVE SET

The calculation of the parametric sensitivities, as outlined in Section 2, hinges on knowledge of the active set at the solution and invertibility of (11). Finding the active set amounts to solving a quadratic program (QP):

$$\begin{aligned} \underset{x \in \mathbb{R}^{n_x}}{\text{minimize}} \quad & \frac{1}{2} x^{\mathrm{T}} \, H \, x + c^{\mathrm{T}} \, x \\ \text{subject to} \quad & x^{\mathrm{lb}} \le x \le x^{\mathrm{ub}}, \quad g^{\mathrm{lb}} \le A \, x \le g^{\mathrm{ub}}, \end{aligned} \tag{18}$$

with KKT conditions (cf. Section 2):

$$H \, x + A^{\mathrm{T}} \, \lambda_g + \lambda_x = 0, \tag{19a}$$

$$x^{\mathrm{lb}} \le x \le x^{\mathrm{ub}}, \quad g^{\mathrm{lb}} \le A \, x \le g^{\mathrm{ub}}, \tag{19b}$$

$$I_x^0 \, \lambda_x + I_x^{\mathrm{ub}} \, x^{\mathrm{ub}} + I_x^{\mathrm{lb}} \, x^{\mathrm{lb}} - I_x \, x = 0, \tag{19c}$$

$$I_g^0 \, \lambda_g + I_g^{\mathrm{ub}} \, g^{\mathrm{ub}} + I_g^{\mathrm{lb}} \, g^{\mathrm{lb}} - I_g \, A \, x = 0. \tag{19d}$$

Problems of this form can in principle be solved with any of the QP solvers interfaced with CasADi, importantly including qpOASES (Ferreau et al., 2014). In the following, we take a different approach and implement an active set method, using the sparse linear solver presented in Section 3. This approach avoids introducing additional external dependencies and allow us to verify that the identified active set correspond to an invertible KKT matrix (11). It also gives us a complete toolchain that allows C code generation for embedded systems.

4.1 A primal-dual active set method

We choose a primal-dual active set algorithm based on the *parametric quadratic programming* approach first proposed in (Best, 1996) and later implemented in qpOASES (Ferreau et al., 2008, 2014). Using a primal-dual method makes sense since we invoke the method at the solution of an NLP method such as IPOPT, meaning that we should be close to both primal and dual feasibility.

We use the following approximation of the KKT conditions, with λ_x made explicit:

$$\underbrace{\begin{bmatrix} I_x^0 \, H + I_x & I_x^0 \, A^{\mathrm{T}} \\ I_g \, A & -I_g^0 \end{bmatrix}}_{:=K} \begin{bmatrix} x \\ \lambda_g \end{bmatrix} = \underbrace{\begin{bmatrix} I_x^{\mathrm{ub}} \, x^{\mathrm{ub}} + I_x^{\mathrm{lb}} \, x^{\mathrm{lb}} \\ I_g^{\mathrm{ub}} \, g^{\mathrm{ub}} + I_g^{\mathrm{lb}} \, g^{\mathrm{lb}} \end{bmatrix}}_{:=r} \tag{20a}$$

$$\begin{aligned} g = A \, x, \quad \lambda_x = {} & I_x^{\mathrm{ub}} \max(-H \, x - A^{\mathrm{T}} \, \lambda_g, \epsilon_{\mathrm{f}}) \\ & + I_x^{\mathrm{lb}} \min(-H \, x - A^{\mathrm{T}} \, \lambda_g, -\epsilon_{\mathrm{f}}) \end{aligned} \tag{20b}$$

$$x^{\mathrm{lb}} \le x \le x^{\mathrm{ub}}, \quad g^{\mathrm{lb}} \le g \le g^{\mathrm{ub}}, \quad H \, x + A^{\mathrm{T}} \, \lambda_g + \lambda_x = 0. \tag{20c}$$

In this formulation, (20a) describes a linear system of equations that can be used to find a feasible search direction.

The two dependent quantities g and λ_x are calculated in (20b). In the calculation of λ_x, we denote by ϵ_{f} the smallest number which in floating point arithmetics is greater than zero. In C++ and `double` precision, this number can be obtained using `std::numeric_limits<double>::min()` and is typically in the order 10^{-308}. Note that I_x^{lb} and I_x^{ub} can be safely be retrieved from λ_x by comparing the floating point values to zero. Finally, (20c) states primal and dual feasibility. We monitor the feasibility violations:

$$\begin{aligned} \epsilon_{\mathrm{p}} = \min(\epsilon): &\qquad x^{\mathrm{lb}} - \epsilon \le x \le x^{\mathrm{ub}} + \epsilon, \\ &\qquad g^{\mathrm{lb}} - \epsilon \le g \le g^{\mathrm{ub}} + \epsilon, \\ \epsilon_{\mathrm{d}} = \min(\epsilon): &\qquad -\epsilon \le H\,x + A^{\mathrm{T}}\,\lambda_g + \lambda_x \le \epsilon. \end{aligned} \tag{21}$$

The goal of the proposed algorithm is to minimize the primal-dual error $\epsilon_{\mathrm{pd}} = \max(C\,\epsilon_{\mathrm{p}}, \epsilon_{\mathrm{d}})$, for some given constant C. We choose $C = 1000$ in our numerical tests, reflecting the fact that dual feasibility, i.e. optimality, typically makes little sense without primal feasibility.

4.2 Calculating the initial search direction

For a given solution guess, we calculate a search direction:

$$\begin{bmatrix} \Delta x \\ \Delta\lambda_g(0) \end{bmatrix} = \begin{cases} \alpha\, v : K\,v = 0, \|v\| = 1, & \text{if } K \text{ is singular} \\ K^{-1}\,(r - K\,[x;\lambda_g]) & \text{otherwise} \end{cases} \tag{22}$$

When K is singular, the nullspace vector v, defined in Section 3, is chosen as the search direction. To find the scalar factor $\alpha \neq 0$, identify an index $\tilde{k}$ such that setting $[\lambda_x;\lambda_g]_{\tilde{k}} := \tilde{s}$ with $\tilde{s} \in \{-\epsilon_{\mathrm{f}}, 0, \epsilon_{\mathrm{f}}\}$ increases the rank of K. We choose α so that a full step would bring $[x;g]_{\tilde{k}}$ to the bound (if constraint $\tilde{k}$ is inactive) or bring $[\lambda_x;\lambda_g]_{\tilde{k}}$ to zero (if constraint $\tilde{k}$ is active). See also Section 4.7 below.

We also have a search direction for g and λ_x:

$$\begin{bmatrix} \Delta g \\ \Delta\lambda_x(0) \end{bmatrix} = \begin{bmatrix} A\,\Delta x \\ -I_\Delta\,(H\,\Delta x + A^{\mathrm{T}}\,\lambda_g(0)) \end{bmatrix}, \tag{23}$$

where $I_\Delta = I_\Delta(x, \lambda_g(0))$ can be derived from (20b).

4.3 Piecewise linear line-search

Given the initial search direction, we proceed to find the largest step $\tau_{\mathrm{p}} \le 1$ such that $x(\tau_{\mathrm{p}}) = x + \tau_{\mathrm{p}}\,\Delta x$ and $g(\tau_{\mathrm{p}}) = g + \tau_{\mathrm{p}}\,\Delta g$ violate bounds with at most $\max(\epsilon_{\mathrm{p}}, \epsilon_{\mathrm{d}}/C)$. If $\tau_{\mathrm{p}} < 1$, identify the *primal blocking constraint* and corresponding index k.

Next, we find the largest step $\tau \le \tau_{\mathrm{p}}$ such that the new dual infeasibility does not exceed $\max(C\,\epsilon_{\mathrm{p}}, \epsilon_{\mathrm{d}})$. Should a trial step result in a sign change for an inequality constraint multiplier, the value is capped at $\pm\epsilon_{\mathrm{f}}$ and the corresponding dual search direction becomes zero. This means that $\lambda_x(\tau)$ and $\lambda_g(\tau)$ are piecewise linear functions, which must be taken into account in the infeasibility calculation. If $\tau < \tau_p < 1$, discard any primal blocking constraint and identify the *dual blocking constraint*. Note that we permit sign changes for equality constraint multipliers.

4.4 Updating the active set

To update the active set, we try to identify an index k and corresponding new multiplier value $s \in \{-\epsilon_{\mathrm{f}}, 0, \epsilon_{\mathrm{f}}\}$ using the following approach:

- Use the primal blocking constraint, if any
- Otherwise, if K was singular, use index $\tilde{k}$ and sign $\tilde{s}$, as identified in Section 4.2
- If there is still no active set change, try to *enforce* a constraint if the inequality $C\,\epsilon_{\mathrm{p}} \ge \epsilon_{\mathrm{d}}$ holds, otherwise try to *drop* a constraint. To enforce a constraint, we use the index corresponding to the largest feasibility violation, if it is not already enforced. To drop a constraint, we use the constraint corresponding to the largest sensitivity in ϵ_{d}, which can be dropped without increasing maximum dual infeasibility.

If an index k was found, we perform the active set change. If K was nonsingular, we also check if an additional constraint needs to be flipped in order to avoid singularity. Such constraints can be found using a linear system solve with the (old) K. This second active set change is accepted unless it increases dual infeasibility beyond $\max(C\,\epsilon_{\mathrm{p}}, \epsilon_{\mathrm{d}})$.

We summarize the proposed algorithm in Algorithm 1.

Algorithm 1 Primal-dual active set method in `qrqp`

```
function QRQP($x, \lambda_x, \lambda_g$)                ▷ Primal-dual guess
    repeat
        Identify $I_x^{lb}$, $I_x^{ub}$, $I_g^{lb}$, $I_g^{ub}$ from $\lambda_x$ and $\lambda_x$
        Calculate $I_x$, $I_x^0$, $I_g$ and $I_g^0$
        Calculate $g$ and $\lambda_x$                          ▷ (20b)
        Calculate $\epsilon_p$, $\epsilon_d$ and $\epsilon_{pd}$                      ▷ (21)
        Calculate and factorize $K$, check singularity
        Calculate $\Delta x$ and $\Delta\lambda_g(0)$                   ▷ (22)
        Calculate $\Delta g$ and $\Delta\lambda_x(0)$                   ▷ (23)
        Find largest primal-dual step $\tau$       ▷ Section 4.3
        Update $x$, $g$, $\lambda_x$ and $\lambda_g$
        Update the active set                  ▷ Section 4.4
    until no more active set change
    return $x, g, \lambda_x, \lambda_g$                ▷ Primal-dual solution
end function
```

The most fundamental difference between the proposed algorithm and that proposed in (Best, 1996) and (Ferreau et al., 2008) is how the line-search parameter τ is chosen. Rather than terminating the line-search whenever a constraint becomes active ("primal blocking constraint") or whenever a constraint becomes inactive ("dual blocking constraint"), the line-search continues as long as the new primal feasibility violation remains below $\max(\epsilon_{\mathrm{p}}, \epsilon_{\mathrm{d}}/C)$ and dual feasibility violation remains below $\max(C\,\epsilon_{\mathrm{p}}, \epsilon_{\mathrm{d}})$. This relaxation allows the method to take more aggressive steps towards the solution.

Another key difference to other implementations is the use of a sparse QR factorization applied to a non-symmetric formulation of the KKT conditions. Importantly, this factorization allows singularity to be detected and resolved.

We detail some aspects of the algorithm in the following.

4.5 Implementation

The sparse active set QP solver presented in this section has been implemented in the code `qrqp`, which is distributed with CasADi starting with version 3.4.2. It has been implemented in about 1000 lines of templated C++ code, designed to allow automatic transformation to self-contained C code, with no dynamic memory allocation.

qrqp can be used as a stand-alone sparse QP solver, with (as of this writing, partial) support for code generation. All sparsity patterns and symbolic factorizations of linear systems are precalculated before code generation.

4.6 Purity, hot-starting

We have designed qrqp to be *pure* in the functional programming sense, meaning that the primal-dual solution depends solely on the initial guess, and more specifically, on x, λ_g and the *signs* of λ_x. The method has no state that can be used to speed up the solution when called repeatedly and the initial active set is the one provided as inputs, save for correction of illegal values (e.g. if $\lambda_{x,i} > 0$, but $x_i^{\text{ub}} = \infty$). Therefore, the method is naturally "hot-started" whenever it is initialized with a previous solution. Using work vectors avoids repeated memory allocation.

4.7 Handling of singularity

We detect singularity in the KKT conditions by using the sparse QR factorization described in Section 3. If any diagonal entry of the R factor falls below a threshold (10^{-12} in our tests), the matrix K is declared singular and the *first* diagonal entry below this threshold is used to generate a linear combination of the columns in K, i.e. $K\,v \approx 0$ with $\|v\| = 1$. Note that this vector is orthogonal to all existing rows in K. To increase the rank of K, we look for a constraint that be flipped, resulting in a new row of K. This new row must not be orthogonal to v.

A second requirement for improving rank can be found by recalculating the QR factorization for K^{T}, which has the same sparsity structure and can therefore use the same symbolic factorization. This second factorization gives us a linear combination of the rows in K, or more generally a set of linear combinations if the small R diagonal entry was not encountered in the lower-right corner of R. In other words, we find a set of vectors, $w_1, w_2, \ldots$ with $K^{\text{T}}\,w_i \approx 0$ and $\|w_i\| = 1$. If a constraint is not part of such a linear combination, flipping the constraint cannot increase rank.

These two conditions are used to identify constraints that can be changed from active to inactive, or vice versa, in order to increase rank. If there are multiple candidates, we choose the constraint corresponding to a direction of steepest descent for primal or dual infeasibility.

Restoring feasibility typically only happens in the beginning of the algorithm. Once K is non-singular, we can maintain non-singularity by checking if a prospective new row in K would result in singularity. In this situation, we can look for another constraint that can be changed as well in order to maintain singularity. Maintaining non-singularity is discussed in (Ferreau et al., 2014).

5. EXAMPLES

The components in this paper have been tested in different contexts. The linear solver presented in Section 3 is the default linear solver in CasADi since version 3.4 and as such has been thoroughly tested. The qrqp code in Section 4 is a newer addition with several aspects of the algorithm, in particular with regards to singularity handling, still in flux as of this writing. The solver has been demonstrated to work on a set of small QPs which mainly checks common pathologies, including semi-definiteness Hessian matrices and redundant constraints. More testing is needed, however, as discussed in Section 6.

Table 1. Number of active set changes for the chain QP example

M	8	40	200	1000
qrqp	2	9	39	182
qpOASES	15	82	416	2085

The calculation of NLP sensitivities is mainly being tested as part of the development of paresto, a tool for parameter estimation. This tool is written in MATLAB/Octave and relies on the support for NLP sensitivity calculations in CasADi presented in this paper.

5.1 A QP example

We model a chain as M point masses connected by $M-1$ springs, where the first and the last points are fixed. Each mass $i \in \{1, \ldots, M\}$ has position (y_i, z_i) and the chain is assumed to be in the equilibrium point of the system, which minimizes the potential energy:

$$E(x) = \sum_{i=1}^{M-1} \frac{D}{2} \left((y_i - y_{i+1})^2 + (z_i - z_{i+1})^2 \right) + \sum_{i=1}^{M} m\, g\, z_i,$$

where $x = [y_1, z_1, \ldots, y_M, z_M]$, $g = 9.81$, $m = 40/M$ and $D = 70\,M$.

The boundary constraints are $(y_1, z_1) = (-2, 1)$ and $(y_M, z_M) = (2, 1)$, respectively and the chain is subject to two linear inequality constraints, modeling the ground:

$$z_i \geq 0.5, \quad z_i - 0.1\, y_i \geq 0.5, \quad i = 1, \ldots, M.$$

The number of active set changes needed to find the problem, for different values of M, is shown in Table 1. We have compared the result with qpOASES, version 3.2 (Ferreau et al., 2014). qpOASES is run in "Schur complement mode", using MA27 (HSL, 2018).

While little can be inferred from a single example, initial results for qrqp suggest that the algorithm can be competitive with a more mature QP code such as qpOASES.

5.2 Sensitivity analysis for an optimal control problem

Consider the following optimal control problem (OCP):

$$\begin{array}{ll} \underset{x(\cdot),\, u(\cdot)}{\text{minimize}} & \displaystyle\int_0^{10} \left(x_1^2 + x_2^2 + u^2 \right) dt \\ \text{subject to} & x_1(0) = p_1, \quad x_2(0) = p_2, \\ & \left\{ \begin{array}{l} \dot{x}_1 = (1 - x_2^2)\, x_1 - x_2 + u, \\ \dot{x}_2 = x_1, \\ -1.0 \leq u \leq 0.85, \\ x_1 \geq -0.25, \end{array} \right. \quad t \in [0, 10] \end{array} \tag{24}$$

where $x(\cdot) \in \mathbb{R}^2$ is the state, $u(\cdot) \in \mathbb{R}$ is the free control and $p \in \mathbb{R}^2$ is a fixed parameter.

Problems like (24) can be solved with CasADi using several different methods, as described in (Andersson et al., 2018). Here, we will use a direct collocation parameterization with 50 uniformly distributed, piecewise constant control

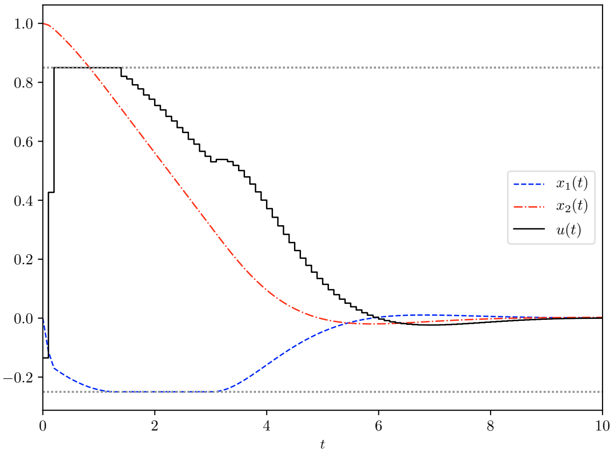

Fig. 1. Solution to the optimal control problem (24)

intervals. The differential equations are parameterized using third order Legendre polynomials. The result of the optimization, obtained using IPOPT (Wächter and Biegler, 2006), for $p_1 = 0, p_2 = 1$ is shown in Figure 1. Note that the both the upper bound on u and the lower bound on x_1 become active for parts of the solution trajectory.

We consider how the solution trajectory depends on perturbations in (p_1, p_2). We calculate perturbation using three different approaches; (1) using finite differences applied to the IPOPT solver routine, (2) using the forward sensitivity analysis as described in Section 2 and (3) using adjoint sensitivity analysis as described in Section 2. All three method gives comparable answers. Forward sensitivity analysis is about 1000 faster than finite differences, while not relying on an (error prone) perturbation parameter. Forward and adjoint sensitivity analysis have comparable computational cost, but calculate different things: Whereas forward sensitivity analysis can calculate how all outputs depend on a single input, say p_1, adjoint sensitivity analysis can calculate how all inputs (here p_1 and p_2) depend on a single output.

5.3 Calculation of covariances and confidence intervals

In the final example we consider the parameter estimation problem described in (Rawlings and Ekerdt, 2015, Section 9.3). This model describes a well-stirred semi-batch reactor with concentrations for four different components evolving in time. We have measurement values for the relative ratio between two of the components. The parameter estimation problems consists of estimating the initial amount of substance for the starting material, along with two rate constants. This problem was previously solved using a single-shooting discretization and a Gauss-Newton optimization approach, with finite-differences to approximate the reduced Hessian, which can be interpreted as the covariance matrix of the estimated parameters. Using the framework presented in this paper, and with a direct collocation implementation based on IPOPT, we are able to reproduce the results using analytical calculation of the derivatives. Confidence intervals with 95 % certainty for the estimated parameters were calculated using an *F probability function* as described in (Rawlings and Ekerdt, 2015, Chapter 9). The implementation was done using CasADi's interface to MATLAB/Octave and the code for this and other parameter estimation problems is available as `paresto` on Github (cf. `github.com/rawlings-group/paresto`).

6. DISCUSSION AND OUTLOOK

We have shown a complete, open-source toolchain that enables automatic forward and adjoint sensitivity analysis for NLPs. Initial tests show promising results.

Future work will include a more thorough theoretical analysis as well as analyzing the performance of `qrqp` with the help of benchmark collections. The analysis and benchmarking should help resolve some design questions, in particular regarding the logic for active set changes presented in Section 4.4.

An open question is how to best use `qrqp` for sequential quadratic programming (SQP). Initial tests show very promising results in using the code in an SQP framework without QP regularization. While we could see early termination in some SQP iterations due to QP non-convexity, the SQP algorithm still converged quickly to the solution.

REFERENCES

Albersmeyer, J. and Diehl, M. (2010). The Lifted Newton Method and its Application in Optimization. *SIAM Journal on Optimization*, 20(3), 1655–1684.

Amestoy, P.R., Davis, T.A., and Duff, I.S. (1996). An Approximate Minimum Degree Ordering Algorithm. *SIAM J. Matrix Anal. & Appl*, 17(4), 886–905.

Andersson, J.A.E., Gillis, J., Horn, G., Rawlings, J.B., and Diehl, M. (2018). CasADi: a software framework for nonlinear optimization and optimal control. *Math. Prog. Comp.* doi:10.1007/s12532-018-0139-4.

Best, M.J. (1996). *Applied Mathematics and Parallel Computing*, chapter An Algorithm for the Solution of the Parametric Quadratic Programming Problem, 57–76. Physica-Verlag HD.

Davis, T.A. (2006). *Direct Methods for Sparse Linear Systems*. SIAM.

Ferreau, H., Bock, H., and Diehl, M. (2008). An online active set strategy to overcome the limitations of explicit MPC. *International Journal of Robust and Nonlinear Control*, 18(8), 816–830.

Ferreau, H., Kirches, C., Potschka, A., Bock, H., and Diehl, M. (2014). qpOASES: A parametric active-set algorithm for quadratic programming. *Math. Prog. Comp.*, 6(4), 327–363.

Fiacco, A.V. and Ishizuka, Y. (1990). Sensitivity and stability analysis for nonlinear programming. *Annals of Operations Research*, 27, 215–235.

HSL (2018). A collection of Fortran codes for large scale scientific computation. `http://www.hsl.rl.ac.uk`.

Pirnay, H., López-Negrete, R., and Biegler, L.T. (2012). Optimal sensitivity based on IPOPT. *Math. Prog. Comp.*, 4(4), 307–331.

Rawlings, J.B. and Ekerdt, J.G. (2015). *Chemical Reactor Analysis and Design Fundamentals*. Nob Hill, 2^{nd} ed.

Wächter, A. and Biegler, L. (2006). On the Implementation of a Primal-Dual Interior Point Filter Line Search Algorithm for Large-Scale Nonlinear Programming. *Math. Program.*, 106(1), 25–57.