Noname manuscript No.
(will be inserted by the editor)

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

João P. Hespanha

January 4, 2022

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 equilibria). `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 microprocessor 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 scale with the size of the problem.

Keywords Optimization, linear algebra, symbolic differentiation, code generation, interior-point methods

MSC classification 90C30 - Nonlinear programming, 49M15 - Newton-type methods

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

J. Hespanha
University of California, Santa Barbara
orcid.org/0000-0003-2809-4718
E-mail: hespanha@ece.ucsb.edu

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 datasets and very significant time savings are possible by building solvers that have been constructed for a 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 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 at the code-generation time to minimize the solve time.

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

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

and to compute two-player Nash equilibria defined by

$$\begin{aligned} f_u(u^*,d^*,p) = \min\big\{f_u(u,d^*,p) :& F_u(u,d^*,p) \geq 0, \\ & G_u(u,d^*,p) = 0,\ u \in \mathbb{R}^{n_u}\big\}, \qquad (2a) \\ g_d(u^*,d^*,p) = \min\big\{g_d(u^*,d,p) :& F_d(u^*,d,p) \geq 0, \\ & G_d(u^*,d,p) = 0,\ d \in \mathbb{R}^{n_d}\big\}. \qquad (2b) \end{aligned}$$

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-player problem (2). The functions

$$\begin{aligned} f &: \mathbb{R}^{n_u} \times \mathbb{R}^{n_p} \to \mathbb{R}, && (3a) \\ f_u &: \mathbb{R}^{n_u} \times \mathbb{R}^{n_d} \times \mathbb{R}^{n_p} \to \mathbb{R}, && (3b) \\ g_d &: \mathbb{R}^{n_u} \times \mathbb{R}^{n_d} \times \mathbb{R}^{n_p} \to \mathbb{R}, && (3c) \end{aligned}$$

encode the *optimization criteria*,

$$\begin{aligned} G &: \mathbb{R}^{n_u} \times \mathbb{R}^{n_p} \to \mathbb{R}^{n_G}, && (4a) \\ G_u &: \mathbb{R}^{n_u} \times \mathbb{R}^{n_d} \times \mathbb{R}^{n_p} \to \mathbb{R}^{n_{Gu}}, && (4b) \\ G_d &: \mathbb{R}^{n_u} \times \mathbb{R}^{n_d} \times \mathbb{R}^{n_p} \to \mathbb{R}^{n_{Gd}} && (4c) \end{aligned}$$

encode *equality constraints*, and

$$\begin{aligned} F &: \mathbb{R}^{n_u} \times \mathbb{R}^{n_p} \to \mathbb{R}^{n_F}, && (5a) \\ F_u &: \mathbb{R}^{n_u} \times \mathbb{R}^{n_d} \times \mathbb{R}^{n_p} \to \mathbb{R}^{n_{Fu}}, && (5b) \\ F_d &: \mathbb{R}^{n_u} \times \mathbb{R}^{n_d} \times \mathbb{R}^{n_p} \to \mathbb{R}^{n_{Fd}} && (5c) \end{aligned}$$

encode (element-wise) *inequality constraints*.

The toolbox's name `TensCalc` results from the merger 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 allows these functions to take as inputs and outputs any number of real-valued tensors 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` generates primal-dual interior point solvers for 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 symbolically by `TensCalc`.
3. The sparsity structures of all the gradient and Hessian matrices used 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 into 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 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 in run time, at each iteration of the primal-dual algorithm and from one optimization to the next. 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 microprocessor instruction pipelining.

From a user perspective, the fundamental distinguishing feature of `TensCalc` is the ability to solve medium-size nonlinear optimizations (up to a few thousand of optimization variables and constraints) much faster than what could be done with general purpose solvers.

From a technical perspective, this speed up is mostly a consequence of items 3 and 4 above. The idea of incorporating matrix sparsity structures into the code of

optimization solvers was motivated by CVXGEN [19], which generates C code to solve quadratic programs with linear constraints. One of the key contributions of our work is demonstrating that the process of discovering and exploiting sparsity structures can be extended to general nonlinear programs in a computationally effective fashion. Encoding computational dependencies in a graph that can be used both to avoid redundant computations and minimize memory footprint is directly inspired by ideas from compiler optimization. However, `TensCalc` constructs the graph at a higher level of abstraction, based on the primal-dual interior point algorithm to generate C code, rather than based on C code to generate assembly code. This enables global optimizations over hundreds of thousands of operations and variables, far beyond what can be effectively accomplished by current compilers, at a fraction of the computational cost. Essentially, we take advantage of the fact that a large collection of linear algebra computations has much more structure than a general fragment of C code (all variables are of the same type, no indirect references to variables, no calls to sub-routines, etc.).

The `TensCalc` toolbox is freely available at `https://github.com/hespanha/tenscalc` under the GNU General Public License v3.0.

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 [15], YALMIP [18], and CasADi [1] are the noncommercial optimization modeling languages most closely related to `TensCalc`. All three 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 and CasADi permit 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 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. In terms of functionality `TensCalc` is very similar to CasADi and, in fact, it is generally very straightforward to translate an optimization expressed in `TensCalc` to CasADi's Opti stack interface. The key difference with respect to CVX and YALMIP is that `TensCalc` generates standalone C code that does not require external optimization solvers or external libraries. While CasADi enables the generation of C code to evaluate functions and their derivatives, it relies on external optimization solvers to actually perform the optimization.

A key feature of `TensCalc` is that the code generated is highly optimized and directly integrated with the optimization solver. In practice, this generally enables the solution of one instance of an optimization much faster than what could be achieved with general purpose solvers like Ipopt [30,3], SeDuMi [25], SDPT3 [26], or Gurobi [16]. In this respect, `TensCalc` is much closer to CVXGEN [19] , 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 [20, 21] 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 [22, 23]. Nevertheless, our experience has been that it is still extremely effective for many nonlinear and nonconvex optimizations. The specific algorithms used by `TensCalc` are heavily inspired by [28] and are also very close to the ones used by Ipopt [30,3]. However, we have opted to not include in `TensCalc` any form of automatic scaling. While this adds extra burden to the user, we did not want to introduce computational penalties that are not always needed or could be resolved by the user offline. In practice, this means that `TensCalc` may require a few extra iterations of the solver, but we shall see that the computation savings enabled by specialized code greatly outweigh this shortcoming.

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 [12]. However, when the matrix $H \in \mathbb{R}^{N\times N}$ that defines the system of equations is sparse, the number of operations needed to solve these equations can scale much better than $O(N^3)$. `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 [8]. For the minimization problem (1), the matrix H is symmetric and "almost" quasi-definite and therefore admits an LDL factorization for every symmetric permutation [28]. Moreover, these factorizations are generally numerically stable [11]. 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 variables 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. This was achieved by overloading the builtin functions and operators, including `subsref`, `reshape`, `vertcat`, `horzcat`, `cat`, `uplus`, `plus`, `+`, `uminus`, `minus`, `-`, `sum`, `exp`, `log`, `sqrt`, `cos`, `sin`, `tan`, `atan`, `round`, `ceil`, `floor`, `abs`, `relu`, `heaviside`, `times`, `*`, `mtimes`, `.*`, `rdivide`, `./`, `inv`, `trace`, `det`, `lu`, `ldl`. The following logic-valued functions and operators were also overloaded 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 [9].

`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` MATLAB class called `Tcalculus`, which overloads the standard MATLAB functions and operators listed above so that they can be applied to `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)
```

[1] While MATLAB regards scalars and vectors still as matrices with a single column and/or row, this is not the case for `TensCalc`.

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 its entry $(i_1, i_2, \ldots, i_K, j_1, j_2, \ldots, j_L)$ is given by

$$\frac{\partial \texttt{expr}_{i_1,i_2,\ldots,i_K}}{\partial \texttt{var}_{j_1,j_2,\ldots,j_L}}$$

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

The toolbox introduces two functions cmex2optimizeCS and cmex2equilibriumLatentCS that take as inputs TensCalc symbolic expressions and generate C code to solve the optimizations (1) and (2), respectively. A typical call to the function cmex2optimizeCS is of the following form:

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

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 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. 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 optimization variables into (single-dimension) vectors and stack these vectors appropriately to form 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.

The code-generation engine of `TensCalc` can also be used to perform repeated computations very efficiently, beyond the specific optimizations (1)–(2). This is accomplished by specifying a set of computations using the `TensCalc` class `csparse` and subsequently generating code using the command `cmex2compute`. While a detailed description of `csparse` and `cmex2compute` is outside the scope of this paper, it worth noting that `cmex2compute` takes advantages of all the optimizations described in Section 6.

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 the parameter vector 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 minimum) *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*

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

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

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

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

$$f(u) \leq \epsilon_f + \min\left\{ f(\bar{u}) : F(\bar{u}) \geq 0, G(\bar{u}) = 0, \bar{u} \in \mathbb{R}^{n_u} \right\}, \qquad \epsilon_f := \lambda \cdot F(u). \tag{7}$$

□

The following notation is used in (6)–(7) and below: Given an integer n, we denote by $\mathbf{0}_n$ and by $\mathbf{1}_n$ the n-vectors with all entries equal to 0 and 1, respectively. 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; and by $x \cdot y \in \mathbb{R}$, $x \odot y \in \mathbb{R}^n$, and $x \oslash y \in \mathbb{R}^n$ the inner product, entry-wise product, and entry-wise division of the two vectors, respectively.

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 $\lambda \cdot F(u) = 0$ may result in a suboptimal value for u, but the level of suboptimality is no larger than $\epsilon_f := \lambda \cdot F(u)$. It is convenient that this does not require strong duality.

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)
$$\nabla_u L_f(u, \lambda, \nu) = \mathbf{0}_{n_u}, \tag{8}$$
where $\nabla_u L_f$ denotes the gradient of L_f with respect to the variable u;
2. the equality constraint (6b); and
3. the equation
$$F(u) \odot \lambda = \mu \mathbf{1}_{n_F}, \tag{9}$$
for some $\mu > 0$, which lead to
$$\epsilon_f := \lambda \cdot F(u) = \mu \, n_F.$$

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 progress. This is done in the context of an interior-point method, meaning that all variables will be initialized so that the inequality constraints (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 [27].

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*

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

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*

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

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*

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

which leads to

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

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

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

$$\alpha \coloneqq \min\{\alpha_{\text{primal}}, \alpha_{\text{dual}}\},$$

and

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

$$\alpha_{\text{dual}} \coloneqq \max\Big\{\alpha \in [0,1] : \lambda_k + \frac{\alpha}{.99}\Delta \lambda \geq 0\Big\}. \tag{14}$$

Step 4. Update μ_k according to

$$\mu_{k+1} = \begin{cases} \gamma_{\text{aggressive}}\, \mu_k & \text{if } \alpha \geq .5,\ \|G(u_{k+1})\|_\infty \leq 100\epsilon_G, \\ & \qquad \|\nabla_u L_f(u_{k+1}, \lambda_{k+1}, \nu_{k+1})\|_\infty \leq 100\epsilon \\ \gamma_{\text{conservative}}\, \mu_k & \text{otherwise} \end{cases} \tag{15}$$

with $0 < \gamma_{\text{aggressive}} < \gamma_{\text{conservative}} < 1$. Typically, we use $\gamma_{\text{aggressive}} = 1/3$ and $\gamma_{\text{conservative}} = .75$, 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 constraints 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 Step 2 with an incremented value for k until

$$\|G(u_k)\|_\infty \leq \epsilon_G, \qquad \|\nabla_u L_f(u_k, \lambda_k, \nu_k)\|_\infty \leq \epsilon, \qquad \lambda_k \cdot F(u_k) \leq \epsilon_{\text{gap}}. \tag{16}$$

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 Hessian $\nabla_{uu} L_f$; 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 through (optional) input parameters to the function `cmex2optimizeCS`.

Remark 1 (Algorithm improvements) One can find in [27] 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 [27]. 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 [19], consists of replacing (11b) by

$$\begin{bmatrix} \nabla_{uu} L_f(u_k, \lambda_k, \nu_k) + \nabla_u F(u_k)' \operatorname{diag}[\lambda_k \oslash F(u_k)] \nabla_u F(u_k) + \delta I & \nabla_u G(u_k)' \\ \nabla_u G(u_k) & -\delta I \end{bmatrix}$$
$$\times \begin{bmatrix} \Delta u \\ \Delta \nu \end{bmatrix} = -\begin{bmatrix} \nabla_u f(u_k) + \nabla_u G(u_k)' \nu_k - \mu_k \nabla_u F(u_k)' \big(\mathbf{1}_{n_F} \oslash F(u_k)\big) \\ G(u_k) \end{bmatrix} \tag{17}$$

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 sub-matrix positive definite and the bottom-right one negative definite. It turns out that quasi-definite matrices are strongly factorizable, i.e., they admit an LDL factorization for every symmetric permutation, which may not be the case for the original matrix with $\delta = 0$ [28]. Moreover, their LDL factorization is generally numerically stable [11]. 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 result 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 [28] 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

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

which are equivalent to the following smooth optimizations

$$\begin{aligned} &\min \left\{ \mathbf{1}_m \cdot v : x \in \mathbb{R}^n, v \in \mathbb{R}^m, -v \leq Ax - b \leq v \right\}, \\ &\min \left\{ v : x \in \mathbb{R}^n, v \in \mathbb{R}, -v\mathbf{1}_m \leq Ax - b \leq v\mathbf{1}_m \right\}, \end{aligned}$$

respectively. Additional examples of such criteria and the corresponding transformations can be found, e.g., in [14]. □

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 the 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 [4].

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*

$$L_f(u,d,\lambda_{fu},\nu_{fu}) = \min_{\bar u \in \mathbb{R}^{n_u}} L_f(\bar u,d,\lambda_{fu},\nu_{fu}), \tag{18a}$$
$$L_g(u,d,\lambda_{gd},\nu_{gd}) = \min_{\bar d \in \mathbb{R}^{n_d}} L_g(u,\bar d,\lambda_{gd},\nu_{gd}), \tag{18b}$$
$$G_u(u,d) = 0, \qquad G_d(u,d) = 0, \tag{18c}$$
$$F_u(u,d) \geq \mathbf{0}_{n_{F_u}},\ \lambda_{fu} \geq \mathbf{0}_{n_{F_u}}, \qquad F_d(u,d) \geq \mathbf{0}_{n_{F_d}},\ \lambda_{gd} \geq \mathbf{0}_{n_{F_d}}, \tag{18d}$$

where

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

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

$$f(u,d) \leq \epsilon_f + \min\{f_u(\bar u,d) : F_u(\bar u,d) \geq 0, G_u(\bar u,d) = 0, \bar u \in \mathbb{R}^{n_u}\}, \tag{19a}$$
$$g(u,d) \leq \epsilon_g + \min\{g_d(u,\bar d) : F_d(u,\bar d) \geq 0, G_d(u,\bar d) = 0, \bar d \in \mathbb{R}^{n_d}\}. \tag{19b}$$

with

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

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 (18a)–(18b):
$$\nabla_u L_f(u,d,\lambda_{fu},\nu_{fu}) = \mathbf{0}_{n_u}, \qquad \nabla_d L_g(u,d,\lambda_{gd},\nu_{gd}) = \mathbf{0}_{n_d}; \tag{20}$$
2. the equality constraints (18c); and
3. the equations
$$F_u(u,d) \odot \lambda_{fu} = \mu \mathbf{1}_{n_{F_u}}, \qquad F_d(u,d) \odot \lambda_{gd} = \mu \mathbf{1}_{n_{F_d}}, \tag{21}$$
for some $\mu > 0$, which lead to
$$\epsilon_f := \lambda_{fu} \cdot F_u(u,d) = \mu\, n_{F_u}, \qquad \epsilon_g := \lambda_{gd} \cdot F_d(u,d) = \mu\, n_{F_d}.$$

Since our goal is to find primal variables u, d for which (19a) holds with $\epsilon_f = \epsilon_g = 0$, we shall make the variable μ converge to zero as the Newton iterations progress. Defining

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

we can re-write (20), (18c), and (21) as

$$\nabla_u L_f(z,\lambda,\nu) = \mathbf{0}_{n_u}, \qquad \nabla_d L_g(z,\lambda,\nu) = \mathbf{0}_{n_d}, \tag{22a}$$
$$G(z) = \mathbf{0}_{K_u+K_d}, \qquad \lambda \odot F(z) = \mu \mathbf{1}_{M_u+M_d}, \tag{22b}$$

and (18d) as

$$\lambda \geq \mathbf{0}_{M_u+M_d}, \qquad F(z) \geq \mathbf{0}_{M_u+M_d}. \tag{23}$$

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 (22) around a current estimate z_k, λ_k, ν_k, leading to

$$\begin{bmatrix} \nabla_{uz} L_f(z_k,\lambda_k,\nu_k) & \nabla_{u\nu} L_f(z_k) & \nabla_{u\lambda} L_f(z_k) \\ \nabla_{dz} L_g(z_k,\lambda_k,\nu_k) & \nabla_{d\nu} L_g(z_k) & \nabla_{d\lambda} L_g(z_k) \\ \nabla_z G(z_k) & 0 & 0 \\ \nabla_z F(z_k) & 0 & \operatorname{diag}[F(z_k) \oslash \lambda_k] \end{bmatrix} \begin{bmatrix} \Delta z \\ \Delta \nu \\ \Delta \lambda \end{bmatrix}$$
$$= - \begin{bmatrix} \nabla_u L_f(z_k,\lambda_k,\nu_k) \\ \nabla_d L_g(z_k,\lambda_k,\nu_k) \\ G(z_k) \\ F(z_k) - \mu \mathbf{1} \oslash \lambda_k \end{bmatrix}. \tag{24}$$

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

$$\Delta\lambda = -\lambda_k - \operatorname{diag}[\lambda_k \oslash F(z_k)] \nabla_z F(z_k) \Delta z + \mu \mathbf{1} \oslash F(z_k) \tag{25a}$$

which leads to

$$\begin{bmatrix} \nabla_{uz} L_f(z_k,\lambda_k,\nu_k) - \nabla_{u\lambda} L_f(z_k) \operatorname{diag}[\lambda_k \oslash F(z_k)] \nabla_z F(z_k) & \nabla_{u\nu} L_f(z_k) \\ \nabla_{dz} L_g(z_k,\lambda_k,\nu_k) - \nabla_{d\lambda} L_g(z_k) \operatorname{diag}[\lambda_k \oslash F(z_k)] \nabla_z F(z_k) & \nabla_{d\nu} L_g(z_k) \\ \nabla_z G(z_k) & 0 \end{bmatrix}$$
$$\times \begin{bmatrix} \Delta z \\ \Delta \nu \end{bmatrix} = - \begin{bmatrix} \nabla_u f(z_k) + \nabla_u \big((\nu_{fu})_k G_u(z_k) \big) + \mu \nabla_{u\lambda} L_f(z_k) \big(\mathbf{1} \oslash F(z_k) \big) \\ \nabla_d g(z_k) + \nabla_d \big((\nu_{gd})_k G_d(z_k) \big) + \mu \nabla_{d\lambda} L_g(z_k) \big(\mathbf{1} \oslash F(z_k) \big) \\ G(z_k) \end{bmatrix} \tag{25b}$$

A notable difference between the previous system of equations in (11b) and the one in (25b) 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 on the code generation for the Algorithm 1 that solves (1), with the understanding that the same discussion applies to the algorithm used by `TensCalc` to solve (2), with (11b) replaced by (25b).

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 (25b) used to compute the Newton search directions. Using Gauss elimination to solve these equations can require up to $O(N^3)$ floating-point operations [12], 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 (25b) 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 (25b), 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 $L_f(u, \lambda, \nu) := f(u) - \lambda \cdot F(u) + \nu \cdot G(u)$ with respect to pairs of variables in u, λ, and ν are structurally zero. For example, any second-order partial derivative of $L_f(u, \lambda, \nu)$ 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 introduce latent variables and equality constraints to obtain larger but more sparse matrices in (11b) and (25b). 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 constraints 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:

$$J = \sum_{k=1}^{N} x_k^2 + u_k^2, \tag{26a}$$

where

$$x_1 = 10, \qquad x_{k+1} = x_k + u_k, \quad \forall k \in \{1, 2, \dots, N-1\}, \tag{26b}$$

subject to the constraints that

$$|u_k| \le 1, \quad \forall k \in \{1, 2, \dots, N\}. \tag{26c}$$

One option to solve this problem consists of using (26b) 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 (26a) 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:

$$f(u) := \sum_{k=1}^{N} \Big(10 + \sum_{j=1}^{k-1} u_j\Big)^2 + u_k^2, \qquad F(u) := \begin{bmatrix} 1-u \\ 1+u \end{bmatrix} \ge 0. \tag{27}$$

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 (26b) as equality constraints. This would still lead to a problem of the form (1), but with

$$f(u,x) := \sum_{k=1}^{N} x_k^2 + u_k^2, \qquad F(u,x) := \begin{bmatrix} 1-u \\ 1+u \end{bmatrix} \ge 0, \tag{28a}$$

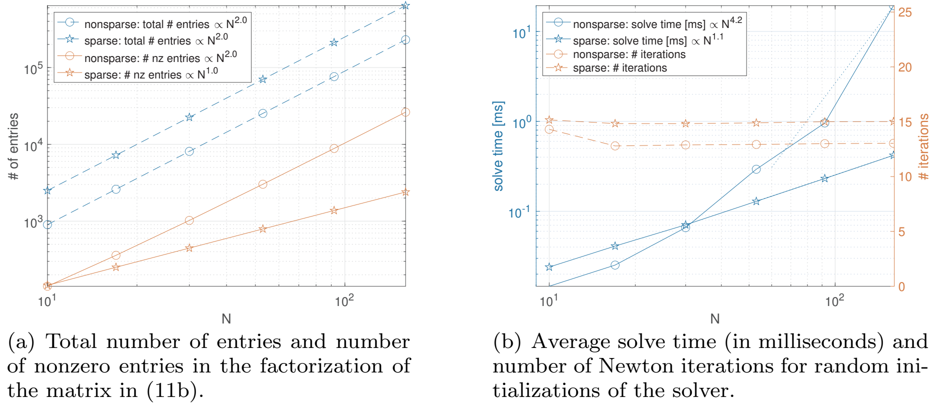
$$G(u,x) = \begin{bmatrix} x_1 - 10 \\ x_2 - (x_1 + u_1) \\ \vdots \\ x_N - (x_{N-1} + u_{N-1}) \end{bmatrix} = 0, \tag{28b}$$

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

The matrix in (11b) for (27) is $N \times N$ but most of its entries are nonzero because most second-order derivatives of $f(u)$ in (27) are nonzero. In contrast, the same matrix for (28) is $3N \times 3N$ and therefore has 9 times the number of entries. However, most second order derivatives of $f(u,x)$ in (28) 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 (28) actually has a much smaller number of (structurally) nonzero entries than the $N \times N$ matrix corresponding to (27). This ultimately leads to a solver for (28) that requires less memory and is faster. We can see in Figure 1 that the formulation (27) leads to a number of nonzero entries of the matrix in (11b) that scales with N^2, which eventually leads to solve times that scale roughly with $N^{4.2}$. In contrast, the formulation (28) leads to a number of nonzero entries of the matrix in (11b) that scales linearly with N and solve times that also scale roughly with N.

Since the Hessian matrix $\nabla_{uu} L_f(u, \lambda, \nu)$ associated with (27) is essentially full, one should expect the Gauss elimination involved in computing the Newton direction to scale no worse than N^3. However, we see in Figure 1 solve times for

(27) that scale roughly with $N^{4.2}$. This is because, even though the number of floating-point operations scales no worse than N^3, as the code size increases we see additional speed degradation due to a larger code that does not fit into the CPU cache.



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

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

Fig. 1 Comparison of the solvers associated with the two alternative formulations (27) and (28) of the same original problem (26), for different values of the horizon length N. In the legends, "nonsparse" refers to the formulations (27) and "sparse" refers to the formulation in (28). See Section 8 for details on the computer, operating system, and compiler used.

5.2 Exploiting Sparsity

The sparsity of the matrices and vectors that appear in (11b) and (25b) is mostly explored in three operations:

1. Additions and 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 (25b) are solved by performing an LU factorization of the matrices in the left-hand side, which takes advantage of the entries that are structurally zero to greatly increase efficiency. For the particular case of (11b), the matrix to factorize is symmetric and therefore we can perform an LDL factorization, which generally has lower memory and computation costs [12].

To make use of the sparsity structure of the left-hand side matrices in (11b) and (25b), 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 [8,7]. 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 known, but precise values of the nonzero entries are generally not known. 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.

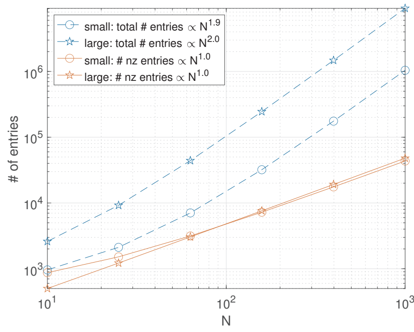
While the use of a random matrix works for a large number of problems, in some cases it may generate row-permutations that result in numerically unstable pivoting; especially in the computation of Nash equilibria, where the matrix in (25b) cannot be made quasi-definite. To overcome this, we also permit 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. Since these permutations essentially define the pivots used by Gauss elimination, this can cause numerical issues if there is no fixed choice of pivots that performs satisfactory across all iterations of the optimization algorithm. However, our experience has been that this is not a significant issue for most problems; especially when we replace (11b) by (17), which improves the stability of the LDL factorization, as discussed in Remark 1.

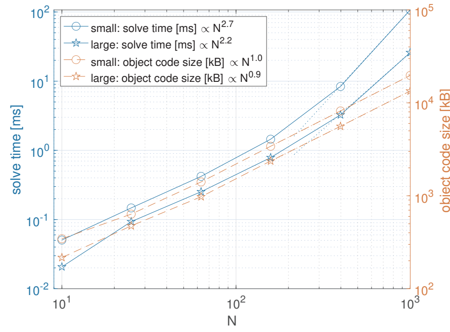
Remark 4 (Small vs. large Newton-step matrix) In Step 2 of Algorithm 1, one can compute the search direction by solving either (10) or (11). The latter may seem preferable because it involves a smaller system of equations, but this is not always the case. When $\nabla_u F(u_k)$ has one or more rows that are not sparse, the product

$$\nabla_u F(u_k)' \operatorname{diag}[\lambda_k \oslash F(u_k)] \nabla_u F(u_k) \tag{29}$$

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 is generally preferable to work with (10) directly. Even when (29) is sparse, it may still be preferable to let the COLAMD or SYMAMD determine the best order to eliminate variables in (10), rather than first eliminating $\Delta\lambda$, which is what was done to obtain (11). This is shown in Figure 2 for the Soft-margin SVM classifier that we shall encounter in Section 8.1. We can see that while (11) requires the factorization of a smaller matrix, (10) actually leads to smaller code and shorter solve times. This is the case for essentially all the examples in Section 8.1, which use (10) rather than (11). However, `TensCalc` permits the user to select between these two options using the parameter `smallerNewtonMatrix`. □



(a) Total number of entries and number of nonzero entries in the factorization of the matrices used to compute the Newton direction.



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

Fig. 2 Comparison of the solvers for the Soft-margin SVM classifier described in Section 8.1 obtained by solving (10) (label "large") or (11) (label "small"). See Section 8 for details on the computer, operating system, and compiler used.

6 Scalarization and Computation Graph

The first step towards generating the C code needed to implement Algorithm 1 consists of "scalarizing" all computations needed to execute the full algorithm and organizing all these computations in the form of a single large *computation graph*. By "scalarized", we mean that we break each vector- and 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

$$u_{k+1} = u_k + \alpha \Delta u \ \in \mathbb{R}^{n_u} \tag{30}$$

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 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 encodes the dependencies between primitive operations into the *computation graph*, which 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 (30) 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 mean 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 Algorithm 1, from the variable initialization in Step 1 to the checking the exit conditions in Step 5.

The edges of the computation graph $G = (\mathcal{N}, \mathcal{E})$ 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 optimization parameters (see Section 3), to initialize μ_0, to initialize the primal variable u_0, and to 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 interacts with the computation graph through *get events*, 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 set events that overwrite these values in the nodes corresponding to u_k, ν_k, λ_k, in preparation for the following iteration of the algorithm.

The following pseudo-code shows the structure of the code that will be generated by TensCalc to execute Algorithm 1, in terms of the set and get events discussed above:

```
10 function solve()
11    % Step 1 of Algorithm 1
12    set_event(parameter_nodes, parameters_values);
13    set_event(mu_node, mu0_initial_value);
14    set_event(primal_variables_nodes,
15              primal_variables_initial_values)
16    set_event(dual_variables_nodes,
17              dual_variables_initial_values)
18    repeat {
19      % Steps 2-3 of Algorithm 1
20      (u,nu,lambda) = get_event(next_primal_variables_nodes,
21                                next_dual_variables_nodes)
22      set_event(primal_variables_nodes, u)
23      set_event(dual_variables_nodes, (nu,lambda))
24      % Step 4 of Algorithm 1
25      set_event(mu_node, {equation 15});
26      % Step 5 of Algorithm 1
27      stop = get_event(termination_condition_node)
28    } until (stop)
29    return get_event(output_expression_nodes)
30 end
```

where the function `set_event(nodes,values)` assigns numerical values to specific nodes of the computational graph and the function `values=get_event(nodes)` re-

trieves the values associated with the given set of nodes. All the actual computations are embedded within the `get_event` functions and are carried out "as-needed" based on the structure of the computation graph, as discussed in the following sections.

In the pseudo-code above, `parameter_nodes` refers to the nodes of the computation graph that hold the values of the optimization parameters declared in line 7 of the call to `cmex2optimizeCS` in Section 3; `mu_node` refers to the node that holds the value of μ_k in (12); `primal_variables_nodes` refers to the nodes that hold the value of the primal variable u_k; `dual_variables_nodes` refers to the nodes that hold the value of the dual variables ν_k, λ_k; `next_primal_variables_nodes` refers to the nodes that hold the value of the primal variable u_{k+1} in (12); `next_dual_variables_nodes` refers 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 (16), and `output_expression_nodes` refers to the nodes that hold the values of the output expressions declared in line 9 of the call to `cmex2optimizeCS` in Section 3.

The computation graph is heavily used in the process of generating code for the solver: it is needed to determine the correct scheduling of computations to make sure that the calls to `get_event` return the correct values and it also enables several forms of optimization that significantly decrease the run-time of the solver, as discussed in the remainder of this section.

6.1 Avoiding redundant computations

We can see in Algorithm 1 that several 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 of Step 5. Moreover, the specific structure of the function f that defines the optimization criteria and of the constraint functions F and G typically hide additional computation redundancies: Suppose for example that

$$F(u) = \frac{1}{2}u'Pu, \qquad G(u) = \frac{1}{2}u'Qu.$$

In this case, the results obtained in the computation of the gradient $\nabla_u F(u_k) = u_k'P$ that is needed for the matrix in the left-hand side of (10) can be reused to compute $F(u_k) = \frac{1}{2}u_k'Pu_k = \frac{1}{2}\nabla_u F(u_k)u_k$ in the right-hand side of (10). Regarding the equality constraint, we can see in (15) that $G(u_{k+1}) = \frac{1}{2}(u_{k+1}'Q)u_{k+1}$ is computed at the end of iteration k and we will need $\nabla G(u_{k+1}) = u_{k+1}'Q$ at the next iteration. In this case, we can reuse the intermediate result $u_{k+1}'Q$ computed at the end of iteration k to obtain $\nabla G(u_{k+1})$ at iteration $k+1$, without any further computation.

Computational redundancies like the ones noted 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 operator) 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 graph. For example, the concatenation of the matrix

$$\begin{bmatrix} \nabla_{uu} L_f(u_k, \lambda_k, \nu_k) + \nabla_u F(u_k)' \operatorname{diag}[\lambda_k \oslash F(u_k)] \nabla_u F(u_k) & \nabla_u G(u_k)' \\ \nabla_u G(u_k) & 0 \end{bmatrix} \tag{31}$$

in (11b) from its 4 constituent blocks or the partition of the vector $\left[\begin{smallmatrix} \Delta u \\ \Delta \nu \end{smallmatrix}\right]$ into the two distinct variables Δu and $\Delta \nu$ do not require new nodes in the computation graph, 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 (31) and the computations in (11a), (12) directly use the nodes associated with the entries of the vector $\left[\begin{smallmatrix} \Delta u \\ \Delta \nu \end{smallmatrix}\right]$ 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 graph and, in practice, are “free” in the sense that they do not translate into any C code or memory allocation 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

$$(i, j) \in \mathcal{E} \quad \Rightarrow \quad o_i < o_j. \tag{32}$$

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 [17], 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 $\nabla_{uu} L_f$, $\nabla_{u\nu} L_f$, $\nabla_{u\lambda} L_f$ 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 can 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. At run time, 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 b_j associated with 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:

```
31  function set_event(nodes,values)
32     for i in nodes
33         s_i = values[i]
34     for j in descendants_of(nodes)
35         b_j = true
36  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:

```
37  function get_event(nodes)
38     for i in topological(ancestors_of(nodes))
39         if b_i == true
40            { ... compute node s_i ... }
41            b_i = false
42     return [s_i for i in nodes]
43  end
```

where `ancestors_of(nodes)` returns a set contain all nodes in `nodes`, their parents, their parents' parents, and so on; and `topological(s)` sorts the set of nodes `x` according to the topological order o_i in (32).

In practice, the use of one Boolean variable b_i for each node $i \in \mathcal{N}$ would lead to a prohibitively expensive run-time 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 `set_event(nodes,values)` required by Algorithm 1, the nodes i and j either both belong to `descendants_of(nodes)` or none does; and
2. for every get event `get_event(nodes)` required by Algorithm 1, the nodes i and j either both belong to `ancestors_of(nodes)` or none does.

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

$$\{\mathcal{G}_1, \mathcal{G}_2, \dots, \mathcal{G}_L\}, \qquad \mathcal{N} = \bigcup_{k=1}^{L} \mathcal{G}_k, \quad \mathcal{G}_k \cap \mathcal{G}_\ell = \emptyset, \ \forall k \neq \ell, \tag{33}$$

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 `set_event(nodes,values)` and `get_event(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 of each set and get event the precise list of variables b_k that need to be tested and set, leading to very little run-time overhead.

The algorithm described above has a significant impact on the total time it takes to solve an optimization, by avoiding redundant computations both within a single iteration and across the multiple iterations of Algorithm 1. In addition, it typically also saves significant computations from one optimization to the next. This effect can be seen in Figure 3, where we plot the times is takes to solve each iteration of Algorithm 1 for multiple instances of the optimization in (28). We can see that the first iteration of the first optimization requires about 6 times more computation than subsequent iterations, which is explained by the fact that there are very large structures within (10) and (24) that remain unchanged across different instances of the same optimization. Effects of this magnitude can be seen in essentially every one of the examples that we discuss in Section 8 and not just for quadratic programs like (28).

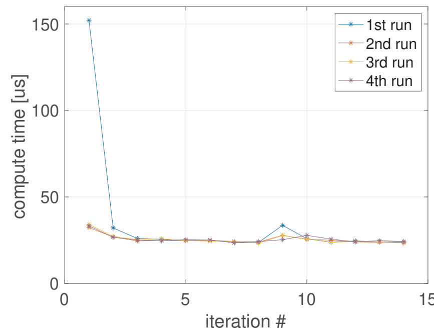


Fig. 3 Computation times for each iteration of Algorithm 1 for the optimization in (28) with $N = 100$. The plot shows times for 4 optimizations solved consecutively with different initializations. The first time the optimization is executed, the first iteration takes about 150us, whereas subsequent iterations take only about 25us. In subsequent optimizations, the first iteration only takes about 35us, as all computations that do not depend on the parameters that changed can be re-used. The small increase in computation time that we see around iteration 9 or 10 corresponds to the iteration at which μ_k starts to change through the top branch in (15). When μ_k does not change, a larger number of computations can be re-used from one iteration to the next.

6.4 Minimizing memory footprint

The computation graph $G = (\mathcal{N}, \mathcal{E})$ and the dependency groups in (33) are also used to determine the run-time memory required by the computations involved in Algorithm 1, avoiding 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, based on ideas from compiler liveness analysis [2]. Essentially, 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 `get_event(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 (32).

The first item guarantees that we do not discard a value s_j that needs to be returned by a `get_event(nodes)` call; 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 that needs to be generated to solve Algorithm 1 consists of the following collection of C functions:

1. The `solve()` function outlined in Section 6, which contains the main loop of Algorithm 1.
2. The `set_event(nodes,values)` and `get_event(nodes)` functions that appear in the pseudo-code of Algorithm 1 and are used to set and retrieve the values associated with the nodes of the computation graph.
3. One function `compute_group(k)` for each dependency group $\mathcal{G}_k$, which updates the values s_i associated with all the nodes $i \in \mathcal{G}_k$. These functions are called in line 40 of the `get_event(nodes)` pseudo-code to compute the values of the nodes in $\mathcal{G}_k$.

The bulk of the computation time is spent in the computations of the update for the primal and dual variables in line 20 of the `solve()` pseudo-code; in particular, in the functions `compute_group(k)` within `get_event(nodes)`, which 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 the scalarization process used to compute the computation graph, can be 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 microprocessor 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.

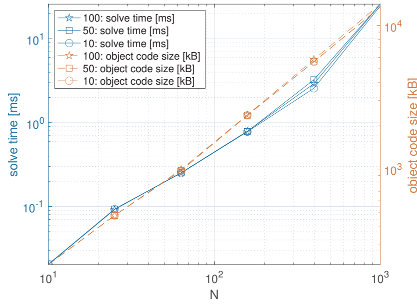
The assignment of computations to memory locations is performed at code generation time and maximizes memory reuse, as discussed in Section 6.4. This permits storing all computations in a single array, which we call the *scratchbook*, that is allocated when the code starts to execute, either as a global 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. As discussed in Section 6.3, the effect of reusing scratchbook variables can be clearly seen in Figure 3.

Remark 5 (Loop rolling/unrolling) To minimize the size of the code, `TensCalc` looks for groups of similar scalar instructions that read/write memory locations of the scratchbook following a fixed pattern. The user-defined parameter `minInstructions4loop`, provides a threshold above which such sets of instructions are replaced by loops, in a process that can be viewed as the reverse of loop unrolling. This can reduce the size of the code with minimal penalty in terms of solve times. In fact, for very large problems this can result in faster solve times, as it permits a larger portion of the solver's code to remain in the microprocessor's memory cache. However, one must keep in mind that compiler optimization can "undo" `TensCalc`'s loop rolling.

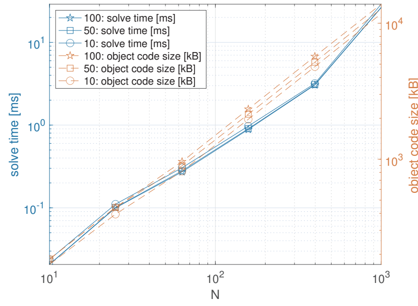
Figure 4 shows the effect of `minInstructions4loop` in the soft-margin SVM classifier and the MPC-MHE control problem that we shall encounter in Section 8.1. We can see that with the compiler optimization focused mostly on execution speed (`-Ofast`), varying `minInstructions4loop` form 10 to 100 has almost no effect on the size of the object code or the solve time, as the compiler optimization seems to override `TensCalc`'s attempt to unroll loops. However, with the compiler optimization focused on code size (`-Os`), smaller values for `minInstructions4loop` do result in smaller code, with a relatively small impact on execution speed. The optimal value for `minInstructions4loop` depends on the problem size and also on the processor's memory cache, but we found that `minInstructions4loop=50` provides consistently good results, which is the default value used by the toolbox. □

7.1 MATLAB interface to the solver

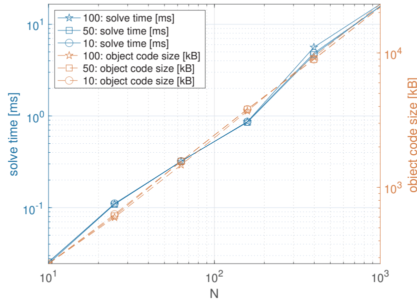
While `TensCalc` generates C code that can run independently from MATLAB, this toolbox also generates wrapper code to call the solver from within the MATLAB



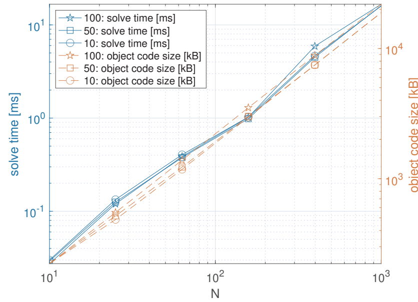
(a) Soft-margin SVM classifier ($K = 20$ features, N training examples) compiled with `-Ofast`



(b) Soft-margin SVM classifier ($K = 20$ features, N training examples) compiled with `-Os`



(c) MPC-MHE (N horizon length) compiled with `-Ofast`



(d) MPC-MHE (N horizon length) compiled with `-Os`

Fig. 4 Solve time and code size, as a function of the `minInstructions4loop` parameter, for C-code compilation with optimization `-Os` and `-Ofast`.

environment. The solver code is compiled into a library that is dynamically linked to MATLAB and functions callable from within MATLAB are created to

1. call the `set_event()` functions that are needed to set the optimization parameters declared in line 7 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. call the `get_event()` functions used to get the output expressions declared in line 9 of the call to `cmex2optimizeCS`.

`TensCalc` also generates a MATLAB class to provide an object-oriented interface to the solver.

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 gen-

erated by the former has exactly the same methods as the wrapper class generated by the later. While the MATLAB solvers generated by `class2optimizeCS` and `class2equilibriumLatentCS` are generally much slower than the C-code solvers generated by `cmex2optimizeCS` and `cmex2equilibriumLatentCS`, they can be useful for debugging numerical issues.

Internally, `TensCalc` uses the class `csparse` to store all the set events, get events, and computation graph associated with Algorithm 1. The commands `cmex2optimizeCS` and `cmex2equilibriumLatentCS` start by constructing an appropriate instance of this class and then generate C code to perform the corresponding computations. `TensCalc` provides direct access to the class `csparse` and a command `cmex2compute` that can be used to generate the corresponding C code. The code generated by this function benefits from all the optimizations described in Section 6 and also includes a wrapper MATLAB class that provides an object-oriented interface, with methods to call the access the different set and get events.

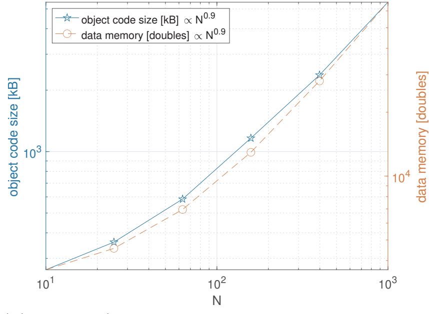
8 Examples

We demonstrate the functionality and performance of `TensCalc` through six 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, ranging from small problems with just a few 10s of optimization variables/constraints to large problems with 1000s of variables/constraints.

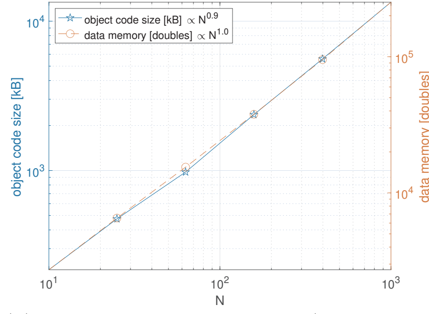
The solve times and number of iterations reported in Figures 6-10 correspond to averages obtained from solving a large number of random instances of the optimization problem (typically 10,000). For the examples corresponding to convex optimizations (presented in Sections 8.1, 8.2, and 8.5) there were no solver "failures" in the sense that the solver always terminated with the exit conditions (16) on the equality constraints, norm of the gradient, and duality gap satisfied. For the remaining (nonconvex) problems, there were no solver failures for distance-based localization (Section 8.4), Hougen-Watson model identification (Section 8.3) reached the maximum number of iterations 1.8% of the times, and the MPC-MHE min-max computation (Section 8.6) reached the maximum number of iterations less than .8% of the times. The maximum number of iterations was set to 100 for all problems.

The complexity scaling exponents that appear in the legends of Figures 1, 2, 5, and 6 were obtained through fitting using the right-most few points in the figures (corresponding to the largest values of N) and the fitting is shown as a dotted line in the plots. These scalings are empirical and should not be taken as true measures of memory/computational complexity.

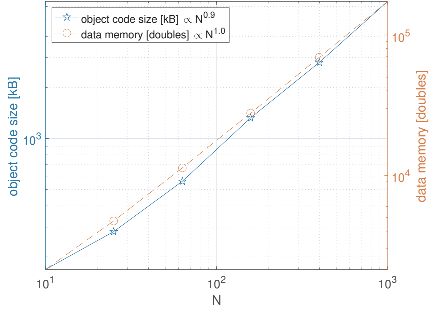
The size of the object code and solve times depend on the compiler and optimization flags used. For consistency, all code was compiled on OSX 10.15.7 with the clang compiler version Apple 12.0.0, with the `-Ofast` optimization flag (unless noted otherwise). The solver times were obtained in a Late 2013 iMac with an 3.5 GHz Quad-Core Intel Core i7, 32GB memory, 256KB L2 Cache, 8MB L3 Cache. All `TensCalc` solvers run on a single thread.



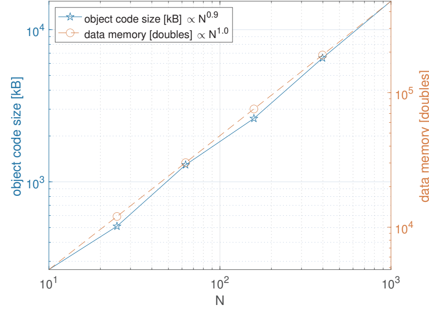
(a) Lasso ($K = 20$ features, N training examples)



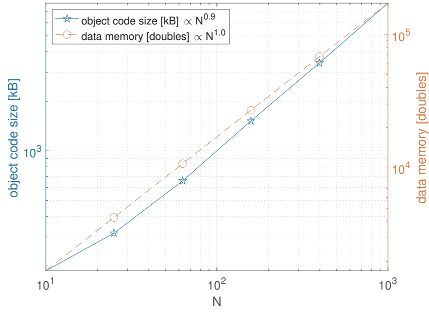
(b) Soft-margin SVM classifier ($K = 20$ features, N training examples)



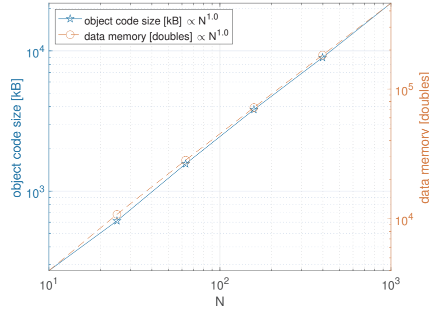
(c) Hougen-Watson model identification (N measurements)



(d) Distance-based localization (N measurements)



(e) MPC (N horizon length)



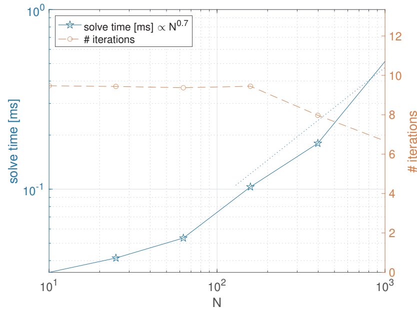
(f) MPC-MHE (N horizon lengths)

Fig. 5 Data and code memory footprint

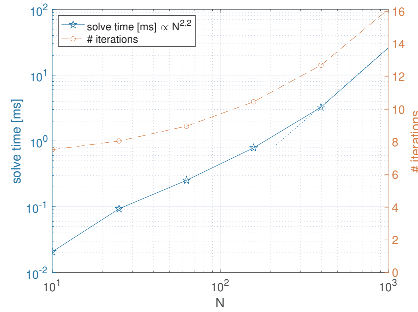
8.1 Lasso

Given N training pairs $(x_i, y_i) \in \mathbb{R}^K \times \mathbb{R}$, $i \in \{1, 2, \dots, N\}$, we solve

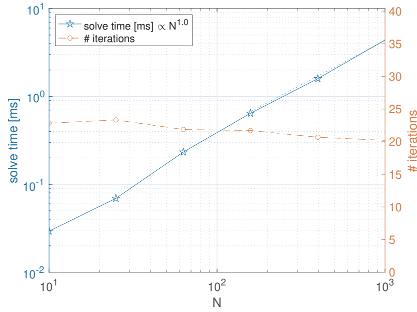
$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^{N} (y_i - \beta \cdot x_i)^2 \\ \text{w.r.t.} \quad & \beta := \begin{bmatrix} \beta_1 & \beta_2 & \cdots & \beta_K \end{bmatrix} \in \mathbb{R}^K \\ \text{subject to} \quad & \sum_{j=1}^{K} |\beta_j| \leq \lambda. \end{aligned}$$



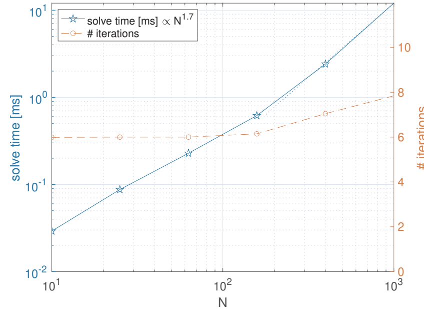
(a) Lasso ($K = 20$ features, N training examples)



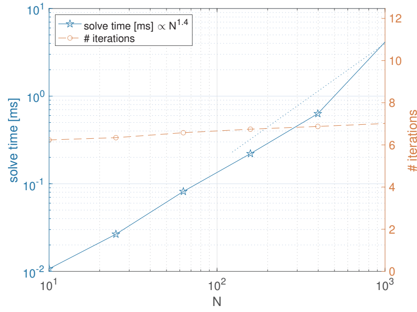
(b) Soft-margin SVM classifier ($K = 20$ features, N training examples)



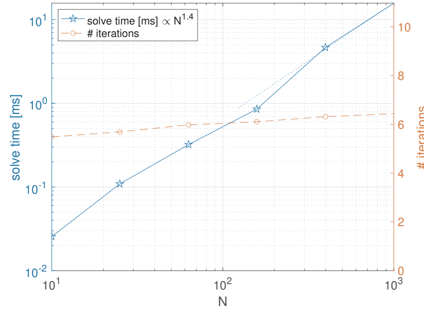
(c) Hougen-Watson model identification (N measurements)



(d) Distance-based localization (N measurements)



(e) MPC (N horizon length)



(f) MPC-MHE (N horizon lengths)

Fig. 6 Solve time and number of iterations

In this problem, the total number of primal and dual variables scales linearly with the dimension K of the feature vectors x_i and therefore the total number of entries in the Newton-step matrix is $O(K^2)$. While the Newton-step matrix has a large number of zero entries, it still has a dense component with $O(K^2)$ nonzero entries. Figures 5–6 show the memory footprint and solution time for the solver as a function of the number N of training pairs, for $K = 20$. We can see that the code size and solver times both scale roughly linearly with N.

8.2 Soft-margin SVM classifier

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

$$\begin{aligned} \text{minimize} \quad & \frac{1}{N}\sum_{i=1}^{N} \zeta_i + \lambda \|\beta\|^2 \\ \text{w.r.t.} \quad & \beta := \begin{bmatrix}\beta_1 \ \beta_2 \ \cdots \ \beta_K\end{bmatrix} \in \mathbb{R}^K \\ \text{subject to} \quad & y_i(\beta \cdot x_i + b) \geq 1 - \zeta_i, \ \zeta_i \geq 0, \ \forall i \in \{1, 2, \ldots, N\} \end{aligned}$$

In this problem, the total number of primal and dual variables scales linearly with the number N of training pairs and therefore the total number of entries in the Newton-step matrix is $O(N^2)$. However, the number of nonzero entries only scales with $O(N)$. Figures 5–6 show the memory footprint and solution time for the solver as a function of N, for feature vectors x_i with $K = 20$ dimensions. We can see that the code size scales linearly with N, whereas the solver times scale roughly with $N^{2.2}$.

8.3 Hougen-Watson model identification

We 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 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 and identically distributed (iid) noise, the maximum likelihood estimate of the parameter β is given by

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

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. This minimization problem is not convex.

In this problem, the total number of primal and dual variables scales linearly with the number N of input-output measurements and therefore the total number of entries in the Newton-step matrix is $O(N^2)$. However, the number of nonzero

entries only scales with $O(N)$. Figures 5–6 show the memory footprint and solution time for the solver as a function of N. We can see that the code and data memory size scales linearly with N, as well as the solve times.

8.4 Distance-based localization

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

$$\begin{aligned} a(t) &:= v(t+1) - v(t), \ \forall t \in \{1, \dots, N-2\}, \\ v(t) &:= p(t+1) - p(t), \ \forall t \in \{1, \dots, N-1\}. \end{aligned}$$

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

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

where $p_{\min}, p_{\min} \in \mathbb{R}^3$ define a bounding box for the point's positions. This minimization problem is not convex.

In this problem, the total number of primal and dual variables scales linearly with the number N of time instants and therefore the total number of entries in the Newton-step matrix is $O(N^2)$. However, the number of nonzero entries only scales with $O(N)$. Figures 5–6 show the memory footprint and solution time for the solver as a function of N for $M = 5$ beacons. We can see that the code and data memory size scales linearly with N and the solve time with $N^{1.7}$.

8.5 MPC for linear quadratic problem with constraints

We control a linear system modeled by an ARXmodel

$$\begin{aligned} y_{k+1} = \alpha_0 y_k + \alpha_1 y_{k-1} + \cdots + \alpha_{n-1} y_{k-n+1} \\ + \beta_0 u_k + \beta_1 u_{k-1} + \cdots + \beta_{n-1} u_{k-n+1}, \end{aligned} \tag{34}$$

where the $u_k \in \mathbb{R}$ denote control inputs and the $y_k \in \mathbb{R}$ measured outputs. Given n past outputs $y_{k-n+1}, \dots, y_k \in \mathbb{R}$ and $n-1$ past control inputs, $u_{k-n+1}, \dots, u_{k-1} \in \mathbb{R}$, our goal is to compute the future control inputs $u_k, \dots, u_{K+T-1} \in \mathbb{R}$, for a criteria of the form

$$\min_{\substack{u_k, \dots, u_{k+T-1} \\ |u_\ell| \le u_{\max}}} \sum_{\ell=k}^{k+T-1} y_{\ell+1}^2 + u_\ell^2.$$

As discussed in Section 5.1, we use the future outputs $y_{k+1}, \dots, y_{k+T}$ as additional optimization variables subject to the equality constraints given by (34) to promote sparsity.

In this problem, the total number of primal and dual variables scales linearly with the horizon length T and therefore the total number of entries in the Newton-step matrix is $O(T^2)$. However, the number of nonzero entries of this matrix only scales with $O(T)$. Figures 5–6 show the memory footprint and solution time for the solver as a function of $N := T$. We can see that the code size scales linearly with N and the solve time with $N^{1.2}$. This scaling is consistent with the observations made in Section 5.1 and results from the fact that the number of nonzero entries of the matrix in (10) scales linearly with N.

8.6 MPC-MHE for linear quadratic problem with constraints

We control a linear system modeled by an ARXmodel

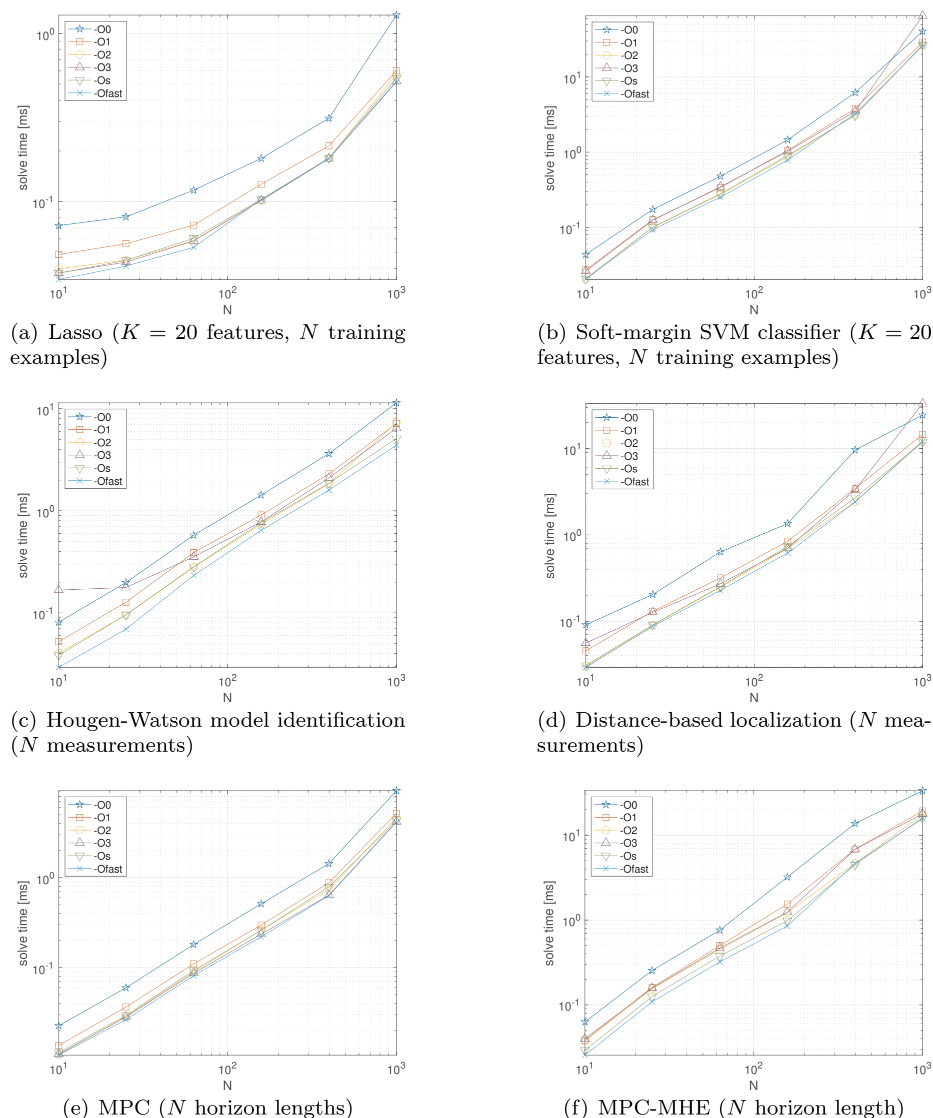
$$\begin{aligned} y_{k+1} = \alpha_0 y_k + \alpha_1 y_{k-1} + \cdots + \alpha_{n-1} y_{k-n+1} \\ + \beta_0 u_k + \beta_1 u_{k-1} + \cdots + \beta_{n-1} u_{k-n+1} + d_k, \end{aligned} \quad (35)$$

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 for which we only have noisy measurements. Given L past noisy measurements $\tilde{y}_{k-L+1}, \dots, \tilde{y}_k \in \mathbb{R}$ of the actual outputs $y_{k-L+1}, \dots, y_k \in \mathbb{R}$ and $L-1$ past control inputs, $u_{k-L+1}, \dots, u_{k-1} \in \mathbb{R}$, our goal is to compute the future control inputs $u_k, \dots, u_{K+T-1} \in \mathbb{R}$, for worst case initial outputs $y_{k-L+1}, \dots, y_{k-L+n}$, disturbances $d_{k-L+n}, \dots, d_{k+T-1}$, and measurement noise $\tilde{y}_{k-L+1} - y_{k-L+1}, \dots, \tilde{y}_k - y_k$, for a criteria of the form:

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

[4]. 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 $y_{k-L+n+1}, \dots, y_{k+T}$ as additional optimization variables subject to the equality constraints given by (35) to promote sparsity.

In this problem, the total number of primal and dual variables scales linearly with the horizon length $T+L$ and therefore the total number of entries in the Newton-step matrix is $O\big((T+L)^2\big)$. However, by using future outputs as additional optimization variables subject, the number of nonzero entries only scales with $O(T+L)$. Figures 5–6 show the memory footprint and solution time for the solver as a function of $N := L = T$. We can see that the code size scales linearly with N and the solve time with $N^{1.3}$.



(a) Lasso ($K = 20$ features, N training examples)

(b) Soft-margin SVM classifier ($K = 20$ features, N training examples)

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

(d) Distance-based localization (N measurements)

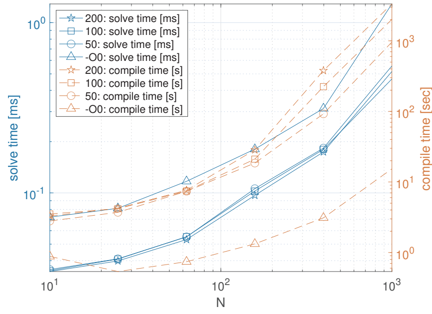
(e) MPC (N horizon lengths)

(f) MPC-MHE (N horizon length)

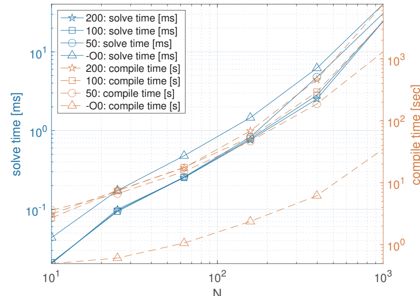
Fig. 7 Solve time for different levels of compiler optimization.

8.7 Compiler optimization

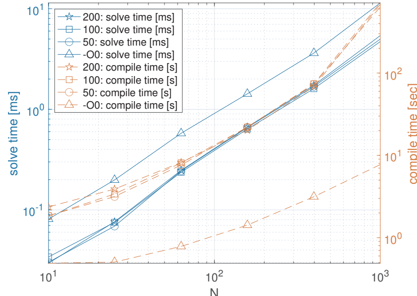
We can see in Figure 7 that the `-O1` optimization flag roughly cuts the size of the object code and solver times to about one half, with respect to the `-O0` flag (no optimization). By disassembling the code generated with `-O0` and `-O1`, we can see that the key difference is that the memory address of the “scratchbook” array (see Section 7) is kept in a CPU register with `-O1` optimization. Since this address never changes, this variable only needs to be read once when the solver starts. In



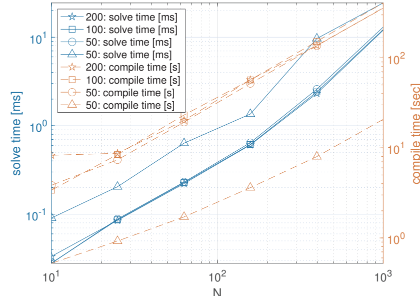
(a) Lasso ($K = 20$ features, N training examples)



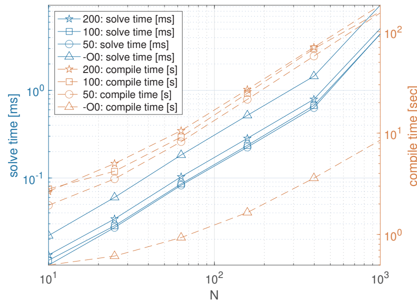
(b) Soft-margin SVM classifier ($K = 20$ features, N training examples)



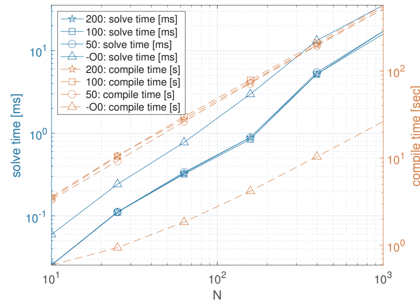
(c) Hougen-Watson model identification (N measurements)



(d) Distance-based localization (N measurements)



(e) MPC (N horizon length)



(f) MPC-MHE (N horizon length)

Fig. 8 Solver and compile times as a function of the `maxInstructionsPerFunction` parameter for C-code compiled with optimization `-Ofast`. For comparison, we also include in the plots the solver and compile times without compiler optimization (`-O0`).

contrast, with `-O0` this variable is read for every single computation (often multiple times). This explains the large solve-time savings observed with `-O1`. In addition, `-O1` also keeps in CPU registers scratchbook variables that are reused shortly after they are first computed, which saves code and time by not having to reload those variables from memory. More aggressive optimization settings like `-O2`, `-O3`, and `-Ofast` result in additional reductions of the solve time.

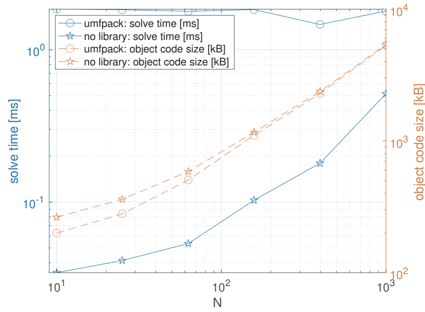
We have observed that the register allocation algorithms can become very slow when the code inside a function becomes very large. To overcome this issue, the current version of `TensCalc` can limit the size of code inside a single function by breaking functions into pieces. While this restricts the ability of a compiler to optimize register allocation, the performance penalty is generally minimal and, for some compilers, the resulting compile times may be significantly reduced. The parameter `maxInstructionsPerFunction` permits the user to select the maximum number of instructions that will be included in a single function. Figure 8 shows how the solve time and clang's compile time vary, as a function of `maxInstructionsPerFunction`. For the large versions of the lasso and soft-margin SVM problems we see a significant reduction in the compilation times as we decrease `maxInstructionsPerFunction`, but this difference is not as noticeable in the other problems. By and large, we see very small changes in solve times as we vary `maxInstructionsPerFunction`. Aside from this figure, all results in this paper used the default value of `maxInstructionsPerFunction=100`.

In spite of the reduction in compile time introduced by `maxInstructionsPerFunction`, we can see in Figure 8 that compiler optimization through `-Ofast` still introduces a significant computational burden, which reflects the fact that compilers are not optimized for code as long as that generated by `TensCalc` for large problems. For `TensCalc`-generated code, the use of general purpose compiler optimization may be an overkill as the computation graph generated by `TensCalc` could be used, e.g., to judiciously select which variables to keep as CPU registers. In fact, in an earlier version of `TensCalc` we directly generated 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 microprocessor.

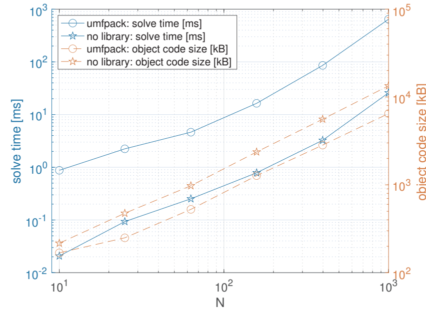
8.8 UMFPACK

As mentioned before, the most expensive computation that needs to be scalarized is the solution of the system of equations in (11b). `TensCalc` provides the option to perform this operation using the UMFPACK library [6,5], instead of the exhaustive scalarization described in Section 6. When the UMFPACK library is used, all operations are scalarized except for solving (11b), which is performed by UMFPACK functions as an atomic operation.

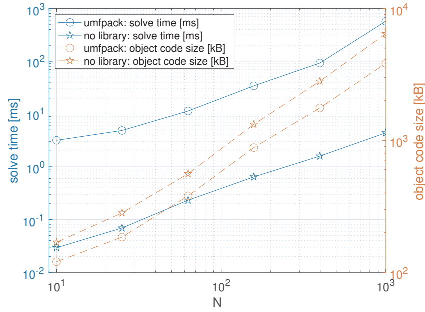
Figure 9 compares the results obtained with the exhaustive scalarization of all operations versus a partial scalarization that used the UMFPACK library for matrix factorizations. The use of UMFPACK results in smaller code, but at the expense of a significant increase in solve time: around one order of 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 an important advantage of using UMFPACK is that pivoting need not be determined at code generation time and therefore can be optimized for numerical stability at run time.



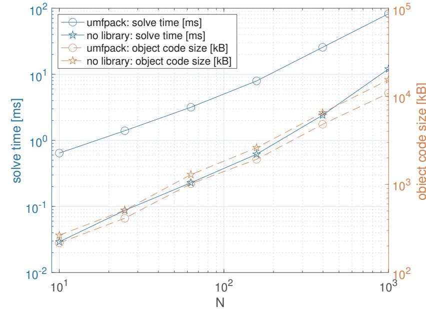
(a) Lasso ($K = 20$ features, N training examples)



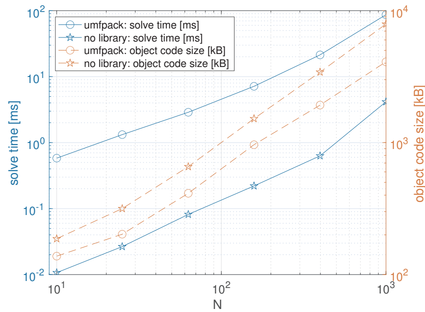
(b) Soft-margin SVM classifier ($K = 20$ features, N training examples)



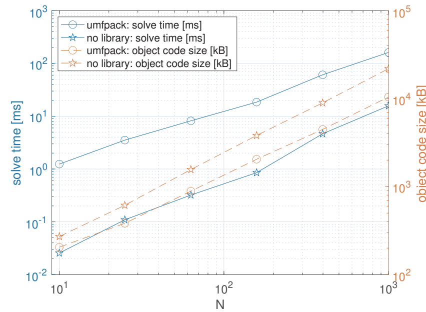
(c) Hougen-Watson model identification (N measurements)



(d) Distance-based localization (N measurements)



(e) MPC (N horizon lengths)

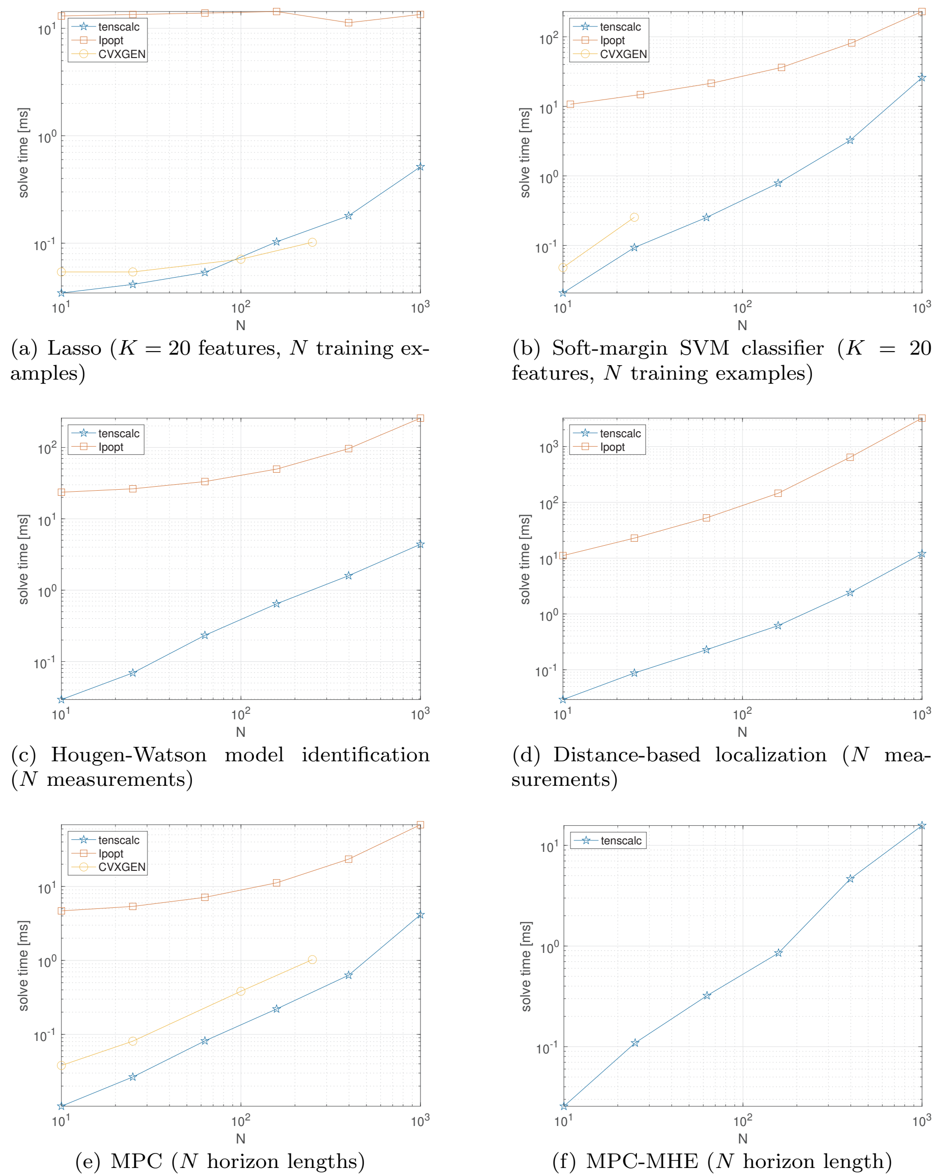


(f) MPC-MHE (N horizon length)

Fig. 9 Solve time and code size with and without using the UMFPACK library.

8.9 Ipopt and CVGEN

As noted in Section 2, the nonlinear programming algorithm used by `TensCalc` is very similar to the one used by Ipopt, with the caveat that, in the interest of speed, we left out automatic scaling and some of the improvements described in [30,3] (including the line-search filter, inertia correction, accelerating heuristics, and KKT error restoration). Our goal was to generate code for very light and fast



(a) Lasso ($K = 20$ features, N training examples)

(b) Soft-margin SVM classifier ($K = 20$ features, N training examples)

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

(d) Distance-based localization (N measurements)

(e) MPC (N horizon lengths)

(f) MPC-MHE (N horizon length)

Fig. 10 Solve times using `TensCalc`, Ipopt (through the CasADi interface), and CVXGEN.

optimization solvers that would take full advantage of the fixed sparsity pattern of the matrices involved in the computation of the Newton search direction.

Figure 10 compares the results of the solver generated by `TensCalc` with the Ipopt solver accessed through CasADi's MATLAB interface using the Opti stack. This CasADi help class can be used to generate calls to Ipopt using a problem description that is very close to that of `TensCalc` (see appendix). Especially for

large problems, Ipopt can achieve convergence with a smaller number of iterations. For example in th MPC example, Ipopt typically requires only 4 iterations, whereas `TensCalc` requires at least 6. However, all the processing done by `TensCalc` at code-generation time to optimize computations translates into very large computational savings, generally on the order of 10-100 times faster.

One crucial difference between the solvers generated by `TensCalc` and Ipopt is that the size of the executable code generated by `TensCalc` grows with the problem size, as we can see in Figure 5. With Ipopt, the data memory grows with the problem size, but not the size of the executable.

Three of the problems considered here fall in the class of convex quadratic optimizations with linear constraints, for which CVXGEN can generate specialized solvers that can also take advantage of the sparsity structure of the problem. CVXGEN is somewhat limited in problem size, but we included in Figure 10 the solve times obtained with code generated by CVXGEN, for the problem sizes that it can support. As with `TensCalc`, we can see the benefits of using code that has been specialized for the structure of the problem. We conjecture that the main reason why `TensCalc` achieves smaller solve times for most problems is the algorithm discussed in Section 6.1 to avoid redundant computations.

9 Conclusions and Future Work

We developed a toolbox that generates specialized C code to 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 associated with the specific optimization and minimizing the amount of computation needed for each iteration of the algorithm. The generation of C code, which becomes specialized for a specific optimization problem, takes advantage of a set of optimizations that result in very fast solve times.

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, including discarding additions of zero and multiplications by zero or one. We believe that improving the symbolic engine could improve performance for many problems. `TensCalc` uses symbolic differentiation performed over matrix/tensor-valued functions rather than automatic differentiation, which is typically carried out over scalar-valued operations. Significant work has been done to construct efficient algorithms to minimize computation in the automatic differentiation of sparse Jacobian/Hessian matrices and it remains to explore how these could be used to improve `TensCalc` [10, 13, 24, 29].

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. This provides significant opportunities to reduce memory usage that are not explored in the current version of TensCalc.

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 multiple cores.

Operations like matrix multiplication or LU/LDL factorizations of matrices with regular sparsity patterns involve the repeated execution of a large number of similar fragments of code that could take advantage of processors with single instruction, multiple data (SIMD) instructions. Complex code fragments could even be implemented in specialized field-programmable gate array (FPGA) fabrics. This is also a topic for future research.

Acknowledgments

The authors would like to thank Prof. John Hauser and Prof. Moritz Diehl for suggestions that improved this paper. We also acknowledge and thank the help provided by several users of this toolbox that greatly contributed to its improvement, including David Copp, Justin Pearson, Murat Erdal, Raphael Chinchilla, Richard Erwin, Robin Straesser, Sharad Shankar, and Steven Quintero.

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

```
44  % parameters
45  Tvariable X [N,K];
46  Tvariable y [N];
47  Tvariable lambda [];
48  % optimization variables
49  Tvariable beta [K];
50  Tvariable abs_beta [K];
51  % criteria
52  J = norm2(y-X*beta);
53  % constraints
54  constraints={ beta>=-abs_beta;
55                beta<=abs_beta;
56                sum(abs_beta,1)<=lambda; };
57  cmex2optimizeCS('classname','lasso_solver',...
58                  'objective',J,...
59                  'optimizationVariables',{beta,abs_beta},...
60                  'constraints',constraints,...
61                  'outputExpressions',{J,beta,abs_beta},...
62                  'parameters',{X,y,lambda});
```

For comparison, we include below the corresponding code using CasADi's Opti stack MATLAB interface.

```
63 opti=casadi.Opti();
64 % parameters
65 X=opti.parameter(N,K);
66 y=opti.parameter(N,1);
67 lambda=opti.parameter();
68 % optimization variables
69 beta=opti.variable(K,1);
70 abs_beta=opti.variable(K,1);
71 % criteria
72 opti.minimize(sum((y-X*beta).^2));
73 % constraints
74 opti.subject_to(beta>=-abs_beta);
75 opti.subject_to(beta<=abs_beta);
76 opti.subject_to(sum(abs_beta,1)<=lambda);
77 opti.solver('Ipopt');
```

Soft-margin SVM classifier

```
78 % parameters
79 Tvariable y [N];
80 Tvariable X [N,K];
81 Tvariable lambda [];
82 % optimization variables
83 Tvariable beta [K];
84 Tvariable b    [];
85 Tvariable zeta [N];
86 % criteria
87 J = sum(zeta,1)/N+lambda*norm2(beta);
88 % constraints
89 constraints={ y.*(X*beta+b)>= 1-zeta;
90               zeta>=0; };
```

Hougen-Watson model identification

```
91 % parameters (measured inputs & outputs)
92 Tvariable tilde_abc [N,3];
93 Tvariable tilde_y [N,1];
94 Tvariable lambda [];
95 Tvariable beta_bounds [5,2];
96 % optimization variables
97 %    (noiseless inputs/outputs & model parameters)
98 Tvariable abc [N,3];
99 Tvariable y [N,1];
100 Tvariable beta [5,1];
101 % criteria (log-likelihood)
102 J = norm2(tilde_y-y)+lambda*norm2(tilde_abc-abc);
103 % constraints
104 constraints={
105   (1+abc(:,1)*beta(2,1)+abc(:,2)*beta(3,1)..
106                         +abc(:,3)*beta(4,1)).*y...
107            ==abc(:,2)*beta(1,1)-abc(:,3)*beta(5,1);
108     beta>=beta_bounds(:,1);
109     beta<=beta_bounds(:,2); };
```

Distance-based localization

```
110 % parameters
111 Tvariable B [3,1,M];
112 Tvariable tilde_d [N,M];
113 Tvariable lambda [];
114 Tvariable pbox [3,2];
115 % optimization variables
116 Tvariable p [3,N];
117 % criteria
118 p1=reshape(p,[3,N,1]);
119 pB=p1(:,:,ones(1,M))-B(:,ones(1,N),:);
120                      % vectors from beacons to point
121 d=sqrt(tprod(pB,[-1,1,2],pB,[-1,1,2]));
122                      % distances from beacons to point
123 v=p(:,2:end)-p(:,1:end-1);              % velocity
124 a=v(:,2:end)-v(:,1:end-1);              % acceleration
125 J=norm2(tilde_d-d)+lambda*norm2(a);     % log-likelihood
126 % constraints
127 constraints={ p>=pbox(:,ones(1,N));
128               p<=pbox(:,2*ones(1,N)); };
```

MPC for linear quadratic problem with constraints

```
129 % parameters
130 Tvariable u_past   [n-1,1];   % [ u(k-n+1) ... u(k-1) ]
131 Tvariable y0       [n,1];     % [ y(k-n+1)   ... y(k) ]
132 Tvariable lambda_u [];
133 Tvariable alph     [n,1];
134 Tvariable beta     [n,1];
135 Tvariable umax     [];
136 % optimization variables
137 Tvariable u_future [T,1];     % [ u(k) ... u(k+T-1) ]
138 Tvariable y1       [T,1];     % [ y(k+1) ... y(k+T) ]
139 y=[y0;y1];                    % [ y(k-n+1)   ... y(k+T) ]
140 % criteria
141 J=norm2(y(n+1:end,1))...      % [ y(k+1)   ... y(k+T) ]
142   +lambda_u*norm2(u_future);  % [ u(k) ... u(k+T-1) ]
143 % constraints
144 u=[u_past;u_future];          % [ u(k-n+1) ... u(k+T-1) ]
145 yy=Tzeros(T,1);               % [ y(k+1) ... y(k+T) ]
146 for i=1:n
147     % += alph(i) * [ y(k+1-i) ... y(k+T-i) ]
148     %    beta(i)  * [ u(k+1-i) ... u(k+T-i) ]
149     yy=yy+y(n-i+1:end-i,1)*alph(i,1)+u(n-i+1:end-i+1,1)*beta(i,1);
150 end
151 % constraints
152 constraints={ -umax<=u_future; u_future<=umax;
153               y(n+1:end,1)==yy; }
154 
155 [classname,code]=cmex2optimizeCS(...
156           'classname','mpc_solver',...
157           'objective',J,...
158           'optimizationVariables',{u_future,y1},...
159           'constraints',constraints,...
160           'outputExpressions',{J,u_future,y1},...
161           'parameters',{lambda_u,umax,...
162                         alph,beta,...
163                         y0,u_past});
```

MPC-MHE for linear quadratic problem with constraints

```
164 % parameters
165 Tvariable tilde_y  [L,1];     % [ y(k-L+1) ... y(k) ]
166 Tvariable u_past   [L-1,1];   % [ u(k-L+1) ... u(k-1) ]
167 Tvariable lambda_u [];
168 Tvariable lambda_n [];
169 Tvariable lambda_d [];
170 Tvariable alph     [n,1];
171 Tvariable beta     [n,1];
172 Tvariable umax     [];
173 Tvariable dmax     [];
174 Tvariable nmax     [];
175 % optimization variables
176 Tvariable u_future [T,1];      % [ u(k) ... u(k+T-1) ]
177 Tvariable d        [T+L-n,1]; % [ d(k-L+n)   ... d(k+T-1) ]
178 Tvariable y0       [n,1];      % [ y(k-L+1)   ... y(k-L+n) ]
179 Tvariable y1       [T+L-n,1]; % [ y(k-L+n+1) ... y(k+T) ]
180 y=[y0;y1];                     % [ y(k-L+1)   ... y(k+T) ]
181 % criteria
182 J=norm2(y(L+1:end,1))+lambda_u*norm2(u_future)...
183   -lambda_n*norm2(y(1:L,1)-tilde_y)-lambda_d*norm2(d);
184 % constraints
185 u=[u_past;u_future];           % [ u(k-L+1) ... u(k+T-1) ]
186 yy=d;                          % [ y(k-L+n+1) ... y(k+T) ]
187 for i=1:n
188     % += alpha(i) * [ y(k-L+1+n-i) ... y(k+T-i) ]
189     %    beta(i)  * [ u(k-L+1+n-i) ... u(k+T-i) ]
190     yy=yy+y(n-i+1:end-i,1)*alpha(i,1)+u(n-i+1:end-i+1,1)*beta(i,1);
191 end
192 % minimizer constraints
193 P1constraints={ -umax<=u_future; u_future<=umax; };
194 % maximizer constraints
195 P2constraints={ -dmax<=d; d<=dmax;
196                 -nmax<=y(1:L,1)-tilde_y; y(1:L,1)-tilde_y<=nmax};
197 % common constraints
198 Lconstraints={ y(n+1:end,1)==yy; };
199
200 [classname,code]=cmex2equilibriumLatentCS(...
201       'classname','mpcmhe_solver',...
202       'P1objective',J,...
203       'P2objective',-J,...
204       'P1optimizationVariables',{u_future},...
205       'P2optimizationVariables',{d,y0},...
206       'latentVariables',{y1},...
207       'P1constraints',P1constraints,...
208       'P2constraints',P2constraints,...
209       'latentConstraints',Lconstraints,...
210       'outputExpressions',{J,u_future,d,y(1:L,1)-tilde_y,y0,y1},...
211       'parameters',{lambda_u,lambda_n,lambda_d,...
212                     umax,dmax,nmax,...
213                     alph,beta,...
214                     tilde_y,u_past});
```

References

1. Andersson, J.A., Gillis, J., Horn, G., Rawlings, J.B., Diehl, M.: CasADi: a software framework for nonlinear optimization and optimal control. Mathematical Programming Computation **11**(1), 1–36 (2019)
2. Appel, A.W.: Modern compiler implementation in C. Cambridge university press (2004)
3. Biegler, L.T., Zavala, V.M.: Large-scale nonlinear programming using IPOPT: An integrating framework for enterprise-wide dynamic optimization. Computers & Chemical Engineering **33**(3), 575–582 (2009)

4. Copp, D.A., Hespanha, J.P.: Simultaneous nonlinear model predictive control and state estimation. Automatica **77**, 143–154 (2017)
5. Davis, T.: SuiteSparse: A suite of sparse matrix software. `http://faculty.cse.tamu.edu/davis/suitesparse.html`
6. Davis, T.A.: Algorithm 832: UMFPACK v4. 3—an unsymmetric-pattern multifrontal method. ACM Transactions on Mathematical Software (TOMS) **30**(2), 196–199 (2004)
7. Davis, T.A., Gilbert, J.R., Larimore, S.I., Ng, E.G.: Algorithm 836: COLAMD, a column approximate minimum degree ordering algorithm. ACM Trans. Math. Software **30**(3), 377–380 (2004)
8. Davis, T.A., Gilbert, J.R., Larimore, S.I., Ng, E.G.: A column approximate minimum degree ordering algorithm. ACM Trans. Math. Software **30**(3), 353–376 (2004)
9. Farquhar, J.: tprod – arbitrary tensor product between n-d arrays (2007). MathWorks File Exchange
10. Gebremedhin, A.H., Tarafdar, A., Pothen, A., Walther, A.: Efficient computation of sparse Hessians using coloring and automatic differentiation. INFORMS Journal on Computing **21**(2), 209–223 (2009)
11. Gill, P.E., Saunders, M.A., Shinnerl, J.R.: On the stability of Cholesky factorization for symmetric quasidefinite systems. SIAM J. Matrix Anal. and Appl. **17**(1), 35–46 (1996)
12. Golub, G.H., Loan, C.F.V.: Matrix Computations, 2nd edn. Johns Hopkins Series in Mathematical Sciences. The Johns Hopkins University Press, Baltimore (1990)
13. Gower, R.M., Mello, M.P.: A new framework for the computation of Hessians. Optimization Methods and Software **27**(2), 251–273 (2012)
14. Grant, M., Boyd, S.: Graph implementations for nonsmooth convex programs. In: V. Blondel, S. Boyd, H. Kimura (eds.) Recent Advances in Learning and Control, *Lecture Notes in Control and Information Sciences*, vol. 371, pp. 95–110. Springer Berlin / Heidelberg (2008)
15. Grant, M., Boyd, S., Ye, Y.: CVX: Matlab Software for Disciplined Convex Programming. Stanford University, Palo Alto, CA (2008). Available at `http://cvxr.com/cvx/`
16. Gurobi Optimization: Gurobi optimizer reference manual (2017). URL `http://www.gurobi.com`
17. Gwinner, F.: Transitive reduction of a DAG. Mathworks® File Exchange (2011)
18. Lofberg, J.: YALMIP: A toolbox for modeling and optimization in MATLAB. In: Computer Aided Control Systems Design, 2004 IEEE International Symposium on, pp. 284–289. IEEE (2004)
19. Mattingley, J., Boyd, S.: CVXGEN: a code generator for embedded convex optimization. Optimization and Engineering pp. 1–27 (2012)
20. McShane, K.A., Monma, C.L., Rutcor, D.S.: An implementation of a primal-dual interior point method for linear programming. ORSA Journal on Computing **1**(2), 70 (1989)
21. Mehrotra, S.: On the implementation of a primal-dual interior point method. SIAM J. on Optimization **2**(4), 575–601 (1992)
22. Nesterov, Y.E., Todd, M.J.: Self-scaled barriers and interior-point methods for convex programming. Mathematics of Operations research **22**(1), 1–42 (1997)
23. Nesterov, Y.E., Todd, M.J.: Primal-dual interior-point methods for self-scaled cones. SIAM J. on Optimization **8**(2), 324–364 (1998)
24. Petra, C.G., Qiang, F., Lubin, M., Huchette, J.: On efficient hessian computation using the edge pushing algorithm in Julia. Optimization Methods and Software **33**(4-6), 1010–1029 (2018)
25. Sturm, J.F.: Using SeDuMi 1.02, a MATLAb toolbox for optimization over symmetric cones. Optimization methods and software **11**(1-4), 625–653 (1999)
26. Tütüncü, R.H., Toh, K.C., Todd, M.J.: Solving semidefinite-quadratic-linear programs using SDPT3. Mathematical Programming **95**, 189–217 (2003)
27. Vandenberghe, L.: The CVXOPT linear and quadratic cone program solvers. Tech. rep., Univ. California, Los Angeles (2010). URL `http://cvxopt.org/documentation`
28. Vanderbei, R.J.: Symmetric quasidefinite matrices. SIAM J. on Optimization **5**(1), 100–113 (1995)
29. Vigerske, S., Gleixner, A.: SCIP: Global optimization of mixed-integer nonlinear programs in a branch-and-cut framework. Optimization Methods and Software **33**(3), 563–593 (2018)
30. Wächter, A., Biegler, L.T.: On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. Mathematical programming **106**(1), 25–57 (2006)

Declarations

Funding: This work was partially funded by the National Science Foundation.

Conflicts of interest/Competing interests: The author declares no conflict of interests nor competing interests.

Availability of data and material: Not applicable.

Code availability: All code is available at `https://github.com/hespanha/tenscalc` under the GNU General Public License v3.0 and the documentation is available at `https://tenscalc.readthedocs.io/`

Authors' contributions: Single author, not applicable.