TensCalc – A toolbox to generate fast code to solve nonlinear constrained minimizations and compute Nash equilibria

Technical report

João P. Hespanha

June 2, 2017*

Abstract

We describe the toolbox TensCalc that generates specialized C-code to solve nonlinear constrained optimizations and to compute Nash equilibria. TensCalc is aimed at scenarios where one needs to solve very fast a large number of optimizations that are structurally similar. This is common in applications where the optimizations depend on measured data and one wants to compute optima for large or evolving datasets, e.g., in robust estimation and classification, maximum likelihood estimation, model predictive control (MPC), moving horizon estimation (MHE), and combined MPC-MHE (which requires the computation of a saddle-point equilibrium). TensCalc is mostly aimed at generating solvers for optimizations with up to a few thousands of optimization variables/constraints and solve times up to a few milliseconds. The speed achieved by the solver arises from a combination of features: reuse of intermediate computations across and within iterations of the solver, detection and exploitation of matrix sparsity, avoidance of run-time memory allocation and garbage collection, and reliance on flat code that improves the efficiency of the micro-processor pipelining and caching. All these features have been automated and embedded into the code generation process. We include a few representative examples to illustrate how the speed and memory footprint of the solver scales with the size of the problem.

1 Introduction

In the sciences and engineering, numerical optimizations are often used to determine parameter values based on measured or simulated data. This arises in the estimation of parameters using maximum likelihood or robust regression and classification. It also arises in the computation of optimal control signals using model predictive control (MPC), moving horizon estimation (MHE), or combined MPC-MHE. In these applications, it is common to solve many instances of a particular optimization for different data sets and very significant time savings are possible by building solvers that has been optimized for the specific optimization.

The TensCalc toolbox generates C code to solve nonlinear constrained minimizations and to compute Nash equilibria. The main goal of the toolbox is to take an intuitive description of the optimization problem expressed in a MATLAB®-like syntax and completely automate the process of generating C code capable of solving the optimization very fast for different data sets. To achieve this goal, the structure of the optimization and the computations needed to solve it are analyzed in detail at the code-generation time to minimize the solve time.

*Draft version.

Specifically, the `TensCalc` toolbox generates C code to solve nonlinear constrained minimizations of the general form

$$f(u^*, p) = \min \{ f(u, p) : F(u, p) \geq 0, G(u, p) = 0, u \in \mathbb{R}^{n_u} \} \tag{1}$$

and to compute two-player Nash equilibria defined by

$$f_u(u^*, d^*, p) = \min \{ f_u(u, d^*, p) : F_u(u, d^*, p) \geq 0, G_u(u, d^*, p) = 0, u \in \mathbb{R}^{n_u} \}, \tag{2a}$$

$$g_u(u^*, d^*, p) = \min \{ g_u(u^*, d, p) : F_d(u^*, d, p) \geq 0, G_d(u^*, d, p) = 0, d \in \mathbb{R}^{n_d} \}. \tag{2b}$$

In either problem, $p \in \mathbb{R}^{n_p}$ denotes a vector of *parameters* that typically changes from one instance of the optimization to the next. The vectors $u \in \mathbb{R}^{n_u}$ and $d \in \mathbb{R}^{n_d}$ correspond to the *optimization variables*, the latter only appearing in the two-play problem (2). The functions

$$f : \mathbb{R}^{n_u} \times \mathbb{R}^{n_p} \to \mathbb{R}, \qquad f_u : \mathbb{R}^{n_u} \times \mathbb{R}^{n_d} \times \mathbb{R}^{n_p} \to \mathbb{R}, \qquad g_d : \mathbb{R}^{n_u} \times \mathbb{R}^{n_d} \times \mathbb{R}^{n_p} \to \mathbb{R}, \tag{3}$$

encode the *optimization criteria*,

$$G : \mathbb{R}^{n_u} \times \mathbb{R}^{n_p} \to \mathbb{R}^{n_G}, \qquad G_u : \mathbb{R}^{n_u} \times \mathbb{R}^{n_d} \times \mathbb{R}^{n_p} \to \mathbb{R}^{n_{Gu}}, \qquad G_d : \mathbb{R}^{n_u} \times \mathbb{R}^{n_d} \times \mathbb{R}^{n_p} \to \mathbb{R}^{n_{Gd}} \tag{4}$$

encode *equality constraints*, and

$$F : \mathbb{R}^{n_u} \times \mathbb{R}^{n_p} \to \mathbb{R}^{n_F}, \qquad F_u : \mathbb{R}^{n_u} \times \mathbb{R}^{n_d} \times \mathbb{R}^{n_p} \to \mathbb{R}^{n_{Fu}}, \qquad F_d : \mathbb{R}^{n_u} \times \mathbb{R}^{n_d} \times \mathbb{R}^{n_p} \to \mathbb{R}^{n_{Fd}} \tag{5}$$

encode (element-wise) *inequality constraints*.

The toolbox's name `TensCalc` results from the merge of the words "tensor" and "calculus" and is motivated by the fact that *tensors* of arbitrary dimension are the basic elements used to construct the optimization criteria and constraints. In fact, while in the description above we restricted the formulation so that the domains and co-domains of all functions in (3)–(5) are organized as real-valued vectors, the toolbox actually accepts the domains and co-domains of all these functions to be Cartesian products of real-valued tensor-spaces of arbitrary sizes, with the exception of the functions in (3) that define the optimization criteria that must produce scalars. The "calculus" in `TensCalc` refers to the fact that the toolbox performs symbolic differentiation of tensors of arbitrary dimension with respect to tensors of arbitrary dimension, which is instrumental to solve the optimizations in (1)–(2).

`TensCalc` has several features that we highlight here:

1. The user-interface uses an optimization modeling language that allows the user to organize the optimization variables, parameters, and constraints as arbitrary collections of vectors, matrices, or high-dimensional tensors that can be manipulated using standard operations of matrix calculus that are intuitive and generally compatible with the MATLAB® syntax. The main features of this interface are described in Section 3.

2. `TensCalc` uses primal-dual interior point algorithms to solve the optimizations (1) and (2). These algorithms, which are discussed in Section 4, use exact formulas for the gradients and hessian matrices that are computed symbolic by `TensCalc`.

3. The sparsity structures of all the gradient and hessian matrices needed by the interior point methods are determined at code-generation time and directly embedded into the code. This enables the memory management to be resolved at code-generation time and the mapping of the different nonzero entries of vectors/matrices to memory locations to be hardwired into the code. Section 5 discusses how sparsity can be promoted and how it is exploited by `TensCalc`.

4. All computations needed to carry out one iteration of the primal-dual interior point method are encoded into a computation graph whose nodes correspond to scalar-valued operations and the edges express computational dependencies between the nodes. Code generation makes use of this graph to minimize the number of computations that need to be performed at each iteration of the primal-dual algorithm. The construction of the computation graph is described in Section 6, which also includes a discussion of how it is used to minimize the memory footprint of the solver.

5. The code generated does not use external libraries, does not require dynamic memory allocation, and has few branches and decision points. As discussed in Section 7, this code is extremely portable, friendly towards compiler optimization, and generally results in an efficient use of instruction pipelining.

2 Related work

The development of languages to define optimization problems has exploded in recent years, mostly prompted by the desire to facilitate applying different numerical solvers to a particular optimization. CVX [10] and YALMIP [12] are the noncommercial optimization modeling languages most closely related to `TensCalc`. Both are offered as free MATLAB® toolboxes and essentially overload the standard MATLAB® functions and operators to enable the specification of objective functions and constraints with MATLAB®-like syntax. CVX is focused on convex optimization and only permits the construction of problems that can be solved using semi-definite programming. YALMIP permits a wider variety of optimizations, supported by a very large collection of internal and external solvers. The optimization modeling language used by `TensCalc` is heavily inspired by CVX and YALMIP and it also overloads the basic MATLAB® operators to accept a MATLAB®-like syntax. Because `TensCalc` is not restricted to convex optimizations, it accepts many constructions not permitted by CVX. However, it falls short of CVX in that its current version does not explicitly accept semi-definite constraints (unless the user expresses them, e.g., as inequality constraints on the minors of a matrix, which is only effective for small matrices). With respect to YALMIP, `TensCalc` falls short in that mixed-integer programs are not allowed and it only permits optimization criteria that are twice differentiable with respect to the optimization variables. The key difference with respect to CVX and YALMIP is that `TensCalc` generates standalone C code that does not require external solvers or external libraries. Moreover, the code generated is highly optimized and can solve one instance of the optimization much faster than what could be achieved with CVX or YALMIP. In this respect, `TensCalc` is much closer to CVXGEN [13], which uses a web-based interface to generate fast custom C code for optimizations that can be expressed as a linear program or a convex quadratic program.

The algorithms used by `TensCalc` to construct the solvers are primal-dual interior-point methods. This class of optimization algorithms was originally proposed for linear programs [14, 15] and essentially amounts to using Newton's method to solve a modified form of the Karush-Kuhn-Tucker (KKT) optimality conditions, with the progression along the Newton direction constrained so that the inequality constraints are not violated. This basic approach can be applied to very general nonlinear programs, but convergence can typically only be guaranteed for certain classes of convex problem, which include linear programming, semidefinite programming, and second-order cone programming [16, 17]. Nevertheless, our experience has been that solving a set of modified KKT optimality conditions using Newton's method is still extremely effective for many nonlinear and nonconvex optimizations.

Primal-dual interior-point methods are very attractive because they generally converge to very accurate solutions with a small number of Newton iterations; typically in the range 10-20, regardless of the problem

size. The main difficulty with these methods lies in the need to solve a linear system of N equations and unknowns to find the Newton direction, where N is the total number of primal and dual variables. Using Gauss elimination to solve these equations generally require $O(N^3)$ floating-point operations [8]. However, when the matrix $H \in \mathbb{R}^{N\times N}$ that defines the system of equations is sparse, finding the search direction can be much less costly. `TensCalc` takes advantage of this and finds the Newton search direction by applying row and column permutations to H to reduce fill-in of the L and U factors [4]. For the minimization problem (1), the matrix H is symmetric and "almost" quasi-definite and therefore admits an LDL factorization for every symmetric permutation [19]. Moreover, these factorizations are generally numerically stable [7]. For the computation of the Nash equilibrium (2), the matrix H is no longer symmetric so numerical issues are more likely to arise.

3 TensCalc's optimization modeling language

The optimizations (1)–(2) are specified by first declaring a set of symbolic variables that correspond to the optimization parameter p and the optimization variables u and d, and then using these variable to construct the optimization criteria f, f_u, g_d and the functions F, F_u, F_d, G, G_u, G_d that define the constraints.

In `TensCalc`, symbolic variables are tensors (i.e., multi-dimensional arrays) and are declared within MATLAB® using the `TensCalc` command

```
1 Tvariable xpto [n1,n2,...,nK]
```

This command creates in the MATLAB® workspace a symbolic variable called xpto and declares it as a tensor[1] with n1 indices in the 1st dimension, n2 indices in the 2nd dimension, etc. The variable xpto is assumed to be full in the sense that `TensCalc` does not assume that any particular entry will always be equal to zero (even though some entries may turn out to be zero).

The variables declared with **Tvariable** can be used to construct arbitrarily complex tensor-valued symbolic expressions using standard MATLAB® syntax. The following MATLAB® functions and operators have been overloaded so that they can be used to construct `TensCalc` expressions: **subsref**, **reshape**, **full**, **vertcat**, **horzcat**, **cat**, **uplus**, **plus**, +, **uminus**, **minus**, −, **sum**, **exp**, **log**, **sqrt**, **cos**, **sin**, **tan**, **atan**, **times**, ∗, **mtimes**, .∗, **rdivide**, and ./. The following additional functions and operators can be use to express constraints: **eq**, ==, **gt**, >, **lt**, and <. Aside from the standard MATLAB® functions defined above, `TensCalc` recognizes a few additional expressions that facilitate constructing expressions using tensors. Among those, we highlight **tprod**, which provides a very flexible generalization of matrix multiplication for tensors of arbitrary size [6].

`TensCalc` symbolic expressions can also use numerical constants that are declared with the `TensCalc` command

```
2 cpto=Tconstant(expr)
```

This command creates a variable called cpto with the value given by the MATLAB® expression expr. `TensCalc` looks for zero entries in expr and will eventually use its sparsity structure to optimize the code. The value of variables declared with **Tconstant** will be hard-wired into the code of the solver, in contrast to the variables declared using **Tvariable** that can be changed from one call to the solver to the next.

The construction of symbolic variables and expressions is supported by a `TensCalc` class called **Tcalculus**, which overloads the standard MATLAB® functions and operators listed above so that they can be applied to

[1]While MATLAB® regards scalars and vectors still as matrices with one or both dimensions with just one element, this is not the case for `TensCalc`.

TensCalc symbolic variables and expressions. All TensCalc symbolic expressions are represented internally through a tree whose nodes are operations between symbolic operands and whose branches connect a node to all its operands. Nodes corresponding to symbolic variables declared using **Tvariable** and **Tconstant** do not have operands, corresponding to final leaves of the tree.

An important operation supported by the class **Tcalculus** is symbolic differentiation, which is carried out through the function

```
3    grad=gradient(expr, var)
```

that computes the derivative of the TensCalc expression expr with respect to the variable var (the latter necessarily declared using **Tvariable**). Both expr and var may be tensors of arbitrary sizes and the resulting symbolic expression grad will be a tensor whose size is the concatenation of the sizes of expr and var. Specifically, if expr and var are tensors with sizes $m_1 \times m_2 \times \cdots \times m_K$ and $n_1 \times n_2 \times \cdots \times n_L$, respectively, then grad has size $m_1 \times m_2 \times \cdots \times m_K \times n_1 \times n_2 \times \cdots \times n_L$, and it entry $(i_1, i_2, \ldots, i_K, j_1, j_2, \ldots, j_L)$ is given by

$$\frac{\partial expr_{i_1,i_2,\ldots,i_K}}{\partial var_{j_1,j_2,\ldots,j_L}}$$

where $expr_{i_1,i_2,\ldots,i_K}$ denotes the entry $(i_1, i_2, \ldots, i_K)$ of expr and $var_{j_1,j_2,\ldots,j_L}$ the entry $(j_1, j_2, \ldots, j_L)$ of var.

The commands **cmex2optimizeCS** and **cmex2equilibriumLatentCS** take TensCalc symbolic expressions and generate the code needed to solve the optimizations (1) and (2), respectively. A typically call to **cmex2optimizeCS** is of the following form:

```
4    cmex2optimizeCS('classname','c1' ,...
5                    'objective',f,...
6                    'optimizationVariables',{x1,x2},...
7                    'constraints',{e1,e2,e3},...
8                    'outputExpressions',{y1,y2},...
9                    'parameters',{p1,p2,p3});
```

where f must be a scalar-valued TensCalc symbolic expression that defines the cost f in (1); x1 and x2 are optimization variables declared using **Tvariable**; p1, p2, and p3 are parameters declared using **Tvariable**; and e1, e2, and e3 are TensCalc symbolic expressions defining equality and/or inequality constraints. This call generates C-code that solves the minimization in (1) and then computes the numerical values of the TensCalc symbolic expressions y1, y2 at the optimum. In addition, **cmex2optimizeCS** also creates a MATLAB® class named c1 that permits access to the solver from within the MATLAB® environment using cmex functions. This class permits passing to the solver numerical values for the parameters p1, p2, and p3; calling the solver; and retrieving the numerical values of y1 and y2. The command **cmex2equilibriumLatentCS** has a similar syntax, but permits the definition of the two objective functions and the two sets of constraints needed by (2). We refer the reader to the appendix for specific examples of calls to **cmex2optimizeCS** and **cmex2equilibriumLatentCS**.

The functions **cmex2optimizeCS** and **cmex2equilibriumLatentCS** internally "reshape" all the symbolic variables declared as parameters and optimization variables into (single-dimension) vectors and stack these vectors appropriately to form the single parameter vector $p \in \mathbb{R}^{n_p}$ and the optimization variables $u \in \mathbb{R}^{n_u}$ and $d \in \mathbb{R}^{n_d}$ that appear in (1). The equality and inequality constraints passed to **cmex2optimizeCS** and **cmex2equilibriumLatentCS** can be expressed through tensors of arbitrary size (with inequalities understood entry-wise). These tensors are also reshaped into (single-dimension) vectors and stacked appropriately to form the functions F, F_u, F_d, G, G_u, G_d that appear in (1) and (2). The reshaping and stacking of variables

and expressions is done symbolically, because we need to subsequently perform symbolic differentiation of the different functions with respect to the optimization variables to compute the gradient and hessian matrices required by the primal-dual interior-point method. However, all this is handled internally by `TensCalc` and hidden from the user.

4 Optimization Algorithms

Both optimizations (1) and (2) are solved using primal-dual interior-point methods based on the results discussed below. For simplicity of presentation, in the remainder of this section we ignore the dependence on p of the functions f, f_u, g_d that define the optimization criteria and of the functions F, F_u, F_d, G, G_u, G_d that define the constraints.

4.1 Minimization

The algorithm used to solve (1) is based on the following simple duality result.

Lemma 1 (Approximate equilibrium)**.** *Suppose that we have found* primal variables $u \in \mathbb{R}^{n_u}$ *and* dual variables $\lambda \in \mathbb{R}^{n_F}$, $\nu \in \mathbb{R}^{n_G}$ *that simultaneously satisfy the following conditions*[2]

$$L_f(u, \lambda, \nu) = \min_{\bar{u} \in \mathbb{R}^{n_u}} L_f(\bar{u}, \lambda, \nu), \tag{6a}$$

$$G(u) = \mathbf{0}_{n_G}, \tag{6b}$$

$$F(u) \geq \mathbf{0}_{n_F}, \qquad \lambda \geq \mathbf{0}_{n_F}, \tag{6c}$$

where $L_f(u, \lambda, \nu) := f(u) - \lambda F(u) + \nu G(u)$*. Then* u *approximately satisfies* (1) *in the sense that*

$$f(u) \leq \epsilon_f + \min_{u \in \mathcal{U}} f(u), \qquad \epsilon_f := \lambda F(u). \tag{7}$$

□

When the functions f, G, and F are continuously differentiable, replacing the unconstrained optimization in (6a) by its first-order necessary condition for optimality and specializing Lemma 1 to the case $\epsilon_f := \lambda F(u) = 0$ leads to the Karush-Kuhn-Tucker (KKT) first-order necessary conditions for optimality. Lemma 1 shows that dropping the KKT complementary slackness condition $\lambda F(u) = 0$ may result in a suboptimal value for u, but the level of suboptimality is no larger than $\epsilon_f := \lambda F(u)$. It convenient that this does not require strong duality. We do not provide here the proof of Lemma 1 because it can be viewed as a special case of Lemma 2 that appears in Section 4.2.

The iterative algorithm used by `TensCalc` to solve (1) consists of using Newton iterations to solve the following system of nonlinear equations on the primal variables $u \in \mathbb{R}^{n_u}$ and on the dual variables $\lambda \in \mathbb{R}^{n_F}$, $\nu \in \mathbb{R}^{n_G}$:

1. the first-order optimality condition for the unconstrained minimizations in (6a)[3]

   $$\nabla_u L_f(u, \lambda, \nu) = \mathbf{0}_{n_u}, \tag{8}$$

   where $\nabla_u L_f$ denotes the gradient of L_f with respect to the variable u;

[2]Given two vectors $x, y \in \mathbb{R}^n$ we denote by $x \geq y$ the entry-wise "greater than or equal to" comparison of the entries of x and y.
[3]Given an integer M, we denote by $\mathbf{0}_M$ and by $\mathbf{1}_M$ the M-vectors with all entries equal to 0 and 1, respectively.

2. the equality constraint (6b); and
3. the equation[4]

$$F(u) \odot \lambda = \mu \mathbf{1}_{n_F}, \tag{9}$$

for some $\mu > 0$, which lead to

$$\epsilon_f := \lambda F(u) = \mu\, n_F.$$

Since our goal is to find primal variables u for which (7) holds with $\epsilon_f = 0$, we shall make the variable μ converge to zero as the Newton iterations progresses. This is done in the context of an interior-point method, meaning that all variables will be initialized so that the inequality constraint (6c) hold *strictly* and, at each iteration, the progression along the Newton direction is selected so that these constraints are never violated. The specific steps of the algorithm that follows are based on the primal-dual interior-point method described in [18].

Algorithm 1 (Primal-dual optimization for the minimization (1))**.**

Step 1. Start with estimates u_0, λ_0, ν_0 *that satisfy the inequality* $\lambda_0 > 0$, $F(u_0) > 0$ *in* (6c) *and set* $k = 0$. *We typically start with*

$$\mu_0 = 1, \qquad \nu_0 = 0, \qquad \lambda_0 = \mu \mathbf{1}_{n_F} \oslash F(u_0),$$

which guarantees that we initially have $\lambda_0 \odot F(u_0) = \mu_0 \mathbf{1}_{n_F}$.

Step 2. Linearize the equations in (8), (6b), (9) *around the current estimate* u_k, λ_k, ν_k*, leading to*

$$\begin{bmatrix} \nabla_{uu} L_f(u_k, \lambda_k, \nu_k) & \nabla_u G(u_k)' & -\nabla_u F(u_k)' \\ \nabla_u G(u_k) & 0 & 0 \\ -\nabla_u F(u_k) & 0 & -\operatorname{diag}[F(u_k) \oslash \lambda_k] \end{bmatrix} \begin{bmatrix} \Delta u \\ \Delta \nu \\ \Delta \lambda \end{bmatrix} = - \begin{bmatrix} \nabla_u L_f(u_k, \lambda_k, \nu_k) \\ G(u_k) \\ -F(u_k) + \mu_k \mathbf{1}_M \oslash \lambda_k \end{bmatrix}, \tag{10}$$

where $\nabla_{uu} L_f$ *denotes the Hessian matrix of* L_f *with respect to* u. *Since* $F(u_k) > 0$ *and* $\lambda_k > 0$, *we can solve this system of equations by first eliminating*

$$\Delta \lambda = -\lambda_k - \operatorname{diag}[\lambda_k \oslash F(u_k)] \nabla_u F(u_k) \Delta u + \mu_k \mathbf{1}_M \oslash F(u_k), \tag{11a}$$

which leads to

$$\begin{bmatrix} \nabla_{uu} L_f(u_k, \lambda_k, \nu_k) + \nabla_u F(u_k)' \operatorname{diag}[\lambda_k \oslash F(u_k)] \nabla_u F(u_k) & \nabla_u G(u_k)' \\ \nabla_u G(u_k) & 0 \end{bmatrix} \begin{bmatrix} \Delta u \\ \Delta \nu \end{bmatrix} = - \begin{bmatrix} \nabla_u f(u_k) + \nabla_u G(u_k)' \nu_k - \mu_k \nabla_u F(u_k)' (\mathbf{1}_M \oslash F(u_k)) \\ G(u_k) \end{bmatrix}. \tag{11b}$$

However, as we shall further discuss in Remark 4, for some problems solving (10) *may actually be preferable to solving* (11).

[4] Given two vectors $x, y \in \mathbb{R}^n$ we denote by $x \odot y \in \mathbb{R}^n$ and by $x \oslash y \in \mathbb{R}^n$ the entry-wise product and division of the two vectors, respectively.

Step 3. Update the estimates along the Newton search direction determined by (11) so that the inequalities in (6c) hold strictly:

$$u_{k+1} = u_k + \alpha \Delta u, \qquad \nu_{k+1} = \nu_k + \alpha \Delta \nu, \qquad \lambda_{k+1} = \lambda_k + \alpha \Delta \lambda \tag{12}$$

where

$$\alpha := \min\{\alpha_{\text{primal}}, \alpha_{\text{dual}}\},$$

and

$$\alpha_{\text{primal}} := \max\Big\{\alpha \in [0,1] : F(u_k + \frac{\alpha}{.99}\Delta u) \geq 0\Big\}, \qquad \alpha_{\text{dual}} := \max\Big\{\alpha \in [0,1] : \lambda_k + \frac{\alpha}{.99}\Delta \lambda \geq 0\Big\}. \tag{13}$$

Step 4. Update μ_k according to

$$\mu_{k+1} = \begin{cases} \gamma_{\text{aggressive}}\mu_k & \text{if } \alpha \geq .5,\ \|G(u_{k+1})\|_\infty \leq 100\epsilon_G,\ \|\nabla_u L_f(u_{k+1}, \lambda_{k+1}, \nu_{k+1})\|_\infty \leq 100\epsilon \\ \gamma_{\text{conservative}}\mu_k & \text{otherwise} \end{cases}$$

with $0 < \gamma_{\text{aggressive}} < \gamma_{\text{conservative}} < 1$. Typically, we use $\gamma_{\text{aggressive}} = 1/3$ and $\gamma_{\text{conservative}} = .99$, which means that we only allow for a significant decrease in μ_k if sufficient progress was possible along the search direction (large value for α)), the equality constrains are approximately satisfied, and the gradient $\nabla_u L_f(u_{k+1}, \lambda_{k+1}, \nu_{k+1})$ is sufficiently small.

Step 5. Repeat from 2 with an incremented value for k until

$$\|G(u_k)\|_\infty \leq \epsilon_G, \qquad \|\nabla_u L_f(u_k, \lambda_k, \nu_k)\|_\infty \leq \epsilon, \qquad \lambda_k' F(u_k) \leq \epsilon_{\text{gap}}. \tag{14}$$

for sufficiently small tolerances ϵ, ϵ_G, ϵ_{gap}. □

`TensCalc` automatically computes the gradients $\nabla_u L_f$, $\nabla_u G$, $\nabla_u F$; the Hessians $\nabla_{uu} L_f$; and assembles the matrix and vectors in (11) based on the symbolic expressions provided by the user. Values for the parameters $\gamma_{\text{aggressive}}, \gamma_{\text{conservative}}$ and the tolerances ϵ, ϵ_G, ϵ_{gap} can be set though (optional) input parameters to the function **cmex2optimizeCS**.

Remark 1 (Algorithm improvements). One can find in [18] several variations of this algorithm that can lead to faster convergence, some of which have been implemented in `TensCalc`. Among these we highlight the following two: (i) one can include in (10) a second-order Mehrotra correction term that often reduces the number of iterations and (ii) at each iteration, one can compute an "optimal" value for μ_k by first solving (10) for the "ideal" value of $\mu = 0$ and then selecting μ_k based on how much progress is possible along the resulting search direction (until the constraints are violated). These variations are discussed at length in [18]. Our experience is that for convex problems they can lead to significant performance improvements, but this is often not the case for nonconvex problems. An additional improvement inspired by [13], consists of replacing (11b) by

$$\begin{bmatrix} \nabla_{uu} L_f(u_k, \lambda_k, \nu_k) + \nabla_u F(u_k)' \operatorname{diag}[\lambda_k \oslash F(u_k)] \nabla_u F(u_k) + \delta I & \nabla_u G(u_k)' \\ \nabla_u G(u_k) & -\delta I \end{bmatrix} \begin{bmatrix} \Delta u \\ \Delta \nu \end{bmatrix}$$

$$= -\begin{bmatrix} \nabla_u f(u_k) + \nabla_u G(u_k)' \nu_k - \mu_k \nabla_u F(u_k)'(\mathbf{1}_M \oslash F(u_k)) \\ G(u_k) \end{bmatrix}$$

for a small constant $\delta > 0$. In this case, the matrix in the left-hand side is quasi-definite in that it has the top-left submatrix positive definite and the bottom-right one negative definite. It turns out that quasi-definite matrices are strongly factorizable, i.e., they admit and LDL factorization for every symmetric permutation, which may not be the case for the original matrix with $\delta = 0$ [19]. Moreover, their LDL factorization is generally numerically stable [7]. It turns out that the addition of $\pm\delta I$ to the diagonal blocks of the matrix in (11b), corresponds to adding similar terms to the top and middle diagonal blocks of the matrix in (10) and this does not alter the fact that the fixed points of the iteration for u_k, ν_k, λ_k (i.e., points for which we have $\Delta u = 0$, $\Delta \nu = 0$, $\Delta \lambda = 0$) necessarily correspond to values of the primal and dual variables for which the right-hand side of (10) is equal to zero, which guarantees that (8), (6b), (9) hold. The selection of δ is guided by a trade-off between doing the exact Newton step ($\delta = 0$), which would results in a smaller number of iterations if computations could be done without numerical errors, and selecting a large value for δ that would improve numerical stability. The heuristic proposed in [19] of setting δ equal to the square root of the arithmetic's precision seems to lead to good results for the vast majority of the problems we have considered, while setting $\delta = 0$ still works very well for a surprisingly large number of problems. □

Remark 2 (Smoothness). Algorithm 1 requires the functions f, F, G to be twice differentiable for the computation of the matrices that appear in (10). However, this does not preclude the use of this algorithm in many optimizations with nonsmooth criteria and/or constraints, because it is often possible to re-formulate non-smooth optimizations into smooth ones by appropriate transformations. Common examples of such optimizations include the minimization of criteria involving ℓ_p norms, such as the (non-differentiable) ℓ_1 and ℓ_∞ optimizations

$$\min\{\|A_{m\times n}x - b\|_{\ell_1} : x \in \mathbb{R}^n\}, \qquad \min\{\|A_{m\times n}x - b\|_{\ell_\infty} : x \in \mathbb{R}^n\}$$

which are equivalent to the following smooth optimizations

$$\min\{v'\mathbf{1}_m : x \in \mathbb{R}^n, v \in \mathbb{R}^m, -v \le Ax - b \le v\},$$
$$\min\{v : x \in \mathbb{R}^n, v \in \mathbb{R}, -v\mathbf{1}_m \le Ax - b \le v\mathbf{1}_m\},$$

respectively. Additional examples of such criteria and the corresponding transformations can be found, e.g., in [9]. □

Remark 3 (Initial feasibility). The algorithm described above must be initialized with a value u_0 for the primal variable that satisfies $F(u_0) > 0$. Often it is straightforward to find initial values for the primal variable that strictly satisfy the inequality constraints. When this is not case, a simple alternative is to introduce an additional optimization variable $s \in \mathbb{R}^{n_F}$ and replace the original inequality constraint $F(u) > 0$ by the following two constraints:

$$F(u) = s, \quad s > \mathbf{0}_{n_F}.$$

It is now trivial to find an initial value for s that satisfies the inequality constraints, e.g., $s_0 = \mathbf{1}_{n_F}$. The price paid is that we have an additional equality constraint $F(u) = s$. □

4.2 Nash equilibrium

The algorithm used to solve (2) is based on the following result from [1].

Lemma 2 (Approximate equilibrium)**.** *Suppose that we have found* primal variables $u \in \mathbb{R}^{n_u}, d \in \mathbb{R}^{n_d}$ *and* dual variables $\lambda_{fu} \in \mathbb{R}^{n_{F_u}}, \lambda_{gd} \in \mathbb{R}^{n_{F_d}}$*,* $\nu_{fu} \in \mathbb{R}^{n_{G_u}}, \nu_{gd} \in \mathbb{R}^{n_{G_d}}$ *that simultaneously satisfy all of the following conditions*

$$L_f(u,d,\lambda_{fu},\nu_{fu}) = \min_{u\in\mathbb{R}^{n_u}} L_f(u,d,\lambda_{fu},\nu_{fu}), \qquad L_g(u,d,\lambda_{gd},\nu_{gd}) = \min_{d\in\mathbb{R}^{n_d}} L_g(u,d,\lambda_{gd},\nu_{gd}), \tag{15a}$$

$$G_u(u,d) = 0, \qquad G_d(u,d) = 0, \tag{15b}$$

$$F_u(u,d) \geq \mathbf{0}_{n_{F_u}}, \quad \lambda_{fu} \geq \mathbf{0}_{n_{F_u}}, \qquad F_d(u,d) \geq \mathbf{0}_{n_{F_d}}, \quad \lambda_{gd} \geq \mathbf{0}_{n_{F_d}}, \tag{15c}$$

where

$$L_f(u,d,\lambda_{fu},\nu_{fu}) := f_u(u,d) - \lambda_{fu}F_u(u,d) + \nu_{fu}G_u(u,d),$$
$$L_g(u,d,\lambda_{gd},\nu_{gd}) := g_d(u,d) - \lambda_{gd}F_d(u,d) + \nu_{gd}G_d(u,d).$$

Then (u,d) *approximately satisfy* (2) *in the sense that*

$$f(u,d) \leq \epsilon_f + \min_{u\in\mathcal{U}[d]} f(u,d), \qquad g(u,d) \leq \epsilon_g + \min_{d\in\mathcal{D}[u]} g(u,d), \tag{16}$$

with

$$\epsilon_f := \lambda_{fu}F_u(u,d), \qquad \epsilon_g := \lambda_{gd}F_d(u,d). \qquad \square$$

The algorithm used by `TensCalc` to solve (2) now consists of using Newton iterations to solve the following system of nonlinear equations on the primal variables $u \in \mathbb{R}^{n_u}, d \in \mathbb{R}^{n_d}$ and on the dual variables $\lambda_{fu} \in \mathbb{R}^{n_{F_u}}, \lambda_{gd} \in \mathbb{R}^{n_{F_d}}, \nu_{fu} \in \mathbb{R}^{n_{G_u}}, \nu_{gd} \in \mathbb{R}^{n_{G_d}}$ introduced in Lemma 2:

1. the first-order optimality conditions for the unconstrained minimizations in (15a):

$$\nabla_u L_f(u,d,\lambda_{fu},\nu_{fu}) = \mathbf{0}_{n_u}, \qquad \nabla_d L_g(u,d,\lambda_{gd},\nu_{gd}) = \mathbf{0}_{n_d}; \tag{17}$$

2. the equality constraints (15b); and

3. the equations

$$F_u(u,d) \odot \lambda_{fu} = \mu \mathbf{1}_{n_{F_u}}, \qquad F_d(u,d) \odot \lambda_{gd} = \mu \mathbf{1}_{n_{F_d}}, \tag{18}$$

   for some $\mu > 0$, which lead to

$$\epsilon_f := \lambda_{fu}F_u(u,d) = \mu\, n_{F_u}, \qquad \epsilon_g := \lambda_{gd}F_d(u,d) = \mu\, n_{F_d}.$$

Since our goal is to find primal variables u, d for which (16) holds with $\epsilon_f = \epsilon_g = 0$, we shall make the variable μ converge to zero as the Newton iterations progress. Defining

$$z := \begin{bmatrix} u \\ d \end{bmatrix}, \qquad \lambda := \begin{bmatrix} \lambda_{fu} \\ \lambda_{gd} \end{bmatrix}, \qquad \nu := \begin{bmatrix} \nu_{fu} \\ \nu_{gd} \end{bmatrix}, \qquad G(z) := \begin{bmatrix} G_u(u,d) \\ G_d(u,d) \end{bmatrix}, \qquad F(z) := \begin{bmatrix} F_u(u,d) \\ F_d(u,d) \end{bmatrix},$$

we can re-write (17), (15b), and (18) as

$$\nabla_u L_f(z,\lambda,\nu) = \mathbf{0}_{n_u}, \quad \nabla_d L_g(z,\lambda,\nu) = \mathbf{0}_{n_d}, \quad G(z) = \mathbf{0}_{K_u+K_d}, \quad \lambda \odot F(z) = \mu \mathbf{1}_{M_u+M_d}, \tag{19a}$$

and (15c) as

$$\lambda \geq \mathbf{0}_{M_u+M_d}, \qquad F(z) \geq \mathbf{0}_{M_u+M_d}. \tag{19b}$$

These equations are similar in structure to the ones that we encountered in (8), (6b), (9), permitting the development of an algorithm to solve (2) that is essentially identical to the Algorithm 1 used to solve (1). The key difference is that the Newton search direction is now obtained through the linearization of (19a) around a current estimate z_k, λ_k, ν_k, leading to

$$\begin{bmatrix} \nabla_{uz}L_f(z_k,\lambda_k,\nu_k) & \nabla_{u\nu}L_f(z_k) & \nabla_{u\lambda}L_f(z_k) \\ \nabla_{dz}L_g(z_k,\lambda_k,\nu_k) & \nabla_{d\nu}L_g(z_k) & \nabla_{d\lambda}L_g(z_k) \\ \nabla_z G(z_k) & 0 & 0 \\ \nabla_z F(z_k) & 0 & \operatorname{diag}[F(z_k)\oslash\lambda_k] \end{bmatrix} \begin{bmatrix} \Delta z \\ \Delta\nu \\ \Delta\lambda \end{bmatrix} = - \begin{bmatrix} \nabla_u L_f(z_k,\lambda_k,\nu_k) \\ \nabla_d L_g(z_k,\lambda_k,\nu_k) \\ G(z_k) \\ F(z_k) - \mu\mathbf{1}\oslash\lambda_k \end{bmatrix}. \tag{20}$$

where $\nabla_{xy}L_f$ denotes the Hessian matrix of L_f with respect to the variables x and y. Also here we can solve this system of equations by first eliminating

$$\Delta\lambda = -\lambda_k - \operatorname{diag}[\lambda_k \oslash F(z_k)]\nabla_z F(z_k)\Delta z + \mu\mathbf{1}\oslash F(z_k) \tag{21a}$$

which leads to

$$\begin{bmatrix} \nabla_{uz}L_f(z_k,\lambda_k,\nu_k) - \nabla_{u\lambda}L_f(z_k)\operatorname{diag}[\lambda_k\oslash F(z_k)]\nabla_z F(z_k) & \nabla_{u\nu}L_f(z_k) \\ \nabla_{dz}L_g(z_k,\lambda_k,\nu_k) - \nabla_{d\lambda}L_g(z_k)\operatorname{diag}[\lambda_k\oslash F(z_k)]\nabla_z F(z_k) & \nabla_{d\nu}L_g(z_k) \\ \nabla_z G(z_k) & 0 \end{bmatrix} \begin{bmatrix} \Delta z \\ \Delta\nu \end{bmatrix}$$
$$= - \begin{bmatrix} \nabla_u f(z_k) + \nabla_u((\nu_{fu})_k G_u(z_k)) + \mu\nabla_{u\lambda}L_f(z_k)(\mathbf{1}\oslash F(z_k)) \\ \nabla_d g(z_k) + \nabla_d((\nu_{gd})_k G_d(z_k)) + \mu\nabla_{d\lambda}L_g(z_k)(\mathbf{1}\oslash F(z_k)) \\ G(z_k) \end{bmatrix} \tag{21b}$$

However, a notable difference between the previous system of equations in (11b) and the one in (21b) is that now the matrix in the left-hand side is no longer symmetric, which forces an LU factorization, rather than an LDL factorization. Aside from computationally more intensive, the LU factorization can also be numerically more unstable.

In the remainder of the paper, we focus our discussion in the code generation for the Algorithm 1 that solves (1), with the understanding that the same discussion applies to the algorithm that solves (2), with (11b) replaced by (21b).

5 Promoting and Exploring Sparsity

The bulk of the computation needed to solve (1) and (2) is associated with constructing and solving the system of equations (11b) and (21b) used to compute the Newton search directions. Using Gauss elimination to solve these equations generally require $O(N^3)$ floating-point operations [8], where N denotes the total number of entries in the primal variables plus the number of entries in the dual variables associated with the equality constraints. However, the matrices that appear in (11b) and (21b) often have a large number of "structurally zero" entries. By "structurally zero", we mean that these entries will be zero for every iteration of the algorithm and that this can be determined at code-generation time. As we shall see, this permits the construction of solvers with memory and computation complexities much better than $O(N^3)$, often with complexities that scale only linearly with the problem size.

Aside from the zero bottom-right block that we see in the matrices in (11b) and (21b), the remaining blocks of these matrices typical have numerous zero entries. This is explained by two main reasons that we discuss in the context of (11b):

1. In general, most equality constraints (corresponding to the rows of $G(u) = 0$) do not involve every single optimization variable in the vector u and therefore $\nabla_u G(u_k)$ will have a large number of structurally zero entries.

2. Often, many second-order derivatives of $L_f(u, \lambda, \nu) := f(u) - \lambda F(u) + \nu G(u)$ with respect to pairs of variables in u, λ, and ν are structurally zero. For example, any second-order partial derivative with respect to the pair (u_i, λ_j) is nonzero only for those variable u_i that appear in the inequality constraint corresponding to the jth row of $F(u)$.

5.1 Promoting Sparsity

The computation and memory savings due to sparsity can be so significant that it is often beneficial to introduced latent variables and equality constraints to obtain larger but more sparse matrices in (11b) and (21b). In fact, this is almost always the case in problem arising in MPC and/or MHE, where the introduction of the system's state as additional optimization variables (subject to equality constrains corresponding to the system dynamics) results in much more scalable problems. To understand this, consider a prototypical constrained LQR optimization with a cost function of the form:

$$J = \sum_{k=1}^{N} x_k^2 + u_k^2, \tag{22a}$$

where

$$x_1 = 10, \qquad x_{k+1} = x_k + u_k, \quad \forall k \in \{1, 2, \dots, N-1\}, \tag{22b}$$

subject to the constraints that

$$|u_k| \le 1, \quad \forall k \in \{1, 2, \dots, N\}. \tag{22c}$$

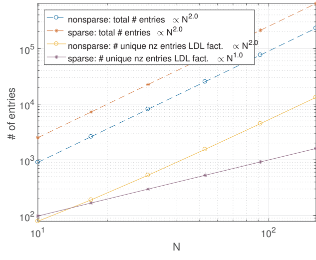
One option to solve this problem consists of using (22b) to conclude that

$$x_k = 10 + \sum_{j=1}^{k-1} u_j, \quad \forall k \in \{1, 2, \dots, N-1\},$$

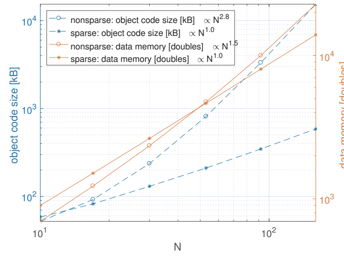
and then replacing x_k in (22a) to express the cost function solely in terms of the optimization variables $u := \{u_1, \dots, u_N\}$, leading to a problem of the form (1) with no equality constraints and the following cost and inequality constraints:

$$f(u) := \sum_{k=1}^{N} \Big(10 + \sum_{j=1}^{k-1} u_j\Big)^2 + u_k^2, \qquad F(u) := \begin{bmatrix} 1 - u \\ 1 + u \end{bmatrix} \ge 0. \tag{23}$$

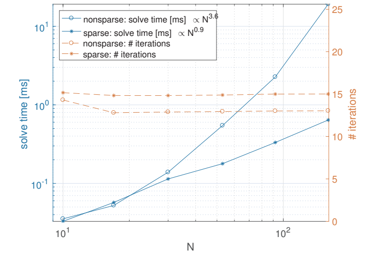
An alternative consists of regarding both the $u := \{u_1, \dots, u_N\}$ and the $x := \{x_1, \dots, x_N\}$ as optimization variables and taking the (22b) as equality constraints. This would still lead to a problem of the form (1), but



(a) Total number of entries and number of nonzero distinct entries in the factorization of the matrix in (11b).



(b) Memory used for data (in units of doubles) and size of object code (in kB) for the C-implementation of the solver.



(c) Average solve time (in milliseconds) and number of Newton iterations for random initializations of the solver.

Figure 1: Comparison of the solvers associated with the two alternative formulations (23) and (24) of the same original problem (22), for different values of the horizon length N. In the legends, "nonsparse" refers to the formulations (23) and "sparse" refers to the formulation in (24). Object code and solver times were measured in a MacBook Pro 2.6 GHz Intel Core i7, running OSX 10.11.6 with the clang compiler version Apple LLVM 8.0.0 with optimization flag set to –O1.

with

$$f(u,x) := \sum_{k=1}^{N} x_k^2 + u_k^2, \qquad F(u,x) := \begin{bmatrix} 1-u \\ 1+u \end{bmatrix} \geq 0, \qquad G(u,x) = \begin{bmatrix} x_1 - 10 \\ x_2 - (x_1 + u_1) \\ \vdots \\ x_N - (x_{N-1} + u_{N-1}) \end{bmatrix} = 0, \tag{24}$$

with the understanding that now the optimization variables include both u and x.

The matrix in (11b) for (23) is $N \times N$ but most of its entries are nonzero because most second-order derivatives of $f(u)$ in (23) are nonzero. In contrast, the same matrix for (24) is $3N \times 3N$ and therefore has 9 times the number of entries. However, most second order derivatives of $f(u,x)$ in (24) are zero. In fact, every second order derivative with respect to u_i, u_j, $i \neq j$ and with respect to u_i, x_j is zero. This means that the larger $3N \times 3N$ matrix corresponding to (24) actually has a much smaller number of (structurally) nonzero entries than the $N \times N$ matrix corresponding to (23). This ultimately leads to a solver for (24) that requires less memory and is faster. We can see in Figure 1 that the formulation (23) leads to a number of nonzero entries of the matrix in (11b) that scales with N^2, which eventually leads to data size that scales roughly with $N^{1.5}$, code size that scales roughly with $N^{2.8}$, and solve times that scale roughly with $N^{3.6}$. In contrast, the formulation (24) leads to a number of nonzero entries of the matrix in (11b) that scales linearly with N, data and code sizes that scale linearly with N, and solve times that also scale roughly with N.

5.2 Exploiting Sparsity

The sparsity of the matrices and vectors that appear in (11b) and (21b) is mostly explored in three operations:

1. Additions/subtractions of matrix/tensor entries that are structurally zero can be omitted.

2. Multiplications by matrix/tensor entries that are structurally zero can be omitted and, in fact, render the products structurally zero.

3. The system of equations in (11b) and (21b) are solved by performing LDL or LU factorizations of the matrices in the left-hand side, which can be made much more efficient by taking advantage of the entries that are structurally zero. For the particular case of (11b), the matrix to factorize is symmetric and therefore we can perform an LDL factorization, which generally has a lower memory and computation cost [8].

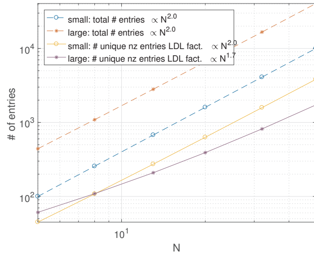
   To make use of the sparsity structure of the left-hand side matrices in (11b) and (21b), we perform the LDL and LU factorizations with pivoting and column permutation to reduce fill-in of the L and U factors. We use the MATLAB® implementation of the COLAMD algorithms that computes approximate minimum degree orderings for sparse matrices [4, 5]. For symmetric matrices we use the SYMAMD algorithm described in the same references. For most of the problems we have encountered, this algorithm results in a total number of nonzero entries in the L and U factors that is of the same magnitude as the number of nonzero entries in the original matrix and that scales similarly with the size of the problem. For example in the problem in Figure 1, the permutations computed using the SYMAMD algorithm lead to a number of nonzero entries in the L factor of the LDL factorization of the matrix in (11b) that is roughly half the total number of nonzero entries in the original matrix.

   To encode the sparsity structure of the matrix in the C-code, the row and column permutations must be determined at code-generation time. At this time, the sparsity structure of the matrix is know, but precise values of the nonzero entries are generally not know. We use two alternative options to overcome this problem: In the absence of any additional information, we generate a random matrix with the known sparsity structure, but with random entries uniformly distributed in the interval [0,1] for the nonzero entries. This matrix is used to obtain row and column permutations that minimize fill-in.

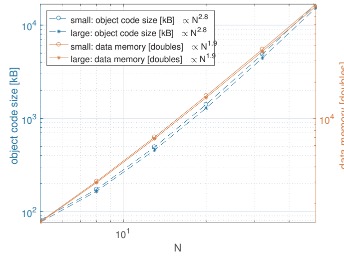
   While the use of a random matrix works for a large majority of problems, in some cases it may generate row-permutations that result in numerically unstable pivoting; especially in the computation of Nash equilibria, where the matrix in (21b) cannot be made quasi-definite. To overcome this, we also permit the user to provide "typical" values for the entries of the matrix, which are used to compute the permutations. To facilitate the generation of "typical" values, the C-code solver can save snapshots of the values of the matrices being factored.

   One challenge with either option is that the row and column permutations will be hardwired into the code so they will have to remain the same for every Newton step. However, our experience has been that this is not a significant issue for most problems.

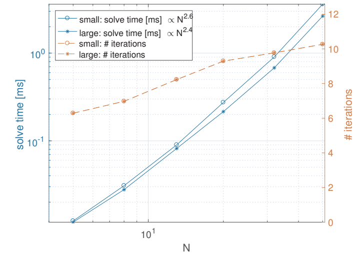
Remark 4. In Step 2 of Algorithm 1, one can compute the search direction by solving either (10) or (11). In general, the latter is preferable because it involves a smaller system of equations, but this is not always the case. When $\nabla_u F(u_k)$ has one or more rows that are not sparse, the product $\nabla_u F(u_k)' \operatorname{diag}[\lambda_k \oslash F(u_k)] \nabla_u F(u_k)$ is full and the top-left block in the matrix in (11) becomes full, even if $\nabla_{uu} L_f(u_k, \lambda_k, \nu_k)$ is sparse. In such cases, it may be preferable to work with (10) directly. To some extent, we see this in the Lasso optimization that we shall encounter in Section 8.2. We can see in Figure 2 that for $N \geq 8$ the larger matrix in (10) actually leads to a smaller number of nonzero entries in the LDL factorization, slightly smaller code, and a faster solver. □



(a) Total number of entries and number of nonzero distinct entries in the factorization of the matrix in (11b).



(b) Memory used for data (in units of doubles) and size of object code (in kB) for the C-implementation of the solver.



(c) Average solve time (in milliseconds) and number of Newton iterations for random initializations of the solver.

Figure 2: Comparison of the solvers for the Lasso optimization described in Section 8.2 obtained by solving (10) (label "large") and (11) (label "small"). Object code and solver times were measured in a MacBook Pro 2.6 GHz Intel Core i7, running OSX 10.11.6 with the clang compiler version Apple LLVM 8.0.0 with optimization flag set to −O1.

6 Scalarization and Computation Graph

The first step towards generating the C-code needed to implement Algorithm 1 consists of "scalarizing" all computations needed for a single iteration of the algorithm and organizing these computations in the form of a *computation graph*. By "scalarized", we mean that we break each vector- or matrix-valued computation that appears in Algorithm 1 into a set of *primitive* operations, each producing a single scalar corresponding to one entry of a matrix or vector needed by Algorithm 1. For example, the update step

$$u_{k+1} = u_k + \alpha \Delta u \ \in \mathbb{R}^{n_u} \tag{25}$$

is converted into n_u multiplications of the scalar α by each entry of Δu, followed by n_u additions that correspond to summations of each entry of u_k with the the corresponding entry of $\alpha\Delta u$. In general, the most expensive computation that needs to be scalarized is the LDL factorization of the matrix in the left-hand side of (11b), which is used to solve (11b) for the $n_u + n_G$ unknowns that correspond to the entries of $\Delta u \in \mathbb{R}^{n_u}$ and $\Delta \nu \in \mathbb{R}^{n_G}$. The scalarization of this and all other operations is done at code generation time and takes advantage of the structural sparsity discussed in Section 5. For example, no primitive operation is generated to perform an addition with or multiplication by an entry of a matrix/tensor that is structurally zero.

The process of scalarization builds a "computation graph" that encodes dependencies between primitives operations. The *computation graph* is a directed graph $G = (\mathcal{N}, \mathcal{E})$, where $\mathcal{N}$ denotes the set of nodes and $\mathcal{E} \subset \mathcal{N} \times \mathcal{N}$ the set of edges. Each node $i \in \mathcal{N}$ corresponds to a scalar-valued variable s_i that is needed to compute an entry of a matrix/vector used by Algorithm 1. For example, the computation of the update step in (25) will require $1 + 3n_u$ nodes: 1 to store the value of α, n_u to store the values of Δu, n_u to store the values of the intermediate computation $\alpha\Delta u$, and finally n_u nodes to store the final u_{k+1}. This does not means that this computation will need $1 + 3n_u$ distinct memory locations to produce u_{k+1} (clearly it will not, because we can reuse memory), but it will need $1 + 3n_u$ nodes in the computation graph that is constructed at code generation time. The number of nodes in $\mathcal{N}$ essentially equals the total number of scalar-valued computations needed to execute one iteration of Algorithm 1, starting from scratch.

The edges of $G = (\mathcal{N}, \mathcal{E})$ encode dependencies between the scalars corresponding to the nodes. Specifically, an edge $(i, j) \in \mathcal{E}$ from a parent node $i \in \mathcal{N}$ to a child node $j \in \mathcal{N}$ indicates that the computation of

the scalar s_j requires the value of the scalar s_i.

Algorithm 1 interacts with the computation graph through "set events" and "get events." *Set events* correspond to assigning values to the scalars s_i and take place during the initialization of the algorithm in Step 1 to assign values to an optimization parameter (see Section 3), to initialize μ_0, to the initialize the primal variable u_0, and to the initialize the dual variables ν_0, λ_0. Set events also occur at the end of each iteration in Step 3 to update the values of the primal and dual variables, in preparation for the next iteration. Algorithm 1 also interact with the computation graph through *get event*, which correspond to getting the values of the scalars s_i. Get events occur in the update of the primal and dual variables in Step 3, the update of μ_k in Step 4, checking the termination condition in Step 5, and, upon termination, the computation of the output expressions selected by the user (see Section 3). Note that the updates in (12) actually require both get and set events: first get events to obtain the values of $u_k + \alpha\Delta u_k$, $\nu_k + \alpha\Delta\nu_k$, $\lambda_k + \alpha\Delta\lambda_k$, followed by a set events that overwrite these values in the nodes corresponding to u_k, ν_k, λ_k, in preparation to the following iteration of the algorithm.

As discussed below, the computation tree is heavily used in the process of generating code for the solver: it is needed to determine the correct scheduling of all the computations involved in one iteration of Algorithm 1 and it also enables several forms of optimization that significantly decrease the run-time of the solver.

6.1 Avoiding redundant computations

We can see in Algorithm 1 that specific computations are used multiple times in each iteration: for example $G(u_k)$ appears in Step 2 in the vector in the right-hand side of (11b), in Step 4 to update μ_k, and in the termination conditions in Step 5. Moreover, the specific structure of the optimization criteria f and the constraint functions F and G typically hide additional computation redundancies. For example, if $f(u) = u'Pu$ then $\nabla_u f(u) = u'P$, which means that we have $f(u) = u'\nabla_u f(u)$ and therefore the computation of $f(u)$ can make use of computations previously done to obtain $\nabla_u f(u)$ or vice-versa.

Computational redundancies like the ones discussed above, can be detected automatically when constructing the computation graph because they lead to two or more nodes that correspond to the same primitive computation (i.e., the same operand) with the same set of parent nodes (i.e., the same operands). When this is detected a new node is not added to the graph and, instead, the existing node is reused.

In addition, some "computations" involving matrices and vectors also do not lead to new nodes in the computation tree. For example, the concatenation of the matrix in (11b) from the 4 block

$$\nabla_{uu} L_f(u_k, \lambda_k, \nu_k) + \nabla_u F(u_k)' \operatorname{diag}[\lambda_k \oslash F(u_k)] \nabla_u F(u_k), \qquad \nabla_u G(u_k)', \qquad \nabla_u G(u_k), \qquad 0 \tag{26}$$

or the partition the vector $[\,\Delta u' \;\; \Delta\nu\,]'$ into the two distinct variables Δu and $\Delta\nu$ do not require new nodes in the computation tree, which means that these operations do not consume memory or computation time in the final C-code. Instead, the LDL-factorization of the matrix in (11b) directly uses the nodes associated with the values of the entries of the three non-zero block matrices in (26) and the computations in (11a), (12) directly use the nodes associated with the entries of the vector $[\,\Delta u' \;\; \Delta\nu\,]'$ obtained by solving (11b). In practice, this means that some of the operations used to construct `TensCalc` expressions (including **subsref**, **reshape**, **vertcat**, **horzcat**, **cat**) do not require adding new nodes to the computation tree and also do not translate into C-code for the solver.

6.2 Computation scheduling

The computation graph $G = (\mathcal{N}, \mathcal{E})$ permits the establishment of an order of computation that makes sure that a parent node is evaluated before all its child nodes are evaluated. This is done by performing a topological sorting of the graph nodes. Specifically, we need to assign to each node $i \in \mathcal{N}$ an integer o_i such that

$$(i, j) \in \mathcal{E} \quad \Rightarrow \quad o_i < o_j. \tag{27}$$

If we then order the computation of the nodes by increasing values of the o_i, we can be sure that if the computation corresponding to the child node j requires the value produced by the parent node i, then the parent node will be computed before the child node.

Topological sorting can be performed very rapidly for large graphs. `TensCalc` uses the topological sorting algorithm in [11], which sorts graphs with 100,000s of nodes in just a few milliseconds. This sorting is performed at code-generation time to make sure that the solver's code performs the computations needed by Algorithm 1 in an appropriate order.

6.3 Minimizing recomputation

Some but not *all* values of the matrices and vectors in Algorithm 1 change from one iteration to the next. For example, at each iteration all entries of u_k will typically change, but many of the entries in the matrix and vector in (11b) will not. This is the case, e.g., when we have linear equality constraints for which the corresponding rows of $\nabla_u G(u_k)$ are constant and independent of u_k. Similarly, many entries of $\nabla_{uu} L_f$, $\nabla_{uv} L_f$, $\nabla_{u\lambda} L_f$ remain unchanged from one iteration to the next.

The computation graph $G = (\mathcal{N}, \mathcal{E})$ can be used to determine the smallest set of nodes that needs to be recomputed to perform each iteration of Algorithm 1. To accomplish this, we associate to each node $i \in \mathcal{N}$ a Boolean variable b_i that indicates that the scalar s_i associated with the node i needs to be (re)computed. All the b_i, $i \in \mathcal{N}$ should be initialized with **true** to indicate that all nodes need to be computed and then set to **false** once a node is computed. Any subsequent set event that changes the values s_i of the node $i \in \mathcal{N}$ should reset all the descendants of i back to **true**, to trigger subsequent recomputations of those nodes. Specifically, a set event that assigns values to a given group of nodes is implemented as follows:

```
10  function set_event(nodes, values)
11     for i in nodes
12         s_i = values[i]
13     for j in descendants_of(nodes)
14         b_j = true
15  end
```

where descendants_of(nodes) returns a set containing all children of the nodes in nodes, their children's children, and so on. A get event that retrieves the values of a given group of nodes is implemented as follows:

```
16  function get_event(nodes)
17     for i in topological(ancestors_of(nodes))
18         if b_i == true
19            { ... compute node s_i ... }
20            b_i = false
21     return [s_i for i in nodes]
22  end
```

where $\text{ancestors_of}(\text{nodes})$ returns a set contain all nodes in nodes, their parents, their parents' parents, and so on; and $\text{topological}(s)$ sorts the set of nodes x according to the topological order o_i in (27).

In practice, the use of one Boolean variable b_i for each node $i \in \mathcal{N}$ would lead to a prohibitively expensive overhead. However, this is not needed because the functions above will result in large groups of nodes always having the same values for their variables b_i. To understand why this is so, suppose that we say that two nodes $i, j \in \mathcal{N}$ are *dependency equivalent* if the following two properties hold:

1. for every set event $\text{set_event}(\text{nodes}, \text{values})$ required by Algorithm 1, the nodes i and j either both belong to $\text{descendants_of}(\text{nodes})$ or none thus; and
2. for every get event $\text{get_event}(\text{nodes})$ required by Algorithm 1, the nodes i and j either both belong to $\text{ancestors_of}(\text{nodes})$ or none thus.

Dependency equivalence defines an equivalence relation in the set of nodes $\mathcal{N}$, which can be used to partition $\mathcal{N}$ into equivalence classes that we call *dependency groups* and denote by

$$\{\mathcal{G}_1, \mathcal{G}_2, \dots, \mathcal{G}_L\}, \qquad \mathcal{N} = \bigcup_{k=1}^{L} \mathcal{G}_k, \quad \mathcal{G}_k \cup \mathcal{G}_\ell = \emptyset, \ k \neq \ell, \tag{28}$$

From the definition of dependency equivalence, for every two nodes $i, j \in \mathcal{N}$ in the same equivalence class, the variables b_i and b_j will always remain equal to each other since all calls to $\text{set_event}(\text{nodes}, \text{values})$ and $\text{get_event}(\text{nodes})$ assign the same value to both variables. This means that we do not need one Boolean variable b_i for each node $i \in \mathcal{N}$; instead we only need one Boolean variable b_k for each equivalence class $\mathcal{G}_k$. While the total number of nodes in a computation graph often grows to the 100,000s, the number of equivalence classes rarely grows above 100. It should also be noted that the number of equivalence classes and which equivalence classes are relevant for each set and get event required by Algorithm 1 can be determined at code-generation time. This means that one can hard-wire in the code generated for each set and get event the precise list of variables b_k that need to be tested and set.

6.4 Minimizing memory footprint

The computation graph $G = (\mathcal{N}, \mathcal{E})$ and the dependency groups in (28) are also used to determine the run-time memory required by the computations involved in Algorithm 1, which avoids run-time memory allocation and garbage collection.

Since each node $i \in \mathcal{N}$ corresponds to a scalar number s_i involved in the computation of all the matrices and vectors needed for Algorithm 1, the memory required to store all the computations is never larger than the number of nodes in the graph. However, one can typically substantially reduce the memory footprint of Algorithm 1 by reusing memory locations for different nodes. Specifically, a node i in a dependency group $\mathcal{G}_k$ can reuse a memory location m associated with a node j in the same equivalence group $\mathcal{G}_k$ if the following three conditions hold:

1. the value s_j is not the output of a $\text{get_event}(\text{nodes})$ function;
2. j has no children outside $\mathcal{G}_k$; and
3. j has no child within $\mathcal{G}_k$ that is computed after i, where "computed after" refers to a topological order in (27).

The first item guarantees that we do not discard a value s_j that needs to be returned by get_event (nodes); the second item guarantees that we do not discard a value s_j that could subsequently be needed by a computation in a different dependency group; and the third item guarantees that we do not discard s_j before it is used by its own children nodes within the same dependency group.

The assignment of graph nodes to memory locations is performed at code-generation time, based on the memory reuse rules outlined above. This means that we can store all (non-zero) entries of all the matrices and vectors needed for Algorithm 1 (including all intermediate computations) in a single linear array that can be statically allocated.

7 Code Generation

The code generated to solve Algorithm 1 consists of the following collection of C functions:

1. Several set_event (nodes, values) and get_event (nodes) functions to perform all the computations required by Algorithm 1.
2. One function compute_group(k) for each dependency group $\mathcal{G}_k$ that updates the values s_i associated with all the nodes $i \in \mathcal{G}_k$. These functions are called in line 19 of the get_event (nodes) pseudo-code to compute the values of the nodes.
3. A solve () function that contains the main loop of Algorithm 1 and calls set_event (nodes, values) and get_event (nodes) functions to perform all the required computations.

The solve () is described by the following pseudo-code:

```
23  function solve()
24      set_event( parameter_nodes, parameters_values );
25      set_event( mu_node, mu0_initial_value );
26      set_event( primal_variables_nodes, primal_variables_initial_values)
27      set_event( dual_variables_nodes, dual_variables_initial_values)
28      repeat {
29        (u,nu,lambda) = get_event(next_primal_variables_nodes,
30                                  next_dual_variables_nodes)
31        set_event(primal_variables_nodes, u)
32        set_event(dual_variables_nodes, (nu,lambda) )
33        stop = get_event(termination_condition_node)
34      } until (stop)
35      return get_event(output_expression_nodes)
36  end
```

where parameter_nodes refer to the nodes of the computation graph that hold the values of the optimization parameters declared in line 9 of the call to **cmex2optimizeCS**, mu_node refer to the node that holds the value of μ_k in (12), primal_variables_nodes refer to the nodes that hold the value of the primal variable u_k, dual_variables_nodes refer to the nodes that hold the value of the dual variables ν_k, λ_k, next_primal_variables_nodes refer to the nodes that hold the value of the primal variable u_{k+1} in (12), next_dual_variables_nodes refer to the nodes that hold the value of the dual variables ν_{k+1}, λ_{k+1} in (12), termination_condition_node refers to the node associated with the conjunction of the three conditions in (14), and output_expression_nodes refer to the nodes hold the values of the output expressions declared in line 8 of the call to **cmex2optimizeCS** (see Section 3).

The bulk of the computation time is spent in the computations of the update for the primal and dual variables in line 29 of the pseudo-code above; in particular, in the functions compute_group(k) that actually perform the update of the node values s_i associated with the appropriate dependency groups.

The code for the compute_group(k) functions is generated directly based on the computation graph, with each node of the graph resulting in 1-2 lines of C code that perform the corresponding primitive computation. The order in which the computations appear in the code is determined by the topological order discussed in Section 6.2. The functions compute_group(k) make no use of external libraries and, due to scalarization process used to compute the computation graph, are essentially loop-free, because all vector/matrix operations have been “unrolled” into primitive scalar-value operations. While this generally results in large object-code, the resulting code is very portable and leads to very few execution branches, which improves micro-processor pipelining. Also, since all the memory mapping is resolved at computation time, this code does not need real-time dynamic memory allocation, further improving portability and efficiency.

The assignment of computations to memory locations is performed at code generation time and maximizes memory reuse, as discussed in Section 6.4. This permits storing all computations in a single array, which we call the *scratchbook*, that is allocated when the code starts to execute, either as a global variable or dynamically allocated with a single malloc() instruction. The same *scratchbook* variable is reused not only across the multiple iterations of Algorithm 1, but also across multiple calls to the solver with different optimization parameters, which means that nodes of the computation graph that are not changed from one call to solve() to another are reused. It is not uncommon, for significant portions of the matrix in (11b) to remain the same across multiple optimizations, which means that portions of the computationally expensive LDL/LU-factorization can be reused.

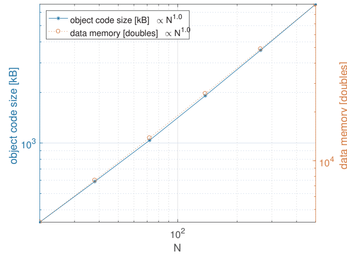
7.1 MATLAB® interface to the solver

While `TensCalc` generates C-code that can run completely independent from MATLAB®, this toolbox also generates wrapper code to call the solver from within the MATLAB® environment. The solver code is compiled into a library that is dynamically linked to MATLAB® and cmex functions are created to

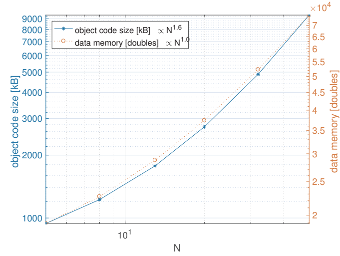
1. call the set_event() functions that are needed to set the optimization parameters declared in line 9 of the call to **cmex2optimizeCS** and to initialize the primal variables declared in lines 6 of the call to **cmex2optimizeCS**;
2. call the function solver() discussed above that executes Algorithm 1;
3. cal the get_event() functions used to get the output expressions declared in line 8 of the call to **cmex2optimizeCS**.

`TensCalc` also generates code that creates a MATLAB® class to provide an object-oriented interface to the solver, with methods to call the different cmex functions.

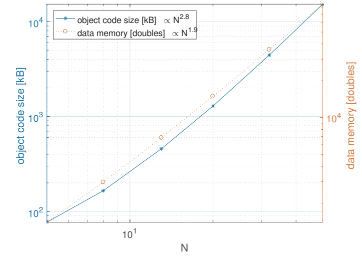
In addition to the **cmex2optimizeCS** and **cmex2equilibriumLatentCS** functions that generate C-code to solve (1) and (2), respectively, the `TensCalc` toolbox includes sister functions **class2optimizeCS** and **class2equilibriumLatentCS** that generate MATLAB® classes to solve (1) and (2), respectively. These classes use the same interior-point algorithms described in Section 4, but implement these algorithms in MATLAB® without performing the scalarization step described in Section 6. The functions **class2optimizeCS** and **class2equilibriumLatentCS** take exactly the same parameters as **cmex2optimizeCS** and **cmex2equilibriumLatentCS** and the MATLAB® class generated by the former has exactly the same methods



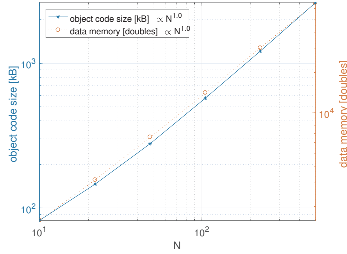
(a) Soft-margin SVM classifier (20 features, N training examples)



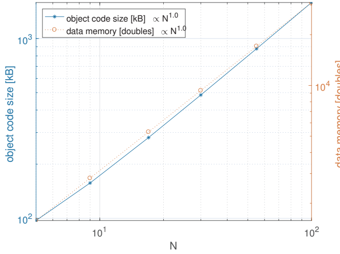
(b) Soft-margin SVM classifier (N features, 500 training examples)



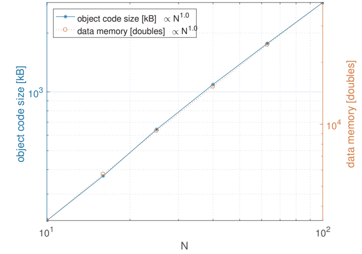
(c) Lasso



(d) Hougen-Watson model identification (N measurements)



(e) Distance-based localization (N measurements)



(f) MPC-MHE (N horizon length)

Figure 3: Data and code memory footprint

as the wrapper class generated by the later. While the MATLAB® solvers generated by **class2optimizeCS** and **class2equilibriumLatentCS** are generally much slower than the C-solvers generated by **cmex2optimizeCS** and **cmex2equilibriumLatentCS**, they are very useful for debugging convergence.

8 Examples

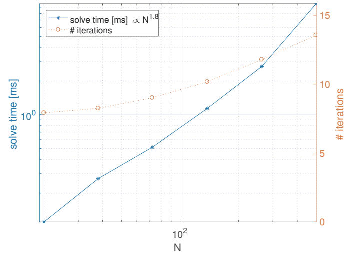
We demonstrate the functionality and performance of `TensCalc` through five optimization examples. For each example, we provide the mathematical formulation for the optimization and the `TensCalc` code needed to generate the solver (in the appendix). We then provide plots showing how the size of the object code, data memory (i.e., the scratchbook size), and solve time scale with the size of the problem. The solve times and number of iterations reported in Figures 4-6 correspond to averages obtained from solving 10000 random instances of the optimization problem.

The size of the object code and solve times depend on the compiler and optimization flags used. For consistency, all code was compiled on OSX using Apple LLVM version 8.0.0 with the −O1 optimization flag. The solver times were obtained in a MacBook Pro, Mid 2012 with a 2.6 GHz Intel Core i7, with 16GB, 1600 MHz DDR3 memory.

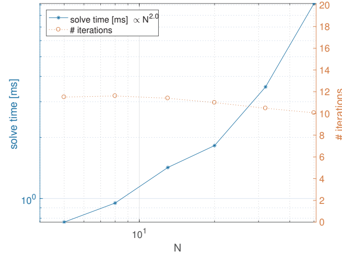
8.1 Soft-margin SVM classifier

Given M training pairs $(x_i, y_i) \in \mathbb{R}^K \times \{-1, +1\}$, $i \in \{1, 2, \ldots, M\}$, we need to solve

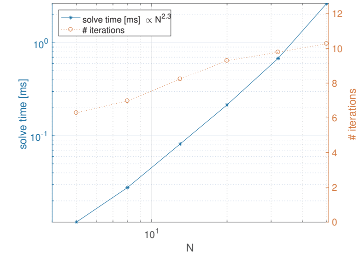
$$\text{minimize} \quad \frac{1}{M}\sum_{i=1}^{M} \zeta_i + \lambda\|\beta\|^2$$



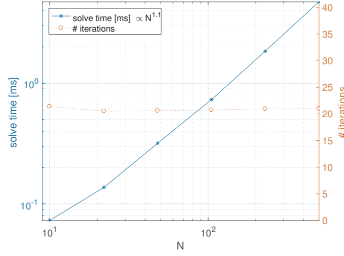
(a) Soft-margin SVM classifier (20 features, N training examples)



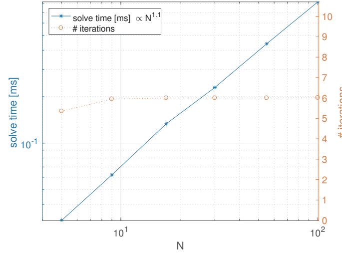
(b) Soft-margin SVM classifier (N features, 500 training examples)



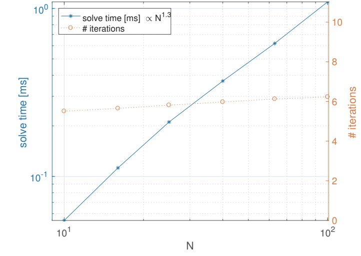
(c) Lasso



(d) Hougen-Watson model identification (N measurements)



(e) Distance-based localization (N measurements)



(f) MPC-MHE (N horizon length)

Figure 4: Solve time and number of iterations

$$
\begin{aligned}
\text{w.r.t.} \quad & \beta := \begin{bmatrix} \beta_1 & \beta_2 & \cdots & \beta_K \end{bmatrix} \in \mathbb{R}^K \\
\text{subject to} \quad & y_i(\langle \beta, x_i \rangle + b) \geq 1 - \zeta_i, \;\; \forall i \in \{1, 2, \ldots, M\}
\end{aligned}
$$

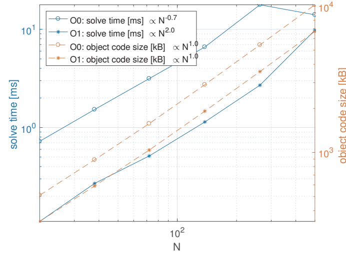
Figures 3–4 show the memory footprint and solution time for the solver as a function of the number M of training pairs and also as a function of the number K of features in each y_i. We can see that the code size scales linearly with M and with $K^{1.6}$, whereas the solver times scale with $M^{1.8}$ and $K^{2.0}$.

8.2 Lasso

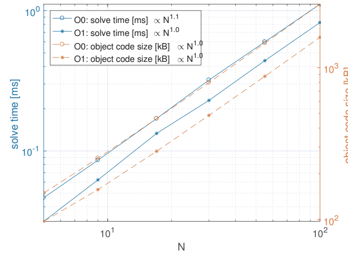
Given M training pairs $(x_i, y_i) \in \mathbb{R}^N \times \mathbb{R}$, $i \in \{1, 2, \ldots, M\}$, we need to solve

$$
\begin{aligned}
\text{minimize} \quad & \sum_{i=1}^{M} (y_i - \langle \beta, x_i \rangle)^2 \\
\text{w.r.t.} \quad & \beta := \begin{bmatrix} \beta_1 & \beta_2 & \cdots & \beta_N \end{bmatrix} \in \mathbb{R}^N \\
\text{subject to} \quad & \sum_{j=1}^{N} |\beta_j| \leq \lambda.
\end{aligned}
$$

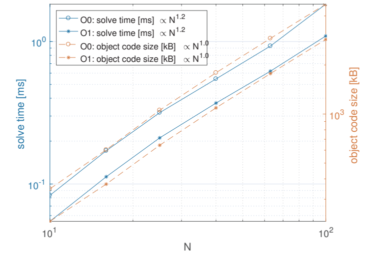
Figures 3–4 show the memory footprint and solution time for the solver as a function of the dimension N of the parameter vector β. The number M of training pairs is chosen to be 10 times larger than N. We can see that the code size scales roughly with $N^{2.8}$ and the solve time with $N^{2.3}$. Since we have $M = 10N$, the problem data scales with $NM \propto N^2$ so a scaling with at least N^2 is inevitable. It should be noted that for $N = 50$, $M = 500$, the solve time is still below 3 milliseconds.



(a) Soft-margin SVM classifier (20 features, N training examples)

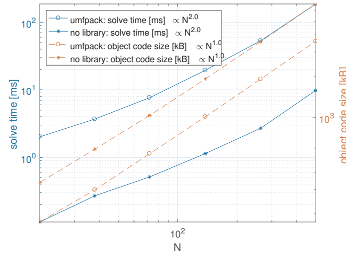


(b) Distance-based localization

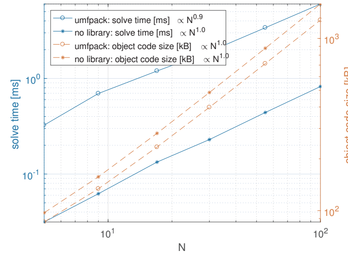


(c) MPC-MHE

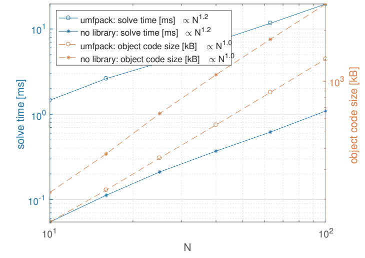
Figure 5: Solve time and code size for compiler optimization –O0 and –O1.



(a) Soft-margin SVM classifier (20 features, N training examples)



(b) Distance-based localization



(c) MPC-MHE

Figure 6: Solve time and code size using the UMFPACK library.

8.3 Hougen-Watson model identification

We want to estimate the parameters $\beta_1, \beta_2, \ldots, \beta_5 \geq 0$ of the Hougen-Watson model

$$y = \frac{\beta_1 b - \beta_5 c}{1 + \beta_2 a + \beta_3 b + \beta_4 c}$$

with input $x := \begin{bmatrix} a & b & c \end{bmatrix} \in \mathbb{R}^3$ and output $y \in \mathbb{R}$, given N pairs of noisy input-output measurements. Assuming that the the inputs measurements $\tilde{a}_i, \tilde{b}_i, \tilde{c}_i$ and the output measurements $\tilde{y}_i$ are all corrupted by zero-mean Gaussian independent distributed (iid) noise, the maximum likelihood estimate of the parameter β is given by

$$\begin{aligned} \text{minimize} \quad & (\tilde{a}_i - a_i)^2 + (\tilde{b}_i - b_i)^2 + (\tilde{c}_i - c_i)^2 + \sum_{i=1}^{N} \Big(\tilde{y}_i - \frac{\beta_1 b_i - \beta_5 c_i}{1 + \beta_2 a_i + \beta_3 b_i + \beta_4 c_i} \Big)^2 \\ \text{w.r.t.} \quad & \beta_j \in \mathbb{R}, \;\; j \in \{1, \ldots, 5\}, \\ & a_i, b_i, c_i \in \mathbb{R}, \;\; i \in \{1, 2, \ldots, M\} \\ \text{subject to} \quad & \beta_j^{\min} \leq \beta_j \leq \beta_j^{\max}, \;\; \forall j \in \{1, \ldots, 5\} \end{aligned}$$

where the a_i, b_i, c_i represents the actual inputs, y_i the actual output, $\tilde{a}_i, \tilde{b}_i, \tilde{c}_i$ the noisy input measurements, and $\tilde{y}_i$ the noisy output measurement. For simplicity, the formula above assumes that all input measurements are corrupted by noise with unit variance, whereas the output measurements are corrupted by noise with variance λ. It should be noted that this minimization problem is not convex.

Figures 3–4 show the memory footprint and solution time for the solver as a function of the number N of input-output measurements. We can see that the code and data memory size scales linearly with N and the solve time with $N^{1.1}$. For $N = 500$ data points we have 405 optimization variables and 100 equality constraints, but the solve time is still below 5 milliseconds.

8.4 Distance-based localization

Given noisy measurements of distances $d_i(t)$ at times $t \in \{1, 2, \ldots, N\}$ from a moving point P to M beacons at fixed positions $b_i \in \mathbb{R}^3$, $i \in \{1, 2, \ldots, M\}$, we want to reconstruct the point's positions $p(t) \in \mathbb{R}^3$, $t \in \{1, 2, \ldots, N\}$. The point's changes in velocity

$$a(t) := v(t+1) - v(t),\ \forall t \in \{1, \ldots, N-2\}, \qquad v(t) := p(t+1) - p(t),\ \forall t \in \{1, \ldots, N-1\}.$$

are assumed to be zero-mean Gaussian independent and identically distributed (iid). Assuming that the distance measurements $\tilde{d}_i(t)$ are corrupted by zero-mean Gaussian iid noise, the maximum likelihood estimate of the point's positions is given by

$$\begin{aligned} \text{minimize} \quad & \sum_{t=1}^{N} \sum_{i=1}^{M} \left(\|p(t) - b_i\| - \tilde{d}_i(t) \right)^2 + \lambda \sum_{t=1}^{N-2} \|a(t)\|^2 \\ \text{w.r.t.} \quad & p(t) \in \mathbb{R}^3, t \in \{1, \ldots, N\} \\ \text{subject to} \quad & p_{\min} \le p(t) \le p_{\min},\ \forall t \in \{1, \ldots, N\} \end{aligned}$$

where $p_{\min}, p_{\min} \in \mathbb{R}^3$ define a bounding box for the point's positions. Figures 3–4 show the memory footprint and solution time for the solver as a function of the number N of times, for $M = 5$ beacons. We can see that the code and data memory size scales linearly with N and the solve time with $N^{1.1}$.

8.5 MPC-MHE for linear quadratic problem with constraints

We want to control a linear system modeled by an ARX model

$$y_{k+1} = \alpha_0 y_k + \alpha_1 y_{k-1} + \cdots + \alpha_{n-1} y_{k-n+1} + \beta_0 u_k + \beta_1 u_{k-1} + \cdots + \beta_{n-1} u_{k-n+1} + d_k, \tag{29}$$

where the $u_k \in \mathbb{R}$ denote control inputs, the $d_k \in \mathbb{R}$ unmeasured disturbances, and the $y_k \in \mathbb{R}$ outputs that can be measured. Given L past noisy measurements $\tilde{y}_{k-L+1}, \tilde{y}_{k-L+2}, \ldots, \tilde{y}_k \in \mathbb{R}$ of the outputs $y_{k-L+1}, y_{k-L+2}, \ldots, y_k \in \mathbb{R}$ and L past control inputs, $u_{k-L+1}, u_{k-L+n+1}, \ldots, u_{k-1} \in \mathbb{R}$, our goal is to compute the future control inputs $u_k, u_{k+1}, \ldots, u_{K+T-1} \in \mathbb{R}$, for worst case measurement noise and disturbances with a criteria of the following form [1]:

$$\min_{\substack{u_k, \ldots, u_{k+T-1} \\ |u_\ell| \le u_{\max}}} \ \max_{\substack{d_{k-L+n}, \ldots, d_{k+T-1} \\ |d_\ell| \le d_{\max},\ |y_\ell - \tilde{y}_\ell| \le n_{\max}}} \ \sum_{\ell=k}^{k+T-1} y_{\ell+1}^2 + u_\ell^2 - \lambda_1 \sum_{\ell=k-L+1}^{k} (y_\ell - \tilde{y}_\ell)^2 - \lambda_2 \sum_{\ell=k-L+n}^{k+T-1} d_\ell^2.$$

When the minimum and maximum commute, this corresponds to a Nash-equilibrium with symmetric costs for the two players (zero-sum), which is an optimization of the form (2). As discussed in section (5.1), we use the actual outputs $y_{k-L+1}, y_{k-L+2}, \ldots, y_{k+T}$ as additional optimization variables subject to the equality constraints given by (29) to promote sparsity. Figures 3–4 show the memory footprint and solution time for the solver as a function of the horizon length $L = T = N$. We can see that the code size scales linearly with N and the solve time with $N^{1.3}$.

8.6 Compiler optimization

We can see in Figure 5 that the -O1 optimization flag roughly cuts the size of the object code and solver times by about one half, with respect to the -O0 flag (no optimization). An analysis of the assembly code indicates that the -O1 optimization reduces the code size and increases speed mostly by keeping in CPU registers scrapbook variables that will be used in another computation shortly after they are first computed (at most a few lines of C-code below). This saves code and time by not having to reload those variables from memory.

For `TensCalc`-generated code, the use of general purpose compiler optimization is an overkill. In fact, the computation graph generated by `TensCalc` could be used to judiciously select which variables to keep as CPU registers. In fact, in an earlier version of `TensCalc` we generated directly assembly code and obtained solver times and code sizes that were essentially the same as those obtained with the -O1 optimization flag. However, we abandoned this approach so that our code was not tied to a particular micro-processor.

8.7 UMFPACK

As mentioned above, the most expensive computation that needs to be scalarized is the solution of the system of equations in (11b). To avoid this, `TensCalc` provides the option to perform this operation using the UMFPACK library [2, 3], instead of the exhaustive scalarization described in Section 6. When the UMFPACK library is used, all operations are scalarized except for solution to (11b), which is performed as an atomic operation.

Figure 6 compares the results obtained with exhaustive versus partial scalarization (using the UMFPACK library). The use of UMFPACK results in smaller code, but at the expense of a significant increase in solve time: around one order of magnitude for the problems considered here. The solve-time penalty is due to the overhead involved in addressing the elements of the sparse matrix. It is not surprising to see similar solve-time scaling laws with the problem size because `TensCalc` uses essentially the same algorithms as UMFPACK to achieve the sparsification of the LDL factors. However, one should note that am important advantage of using UMFPACK is that pivoting need not be determined at code generation time and therefore can be optimized for numerical stability in run time.

9 Conclusions

We developed a toolbox that generates specialized C-code that can solve nonlinear optimizations and compute Nash equilibria. The solvers use a primal-dual interior point method and generate C-code that performs the required computations very efficiently; automatically exploring the sparsity structures that derive from the specific optimization being carried out and minimizing the amount of computation needed for each iteration of the algorithm.

An important feature of the toolbox is that it automatically performs all the symbolic manipulations needed to determine the first and second-order derivatives needed by each Newton step. Through this process, the toolbox automatically determines structural sparsity patterns and computations that can be re-used within and across iterations. Currently only basic symbolic simplifications are performed to reduce computation, such as discarding additions of zero and multiplications by zero or one. We believe that improving the symbolic engine could improve performance for many problems.

The code generation is based on the construction of a computation graph that encodes all the computational dependencies needed for a single iteration of the primal-dual interior point algorithm. This graph

is used to minimize recomputations, reduce the memory footprint, and schedule computations within a single thread. For multi-core processors, one should be able to use this graph to reduce computation time by distributing computation across the multiple cores. This is a topic for future research.

The current algorithm used to reduce the memory footprint starts by finding a topological sorting of the graph nodes and then reuses a memory location when it is no longer needed by subsequent computations. However, topological sorting is not unique and some node orderings are better than others at minimizing memory usage. We this provides significant opportunities to reduce memory usage that are not explored in the current version of `TensCalc`.

Appendix

This appendix contains the `TensCalc` code used to specify the optimization criteria and constraints for the examples discussed in Section 8. In the interest of saving space, we only include the calls to **cmex2optimizeCS** and **cmex2equilibriumLatentCS** in a couple of examples.

Lasso

```
37      % parameters
38      Tvariable X [M,N];
39      Tvariable y [M];
40      Tvariable lambda [];
41      % optimization variables
42      Tvariable beta [N];
43      Tvariable abs_beta [N];
44      % criteria
45      J = norm2(y-X*beta);
46      % constraints
47      constraints={ beta>=-abs_beta;
48                    beta<=abs_beta;
49                    sum(abs_beta,1)<=lambda; };
50      cmex2optimizeCS('classname','lasso_solver',...
51                      'objective',J,...
52                      'optimizationVariables',{beta,abs_beta},...
53                      'constraints',constraints,...
54                      'outputExpressions',{J,beta,abs_beta},...
55                      'parameters',{X,y,lambda});
```

Soft-margin SVM classifier

```
56      % parameters
57      Tvariable y [M];
58      Tvariable X [M,N];
59      Tvariable lambda [];
60      % optimization variables
61      Tvariable beta [N];
62      Tvariable b    [1];
63      Tvariable zeta [M];
64      % criteria
```

```
65      J = sum(zeta,1)/M+lambda*norm2(beta);
66      % constraints
67      constraints={ y.*(X*beta+b(ones(M,1))) >= Tones(M)-zeta;
68                    zeta>=0; };
```

Hougen-Watson model identification

```
69      % parameters (measured inputs & outputs)
70      Tvariable tilde_abc [N,3];
71      Tvariable tilde_y [N,1];
72      Tvariable lambda [];
73      Tvariable beta_bounds [5,2];
74      % optimization variables (noiseless inputs/outputs & model parameters)
75      Tvariable abc [N,3];
76      Tvariable y [N,1];
77      Tvariable beta [5,1];
78      % criteria (log-likelihood)
79      J = norm2(tilde_y-y)+lambda*norm2(tilde_abc-abc);
80      % constraints
81      constraints={
82          (Tones(N,1)+abc(:,1)*beta(2,1)...
83              +abc(:,2)*beta(3,1)+abc(:,3)*beta(4,1)).*y...
84                 ==abc(:,2)*beta(1,1)-abc(:,3)*beta(5,1);
85          beta>=beta_bounds(:,1);
86          beta<=beta_bounds(:,2); };
```

Distance-based localization

```
87      % parameters
88      Tvariable B [3,1,M];
89      Tvariable tilde_d [N,M];
90      Tvariable lambda [];
91      Tvariable pbox [3,2];
92      % optimization variables
93      Tvariable p [3,N];
94      % criteria
95      p1=reshape(p,[3,N,1]);
96      pB=p1(:,:,ones(1,M))-B(:,ones(1,N),:);  % vectors from beacons to point
97      d=sqrt(tprod(pB,[-1,1,2],pB,[-1,1,2])); % distances from beacons to point
98      v=p(:,2:end)-p(:,1:end-1);              % velocity
99      a=v(:,2:end)-v(:,1:end-1);              % acceleration
100     J=norm2(tilde_d-d)+lambda*norm2(a);     % log-likelihood
101     % constraints
102     constraints={ p>=pbox(:,ones(1,N));
103                   p<=pbox(:,2*ones(1,N)); };
```

MPC-MHE for linear quadratic problem with constraints

```
104     % parameters
105     Tvariable tilde_y  [L,1];
106     Tvariable u_past   [L-1,1];
107     Tvariable lambda_u [];
```

```
108     Tvariable lambda_n  [];
109     Tvariable lambda_d  [];
110     Tvariable alpha     [n,1];
111     Tvariable beta      [n,1];
112     Tvariable umax      [];
113     Tvariable dmax      [];
114     Tvariable nmax      [];
115     % optimization variables
116     Tvariable u_future  [T,1];
117     Tvariable d         [T+L-n,1];
118     Tvariable y0        [n,1];
119     Tvariable y1        [T+L-n,1];
120     y=[y0;y1];
121     % criteria
122     J=norm2(y(L+1:end,1))+lambda_u*norm2(u_future)...
123       -lambda_n*norm2(y(1:L,1)-tilde_y)-lambda_d*norm2(d);
124     % constraints
125     u=[u_past;u_future]; % [T+L-1]
126     yy=d;
127     for i=1:n
128         yy=yy+y(n-i+1:end-i,1)*alpha(i,1)+u(n-i+1:end-i+1,1)*beta(i,1);
129     end
130     % minimizer constraints
131     P1constraints={ -umax<=u_future; u_future<=umax; };
132     % maximizer constraints
133     P2constraints={ -dmax<=d; d<=dmax;
134                     -nmax<=y(1:L,1)-tilde_y; y(1:L,1)-tilde_y<=nmax};
135     % common constraints
136     Lconstraints={ y(n+1:end,1)==yy; };
137
138     [classname,code]=cmex2equilibriumLatentCS(...
139                         'classname','mpcmhe_solver',...
140                         'P1objective',J,...
141                         'P2objective',-J,...
142                         'P1optimizationVariables',{u_future},...
143                         'P2optimizationVariables',{d,y0},...
144                         'latentVariables',{y1},...
145                         'P1constraints',P1constraints,...
146                         'P2constraints',P2constraints,...
147                         'latentConstraints',Lconstraints,...
148                         'outputExpressions',{J,u_future,d,y(1:L,1)-tilde_y,y0,y1},...
149                         'parameters',{lambda_u,lambda_n,lambda_d,...
150                                       umax,dmax,nmax,...
151                                       alpha,beta,...
152                                       tilde_y,u_past});
```

References

[1] D. A. Copp and J. P. Hespanha. Simultaneous nonlinear model predictive control and state estimation. Submitted to journal publication., Mar. 2016. (cited in p. 9, 24)

[2] T. Davis. SuiteSparse: A suite of sparse matrix software. http://faculty.cse.tamu.edu/davis/suitesparse.html. (cited in p. 25)

[3] T. A. Davis. Algorithm 832: UMFPACK v4. 3—an unsymmetric-pattern multifrontal method. *ACM Transactions on Mathematical Software (TOMS)*, 30(2):196–199, 2004. (cited in p. 25)

[4] T. A. Davis, J. R. Gilbert, S. I. Larimore, and E. G. Ng. A column approximate minimum degree ordering algorithm. *ACM Trans. Math. Software*, 30(3):353–376, Sept. 2004. (cited in p. 4, 14)

[5] T. A. Davis, J. R. Gilbert, S. I. Larimore, and E. G. Ng. Algorithm 836: COLAMD, a column approximate minimum degree ordering algorithm. *ACM Trans. Math. Software*, 30(3):377–380, Sept. 2004. (cited in p. 14)

[6] J. Farquhar. *tprod – arbitrary tensor product between n-d arrays*, Sept. 2007. MathWorks File Exchange. (cited in p. 4)

[7] P. E. Gill, M. A. Saunders, and J. R. Shinnerl. On the stability of Cholesky factorization for symmetric quasidefinite systems. *SIAM J. Matrix Anal. and Appl.*, 17(1):35–46, 1996. (cited in p. 4, 9)

[8] G. H. Golub and C. F. V. Loan. *Matrix Computations*. Johns Hopkins Series in Mathematical Sciences. The Johns Hopkins University Press, Baltimore, 2nd edition, 1990. (cited in p. 4, 11, 14)

[9] M. Grant and S. Boyd. Graph implementations for nonsmooth convex programs. In V. Blondel, S. Boyd, and H. Kimura, editors, *Recent Advances in Learning and Control*, volume 371 of *Lecture Notes in Control and Information Sciences*, pages 95–110. Springer Berlin / Heidelberg, 2008. (cited in p. 9)

[10] M. Grant, S. Boyd, and Y. Ye. *CVX: Matlab Software for Disciplined Convex Programming*. Stanford University, Palo Alto, CA, June 2008. Available at http://cvxr.com/cvx/. (cited in p. 3)

[11] F. Gwinner. Transitive reduction of a DAG. Mathworks® File Exchange, Aug. 2011. (cited in p. 17)

[12] J. Lofberg. YALMIP: A toolbox for modeling and optimization in MATLAB. In *Computer Aided Control Systems Design, 2004 IEEE International Symposium on*, pages 284–289. IEEE, 2004. (cited in p. 3)

[13] J. Mattingley and S. Boyd. CVXGEN: a code generator for embedded convex optimization. *Optimization and Engineering*, pages 1–27, 2012. (cited in p. 3, 8)

[14] K. A. McShane, C. L. Monma, and D. S. Rutcor. An implementation of a primal-dual interior point method for linear programming. *ORSA Journal on Computing*, 1(2):70, 1989. (cited in p. 3)

[15] S. Mehrotra. On the implementation of a primal-dual interior point method. *SIAM J. on Optimization*, 2(4):575–601, 1992. (cited in p. 3)

[16] Y. E. Nesterov and M. J. Todd. Self-scaled barriers and interior-point methods for convex programming. *Mathematics of Operations research*, 22(1):1–42, 1997. (cited in p. 3)

[17] Y. E. Nesterov and M. J. Todd. Primal-dual interior-point methods for self-scaled cones. *SIAM J. on Optimization*, 8(2):324–364, 1998. (cited in p. 3)

[18] L. Vandenberghe. The CVXOPT linear and quadratic cone program solvers. Technical report, Univ. California, Los Angeles, 2010. URL http://cvxopt.org/documentation. (cited in p. 7, 8)

[19] R. J. Vanderbei. Symmetric quasidefinite matrices. *SIAM J. on Optimization*, 5(1):100–113, 1995. (cited in p. 4, 9)